Analysis of BCNS and NewHope Key-Exchange Protocols

by

Seyedamirhossein Hesamian

A thesis submitted in
partial fulfillment of the
requirements for the degree of

Master of Science
in Computer Science

College of Engineering & Applied Science
University of Wisconsin–Milwaukee
Milwaukee, Wisconsin, USA
Spring-2017

This thesis is submitted to Department of Computer Science at College of Engineering & Applied Science of UW-Milwaukee in partial fulfilment of the requirements for the degree of Master of Science in Computer Science.

Contact Information

Author: Seyedamirhossein Hesamian
Email: hesamian@uwm.edu

Advisor: Dr. Guangwu Xu
Email: gxu4uwm@uwm.edu

Department of Computer Science
College of Engineering & Applied Science
University of Wisconsin–Milwaukee
College of Engineering & Applied Science

3200 North Cramer Street
Milwaukee, WI 53211

Abstract

Lattice-based cryptographic primitives are believed to offer resilience against attacks by quantum computers. Following increasing interest from both companies and government agencies in building quantum computers, a number of works have proposed instantiations of practical post-quantum key-exchange protocols based on hard problems in lattices, mainly based on the Ring Learning With Errors (R-LWE) problem.

In this work we present an analysis of Ring-LWE based key-exchange mechanisms and compare two implementations of Ring-LWE based key-exchange protocol: BCNS and NewHope. This is important as NewHope protocol implementation outperforms state-of-the art elliptic curve based Diffie-Hellman key-exchange X25519, thus showing that using quantum safe key-exchange is not only a viable option but also a faster one. Specifically, this thesis compares different reconciliation methods, parameter choices, noise sampling algorithms and performance.

Keywords: lattice based key-exchange, Ring-LWE, BCNS, NewHope

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Prof. Guangwu Xu, for the patient guidance, encouragement and invaluable advice he has provided throughout my time as his student. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. I can not forget the valuable conversation with and suggestion of Dr. Xu. This thesis would not have been possible without the inspiration and support of Dr. Xu and his guidance into the world of cryptography has been a valuable input for this thesis.

Contents

List of Figures

Notations

Notation	Definition
$\mathbb{N}$	Set of natural numbers
$\mathbb{Z}$	Set of integers
$\mathbb{Q}$	Set of rational numbers
$\mathbb{R}$	Set of real numbers
$\mathbb{C}$	Set of complex numbers
$\mathbb{Z}_p^*$	The following set: $\{x \in \mathbb{Z}_p \| \gcd(x, p) = 1\}$
$\mathbb{Z}/n\mathbb{Z}$	The quotient ring of integers modulo n
$\|\|x\|\|$	The l_2-norm, also denoted by $\|\|x\|\|_2$
$D_{\mathbb{Z}_n,\sigma,v}$	Probability function of the discrete spherical Gaussian distribution over $\mathbb{Z}^n$
λ_1	The length of shortest vector in lattice
R	$\mathbb{Z}[X]/(f)$ where $f \in \mathbb{Z}[X]$ of degree n
R_q	R/qR
$r \leftarrow \chi$	Element r is drawn according to the probability distribution χ

Abbreviations

Abbreviation	Meaning
SVP	Shortest vector problem
γ-SVP	γ-approximate SVP
GapSVP$_\gamma$	Decisional γ-approximate SVP
BDD$_\alpha$	α-bounded distance decoding problem
SIS	Short integer solution problem
LWE	Search learning with errors problem
DLWE	Decisional learning with errors problem
RLWE	Search ring learning with errors problem
R-DLWE	Decisional ring learning with errors problem
R-SIS	Ring short integer solution problem
KEM	Key encapsulation mechanism
PKE	Public key encryption scheme
IND-CPA	Indistinguishability under chosen-plaintext attacks
IND-CCA	Indistinguishability under chosen-ciphertext attacks
KEX	Key-exchange protocol

Chapter 1

Introduction

Cryptography is involved in many parts of our daily life, e.g. credit cards, Internet banking, electronic voting etc. The Oxford Dictionary of English proposes a simple, yet incomplete, definition of cryptology as "the study of codes, or the art of writing and solving them". The roots of this definition are to be found in History: the invention of cryptology comes from the problem of secret communications of diplomatic and military information. The basic idea is to apply a "complicated" transformation to the information to be protected. On one side of cryptology, users utilize secret codes, while on the other side, adversaries attempt to break through the secrecy of the messages to recover the hidden information. One of the oldest and simplest cryptologic technique, Caesar's Cipher, consists of replacing each letter of message by the letter three positions down the alphabet (looping back at the end).

Until the XIXth century, the study of secret codes lacked a precise and consistent theory, and designing or breaking such codes was considered as an art. The construction of "good codes" or deciphering relied on time, patience and ingenuity. With the introduction of mathematical formalism, the study of secret codes became a science that is commonly known as cryptology. This science contains two aspects: cryptography and cryptanalysis. Cryptography aims at designing new methods to ensure the secrecy of communications, and cryptanalysis aims at discovering flaws in these methods. And even though it was typical back then to keep these methods secret to make cryptanalysis more complicated, such a secrecy "by obscurity" is recognized to be delusive.

In this age of digital information and telecommunications, cryptography is now far from being restricted to the military and diplomatic fields. It has became a cornerstone in our daily life. Cryptography is present in our cellphones, our banking cards, our biometric passports, our Internet browsers, and many (often unsuspected) other products, which all require to guarantee security properties on their communications and on their data. Moreover, beyond the confidentiality of the secret information, one should also ensure that these contacts are secure against eavesdropping or injection of illegitimate messages. Thus, the scope of cryptography now includes among other things data integrity that is the fact that the data has not been modified, and data authenticity that is the fact that the sender is legitimate. Therefore, the cryptographer aims at designing systems that ensure these security properties, while the cryptanalyst looks at possible flaws that would reveal that these properties are actually not verified.

In their groundbreaking paper *New directions in Cryptography* [11] published in 1976, W. Diffie and M. Hellman introduced the concept of public-key cryptography and bridged cryptography to complexity theory. Until then, all the cryptographic systems were relying on a common secret shared between the sender and the receiver, or were using a symmetric secret key (symmetric because it was the same for both parties). Typical example of a symmetric encryption scheme is a block cipher. Such a cryptosystem is a pair of families $\{E_k\}_{k \in K}$ and $\{D_k\}_{k \in K}$ of algorithms representing invertible transformations over blocks of fixed length (e.g. 128 bits), inverse of each other, indexed by a symmetric key $k \in K$. When the sender conventionally named Alice, who is sharing a common secret key k with the receiver also conventionally named Bob, wants to confidentially send a message m to Bob, she can send the ciphertext $c = E_k(m)$ from which Bob can recover m by decrypting c, $m = D_k(c)$. Block ciphers remain fundamental and very useful ingredient of today's cryptography; they are extensively used in nearly all systems that are using cryptography.

All symmetric key cryptography (also called secret key cryptography) assumes that the two parties exchanging secret messages share a common secret key. Unfortunately, the secure distribution of such a key is a major issue. The methods of exchanging a secret key through a possibly eavesdropped conversation without sharing any secret beforehand is the fundamental of key-exchange protocols. In details, the key-exchange problem is how to exchange whatever keys or other information are needed so that no one else can obtain a copy. Historically, this required trusted couriers, diplomatic bags, or some other secure channel. So if the attacker is able to passively capture data and later gets an access to the private key, then the attacker could decode all previously captured data. In [11], W. Diffie and M. Hellman proposed an algorithm which revolutionized the concept of key-exchange, in which eliminates the need for a secure key distribution channel by constructing a procedure that enables two parties to derive a shared key over unsecured channel without actually transmitting the shared key itself. Indeed, sending the key in advance over a secure channel would be unrealistic for today's applications. The key-exchange mechanism is an efficient solution to the problem of creating a common secret between two participants, which it can subsequently be used to encrypt all the communications thanks to the symmetric cipher. Moreover, it is one-round which implies that each participant is allowed to talk once and broadcast some data to the other participant. The two main approaches for Diffie-Helman protocol uses either finite field or elliptic curve but they both are not safe under quantum computers due to Shor's quantum algorithm [34].

In recent years, lattice-based cryptography has been recognized for its many attractive properties, such as strong provable security guarantees and apparent resistance to quantum attacks, flexibility for realizing powerful tools like fully homomorphic encryption and high asymptotic efficiency. Indeed, several works have demonstrated that for basic tasks like encryption and authentication, lattice-based primitives can have performance competitive with or even surpassing those based on classical mechanisms like RSA or Diffie-Hellman.

However, there still has been relatively little work done on developing lattice based key-exchange for deployment in real-world cryptosystems and protocols.

Many lattice based cryptography algorithm are based on the Learning With Errors problem (LWE) which is a variant of lattice problems that is as hard to solve as several worst-case lattice problems. The basics of Ring-LWE (ring variant of LWE) based key-exchange protocol was introduced for the first time by Regev in [30] and later the key-agreement algorithm improved by Ding in [12]. Ding's reconciliation or key-agreement method was original as it introduced the concept of sending extra information to improve the success probability of key-agreement. Thereafter, Peikert in [29] addressed the shortcomings of Ding's method and provided a relatively simpler reconciliation algorithm.

BCNS protocol [7], which is sponsored by Microsoft Research, does not introduce a new key-agreement algorithm but it provides parameters. It uses Peikert's key-agreement algorithm which is a derivative of Ding's method and also it is the first optimized C implementation of Ring-LWE based key-exchange. Most importantly, this protocol provided a drop-in replacement (or patch) for the Transport Layer Security (TLS) protocol. In further revisions of Open-SSL, the BCNS protocol is included by default in Open-SSL. This enables users to seamlessly switch to quantum-safe protocol. BCNS protocol goes further and proves that cost of switching from non-quantum-safe key-exchange to quantum-safe is not too high. Thus, post-quantum key-exchange can already be considered practical.

NewHope protocol [4], which is implemented in collaboration with Google, does introduce a new key-agreement algorithm which is a derivative of Peikert's method and also provides improved parameters. The new key-agreement algorithm is a generalization of Peikert's algorithm in 4-dimension instead of 1-dimension that Peikert suggested. NewHope protocol provides a more practical approach in addition to a more optimized C reference code. NewHope protocol later was added to Google's fork of Open-SSL also known as Boring-SSL and adjusted to use it's built-in functions and subsequently it is currently being used by Google and included in Google Chrome web browser. Further, the AVX2 assembly language implementation of this protocol outperforms state-of-the art elliptic curve cryptography based Diffie-Hellman key-exchange, X25519, thus showing that using quantum safe key-exchange is not only a viable option but also a faster one.

In this work we do a detailed analysis of Ring-LWE based key-exchange reconciliation (or key agreement) methods, protocols, parameter choices, noise sampling algorithms, performance and compare two implementation of Ring-LWE based key-exchange protocols: BCNS and NewHope. Throughout this thesis, we study the relation between lattices and lattice based key-exchange and how lattice based cryptography evolved overtime starting from Ajtai's result up-to highly optimized NewHope key-exchange protocol.

This paper is organized as follows: chapter 2 discusses preliminary subjects needed to understand importance of lattices and how Ring-LWE based key-exchange tries to replace existing key-exchange protocols. More specifically, this chapter discusses Diffie-Hellman, Shor's algorithms and how it breaks Diffie-Hellman by solving discrete logarithm problem efficiently. Thereafter, chapter 3 overviews definition of lattice, lattice properties, lattice reduction and most importantly lattice problems. Chapter 4 is the main part of this thesis that overviews basics of Ring-LWE key-exchange, the need for reconciliation and analysis of different key-agreement (or reconciliation) methods and describes BCNS and NewHope protocols in details. Chapter 5 describes implementation specifics of BCNS and NewHope protocols. For example, error sampling algorithm and performance analysis. Chapter 6 discusses future works and conclusion or the main take away from this thesis.

In appendix chapter, we overview early lattice based public-key cryptosystem that was broken using lattice reduction. Further, it discusses basic implementation of all Ring-LWE based key-exchange protocols using SageMath (extension of Python programming language).

Chapter 2

Preliminaries

In this chapter we overview the need for key-exchange, basics of existing key-exchange protocols, how quantum computers will affect existing protocols and the need to replace them before introduction of quantum computers. Lastly, we overview basics of quotient polynomial ring as it is a prerequisite to understand ideal lattices that will be discussed in the next chapter.

2.1 Key-exchange problem overview

For symmetric key cryptography to work for online communications, the secret key must be securely shared with authorized communicating parties and protected from discovery as well as use by unauthorized parties. Key-exchange protocol does nothing about authentication and without authentication, impersonation is feasible, and that includes simultaneous double impersonation, better known as Man-in-the-Middle attack. In Transport Layer Security (TLS) protocol, public key cryptography is used in Cryptographic Signatures to provide authenticity, in conjunction with key-exchange mechanism to provide forward secrecy by signing ephemeral key using server's private key. In details, in DHE_RSA cipher suite, server dynamically generates a Diffie-Hellman public key and sends it to the client; the server also signs what it sends. Then client responds with his/her Diffie-Hellman public key encrypted using server's public key and then connection from this point on is encrypted using the Diffie-Hellman calculated shared key.

2.1.1 Importance of key-exchange

Key-exchange is any method in cryptography by which cryptographic keys are exchanged between two parties, allowing use of a cryptographic algorithm. To clarify, key-exchange is a way of generating a shared secret between two people in such a way that the secret cannot be seen by observing the communication. That is an important distinction: two parties are not sharing information during the key-exchange, they create a shared key together. This is particularly useful because we can use this technique to create a symmetric encryption key with someone, and then start encrypting traffic with that key which is also known as *session key.* And even if the traffic is recorded and later analyzed, there is absolutely no way to figure out what the key was, even though the exchanges that created it may have been visible. This is where perfect forward secrecy comes from. Nobody analyzing the traffic at a later date can break in because the key was never saved, never transmitted, and never made visible anywhere.

Definition 1. Session key is a single-use symmetric key used for encrypting all messages in one communication session.

Like all cryptographic keys, session keys must be chosen so that they cannot be predicted by an attacker, usually requiring them to be chosen randomly. Failure to choose session keys (or any key) properly is a major (and too common in actual practice) design flaw in any cryptosystem.

Definition 2. Forward secrecy (or perfect forward secrecy, PFS) is a property in which compromise of long-term keys does not compromise past session keys. Forward secrecy protects past sessions against future compromises of secret keys or passwords. If forward secrecy is used, encrypted communications and sessions recorded in the past cannot be retrieved and decrypted.

To achieve forward secrecy, if one uses long term secret keys for authentication only and uses short term ephemeral keys for encryption then compromise of long term key does not compromise confidentiality of past massages. To clarify, when server's private key gets leaked then if we simply encrypted the session key using the server's public key, all past communication with that server can be decrypted. This is an unintended consequence. But if an ephemeral Diffie-Hellman key-exchange was used, a private key leak would not compromise past communications, since the keys used for the key exchange are long gone, and the leaked long term key was only used for authentication and not for confidentiality.

It is always an option to use public-key cryptography (e.g. RSA) as a key-exchange protocol (i.e. client encrypts a random key using server's public-key and then server decrypts it using it's private-key). However, public-key algorithms are generally far more complex than key-exchange protocols. There are two primary reasons to use session keys. First, several cryptanalytic attacks become easier as more material encrypted with a specific key is available. By limiting the amount of data processed using a particular key, those attacks are made more difficult. Second, asymmetric encryption is too slow for key-exchange purposes, and all symmetric encryption algorithms require that the key is securely distributed. For example, advantage of using Diffie-Hellman over RSA for generating ephemeral keys is producing a new Diffie-Hellman key pair can be extremely fast; in case of Diffie-Hellman based on elliptic curve, provided finite cyclic group into which shared key is computed is reused or reusing shared polynomial in Ring-LWE variant, both do not entail extra risks.

2.2 Diffie-Hellman problem and protocol

The Diffie–Hellman problem (DHP) is a mathematical problem first proposed by W. Diffie and M. Hellman in the context of cryptography. The motivation for this problem is that many security systems use mathematical operations that are fast to compute, but hard to reverse. The following is the definition of discrete logarithm problem which is closely related to Diffie-Hellman problem.

Definition 3. Consider a cyclic group G with generator g, given an element $h \in G$, discrete logarithm problem is to find the smallest positive integer x such that $h = g^x$

Formally, g is a generator of a cyclic group (typically the multiplicative group of a finite field or an elliptic curve group) and x and y are randomly chosen integers. Note that when g is a generator of the multiplicative group of integers modulo p, then g is called primitive root. Primitive root is an integer whose powers modulo p generates uniformly all integer in range $[1, \phi(p) = p - 1]$ inclusive.

The following is an example of discrete logarithm problem if group is a multiplicative group of a finite field. Given $\mathbb{Z}_5^*$ and generator 2, then the discrete logarithm of 1 is 4 because $2^4 \equiv 1 \bmod 5$. In general, Fermat's theorem tells us that if $g^x \equiv h \bmod p$, then $x + (p-1)k$ is also a solution for any integer k. Therefore, the discrete logarithm can be regarded as a number modulo $p - 1$.

The following is the formal definition of Decisional Diffie–Hellman.

Definition 4. Consider a (multiplicative) cyclic group G of order q, and with generator g. Given g^a and g^b for uniformly and independently chosen $a, b \in \mathbb{Z}_q$, Decisional Diffie–Hellman (DDH) is to distinguish between g^{ab} and a random element in G.

This intuitive notion is that the following two probability distributions are computationally indistinguishable:

- (g^a, g^b, g^{ab}), where a and b are randomly and independently chosen from $\mathbb{Z}_q$
- (g^a, g^b, g^c), where a, b, c are randomly and independently chosen from $\mathbb{Z}_q$

Triples of the first kind are often called DDH triples or DDH tuples.

The following is the formal definition of Computational Diffie–Hellman.

Definition 5. Consider a cyclic group G of order q. The Computational Diffie–Hellman (CDH) assumption states that, given (g, g^a, g^b), for a randomly chosen generator g and random $a, b \in \{0, \ldots, q-1\}$, it is computationally intractable to compute the value g^{ab}.

The CDH assumption is related to the discrete logarithm assumption, which holds that computing the discrete logarithm of a value given generator g as a base of logarithm is hard. If taking discrete logarithms in G were easy, then the CDH assumption would be false: given (g, g^a, g^b), one could efficiently compute g^{ab} in the following way:

- Compute a by taking the discrete logarithm of g^a to base g
- Compute g^{ab} by exponentiation: $g^{ab} = (g^b)^a$

The DDH and CDH assumptions are related to each other. If computing g^{ab} from (g, g^a, g^b) were easy, it would also be easy to detect DDH tuples. It is believed that DDH is a stronger assumption than CDH, because there are groups for which detecting DDH tuples is easy [15], but solving the CDH problem is believed to be hard. The most efficient means known to solve the DHP is to solve the discrete logarithm problem (DLP) and DHP is considered difficult for groups whose order is large enough.

2.2.1 Diffie-Hellman protocol

The Diffie-Hellman protocol is a method for two parties to generate a shared private key with which they can then exchange information across an insecure channel. The following is a description of the protocol when we use the multiplicative group of integers modulo p, but it can be generalized to finite cyclic groups (e.g. elliptic curve group). Let the users be named Alice and Bob. First, they agree on two numbers g and prime p, where p is large (typically at least 1024 bits) and g is a primitive root modulo p (in practice, it is a good idea to choose p such that $\frac{p-1}{2}$ is also prime). The numbers g and p need not be kept secret from others. Then Alice chooses a large random number a as her private key and Bob similarly chooses a large number b. Note that only a, b are kept secret. All the other values p, g, $g^a \bmod p$, and $g^b \bmod p$ are sent in the clear-text. Alice then computes $A = g^a \bmod p$, which she sends to Bob, and Bob computes $B = g^b \bmod p$, which he sends to Alice.

Then both Alice and Bob compute their shared key, which Alice computes as $K = B^a \bmod p = (g^b)^a \bmod p$ and Bob computes as $K = A^b \bmod p = (g^a)^b \bmod p$. More specifically,

$$(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p \tag{2.1}$$

Alice and Bob can now use their shared key K to exchange information without worrying about other users obtaining this information.

In practice, we choose prime p such that $p = 2k + 1$ where k is also a prime, this known as safe prime or Sophie-Germain prime. It is relatively fast to find such p. Then any number in $\mathbb{Z}_p^* = \{x \in \mathbb{Z}_p | \gcd(x, p) = 1\}$ will have an order m such that $m \mid \phi(p)$ hence is one of $1, 2, k, 2k$. We pick a random number x and check if $x, x^2, x^k \not\equiv 1 \bmod p$. If so, then x is a primitive root of p, otherwise, we start over. If we pick random numbers, we will soon find one. The number of primitive roots is $\phi(\phi(p))$, so the probability of hitting a primitive root is about $\frac{1}{2}$ in each try. Since number of primitive roots modulo p equals to $\phi(\phi(p)) = \phi(p-1) = \phi(2k) = \phi(2)\phi(k) = k-1$, and potential primitive root x is in range $[2, p-2]$; hence, success probability would be $\frac{k-1}{p-3} = \frac{k-1}{2k-2} = \frac{1}{2}$.

In order for a potential eavesdropper (Eve) to attack, she would first need to obtain $K = g^{(ab)} \bmod p$ knowing only g, p, $A = g^a \bmod p$ and $B = g^b \bmod p$. This can be done by computing a from $A = g^a \bmod p$ and b from $B = g^b \bmod p$. This is a discrete logarithm problem which is computationally infeasible for large p. Computing the discrete logarithm of a number modulo p takes roughly the same amount of time as factoring the product of two primes the same size as p, which is what the security of the RSA cryptosystem relies on.

Alice		**Bob**
$a \overset{\$}{\leftarrow} \mathbb{Z}_p$		$b \overset{\$}{\leftarrow} \mathbb{Z}_p$
	$\xrightarrow{g^a \bmod p}$	
	$\xleftarrow{g^b \bmod p}$	
$K_{AB} = (g^b)^a \equiv g^{ab} \bmod p$		$K_{AB} = (g^a)^b \equiv g^{ab} \bmod p$

Figure 2.1: Diagram of Diffie-Hellman key-exchange protocol
Note that g is a primitive root mod$p$ and $\overset{\$}{\leftarrow}$ means chosen uniformly random

2.2.2 Discrete logarithm in polynomial time using quantum computing

Shor's algorithm [34], named after mathematician Peter Shor, is a quantum algorithm (an algorithm that runs on a quantum computer) for integer factorization formulated in 1994. Informally it solves the following problem: given an integer N, find its prime factors. On a quantum computer, to factor an integer N, Shor's algorithm runs in polynomial time $\approx \mathrm{O}(n^2)$ where n is number of bits of input. This is substantially faster than the most efficient known classical factoring algorithm, the general number field sieve, that factors large integers (e.g. more than 140 digits) which works in sub-exponential time $e^{(1.923+\mathrm{O}(1))(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}}$. If a quantum computer with a sufficient number of qubits (quantum bits) could operate, Shor's algorithm could be used to break public-key cryptography schemes such as RSA scheme. Security of RSA is based on the assumption that factoring large numbers is computationally intractable. To break RSA, it is essentially factoring the modulus N to primes p, q.

To break Diffie-Hellman, we have to get g^{xy} from g^x, g^y and g. The best known way to do this would be to get x from g^x or y from g^y, the discrete logarithm problem. Shor's algorithm was initially designed to factor integers but later it was shown that it can be modified to solve discrete logarithm problem in polynomial time [6]. In details, both factorization and discrete logarithm are special cases of the hidden subgroup problem over an abelian group (elliptic curve cryptography also falls in the same category). We *do* have an efficient quantum algorithms with polynomial time complexity for solving the hidden subgroup problem over any abelian group (e.g. Shor's algorithm). As for examples of non-abelian hidden subgroup problems such as graph isomorphism and certain lattice problems (e.g. SVP), we do not know how to solve these efficiently on a quantum computer.

To summarize, there is no known classical algorithm that can solve discrete logarithm problem in polynomial time. However, Shor's algorithm shows that solving discrete logarithm problem is efficient on an ideal quantum computer, so it is *feasible* to defeat RSA and Diffie-Hellman protocols by constructing a large quantum computer.

2.2.3 Quantum safe replacement Diffie-Hellman like key-exchange

Lattice-based cryptography is the generic term for asymmetric cryptographic primitives based on lattices. While lattice-based cryptography has been studied for several decades, there has been renewed interest in lattice-based cryptography as prospects for a real quantum computer improve. Unlike more widely used and known public key cryptography such as the RSA or Diffie-Hellman cryptosystems which are attacked by a quantum computer, some lattice-based cryptosystems appear to be resistant to attack by both classical and quantum computers. Further the Learning With Errors (LWE) variants of lattice-based cryptography comes with security proofs which demonstrate that breaking the cryptography is equivalent to solving known hard problems in lattices. Later, Ring-LWE variant of LWE was created to address inefficiency of LWE based cryptosystems.

The Ring-LWE key-exchange is designed to be a *quantum safe* replacement for the widely used Diffie-Hellman key-exchanges as well as it's elliptic curve variant that are used to secure the establishment of secret keys over untrusted communications channels. Like Diffie-Hellman and it's elliptic curve variant, the Ring-LWE key-exchange provides a cryptographic property called forward secrecy; the aim of which is to ensure that there are no long term secret keys that can be compromised would enable bulk decryption.

Ring-LWE key-exchange is similar in nature to Diffie-Hellman (i.e. having a public and private components to derive a shared key) therefore existing authenticated, multiparty or authenticated-multiparty key-exchange protocols can also be used as an abstract layer. The underlying mathematics behind Diffie-Hellman and Ring-LWE have no connection with each other, only intention is to create an alternative Diffie-Hellman like key-exchange protocol.

2.3 Quotient ring

In mathematics, especially in the field of abstract algebra, a polynomial ring or polynomial algebra is a ring formed from the set of polynomials in one or more indeterminate variable (traditionally also called variables) with coefficients in another ring, often a field.

Definition 6. The polynomial ring, $K[x]$, in x over a field K is defined as the set of expressions, called polynomials in x, of the form $p(x) = p_0 + p_1x + p_2x^2 + \cdots + p_{m-1}x^{m-1} + p_mx^m$, where $p_0, p_1, \ldots, p_m$, the coefficients of p, are elements of K. The symbol x is called an indeterminate or variable and degree of a polynomial p, written as $\deg(p)$, is the largest k such that the coefficient of x^k is not zero.

The polynomial ring in x over K is equipped with an addition, a multiplication and a scalar multiplication that make it a commutative algebra. These operations are defined according to the ordinary rules for manipulating algebraic expressions. Specifically, if $p = p_0 + p_1x + p_2x^2 + \cdots + p_mx^m$, and $q = q_0 + q_1x + q_2x^2 + \cdots + q_nx^n$, then $p + q = r_0 + r_1x + r_2x^2 + \cdots + r_kx^k$, and $pq = s_0 + s_1x + s_2x^2 + \cdots + s_lx^l$, where $k = \max(m, n)$, $l = m + n$, $r_i = p_i + q_i$ and $s_i = p_0q_i + p_1q_{i-1} + \cdots + p_iq_0$.

Definition 7. Let R be a ring. An ideal I is a nonempty subset of R such that (i) if $a, b \in I$, then $a + b \in I$, and (ii) if $a \in I$ and $r \in R$, then $ar \in I$ and $ra \in I$.

This definition says that an ideal is a subset of R closed under addition that satisfies a strengthened form of closure under multiplication. Not only is the product of two elements of I also in I, but the product of an element of I and any element of r is an element of I. Therefore we can define addition in the set $R/I = \{a+I | a \in R\}$ by $(a+I)+(b+I) = (a+b)+I$ and multiplication by $(a + I)(b + I) = ab + I$. Hence, R/I is called the *quotient ring* or factor ring of R by I. Let $L = K[x]/(p(x))$ be the quotient ring of the polynomial ring $K[x]$ by the ideal generated by p (i.e. p is a univariate polynomial over a field K), then L is a field if and only if p is irreducible polynomial over K.

Chapter 3

Lattice cryptography

In 1996, Ajtai discovered that there are mathematical problems in the area of lattices that have some desirable properties with respect to cryptography. Since then, lattices have been used to construct several cryptosystems and other cryptographic applications. In this chapter, lattices are introduced and their connection with cryptography is examined.

3.1 Lattice, definition and properties

Lattice is a set of points in n-dimensional space with a periodic structure. More formally, given n-linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$, the lattice generated by them is the set of vectors:

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

Linear independence means that no vector can be written as linear combination of the other vectors. The set of vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ are known as a basis of the lattice. Below are fundamental properties of lattices:

Rank of lattice: The rank of a lattice is defined as the number of linearly independent vectors in any basis for that lattice. Full-rank lattice is defined as a lattice where the number of linearly independent vectors in any basis for this lattice is exactly equal to the number of dimensions in which the lattice is embedded. In such an instance, it is clear that any basis for such a lattice can be described by a set of n vectors, each of n dimensions. We can thus describe the basis as a square integer matrix.

Determinant of lattice: Lattice determinants measures properties about the lattice. Specifically, $\det(\mathcal{L}_{\mathbf{B}})$ is the n-dimensional volume of the fundamental parallelepiped defined by the lattice basis $\mathbf{B}$. Since we will only be operating with full-rank lattices, we can simplify the definition of this lattice determinant as being the absolute value of the determinant of some basis of the lattice.

$$\det(\mathcal{L}_{\mathbf{B}}) = |\det(\mathbf{B})|, \mathbf{B} \in \mathbb{Z}^{n \times n}$$

Definition 8. The unimodular matrix is defined as an integer matrix, whose inverse is also integral. This implies the following properties: $\mathbf{U}$ must be integral matrix, must be square matrix and $\det(\mathbf{U})$ must be exactly $+1$ or -1.

Any multiplication of a lattice basis with a unimodular matrix will produce a new basis that would generate the same lattice. This is due to the determinant of a unimodular matrix being $+1$ or -1. In fact, lattice equality is only achieved if there exists such a unimodular transform between bases. The determinant of a lattice is inverse proportional to its density: the smaller determinant, the denser the lattice would be.

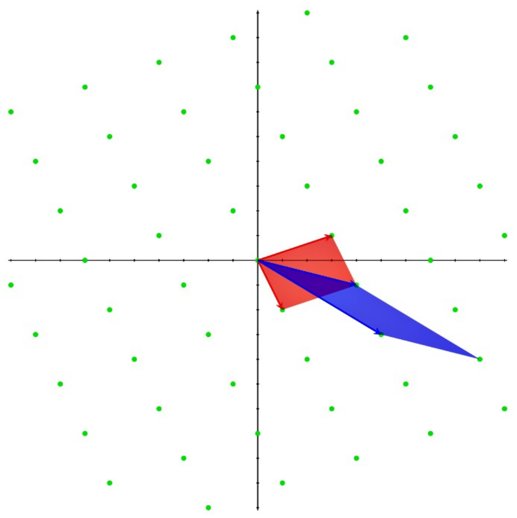


Figure 3.1: Two fundamental parallelepiped of the same lattice
The determinant of lattice is equal to the volume (i.e. area in 2-dimensions) of fundamental parallelepiped.

Hermite normal form: Various authors may prefer to talk about Hermite Normal Form in either row-style or column-style. They are essentially the same up to transposition.

(Row-style) Hermite Normal Form: m by n matrix $\mathbf{A}$ with integer entries has a (row) Hermite normal form (HNF) $\mathbf{H}$ if there is a square unimodular matrix $\mathbf{U}$ where $\mathbf{H} = \mathbf{U} \times \mathbf{A}$ and $\mathbf{H}$ has the following restrictions:

1. $\mathbf{H}_{ij} = 0$ for $j > i$,
2. $\mathbf{H}_{ii} > 0$ for all i, and
3. $\mathbf{H}_{ij} \leq 0$ and $|\mathbf{H}_{ij}| < \mathbf{H}_{ii}$ for $j < i$

Note that the row-style definition has a unimodular matrix $\mathbf{U}$ multiplying $\mathbf{A}$ on the left (meaning $\mathbf{U}$ is acting on the rows of $\mathbf{A}$), while the column-style definition has the unimodular matrix action on the columns of $\mathbf{A}$. The two definitions of Hermite normal forms are simply transposes of each other.

For every matrix, there exist a Hermite normal form and it is unique. In details, for every m by n matrix $\mathbf{A}$ with integer entries has a unique m by n matrix $\mathbf{H}$ (HNF), such that $\mathbf{H} = \mathbf{U} \times \mathbf{A}$ for some square unimodular matrix $\mathbf{U}$.

Typical lattice in $\mathbb{R}^n$ has the form $\mathcal{L} = \{\sum_{i=1}^{n} \alpha_i \mathbf{a}_i \mid \alpha_i \in \mathbb{Z}\}$ where the $\mathbf{a}_i$ are in $\mathbb{R}^n$. If the columns of a matrix $\mathbf{A}$ are the $\mathbf{a}_i$, the lattice can be associated with the columns of a matrix, and $\mathbf{A}$ is said to be a basis of $\mathcal{L}$. Because the Hermite normal form is unique, it can be used to answer many questions about two lattice descriptions. For what follows, $\mathcal{L}_{\mathbf{A}}$ denotes the lattice generated by the columns of $\mathbf{A}$. Because the basis is in the columns of the matrix $\mathbf{A}$, the column-style Hermite normal form must be used. Given two basis for a lattice, $\mathbf{A}$ and $\mathbf{A}'$, the equivalence problem is to decide if $\mathcal{L}_{\mathbf{A}} = \mathcal{L}_{\mathbf{A}'}$. This can be done by checking if the column-style Hermite normal form of $\mathbf{A}$ and $\mathbf{A}'$ are the same.

Euclidean norm: Many problems in lattice theory involve distance minimization. The most intuitive way to measure distance in a multi-dimensional space is by using the Euclidean norm (l_2). This norm comes from Pythagoras' theorem, stating that the distance between two points is the square root of the sum of the axial distances squared. This can be extended to an arbitrary, finite-dimensioned vector space by squaring each of the axial dimensions and taking the square root of the sum. Let w be a vector of $\mathbb{R}^n$. The Euclidean norm is the function $||.||_2$ defined by: $||w||_2 = \sqrt{\sum_{i=1}^{n} |w_i|^2}$

Successive minima: One basic parameter of a lattice is the length of the shortest nonzero vector in the lattice (we have to ask for a nonzero vector since the zero vector is always contained in a lattice and its norm is zero). This parameter is denoted by λ_1. An equivalent way to define λ_1 is the following: it is the smallest r such that the lattice points inside a ball of radius r span a space of dimension 1. This definition leads to the generalization of λ_1, known as successive minima or λ_i; that is the smallest r such that lattice points inside a ball of radius r span a space of dimension i.

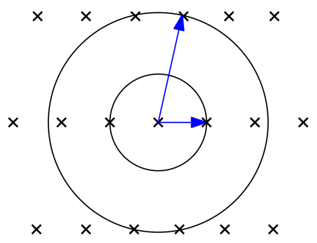


Figure 3.2: Visualization of first and second lattice minima
$\lambda_1(\mathcal{L}) = 1, \lambda_2(\mathcal{L}) = 2.3$

3.1.1 Examples of lattices

1. Below shows the lattice in 2 dimensions generated by the vectors $(1, 0)^t$ and $(0, 1)^t$. This lattice is the set of all points in $\mathbb{R}^2$ with integer coordinates. This can be generalized to n dimensions, where the lattice $\mathbb{Z}^n$ is called the *integer lattice*.

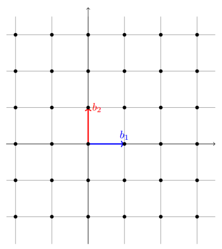


Figure 3.3: Lattice $\mathbb{Z}^n$ with basis vectors $(0, 1)^t$ and $(1, 0)^t$

2. Below shows a different basis for the same lattice, namely the basis consisting of the vectors $(1, 2)^t$ and $(2, 3)^t$.

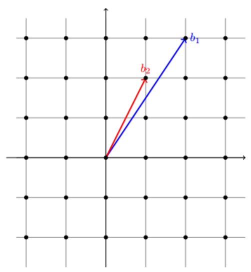


Figure 3.4: Lattice $\mathbb{Z}^2$ with a different basis consisting of vectors $(1, 2)^t$ and $(2, 3)^t$
In fact, any lattice has infinitely many bases

3. Below is a different lattice in 2 dimensions, generated by the basis vectors $(2,0)^t$ and $(1,1)^t$. Note that this is a sub-lattice of $\mathbb{Z}^2$, namely a subset of $\mathbb{Z}^2$ which is also a lattice.

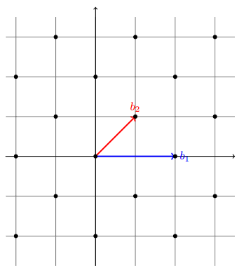


Figure 3.5: A full-rank lattice generated by the basis vectors $(1,1)^t$ and $(2,0)^t$
This is a sub-lattice of $\mathbb{Z}^2$

4. All the examples we saw so far are full-rank lattices. Below shows a lattice in 2 dimensions generated by the vector $(1,1)^t$, this lattice has a rank of 1. The set of points generated by 1 and $\sqrt{2}$ in one dimension is not a lattice. First, this example does not conform to definition of lattice, since 1 and $\sqrt{2}$ are linearly dependent over $\mathbb{R}$. Secondly, any n-dimensional lattice is a discrete subset of $\mathbb{Z}^n$. However, the set generated by 1 and $\sqrt{2}$ is not a discrete subset of $\mathbb{Z}$ since one can generate arbitrarily small numbers as linear combinations of 1 and $\sqrt{2}$.

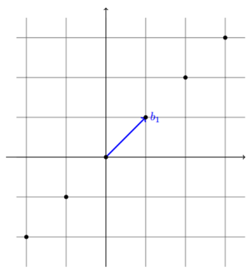


Figure 3.6: A non full-rank lattice with basis vector $(1,1)^t$
Notice basis of lattice is not a square matrix (or full-rank)

If a basis $\mathbf{B}$ can be transformed by a multiplication with a transformation matrix $\mathbf{U}$ such that the new basis $\mathbf{B}'$ yields the same lattice as the original basis $\mathbf{B}$ (i.e. $\mathbf{B}' = \mathbf{U} \times \mathbf{B}, \mathcal{L}_{\mathbf{B}} = \mathcal{L}_{\mathbf{B}'}$), we refer to this transformation $\mathbf{U}$ as a unimodular transformation. As such, any basis of a lattice can be transformed to any other basis for the same lattice through a multiplication with a single unimodular transformation matrix.

3.1.2 Ideal lattices

Research in lattice-based cryptography started with the publication of the public key encryption scheme by Ajtai and Dwork [3], followed by schemes based on cyclic lattices, e.g. Micciancio introduced [26]. Later in [26] ideal lattices, a generalization of cyclic lattices, were introduced.

Lattices via polynomial rings: Lattices cannot only be defined over $\mathbb{R}^n$ and $\mathbb{Z}^n$, but can also be defined via the ring $R = \mathbb{Z}_q[x]/(f(x))$ where R contains all polynomials in x with integer coefficients modulo prime q and polynomial $f(x)$. Note that if $f(x)$ is a monic polynomial, i.e. a polynomial with leading coefficient equal to 1, then $R = \mathbb{Z}_q[x]/(f(x))$ contains polynomials of degree at most $\deg(f(x)) - 1$. We will focus on the case that $f(x)$ is a monic polynomial of degree n.

There is a map between elements from $R = \mathbb{Z}[x]/(f(x))$, i.e. polynomials of bounded degree, and elements of a lattice $\mathcal{L}$, i.e. vectors, which is given as follows:

$$\phi : a_1 + a_2x^1 + a_3x^2 + \cdots + a_nx^{n-1} \mapsto (a_1, a_2, a_3 \ldots, a_n) \tag{3.1}$$

Here we will define ideal lattices and is based on Micciancio [26] and Lyubashevsky and Micciancio [24]. First, recall that an ideal I, $I \subseteq R$ such that $I \neq \emptyset$, and for all $a, b \in I$ and $r \in R$, it holds that $-a \in I$, $a + b \in I$ and $ar \in I$.

A cyclic lattice is a lattice $\mathcal{L} \subseteq \mathbb{Z}^n$ with the property that if $(a_1, a_2, \ldots, a_n) \in \mathcal{L}$ then also $(a_{i+1}, \ldots, a_n, a_1, \ldots, a_i) \in \mathcal{L}$ for all $i \in [1, n]$, the letter that all rotations of a vector are contained in the lattice. Recall that vectors can also be expressed as polynomials, if $\mathcal{L}$ is a cyclic lattice isomorphic to $\mathbb{Z}[x]/(x^n - 1)$ and $\mathbf{a}$ is a lattice vector, then also $x\mathbf{a} \bmod x^n - 1$ is a lattice vector. This can be seen in following equation:

$$x\mathbf{a} = x\sum_{i=1}^{n} a_i x^{i-1} = \sum_{i=1}^{n} a_i x^i \equiv a_n + \sum_{i=1}^{n-1} a_i x^i \bmod x^n - 1 \tag{3.2}$$

Note that $x\mathbf{a}$ corresponds to the cyclic shift $(a_n, a_1, \ldots, a_{n-1})$. Inductively we have that also $x^i\mathbf{a} \in \mathcal{L}$, for integer i. Only few lattices are cyclic lattices, thus needing a cyclic lattice is in practice very restrictive. Therefore the interest of researchers shifted to lattices isomorphic to $\mathbb{Z}[x]/(f(x))$ for various monic $f(x)$, which has lead to a more general class of lattices, named *ideal lattices*. To clarify, Ideal lattices are generalization of cyclic lattices.

Definition 9. Let I be an ideal of $\mathbb{Z}[x]/(f(x))$ where $\deg(f) = n$ and $\mathcal{L}_I = \{(a_0, a_1, \ldots, a_{n-1}) | a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \in I\}$. We call $\mathcal{L}_I$ an ideal lattice.

This shows that the elements of an ideal lattice can be interpreted as polynomials from an ideal over $\mathbb{Z}[x]/(f(x))$.

Theorem 3.1.1. Let $\mathcal{L}$ be a lattice that corresponds to an ideal in the ring $\mathbb{Z}[x]/(x^n + 1)$ and let $\mathbf{u} \in \mathcal{L}$ be a vector in lattice. Then the vectors: $\mathbf{u}, x\mathbf{u}, x^2\mathbf{u}, \ldots, x^{n-1}\mathbf{u}$ are linearly independent.

Note that $x^n + 1$ is an irreducible polynomial when n is a power of 2. In case of cyclic lattices or lattices of the form $\mathbb{Z}[x]/(x^n - 1)$, is not ideal lattice because $x^n - 1$ is a reducible polynomial.

3.1.3 Lattice problems

The way lattices can be used in cryptography is by no means obvious, and was discovered in a breakthrough paper by Ajtai. His result has by now developed into a whole area of research whose main focus is on expanding the scope of lattice-based cryptography and on creating more practical lattice-based cryptosystems. Before discussing this area of research in more detail, let us first describe the the computational problems involving lattices, whose presumed hardness lies at the heart of lattice-based cryptography. Let $\mathcal{L}$ be a lattice of dimension n. The two fundamental computational problems in lattices are:

- Shortest Vector Problem (SVP): find a shortest non-zero vector in $\mathcal{L}$
- Closest Vector Problem (CVP): given a vector $t \in \mathbb{R}^n$ not in $\mathcal{L}$, find a vector in $\mathcal{L}$ that is closest to t

Definition 10. Shortest Independent Vector Problem (SIVP): given a basis of a lattice $\mathcal{L}$ and a parameter $q \in \mathbb{Z}$, find the shortest q linearly independent lattice vectors (i.e. the set of lattice vectors $\mathbf{b}_1, \ldots, \mathbf{b}_q$ contained within the minima λ_q)

Definition 11. Closest Vector Problem (CVP): let $\mathbf{t}$ be some vector. The Closest Vector Problem is to find a vector $\mathbf{v} \in \mathcal{L}$, such that $||\mathbf{v}-\mathbf{t}|| = \text{dist}(\mathbf{t}, \mathcal{L})$ where $\text{dist}(\mathbf{t}, \mathcal{L}) = \min_{\mathbf{u}\in\mathcal{L}} ||\mathbf{u}-\mathbf{t}||$

Definition 12. An instance of GapSVP$_\gamma$ (or decision SVP$_\gamma$) is given by an n-dimensional lattice $\mathcal{L}$ and a number $d > 0$. In YES instances, $\lambda_1(\mathcal{L}) \leq d$ whereas in NO instances $\lambda_1(\mathcal{L}) > \gamma(n) \times d$

The problem GapSVP$_\gamma$ consists of differentiating between the instances of SVP in which the answer is at most 1 or larger than γ, where γ can be a fixed function of n, the number of vectors. Given a basis for the lattice, the algorithm must decide whether $\lambda(\mathcal{L}) \leq 1$ or $\lambda(\mathcal{L}) > \gamma$. Like other promise problems, the algorithm is allowed to error on all other cases.

Definition 13. An instance of SIVP$_\gamma$ is given by an n-dimensional lattice $\mathcal{L}$. The goal is to output a set of n-linearly independent lattice vectors of length at most $\gamma(n) \times \lambda_n(\mathcal{L})$

Definition 14. Approximate Shortest Vector Problem (SVP$_\gamma$): given a lattice $\mathcal{L}$, the Approximate Shortest Vector Problem is to find a non-zero vector $\mathbf{v} \in \mathcal{L}$, such that $||\mathbf{v}|| \leq \gamma\lambda_1(\mathcal{L})$

Definition 15. Approximate Closest Vector Problem (CVP$_\gamma$): let $\mathbf{t}$ be some vector. The Approximate Closest Vector Problem is to find a vector $\mathbf{v} \in \mathcal{L}$ such that $||\mathbf{v}-\mathbf{t}|| \leq \gamma \ \text{dist}(\mathbf{t}, \mathcal{L})$ where $\text{dist}(\mathbf{t}, \mathcal{L}) = \min_{\mathbf{u}\in\mathcal{L}}||\mathbf{u} - \mathbf{t}||$

Notice if $\gamma = 1$, then SVP$_\gamma$ and CVP$_\gamma$ would be the same as SVP and CVP

Definition 16. Bounded Distance Decoding Problem (BDD$_\gamma$): given $\mathcal{L}$ and a target point $\mathbf{t} \in \mathbb{R}^n$ such that $\text{dist}(\mathbf{t}, \mathcal{L}) \leq \gamma\lambda_1(\mathcal{L})$, output a vector $\mathbf{v} \in \mathcal{L}$ such that $||\mathbf{v} - \mathbf{t}|| \leq \gamma\lambda_1(\mathcal{L})$

In SVP, a basis of a vector space V and a l_2 norm are given for a lattice $\mathcal{L}$ and one must find the shortest non-zero vector in V, in $\mathcal{L}$. In other words, the algorithm should output a non-zero vector $\mathbf{v}$ such that $||\mathbf{v}|| = \lambda(\mathcal{L})$. The closest vector problem is a generalization of the shortest vector problem. It is would be trivial to show that given an oracle for CVP$_\gamma$, one can solve SVP$_\gamma$ by making some queries to the oracle. However, the naive method to find the shortest vector by calling the CVP$_\gamma$ oracle to find the closest vector to 0 does not work because 0 is itself a lattice vector and the algorithm could potentially output 0. The correct reduction is explained in [17].

Theorem 3.1.2. (Minkowski): Let $\mathcal{L}$ be lattice of $\mathbb{R}^n$ with rank n, the length of shortest lattice vector of $\mathcal{L}$, $\lambda_1 \leq \sqrt{n} \times \det(\mathcal{L})^{\frac{1}{n}}$

The above theorem gives an upper bound of the length of shortest lattice vectors. Even though it is only a theorem of existence, the upper bound has significant practical interest. In the γ-approximation version SVP$_\gamma$, one must find a non-zero lattice vector of length at most $\gamma\lambda_1(\mathcal{L})$ (note that vector 0 is also in lattice). The exact version of the problem is only known to be NP-hard for randomized reductions [2] [20]. Approach techniques: LLL lattice basis reduction algorithm which runs in polynomial time produces a relatively short vector in polynomial time, but does not solve the problem. HKZ basis reduction algorithm solves the problem in O(n^{2n}) time where n is the dimension.

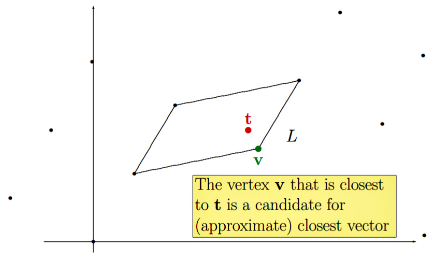

t
v
L
The vertex **v** that is closest to **t** is a candidate for (approximate) closest vector

Figure 3.7: Visualization of CVP problem along with fundamental parallelepiped of lattice
The vertex v of the fundamental domain that is closest to t will be a close lattice point if the basis is *good*, meaning if the basis consists of short vectors that are reasonably orthogonal to one another.

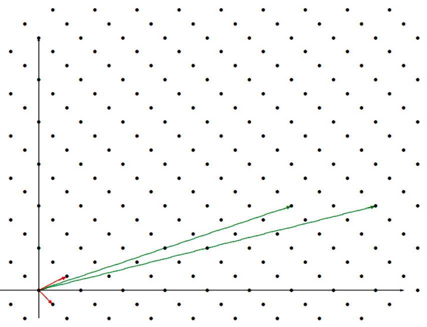


Figure 3.8: Visualization of two different lattice basis
A *good* basis that is short (in red) and a *bad* basis that is relatively longer (in green)

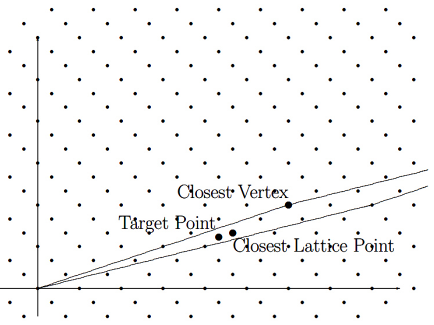


Figure 3.9: Visualization *bad* lattice basis in solving CVP problem
Here is the parallelogram spanned by a *bad* basis and a CVP target point. It is easy to find the vertex of the parallelogram that is closest to the target point. However, the lattice point that actually solves CVP is much closer to the target than the closest vertex found using bad basis. Therefore, having an access to short lattice vectors would make CVP problem straightforward to solve.

3.1.4 Worst-case hardness of lattice problems

Let us describe Ajtai's result more precisely. The cryptographic construction given in [1] is known as a family of one-way functions. Ajtai proved that the security of this family can be based on the worst-case hardness of the n^c-approximate SVP for some constant c. In other words, the ability to invert a function chosen from this family with non-negligible probability implies an ability to solve any instance of n^c-approximate SVP. In his seminal papers, Ajtai showed that the SVP problem is NP-hard and discovered some connections between the worst-case complexity and average-case complexity of some lattice problems. Building on these results, Ajtai and Dwork created a public-key cryptosystem whose security could be proven using only the worst-case hardness of a certain version of SVP, thus making it the first attempt to use worst-case hardness to create secure systems [3].

Strong security guarantees from worst-case hardness. Cryptography inherently requires average-case intractability, i.e., problems for which random instances drawn from a specified probability distribution are hard to solve. This is qualitatively different from the worst-case notion of hardness usually considered in the theory of algorithms and NP-completeness, where a problem is considered hard if there merely exist some intractable instances. Problems that appear hard in the worst-case often turn out to be easier on the average.

Remark 1. To solve SIVP, an algorithm must work for any given input basis $\mathbf{B}$. One can also formulate an average-case variant of SIVP, where the input basis is generated at random according to some probability distribution.

Ajtai in [1] gave connection between the worst-case and the average-case for lattices: he proved that certain problems are hard on the average (for cryptographically useful distributions), as long as some related lattice problems are hard in the worst-case. In details, Ajtai proved that, when $\mathbf{A}$ in $f_{\mathbf{A}} : \mathbf{x} \mapsto \mathbf{A}\mathbf{x} \bmod q$ is chosen uniformly at random, a suitable restriction of function $f_{\mathbf{A}}$ is at least as hard to invert on the average as the worst-case complexity of approximating certain lattice problems within a polynomial factor. Using results of this kind, one can design cryptographic constructions and prove that they are infeasible to break, unless all instances of certain lattice problems are easy to solve (worst implies any and average implies random). In essence, in lattice-based cryptosystem, for a fixed security parameter n, what the reduction shows is the existence of a solver for the lattice problem on input any n-dimensional lattice using the adversary breaking a lattice-based cryptosystem with the security parameter n on the average-case. Therefore, since we can solve any instance, we can solve the hardest one of dimension n.

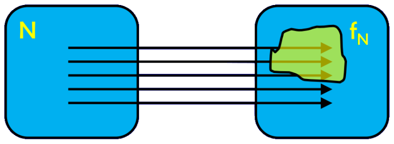


Figure 3.10: Average-case problem (e.g. factorization)

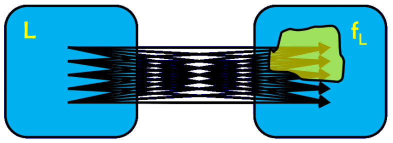


Figure 3.11: Worst-case problem (e.g. lattice problems)

In average-case hardness, the cryptographic function is hard provided almost all N are hard to factor; but in worst-case hardness the cryptographic function is hard provided the lattice problem is hard in the worst-case, this is a much stronger security guarantee and assures us that our distribution is correct. If we solve 1% of lattice based cryptographic function, then we can solve *all instances* of lattice problems.

Following Ajtai's worst-case to average-case reduction many different improvements on estimating SIS hardness have been done. This series of results is summed up in this theorem:

Theorem 3.1.3. ([27], Theorem 5.16) For any $m = poly(n)$, any $\beta > 0$, and any sufficiently large $q \geq \beta \cdot poly(n)$, solving $\text{SIS}_{n,q,\beta,m}$ is at least as hard as solving the decisional approximate shortest vector problem GapSVP_γ and the approximate shortest independent vectors problem SIVP_γ on arbitrary n-dimensional lattices, for some $\gamma = \beta \cdot poly(n)$.

The importance of the worst-case security guarantee is assurance that there are no fundamental flaws in the design of our cryptographic construction.

3.2 Lattice reduction

The goal of lattice basis reduction is given an integer lattice basis as input, to find a basis with short, nearly orthogonal vectors. One measure of nearly orthogonal is the orthogonality defect. This compares the product of the lengths of the basis vectors with the volume of the parallelepiped they define. For perfectly orthogonal basis vectors, these quantities would be the same.

Any particular basis of n vectors may be represented by a matrix $\mathbf{B}$, whose columns are the basis vectors $\mathbf{b}_i$, for $i = 1, \ldots, n$. In the fully dimensional case where the number of basis vectors is equal to the dimension of the space they occupy, this matrix is square, and the volume of the fundamental parallelepiped is simply the absolute value of the determinant of this matrix $\det(\mathbf{B})$. If the number of vectors is less than the dimension of the underlying space, then volume is $\sqrt{\det(\mathbf{B}^T\mathbf{B})}$. For a given lattice $\mathcal{L}$, this volume is the same (up to sign) for any basis, and hence is referred to as the determinant of the lattice denoted by $\det(\mathcal{L})$ or lattice constant $d(\mathcal{L})$.

The orthogonality defect is the product of the basis vector lengths divided by the parallelepiped volume:

$$\delta(\mathbf{B}) = \frac{\Pi_{i=1}^{n}||\mathbf{b}_i||}{\sqrt{\det(\mathbf{B}^T\mathbf{B})}} = \frac{\Pi_{i=1}^{n}||\mathbf{b}_i||}{d(\mathcal{L})} \tag{3.3}$$

From the geometric definition it may be appreciated that $\delta(\mathbf{B}) \geq 1$ with equality if and only if the basis is orthogonal. If the lattice reduction problem is defined as finding the basis with the smallest possible defect, then the problem is NP-complete. However, there exist polynomial time algorithms to find a basis with defect $\delta(\mathbf{B}) \leq c\delta(\mathbf{B}) \leq c$ where c is some constant depending only on dimension of the underlying space.

Although determining the shortest basis is an NP-complete problem, algorithms such as LLL algorithm can find a short, *not necessarily shortest*, basis in polynomial time with guaranteed worst-case performance. The following example demonstrates the computation of lattice reduction algorithm by which we can reduce original basic matrix from: $\mathbf{B} = \left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right) \rightarrow \mathbf{B}' = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 2 \end{smallmatrix}\right)$ and matrix of $\mathbf{U} = \left(\begin{smallmatrix} -2 & 1 \\ 3 & -1 \end{smallmatrix}\right)$ is a unimodular matrix with determinant of -1. Also, $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$ because: $\mathbf{B}' = \mathbf{B} \times \mathbf{U}$.

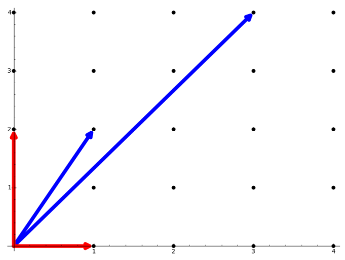


Figure 3.12: Lattice generated by vectors: $\mathbf{v}_1 = (1, 2)$ and $\mathbf{v}_2 = (3, 4)$
Reduced basis consisting of vectors: $\mathbf{v}'_1 = (1, 0)$ and $\mathbf{v}'_2 = (0, 2)$ generates the same lattice

3.2.1 LLL **algorithm**

The Lenstra–Lenstra–Lovász (LLL) lattice basis reduction algorithm is a polynomial time lattice reduction algorithm invented by A. Lenstra, H. Lenstra and L. Lovász in 1982 [21]. Given a basis $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_d\}$ with n-dimensional integer coordinates, for a lattice $\mathcal{L}$ with $d \leq n$, LLL algorithm calculates an LLL-reduced (short, nearly orthogonal) lattice basis in time $\mathrm{O}(d^5 n \log^3 B)$, where B is the largest length of $\mathbf{b}_i$ under the Euclidean norm and d is referring to number of basis vectors in $\mathbf{B}$ (we should use d because the dimension of the lattice might be smaller than the dimension of the $\mathbb{R}^n$ that the lattice is embedded in).

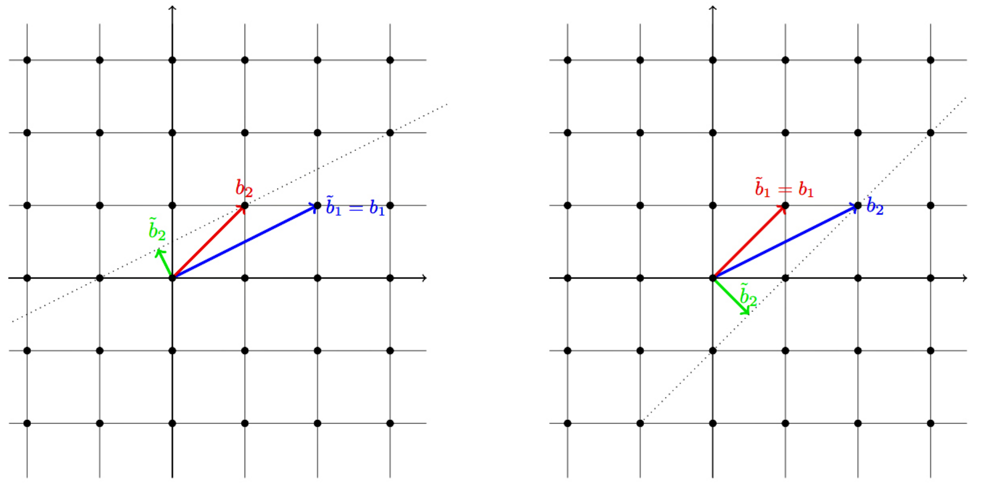
LLL can be used as a cryptanalysis tool (i.e., to break cryptography):

1. Knapsack-based cryptosystem [14] (appendix A describes in details how LLL breaks early knapsack cryptosystem)

2. RSA when the public exponent e is small or when partial knowledge of the secret key is available [38] [10]. LLL is used to find a polynomial that has the same zeroes as the target polynomial but smaller coefficients.

In order to achieve an orthogonal basis, an iterative process can be taken whereby each vector is projected onto a hyperplane perpendicular to the previous vectors. The Gram-Schmidt Orthogonalization algorithm is an iterative approach to orthogonalizing the vectors of a basis. The first vector $\mathbf{b}_1$ of a given basis $\mathbf{B}$ is taken as a reference and the second vector $\mathbf{b}_2$ is projected on to an $(n-1)$-hyperplane perpendicular to $\mathbf{b}_1$. The third vector $\mathbf{b}_3$ is projected onto a $(n-2)$-hyperplane perpendicular to the plane described by $\mathbf{b}_1$ and $\mathbf{b}_2$. This process continues in an iterative fashion until all degrees of freedom are exhausted. The new orthogonal vectors are denoted $\mathbf{b}_i^*$ and the new basis as $\mathbf{B}^*$.

$$\begin{aligned} \mathbf{b}_i^* &= \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j}\mathbf{b}_j^* \\ \mu_{i,j} &= \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} \end{aligned} \tag{3.4}$$



(a) Gram-Schmidt orthogonalization of the vectors b_1 and b_2 in that order.

(b) Gram-Schmidt orthogonalization of the same vectors, but in the opposite order.

Figure 3.13: Gram-Schmidt Orthogonalization.
The vectors $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$ do not form a lattice basis. In fact, the Gram-Schmidt vectors are not necessarily in the lattice.

The precise definition of LLL-reduced basis is as follows: Given a basis $\mathbf{B} = \{\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n\}$, define its Gram–Schmidt process orthogonal basis: $\mathbf{B}^* = \{\mathbf{b}_0^*, \mathbf{b}_1^*, \dots, \mathbf{b}_n^*\}$, and the Gram-Schmidt coefficients $\mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$, for any $1 \leq j < i \leq n$.

Then the basis $\mathbf{B}$ is LLL-reduced if there exists a parameter $\delta \in (0.25, 1]$ such that the following holds:

1. (size-reduced) For $1 \leq j < i \leq n$: $|\mu_{i,j}| \leq 0.5$. By definition, this property guarantees the length reduction of the ordered basis.

2. (Lovász condition) For $k = 1, 2, \dots, n$: $\delta \|\mathbf{b}_{k-1}^*\|^2 \leq \|\mathbf{b}_k^*\|^2 + \mu_{k,k-1}^2 \|\mathbf{b}_{k-1}^*\|^2$. Here, estimating the value of the δ parameter, we can conclude how well the basis is reduced. Greater values of δ lead to stronger reductions of the basis.

Remark 2. The first condition assures the resulting basis is nearly orthogonal. But length reduction alone does not guarantee near orthogonality.

The issue with the length-reduction criterion alone (and the reason the Lovász condition is included in the LLL algorithm) is that the following basis satisfies it:

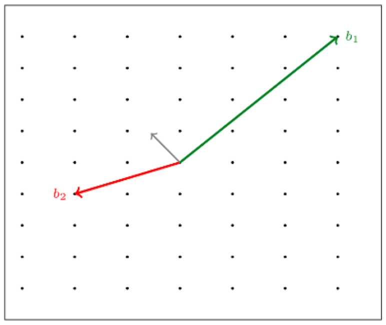


Figure 3.14: Length-reduced lattice basis
The grey arrow is the projection of $\mathbf{b}_2$ to the orthogonal complement of $\mathbf{b}_1$

Clearly, this basis is not very short, nor is it close to being orthogonal (i.e., note that length reduction alone does not necessary imply almost-orthogonality). The Lovász condition is suited to detect such a situation: If the order of the vectors is swapped, another length reduction can easily be performed, yielding the following LLL-reduced basis:

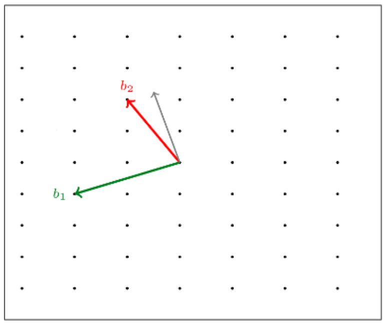


Figure 3.15: LLL-reduced lattice basis

A. Lenstra, H. Lenstra and L. Lovász have noticed that a situation in which length reduction alone is stuck with a very bad basis - as depicted in the first image - always has vectors which are in the "wrong" order comparing their lengths and which are far from being orthogonal, and that it helps in this case to exchange them and continue length-reducing. That is really the core idea of the LLL algorithm.

In summary, the Lovász condition is fulfilled if the vectors are close enough to being orthogonal, or if they are roughly ordered by length. Both of these properties lead to length reduction being quite effective. Initially, A. Lenstra, H. Lenstra and L. Lovász demonstrated the LLL-reduction algorithm for $\delta = \frac{3}{4}$. Note that although LLL-reduction is well-defined for $\delta = 1$, the polynomial-time complexity is guaranteed only for $\delta \in (0.25, 1)$.

The following description of LLL algorithm is based on (Hoffstein, Pipher & Silverman 2008 [18], Theorem 6.68), **Input:**

1. a lattice basis $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{Z}^m$

2. parameter δ with $\frac{1}{4} < \delta < 1$, most commonly $\delta = \frac{3}{4}$

Algorithm 1 LLL lattice reduction Algorithm

```
procedure LLL($\mathbf{b}, \delta$)                                    ▷ Basis and delta
    $ortho := \text{gramSchmidt}(\{\mathbf{b}_0, \ldots, \mathbf{b}_n\}) = \{\mathbf{b}_0^*, \ldots, \mathbf{b}_n^*\}$
                                  ▷ $\mathbf{b} \leftarrow$ Perform Gram-Schmidt, but do not normalize
    Define $\mu_{i,j} := \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$, which must always use the most current values of $\mathbf{b}_i, \mathbf{b}_j^*$.
    $k \leftarrow 1$
    while $k \leq n$ do
        for $j$ form $k-1$ to $0$ do
            if $|\mu_{k,j}| > \frac{1}{2}$ then $\mathbf{b}_k = \mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$
                                  ▷ Update ortho entries and related $\mu_{i,j}$'s as needed
      ▷ Naive method is to recompute $ortho := \text{gramSchmidt}(\{\mathbf{b}_0, \ldots, \mathbf{b}_n\}) = \{\mathbf{b}_0^*, \ldots, \mathbf{b}_n^*\}$
  when a $\mathbf{b}_i$ changes)
        if $\langle \mathbf{b}_k^*, \mathbf{b}_k^* \rangle \geq (\delta - (\mu_{k,k-1})^2) \langle \mathbf{b}_{k-1}^*, \mathbf{b}_{k-1}^* \rangle$ then
            $k = k + 1$
        else
            swap $\mathbf{b}_k$ and $\mathbf{b}_{k-1}$       ▷ Update ortho entries and related $\mu_{i,j}$'s as needed
            $k = \max(k-1, 1)$

    return: LLL reduced basis $\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_n$
```

3.2.2 The BKZ reduction algorithm

Blockwise Korkine-Zolotarev (or BKZ) reduction is a reduction algorithm for lattices. It has been introduced by Schnorr and Euchner [31]. The BKZ algorithm uses so called local blocks to achieve reduction, hence the name. The quality of the reduction that is achieved by BKZ reduction depends on this block size. Increasing block sizes also mean an improvement in the reduction. The BKZ algorithm starts by LLL-reducing a given basis of a lattice. The quality of the reduction is then iteratively improved. This improvement is achieved using the local blocks determined by the input basis and the block size.

Running BKZ algorithm with large block size results in shorter and more orthogonal basis in compassion with LLL, hence it can be used to attack underlying lattice of LWE.

3.3 Short integer solution problem (SIS)

SIS is an average case problem that is used in lattice-based cryptography constructions. Lattice-based cryptography began in 1996 from a seminal work by Ajtai who presented a family of one-way functions based on SIS problem. He showed that it is secure in average-case if SVP_γ (where $\gamma = n^c$ for some constant $c > 0$) is hard in worse-case scenario.

A trapdoor function is a function that is easy to perform one way, but has a secret that is required to perform the inverse calculation efficiently. That is, if f is a trapdoor function, then $y = f(x)$ is easy to compute, but $x = f^{-1}(y)$ is hard to compute without some special knowledge k. Given k, then it is easy to compute $y = f^{-1}(x, k)$. The analogy to a "trapdoor" is something like this: It is easy to fall through a trapdoor, but it is very hard to climb back out and get to where we started unless we have a ladder.

Hash-function is not a trapdoor function because it is not reversible. Instead, it is called a one-way function. One-way function is similar to a trapdoor function in that it's easy to compute and it's very hard to reverse, but there is no special key that allows to reverse the one-way function. For example: RSA is a trapdoor function because it's inverse which is a factorization is hard, but SHA-3 is just a one-way hash function as it's output can be hash of more that one inputs since hash functions are not a one-to-one function.

Definition 17. Given parameters $m, n, q \in \mathbb{Z}$, key $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and input $\mathbf{x} \in \{0,1\}^m$, Ajtai's one-way hash function is a function that outputs $f_{\mathbf{A}}(\mathbf{x}) = \mathbf{A}\mathbf{x} \bmod q$

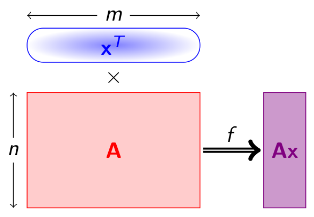


Figure 3.16: Visualization of Ajtai's one-way function

```
from sage.crypto import gen_lattice
from sage.matrix.matrix_space import MatrixSpace
```

```
dimension = 16
modulus = random_prime(2^200, lbound=2^100) # setting upper / lower bound of
                                            # random prime generator

ring = GF(modulus) # Gaussian field modulo prime

# generate a random N x 1 matrix over the Finite Field
x = MatrixSpace(ring, dimension, 1).random_element()

# generate a random matrix and test if matrix forms a lattice
# that is vectors are linearly independent
m = Matrix(gen_lattice(type="modular", m=dimension, q=modulus, lattice=True).basis())

print hex(int(list(sum(m * x)).pop()))
# e.g. output: 0x2baa49dd9e5d3daf27408f8c7cf049bc24075572c7e8bc3061L
```

Listing 3.1: Ajtai hash function given a secret matrix m random value x which is a column vector

Theorem 3.3.1. For $m > n \log q$, if lattice problems (i.e. SIVP) are hard to approximate in the worst-case, then $f_{\mathbf{A}}(\mathbf{x}) = \mathbf{Ax} \bmod q$ is a one-way function.

Definition 18. $\mathrm{SIS}_{n,m,q,\beta}$: let $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ be an $n \times m$ matrix with entries in $\mathbb{Z}_q$ that consists of m uniformly random vectors $\mathbf{a}_i \in \mathbb{Z}_q^n$: $\mathbf{A} = (\mathbf{a}_1, \cdots, \mathbf{a}_m)$. Find a nonzero vector $\mathbf{x} \in \mathbb{Z}^m$ such that:

- $\|\mathbf{x}\| \le \beta$
- $f_{\mathbf{A}}(\mathbf{x}) : \mathbf{Ax} = 0 \in \mathbb{Z}_q^n$

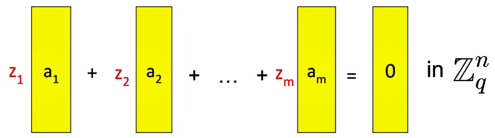


Figure 3.17: Visualization of SIS problem introduces by Ajtai

It should be noted that a solution to SIS without the required constrain on the length of the solution is easy to compute by using Gaussian elimination technique. We also require $\beta < q$, otherwise $x = (q, 0, \ldots, 0) \in \mathbb{Z}^m$ is a trivial solution.

In order to guarantee $f_{\mathbf{A}}(\mathbf{x})$ has non-trivial, short solution, we require:

- $\beta \geq \sqrt{n \log q}$, and
- $m \geq n \log q$ (to get compression)

Remark 3. SIS problem can also form a collision-Resistant Hash function:

Given: $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_m) \in \mathbb{Z}_q^{n \times m}$, define $\text{Hash}_{\mathbf{A}} : \{0,1\}^m \mapsto \mathbb{Z}_q^n$ where $\text{Hash}_{\mathbf{A}}(z_1, \ldots, z_m) = a_1 z_1 + \cdots + a_m z_m$. Domain of $h = \{0,1\}^m$ (size $= 2^m$). Range of $h = \mathbb{Z}_q^n$ (size $= q^n$). Set $m > n \log q$ to get compression. Collision would occur when:

$$\mathbf{a}_1 z_1 + \cdots + \mathbf{a}_m z_m = \mathbf{a}_1 y_1 + \cdots + \mathbf{a}_m y_m \tag{3.5}$$

By adjusting the above we get: $\mathbf{a}_1(z_1 - y_1) + \cdots + \mathbf{a}_m(z_m - y_m) = 0$ and $z_i - y_i$ are in $\{-1, 0, 1\}$. Nothing happens as trivial solution is not a valid solution to SIS problem, further because the only way we could find a collision is when $z_i - y_i = 0 \bmod q$. So SIS is a collision resistant hash function.

Regrading the efficiency of Ajtai's trapdoor function, matrix multiplication of $1 \times m$ matrix with $m \times n$ matrix would yield O(mn) time complexity. Hence, it is efficient in run-time but has very high space needs. The space needs can be reduced by number of ways. One idea could be use a structured matrix i.e. take the first row and the second row would be one bit left/ right circular of the first row and so on allowing just the first row of the matrix to be stored as the public key. Hermite Normal form (i.e. upper triangular matrix) could also be used for the public key to reduce half of the size of the matrix.

In 1996, Ajtai showed that, for good parameters, if there exists an inverter of his hash functions, then there exists an algorithm finding a short vector from any n-dimensional lattice. Goldreich, Goldwasser, and Halevi pointed out if there exists an algorithm that finds a collision in the Ajtai hash function, then there exist an algorithm that finds a short vector from any n-dimensional lattice. The reductions are based on the fact that the sum of short vectors is (relatively) short.

3.3.1 Interpreting SIS problem in lattice words and lattice reduction

Define a lattice related to a matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ as

$$\Lambda = \Lambda_q^{\perp}(\mathbf{A}) = \{\mathbf{e} \in \mathbb{Z}^m \mid \mathbf{A}\mathbf{e} \equiv \mathbf{0} \bmod q\}$$

This Λ is an m-dimensional q-ary lattice that corresponds to the linear code with parity check matrix equal to $\mathbf{A} \bmod q$ (we can check this is a lattice by verifying that for any two vectors $\mathbf{e}, \mathbf{e}' \in \Lambda$, the sum of them lies in Λ again. We can check this is m-dimensional by verifying that any $q\mathbf{u}_i$ is in Λ, where $\mathbf{u}_i$ is a unit vector).

This lattice defines a map from $\mathbb{Z}^m$ to a quotient group $\mathbb{Z}^m/\Lambda$. The map is a hash function. Conversely, a collision $\mathbf{e} \neq \mathbf{e}'$ for $f_{\mathbf{A}}$ implies a short non-zero vector $\mathbf{e} - \mathbf{e}'$ in $\Lambda_q^{\perp}(\mathbf{A})$.

Solving SIS (or breaking Ajtai's trapdoor function) comes down to finding a short vector in the underlying lattice (or $\Lambda_q^{\perp}$). Lattice basis reduction methods like LLL help in the sense that they can reduce a basis with long lattice vectors to a basis with shorter, more orthogonal lattice vectors. If the reduction is strong enough (e.g. BKZ with large enough block size) then we can expect that the first basis vector of the reduced basis is a solution to SIS (how strong the basis reduction should be to find a solution depends on n, m, q). To summarize, LLL can solve the easiest SIS instances, and can assist in solving harder SIS instances by finding a shorter basis, which makes finding even shorter lattice vectors a bit easier.

Finding a short nonzero $z \in \Lambda_q^{\perp}(\mathbf{A})$ for uniformly random $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, where $m \approx n \log q$ (SIS problem) can be reduced to solving $\text{GapSVP}_{\beta\sqrt{n}}$, $\text{SIVP}_{\beta\sqrt{n}}$ on any n-dimensional lattice. In essence, let $\mathbf{S}$ be the set of all integer $z = (z_1, \dots z_n)$ such that $z_1 \times \mathbf{a}_1, \dots, z_n \times \mathbf{a}_n = 0 \bmod q$, then $\mathbf{S}$ is a lattice and SIS problem asks to find a short vector in $\mathbf{S}$. SIS problem is a an average-case hard problem but SVP problem is a worst-case hard problem.

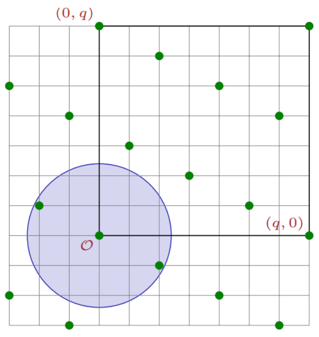


Figure 3.18: Visualization of SIS reduced to GapSVP$_{\beta\sqrt{n}}$, SIVP$_{\beta\sqrt{n}}$
Notice that we are looking for smallest radius to contain enough lattice points to reconstruct the lattice itself.

3.4 LWE and lattice cryptography

The learning with errors (LWE) problem is to efficiently distinguish between vectors created from a "noisy" set of linear equations versus uniformly random vectors. Given a matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ and a vector $\mathbf{v} \in \mathbb{Z}_q^m$, the goal is to determine whether $\mathbf{v}$ has been sampled uniformly at random from $\mathbb{Z}_q^m$ or whether $\mathbf{v} = \mathbf{A}\mathbf{s} + \mathbf{e}$ for some random $\mathbf{s} \in \mathbb{Z}_q^m$ and $\mathbf{e} \in \chi_m$, where χ is a small "noise" distribution over $\mathbb{Z}_q$.

LWE problem is very closely related to coding theory. If we choose the parameter $q = 2$, this becomes the well-studied learning parity with noise (LPN) problem, which is believed to be hard. Recovering the key from the more general LWE problem is essentially equivalent to decoding a noisy linear code, also a long established difficult problem in coding theory. However, for modern cryptographic purposes it is more important to ensure indistinguishability of encryption rather than just security against key recovery. For this purpose it helps to look at the problem from a lattice-based perspective. The vector $\mathbf{v} = \mathbf{A}\mathbf{s} + \mathbf{e}$ can be seen as an element of the q-ary lattice $\Lambda_q^\perp$ with a small perturbation vector added. The task here is to distinguish this from a uniformly random vector. In 2005, Regev [30] formalised this relationship by giving a reduction from worst-case lattice problems to LWE. The informal description of the reduction is as follows: if there exists an efficient algorithm that solves LWE then there exists an efficient algorithm that approximates the decision version of the shortest vector problem (or GapSVP).

Simple properties of LWE :

1. Check a candidate solution $\mathbf{s}' \in \mathbb{Z}_q^n$, we test if $\mathbf{b} - \langle \mathbf{s}', \mathbf{a} \rangle$ is small. If $\mathbf{s}' \neq \mathbf{s}$, then $\mathbf{b} - \langle \mathbf{s}', \mathbf{a} \rangle = \langle \mathbf{s} - \mathbf{s}', \mathbf{a} \rangle + \mathbf{e}$ is well spread in $\mathbb{Z}_q$

2. Shift the secret by any $\mathbf{t} \in \mathbb{Z}_q^n$ given $(\mathbf{a}, \mathbf{b} = \langle \mathbf{s}, \mathbf{a} \rangle + \mathbf{e})$ output:

$$(\mathbf{a}, \mathbf{b}' = \mathbf{b} + \langle \mathbf{t}, \mathbf{a} \rangle = \langle \mathbf{s} + \mathbf{t}, \mathbf{a} \rangle + \mathbf{e}) \tag{3.6}$$

This property allows us to amplify the success probability for random $\mathbf{t}$.

Difference / relation between SIS **and** LWE**:**

1. SIS problem has many valid solutions but LWE has a unique solution

2. If we have a SIS oracle, then we can ask the oracle to find short vector $\mathbf{z}$ such that $\mathbf{Az} = 0 \bmod q$ and as $\mathbf{b}^t = \mathbf{s}^t\mathbf{A} + \mathbf{e}^t$, then $\mathbf{b}^t\mathbf{z} = (\mathbf{As} + \mathbf{e}) \cdot \mathbf{z} = 0 + \mathbf{e}^t\mathbf{z} = \mathbf{ez}$. Note that $\mathbf{ez}$ is small because $\mathbf{z}$ is short but $\mathbf{b}^t\mathbf{z}$ is well spread. Therefore, we just solved LWE. One can find a short vector in the LWE-lattice using a lattice basis reduction method, e.g. LLL reduction. This *distinguishing attack* on LWE is described in details in [22].

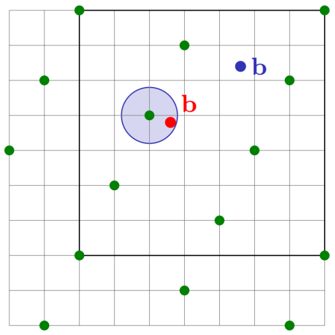


Figure 3.19: Visualization of LWE reduced to Average-case BDD problem
$\mathbf{b}^t$ (in red) $= \mathbf{s}^t\mathbf{A} + \mathbf{e}^t$ vs. $\mathbf{b} \leftarrow \mathbb{Z}_q^m$

3.4.1 LWE problem

Given samples $(x, y = f(x) + \mathrm{e})$ where $x \in \mathbb{Z}_q^n$, $y \in \mathbb{Z}_q$ and a linear function f such that $f : \mathbb{Z}_q^n \to \mathbb{Z}_q$, the idea is to find f or close approximation or it knowing that e (error) comes from a some known noise model.

Definition 19. Learning with errors instance $\mathrm{LWE}_{n,q,\chi}$ is parametrized by:

- $n \in \mathbb{N}$
- $q \in$ Primes
- χ, a probability distribution over $\mathbb{Z}/q\mathbb{Z}$

χ is known as the noise distribution and we would like it to generate short elements, i.e. $||e|| \leq B$ with high probability for some bound $B \ll q$, when $e \leftarrow \chi$. In practice, χ is usually a discrete Gaussian over $\mathbb{Z}$.

Theorem 3.4.1. ([30], Theorem 1.1) Let n, p be integers and $\alpha \in (0, 1)$ be such that $\alpha p > 2\sqrt{n}$. If there exists an efficient algorithm that solves $\mathrm{LWE}_{q,\Psi_\alpha}$ then there exists an efficient quantum algorithm that approximates the decision version of the shortest vector problem (GapSVP) and the shortest independent vectors problem (SIVP) in the worst-case.

For the above theorem to work, it is necessary that χ is chosen to be a discrete Gaussian distribution. This means that sampling from LWE involves taking a lattice point and perturbing it by a small, normally distributed quantity, the idea being that this will look close enough to a uniform distribution if the standard deviation is large enough. Sampling from this discrete Gaussian is simply accomplished by sampling each component from a normal distribution and rounding to the nearest integer.

3.4.2 Search LWE

Suppose we are given an oracle $\mathcal{O}_s^n$ which outputs samples of the form $(a, \langle a, s \rangle + e)$,

- $\mathbf{a} \leftarrow \mathbb{Z}_q^n$ is chosen freshly at random for each sample.
- $\mathbf{s} \in \mathbb{Z}_q^n$ is the secret (and it is the same for every sample).
- $\mathbf{e} \leftarrow \chi$ is chosen freshly according to χ for each sample.

The search-LWE problem is to find the secret $\mathbf{s}$ given access to $\mathcal{O}_{\mathbf{s}}^n$. The LWE$_{n,q,\chi}$ assumption is the assumption that the search-LWE problem is computationally hard.

The LWE problem described above is the search version of the problem. In the decision version (DLWE), the goal is to distinguish between noisy inner products and uniformly random samples (practically, some discretized version of it). Regev [30] showed that the decision and search versions are equivalent when q is a prime bounded by some polynomial in n.

$$\begin{aligned}
14s_1 + 15s_2 + 5s_3 + 2s_4 &\approx 8 \bmod 17 \\
13s_1 + 14s_2 + 14s_3 + 6s_4 &\approx 16 \bmod 17 \\
6s_1 + 10s_2 + 13s_3 + 1s_4 &\approx 3 \bmod 17 \\
10s_1 + 4s_2 + 12s_3 + 16s_4 &\approx 12 \bmod 17 \\
9s_1 + 5s_2 + 9s_3 + 6s_4 &\approx 9 \bmod 17 \\
3s_1 + 6s_2 + 4s_3 + 5s_4 &\approx 16 \bmod 17 \\
&\dots \\
6s_1 + 7s_2 + 16s_3 + 2s_4 &\approx 3 \bmod 17
\end{aligned} \tag{3.7}$$

The LWE problem asks to recover a secret $\in \mathbb{Z}_q^n$ given a sequence of approximate random linear equations on s. For instance, the input might be above where each equation is correct up to some small additive error (say, $\{1, -1\}$), and our goal is to recover $\mathbf{s}$ (answer in this case is $\mathbf{s} = (0, 13, 9, 11)$).

In definition of LWE, error $\mathbf{e}$ should be “short”. Short just means small (in terms of some metric, usually Euclidean norm). We can see that if $\mathbf{e}$ is the zero vector, $\mathbf{s}$ becomes trivial to recover using Gaussian elimination. If $\mathbf{e}$ is uniformly random, then you can imagine that it is impossible to recover any information on $\mathbf{s}$, since it is hidden against a uniformly random backdrop. It might be helpful to picture the following instead. Consider LWE as a lattice, if the vector $\mathbf{e}$ is small enough, $\mathbf{b}$ is close to only one of the points in this lattice. In other words, the LWE problem becomes a bounded-distance decoding problem on this lattice. If $\mathbf{e}$ is too large, $\mathbf{b}$ might be closer to another vector in this lattice. Since we are typically interested in recovering $\mathbf{s}$ and not a set of possible secrets, we must be given a guarantee that $\mathbf{e}$ is sufficiently short (hence bounded distance decoding).

So "short" just means that $\mathbf{s}$ is recoverable. Usually, $\mathbf{e}$ is sampled from a discrete distribution approximating a Gaussian centered around 0, with small width relative to q. This allows one to ensure that $\mathbf{s}$ is recoverable with an arbitrarily high probability, while still making the problem as difficult as possible.

3.4.3 Ring-LWE Problem

Adding more structure by considering ideal lattices instead of random ones, we obtain the Ring-LWE problem, also called the RLWE problem.

Let $n \in \mathbb{N}$ be a power of two, R_q the ring $\mathbb{Z}_q[x]/(x^n + 1)$ for a positive integer q and χ_σ the corresponding (n-dimensional) error distribution on R_q. Given R_q, s, $a \in R_q$ and $e \leftarrow \chi_\sigma$, we define A_{s,χ_σ} to be the distribution of the resulting pairs $(a, as + e) \in R_q \times R_q$.

The RLWE$_{q,\sigma}$ Assumption: the assumption states that it is hard for any polynomial time algorithm with only polynomially many samples to distinguish A_{s,χ_σ} from the uniform distribution on $R_q \times R_q$.

Regev in [25] proved that there exists a probabilistic polynomial-time quantum algorithm that reduces approximate S(I)VP in the worst-case to average-case decision-RLWE. This reduction proves the hardness of RLWE.

The RLWE problem can be stated in two different ways: a "search" version and a "decision" version. Both begin with the same construction. Let:

- $a_i(x)$ be a random element but known polynomials from $\mathbb{Z}_q[x]/(\Phi(x))$ with coefficients from all of $\mathbb{Z}_q$
- $e_i(x)$ be small random element and unknown polynomials relative to a bound b in the ring $\mathbb{Z}_q[x]/(\Phi(x))$
- $s(x)$ be a small unknown polynomial relative to a bound b in the ring $\mathbb{Z}_q[x]/(\Phi(x))$.
- $b_i(x) = (a_i(x) \cdot s(x)) + e_i(x)$

The Search version entails finding the unknown polynomial $s(x)$ given the list of polynomial pairs $(a_i(x), b_i(x))$.

The Decision version of the problem can be stated as follows. Given a list of polynomial pairs $(a_i(x), b_i(x))$, determine whether the $b_i(x)$ polynomials were constructed as $b_i(x) = (a_i(x) \cdot s(x)) + e_i(x)$ or were generated randomly from $\mathbb{Z}_q[x]/(\Phi(x))$ with coefficients from all of $\mathbb{Z}_q$.

The difficulty of this problem is parametrized by the choice of the quotient polynomial $(\Phi(x))$, its degree (n), the field $(\mathbb{Z}_q)$, and the smallness bound (b). In many R-LWE based public key algorithms the private key will be a pair of small polynomials $s(x)$ and $e(x)$. The corresponding public key will be a pair of polynomials $a(x)$, selected randomly from $\mathbb{Z}_q[x]/(\Phi(x))$, and the polynomial $t(x) = (a(x) \cdot s(x)) + e(x)$. Given $a(x)$ and $t(x)$, it should be computationally infeasible to recover the polynomial $s(x)$.

```
sigma = 8 / sqrt(2 * pi)
offset = 20  # offset for plotting purposes, shift center to 20 instead of zero
size = 2^20  # count of samples
data = [0] * 50

T = RealDistribution('gaussian', sigma)

for i in range(size):
    num = int(T.get_random_element())
    data[num + offset] = data[num + offset] + 1

b = bar_chart(data)
plot(b)
```

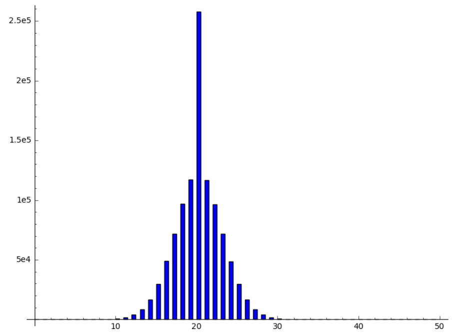


Figure 3.20: Plot of Gaussian distribution centered at 20, $\sigma = \frac{8}{\sqrt{2\pi}}$

Chapter 4

Key-exchange basics and reconciliations

In this chapter, we outline basics of Ring-LWE based key-exchange, the idea of reconciliation or key-agreement, basics of existing reconciliation methods and parameter choices to achieve a reasonable security. The goal of this chapter is to examine Ring-LWE based key-exchange in details and how it aims to be a Diffie-Hellman like replacement candidate.

4.1 Key-exchange using LWE and Ring-LWE

Basic idea of using LWE as a key-exchange mechanism is the following: we multiply a randomly generated matrix with a secret, add a noise to it (also known as LWE sample) and then send the randomly generated matrix and LWE sample to other party.

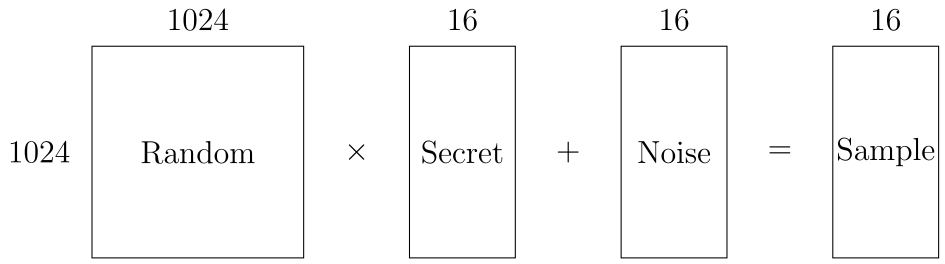


Figure 4.1: Visualization of LWE sample
Payload: random $\mathbb{Z}_{4093}^{1024\times1024}$ and sample $\mathbb{Z}_{4093}^{1024\times16}$

Thereafter, other party multiples LWE sample with his/her secret and result is the shared key. However, matrix multiplication is not commutative, we need to shuffle and transpose key and noise to achieve commutative property. In the following we see the procedure to achieve 16×16 shared key (i.e. matrix).

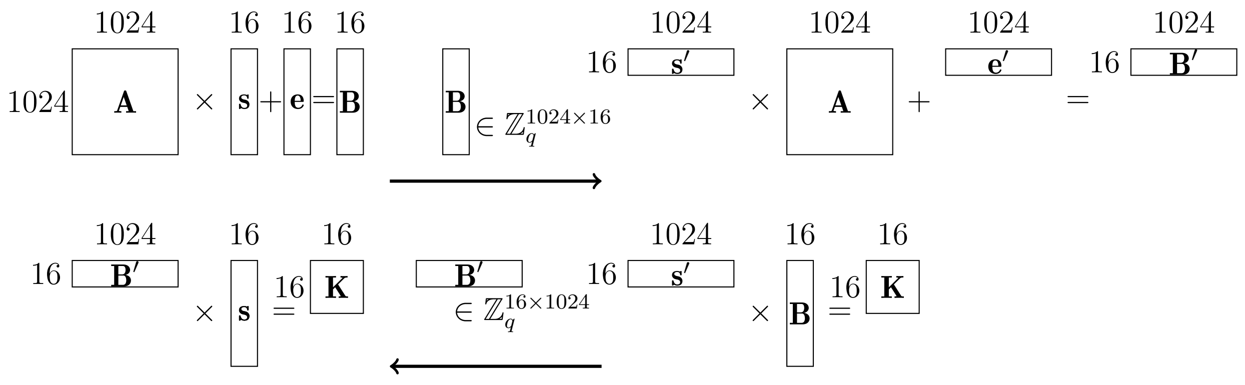


Figure 4.2: Visualization of LWE based key-exchange
Resulting shared key is a 16×16 matrix

The first issue that emerges is matrix inversion, multiplication are both costly in terms of both space and time. Another important issue is sharing a matrix $\mathbf{A}$ is costly with regards to the bandwidth needed for key-exchange. Moreover, to achieve a reasonable security we need at least 1024×1024 matrix to be shared.

$$\begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 8 & 10 \\ 1 & 3 & 3 & 2 \end{bmatrix} \text{ vs. } \left.\underbrace{\begin{bmatrix} 2738 & 3842 & 3345 & 2979 & \dots \\ 2896 & 595 & 3607 & \dots & \\ 377 & 1575 & \dots & & \\ 2760 & \dots & & & \\ \dots & & & & \end{bmatrix}}_{\text{1024 rows}}\right\} \text{1024 columns}$$

Figure 4.3: Toy example $\mathbb{Z}_{13}^{4\times4}$ vs. real-world example $\mathbb{Z}_{4093}^{1024\times1024}$ of random shared matrix
The example on the right is modulo 4093 so each matrix element in the field of $\mathbb{Z}_{4093}^{1024\times1024}$ would require at most 12 bits. Therefore,
$12 \times 1024 \times 1024 = 4291821568$ bits ≈ 1.5 Megabyte.

Hence, to overcome above issues we can use cyclic matrices to give matrix some structure and ultimately reduce the size.

Remark 4. Circulant matrices (cyclic matrix) form a commutative ring, since for any two given circulant matrices $\mathbf{A}$ and $\mathbf{B}$, the sum $\mathbf{A}+\mathbf{B}$ is circulant, the product $\mathbf{A}\times\mathbf{B}$ is circulant, and $\mathbf{A}\times\mathbf{B} = \mathbf{B}\times\mathbf{A}$

Remark 5. A polynomial in a ring $\mathbb{Z}[x]/(x^n - 1)$ can be represented as a $n \times n$ circulant matrix. That is:

$$a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} \mapsto \begin{bmatrix} a_0 & a_1 & \dots & a_{n-1} \\ a_{n-1} & a_0 & \dots & a_{n-2} \\ \dots & \dots & \dots & \dots \\ a_1 & a_2 & \dots & a_0 \end{bmatrix}$$

To solve bandwidth problem, by using cyclic matrix instead then matrix calculation would get a cumulative property for free, so no need to shuffle and transpose matrices as demonstrated previously. As a result, we do not need to send the complete matrix to other party, we can just send the first row and the other party can re-create the matrix. Using polynomials instead of matrix would address the high cost of matrix multiplication and inversion. As noted above, polynomial in $\mathbb{Z}[x]/(x^n - 1)$ can be represented as a cyclic matrix. But as noted before, conjecture is $\mathrm{SVP}_{poly(n)}$ for ideals in $\mathbb{Z}[x]/(f)$ takes time $2^{\Omega(n)}$ when f is irreducible polynomial (if f is not irreducible then multiplication operation would not be invertible in general, and hence we would not get a field and subsequently lattice would not be ideal).

Simple irreducible polynomial can be: $x^n + 1$. We can use a wrapping rule to represent a polynomial in $\mathbb{Z}[x]/(x^n + 1)$ when n is a power of 2. This would remove any pattern in cyclic matrix and subsequently makes it more complex. The wrapping rule can be: $x \mapsto -x \bmod 13$ applied to the shifted value. In other words, do a circular right shift to every row and apply above map to the value of first column of shifted row.

Below shows that by treating the first row of cyclic matrix as a polynomial in quotient polynomial ring over finite field of $\mathbb{Z}_{13}[x]/(x^4 + 1)$, polynomial can be represented as a wrapping rule described above (i.e. $\mathbb{Z}_{13}^{4\times 4}$). In details, we can construct the same matrix using the first row of matrix as a polynomial in quotient ring over finite field and then multiplying it by x^i for $0 \leq i < n$ to re-create original matrix.

$$\begin{bmatrix} 4 & 1 & 11 & 10 \\ 3 & 4 & 1 & 11 \\ 2 & 3 & 4 & 1 \\ 12 & 2 & 3 & 4 \end{bmatrix} \leftrightarrow$$

$$\begin{gathered} x^0 \times (4x^0 + 1x^1 + 11x^2 + 10x^3) = 4x^0 + 1x^1 + 11x^2 + 10x^3 \rightarrow [4, 1, 11, 10] \\ x^1 \times (4x^0 + 1x^1 + 11x^2 + 10x^3) = 3x^0 + 4x^1 + 1x^2 + 11x^3 \rightarrow [3, 4, 1, 11] \\ x^2 \times (4x^0 + 1x^1 + 11x^2 + 10x^3) = 2x^0 + 3x^1 + 4x^2 + 1x^3 \rightarrow [2, 3, 4, 1] \\ x^3 \times (4x^0 + 1x^1 + 11x^2 + 10x^3) = 12x^0 + 2x^1 + 3x^2 + 4x^3 \rightarrow [12, 2, 3, 4] \end{gathered}$$

Figure 4.4: Demonstration of simple wrapping rule to create cyclic matrix in Ring-LWE It is essentially multiplying the first row with x^i for $i \in [0, n-1]$ and extract coefficients.

The above wrapping rules helps to visualize Ring variant of LWE where irreducible polynomial is $x^n + 1$, in terms of a matrix. But polynomial multiplication is substantially more efficient in terms of both space and time. So by using Ring variant of LWE (i.e. treating A, s, e as polynomials instead of matrices), we can make key-exchange process substantially more efficient.

```
modulus = 65537                                     # prime modulus
field = GF(modulus)                                 # number field
dimension = 1024                                    # dimension (resulting matrix is square)

def simple_wrapping_rule_random_matrix(row):
    result = [row]                                  # add first row to resulting matrix
    index = 0
    for _ in range(dimension - 1):                  # iterate (dimension - 1) times
        row = result[index][:]                      # deep clone the previous row of matrix
        num = row.pop()                             # num <-- last element of previous row
        row.insert(0, modulus - num)                # add (modulus - num) to the beginning of new row

        result.append(row)                          # append row to resulting matrix

        index = index + 1                           # increment row index

    return Matrix(result)                           # return resulting matrix

def quotient_ring_over_finite_field(row):
    result = [row]
    # Quotient polynomial ring over finite field
    R.<X> = PolynomialRing(field)                   # Gaussian field of integers
    Y.<x> = R.quotient(X^(dimension) + 1)           # Cyclotomic field
    index = 0
    for index in range(1, dimension):
        result.append( (Y(row) * x^index).list() )

    return Matrix(result)                               # return resulting matrix


row = random_matrix(field, dimension, 1).list()        # create random first row of matrix

# Test if quotient ring over finite field and simple wrapping rule generate the same matrix
print simple_wrapping_rule_random_matrix(row) ==  quotient_ring_over_finite_field(row)

# Test cumulative property of cyclic matrix
a = simple_wrapping_rule_random_matrix(random_matrix(field, dimension, 1).list())
b = simple_wrapping_rule_random_matrix(random_matrix(field, dimension, 1).list())

print a * b == b * a     # returns true if cumulative property exist
```

Listing 4.1: Implementation of wrapping rule and testing if it equals to Quotient ring in SageMath

4.1.1 Reason to use a reconciliation method

Below is a Diffie-Hellman like key-exchange in Ring-LWE. Note that A is a shared polynomial, s is a secret polynomial and e is a noise polynomial. Only polynomials A and b are shared between two parties and s, e are both kept secret.

$$\begin{gathered}\text{Alice chooses small } s,\ e \text{ in } \mathbb{Z}_q\text{, both sampled from a noise distribution}\\ \text{Bob chooses small } s',\ e' \text{ in } \mathbb{Z}_q\text{, both sampled from a noise distribution}\\ \text{Alice} \rightarrow \text{Bob: } b = A \times s + e\\ \text{Bob} \rightarrow \text{Alice: } b' = A \times s' + e'\\ \text{Shared secret Alice calculates: } s \times b' = s \times (A \times s' + e') = A \times s \times s' + \boldsymbol{s \times e'}\\ \text{Shared secret Bob calculates: } s' \times b = s' \times (A \times s + e) = A \times s \times s' + \boldsymbol{s' \times e}\end{gathered}$$

Figure 4.5: Basic Ring-LWE-DH key agreement
A is similar to generator (g) and q (prime) is similar to modulo prime in Diffie-Hellman key exchange

Notice that shared keys do not match because $s \times e' \neq s' \times e$. Although errors are supposed be relatively small compared to shared polynomial A and their product with secret would be small as a result (i.e. $||s \times e'||, ||s' \times e|| \ll ||A||$), but still resulting shared key do not match and it is a problem if one wants to construct a cryptographic key-exchange protocol. To overcome this issue we use rounding methods to extract a bit from every coefficient and as added error is relatively small, hence, the coefficients would not differ significantly.

Therefore, rounding method eliminates the error and extracts the same shared key for both parties. There are two types of rounding methods (or reconciliation techniques):

1. reconciliation without any rounding information (e.g. Regev's basic rounding method)
2. reconciliation with rounding information to substantially increase the probability of extracting the same key.

The idea of sending extra bits was initially introduced by Ding and later improved by Peikert and Alkim et al.

Lemma 4.1.1. ([5], Lemma 2) LWE is no easier if the secret is drawn from the error distribution χ^n

The advantage to sample $\mathbf{s}$ and $\mathbf{e}$ from normal distribution (Gaussian) is the guarantee that $\langle \mathbf{s}, \mathbf{e} \rangle$ is small. Hence, during decryption, terms that look like $\langle \mathbf{s}, \mathbf{e} \rangle$ (inner product of the secret vector $\mathbf{s}$ and error vector $\mathbf{e}$) appears in the decryption. In Ring variant of key-exchange, we receive as shared key $s \times (As' + e') = Ass' + se' \leftrightarrow s' \times (As + e) = Ass' + s'e$. So if both parties choose small s and s' (hence their inner product with error terms would be small) and then chance of reconciliation failing to extract the same key would be reduced substantially.

4.2 Reconciliation methods

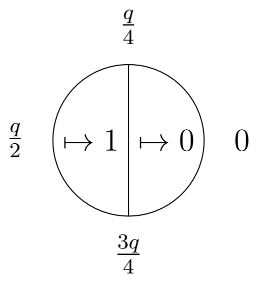
Below is a description of different reconciliation methods in Ring-LWE. These methods apply to every coefficient of resulting shared key which is a polynomial in Ring-LWE. These methods can also be applied to LWE based key-exchange as well by applying them to respective matrix elements.

4.2.1 Rounding method in Ring-LWE suggested by Regev

As each coefficient of polynomial is an integer modulo q, then we can round each coefficient to either to 0 or 1. Regev's suggested to round every coefficient if it in $(\frac{-q}{2}, \frac{-q}{4}] \cup (\frac{q}{4}, \frac{q}{2}]$ to 1, and round if it is in $(\frac{-q}{4}, \frac{q}{4}]$ to 0. This basic method works most of the time and probability of failure is $\frac{1}{2^{10}}$ which is not good enough considering that we have 1024 coefficients in our shared polynomial to achieve a reasonable security.

Alice		**Bob**
$s, e \leftarrow \chi$, $A \leftarrow$ uniformly random		$s', e' \leftarrow \chi$
	$\xrightarrow{A, b = As + e}$	
	$\xleftarrow{b' = As' + e'}$	
$K = \texttt{reconcile}(s \times b')$		$K = \texttt{reconcile}(s' \times b)$

Figure 4.6: Diagram of basic Diffie-Hellman like key-exchange protocol in Ring-LWE
Note that s, e and s', e' are all sampled from a noise distribution but A is generated uniformly at random. Also, `reconcile` function applies Regev's basic rounding method to every coefficient of resulting polynomial.



Alice's calculated shared key:
$$4079331x^0 + 1894732x^1 + \cdots + 472608x^{1022} + 516748x^{1023} = 01\ldots00$$

Bob's calculated shared key:
$$4079332x^0 + 1894733x^1 + \cdots + 472607x^{1022} + 516748x^{1023} = 01\ldots00$$

Figure 4.7: Demonstration of Regev's rounding approach
Notice coefficients of both parties are close but not equal. In Regev's rounding approach, we round each efficient to either 0 or 1 based on the region coefficient is located at.

4.2.2 Improved rounding method as suggested by Ding

Regev's method can be modified slightly by multiplying error terms by 2. The goal is to make sure difference between error terms would be even. Thereafter, we would follow the naive approach of mod 2 to eliminate the terms $2 \times s \times e'$ and $2 \times s' \times e$, hence, resulting shared key would match. However this would not result in any improvement because difference between calculated key of both parties would not always be even, since we are working in $\mathbb{Z}_q$ as oppose to $\mathbb{Z}$ and q is an odd prime. Therefore, further adjustments are needed.

Alice chooses small s, e in $\mathbb{Z}_q$, both sampled from a noise distribution
Bob chooses small s', e' in $\mathbb{Z}_q$, both sampled from a noise distribution
Alice $\rightarrow$ Bob: $b = A \times s + 2 \times e$
Bob $\rightarrow$ Alice: $b' = A \times s' + 2 \times e'$
Shared secret Alice calculates: $s \times b' = s \times (A \times s' + 2 \times e') = A \times s \times s' + 2 \times s \times e'$
Shared secret Bob calculates: $s' \times b = s' \times (A \times s + 2 \times e) = A \times s \times s' + 2 \times s' \times e$
$A \times s \times s' + 2 \times s \times e' - A \times s \times s' + 2 \times s' \times e = 2 \times (s \times e' - s' \times e) \leftarrow$ **not necessarily even difference**

Figure 4.8: Demonstration of using even error values in basic Ring-LWE-DH key agreement
For example, in $\mathbb{Z}$, $2 \times 7 = 14$ which is an even number but in $\mathbb{Z}_{13}$, it would be 1 which is an odd number.

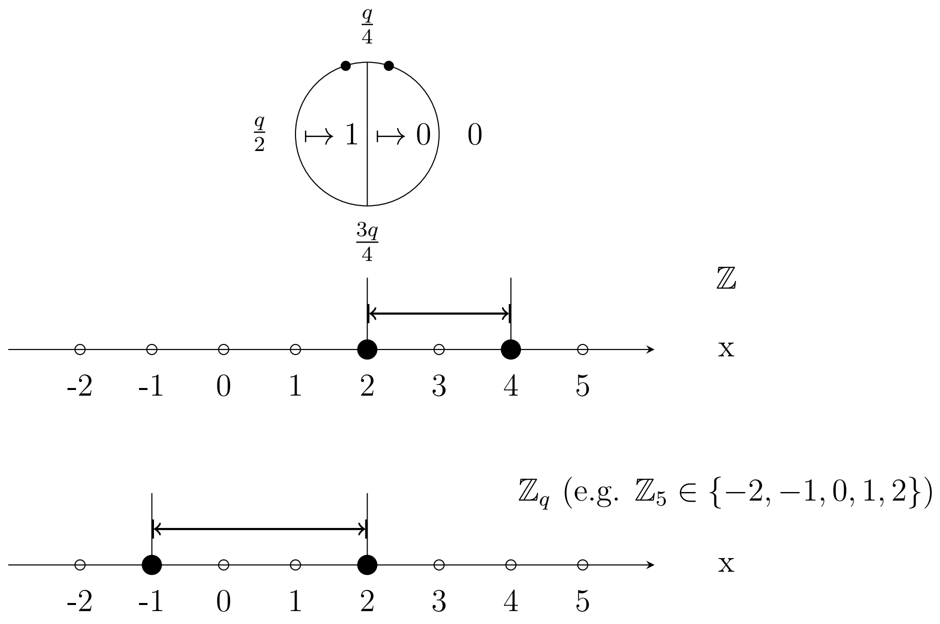


Figure 4.9: Demonstration of only multiplying errors by 2 will not help the reconciliation
In $\mathbb{Z}$ difference between two points is even but in $\mathbb{Z}_5$ difference is odd, hence no benefit in just multiplying error terms by 2. More than just multiplying error terms by 2 is needed.

Ding in [12] proposed a reconciliation method based on Ring-LWE. This protocol is the first to introduce the idea of sending extra information to improve success rate of reconciliation. Peikert improved Ding's method in [29] and presented a KEM (key-exchange mechanism). Later on Peikert's KEM was used in the construction of the BCNS protocol [7] which itself was recently improved, resulting in the NewHope protocol by Alkim et al. [4].

The basic idea of Ding's method can be seen as Diffie-Hellman like key exchange protocol based on the Ring-LWE problem. The key-exchange protocol is secure against passive adversaries if Ring-LWE is hard. For a proof and an exact definition of this security model we refer to [12]. The key exchange protocol is only proven to be secure in the two-user setting. A multi-user variant is proposed in the same paper, but its security is not yet proven. To describe Ding's reconciliation method, we define the following functions: $\delta : \mathbb{Z}_q \mapsto \{0,1\}$, $Signal : \mathbb{Z}_q \mapsto \{0,1\}$, and $Encode : \mathbb{Z}_q \times \{0,1\} \mapsto \{0,1\}$ via:

$$Signal(x) = \delta(x) \mapsto \begin{cases} 0 & \text{if } x \in [-\lfloor \frac{q}{4} \rfloor, \lfloor \frac{q}{4} \rceil] \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

$$Encode(x, \delta) = (x + \delta \times (\frac{q-1}{2}) \bmod q)(\text{mod} 2) \tag{4.2}$$

The basis idea is that if coefficient is in inner region ($\delta = 0$) , then we just $\bmod 2$ because the difference between $2 \times s \times e'$ and $2 \times s' \times e$ is even with high probability, hence in inner region the probability of achieving the same mod2 of coefficient is much higher than outer region. What we want to avoid while $\bmod 2$ is coefficient of two parties are in different regions. So, if signal bit is in outer region ($\delta = 1$), then we add $\frac{q-1}{2}$ to respective coefficient. Therefore, coefficient will now be in inner region and then mod2.

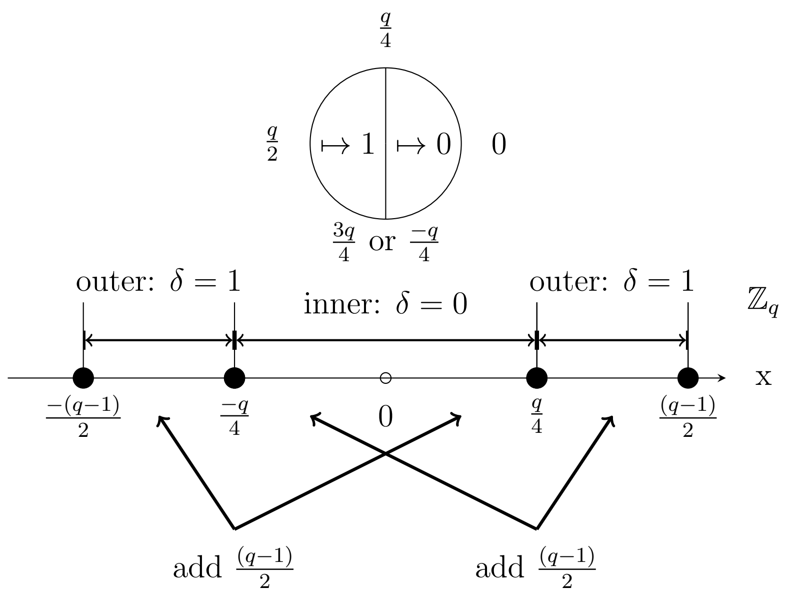


Figure 4.10: Demonstration of signal function in DXL protocol
If $\delta = 1$ or we are in outer region, then we add $\frac{q-1}{2}$ to respective coefficient, hence we will be in inner region and then mod2; if $\delta = 0$ then we just mod2.

Ding in the same paper introduced DXL key-exchange protocol. However, this protocol never got implemented in an efficient manner but it had an impact on future key-exchange protocols. For example, adding secondary noise to the calculated polynomial key to prevent secret leakage and increase entropy (additive noise).

The key exchange in this protocol will take place between two parties. There will be an initiator for the key exchange designated as (I) and a respondent designated as (R). Both I and R know q, n, $a(x)$, and have the ability to generate small polynomials according to the distribution χ_σ with parameter σ. The distribution χ_σ is a discrete Gaussian distribution. The key-exchange begins with the initiator (I) doing the following:

Initiation

1. Generate two polynomials s_I and e_I with small coefficients by sampling from the distribution χ_σ.

2. Compute $p_I = as_I + 2e_I$.

3. The initiator sends the polynomial p_I to the Responder.

Response

1. Generate two polynomials s_R and e_R with small coefficients by sampling from the distribution χ_σ
2. Compute $p_R = as_R + 2e_R$
3. Generate a small e'_R from χ_σ. Compute $k_R = p_I s_R + 2e'_R$. Then $k_R = as_I s_R + 2e_I s_R + 2e'_R$
4. Use the signal function *Signal* to find $w = Signal(k_R)$. This is computed by applying *Signal* function on each coefficient of k_R
5. Respondent side's key stream $sk_R = Encode(k_R, w)$ is calculated, based on the reconciliation information w and the polynomial k_R
6. The Respondent sends p_R and w to the Initiator

Finish

1. Receive p_R and w from the Responder
2. Sample e'_I from χ_σ and Compute $k_I = p_R s_I + 2e'_I = as_I s_R + 2e_R s_I + 2e'_I$

1. Initiator side's key stream is produced as $sk_I = Encode(k_I, w)$ from the reconciliation information w and polynomial k_I

4.2.3 Rounding method as utilized in BCNS protocol

In 2014, Peikert in [29] presented a key transport scheme following the same basic idea of Ding's, where the idea of sending additional 1 bit signal for every coefficient is also utilized. In this improved approach, Bob sends Alice a reconciliation information $\in \{0,1\}^n$ (similar to output of signal function in Ding's method) such that $0 \mapsto region\ \#1$, $1 \mapsto region\ \#2$. In other words, as a reconciliation information one party sends a region number that his or her coefficient is located in. Then based on the region, a particular key extraction rule applies. If v, u are respective coefficients of Alice and Bob, and also $|u - v| \leq \frac{q}{8}$, then this method always works. Due to the clever design of rounding regions and key extractions rules, revealing the region leaks no information and does not compromises security.

We should note that BCNS protocol is an implementation of Peikert's reconciliation method and this protocol does not introduce a new reconciliation method.

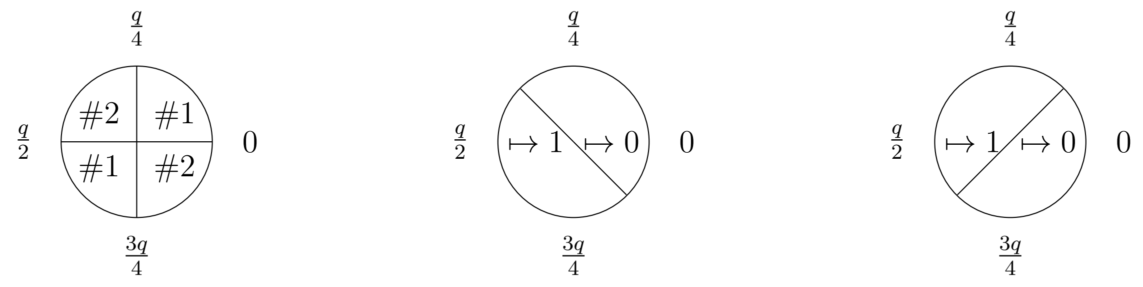


Figure 4.11: Demonstration of Peikert's rounding approach
(a) demonstrates the 4 regions and (b) is a rounding rule when rounding information bit indicates regions #1 and (c) if it indicates region #2. Notice if difference of coefficients is $\leq \frac{q}{8}$ then this method always works.

Alice chooses *small* s, e in $\mathbb{Z}_q$, both sampled from a noise distribution
Bob chooses *small* s', e' in $\mathbb{Z}_q$, both sampled from a noise distribution
Alice $\rightarrow$ Bob: $b = A \times s + e$
Bob $\rightarrow$ Alice: $b' = A \times s' + e'$ and **rounding region** $\in \{0,1\}^n$
Shared secret Alice calculates: $s \times b' = s \times (A \times s' + e') = s \times A \times s' + s \times e'$
Shared secret Bob calculates: $s' \times b = s' \times (A \times s + e) = s' \times A \times s + s' \times e$

Figure 4.12: Exact Ring-LWE-DH key agreement as suggested by Peikert.
Notice Bob also sends his rounding regions to Alice which is similar to Ding's rounding method

The following is the concrete description of Peikert's reconciliation method. We define the reconciliation mechanism where the modulus $q \geq 2$ is even, then define disjoint intervals $I_0 = \{0, 1, \ldots, \lfloor \frac{q}{4} \rceil - 1\}$, $I_1 = \{\lfloor \frac{3q}{4} \rceil, \ldots, q-1\} \bmod q$. Now define the *cross-rounding* function $\langle \cdot \rangle_2 : \mathbb{Z}_q \rightarrow \mathbb{Z}_2$ as:

$$\langle v \rangle_{q,2} = \begin{cases} 0 & v \in I_0 \cup (I_0 + \frac{q}{2}) \\ 1 & v \in I_1 \cup (I_1 + \frac{q}{2}) \end{cases}$$

To compute the shared key the following reconciliation function rec : $\mathbb{Z}_q \times \mathbb{Z}_2 \to \mathbb{Z}_2$ is used:

$$\text{rec}(v, b) = \begin{cases} 0 & \text{if } v \in I_b + ([-\frac{q}{8}, \frac{q}{8}) \cap \mathbb{Z}) \bmod q \\ 1 & \text{otherwise} \end{cases}$$

All of the above applies when q is even, but in applications of Ring-LWE this is not the case. For instance, it is often desirable to let q be a sufficiently large prime, for efficiency and security reasons. When q is odd, while it is possible to use the above methods to agree on a bit derived by rounding a uniform $v \in \mathbb{Z}_q$, the bit will be biased, and hence not suitable as key material. Here we show how to avoid such bias by temporarily *scaling up* to work modulo $2q$, and introducing a small amount of extra randomness.

Define the randomized function dbl : $\mathbb{Z}_q \to \mathbb{Z}_{2q}$ that, given a $v \in \mathbb{Z}_q$, outputs $\bar{v} = 2v - \bar{e} \in \mathbb{Z}_{2q}$ for some random $\bar{e} \in \mathbb{Z}$ that is uniformly random modulo two and independent of v, and small in magnitude (e.g., bounded by one).

$$\text{dbl} : \mathbb{Z}_q \to \mathbb{Z}_{2q}, x \mapsto 2x - \bar{e}, \text{ where } \bar{e} = \begin{cases} -1 & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ 1 & \text{with probability } \frac{1}{4} \end{cases}$$

Moreover, if $w, v \in \mathbb{Z}_q$ are close, then so are $2w, \text{dbl}(v) \in \mathbb{Z}_{2q}$, i.e., if $w = v + e (\text{mod} q)$ for some (small) e, thus $2e$ would be small too, then $2w = \bar{v} + (2e + \bar{e})(\text{mod} 2q)$. Therefore, to (cross-) round from $\mathbb{Z}_q$ to $\mathbb{Z}_2$, we simply apply dbl to the argument and then apply the appropriate rounding function from $\mathbb{Z}_{2q}$ to $\mathbb{Z}_2$. Similarly, to reconcile some $w \in \mathbb{Z}_q$ we apply rec to $2w \in \mathbb{Z}_{2q}$; note that this process is still deterministic.

To summarize, The usual definition of (passive) security for key-exchange requires the agreed-upon key to be indistinguishable from uniformly random. That is not the case for DXL because the bits of the key are biased, not uniform. It is simply because any deterministic map from $\mathbb{Z}_q$ to $0, 1$ must be biased when q is odd. To address this issue, Peikert suggested if the modulus q is odd, it requires to work in $\mathbb{Z}_{2q}$ instead of $\mathbb{Z}_q$ to avoid bias in the derived bits.

Since q is odd in practice, we need use randomized doubling function (dbl). The following lemma shows that the rounding of $\text{dbl}(v) \in \mathbb{Z}_{2q}$ for a uniform random element $v \in \mathbb{Z}_q$ is uniform random in $\mathbb{Z}_{2q}$ given its *cross-rounding*.

Lemma 4.2.1. ([29], Claim 3.3) For odd q, if $v \in \mathbb{Z}_q$ is uniformly random and $\bar{v} \leftarrow \text{dbl}(v) \in \mathbb{Z}_{2q}$, then $\text{rec}(\bar{v})$ is uniformly random given $\langle \bar{v} \rangle_{2q,2}$

4.2.4 Rounding method as utilized in NewHope protocol

In November 2015, Alkim, Ducas, Popplemann, and Schwabe in [4] built a reconciliation scheme based on the work of Peikert and further improved Peikert's randomized doubling function. In addition, Alkim et al. implemented their new reconciliation method as a protocol and named it NewHope. Unlike BCNS protocol, NewHope protocol provides a new reconciliation methods and it is as follows: the sender and the receiver have two almost identical vectors $v_S \approx v_R \in \mathbb{Z}_q^n$. They want to obtain one shared secret key $SK \in \{0, 1\}^{\frac{n}{4}}$ from those two vectors, i.e., the sender and the receiver want to obtain one bit of the key from each four coordinates. Deciding the value of this key bit is done geometrically. In the following, by Voronoi cell we are referring to Polyhedron generated by Diamond cutting algorithm as described in [37] given full rank lattice with basis of $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \end{pmatrix}$, that is identity matrix such that last row is set to $\frac{1}{2}$.

```
from sage.modules.diamond_cutting import calculate_voronoi_cell

sub_dimension = 4 # reconciliation sub dimension

identity_matrix = Matrix.identity(RR, sub_dimension)    # create identity matrix
```

```
half_vector = [1/2 for i in range(sub_dimension)]       # construct 1/2 vector
identity_matrix[sub_dimension - 1] = half_vector        # modify last row of identity matrix

calculate_voronoi_cell(identity_matrix).translation(half_vector).show()  # create voronoi cell
```

Listing 4.2: Polyhedron generated by Diamond cutting algorithm

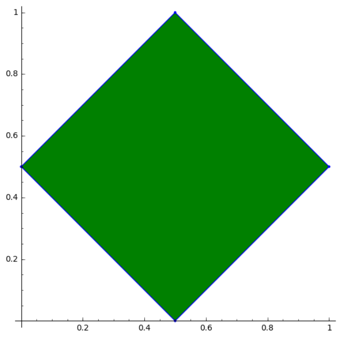


Figure 4.13: Voronoi cell in 2-dimension

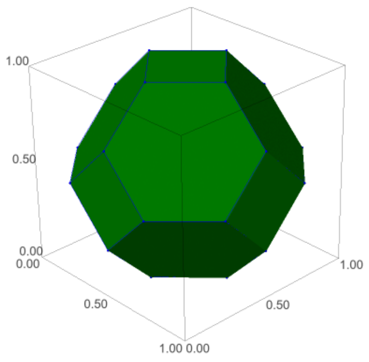


Figure 4.14: Voronoi cell in 3-dimension
This is not being used in NewHope, only for the purpose of visualization

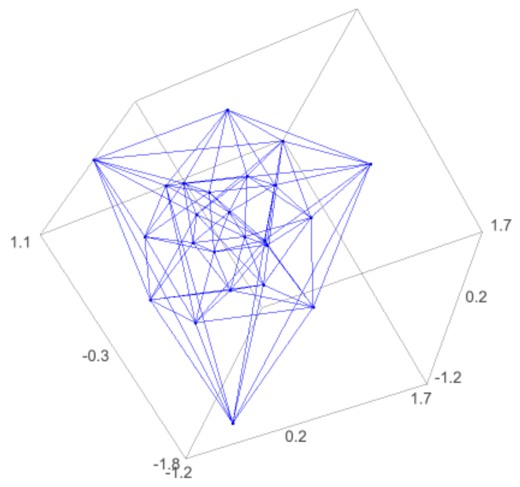


Figure 4.15: Voronoi cell in 4-dimension, (diamond inside a cube)

The essence of this new reconciliation can be summarized as the following:

1. Divide every coefficient of calculated key, that is: $s' \times (As + e)$ or $s \times (As' + e')$ by q. Therefore we a get a list of numbers that are between $[0, 1)$ inclusive
2. Pairwise select every 4 coefficient $(c_i, c_{i+1}, c_{i+2}, c_{i+3})$ and run the $p_i = \mathrm{CVP}_4(c_i, c_{i+1}, c_{i+2}, c_{i+3})$ to get the closest center of Voronoi cell, then store the result into array $[p_1, \ldots, p_{\frac{n}{4}}]$
3. For all p_i in the array, calculate the distance between $(c_i, c_{i+1}, c_{i+2}, c_{i+3})$ and p_i and store the result into another array $[d_i, \ldots, d_{\frac{n}{4}}]$
4. Send the array $[d_i, \ldots, d_{\frac{n}{4}}]$ to other party. Note that similar to Ding and Peikert's method, only one party sends the reconciliation information
5. For all d_i in the array, both parties will then add the d_i to their $(c_i, c_{i+1}, c_{i+2}, c_{i+3})$, so vectors constructed by coefficients of both parties will gets closer to center of closest Voronoi cell. As a result, we will achieve *exact* key agreement
6. If an adjusted vector generated by 4 coefficients is in center Voronoi cell then bit is 1; else bit is 0

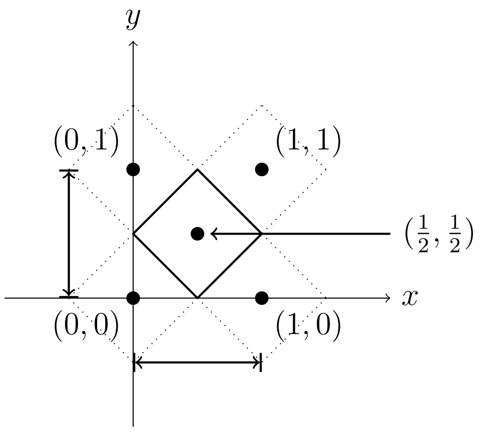


Figure 4.16: 2-dimension Voronoi cell centered at $(\frac{1}{2}, \frac{1}{2})$

Remark 6. The valid Voronoi cell (or Polyhedron generated by Diamond cutting algorithm) should have a volume (or area in 2-dimensions) of $\frac{1}{2}$ [37].

Above implies that probability of $0, 1$ bits are both equal to $\frac{1}{2}$ in reconciled shared key. Since probability of vector to be inside or outside main Voronoi cell are equal, hence, there is no bias in generated key and it is uniformly distributed.

Below is a simple yet efficient procedure to check if vector generated by coefficients is in main Voronoi cell or otherwise. Subsequently, procedure finds the distance between coefficient vector and center of closest Voronoi cell. The procedure below is for 2-dimensions, however, for 4-dimensions it would be similar but 24 inequalities to check instead of 4 inequalities. To extract a key from a vector generated by coefficients, it would be similar but returning 1 if vector is in main Voronoi cell and 0 otherwise.

Algorithm 2 get_distance_voronoi_cell

```
procedure get_distance_voronoi_cell($\mathbf{v}$)
    if $(2.0, 2.0) \cdot \mathbf{v} - 1.0 \geq 0$
        and $(2.0, -2.0) \cdot \mathbf{v} + 1.0 \geq 0$
        and $(-1.0, -1.0) \cdot \mathbf{v} + 1.5 \geq 0$
        and $(-2.0, 2.0) \cdot \mathbf{v} + 1.0 \geq 0$
            return $(0.5, 0.5) - \mathbf{v}$
    else
            return round($\mathbf{v}$) - $\mathbf{v}$
```

Figure 4.17: Finding distance between vector and center of closest Voronoi cell
Note that above multiplications with vector `v` are dot product or scalar product. Also, `round` function, rounds x, y components of vector `v` to nearest integer. As x, y are in range $0 \leq x, y < 1$ then result of `round` function would be $\in \{(0,0),(0,1),(1,0),(1,1)\}$

The idea that one party as a reconciliation information has to send array of (double precision) floating point numbers is not efficient. To resolve the issue Alkim at al. introduced the idea of splitting each Voronoi cells into 2^{rd} =16 sub-cells. Then instead of calculating the distance between vector generated by coefficients and center of closest Voronoi cell, we can find the closest center of Voronoi sub-cell (i.e. sub-cell number that coefficient is located at) and send the sub-cell number to other party as a reconciliation information. Then both parties will add the distance between lattice point and center of sub-cell to the vector generated by their coefficients; hence they shift the vector generated by their coefficients toward center of their Voronoi cell. Therefore, instead of sending array of distances which are (double precision) floating point numbers, we can send the sub-cell numbers instead which are in integers. Note that we send the sub-cell number and not the outer Voronoi cell number to other party. As a result, eavesdropper does not learn anything from reconciliation information.

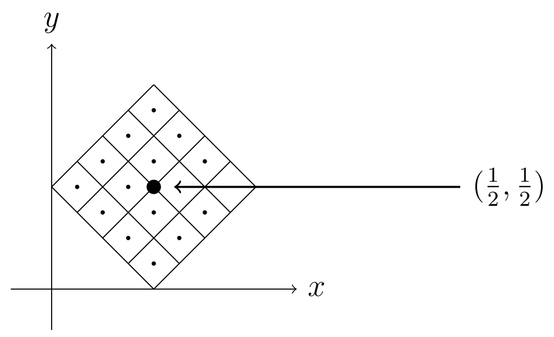


Figure 4.18: 2-dimension Voronoi cell centered at $(\frac{1}{2}, \frac{1}{2})$ split into 16 sub-cells

For details on how NewHope reconciliation methods splits Voronoi cell into sub-cells and efficiently finds the sub-cells number in which vector generated by coefficients is located at, translating sub-cell number into Voronoi cell and subsequently extracting a bit form every 4 coefficients refer to NewHope paper [4].

The following is the concrete description of NewHope reconciliation method. Let $\mathbf{B} = (e_1, e_2, e_3, g)$ where $\mathbf{g} = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})^T$ and e_i is the i-th vector of unity. Moreover, let D_4 be the lattice defined by $\mathbf{B}$. The 4-dimensional space is divided into infinitely many four-dimensional cells, one cell for each lattice vector $\mathbf{v}$. In the following we only look at four coordinates of the original vector $\mathbf{v}_R \in \mathbb{Z}_q^4$, called $\mathbf{v}_R^{(4)} \in \mathbb{Z}_q^4$. The coordinates of $\mathbf{v}_R^{(4)}$ are divided by q to obtain a vector $x \in \mathbb{R}^4/\mathbb{Z}^4$. In modulo $\mathbb{Z}^4$ there are only two cells in $\mathbb{R}^4$, the one corresponding to the lattice vector $\mathbf{g}$ and the one corresponding to the lattice vector $(0, 0, 0, 0)^T$. If a vector $\mathbf{x}$ lies in the cell which belongs to $\mathbf{g}$, it is mapped to 1, otherwise it is mapped to 0. Determining the correct cell can be done by using algorithm CVP_{D_4}. Since the sender and the receiver do not get exactly the same vectors $\mathbf{v}_S^{(4)} \approx \mathbf{v}_R^{(4)}$ that they want to encode in the way described above, they need to be sure that their vectors lie in the same cell. This however is not necessarily true. Hence, they need to exchange some reconciliation information $\mathbf{c}^{(4)}$ which defines how the vector $\mathbf{v}_S^{(4)}$ has to be moved to assure that it is in the same cell as $\mathbf{v}_R^{(4)}$. For this we divide each cell into 2^{4r} sub-cells (if not stated differently, we assume $r = 2$). The receiver assigns the number of the sub-cell of $\mathbf{v}_R^{(4)}$ to $\mathbf{c}^{(4)} \in \{0, 1, 2, 3\}^4$ and sends it to the sender. Afterwards, the sender computes the difference vector of the middle of a cell and the middle of the sub-cell corresponding to $\mathbf{c}$. Then the sender runs the CVP_{D_4} algorithm.

If $\mathbf{v}_S^{(4)} \in \mathbb{Z}_q^4$ and $\mathbf{c}^{(4)} \in \{0,1,2,3\}^4$ then

$$\mathrm{Rec}(\mathbf{v}_S^{(4)}, \mathbf{c}^{(4)}) = \mathrm{Decode}(\frac{1}{q}\mathbf{v}_S^{(4)} - \frac{1}{2^r}\mathbf{B}\mathbf{c}^{(4)}) \tag{4.3}$$

computes one bit of the key from four coefficients.

The function $\mathrm{HelpRec}(\mathbf{v}_R)$ in the protocol of the receiver is defined as $\mathrm{HelpRec}(\mathbf{v}_R^{(4)}) = \mathrm{CVP}_{D_4}(\frac{2^r}{q}(\mathbf{v}_R^{(4)}) + b\mathbf{g}) \bmod 2^r$, where $b \leftarrow \{0,1\}$.

Algorithm 3 $\mathrm{CVP}_{D_4}(\mathbf{x} \in \mathbb{R}^4)$

```
1: procedure CVP($\mathbf{x}$)
2:     $\mathbf{v}_0 \leftarrow \lfloor \mathbf{x} \rceil$
3:     $\mathbf{v}_1 \leftarrow \lfloor \mathbf{x} - \mathbf{g} \rceil$
4:     $k \leftarrow 0$ if $||\mathbf{v} - \mathbf{v}_0||_1 < 1$ else $1$
5:     $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)^t \leftarrow \mathbf{v}_k$
6:     return: $(\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, k)^t + \mathbf{v}_3 \cdot (-1,-1,-1,2)^t$
```

Algorithm 4 Decode $(x \in \mathbb{R}^4/\mathbb{Z}^4)$

```
1: procedure Decode(x)
2:     $\mathbf{v} \leftarrow \mathbf{x} - \lfloor \mathbf{x} \rceil$
3:     $r \leftarrow 0$ if $||\mathbf{v}||_1 \leq 1$ else $1$
4:     return: $r$
```

Furthermore, NewHope protocol generates the shared polynomial pseudo-randomly on every run of the KEM to assure safety against backdoors. Next chapter discusses the method as suggest in NewHope protocol to efficiently share the shared polynomial with other party. In essence, to generate a pseudo-random polynomial A, NewHope uses a 256-bit seed and a type of hash-function that outputs arbitrary length, e.g., SHAKE-128. This can be done similarly for all the other protocols.

4.2.4.1 Generalized form of randomized doubling function suggested by Alkim et al.

The probability of $\mathbf{x}$ being in a center Voronoi cell is the same as for $\mathbf{x}$ being in other Voronoi cells. This would be the case if $\mathbf{x}$ actually followed a continuous uniform distribution. However, the coefficients of $\mathbf{x}$ are discrete values in $\{0, \frac{1}{q}, \ldots, \frac{q-1}{q}\}$ and with the protocol described so far, the bits of $\mathbf{v}$ would have a small bias. The solution is to add to $\mathbf{x}$ with probability $\frac{1}{2}$ the vector $(\frac{1}{2q}, \ldots, \frac{1}{2q})$ before running the error reconciliation. This has close to no effect for most values of $\mathbf{x}$, but, with probability $\frac{1}{2}$ moves $\mathbf{x}$ to another Voronoi cell if it is very close to one side of a border.

In the example below we see that 72 dots (36 red and 36 black ones) remain in their Voronoi cell; the other 9 dots (4 red and 5 black ones) change Voronoi cell with probability $\frac{1}{2}$ which precisely eliminates the bias in the key.

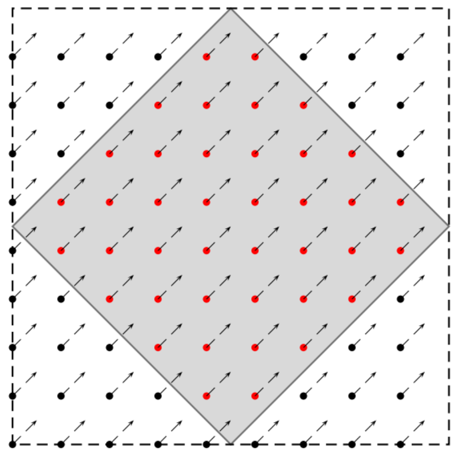


Figure 4.19: Effect of generalized form of randomized doubling function on vectors

4.3 BCNS and NewHope diagram comparison

The basic sketch of both BCNS and NewHope protocols are very similar to Ding's key-exchange protocol (DXL). They all share the basics, initiator sends a polynomial b to responder and responder replies by sending b', c where c is a reconciliation information or a bit string to improve success probability of key-agreement. For every coefficient of responder's calculated key, there is one reconciliation bit and there are at most 1024 coefficients. Therefore, responder is sending at most 1024 bits as a reconciliation information.

$a \leftarrow$ Uniformly random		
Alice		**Bob**
$s, e \xleftarrow{\$} \chi$		$s', e' \xleftarrow{\$} \chi$
$b \leftarrow as + e$	$\xrightarrow{b}$	$b' \leftarrow as' + e'$
		$e'' \xleftarrow{\$} \chi$
		$v \leftarrow bs' + e''$
	$\xleftarrow{b',c}$	$\bar{v} \xleftarrow{\$} \text{dbl(v)}$
		$c \leftarrow \langle \bar{v} \rangle_{2q,2} \in \{0,1\}^n$
$k_A \leftarrow \text{rec}(2b's, c) \in \{0,1\}^n$		$k_B \leftarrow \text{rec}(2bs', c) \in \{0,1\}^n$

Figure 4.20: Diagram of BCNS protocol

In NewHope protocol however, initiator also sends a *seed* derived from a uniformly random distribution that enables the other party to deterministically create a shared polynomial by themselves without actually sending it. This is results in an efficiency in bandwidth needed for key-exchange. Also, to make sure the resulting shared key is uniformly random or number of 0's and 1's are uniformly distributed, NewHope uses SHA-3. Note that if number of 0's and 1's are always exactly equal then we can deterministically find the shared key. Using a hash function will not have any impact on hardness of Ring-LWE, it is just matter of making bits of resulting shared key uniformly distributed and making a sure it is irreversible (i.e. eavesdropper needs to break SHA-3 first and then would be able to break Ring-LWE).

Another important observation is reconciliation information of NewHope is slightly different from BCNS protocol. In NewHope, one party sends a sub-cell number calculated from coordinate tuple generated by every 4 coefficients to other party. Further, NewHope protocol divides each Voronoi cell to 16 sub-cells so for every 4 coefficient it requires up-to 4 bits. Considering that there are total of up-to 1024 coefficients in responder's key then responder is in fact sending $256 \times 4 = 1024$ bits as a reconciliation information, which is the same number of bits as in BCNS protocol.

$q = 12289 < 2^{14}, n = 1024, \Psi_{16}$		
Alice		**Bob**
$seed \overset{\$}{\leftarrow} \{0,1\}^{256}$		
$a \leftarrow \text{parse}(\text{SHAKE-128}(seed))$		
$s, e \overset{\$}{\leftarrow} \Psi_{16}^n$		$s', e', e'' \overset{\$}{\leftarrow} \Psi_{16}^n$
$b \leftarrow as + e$	$\xrightarrow{(b, seed)}$	$a \leftarrow \text{parse}(\text{SHAKE-128}(seed))$
		$u \leftarrow as' + e'$
		$v \leftarrow bs' + e''$
$v' \leftarrow us$	$\xleftarrow{(u, r)}$	$r \overset{\$}{\leftarrow} \text{HelpRec}(v)$
$v \leftarrow \text{Rec}(v', r)$		$v \leftarrow \text{Rec}(v, r)$
$\mu \leftarrow \text{SHA3-256}(v)$		$\mu \leftarrow \text{SHA3-256}(v)$

Figure 4.21: Diagram of NewHope protocol

With regards to e'' added to Bob's calculated key and the reason for it, notice if e'' is not there, then $v = bs'$ in both protocols, which means it would be easy to recover s' given v and b. Since b is sent in the clear over the channel, and a (randomized) function of v appears in the clear as c (or r), without using e'' to hide s', information about s' would likely be leaked both to Alice and to any eavesdropper of the channel. Even if it would not immediately leak all of s', any leakage is clearly bad. As a result, e'' is used to make sure that $v = bs' + e''$ is indistinguishable from random, i.e. the distribution of v is independent of s', assuming Ring-LWE is hard. See also the security proof in the BCNS paper [7], where Game 1 and Game 2 are assumed to be indistinguishable under the Ring-LWE assumption. The reason Alice does not add some e'' as well is because Alice uses e to hide her own secret s, just like Bob uses e' to hide his secret s'. To agree on the key (both parties have different noise on the shared key, which they do not wish to disclose). Bob has to send this extra key reconciliation message, for which he again hides his secret s' with fresh random noise e''.

4.4 Parameter choices for Ring-LWE key-exchange

The Ring-LWE key-exchange presented above worked in the Ring of polynomials of degree $n - 1$ or less mod a polynomial $\Phi(x)$. Note that n is a power of 2 and q is a prime which is congruent to 1 mod $2n$. Following the guidance given in Peikert's paper [29], V. Singh in [35] suggested two sets of parameters for the RLWE-KEX and in both coefficient of error polynomial is sampled from normal distribution with standard deviation $\sigma = \frac{8}{\sqrt{2\pi}}$. Suggested parameters are as follows:

1. For 128 bits of security, $n = 512$, $q = 25601$, and $\Phi(x) = x^{512} + 1$

2. For 256 bits of security, $n = 1024$, $q = 40961$, and $\Phi(x) = x^{1024} + 1$

Because RLWE based key exchange uses random sampling and fixed bounds, there is a small probability that the key-exchange will fail to produce the same key for the initiator and responder. Given Gaussian parameter σ equals to $\frac{8}{\sqrt{2\pi}} = 3.192$, Singh in [35] calculated that probability of key agreement failure to be less than 2^{-71} for the 128-bit secure parameters and less than 2^{-91} for the 256-bit secure parameters. BCNS protocol kept the σ and Φ as suggested by Singh for 256 bits of security, but changed the prime modulus of finite field to $2^{32} - 1$. This resulted in failure probability of $2^{-131072}$.

Originally (in early versions of protocol), NewHope kept the Φ as suggested by Singh, but used 12-bit binomial distribution instead with $\sigma = \sqrt{\frac{12}{2}} = 2.449$ and changed the prime modulus of finite field to 12289. This represents a 70% reduction in public key size over the parameters of Singh. Hence, reducing the prime modulus size resulted in failure probability of 2^{-105}. But in later version (and after discussion with A. Langley from Google), Alkim et al. used 16-bit binomial distribution with $\sigma = \sqrt{\frac{16}{2}} = 2.828$ which resulted in failure probability of 2^{-60}. In fact, this later parameter was implemented in Google's Chrome Canary project as well as Boring-SSL which is a fork of Open-SSL that is designed to meet Google's needs [8].

Remark 7. Notice the smaller q results in faster key-exchange by reducing over-head size and increase in security, and most importantly increasing impact of error terms, but this yields higher failure probability.

n	q	Distribution	Parameter	Security Claim	Rec. Algorithm	Citation
512	1051649	Gaussian	$3.19/\sqrt{2\pi}$	2^{128}	DXL	[16]
820	49261	Gaussian	$8/\sqrt{2\pi}$	2^{256}	Peikert	[36]
1024	12289	Binomial	$\sqrt{16/2}$	2^{128}	NewHope	[4]
1024	40961	Gaussian	$8/\sqrt{2\pi}$	2^{256}	Peikert	[36]
1024	$2^{32}-1$	Gaussian	$8/\sqrt{2\pi}$	2^{128}	Peikert	[7]

Table 4.1: Listing of a number of different parameter choices for KEXs using the Ring Learning with Errors problem

4.5 Lattice based authenticated key-exchange

Key-exchange protocol (KEX) is a cryptographic primitive to derive a common secret key via a public network communication between a number of parties. The parties do not share any secret information beforehand. KEX that also authenticates the identities of the involved parties is called authenticated key-exchange protocol (AKE). More formally, in an AKE protocol each party has a static public-secret key pair. The static public key is certified with the party's identity. When running the protocol, each party generates an ephemeral secret key and, depending on that, a corresponding ephemeral public key. The public key is sent to the other party. Each party computes a shared session key by using the ephemeral key and the static key (e.g. password).

An authenticated key-exchange protocol is said to have perfect forward secrecy (PFS) if a compromise of its static keys does not lead to a compromise of previously established and deleted session keys. There are many existing authenticated key-exchange approaches built for Diffie-Hellman, however they all can be used with Ring-LWE instead because of similarities in design. Authenticated key-exchange aims to overcome man-in-the-middle attack, which is an attack where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other. The following are basic authenticated Diffie-Hellman approaches that can be applied to RLWE-KEX:

1. Use key-exchange in conjunction with a signing certificate (use either RSA or DSA as a public key). The server sends its certificate which client validates and ephemeral key signed (or encrypted) using server's private key. Client validates the ephemeral key using certificate and encrypts the session using ephemeral key.

2. Concatenate (or pad) ephemeral key (or result of unauthenticated key-exchange) with a password and use the hash of that as a session key to achieve a secure communication

3. Encrypt $b = As + e$ (in Ring-LWE) or $K = g^a \bmod p$ (in Diffie-Hellman) using a password as an encryption key before sending it to other party.

Chapter 5

Implementation specifications

In this chapter, we overview specifics of Ring-LWE based protocols which includes: methods of error sampling, methods to generate shared polynomial between two parties and performance analysis. The goal of this chapter is to explain in details how key-exchange reconciliations that was discussed in previous chapter was implemented as a protocol.

5.1 Error sampling algorithm

Lattice based cryptography began with the seminal work of Ajtai, who built a one-way function based on the worst-case hardness based on certain lattice problems [1]. These lattice problems are believed to be hard even in the presence of large quantum computers and such a promising post-quantum replacement for standard cryptography. The most general public key primitives like encryption schemes [25] and digital signatures [23] already have practical lattice based instantiations.

Many recent lattice based schemes require sampling from discrete Gaussian. The parameters of discrete Gaussian are governed by the security proofs of the particular schemes. A finite machine cannot sample from a discrete Gaussian distribution, hence one has to sample from a distribution close to it. It is a common practice to require that the statistical distance of the sampled distribution from the desired discrete Gaussian be less than 2^{-100}.

Computing the probabilities requires floating point operations of at least 100 bit precision if one wants to achieve a statistical distance less than 2^{-100} [19]. Whereas any precomputation means storing a variable amount of values of the same precision. This can highly affect the sampling performance on personal computers and even make the implementation completely impractical on constrained devices. Weiden et al. [39] report that the Gaussian sampling takes up 50% of the running time of Lyubashevsky's signature scheme (also known as BLISS) [23]. Thus efficient sampling from discrete Gaussians plays a crucial role in the performance of these primitives.

S. Galbraith in [13] discussed different discrete Gaussian sampling algorithms suitable for constrained devices. By a constrained device we can think of an embedded or portable device with a small amount of memory (measured in kilobytes instead of gigabytes) and a modest processor that has to be economical with respect to power usage. Also these kind of devices do not necessarily come with floating point arithmetic capability. Even if a platform provides floating point arithmetic, the required precision is usually not supported natively. This means that software libraries have to be used for this functionality, and these can significantly worsen performance and take up additional space in the already tight memory.

A particular discrete Gaussian samplers may apply different techniques to increase the performance and reduce or even avoid the floating point operations, which usually utilize precomputed tables, hence not requiring floating point arithmetic. Many factors can affect the performance and memory consumption (i.e., the size and number of the potential precomputed tables). Such factors are the size of the Gaussian parameter, whether the center is zero or not, and whether the parameters are fixed or changing (or more precisely, the number of the needed parameter combinations). To evaluate the practicality of the discrete Gaussian samplers in lattice based cryptography one needs to assess the parameters of the distributions required by the different cryptographic schemes.

The techniques utilized by different samplers require various amount of memory and floating point operations, which results in different overall performance on the particular platforms. Thus for the evaluation of their practical performance one needs to collect the characteristics of the discrete Gaussian samplers too.

In the following by PDF and CDF we are referring to *Probability Density Function* and *Cumulative Distribution Function.* Note that the relation between the probability density function f and the cumulative distribution function F is $F(k) = \sum_{i \le k} f(i)$ if f is discrete and $F(x) = \int_{y \le x} f(y)\, dy$ if f is continuous.

Definition 20. The probability density of the normal distribution is: $f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Where:

- μ is mean or expectation of the distribution (center of PDF)
- σ is standard deviation
- σ^2 is variance

Remark 8. Coefficients of polynomials in Ring-LWE are $\in \mathbb{Z}$. Therefore, error sampling methods should sample from *Discrete Distributions* as oppose to *Continuous Distributions.*

5.1.1 Inversion sampling

Inverse transform sampling (also known as Smirnov transform) is a basic method for pseudo-random number sampling, i.e. for generating sample numbers at random from any probability distribution given its cumulative distribution function.

Inverse transformation sampling takes uniform samples of a number u between 0 and 1, interpreted as a probability, and then returns the largest number x from the domain of the distribution $P(X)$ such that $P(-\infty < X < x) \le u$.

In this method, we are randomly choosing a proportion of the area under the curve and returning the number in the domain such that exactly this proportion of the area occurs to the left of that number. Intuitively, we are unlikely to choose a number in the far end of tails because there is very little area in them which would require choosing a number very close to zero or one.

Computationally, this method involves computing the quantile function of the distribution; in other words, computing the cumulative distribution function (CDF) of the distribution (which maps a number in the domain to a probability between 0 and 1) and then inverting that function. This is the source of the term "inverse" or "inversion" in most of the names for this method. Note that for a discrete distribution, computing the CDF is not in general too difficult: we simply add up the individual probabilities for the various points of the distribution. For a continuous distribution, however, we need to integrate the probability density function (PDF) of the distribution, which is impossible to do analytically for most distributions including the normal distribution. As a result, this method may be computationally inefficient for many distributions and other methods are preferred.

For the normal distribution, the lack of an analytical expression for the corresponding quantile function means that other methods may be preferred computationally. It is often the case that, even for simple distributions, the inverse transform sampling method can be improved on, for example, the ziggurat algorithm and rejection sampling [19].

5.1.1.1 Naive sampling approach using PDF function

One simple approach which generates only positive integer samples is calculating probabilities using $\text{PDF}(x)\ \forall x \in [0, \text{variance}]$ and putting them in an array. Note that, sum of probability array would not be equal to one because we ignored the probabilities of negative values. Thereafter, we need to normalize the probability array such that sum would equal to 1. Then, create a new array with the same size as probability array but setting first element of new array as first element of probability array. Thereafter, we set new array at any index to sum of probability array until that index. We can call the probability array, vector of PDF values and also call second array, vector of CDF values. Note that value of last element of CDF vector should equal to 1 because we normalized the probability list and sum of probability list should equal to 1.

To sample integers in range $[0, \text{variance}]$ using this method, simply generate a random number between $[0,1)$ uniformly at random, lets call it u. Then find a index such that $u <$ CDF[*index*]. The value of *index* is a sample from the distribution.

Below is an implementation of the naive Gaussian sampler that we described above; centered at 0 with $\sigma = \frac{8}{\sqrt{2\pi}}$ as suggested in early draft of BCNS protocol.

Algorithm 5 naive Gaussian sampler by estimating cdf values

```
1: sigma ← $8/\sqrt{2\pi}$
2: variance ← sigma$^2$
3: mu ← 0
4: domain ← ⌈variance⌉
                                        ▷ probability density of the normal distribution
5: function pdf(x)
6:     return: $(1/\sqrt{2\pi \times \text{variance}}) \times e^{-(x-\text{mu})^2/(2\times\text{variance})}$
                ▷ note that sum of probabilities that this function generates is equal to 1/2
                        ▷ because we are ignoring probabilities of negative numbers
7: function create_pdf_vector
8:     pdf_vector ← empty array of size = domain
9:     for $x$ ← 0 to domain do
10:        pdf_vector[x] ← pdf(x)
11:    return: pdf_vector
                ▷ normalize probability list such that sum of probability would equal to 1
12: function normalize_pdf_vector(pdf_vector)
13:    $s$ ← sum(pdf_vector)
14:    for $i$ ← 0 to domain do
15:        pdf_vector[$i$] = pdf_vector[$i$] /sum
16:        $i \leftarrow i + 1$
17:    return: pdf_vector
                                        ▷ create cdf vector from pdf_vector
18: function create_cdf_vector(pdf_vector)
19:    cdf_vector ← empty array of size = domain
20:    total ← 0
21:    for $i$ ← 0 to domain do
22:        total ← total + pdf_vector[$i$]
23:        cdf_vector[$i$] ← total
24:    return: cdf_vector
```

```
                                              ▷ sample from cdf_vector
25: function sample
26:     rand ← select a number in range $[0, 1)$ uniformly random
27:     index ← 0
28:     while rand > cdf_vector[index] do
29:         index ← index +1
30:     return: index
                                              ▷ initialize the sampler
31: pdf_vector ← create_pdf_vector()
32: pdf_vector ← normalize_pdf_vector(pdf_vector)
33: cdf_vector ← create_cdf_vector(pdf_vector)
34: s ← sample(cdf_vector)
```

```python
import random

sigma = 8/sqrt(2 * pi)  # standard deviation
variance = sigma^2      # variance
mu = 0                  # mean (or center of distribution)

# probability density of the normal distribution
def pdf(x, sigma, variance, mu):
    return N((1 / sqrt(2 * variance * pi)) * e^(-(x - mu)^2 / (2 * variance)))

# note that sum of probabilities that this function generates is equal to 1/2
# because we are ignoring probabilities of negative numbers
def create_pdf_vector():
    pdf_vector = [0] * ceil(variance)

    for x in range(0, variance):
        pdf_vector[x] = pdf(x, sigma, variance, mu)

    return pdf_vector

# normalize probability list such that sum of probability would equal to 1
def normalize_pdf_vector(pdf_vector):
    s = sum(pdf_vector)
    return map((lambda x: x / s), pdf_vector)

# create cdf vector from pdf vector
def create_cdf_vector(pdf_vector):
    domain = len(pdf_vector)
    cdf_vector = [0] * domain
    total = 0

    for i in range(domain):
        total = total + pdf_vector[i]
        cdf_vector[i] = total

    return cdf_vector

# sample from cdf vector
def sample(cdf_vector):
    rand = random.uniform(0, 1)
    index = 0

    while(rand > cdf_vector[index]):
        index = index + 1

    return index

# create PDF vector and normalize it
pdf_vector = create_pdf_vector()
pdf_vector = normalize_pdf_vector(pdf_vector)

# create CDF vector from PDF vector
cdf_vector = create_cdf_vector(pdf_vector)

arr = [0] * len(pdf_vector)
for i in range(100000):    # count of samples
    s = sample(cdf_vector)
    arr[s] = arr[s] + 1

bar_chart(array)    # create bar char graph of samples
```

Listing 5.1: Naive Gaussian sampler

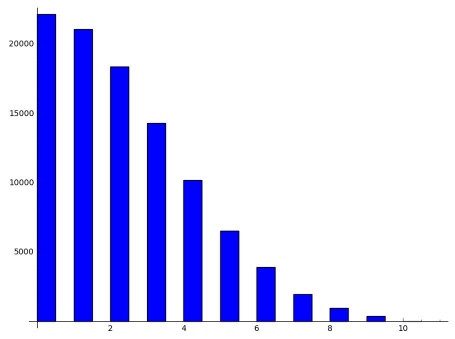


Figure 5.1: Plot (or histogram) of Gaussian samples using the naive method
Above is centered at 0 and $\sigma = \frac{8}{\sqrt{2\pi}}$

5.1.1.2 Inversion sampling using PDF function and bisection search

Above method can be improved using Taylor approximation to estimate the CDF and bisection search to estimate the inverse of CDF. It is important to note that CDF function of normal distribution is not invertible so to calculate the inverse of CDF function we need to use Taylor series to find the integral (or area under the graph) of PDF function.

The bisection search method is a root-finding method that repeatedly bisects an interval and then selects a sub-interval in which a root must lie for further processing. It is a very simple and robust method, but it is also relatively slow. Because of this, it is often used to obtain a rough approximation to a solution which is then used as a starting point for more rapidly converging methods. This method is also called the interval halving method. Bisection search method does not really takes the advantage of the fact that coefficients of error polynomial are integers, so we need to round the sample to nearest integer which is not the most efficient approach. Also, to calculate the inverse of CDF we are constantly calculating CDF and also so we are not storing the CDF values (memorizing concept) of particular probability. As a result, this approach is very slow.

Algorithm 6 Gaussian sampler using Taylor series and bisection search

```
1: sigma $\leftarrow 8/\sqrt{2\pi}$, variance $\leftarrow$ sigma$^2$
                                        ▷ probability density of the normal distribution
2: function pdf(x) return $(1/\sqrt{2\pi \times \text{variance}}) \times e^{-(x-\text{mu})^2/(2\times\text{variance})}$
                              ▷ calculates cumulative distribution using Taylor approximation
3: function cdf(x)
4:     if x < -1 × variance then return 0.0
5:     else if x > variance then return 1.0
6:     sum $\leftarrow$ 0.0
7:     term $\leftarrow x$
8:     $i \leftarrow 3$
9:     while sum + term $\neq$ sum do
10:        sum $\leftarrow$ sum + term
11:        term $\leftarrow$ term $\times x^2/i$
12:        $i \leftarrow i + 2$
       return 0.5+ sum × pdf(x)
                           ▷ bisection search to find a value $y$ such that it's CDF equals to $x$
13: function inverse_cdf(x, delta, lo, hi)
14:     mid $\leftarrow$ lo + (hi − lo) /2
15:     if hi − lo < delta then return mid
16:     else if cdf(mid) > x then
17:         return inverse_cdf(x, delta, lo, mid)
18:     else
19:         return inverse_cdf(x, delta, mid, hi)
                                                          ▷ initialize the bisection search
20: function inverse_cdf_init(x)
21:     mid $\leftarrow$ lo + (hi − lo) /2
22:     return inverse_cdf(x, 0.00000001, -variance, variance)
```

```
                ▷ sample from Gaussian by generating a random number and find it's inverse of CDF
23: function sample
24:     $x \leftarrow$ generate a number between $[0, 1)$ uniformly random
25:     return inverse_cdf(x, variance)
                                                                  ▷ initialize the sampler
26: pdf_vector ← create_pdf_vector()
27: pdf_vector ← normalize_pdf_vector(pdf_vector)
28: cdf_vector ← create_cdf_vector(pdf_vector)
29: s ← sample(cdf_vector)
```

```python
import random

# return pdf(x) = standard Gaussian pdf
def pdf(x):
    return e^(-x * x / 2) / sqrt(2 * pi)

# return pdf(x, mu, signma) = Gaussian pdf with mean mu and stddev sigma
def pdf_init(x, mu, sigma):
    return pdf((x - mu) / sigma) / sigma

# return cdf(z) = standard Gaussian cdf using Taylor approximation
def cdf(z, variance):
    if z < -variance:
        return 0.0
    if z > variance:
        return 1.0
    sum = 0.0
    term = z
    i = 3
    while sum + term != sum:
        sum = sum + term
        term = term * z * z / i
        i += 2
    return 0.5 + sum * pdf(z)

# return cdf(z, mu, sigma) = Gaussian cdf with mean mu and stddev sigma
def cdf_init(z, mu, sigma):
    return cdf((z - mu) / sigma)

# Compute z such that cdf(z) = y via bisection search
def inverse_CDF_init(y, variance):
    return inverse_CDF(y, 0.00000001, -variance, variance, variance)

# bisection search
def inverse_CDF(y, delta, lo, hi, variance):
    mid = lo + (hi - lo) / 2
    if hi - lo < delta:
        return mid
    if cdf(mid, variance) > y:
        return inverse_CDF(y, delta, lo, mid, variance)
    else:
        return inverse_CDF(y, delta, mid, hi, variance)
```

```


sigma = 8/sqrt(2 * pi)   # standard deviation
variance = ceil(sigma^2) # variance
array = [0] * variance * 2

for i in range(300):   # count of samples
    sample = round(inverse_CDF_init(random.uniform(0, 1), variance) * sigma)
    array[variance + sample] += 1

bar_chart(array).show()
```

Listing 5.2: Improved Gaussian sampler

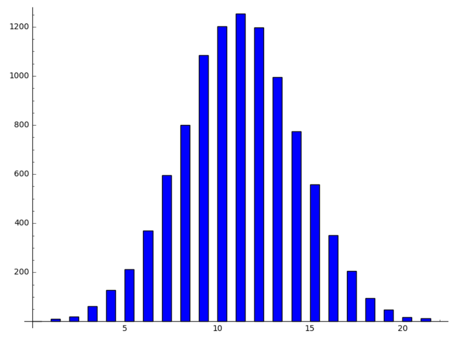


Figure 5.2: Plot (or histogram) Gaussian sampler using bisection search and Taylor series Above is centered at variance^2, $\sigma = \frac{8}{\sqrt{2\pi}}$

5.1.1.3 Inversion sampling using pre-calculated CDF table

BCNS protocol uses inversion sampling with pre-computed look-up table of CDF. If look-up table was not available, then we had to calculate the CDF. The specification of CDF look-up table is the following:

$$\begin{aligned} \text{table}[0] &= 2^{189} \\ \text{table}[51] &= 2^{192} \\ \text{table}[i] < \text{table}[i+1] &: \forall_i \in 0 \le i \le 50 \end{aligned} \tag{5.1}$$

To samples form this look-up table, we independently generate a 192-bit integer v_j uniformly at random, and compute the unique smallest integer index $ind_j \in [0, 50]$ such that $v_j < \text{table}[ind_j]$. We then generate one additional random bit to decide the sign

$sign_j \in \{-1, 1\}$, and return the j-th coefficient as $s_j \leftarrow sign_j \times ind_j$. BCNS uses little Endian memory storing format (i.e. 192 bits stored as `struct` of $3 \times 64 = 192$ bits) to store the array. Little endians in essence means storing the least significant byte in the smallest address.

Algorithm 7 gaussian sampler given CDF table as written in BCNS protocol

```
 1: rlwe_table ← $52 \times 3$ matrix of 64 bit little endians
 2: big_integers ← empty array with size equals to number of rows in rlwe_table (i.e. 52)
                                  ▷ convert little-endian as stored above to Big-endian
 3: function little_to_big_endian
 4:     for $i \leftarrow 0$ to height of rlwe_table do
 5:         big ← 0
 6:         for $j \leftarrow 0$ to width of rlwe_table do
 7:             big ← big OR (rlwe_table[$i$][$j$] LSHIFT ($j \times 64$))
 8:         big_integers[$i$] ← big
 9:     return big_integers

          ▷ this function uses bits that are sampled uniformly at random to sample Gaussian
10: function sample
11:     $n \leftarrow$ get a 192 bits uniformly random
12:     $i \leftarrow 0$
13:     for $i \leftarrow 0$ to length(big_integers) do
14:         if $n <$ big_integers[$i$] then
15:             return $i$
```

```python
import random, struct
# BCNS protocol CDF lookup table
rlwe_table = [
    [0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, 0x1FFFFFFFFFFFFFFF],
    [0xE0C81DA0D6A8BD22, 0x161ABD186DA13542, 0x5CEF2C248806C827],
    [0x8D026C4E14BC7408, 0x4344C125B3533F22, 0x9186506BCC065F20],
    [0x10AC7CEC7D7E2A3B, 0x5D62CE65E6217813, 0xBAAB5F82BCDB43B3],
    [0x709C92996E94D801, 0x1411F551608E4D22, 0xD7D9769FAD23BCB1],
    [0x6287D827008404B7, 0x7E1526D618902F20, 0xEA9BE2F4D6DDB5ED],
    [0x34CBDC118C15F40E, 0xE7D2A13787E94674, 0xF58A99474919B8C9],
    [0xD521F7EBBBE8C3A2, 0xE8A773D9A1EA0AAB, 0xFB5117812753B7B8],
    [0xC3D9E58131089A6A, 0x148CB49FF716491B, 0xFE151BD0928596D3],
    [0x2E060C4A842A27F6, 0x07E44D009ADB0049, 0xFF487508BA9F7208],
    [0xFCEDEFCFAA887582, 0x1A5409BF5D4B039E, 0xFFC16686270CFC82],
    [0x4FE22E5DF9FAAC20, 0xFDC99BFE0F991958, 0xFFEC8AC3C159431B],
    [0xA36605F81B14FEDF, 0xA6FCD4C13F4AFCE0, 0xFFFA7DF4B6E92C28],
    [0x9D1FDCFF97BBC957, 0x4B869C6286ED0BB5, 0xFFFE94BB4554B5AC],
    [0x6B3EEBA74AAD104B, 0xEC72329E974D63C7, 0xFFFFAADE1B1CAA95],
    [0x48C8DA4009C10760, 0x337F6316C1FF0A59, 0xFFFFEDDC1C6436DC],
    [0x84480A71312F35E7, 0xD95E7B2CD6933C97, 0xFFFFFC7C9DC2569A],
    [0x23C01DAC1513FA0F, 0x8E0B132AE72F729F, 0xFFFFFF61BC337FED],
    [0x90C89D6570165907, 0x05B9D725AAEA5CAD, 0xFFFFFFE6B3CF05F7],
    [0x692E2A94C500EC7D, 0x99E8F72C370F27A6, 0xFFFFFFFC53EA610E],
    [0x28C2998CEAE37CC8, 0xC6E2F0D7CAFA9AB8, 0xFFFFFFFF841943DE],
    [0xC515CF4CB0130256, 0x4745913CB4F9E4DD, 0xFFFFFFFFF12D07EC],
    [0x39F0ECEA047D6E3A, 0xEE62D42142AC6544, 0xFFFFFFFFFE63E348],
    [0xDF11BB25B50462D6, 0x064A0C6CC136E943, 0xFFFFFFFFFFD762C7],
    [0xCDBA0DD69FD2EA0F, 0xC672F3A74DB0F175, 0xFFFFFFFFFFFC5E37],
    [0xFDB966A75F3604D9, 0x6ABEF8B144723D83, 0xFFFFFFFFFFFFB48F],
    [0x3C4FECBB600740D1, 0x697598CEADD71A15, 0xFFFFFFFFFFFFFA72],
    [0x1574CC916D60E673, 0x12F5A30DD99D7051, 0xFFFFFFFFFFFFFFA1],
    [0xDD3DCD1B9CB7321D, 0x4016ED3E05883572, 0xFFFFFFFFFFFFFFFA],
    [0xB4A4E8CF3DF79A7A, 0xAF22D9AFAD5A73CF, 0xFFFFFFFFFFFFFFFF],
    [0x91056A8196F74466, 0xFBF88681905332BA, 0xFFFFFFFFFFFFFFFF],
    [0x965B9ED9BD366C04, 0xFFD16385AF29A51F, 0xFFFFFFFFFFFFFFFF],
    [0xF05F75D38F2D28A3, 0xFFFE16FF8EA2B60C, 0xFFFFFFFFFFFFFFFF],
    [0x77E35C8980421EE8, 0xFFFFEDD3C9DDC7E8, 0xFFFFFFFFFFFFFFFF],
    [0x92783617956F140A, 0xFFFFFF63392B6E8F, 0xFFFFFFFFFFFFFFFF],
    [0xA536DC994639AD78, 0xFFFFFFFB3592B3D1, 0xFFFFFFFFFFFFFFFF],
    [0x8F3A871874DD9FD5, 0xFFFFFFFFDE04A5BB, 0xFFFFFFFFFFFFFFFF],
    [0x310DE3650170B717, 0xFFFFFFFFFF257152, 0xFFFFFFFFFFFFFFFF],
    [0x1F21A853A422F8CC, 0xFFFFFFFFFFFB057B, 0xFFFFFFFFFFFFFFFF],
    [0x3CA9D5C6DB4EE2BA, 0xFFFFFFFFFFFFE5AD, 0xFFFFFFFFFFFFFFFF],
    [0xCFD9CE958E59869C, 0xFFFFFFFFFFFFFF81, 0xFFFFFFFFFFFFFFFF],
    [0xDB8E1F91D955C452, 0xFFFFFFFFFFFFFFFD, 0xFFFFFFFFFFFFFFFF],
    [0xF78EE3A8E99E08C3, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFE1D7858BABDA25, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFF9E52E32CAB4A, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFEE13217574F, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFFFD04888041, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFFFFF8CD8A56, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFFFFFFF04111, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFFFFFFFFE0C5, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFFFFFFFFFFC7, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF],
    [0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF]
]

bigIntegers = []
# convert little-endian as stored above to Big-endian
for row in rlwe_table:
    big = 0
    for (element, j) in zip(row, range(len(row))):
        big = big | (element << (j * 64))
    bigIntegers.append(big)

bar_chart(bigIntegers).show()   # create a  bar chart of CDF array

def sample():   # this function uses uniformly random bits to sample Gaussian
    n = random.getrandbits(192)
    for index in range(len(bigIntegers)):
        if n < bigIntegers[index]:
            return index

array = [0] * len(bigIntegers)  # create empty array with size = length of array
for i in range(100000):         # count of samples
    s = sample()
    array[s] += 1
```

```
bar_chart(array).show()         # create bar chart of samples (using occurrences)
```

Listing 5.3: Gaussian sampler using pre-calculated table of CDF

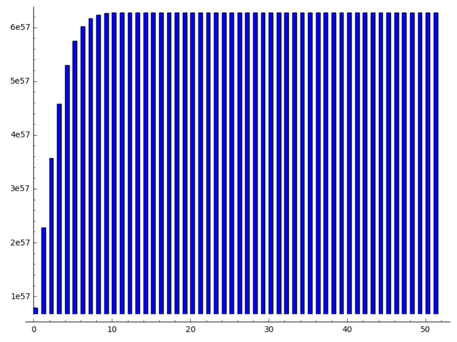


Figure 5.3: Plot of precomputed CDF table as implemented in BCNS protocol
Notice the last column is just 1s.

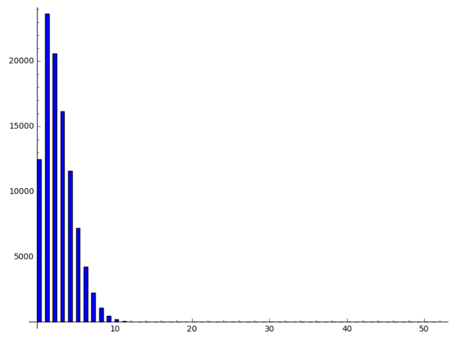


Figure 5.4: Plot (or histogram) of BCNS Gaussian sampler
Close to a perfect discrete Gaussian sampler (statistical difference $< 2^{-128}$)

5.1.2 Binomial distribution

NewHope protocol uses binomial distribution (Ψ_{16}) as oppose to Gaussian distribution for efficiency reasons. Interestingly, difference of two independently uniformly sampled random variables is a binomial distribution sample when bits are uniformly sampled. The method they use is to sample b, b' each 16 bits, then their difference forms a binomial distribution sample centered at zero. This distribution has standard deviation $\sigma = \sqrt{\frac{k}{2}}$, where k is number of bits. In the paper [4], Alkim et al. provides a proof that such a binomial distribution is a good approximation of Gaussian distribution.

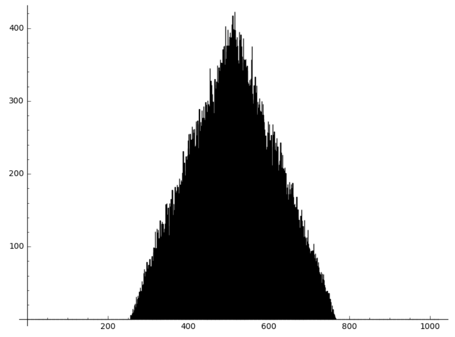


Figure 5.5: Plot of Binomial distribution Ψ_8, shifted for visualization purposes
This binomial distribution would have a standard deviation of $\sigma = \sqrt{\frac{8}{2}}$

Algorithm 8 Binomial distribution sampler with Ψ_{16} as written in NewHope protocol

```
                    ▷ generate two 16 bit uniformly random number return their difference
1: function sample(big_integers)
2:     $x \leftarrow$ generate a 16 bits number uniformly random
3:     $y \leftarrow$ generate a 16 bits number uniformly random
4:     return: $x - y$
```

```python
import random

# generate n bit uniformly random number
def random_bit(bit_len):
    return random.getrandbits(bit_len)

# sample two numbers and return their difference
def sample(bit_len):
    b_1 = random_bit(bit_len)
    b_2 = random_bit(bit_len)

    return int(b_1 - b_2)

# initialize a histogram, shifting is only for visualization purposes
bit_len = 8
dist = [0] * 2ˆbit_len * 2 * 2
offset = 2ˆbit_len

for i in range(100000): # number of samples
    s = sample(bit_len)
    dist[s +  offset] = dist[s + offset] + 1

bar_chart(dist).show()
```

Listing 5.4: Binomial distribution sampler with Ψ_8

5.2 Protocol specifications and speed comparisons

In this section we compare the specifics of implementations of BCNS and NewHope. In particular, generation of shared polynomial and polynomial multiplication algorithm. They are both important as they directly impact the bandwidth and speed of key-exchange.

5.2.1 Sending shared polynomial, two fundamentally different approaches

Shared polynomial is specially important as it needs to be shared with other party to enable key-exchange. Typically, the file size of this polynomial is relatively large as it should be generated from a uniform distribution (i.e. unlike error or noise polynomial, no upper-bound should be applies to it's coefficients). The following table shows the size of shared polynomial in Kilobytes for BCNS and NewHope protocols:

Protocol	Number of coefficients	Modulus	Size of shared polynomial	Payload
BCNS	1024	$2^{32} - 1$	1024×32bits $\approx$ 4KB	$\approx$ 4KB
NewHope	1024	12289	1024×14bits $\approx$ 1.8KB	256-bit seed

Table 5.1: Maximum size of shared polynomial and payload

BCNS during install (or build) of Open-SSL will generate a uniformly random shared polynomial and will use it for all the key-exchanges. Reusing shared polynomial more than once is completely fine under Ring-LWE hardness assumption. But the downside is key-exchange initiator also needs to send the shared polynomial to other party and sending 4KB of data is not an ideal solution. NewHope protocol addresses the issue by sending a *seed* so that other party can recreate the shared polynomial by themselves instead. In details, NewHope uses 256-bit seed through SHAKE-128 which is a hash function with arbitrary length output. Then slicing the resulting 16384 bits into 1024×16 bit numbers. Each of those integers is reduced to modulo 2^{14} (i.e. the two most-significant bits are set to zero) and then used as a coefficient of shared polynomial if it is smaller than q and rejected otherwise (or 0 is used instead).

The advantage that can be observed here is that some of the coefficients of shared polynomial would be set to 0 hence shared polynomial would not be too large and not too small. Therefore, in polynomial multiplication, we would not be dealing with a polynomial with relatively large coefficients, this results in a increased speed in key-exchange.

The following is a demonstration of creating a uniformly random polynomial from a random seed and extracting polynomial coefficients from it:

Algorithm 9 Extracting coefficients of shared polynomial as described in NewHope

```
1:  modulus ← 12289
2:  number_of_coefficients ← 1024

3:  function create_coefficient_array(seed)
4:                                              ▷ set the seed and length of output in bits
5:      raw ← SHAKE-128(seed, number_of_coefficients ×16)
6:      chunks ← convert hex digest of ‘raw’ to chunks (or sub-strings) of bit length = 16
7:      for i ← 0 to length(chunks) do
8:                          ▷ reduce chunk to modulo $2^{14}$ by setting 2 significant bits to zero
9:          chunks[i] ← chunk[i] XOR 0xC000
10:                                 ▷ if coefficient is greater than modulus then set to 0
11:         chunks[i] ← 0 if chunk[i] $\geq$ modulus else chunks[i]
        return chunks
                                                ▷ set the seed and then extract coefficients
12: seed ← 256 bit sampled uniformly random
13: coefficients ← create_coefficient_array(seed)
```

```python
import random, hashlib, sha3

shake_block_size = 16                       # by default SHA-3 block size is 16 bits

dimension = count_of_coefficients = 1024    # degree of polynomials
modulus = 12289                             # modulus

# Quotient polynomial ring
R.<X> = PolynomialRing(GF(modulus))         # Gaussian field of integers
Y.<x> = R.quotient(X^(dimension) + 1)       # Cyclotomic field

# seed should be {0, ..., 255}^32 or 8 X 32 = 256 bit uniformly random number
seed = random.getrandbits(8 * 32)

# calculate hex digest of seed
digest = sha3.shake_128(str(seed)).hexdigest(int(2 * count_of_coefficients))

# fill the binary representation with zero so ’110’ becomes ’0110’
b = str(bin(int(digest, 16)))[2:].zfill(count_of_coefficients * shake_block_size)

```

```
# convert list to chunks of size = shake_block_size
chunks = [int(b[i:i+shake_block_size], 2) for i in range(0, len(b), shake_block_size)]

# reduce chunks to modulu 2^14 by setting two most significant bits to zero
chunks = map((lambda x: x ^^ 0xC000), chunks)

# if chunk is less than modulus then keep it else reject it
chunks = map((lambda x: x if x < modulus else 0), chunks)

# convert list of coefficients to polynomial modulo
uniform_a = Y(chunks)

print uniform_a     # print the resulting polynomial
```

Listing 5.5: Extracting coefficients of shared polynomial as described in NewHope protocol using SageMath

```
import random, hashlib, sha3

shake_block_size = 16                       # by default SHA-3 block size is 16 bits

dimension = count_of_coefficients = 1024    # degree of polynomials
modulus = 12289                             # modulus

def test():
    # seed should be {0, ..., 255}^32 or 8 X 32 = 256 bit uniformly random number
    seed = random.getrandbits(8 * 32)

    # calculate hex digest of seed
    digest = sha3.shake_128(str(seed)).hexdigest(int(2 * count_of_coefficients))

    # fill the binary representation with zero so '110' becomes '0110'
    b = str(bin(int(digest, 16)))[2:].zfill(count_of_coefficients * shake_block_size)

    # convert list to chunks of size = shake_block_size
    chunks = [int(b[i:i+shake_block_size], 2) for i in range(0, len(b), shake_block_size)]

    # reduce chunks to modulu 2^14 by setting two most significant bits to zero
    chunks = map((lambda x: x ^^ 0xC000), chunks)

    # if chunk is less than modulus then keep it else reject it
    chunks = map((lambda x: x if x < modulus else 0), chunks)

    return sum( [1 if coefficient > 0 else 0 for coefficient in chunks] )

bar_chart( [test() for _ in range(1000)] ).plot().show()
```

Listing 5.6: Plot of number of non-zero coefficients as a result of NewHope's shared polynomial generation algorithm using SageMath

In the following bar chart plot we can see that number of non-zero coefficients of shared polynomial is on-average 200 out of all 1024 coefficients. This is good as it makes the shared polynomial sufficiently large, not too small, but still uniformly at random.

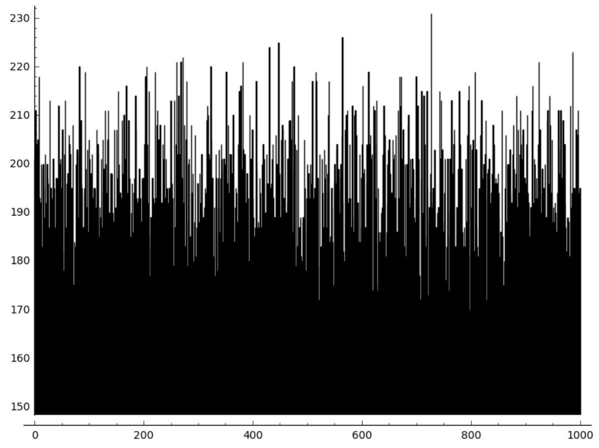


Figure 5.6: Plot of number of non-zero coefficients after creating 1000 shared polynomials

5.2.2 Polynomial multiplication algorithm

BCNS protocol uses Split-radix FFT algorithm which is a variant of Mersenne Number Transform ($m = 2^k - 1$) outlined by H. Nussbaumer in [28] known. This algorithm is based on recursive negacyclic convolutions and this naturally applies to cyclotomic rings where the degree is a power of 2. It requires to use a composite modulus. Note that $2^{32} - 1$ is not a prime as $q = (2^1+1)(2^2+1)(2^4+1)(2^8+1)(2^{16}+1)$. This polynomial multiplication method requires $\mathrm{O}(n \log n)$ multiplications in $\mathbb{R}$ for n-digit numbers.

On the other hand, NewHope protocol uses Schonhage-Strassen algorithm which is a variant of Fermat Number Transform ($m = 2^{2^k}+1$) outlined by A. Schonhage and V. Strassenin [32]. Akim, author of NewHope protocol, chose $q = 12289$ as it is the smallest prime for which it holds that $q \equiv 1(\text{mod } 2n)$ so that the number-theoretic transform (NTT) can be realized efficiently. Note that $2n = 2^{11} \mid (q-1) = 12288 = 2^{12} + 2^{13}$. This polynomial multiplication method requires $\mathrm{O}(n \log(n) \log(\log(n)))$ multiplications in $\mathbb{Z}$ for n-digit numbers. Alkim discussed using Nussbaumer's Split-radix FFT algorithm instead due to a better time complexity but the C implementation was not much faster than Schonhage-Strassen algorithm implementation. But he hinted it is a possibility in future implementations to switch to Nussbaumer.

5.2.3 Performance analysis

The following is table comparison of clock cycles of BCNS and NewHope protocols (both C and AVX2 ref. codes). Clearly NewHope dominates the BCNS protocol in terms of speed and it's assembly language implementation is $\approx 4\times$ faster than C ref. code. Note that BCNS protocol generates the shared polynomial during the install (or build) of Open-SSL so the numbers below for BCNS does not include the clock cycles of that. However, the numbers for NewHope includes a $\approx$ 37000 clock cycles for generation of shared polynomial. But we always have an option to cache the shared polynomial indefinitely so we do not count it in clock cycles analysis but it would directly impact the required bandwidth for key-exchange. Note that Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture for microprocessors from Intel and AMD.

The reason for speed increase of NewHope is due to smaller modulus and choice of Binomial noise sampler which is more efficient to sample as oppose to Gaussian sampler.

Protocol	BCNS	NewHope's C ref.	NewHope's AVX2 ref.	X25519
Key generation (server)	2477958	258246	88920	52169
Key generation (client)	3995977	384994	110986	52169
Shared key computation	481937	86280	19422	159128
Sum of clock cyles	$\approx$ 6955K	$\approx$ 729K	$\approx$ 219K	$\approx$ 263K

Table 5.2: Clock cycle comparisons of BCNS, NewHope and fastest X25519 implementation

Clearly NewHope is faster with regards to sum of clock cycles in comparison with state-of-the art elliptic curve cryptography based Diffie-Hellman key-exchange (ECC). Total clock cycles of 219K vs. 263K. More specifically X25519; the fastest implementation is called Sandy2x written in 2015 by T. Chou (student of D. Bernstein) which is also written in AVX2 assembly language [9]. This means that lattice based key-exchange is not only a quantum safe option but it also faster than today's key-exchange protocols.

Alkim et al. went further and provided ARM Cortex implementation which can be used for portable devices, because these devices usually use ARM architecture in contrast with AVX2 which is primarily used in Desktop computers. The ARM reference implementation runs in $\approx$ 1.5M cycles in contrast with $\approx$ 3.6M cycles of fastest X25519 implementation in ARM.

NewHope's AVX2 reference implementation (not the C implementation to make comparison fair) out performs the fastest elliptic curve based Diffie-Hellman in conjunction with either RSA or ECDSA (elliptic curve Digital Signature Algorithm), hence, it would make it easy pick to replace existing key-exchange protocol. Also, NewHope outperforms post-quantum competitors by factor of two. Below shows the speed comparison of NTRUEncrypt (or NTRU), Frodo (which is identical to BCNS but uses LWE as oppose to Ring-LWE variant), BCNS and NewHope.

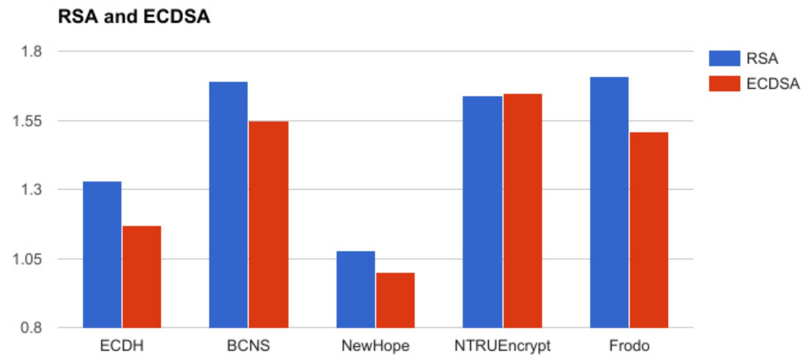


Figure 5.7: TLS handshake latency (KEX protocols in conjunction with RSA and ECDSA) The shorter the bar chart, the faster TLS handshake would be.

To summarize, NewHope is a key-exchange protocol that is believed to be resistant to attacks by quantum computers but RSA and ECDSA are vulnerable. However, quantum computers are still being development and they are not available yet, so switching to NewHope would not result in any overhead, but it would also make key-exchange resistant to quantum computers and faster.

5.2.4 Hybrid approach with X25519

Authors of both BCNS and NewHope note that the Ring-LWE problem is hypothesised to be resistant to quantum attack, but we do not know that. In fact, we also do not know that it is resistant to attacks by classical computers. It is possible that someone will develop a classical algorithm tomorrow that breaks the scheme above. Thus Ring-LWE key-exchange should be used concurrently with a standard Diffie-Hellman (e.g. X25519) and their shared key should concatenated (padded with) with each other to combine security of both future and past. Both BCNS and NewHope implementations in Open-SSL and Boring-SSL have options for hybrid key-exchange but they are set to `false` by default.

5.2.5 Addition to popular cryptographic libraries

Ring-LWE-based ciphersuite described in the previous subsection are both implemented into Open-SSL and Boring-SSL. Specifically, two new ciphersuites, all designed to achieve a 128-bit security level. The two ciphersuites are: `RLWE-ECDSA-AES128-GCM-SHA256` and `RLWE-RSA-AES128-GCM-SHA256`, and they consist of:

- Key-exchange based on Ring-LWE key exchange
- Authentication based on ECDSA (elliptic curve Digital Signature Algorithm) or RSA digital signatures
- Authenticated encryption based on AES-128 in GCM (Galois Counter Mode)
- Key derivation and hashing based on SHA-256

All these ciphersuite require TLSv1.2 because of the use of AES-GCM (provides both privacy or encryption and integrity), but TLSv1.0 ciphersuites using AES in CBC mode are possible.

Chapter 6

Summary, conclusion and future work

LWE and Ring-LWE promise a quantum safe passive key-exchange and recent implementations not only offer a drop-in replacement for cryptography software libraries but they also outperform the existing key-exchange algorithm as well. Understanding the concept behind Ring-LWE problem and how it relates to lattices and lattice problems (e.g SVP problem) is an essential precursor to understand the Ring-LWE based key-exchange protocols. Reconciliation algorithms plays a key role that enables the key-exchange protocols to extract the same bits with very small probability of failure.

This thesis provides a brief survey on lattice based cryptography how it relates to LWE problem. Thereafter, it discusses the reason behind creation of Ring-LWE which provides a structure to LWE matrix of module finite field by introducing concept of quotient polynomials modulo finite field. Then discussion of the basics of Ring-LWE key-exchange and the need for reconciliation. Next, it reviewed parameter choices and compared the specifics of different reconciliation algorithms. Further, it reviews the two real-world and highly optimized implementations of Ring-LWE key-exchange protocols; in details, noise sampler, ways to generate shared polynomial and polynomial multiplication algorithm. In the end, it discusses the performance analysis and possibility of using Hybrid (ECC+Ring-LWE) approach as opposed to only Ring-LWE.

In the future work, Ring-LWE based key-exchange might benefit from improved reconciliation algorithms that are simpler to implement yet yield higher success probability. Current key-exchange protocol such as elliptic curve Diffie Hellman and RSA have been studied extensively and their shortcoming have been identified and addressed. For example, Wiener's attack (small private key d) and Coppersmith's attack (small public exponent e) for RSA. However, lattice based algorithm are relatively new and the claim of security against quantum computers and even classical computers needs to be studied further as field of quantum computing grows.

Lastly, more work needs to be done to standardize key-exchange protocols and their efficient implementations in various applications, frameworks and programming languages (e.g. Ring-LWE key-exchange as a session-key in context of client–server model inside web frameworks). Exploring other candidates for post-quantum cryptography can also be done as future works. For example, McEliece cryptosystem, NTRU encryption, Supersingular isogeny key-exchange and LWE based key-exchange (without a ring) and attempts to make them more efficient in terms of both space and time complexity.

Appendix A

Merkle–Hellman knapsack cryptosystem

The Merkle–Hellman knapsack cryptosystem was one of the earliest public key cryptosystems invented by Ralph Merkle and Martin Hellman in 1978 and it has been broken.

Merkle-Hellman is an asymmetric-key cryptosystem, meaning that two keys are required for communication: a public key and a private key. Furthermore, unlike RSA, it is one-way: the public key is used only for encryption, and the private key is used only for decryption. The Merkle-Hellman system is based on the subset sum problem (a special case of the knapsack problem). The problem is as follows: given a set of numbers $\mathbf{A}$ and a number b, find a subset of $\mathbf{A}$ which sums to b. In general, this problem is known to be NP-complete. However, if the set of numbers (called the knapsack) is super-increasing, meaning that each element of the set is greater than the sum of all the numbers in the set lesser than it, the problem is "easy" and solvable in polynomial time with a simple greedy algorithm.

Key generation: In Merkle-Hellman, the keys are two knapsacks. The public key is a *hard* knapsack $\mathbf{A}$, and the private key is an *easy*, or superincreasing, knapsack $\mathbf{B}$, combined with two additional numbers, a multiplier and a modulus. The multiplier and modulus can be used to convert the superincreasing knapsack into the hard knapsack. These same numbers are used to transform the sum of the subset of the hard knapsack into the sum of the subset of the easy knapsack, which is a problem that is solvable in polynomial time.

To encrypt n-bit messages, choose a superincreasing sequence $w = (w_1, w_2, \ldots, w_n)$ of n nonzero natural numbers. Pick a random integer q, such that $q > \sum_{i=1}^{n} w_i$, and a random integer r, such that $\gcd(r, q) = 1$ (i.e. r and q are coprime). q is chosen this way to ensure the uniqueness of the ciphertext. If it is any smaller, more than one plaintext may encrypt to the same ciphertext. Since q is larger than the sum of every subset of w, no sums are congruent modulo q and therefore none of the private keys sums will be equal. r must be coprime to q or else it will not have an inverse modulo q. The existence of the inverse of r is necessary so that decryption is possible. Now calculate the sequence: $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ where $\beta_i = rw_i \bmod q$. The public key is β, while the private key is: (w, q, r).

Encryption: To encrypt a message, a subset of the hard knapsack $\mathbf{A}$ is chosen by comparing it with a set of bits (the plaintext) equal in length to the key. Each term in the public key that corresponds to a 1 in the plaintext is an element of the subset $\mathbf{A}_m$, while terms that corresponding to 0 in the plaintext are ignored when constructing $\mathbf{A}_m$ they are not elements of the key. The elements of this subset are added together and the resulting sum is the ciphertext.

To encrypt an n-bit message $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, where α_i is the i-th bit of the message and $\alpha_i \in \{0, 1\}$, calculate $c = \sum_{i=1}^{n} \alpha_i \beta_i$. The ciphertext then is c.

Decryption: Decryption is possible because the multiplier and modulus used to transform the easy knapsack into the public key can also be used to transform the number representing the ciphertext into the sum of the corresponding elements of the superincreasing knapsack. Then, using a simple greedy algorithm, the easy knapsack can be solved using O(n) arithmetic operations, which decrypts the message.

In order to decrypt a ciphertext c a receiver has to find the message bits α_i such that they satisfy $c = \sum_{i=1}^{n} \alpha_i \beta_i$. This would be a hard problem if the β_i were random values because the receiver would have to solve an instance of the subset sum problem, which is known to be NP-hard. However, the values βi were chosen such that decryption is easy if the private key (w, q, r) is known.

The key to decryption is to find an integer s that is the modular inverse of r modulo q. That means s satisfies the equation $s \times r \bmod q = 1$ or equivalently there exist an integer k such that $s \times r = kq + 1$. Since r was chosen such that $\gcd(r, q) = 1$ it is possible to find s and k by using the Extended Euclidean algorithm.

Next the receiver of the ciphertext c computes $c' \equiv cs \pmod{q}$. Hence $c' \equiv cs \equiv \sum_{i=1}^{n} \alpha_i \beta_i s \pmod{q}$. Because of $r \times s \bmod q = 1$ and $\beta_i = r \times w_i \bmod q$ follows $\beta_i s \equiv w_i r s \equiv w_i \pmod{q}$.

Hence $c' \equiv \sum_{i=1}^{n} \alpha_i w_i \pmod{q}$. The sum of all values w_i is smaller than q and hence $\sum_{i=1}^{n} \alpha_i w_i$ is also in the interval $[0, q-1]$. Thus the receiver has to solve the subset sum problem $c' = \sum_{i=1}^{n} \alpha_i w_i$.

This problem is easy because w is a superincreasing sequence. Take the largest element in w, say w_k. If $w \times k > c'$, then $\alpha_k = 0$, if $w_k \leq c'$, then $\alpha_k = 1$. Then, subtract $w_k \times \alpha_k$ from c', and repeat these steps until we have figured out α.

Example of Merkle–Hellman knapsack cryptosystem

First, a w is created: $w = \{2, 7, 11, 21, 42, 89, 180, 354\}$. This is the basis for a private key. From this, calculate the sum. $\sum w = 706$, then, choose a number q that is greater than the sum, say, $q = 881$.

Also, choose a number r that is in the range $[1, q)$ and is coprime to q, say, $r = 588$. The private key consists of q, w and r. To calculate a public key, generate the sequence β by multiplying each element in w by $r \bmod q$. Then, $\beta = \{295, 592, 301, 14, 28, 353, 120, 236\}$ because:

$$(2 \times 588) \bmod 881 = 295$$

$$(7 \times 588) \bmod 881 = 592$$

$$(11 \times 588) \bmod 881 = 301$$

$$(21 \times 588) \bmod 881 = 14$$

$$(42 \times 588) \bmod 881 = 28$$

$$(89 \times 588) \bmod 881 = 353$$

$$(180 \times 588) \bmod 881 = 120$$

$$(354 \times 588) \bmod 881 = 236$$

The sequence β makes up the public key. Say Alice wishes to encrypt 'a'. First, she must translate 'a' to binary (in this case, using ASCII) 01100001. She multiplies each respective bit by the corresponding number in β:

$$a = 01100001$$

$$\rightarrow 0 \times 295 + 1 \times 592 + 1 \times 301 + 0 \times 14 + 0 \times 28 + 0 \times 353 + 0 \times 120 + 1 \times 236 = 1129$$

She sends this to the recipient. To decrypt, Bob multiplies 1129 by $r^{-1} \bmod q$, $1129 \times 442 \bmod 881 = 372$. Now Bob decomposes 372 by selecting the largest element in w which is less than or equal to 372. Then selecting the next largest element less than or equal to the difference, until the difference is 0:

$$372 - 354 = 18$$

$$18 - 11 = 7$$

$$7 - 7 = 0$$

The elements we selected from our private key correspond to the 1 bits in the message: 01100001. When translated back from binary, this 'a' is the final decrypted message.

Breaking Merkle–Hellman knapsack cryptosystem using LLL

This system was very popular for a while since it is very fast to implement. However, in the early 1980's, Shamir in [33] was able to peel away this disguise and obtain superincreasing sequence (or one that was equivalent to it). Consider a knapsack problem to be solved: $t = x_1 \times a_1 + x_2 \times a_2 + \cdots + x_n \times a_n$ such that t is a ciphertext and $\forall i : a_i \in \beta$. Then define a lattice $\mathcal{L}_a$ using the

$$\mathcal{L}_a = \begin{pmatrix} 1 & 0 & \dots & 0 & a_1 \\ 0 & 1 & \dots & 0 & a_2 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \dots & 1 & a_n \\ 0 & 0 & \dots & 0 & -t \end{pmatrix} \tag{A.1}$$

If $x = (x_1, \dots, x_n) \in \{0,1\}^n$ solves above, then $v = (x_1, \dots, x_n, 0) \in \mathcal{L}_a$. Note that v is a short vector. If it is the shortest vector in $\mathcal{L}_a$, then LLL finds v. The larger `MESSAGE_LEN`, the probability of success will be lowered because the reduced basis of lattice (although small) would not always yield shortest lattice vectors (i.e. solution of knapsack problem).

The reason lies in the facts that for any vector x of reduced lattice basis $\mathcal{L}_a$ that is reduced using LLL, $||b_1|| \leq 2^{\frac{n-1}{2}}||x||$ holds and as n (i.e. MESSAGE_LENGTH or dimension) gets larger, then vectors would be far from *shortest vector* of lattice. The LLL algorithm will not always produce the desired vector and therefore, the attack is not always successful. However, in practice, the lattice reduction attack is highly effective against the original Merkle-Hellman knapsack.

```
from sage.numerical.knapsack import Superincreasing
import random

MESSAGE_LEN = 24    # length of message (or superincreasing sequence)

def create_superincreasing_sequence(start, length): # this function creates superincreasing sequence
    sequence = []                                   # as described in Merkle-Hellman cryptosystem
    s = start
    val = start
    while length > 0:
        sequence.append(val)
        val = s + random.randrange(1, 10)           # add a random number between (0, 10)
        s = s + val
        length = length - 1

    return sequence

# generate private-key
W = create_superincreasing_sequence(10, MESSAGE_LEN)
print "valid W: ", Superincreasing(W).is_superincreasing()  # validate W is superincreasing

q = next_prime(sum(W) + 1)  # generate a prime greater than sum of W
r = random.randint(1, q)    # random number between 1 inclusive and q exclusive

# generate public-key
beta = [(r * W[i]) % q for _ in range(len(W))]

# create dummy message
message = [random.randrange(10) % 2 for _ in range(MESSAGE_LEN)]

# generate ciphertext
ciphertext = sum([beta[i] * message[i] for _ in range(MESSAGE_LEN)])

# decrypt ciphertext
c_dash = (inverse_mod(r, q) * ciphertext) % q

# decompose c_dash using W
subset = Superincreasing(W).subset_sum(c_dash)

# testing the cryptosystem
reconstructed_message = []
for i in W:
    if i in subset:
        reconstructed_message.append(1)
    else:
        reconstructed_message.append(0)

print "message == decrypt(ciphertext): ", message == reconstructed_message # should print True

# breaking the cryptosystem
l = matrix.identity(MESSAGE_LEN + 1)    # create identity matrix of size n+1

# insert beta and ciphertext into matrix
l.set_column(MESSAGE_LEN, beta + [-1 * ciphertext])

# reduce the lattice given basis matrix (or find the short vector in lattice)
l = l.LLL()

for i in range(l.nrows()):
# if row contains only {0, 1} then it might be a solution
    if all(x in [0, 1] for x in l.row(i)[:-1]):
        if l.row(i)[:-1].list() == message:
            print "Merkle-Hellman Knapsack Cryptosystem is broken!"
```

Listing A.1: Merkle–Hellman knapsack cryptosystem implemented using SageMath

Algorithm 10 Breaking Merkle–Hellman knapsack cryptosystem using LLL

```
1: message_length ← 24                ▷ larger this variable harder to break the cryptosystem
                                       ▷ this function creates superincreasing sequence
2: function create_superincreasing_sequence
3:     sequence ← create empty list of size equal to length
4:     $s$ ← 10
5:     val ← $s$
6:     index ← index +1
7:     for index ← 0 to message_length $-1$ do
8:         sequence[index] ← val
9:         val ← s + random integer $\in [1, 10]$
10:        s ← s + val
11:    return sequence

12: function create_public_key($w, r, q$)
13:    beta ← create empty list of size equal to message_length
14:    for index ← 0 to message_length - 1 do
15:        beta[index] ← ($r \times w$[index]) mod $q$
16:    return beta

17: function encrypt(beta, message)
18:    index ← 0
19:    ciphertext ← create empty list of size equal to message_length
20:    for index ← 0 to message_length do
21:        ciphertext[index] ← beta[index] × message[index]
22:    return sum(ciphertext)

23: function decrypt(ciphertext, $r, q$)
24:    return (inverse_mod(r, q) × ciphertext) mod $q$
```

```
                    ▷ breaks the cryptosystem if LLL produces shortest vector (not just shorter)
25: function break(ciphertext)
26:     L ← identity matrix with size equal to length of ciphertext +1
27:     L[last column] ← beta $\cup$ [$-1 \times$ ciphertext]
28:     L ← LLL(L)
29:     return the first row of L that contains only 0s and 1s

30: w ← create_superincreasing_sequence(10, message_length)
31: q ← random prime greater than sum(w) +1
32: r ← random number $\in (1, q)$
33: beta ← create_public_key(w, r, q)
34: message ← create a dummy binary message of size equal to message_length
35: ciphertext ← encrypt(beta, message)
36: break(ciphertext) $\overset{?}{=}$ message
```

Appendix B

Basic implementations of all Ring-LWE key-exchange reconciliations

One of the goals of this thesis was to implement a simple to understand, easy to modify and straightforward implementation of all four reconciliation mechanisms (i.e. Regev, Ding, Peikert and NewHope). However, after struggling with a well known number theory libraries such as NTL and FLINT which are indeed great packages written in C but mainly for ones who have studied and used number theory for a long time and speed is their primary concern, we selected Python language. Python has a syntax similar to pseudo-code but setting up an environment in Python to use all the libraries needed for the reconciliation implementation *and* achieve three criteria above would not be possible (i.e. setting up polynomial library, then configure quotient ring and etc.). Also, Python's popular mathematical libraries such as numpy do not offer a quotient Polynomial ring modulo finite field without using a work around like calling NTL library which is written in C. After searching further we came upon SageMath which uses a Python-like syntax, supporting procedural, functional and object-oriented constructs.

W. Stein, creator of SageMath who is a mathematician at the University of Washington, realized when designing Sage that there were many open-source mathematics software packages already written in different languages, namely C, C++, Common Lisp, Fortran and Python. Rather than reinventing the wheel, Sage which is written mostly in Python integrates many specialized mathematics software packages into a common interface, for which a user needs to know only Python. However, Sage contains hundreds of thousands of unique lines of code adding new functions and creating the interface between its components.

SageMath is an open-source project and it truly a powerful tool but it does not receive enough attention as it deserves specially for it's powerful polynomial package. We moved away from java after trying to implement a univariate polynomial ring using java (i.e. reinventing the wheel) and seeing how difficult it is to implement a proper polynomial parser to works with edge cases, although it was a rewarding experience, it was time consuming. We should also give Sage credit for providing comprehensive Matrix library which includes LLL (by calling NTL's LLL implementation) and many other implementations of lattice algorithms (e.g. BKZ, SVP, CVP and more).

Sage uses many open source libraries and links them all together using Python. It is a simple idea but it requires clever implementation which Sage truly delivers. It is fascinating to see a smart type system that can handle the linkage between all those open-source libraries.

The following are demonstration of Peikert and Alkim et al. reconciliation algorithms using SageMath. Note that these are not efficient implementations and they only serve as a platform to modify and experiment with the reconciliation algorithms. In all implementations below we used Sage's built in function: `DiscreteGaussianDistributionPolynomialSampler` which realizes oracles which returns polynomials in $\mathbb{Z}[x]$ where each coefficient is sampled independently with a probability proportional to $e^{(-(x-c)^2/(2\sigma^2))}$. Thereafter, we modified the ring of returned polynomial to quotient polynomial ring over finite field of integers using `Y(f)`

For the complete code and all four reconciliation methods please refer to:

https://github.com/amir734jj/LWE-KEX

The following are implementations of reconciliation algorithms not protocols. For example, with regards to the NewHope implementation, we used the Gaussian distribution sampler as oppose to Binomial distribution that NewHope protocol uses.

```
from sage.stats.distributions.discrete_gaussian_polynomial import DiscreteGaussianDistributionPolynomialSampler

dimension = 1024     # degree of polynomials
modulus = 40961      # modulus
sigma = 8/sqrt(2*pi) # sigma

# Quotient polynomial ring
R.<X> = PolynomialRing(GF(modulus))     # Gaussian field of integers
Y.<x> = R.quotient(X^(dimension) + 1)   # Cyclotomic field

def generate_error():
    # dimension = 5 (enough for error polynomial) ;  variance = sigma
    f = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 5, sigma)()
    return Y(f)

def generate_polynomial():
    # uniformly sampled from Quotient Polynomial Ring in x over finite field
    return Y.random_element()
```

Listing B.1: Configuration of quotient polynomial ring over finite field using SageMath

Below is a tester to check if reconciliation algorithm works correctly. This can be used in conjunction with any reconciliation algorithm.

```
shared = generate_polynomial()  # Shared matrix (A)

# Alice values
alice_secret = generate_error() # secret generated from error distribution
alice_error = generate_error()
alice_value = shared * alice_secret + alice_error   # create R-LWE sample

# Bob values
bob_secret = generate_error()   # secret generated from error distribution
bob_error = generate_error()
bob_value = shared * bob_secret + bob_error         # create R-LWE sample

# Bob key
temp_error = generate_error()   # create secondary error and add it to calculated key to increase entropy
bob_key = alice_value * bob_secret + temp_error
w = generate_signal(bob_key)    # generate signal (or reconciliation bits) from Bob's key
bob_key_binary = reconcile(bob_key, w)

# Alice key
alice_key = bob_value * alice_secret                # no need to add secondary error
alice_key_binary = reconcile(alice_key, w)          # reconcile using Bob's reconciliation bits

if (alice_key_binary == bob_key_binary):
    print "Keys match!", hex(int(alice_key_binary, 2)) # print hex value of shared key if they match
else:
    print "Keys do not match!"
```

Listing B.2: Tester for reconciliations using SageMath

Below is the implementation of Peikert's reconciliation algorithm. Notice the randomized doubling function that multiplies every coefficient of polynomial by two and then adds a secondary noise. BCNS protocol samples two bits and use it as an index to uniformly sample from: $\{-1, 0, 0, 1\}$.

```
temp_modulus = (2 * modulus) if is_odd(modulus) else modulus    # if modulus is odd then multiply it by 2
temp = temp_modulus / 8                 # q/8
value_1 = temp + (temp_modulus / 4)     # q/8 + q/4
value_2 = temp + (3 * temp_modulus / 4) # q/8 + 3q/4
value_3 = temp                          # q/8
value_4 = temp + (temp_modulus / 2)     # q/8 + q/2

# randomized double function, notice probability of 0 => 0.5
def dbl(coefficient):
    return  ( 2 * int( coefficient ) - numpy.random.choice([-1, 0, 1], p=[0.25, 0.5, 0.25]) ) % temp_modulus

def generate_signal(poly):
    coefficients = map(dbl, poly.list())    # apply dbl function to all coefficient
    signal = []
    for coefficient in coefficients:
        # if coefficient [0, q/4] OR [q/2, 3q/4] then signal bit = 1 else 0
        if (coefficient) <= (temp_modulus / 4) or \
            ((coefficient) <= (3 * temp_modulus / 4) and (coefficient) >= (temp_modulus / 2)):
            signal.append(1)
        else:
            signal.append(0)
    return signal

def reconcile(poly, w):
    coefficients = map(dbl, poly.list())    # apply dbl function to all coefficient
    key = []
    # use signal bit to reconcile
    for coefficient, bit in zip(coefficients, w):
        if bit == 1:
            key.append(1 if coefficient >= value_1 and coefficient <= value_2 else 0)
        else:
            key.append(1 if coefficient >= value_3 and coefficient <= value_4 else 0)
    return "".join(map(str, key))
```

Listing B.3: Peikert reconciliation implementation using SageMath

Below is the implementation of NewHope reconciliation algorithm. We did not implement the concept of splitting Voronoi cell into sub-cells as it is part of NewHope protocol not the reconciliation. Notice the structure is basically the same as Peikert's reconciliation (i.e.

`dbl`, `generate_signal`, `reconcile`). The `initialize` function will return an integer lattice created using identity matrix of dimension = `sub_dimension`. It also returns a polyhedron (or voronoi cell) centered at $(1/2, \ldots, 1/2)$ inside a lattice with the following basis (when `sub_dimension = 4`): $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \end{pmatrix}$

```python
sub_dimension = 4 # reconciliation sub dimension

# helper function to convert from [c_0, c_1, ..., c_1023] to [(c_0, c_1, c_2, c_3), ..., (..., c_1023)]
def grouped(iterable, n):
    return zip(*[iter(iterable)]*n)

def initialize():
    identity_matrix = Matrix.identity(RR, sub_dimension)    # create identity matrix
    integer_lattice = IntegerLattice(identity_matrix)       # construct integer lattice from identity matrix

    half_vector = [1/2 for i in range(sub_dimension)]       # construct 1/2 vector
    identity_matrix[sub_dimension - 1] = half_vector        # modify last row of identity matrix

    main_polyhedron = calculate_voronoi_cell(identity_matrix).translation(half_vector)  # create voronoi cell
                                                                                       #  from modified matrix
    return (integer_lattice, main_polyhedron) # return integer lattice and polyhedron centered at

def dbl(coefficient_vector):    # add 1(2q) to all coefficients with probability 1/2
    return coefficient_vector + \
    vector( numpy.random.choice([0, 1], p=[0.5, 0.5]) * vector([1/(2*modulus) for _ in range(sub_dimension)]))

def generate_signal(poly):      # signal generation function
    coefficients = map(lambda x: RR(x) / modulus, poly.list())  # divide coefficient by modulus
    distances = []
    for v in grouped(coefficients, sub_dimension):
        v = dbl(vector(v))  # apply randomized double function
        # if point (or vector) is in main polyhedron then use center of main polyhedron else use lattice CVP
        if main_polyhedron.contains(vector(v)):
            distance = main_polyhedron.center() - v
        else:
            distance = integer_lattice.closest_vector(v) - v
        distances.append(distance)
    return distances

def reconcile(poly, w):         # reconcile using reconciliation information
    coefficients = map(lambda x: RR(x) / modulus, poly.list())  # divide coefficient by modulus
    key = []
    for difference, v in zip(w, grouped(coefficients, sub_dimension)):
        v = dbl(vector(v))  # apply randomized double function
        coordinate = vector([round(point, 1) for point in (v + difference) ]) # round point to 1 decimal point
        key.append(1 if coordinate == main_polyhedron.center() else 0)
    return "".join(map(str, key))

(integer_lattice, main_polyhedron) = initialize()
```

Listing B.4: NewHope reconciliation implementation using SageMath

Bibliography

[1] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108. ACM, 1996.

[2] Miklós Ajtai. The shortest vector problem in l2 is np-hard for randomized reductions (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 10–19, New York, NY, USA, 1998. ACM.

[3] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 284–293, New York, NY, USA, 1997. ACM.

[4] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016.

[5] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '09, pages 595–618, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Dan Boneh and Richard J. Lipton. *Quantum Cryptanalysis of Hidden Linear Functions*, pages 424–437. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[7] Joppe W Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570. IEEE, 2015.

[8] Matt Braithwaite. Experimenting with post-quantum cryptography, Jul 2016.

[9] Tung Chou. Sandy2x: New curve25519 speed records. Cryptology ePrint Archive, Report 2015/943, 2015. http://eprint.iacr.org/2015/943.

[10] Don Coppersmith. Finding small solutions to small degree polynomials. In *Revised Papers from the International Conference on Cryptography and Lattices*, CaLC '01, pages 20–31, London, UK, UK, 2001. Springer-Verlag.

[11] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov 1976.

[12] Jintai Ding, Xiang Xie, and Xiaodong Lin. A simple provably secure key exchange scheme based on the learning with errors problem. *IACR Cryptology ePrint Archive*, 2012:688, 2012.

[13] Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete gaussians for lattice-based cryptography on a constrained device. *Appl. Algebra Eng., Commun. Comput.*, 25(3):159–180, June 2014.

[14] A. M. Frieze. On the lagarias-odlyzko algorithm for the subset sum problem. *SIAM Journal on Computing*, 15(2):536–539, 1986.

[15] Steven D Galbraith and Victor Rotger. Easy decision-diffie-hellman groups. Cryptology ePrint Archive, Report 2004/070, 2004. http://eprint.iacr.org/2004/070.

[16] Satrajit Ghosh and Aniket Kate. Post-quantum forward-secure onion routing (future anonymity in today's budget). Cryptology ePrint Archive, Report 2015/008, 2015. http://eprint.iacr.org/2015/008.

[17] O. Goldreich, D. Micciancio, S. Safra, and J. P. Seifert. Approximating shortest lattice vectors is not harder than approximating closet lattice vectors. *Inf. Process. Lett.*, 71(2):55–61, July 1999.

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

[19] Folláth János. *tmmp*, volume 60, chapter Gaussian Sampling in Lattice Based Cryptography. Tatra Mountains Mathematical Publications, 2017 2014.

[20] S. Khot. Hardness of approximating the shortest vector problem in lattices. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 126–135, Oct 2004.

[21] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[22] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011*, CT-RSA'11, pages 319–339, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] Vadim Lyubashevsky. Lattice signatures without trapdoors. Cryptology ePrint Archive, Report 2011/537, 2011. `http://eprint.iacr.org/2011/537`.

[24] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact knapsacks are collision resistant. In Ingo Wegener, Vladimiro Sassone, and Bart Preneel, editors, *Proceedings of the 33rd international colloquium on automata, languages and programming - ICALP 2006*, volume 4052 of *Lecture Notes in Computer Science*, pages 144–155, Venice, Italy, July 2006. Springer-Verlag.

[25] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Report 2012/230, 2012. `http://eprint.iacr.org/2012/230`.

[26] Daniele Micciancio. Generalized compact knapsaks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, December 2007. Prelim. in FOCS 2002.

[27] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, April 2007.

[28] H. Nussbaumer. Fast polynomial transform algorithms for digital convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(2):205–215, Apr 1980.

[29] Chris Peikert. Lattice cryptography for the internet. In *International Workshop on Post-Quantum Cryptography*, pages 197–219. Springer, 2014.

[30] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.

[31] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Math. Programming*, pages 181–191, 1993.

[32] A. Schonhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3):281–292, 1971.

[33] Adi Shamir. A polynomial time algorithm for breaking the basic merkle-hellman cryptosystem. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 145–152, Washington, DC, USA, 1982. IEEE Computer Society.

[34] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, Washington, DC, USA, 1994. IEEE Computer Society.

[35] Vikram Singh. A practical key exchange for the internet using lattice cryptography. Cryptology ePrint Archive, Report 2015/138, 2015.

[36] Vikram Singh and Arjun Chopra. Even more practical key exchanges for the internet using lattice cryptography. Cryptology ePrint Archive, Report 2015/1120, 2015. http://eprint.iacr.org/2015/1120.

[37] E. Viterbo and E. Biglieri. Computing the voronoi cell of a lattice: the diamond-cutting algorithm. *IEEE Transactions on Information Theory*, 42(1):161–171, Jan 1996.

[38] Xiaoyun Wang, Guangwu Xu, Mingqiang Wang, and Xianmeng Meng. *Mathematical Foundations of Public Key Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 2015.

[39] Patrick Weiden, Andreas Hülsing, Daniel Cabarcas, and Johannes Buchmann. Instantiating treeless signature schemes. Cryptology ePrint Archive, Report 2013/065, 2013. http://eprint.iacr.org/2013/065.