



INDIAN STATISTICAL INSTITUTE

MASTER'S THESIS

**Tweaking ML-KEM (Kyber) and
ML-DSA (Dilithium)**

Author:

Kumar Rahul

Supervisors:

Asso.Prof Y.V. Subba Rao

Prof. Subhamoy Maitra

*A thesis submitted in fulfillment of the requirements
for the degree of Master of Technology*

in

Cryptology and Network Security

CERTIFICATE

I certify that I have read the thesis titled **Tweaking ML-KEM (Kyber) and ML-DSA (Dilithium)** – with a focus on Government Oriented Applications – prepared under our guidance by **Kumar Rahul**, and in our opinion it is fully adequate in scope and in quality as a dissertation for the degree of **Master of Technology in Cryptology and Network Security** of the Indian Statistical Institute.



Y.V. Subba Rao

Associate Professor

School of Computer and Information
Sciences

University of Hyderabad

(Internal Guide)

Subhamoy Maitra

Professor and Head

Applied Statistics Unit
Indian Statistical Institute

Kolkata

(External Guide)

Kolkata

June, 2024.

Declaration of Authorship

I, Kumar Rahul, declare that this thesis titled, “Tweaking ML-KEM (Kyber) and ML-DSA (Dilithium)” with a focus on government oriented post quantum applications and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Master’s degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself

Signed:



Date:

24-06-25

Abstract

Lattice-based cryptography is the use of conjectured hard problems on point lattices in \mathbb{R}^n as the foundation for secure cryptographic systems. Attractive features of lattice cryptography include apparent resistance to quantum attacks (in contrast with most number-theoretic cryptography), high asymptotic efficiency and parallelism, security under worst-case intractability assumptions, and solutions to long-standing open problems in cryptography.

This work surveys the structure, security, and optimization potential of two leading lattice-based cryptographic schemes: ML-KEM (Kyber) and ML-DSA (Dilithium). Special attention is given to their applicability in government-oriented post-quantum cryptographic systems, focusing on performance, implementation considerations, and resilience against known quantum threats. In particular, the study introduces *tweaks to implementational-level components*—such as encoding, compression, and sampling routines—to enhance efficiency and adaptability. Emphasis is placed on how the underlying Short Integer Solution (SIS) and Learning With Errors (LWE) problems—and their ring-based variants—form the mathematical backbone of these NIST-standardized algorithms.

Acknowledgements

I would like to express my profound gratitude to my research guides, Professor Subhamoy Maitra and Associate Professor Y. V. Subba Rao, for their unwavering guidance and insightful supervision. Their expertise and encouragement were pivotal in shaping every aspect of this thesis.

Special thanks to Dr. Arpita Maitra for her invaluable suggestions, which helped refine the direction of my work, and to Professor Mriganka Mandal, whose introduction to the realm of Quantum and Post-Quantum Cryptography sparked my deep interest in this field.

Finally, I am deeply thankful to my wife, Dr. Akanksha Singh, for her constant encouragement and steadfast support throughout this journey.

Thank you all for your patient support and belief in my work.

Contents

Declaration of Authorship	ii
Abstract	iii
Acknowledgements	iv
1 Introduction	1
2 Background	6
2.1 Notation	6
2.2 Lattices	6
2.2.1 Basic Definitions	6
2.2.2 Computational Problems	8
2.3 LWE	10
2.3.1 Introduction	10
2.3.2 The LWE setup	10
2.3.3 The LWE assumption	11
The search-LWE assumption	11
The distinguishing-LWE assumption	11
Difference between the two assumptions	11
2.4 (Discrete) Gaussians and Subgaussians	12
2.5 Cryptographic Background	14
2.5.1 Function Families and Security Properties	14
2.5.2 Public-Key Encryption	15
2.5.3 Richer Forms of Encryption	16
3 ML-KEM: KYBER	18
3.1 Overview of CRYSTAL-KYBER	18
3.1.1 Key-Encapsulation Mechanisms	18
3.1.2 The KYBER Scheme	20
The Computational Assumption	20
The KYBER Construction	21

	Parameter Sets and Algorithms	21
3.2	Preliminaries and Notation	23
3.2.1	Bytes and Byte Arrays	23
3.2.2	Polynomial Rings and Vectors	23
3.2.3	Modular Reductions	23
3.2.4	Rounding	24
3.2.5	Sizes of Elements	24
3.2.6	Sets and Distributions	24
3.2.7	Compression and Decompression.	24
3.2.8	Symmetric Primitives	25
3.2.9	NTTs, Multiplication, and Bit-Reversed Order	25
3.2.10	Sampling from a binomial distribution.	26
3.2.11	Sets and Distributions	27
3.2.12	Encoding and Decoding	27
3.3	Specification of KYBER.CPAPKE	28
3.4	Specification of KYBER.CCAKEM	31
3.5	Design rationale	33
3.6	Expected Security Strength	34
3.6.1	Security Definition	34
3.6.2	Rationale of Our Security Estimates	34
3.6.3	Security Assumption	35
	Tight Reduction from MLWE in the ROM	35
	Non-Tight Reduction from MLWE in the QROM	35
3.6.4	Estimated security strength	36
	The Impact of the Deterministic Noise Caused by Compress_q on Kyber512	36
	The Impact of MAXDEPTH	36
3.7	Analysis with Respect to Known Attacks	37
3.7.1	Attacks Against the Underlying MLWE Problem	37
	MLWE as LWE	37
	Attacks Against LWE	37
	Enumeration vs. Sieving	38
	Primal Attack	38
	Dual Attack	39
3.7.2	Beyond core-SVP hardness	39
	A tentative gate-count estimate accounting for recent progress	41
3.7.3	Attacks exploiting decryption failures	42
3.8	KYBER Implementation	44

4	ML-DSA: Dilithium	50
4.1	CRYSTALS: Dilithium	50
4.1.1	Dilithium Parameters Overview	51
	Parameters	51
	Notations	51
	Algorithms for Dilithium	51
4.2	Build and Execution Guide	62
4.2.1	Installing Dependencies	62
4.2.2	Building <code>liboqs</code>	62
4.2.3	Sample Signing Code	62
4.2.4	Compiling the Sample Signing Code	65
4.2.5	Running the Executable	66
4.2.6	Expected Output	66
5	Tweaks To Kyber	67
5.1	Performance Analysis of Kyber	67
5.2	Security Analysis with respect to known attacks	70
6	Tweaks to Dilithium	74
6.1	Introduction	74
6.2	Tweak 1: Switching Hash Function	74
6.2.1	Description	74
6.2.2	Why It Works	74
6.2.3	Implementation	74
6.2.4	Impact	75
6.3	Tweak 2: Expanding Challenge Coefficients	76
6.3.1	Description	76
6.3.2	Why It Works	76
6.3.3	Implementation	76
6.3.4	Impact	77
6.4	Tweak 3: Modified Rejection Sampling	77
6.4.1	Description	77
6.4.2	Why It Works	77
6.4.3	Implementation	77
6.4.4	Impact	78
6.5	Benchmarking	78
6.5.1	Experimental Setup	78
6.5.2	Results	78

7	Code Snippets and Repository Access	80
8	Conclusion, Summary, and Future Work	82
8.1	Conclusion	82
8.2	Key Takeaways	83
8.3	Future Work	84
8.3.1	1. Advanced Parameter Tuning	84
8.3.2	2. Hybrid Noise Distributions	84
8.3.3	3. Hardware-Centric Optimizations	84
8.3.4	4. Multi-scheme Integration and Policy-Aware Switching	85
8.3.5	5. Formal Security Auditing and Compliance Layers	85
8.3.6	6. PQC-Enabled Government Applications	85
8.4	Closing Remark	85
	Bibliography	86

List of Figures

2.1	A good basis of a lattice is short and somewhat orthogonal, which allows rounding a point to a nearby lattice point successfully. A bad basis, with long and non-orthogonal vectors, does not support effective rounding.	7
2.2	A simplified diagram showing the reductions between Ideal-SVP, NTRU, Ring-LWE, and unique module-SVP problems. These structured problems underpin the hardness assumptions of lattice-based cryptography.	12
3.1	Overview of Key Encapsulation Mechanism (KEM) used in secure key exchange protocols.	19
4.1	ExpandMask function defined in Algorithm 13	55
4.2	Key auxiliary algorithms involved in Dilithium signature scheme operations.	56
4.3	The template for CRYSTALS-Dilithium at a glance	58
4.4	The basic construct for CRYSTALS: Dilithium Scheme	61
5.1	Test Result 1	68
5.2	Test Result 2	68
5.3	Test Result 3	68
5.4	Test Result 4	69
5.5	Test Result 5	71
5.6	Test Result 6	71
5.7	Test Result 7	71
5.8	Test Result 8	72

List of Tables

3.1	Kyber parameter sets	22
4.1	Parameters for CRYSTALS DILITHIUM	51
4.2	Notations for CRYSTALS DILITHIUM	52
5.1	Performance analysis of Kyber512 for different (d_u, d_v) values . . .	69
5.2	Performance analysis of Kyber768 for different (d_u, d_v) values . . .	69
5.3	Performance analysis of Kyber1024 for different (d_u, d_v) values . . .	70
5.4	Performance analysis of Kyber with different η_1, η_2 values	70
5.5	Security analysis of Kyber512 with different d_u, d_v values	71
5.6	Security analysis of Kyber768 with different d_u, d_v values	72
5.7	Security analysis of Kyber1024 with different d_u, d_v values	72
5.8	Security analysis of Kyber with different η_1, η_2 values	73
6.1	Benchmark Results: Dilithium Before and After Applying Tweak . .	79

List of Abbreviations

ML-KEM	Module Lattice-based Key Encapsulation Mechanism
ML-DSA	Module Lattice-based Digital Signature Algorithm
NIST	National Institute of Standards and Technology
KEM	Key Encapsulation Mechanism
DSA	Digital Signature Algorithm
PQC	Post-Quantum Cryptography
CPA	Chosen Plaintext Attack
CCA	Chosen Ciphertext Attack
LWE	Learning With Errors
SIS	Short Integer Solution
MLWE	Module Learning With Errors
MSIS	Module Short Integer Solution
NTT	Number Theoretic Transform
CRT	Chinese Remainder Theorem
API	Application Programming Interface
OQS	Open Quantum Safe
CLI	Command Line Interface
IND-CPA	Indistinguishability under Chosen Plaintext Attack
IND-CCA	Indistinguishability under Chosen Ciphertext Attack

Chapter 1

Introduction

The emergence of quantum computing has highlighted critical weaknesses in conventional cryptographic schemes, which are not designed to withstand attacks in the quantum era. Quantum algorithms, particularly those introduced by Shor and Grover, have demonstrated the ability to efficiently break widely used classical encryption methods. In response to this emerging threat, lattice-based cryptographic approaches have gained considerable attention, as their mathematical hardness assumptions are believed to remain secure even against quantum adversaries. These schemes offer a promising direction for both short- and medium-term cryptographic resilience.

To proactively address the challenges posed by quantum computing, the National Institute of Standards and Technology (NIST) launched a standardization initiative in 2016 aimed at identifying suitable post-quantum cryptographic algorithms. Among the selected candidates in the third round were four algorithms, including the lattice-based CRYSTALS-Kyber and CRYSTALS-Dilithium. These algorithms are currently being standardized and continue to undergo improvements to address implementation challenges, such as resistance to side-channel attacks and optimization of performance metrics.

Quantum computing represents a cutting-edge convergence of physics and computer science. This computational model was initially proposed by Paul Benioff in 1980, when he applied Schrödinger's equation to describe a quantum version of the classical Turing machine [2]. Quantum systems process information by leveraging the principles of quantum mechanics—most notably, superposition and entanglement—through quantum bits, or qubits, which serve as the fundamental units of quantum computation [30].

One of the core areas of study in quantum computing revolves around qubit behavior, particularly superposition and quantum error correction. Superposition allows a

qubit to exist simultaneously in a linear combination of the classical binary states “0” and “1”. Unlike classical bits, qubits can represent probabilistic states, which are often visualized using the Bloch sphere—a geometrical representation where the poles denote the classical states “0” and “1”, and the position of a vector on the sphere indicates the superposed state.

The quantum state of a single qubit can be described as a linear combination of the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where α and β are complex probability amplitudes satisfying the normalization condition:

$$|\alpha|^2 + |\beta|^2 = 1.$$

These amplitudes represent the probabilities of measuring the qubit in state $|0\rangle$ or $|1\rangle$, respectively. Due to the physical nature of quantum systems, interactions with the environment introduce noise, making quantum states prone to errors. As a result, quantum error correction techniques are employed to preserve coherence and ensure reliable computation.

Conjectured security against quantum attacks

Most number-theoretic cryptography, such as the Diffie-Hellman protocol [6] and RSA cryptosystem [28], relies on the conjectured hardness of integer factorization or the discrete logarithm problem in certain groups. However, Shor [31] gave efficient quantum algorithms for all these problems, which would render number-theoretic systems insecure in a future where large-scale quantum computers are available. By contrast, no efficient quantum algorithms are known for the problems typically used in lattice cryptography; indeed, generic (and relatively modest) quantum speedups provide the only known advantage over non-quantum algorithms.

Algorithmic simplicity, efficiency, and parallelism

Lattice-based cryptosystems are often algorithmically simple and highly parallelizable, consisting mainly of linear operations on vectors and matrices modulo relatively small integers. Moreover, constructions based on “algebraic” lattices over certain rings (e.g., the NTRU cryptosystem [15]) can be especially efficient, and in some cases even outperform more traditional systems by a significant margin.

Strong security guarantees from worst-case hardness

Cryptography fundamentally relies on average-case intractability—that is, problems for which randomly generated instances (drawn from a specified probability distribution) are hard to solve. This contrasts with the worst-case hardness typically studied in algorithm theory and NP-completeness, where a problem is considered hard if it contains some difficult instances. Problems that are hard in the worst case often become easier on average, especially when the instance distribution exhibits extra structure, such as a secret key used for decryption.

In a seminal work, Ajtai [1] gave a remarkable 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. 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.

Constructions of Versatile and Powerful Cryptographic Objects

Cryptography has evolved far beyond its original purpose of securing secret communication. Today, it encompasses a broad range of goals related to secure computation and interaction in adversarial settings. One notable advancement is fully homomorphic encryption (FHE), which enables arbitrary computations on encrypted data without revealing the underlying plaintext. Originally proposed as a theoretical idea by Rivest et al. [27], FHE remained unrealized for decades until Gentry [12] introduced the first practical construction, grounded in lattice-based assumptions.

Lattices have since become central to achieving advanced cryptographic functionalities. They underpin the only known constructions of powerful primitives such as attribute-based encryption for complex access policies [11, 14] and general-purpose program obfuscation [3]. These developments highlight the unique versatility and strength of lattice-based cryptography in building next-generation secure systems.

Tweaking in cryptography refers to making small, targeted modifications to existing schemes to improve performance, security, or adaptability. In my work, Kyber (ML-KEM) was tweaked by modifying matrix generation and compression routines for better control and potential hardware optimization. Dilithium (ML-DSA) was tweaked by adjusting polynomial sampling and encoding steps to suit specific implementation needs.

Objectives and Contributions

This thesis aims to investigate, design, and evaluate modifications to Kyber and Dilithium that are specifically tailored to the needs of high-assurance government-scale deployment. The main objectives are as follows:

- To explore the structure and behavior of the NIST-approved ML-KEM (Kyber) and ML-DSA (Dilithium) schemes, with emphasis on their reference implementations.
- To identify performance bottlenecks, parameter tuning opportunities, and transform optimizations suitable for government-specific use cases.
- To apply targeted tweaks to the internal components (e.g., polynomials, NTTs, matrix generation) of Kyber and Dilithium and validate them through reference implementations.
- To evaluate the security and performance trade-offs introduced by the proposed tweaks using benchmarks (e.g., cycle counts, latency) and qualitative security assessments.
- To compare the original and modified schemes in terms of deployment readiness, efficiency, and long-term viability for governmental applications.

The key contributions of this thesis include:

- A comprehensive review of ML-KEM and ML-DSA internals from a government deployment lens.
- Practical tweaks to Kyber and Dilithium, including parameter reconfiguration and structural code changes.
- Benchmark-driven performance comparisons of original vs. modified schemes.
- A discussion on the implications of these optimizations for future post-quantum standard adoption in governmental infrastructure.

Scope and Organization

This thesis focuses on two core average-case lattice problems—Learning With Errors (LWE) and Short Integer Solution (SIS)—which underpin the security of several post-quantum cryptographic schemes. Building on these foundations, the work explores the structure and implementation of lattice-based algorithms, namely ML-KEM (Kyber) and ML-DSA (Dilithium), with an emphasis on optimizing them for real-world, large-scale deployments, particularly in government infrastructure.

The structure of the thesis is outlined below:

Chapter 1: Introduction Introduces post-quantum cryptography, outlines the motivation, and defines the scope and objectives of the thesis.

Chapter 2: Background Discusses foundational lattice-based problems (LWE, SIS, MLWE, MSIS) and the evolution of PQC schemes, including a comparison between key encapsulation and signature algorithms.

Chapter 3: ML-KEM (Kyber) Explains the internal design, parameterization, and integration of Kyber with the Open Quantum Safe (OQS) library.

Chapter 4: ML-DSA (Dilithium) Describes the architecture, parameters, signing and verification mechanisms of Dilithium, and its implementation with OQS.

Chapter 5: Tweaks to Kyber Details the rationale, design, and implementation of algorithmic tweaks to Kyber. Includes code-level modifications and benchmarking results.

Chapter 6: Tweaks to Dilithium Presents modifications to Dilithium's polynomial and transformation layers, including performance and security assessments.

Chapter 7: Code Listings Provides the complete reference code for both original and modified versions of Kyber and Dilithium.

Chapter 8: Conclusion, Summary and Future Work Summarizes key findings, discusses practical implications for government use, and outlines future directions for enhancement and deployment.

Bibliography Lists all referenced works, including foundational papers and recent research in post-quantum cryptography.

Chapter 2

Background

2.1 Notation

For a real number $x \in \mathbb{R}$, we let $\lfloor x \rfloor$ denote the largest integer not greater than x , and $\lceil x \rceil := \lfloor x + 1/2 \rfloor$ denote the integer closest to x , with ties broken upward.

We use bold lower-case letters like \mathbf{x} to denote column vectors; for row vectors we use the transpose \mathbf{x}^t . We use bold upper-case letters like \mathbf{A} to denote matrices, and sometimes identify a matrix with its ordered set of column vectors. We denote the horizontal concatenation of vectors and/or matrices using a vertical bar, e.g., $[\mathbf{A} \mid \mathbf{A}\mathbf{x}]$. We sometimes apply functions entry-wise to vectors, e.g., $\lceil \mathbf{x} \rceil$ rounds each entry of \mathbf{x} to its nearest integer.

For a positive integer q , let $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ denote the quotient ring of integers modulo q , i.e., the collection of cosets $a + q\mathbb{Z}$ with the induced addition and multiplication operations. Similarly, because \mathbb{Z}_q is an additive group, it supports multiplication by integers, i.e., $z \cdot a \in \mathbb{Z}_q$ for an integer $z \in \mathbb{Z}$ and $a \in \mathbb{Z}_q$. We often write $z \bmod q$ to denote the coset $z + q\mathbb{Z}$, and $y = z \pmod{q}$ to denote that $y + q\mathbb{Z} = z + q\mathbb{Z}$.

We use standard asymptotic notation $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, etc. In addition, tildes (e.g., $\tilde{O}(\cdot)$) indicate that logarithmic factors in the main parameter are suppressed.

2.2 Lattices

This survey requires minimal knowledge of lattices beyond some basic definitions and computational problems, which we recall here. (See the other resources listed in the introduction for much more background.)

2.2.1 Basic Definitions

An n -dimensional lattice \mathcal{L} is any subset of \mathbb{R}^n that is both:

1. an additive subgroup: $\mathbf{0} \in \mathcal{L}$, and $-\mathbf{x}, \mathbf{x} + \mathbf{y} \in \mathcal{L}$ for every $\mathbf{x}, \mathbf{y} \in \mathcal{L}$; and
2. discrete: every $\mathbf{x} \in \mathcal{L}$ has a neighborhood in \mathbb{R}^n in which \mathbf{x} is the only lattice point.

Examples include the integer lattice \mathbb{Z}^n , the scaled lattice $c\mathcal{L}$ for any real number c and lattice \mathcal{L} , and the “checkerboard” lattice $\{\mathbf{x} \in \mathbb{Z}^n : \sum_i x_i \text{ is even}\}$.

The minimum distance of a lattice \mathcal{L} is the length of a shortest nonzero lattice vector:

$$\lambda_1(\mathcal{L}) := \min_{\mathbf{v} \in \mathcal{L} \setminus \{\mathbf{0}\}} \|\mathbf{v}\|.$$

(Unless otherwise specified, $\|\cdot\|$ denotes the Euclidean norm.) More generally, the i th successive minimum $\lambda_i(\mathcal{L})$ is the smallest r such that \mathcal{L} has i linearly independent vectors of norm at most r .

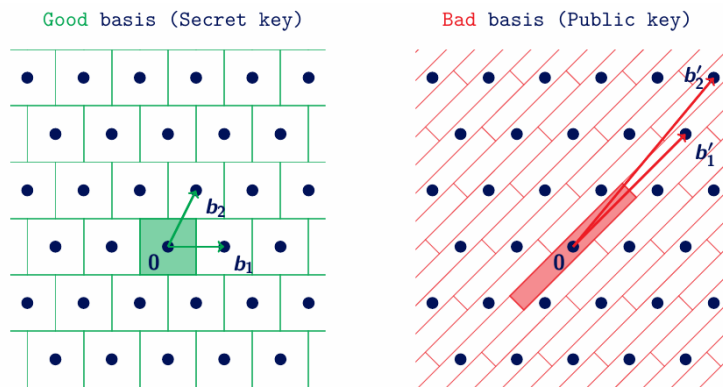


FIGURE 2.1: A good basis of a lattice is short and somewhat orthogonal, which allows rounding a point to a nearby lattice point successfully. A bad basis, with long and non-orthogonal vectors, does not support effective rounding.

As shown in Figure 2.1, the quality of a lattice basis significantly affects computational operations. A good basis facilitates algorithms such as lattice rounding and basis reduction due to its short and nearly orthogonal vectors. In contrast, a bad basis makes such operations inefficient, as the longer and skewed vectors lead to imprecise approximations and increased computational effort.

Because a lattice \mathcal{L} is an additive subgroup of \mathbb{R}^n , we have the quotient group $\mathbb{R}^n / \mathcal{L}$ of cosets

$$\mathbf{c} + \mathcal{L} = \{\mathbf{c} + \mathbf{v} : \mathbf{v} \in \mathcal{L}\}, \quad \mathbf{c} \in \mathbb{R}^n,$$

with the usual induced addition operation $(\mathbf{c}_1 + \mathcal{L}) + (\mathbf{c}_2 + \mathcal{L}) = (\mathbf{c}_1 + \mathbf{c}_2) + \mathcal{L}$. A fundamental domain of \mathcal{L} is a set $\mathcal{F} \subset \mathbb{R}^n$ that contains exactly one representative $\bar{\mathbf{c}} \in (\mathbf{c} + \mathcal{L}) \cap \mathcal{F}$ of every coset $\mathbf{c} + \mathcal{L}$. For example, the half-open intervals $[0, 1)$

and $[-\frac{1}{2}, \frac{1}{2})$ are fundamental domains of the integer lattice \mathbb{Z} , where coset $c + \mathbb{Z}$ has representative $c - \lfloor c \rfloor$ and $c - \lceil c \rceil$, respectively.

Bases and fundamental parallelepipeds. Although every (non-trivial) lattice \mathcal{L} is infinite, it is always finitely generated as the integer linear combinations of some linearly independent basis vectors $\mathbf{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_k\}$:

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) := \mathbf{B} \cdot \mathbb{Z}^k = \left\{ \sum_{i=1}^k z_i \mathbf{b}_i : z_i \in \mathbb{Z} \right\}.$$

The integer k is called the rank of the basis, and is an invariant of the lattice. For the remainder of this survey we restrict our attention to full-rank lattices, where $k = n$. A lattice basis \mathbf{B} is not unique: for any unimodular matrix $\mathbf{U} \in \mathbb{Z}^{n \times n}$ (i.e., one having determinant ± 1), $\mathbf{B} \cdot \mathbf{U}$ is also a basis of $\mathcal{L}(\mathbf{B})$, because $\mathbf{U} \cdot \mathbb{Z}^n = \mathbb{Z}^n$.

For a lattice \mathcal{L} having basis \mathbf{B} , a commonly used fundamental domain is the origin-centered fundamental parallelepiped $\mathcal{P}(\mathbf{B}) := \mathbf{B} \cdot [-\frac{1}{2}, \frac{1}{2})^n$, where coset $c + \mathcal{L}$ has representative $c - \mathbf{B} \cdot \lfloor \mathbf{B}^{-1} \cdot c \rfloor$.

The dual lattice. The dual (sometimes called reciprocal) of a lattice $\mathcal{L} \subset \mathbb{R}^n$ is defined as

$$\mathcal{L}^* := \{\mathbf{w} : \langle \mathbf{w}, \mathcal{L} \rangle \subseteq \mathbb{Z}\},$$

i.e., the set of points whose inner products with the vectors in \mathcal{L} are all integers. It is straightforward to verify that \mathcal{L}^* is a lattice. For example, $(\mathbb{Z}^n)^* = \mathbb{Z}^n$, and $(c\mathcal{L})^* = c^{-1}\mathcal{L}^*$ for any nonzero real c and lattice \mathcal{L} . It is also easy to verify that if \mathbf{B} is a basis of \mathcal{L} , then $\mathbf{B}^{-t} := (\mathbf{B}^t)^{-1} = (\mathbf{B}^{-1})^t$ is a basis of \mathcal{L}^* .

2.2.2 Computational Problems

We now define some of the computational problems on lattices that have been most useful in cryptography, and recall some results about their complexity. (There are many other problems that have been extensively studied in mathematics and computational complexity, but so far have had less direct importance to cryptography.)

Perhaps the most well-studied computational problem on lattices is the shortest vector problem:

Definition 1 (Shortest Vector Problem (SVP)) *Given an arbitrary basis \mathbf{B} of some lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, find a shortest nonzero lattice vector, i.e., a $\mathbf{v} \in \mathcal{L}$ for which $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$.*

Particularly important to lattice cryptography are approximation problems, which are parameterized by an approximation factor $\gamma \geq 1$ that is typically taken to be a function of the lattice dimension n , i.e., $\gamma = \gamma(n)$. For example, the approximation version of SVP is as follows (note that by setting $\gamma(n) = 1$ we recover the problem defined above):

Definition 2 (Approximate Shortest Vector Problem (SVP $_\gamma$)) *Given a basis \mathbf{B} of an n -dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, find a nonzero vector $\mathbf{v} \in \mathcal{L}$ for which $\|\mathbf{v}\| \leq \gamma(n) \cdot \lambda_1(\mathcal{L})$.*

As described in later sections, several cryptosystems can be proved secure assuming the hardness of certain lattice problems, in the worst case. However, to date no such proof is known for the search version of SVP $_\gamma$. Instead, there are proofs based on the following decision version of approximate-SVP, as well as a search problem related to the n th successive minimum:

Definition 3 (Decisional Approximate SVP (GapSVP $_\gamma$)) *Given a basis \mathbf{B} of an n -dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$ where either $\lambda_1(\mathcal{L}) \leq 1$ or $\lambda_1(\mathcal{L}) > \gamma(n)$, determine which is the case.*

Definition 4 (Approximate Shortest Independent Vectors Problem (SIVP $_\gamma$)) *Given a basis \mathbf{B} of a full-rank n -dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, output a set $S = \{\mathbf{s}_i\} \subset \mathcal{L}$ of n linearly independent lattice vectors where $\|\mathbf{s}_i\| \leq \gamma(n) \cdot \lambda_n(\mathcal{L})$ for all i .*

A final important problem for cryptography is the following bounded-distance decoding BDD $_\gamma$ problem, which asks to find the lattice vector that is closest to a given target point $\mathbf{t} \in \mathbb{R}^n$, where the target is promised to be “rather close” to the lattice. This promise, and the uniqueness of the solution, are what distinguish BDD $_\gamma$ from the approximate closest vector problem CVP $_\gamma$, wherein the target can be an arbitrary point. (Because no cryptosystem has yet been proved secure based on CVP $_\gamma$, we do not formally define that problem here.)

Definition 5 (Bounded Distance Decoding Problem (BDD $_\gamma$)) *Given a basis \mathbf{B} of an n -dimensional lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$ and a target point $\mathbf{t} \in \mathbb{R}^n$ with the guarantee that $\text{dist}(\mathbf{t}, \mathcal{L}) < d = \lambda_1(\mathcal{L})/(2\gamma(n))$, find the unique lattice vector $\mathbf{v} \in \mathcal{L}$ such that $\|\mathbf{t} - \mathbf{v}\| < d$.*

Algorithms and complexity. The above lattice problems have been intensively studied and appear to be intractable, except for very large approximation factors. Known polynomial-time algorithms like the one of Lenstra, Lenstra, and Lovász [18] and its descendants (e.g., [29]) obtain only slightly subexponential approximation factors $\gamma = 2^{\Theta(n \log \log n / \log n)}$ for all the above problems (among many others that are

less relevant to cryptography). Known algorithms that obtain polynomial $\text{poly}(n)$ or better approximation factors either require super-exponential $2^{\Theta(n \log n)}$ time, or exponential $2^{\Theta(n)}$ time and space. There are also time-approximation tradeoffs that interpolate between these two classes of results, to obtain $\gamma = 2^k$ approximation factors in $2^{\tilde{\Theta}(n/k)}$ time. Importantly, the above also represents the state of the art for quantum algorithms, though in some cases the hidden constant factors in the exponents are somewhat smaller. By contrast, recall that the integer factorization and discrete logarithm problem (in essentially any group) can be solved in polynomial time using Shor's quantum algorithm [32].

2.3 LWE

2.3.1 Introduction

In this subsection, we will explain at a high level about Learning With Errors (LWE), a lattice-based cryptography framework. To keep things simple, parameters are not yet instantiated and certain distributions are not yet made explicit. Later on, whenever we discuss the specific variants of LWE, we will focus more on the precise instantiations and also focus on what instantiations are used in the NIST candidates.

2.3.2 The LWE setup

Learning with errors always works with modular arithmetic modulo some number $q \in \mathbb{N}_{>1}$, which is called the modulus and is often chosen to be prime or a prime power.

The secret key s in LWE is a vector $s \in (\mathbb{Z}/q\mathbb{Z})^n$, i.e., $s_i \in \mathbb{Z}/q\mathbb{Z}$ for $i \in \{1, \dots, n\}$. This secret vector s is sampled uniformly beforehand.

The public information consists of m slightly perturbed random inner products with s . That is, the public information consists of m pairs of the shape (a, b) such that $b = a \cdot s + e \in \mathbb{Z}/q\mathbb{Z}$, where, in each pair (a, b) , the vector a is drawn uniformly random from $(\mathbb{Z}/q\mathbb{Z})^n$ and e is drawn from a specific distribution over $\mathbb{Z}/q\mathbb{Z}$ close to zero. For intuition, it suffices to think of e to be uniformly drawn from $\{-2, -1, 0, 1, 2\} \pmod{q}$ for each pair. The dot in $a \cdot s$ denotes the inner product of the two vectors in $(\mathbb{Z}/q\mathbb{Z})^n$ and results in a number in $\mathbb{Z}/q\mathbb{Z}$.

For each of the m public pairs (a, b) , the associated errors e are unknown to the public. Also, this error is sampled again for each pair. Both these facts are important to understand the hardness of the LWE assumption.

2.3.3 The LWE assumption

Recall that the public observer in the LWE setup gets to know m pairs of the shape (a, b) , where $a \in (\mathbb{Z}/q\mathbb{Z})^n$ and $b \in \mathbb{Z}/q\mathbb{Z}$. Here, the public observer knows that there exists some secret s such that $b = a \cdot s + e$, but does not know the value of s .

The search-LWE assumption

The search-LWE assumption can be phrased as follows: given m such pairs of the shape (a, b) as in the LWE setup, it is hard to find the secret s .

Different authors have different assumptions on this particular hardness, and this hardness also relies on the various parameters of the LWE setup: the modulus q , the dimension n , the number of samples m and the error distribution of the e .

The distinguishing-LWE assumption

The distinguishing-LWE assumption essentially states that the m samples of the shape (a, b) look very much like uniform samples. In other words, this assumption states that it is hard to algorithmically distinguish between m samples (a, b) from the LWE setup, and m uniform samples (\tilde{a}, \tilde{b}) , i.e., where \tilde{a} is uniformly random distributed over $(\mathbb{Z}/q\mathbb{Z})^n$ and \tilde{b} is uniformly random distributed over $\mathbb{Z}/q\mathbb{Z}$ (and where each of the m uniform samples are independent of each other).

As illustrated in Figure 2.2, there exists a sequence of reductions among structured lattice problems that are crucial for constructing efficient and secure lattice-based cryptographic schemes. In particular, the reductions from Ideal-SVP and NTRU to Ring-LWE and further to module-SVP highlight the foundational hardness assumptions on which modern post-quantum schemes like Kyber and Dilithium are based. These reductions justify the use of structured lattices without significantly compromising security, while enabling better efficiency and compact representations.

Difference between the two assumptions

In the search-LWE assumption the public observer is promised that the samples are from the LWE setup. That means that the public observer is given the (true) promise that the samples (a, b) are of the shape $b = a \cdot s + e$ for some secret s and some given distribution of the errors e .

In the decision-LWE assumption the public observer is given no such promise. In fact, deciding whether there exists such secret s (in the LWE setup) or not (in the uniform setup) is the computational task that is assumed to be hard.

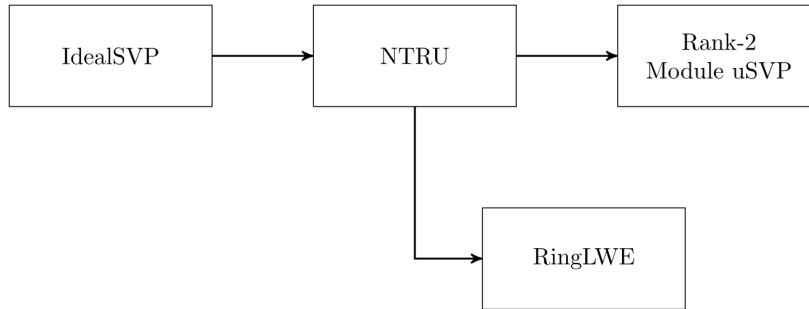


FIGURE 2.2: A simplified diagram showing the reductions between Ideal-SVP, NTRU, Ring-LWE, and unique module-SVP problems. These structured problems underpin the hardness assumptions of lattice-based cryptography.

Regev showed that these two assumptions are essentially equivalent, whenever q is bounded by a polynomial in n [25, 24].

2.4 (Discrete) Gaussians and Subgaussians

Many modern works on lattices in complexity and cryptography rely on Gaussian-like probability distributions over lattices, called discrete Gaussians. Here we recall the relevant definitions.

Gaussians. For any positive integer n and real $s > 0$, which is taken to be $s = 1$ when omitted, define the Gaussian function $\rho_s : \mathbb{R}^n \rightarrow \mathbb{R}_+$ of parameter (or width) s as

$$\rho_s(\mathbf{x}) := \exp(-\pi\|\mathbf{x}\|^2/s^2) = \rho(\mathbf{x}/s).$$

Notice that ρ_s is invariant under rotations of \mathbb{R}^n , and that $\rho_s(\mathbf{x}) = \prod_{i=1}^n \rho_s(x_i)$.

The (continuous) Gaussian distribution D_s of parameter s over \mathbb{R}^n is defined to have probability density function proportional to ρ_s , i.e.,

$$f(\mathbf{x}) := \rho_s(\mathbf{x}) / \int_{\mathbb{R}^n} \rho_s(\mathbf{z}) d\mathbf{z} = \rho_s(\mathbf{x}) / s^n.$$

For a lattice coset $\mathbf{c} + \mathcal{L} \subset \mathbb{R}^n$ and parameter $s > 0$, the discrete Gaussian probability distribution $D_{\mathbf{c}+\mathcal{L},s}$ is simply the Gaussian distribution restricted to the coset:

$$D_{\mathbf{c}+\mathcal{L},s}(\mathbf{x}) \propto \begin{cases} \rho_s(\mathbf{x}) & \text{if } \mathbf{x} \in \mathbf{c} + \mathcal{L} \\ 0 & \text{otherwise.} \end{cases}$$

Smoothing parameter. Micciancio and Regev [21] introduced a very important quantity called the smoothing parameter of a lattice \mathcal{L} . Informally, this is the amount of Gaussian “blur” required to “smooth out” essentially all the discrete structure of \mathcal{L} . Alternatively, it can be seen as the smallest width $s > 0$ such that every coset $\mathbf{c} + \mathcal{L}$ has nearly the same Gaussian mass $\rho_s(\mathbf{c} + \mathcal{L}) := \sum_{\mathbf{x} \in \mathbf{c} + \mathcal{L}} \rho_s(\mathbf{x})$, up to some small relative error.

Formally, the smoothing parameter $\eta_\varepsilon(\mathcal{L})$ is parameterized by a tolerance $\varepsilon > 0$, and is defined using the dual lattice as the minimal $s > 0$ such that $\rho_{1/s}(\mathcal{L}^*) \leq 1 + \varepsilon$. This condition can be used to formalize and prove the above-described “smoothing” properties. For the purposes of this survey, we often omit ε and implicitly take it to be very small, e.g., a negligible $n^{-\omega(1)}$ function in the dimension n of the lattice.

The smoothing parameter is closely related to other standard lattice quantities. For example:

Theorem 1 For any full-rank lattice $\mathcal{L} \subseteq \mathbb{R}^n$, we have $\eta_{2^{-n}}(\mathcal{L}) \leq \sqrt{n}/\lambda_1(\mathcal{L}^*)$.

Theorem 2 For any full-rank lattice $\mathcal{L} \subseteq \mathbb{R}^n$ and $\varepsilon \in (0, 1/2)$,

$$\eta_\varepsilon(\mathcal{L}) \leq \min_{\text{basis } \mathbf{B} \text{ of } \mathcal{L}} \|\tilde{\mathbf{B}}\| \sqrt{\log O(n/\varepsilon)} \leq \lambda_n(\mathcal{L}) \sqrt{\log O(n/\varepsilon)},$$

where $\|\tilde{\mathbf{B}}\| = \max_i \|\tilde{\mathbf{b}}_i\|$ denotes the maximal length of the Gram-Schmidt orthogonalized vectors $\{\tilde{\mathbf{b}}_i\}$ of the ordered basis $\mathbf{B} = \{\mathbf{b}_i\}$.

Several works have shown that when $s \geq \eta(\mathcal{L})$, the discrete Gaussian distribution $D_{\mathbf{c} + \mathcal{L}, s}$ behaves very much like the continuous Gaussian D_s in many important respects. For example, their moments and tails are nearly the same, and the sum of independent discrete Gaussians is a discrete Gaussian.

Subgaussianity. Informally, a random variable (or its distribution) is subgaussian if it is dominated by a Gaussian. Formally, a real random variable X is subgaussian with parameter s if for every $t \geq 0$, we have

$$\Pr[|X| > t] \leq 2 \exp(-\pi t^2/s^2).$$

More generally, a random vector \mathbf{x} over \mathbb{R}^n is subgaussian with parameter s if every marginal $\langle \mathbf{x}, \mathbf{u} \rangle$ is, for all unit vectors $\mathbf{u} \in \mathbb{R}^n$. It is not hard to show that the concatenation of independent subgaussian random variables or vectors of common parameter s is itself a subgaussian vector of parameter s .

Examples of subgaussian distributions with parameter s include any symmetric random variable having magnitude bounded by $s/\sqrt{2\pi}$; the continuous Gaussian D_s and discrete Gaussian $D_{\mathcal{L},s}$ over any lattice \mathcal{L} ; and the discrete Gaussian $D_{\mathbf{c}+\mathcal{L},s}$ over any lattice coset when $s \geq \eta(\mathcal{L})$ (under a slight relaxation of subgaussianity; Section 2.4).

2.5 Cryptographic Background

Cryptography is concerned with a variety of different kinds of objects and security properties they can satisfy. Here we give a brief, informal overview of the main concepts that are relevant to this survey. For further details and formalisms, see, e.g., [16, 13].

In complexity-theoretic (as opposed to information-theoretic) cryptography, the security parameter λ regulates the running times of all algorithms, including the attacker, along with the latter's measure of success, called its advantage. A typical requirement is that all algorithms (including the attacker) have running times that are polynomial $\lambda^{O(1)}$ in the security parameter, and that the attacker's advantage is negligible $\lambda^{-\omega(1)}$, i.e., asymptotically smaller than the inverse of any polynomial in λ . (One can be even more permissive about the adversary, e.g., allowing its running time and/or inverse advantage to be subexponential in λ , allowing it to be non-uniform, etc.) In what follows, all function families are implicitly indexed by λ , and all algorithms (including the attacker) are given λ as an input.

2.5.1 Function Families and Security Properties

A function family is a function $f : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ for some space \mathcal{K} of keys, domain \mathcal{X} , and range \mathcal{Y} . We call it a family because it defines the set of functions $f_k(\cdot) = f(k, \cdot)$ for keys $k \in \mathcal{K}$. If $|\mathcal{X}| > |\mathcal{Y}|$, i.e., the function “compresses” its input by some amount, it is often called a hash function. The following are two commonly used security properties of function families:

- A family f is *one way* if it is “hard to invert” for random inputs. More precisely, given $k \in \mathcal{K}$ and $y = f_k(x) \in \mathcal{Y}$ where k, x are randomly chosen from prescribed distributions, it is infeasible to find any preimage $x' \in f_k^{-1}(y)$.
- A family f is *collision resistant* if it is hard to find a collision for a random key. More precisely, given a random $k \in \mathcal{K}$ (chosen from a prescribed distribution), it is infeasible to find distinct $x, x' \in \mathcal{X}$ such that $f_k(x) = f_k(x')$.

2.5.2 Public-Key Encryption

An asymmetric (also known as public-key) encryption scheme is a triple of randomized algorithms having the following interfaces:

- The key generator, given the security parameter, outputs a public key and secret key.
- The encryption algorithm takes a public key and a message (from some known set of valid messages) and outputs a ciphertext.
- The decryption algorithm takes a secret key and a ciphertext, and outputs either a message or a distinguished “failure” symbol.

Naturally, the scheme is said to be correct if generating a key pair, then encrypting a valid message using the public key, then decrypting the resulting ciphertext using the secret key, yields the original message (perhaps with all but negligible probability).

A standard notion of security, called *semantic security*, or *indistinguishability under chosen-plaintext attack* (IND-CPA), informally guarantees that encryption reveals nothing about encrypted messages to a passive (eavesdropping) adversary. Formally, the definition considers the following experiment, which is parameterized by a bit $b \in \{0, 1\}$:

1. Generate a public/secret key pair, and give the public key to the attacker, who must reply with two valid messages m_0, m_1 . (If the valid message space is just $\{0, 1\}$, then we may assume that $m_b = b$ without loss of generality.)
2. Encrypt m_b using the public key and give the resulting “challenge ciphertext” to the attacker.
3. Finally, the attacker either accepts or rejects.

An encryption scheme is said to be semantically (or IND-CPA) secure if it is infeasible for an attacker to distinguish between the two cases $b = 0$ and $b = 1$. That is, its probabilities of accepting in the two cases should differ by only a negligible amount.

A much stronger notion of security, called *active security*—or more formally, *indistinguishability under chosen-ciphertext attack* (IND-CCA)—augments the above experiment by giving the attacker access to a decryption oracle, i.e., one that runs the decryption algorithm (with the secret key) on any ciphertext the attacker may query, except for the challenge ciphertext. (This restriction is of course necessary, because otherwise the attacker could just request to decrypt the challenge ciphertext and thereby learn the value of b .) The scheme is said to be actively (or IND-CCA)

secure if it is infeasible for an attacker to distinguish between the two cases $b = 0$ and $b = 1$.

2.5.3 Richer Forms of Encryption

Over the past several years, there has been an increasing interest in encryption systems having additional useful features. For example, *homomorphic encryption* allows an untrusted worker to perform meaningful computations on encrypted data, without revealing anything about the data to the worker. In this context, the basic syntax and notion of IND-CPA security remain exactly the same, but there is an additional algorithm for performing a desired homomorphic computation on ciphertexts.

Another example is *identity-based encryption* (IBE), in which any string (e.g., a user's email address) can serve as a public key. Here the model is slightly different: an IBE is a four-tuple of randomized algorithms having the following interfaces:

- The setup algorithm, given the security parameter, outputs a “master” public and secret key pair.
- The key extraction algorithm takes a master secret key and an identity string, and outputs a secret key for that particular identity.
- The encryption algorithm takes a master public key, an identity string, and a valid message, and outputs a ciphertext.
- The decryption algorithm takes an identity secret key and a ciphertext, and outputs a message (or a failure symbol).

Correctness is defined in the expected way: for a ciphertext encrypted to a particular identity string, decrypting using a corresponding secret key (produced by the key extraction algorithm) should return the original message.

Informally, semantic security for IBE is defined by modifying the standard IND-CPA experiment to model the property that even if many users collude by combining their secret keys, they still learn nothing about messages encrypted to another user. More formally, we consider the following experiment: the attacker is given the master public key, along with oracle access to the key extraction algorithm (with the master secret key built in), thus allowing it to obtain secret keys for any identities of its choice. At some point, the attacker produces two messages m_0, m_1 and a target identity, which must be different from all the queries it ever makes to its key-extraction oracle. The attacker is then given a ciphertext that encrypts m_b to the

target identity, and may make further queries to its oracle before either accepting or rejecting.

Chapter 3

ML-KEM: KYBER

3.1 Overview of CRYSTAL-KYBER

Over the past several years, there has been steady progress toward building quantum computers. If large-scale quantum computers are realized, the security of many commonly used public-key cryptosystems will be at risk. This would include key-establishment schemes and digital signature schemes whose security depends on the difficulty of solving the integer factorization and discrete logarithm problems (both over finite fields and elliptic curves). As a result, in 2016, NIST initiated a public Post-Quantum Cryptography (PQC) Standardization process to select quantum-resistant public-key cryptographic algorithms. A total of 82 candidate algorithms were submitted to NIST for consideration. After three rounds of evaluation and analysis, NIST selected the first four algorithms for standardization. These algorithms are intended to protect sensitive information well into the foreseeable future, including after the advent of cryptographically-relevant quantum computers. Here we will be studying a variant of the algorithm CRYSTALS-KYBER, a lattice-based key-encapsulation mechanism (KEM) designed by Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding.

3.1.1 Key-Encapsulation Mechanisms

The following is a high-level overview of key-encapsulation mechanisms (KEMs). A KEM is a cryptographic scheme that, under certain conditions, can be used to establish a shared secret key between two communicating parties. This shared secret key can then be used for symmetric-key cryptography. A KEM consists of three algorithms and a collection of parameter sets. The three algorithms are:

- A probabilistic key generation algorithm denoted by KeyGen

- A probabilistic "encapsulation" algorithm denoted by Encaps
- A deterministic "decapsulation" algorithm denoted by Decaps

Figure 3.1 illustrates the fundamental concept of a Key Encapsulation Mechanism (KEM), which is a central component in post-quantum secure key exchange protocols. In a KEM, a shared secret key is established between a sender and a receiver without prior key sharing. The encapsulation process securely transmits a ciphertext from the sender, and the receiver uses a private key to decapsulate it and retrieve the shared key. This mechanism ensures confidentiality and is widely used in schemes like ML-KEM (Kyber).

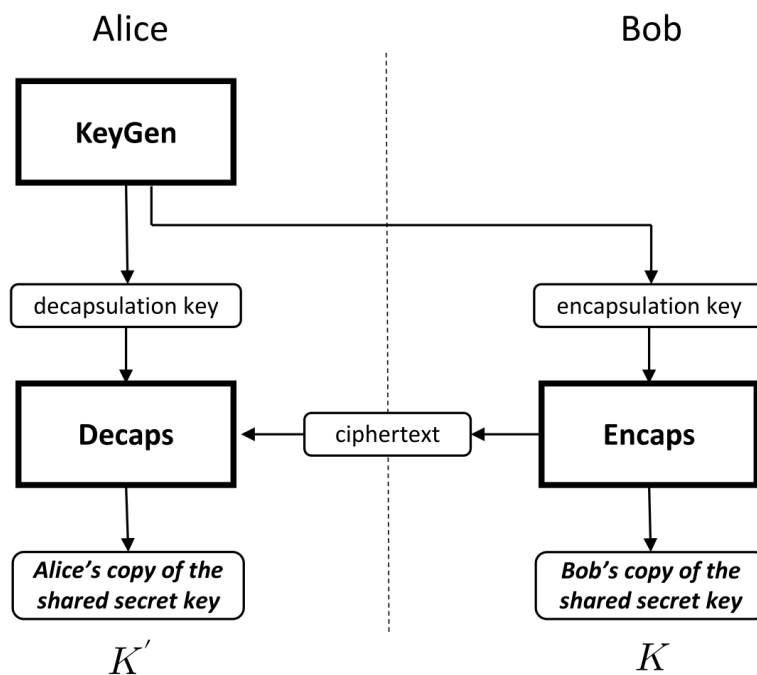


FIGURE 3.1: Overview of Key Encapsulation Mechanism (KEM) used in secure key exchange protocols.

The collection of parameter sets is used to select a trade-off between security and efficiency. Each parameter set in the collection is a list of specific (typically numerical) values, one for each parameter required by the three algorithms.

In the typical application, a KEM is used to establish a shared secret key between two parties (here referred to as Alice and Bob) as described in Figure 1. Alice begins by running KeyGen in order to generate a (public) encapsulation key and a (private) decapsulation key. Upon obtaining Alice's encapsulation key, Bob runs the Encaps algorithm, which produces Bob's copy K of the shared secret key along with an associated ciphertext. Bob sends the ciphertext to Alice, and Alice completes the process by running the Decaps algorithm using her decapsulation key and the

ciphertext. This final step produces Alice’s copy K' of the shared secret key. After completing this process, Alice and Bob would like to conclude that their outputs satisfy $K = K'$ and that this value is a secure, random, shared secret key. However, these properties only hold if certain important conditions are satisfied, as discussed in SP 800-227.

3.1.2 The KYBER Scheme

KYBER is a key-encapsulation mechanism that was initially described in [4]. The following is a brief and informal description of the computational assumption underlying KYBER and how the KYBER scheme is constructed.

The Computational Assumption

The security of KYBER is based on the presumed hardness of the so-called Module Learning with Errors (MLWE) problem [17], which is a generalization of the Learning With Errors (LWE) problem introduced by Regev in 2005 [26]. The hardness of the MLWE problem is itself based on the presumed hardness of certain computational problems in module lattices.

In the LWE problem, the input is a set of random “noisy” linear equations in some secret variables $x \in \mathbb{Z}_q^n$, and the task is to recover x . The noise in the equations is such that standard algorithms (e.g., Gaussian elimination) are intractable. The LWE problem naturally lends itself to cryptographic applications. For example, if x is interpreted as a secret key, then one can encrypt a one-bit plaintext value by sampling either an approximately correct linear equation (if the plaintext is zero) or a far-from-correct linear equation (if the plaintext is one). Plausibly, only a party in possession of x can distinguish these two cases. Encryption can then be delegated to another party by publishing a large collection of noisy linear equations, which can be combined appropriately by the encrypting party. The result is an asymmetric encryption scheme.

The MLWE problem is similar to the LWE problem. An important difference is that, in MLWE, \mathbb{Z}_q^n is replaced by a certain module R_q^k , which is constructed by taking the k -fold Cartesian product of a certain polynomial ring R_q . In particular, the secret in the MLWE problem is an element x of the module R_q^k . The ring R_q is discussed in detail in Section 4.3.

The KYBER Construction

At a high level, the construction of the scheme KYBER proceeds in two steps. First, the ideas discussed previously are used to construct a public-key encryption (PKE) scheme from the MLWE problem. Second, this PKE scheme is converted into a key-encapsulation mechanism using the so-called Fujisaki-Okamoto (FO) transform [10]. Due to certain properties of the FO transform, the resulting KEM provides security in a significantly more general attack model than the PKE scheme. As a result, KYBER is believed to satisfy so-called IND-CCA2 security.

A notable feature of KYBER is the use of the number-theoretic transform (NTT). The NTT converts a polynomial $f \in R_q$ to an alternative representation as a vector \mathbf{f} of linear polynomials. Working with NTT representations enables significantly faster multiplication of polynomials. Other operations (e.g., addition, rounding, and sampling) can be done in either representation.

KYBER satisfies the essential KEM property of correctness. This means that in the absence of corruption or interference, the process in Figure 1 will result in $K' = K$ with overwhelming probability. KYBER also comes with a proof of asymptotic theoretical security in a certain heuristic model. Each of the parameter sets of KYBER comes with an associated security strength that was estimated based on current cryptanalysis.

Parameter Sets and Algorithms

Recall that a KEM consists of algorithms `KeyGen`, `Encaps`, and `Decaps`, along with a collection of parameter sets. In the case of KYBER, the three aforementioned algorithms are:

1. `Kyber.CCAKEM.KeyGen`
2. `Kyber.CCAKEM.Encaps`
3. `Kyber.CCAKEM.Decaps`

These algorithms are described and discussed in detail in Section 3.

KYBER comes equipped with three parameter sets:

- KYBER-512 (security category 1)
- KYBER-768 (security category 3)
- KYBER-1024 (security category 5)

The security categories 1-5 are defined in SP 800-57. Each parameter set assigns a particular numerical value to five integer variables: $k, \eta_1, \eta_2, d_u, d_v$. In addition to these five variable parameters, there are also two constants: $n = 256$ and $q = 3329$.

We define three parameter sets for Kyber: `Kyber512`, `Kyber768`, and `Kyber1024`. These configurations are summarized in Table 3.1, alongside the derived parameter δ , which represents the failure probability during decapsulation of a valid `Kyber.CCAKEMciphertext`. The parameters are chosen to balance efficiency, security, and correctness.

	n	k	q	η_1	η_2	(d_u, d_v)	δ
<code>Kyber512</code>	256	2	3329	3	2	(10, 4)	2^{-139}
<code>Kyber768</code>	256	3	3329	2	2	(10, 4)	2^{-164}
<code>Kyber1024</code>	256	4	3329	2	2	(11, 5)	2^{-174}

TABLE 3.1: Kyber parameter sets

- The polynomial degree n is fixed at 256 to align with the goal of encapsulating keys with 256-bit entropy in `Kyber.CPAPKE.Enc`. Choosing smaller n would require encoding multiple key bits in a single coefficient, which reduces noise margins and compromises security. Larger values would complicate scaling and efficiency.
- The modulus $q = 3329$ is a small prime chosen such that $n \mid (q - 1)$, enabling efficient polynomial multiplication via the Number Theoretic Transform (NTT). Although smaller primes like 257 and 769 also meet this divisibility condition, they result in higher decryption failure rates, making them unsuitable for CCA-secure constructions.
- The parameter k determines the matrix dimension and serves as the primary knob for adjusting the security level. Each increase in k leads to improved security at the cost of greater computational and bandwidth requirements.
- Parameters $\eta_1, \eta_2, d_u,$ and d_v are carefully tuned to maintain a balance between security, ciphertext size, and decapsulation failure probability. All three parameter sets achieve a failure probability comfortably below 2^{-128} , which is considered negligible in practice.
- The parameter η_1 defines the noise of s and e in Algorithm 3 and of r in Algorithm 4. The parameter η_2 defines the noise of e_1 and e_2 in Algorithm 4.

3.2 Preliminaries and Notation

3.2.1 Bytes and Byte Arrays

All input and output parameters in Kyber’s API are represented as byte arrays. Let \mathbb{B} denote the set $\{0, 1, \dots, 255\}$, corresponding to all 8-bit unsigned integers (i.e., bytes). A byte array of fixed length k is denoted as \mathbb{B}^k , while \mathbb{B}^* refers to byte arrays of arbitrary length (or byte streams).

Given two byte arrays a and b , their concatenation is represented as $(a||b)$. For a byte array a , the notation $a + k$ refers to the subarray of a starting from the k^{th} byte, assuming zero-based indexing.

As an example, if a is a byte array of length ℓ and b is another byte array, then $c = (a||b)$ implies that $b = c + \ell$.

In scenarios where it is necessary to work with individual bits rather than bytes, we explicitly convert using a function `BytesToBits`, which transforms an input array of ℓ bytes into a bit array of length 8ℓ . The i^{th} bit β_i in the output is computed from the input byte array b using the formula:

$$\beta_i = \left(\frac{b_{\lfloor i/8 \rfloor}}{2^{(i \bmod 8)}} \right) \bmod 2.$$

3.2.2 Polynomial Rings and Vectors

We denote by R the ring $\mathbb{Z}[X]/(X^n + 1)$ and by R_q the ring $\mathbb{Z}_q[X]/(X^n + 1)$, where $n = 2n' - 1$ such that $X^n + 1$ is the $2n'$ -th cyclotomic polynomial. Throughout this document, the values of n , n' , and q are fixed to $n = 256$, $n' = 9$, and $q = 3329$. Regular font letters denote elements in R or R_q (which includes elements in \mathbb{Z} and \mathbb{Z}_q), and bold lower-case letters represent vectors with coefficients in R or R_q . By default, all vectors are column vectors. Bold upper-case letters denote matrices. For a vector v (or matrix A), we denote by v^T (or A^T) its transpose. For a vector v , we write $v[i]$ to denote its i -th entry (with indexing starting at zero); for a matrix A , we write $A[i][j]$ to denote the entry in row i , column j (again, with indexing starting at zero).

3.2.3 Modular Reductions

For an even (resp. odd) positive integer α , we define $r_0 = r \bmod \pm \alpha$ to be the unique element r_0 in the range $-\frac{\alpha}{2} < r_0 \leq \frac{\alpha}{2}$ (resp. $-\frac{\alpha-1}{2} \leq r_0 \leq \frac{\alpha-1}{2}$) such that $r = r_0 \bmod \alpha$. For any positive integer α , we define $r^+ = r \bmod \alpha$ to be the

unique element r^0 in the range $0 \leq r^0 < \alpha$ such that $r^0 = r \pmod{\alpha}$. When the exact representation is not important, we simply write $r \pmod{\alpha}$.

3.2.4 Rounding

For an element $x \in \mathbb{Q}$, we denote by $\lfloor x \rfloor$ rounding of x to the closest integer with ties being rounded up.

3.2.5 Sizes of Elements

For an element $w \in \mathbb{Z}_q$, we write $\|w\|_\infty$ to mean $|w \pmod{\pm q}|$. We now define the ℓ_∞ and ℓ_2 norms for $w = w_0 + w_1X + \cdots + w_{n-1}X^{n-1} \in R$:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \quad \|w\| = \sqrt{\|w_0\|_\infty^2 + \cdots + \|w_{n-1}\|_\infty^2}.$$

Similarly, for $w = (w_1, \dots, w_k) \in R^k$, we define:

$$\|w\|_\infty = \max_i \|w_i\|_\infty, \quad \|w\| = \sqrt{\|w_1\|^2 + \cdots + \|w_k\|^2}.$$

3.2.6 Sets and Distributions

For a set S , we write $s \leftarrow S$ to denote that s is chosen uniformly at random from S . If S is a probability distribution, then this denotes that s is chosen according to the distribution S .

3.2.7 Compression and Decompression.

Compression and decompression. Recall that $q = 3329$, and that the bit length of q is 12. For $d < 12$, define

$$\begin{aligned} \text{Compress}_d : \mathbb{Z}_q &\rightarrow \mathbb{Z}_{2^d} \\ x &\mapsto \left\lfloor \frac{2^d}{q} \cdot x \right\rfloor \pmod{2^d}. \end{aligned} \tag{4.7}$$

$$\begin{aligned} \text{Decompress}_d : \mathbb{Z}_{2^d} &\rightarrow \mathbb{Z}_q \\ y &\mapsto \left\lfloor \frac{q}{2^d} \cdot y \right\rfloor. \end{aligned} \tag{4.8}$$

The **Compress** and **Decompress** algorithms satisfy two important properties. First, decompression followed by compression preserves the input. That is,

$$\text{Compress}_d(\text{Decompress}_d(y)) = y$$

for all $y \in \mathbb{Z}_{2^d}$ and all $d < 12$. Second, if d is large (i.e., close to 12), compression followed by decompression does not significantly alter the value.

3.2.8 Symmetric Primitives

Kyber relies on several symmetric cryptographic primitives to ensure efficiency and security. It utilizes a pseudorandom function (PRF), defined as $\text{PRF} : \mathcal{B}^{32} \times \mathcal{B} \rightarrow \mathcal{B}^*$, and an extendable-output function (XOF), represented as $\text{XOF} : \mathcal{B}^* \times \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}^*$.

Additionally, Kyber employs two cryptographic hash functions:

- $H : \mathcal{B}^* \rightarrow \mathcal{B}^{32}$ – a standard hash outputting a 32-byte digest.
- $G : \mathcal{B}^* \rightarrow \mathcal{B}^{32} \times \mathcal{B}^{32}$ – a function that produces two 32-byte outputs, typically used in key material expansion.

It also incorporates a key derivation function (KDF), defined as $\text{KDF} : \mathcal{B}^* \rightarrow \mathcal{B}^*$, to derive cryptographic keys from shared secret material.

3.2.9 NTTs, Multiplication, and Bit-Reversed Order

An efficient approach for polynomial multiplication in R_q involves the use of the Number-Theoretic Transform (NTT), a finite field analogue of the Discrete Fourier Transform.

For the chosen prime modulus $q = 3329$, we observe that $q - 1 = 2^8 \cdot 13$, which means that the field \mathbb{Z}_q supports primitive 256-th roots of unity, but not 512-th roots. Consequently, the polynomial ring $R = \mathbb{Z}_q[X]/(X^{256} + 1)$ decomposes into 128 irreducible quadratic factors modulo q . Applying the NTT to a polynomial $f \in R_q$ effectively transforms it into a vector of 128 linear polynomials.

In typical in-place NTT implementations, the output is arranged in **bit-reversed order** to optimize memory access and compatibility with SIMD instructions (e.g., AVX). Specifically, let $\zeta = 17$ be a primitive 256-th root of unity in \mathbb{Z}_q . The set $\{\zeta, \zeta^3, \zeta^5, \dots, \zeta^{255}\}$ represents all such roots. The polynomial $X^{256} + 1$ can then be

factorized as:

$$\begin{aligned} X^{256} + 1 &= \prod_{i=0}^{127} (X^2 - \zeta^{2^{i+1}}) \\ &= \prod_{i=0}^{127} (X^2 - \zeta^{2^{\text{br}_7(i)+1}}), \end{aligned}$$

where $\text{br}_7(i)$ denotes the 7-bit bit-reversal of index i .

Using this factorization, the NTT of f is represented by the vector:

$$(f \bmod (X^2 - \zeta^{2^{\text{br}_7(0)+1}}), \dots, f \bmod (X^2 - \zeta^{2^{\text{br}_7(127)+1}})).$$

This vector is serialized into a canonical 256-dimensional vector over \mathbb{Z}_q . To facilitate efficient in-place operations without introducing new data types, we define the NTT as a bijective mapping:

$$\text{NTT} : R_q \rightarrow R_q,$$

which transforms a polynomial f into a new polynomial \hat{f} with reordered coefficients:

$$\text{NTT}(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \dots + \hat{f}_{255} X^{255},$$

where the transformed coefficients are computed as:

$$\begin{aligned} \hat{f}_{2i} &= \sum_{j=0}^{127} f_{2j} \cdot \zeta^{(2^{\text{br}_7(i)+1})j}, \\ \hat{f}_{2i+1} &= \sum_{j=0}^{127} f_{2j+1} \cdot \zeta^{(2^{\text{br}_7(i)+1})j}. \end{aligned}$$

3.2.10 Sampling from a binomial distribution.

Noise in Kyber is sampled from a centered binomial distribution \mathcal{B}_η for $\eta = 2$ or $\eta = 3$. We define \mathcal{B}_η as follows:

Sample $(a_1, \dots, a_\eta, b_1, \dots, b_\eta) \leftarrow \{0, 1\}^{2\eta}$ and output $\sum_{i=1}^\eta (a_i - b_i)$.

When we write that a polynomial $f \in R_q$ or a vector of such polynomials is sampled from \mathcal{B}_η , we mean that each coefficient is sampled from \mathcal{B}_η .

For the specification of Kyber we need to define how a polynomial $f \in R_q$ is sampled according to \mathcal{B}_η deterministically from 64η bytes of output of a pseudorandom function (we fix $n = 256$ in this description). This is done by the function CBD (for "centered binomial distribution") defined as follows:

Algorithm 1 $\text{CBD}_\eta : \mathbb{B}^{64\eta} \rightarrow R_q$

Require: Byte array $B = (b_0, b_1, \dots, b_{64\eta-1}) \in \mathbb{B}^{64\eta}$
Ensure: Polynomial $f \in R_q$

- 1: $(\beta_0, \dots, \beta_{512\eta-1}) := \text{BytesToBits}(B)$
 - 2: **for** i from 0 to 255 **do**
 - 3: $a := \sum_{j=0}^{\eta-1} \beta_{2i\eta+j}$
 - 4: $b := \sum_{j=0}^{\eta-1} \beta_{2i\eta+\eta+j}$
 - 5: $f_i := a - b$
 - 6: **end for**
 - 7: **return** $f_0 + f_1X + f_2X^2 + \dots + f_{255}X^{255}$
-

3.2.11 Sets and Distributions

Given a finite set S , the notation $s \leftarrow S$ indicates that the element s is sampled uniformly at random from S . In cases where S represents a probability distribution rather than a set, the same notation implies that s is drawn according to the distribution defined by S .

3.2.12 Encoding and Decoding

Kyber requires serialization of two primary data types: byte arrays and polynomials (including vectors of polynomials). Serialization of byte arrays is straightforward—they are stored as-is. The more critical aspect involves encoding and decoding polynomials to and from byte arrays.

To represent polynomials in a compact byte format, each coefficient must be mapped to a fixed number of bits. For a polynomial $f = f_0 + f_1X + \dots + f_{255}X^{255}$, where $n = 256$ and each $f_i \in \{0, \dots, 2^\ell - 1\}$, we use a byte array of size 32ℓ to store the entire polynomial.

The function `Decode` converts such a byte array into its corresponding polynomial form, reconstructing each coefficient from the appropriate bit segment. Conversely, `Encode` performs the inverse operation: it serializes the coefficients of the polynomial into a byte array format, suitable for transmission or storage.

Algorithmic details for both `Encode` and `Decode` are presented in the pseudocode in Algorithm 2.

Algorithm 2 Decodel: $\mathbb{B}^{32\ell} \rightarrow R_q$

```

1: Input: Byte array  $B \in \mathbb{B}^{32\ell}$ 
2: Output: Polynomial  $f \in R_q$ 
3:  $(\beta_0, \dots, \beta_{256\ell-1}) := \text{BytesToBits}(B)$ 
4: for  $i$  from 0 to 255 do
5:    $f_i := \sum_{j=0}^{\ell-1} \beta_{i\ell+j} 2^j$ 
6: end for
7: return  $f_0 + f_1X + f_2X^2 + \dots + f_{255}X^{255}$ 

```

3.3 Specification of KYBER.CPAPKE

KYBER.CPAPKE is similar to the LPR encryption scheme that was introduced (for Ring-LWE) by Lyuba-shevsky, Peikert, and Regev in the presentation of [20] at Eurocrypt 2010. KYBER.CPAPKE is parameterized by integers $n, k, q, \eta_1, \eta_2, d_u$, and d_v . As stated before, throughout this document n is always 256 and q is always 3329. Using the notations as described above, we give the definition of key generation, encryption, and decryption of the KYBER.CPAPKE public-key encryption scheme in Algorithms 3, 4, and 5.

Algorithm 3 Kyber.CPAPKE.KeyGen(): key generation**Ensure:** Secret key $sk \in \mathbb{B}^{12 \cdot k \cdot n/8}$ **Ensure:** Public key $pk \in \mathbb{B}^{12 \cdot k \cdot n/8 + 32}$

```

1:  $d \leftarrow \mathbb{B}^{32}$ 
2:  $(\rho, \sigma) := G(d)$ 
3:  $N := 0$  ▷ Generate matrix  $\hat{A} \in R_q^{k \times k}$  in NTT domain
4: for  $i$  from 0 to  $k - 1$  do
5:   for  $j$  from 0 to  $k - 1$  do
6:      $\hat{A}[i][j] := \text{Parse}(\text{XOF}(\rho, j, i))$ 
7:   end for
8: end for ▷ Sample  $s \in R_q^k$  from  $\mathcal{B}_{\eta_1}$ 
9: for  $i$  from 0 to  $k - 1$  do
10:   $s[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$ 
11:   $N := N + 1$ 
12: end for ▷ Sample  $e \in R_q^k$  from  $\mathcal{B}_{\eta_1}$ 
13: for  $i$  from 0 to  $k - 1$  do
14:   $e[i] := \text{CBD}_{\eta_1}(\text{PRF}(\sigma, N))$ 
15:   $N := N + 1$ 
16: end for
17:  $\hat{s} := \text{NTT}(s)$ 
18:  $\hat{e} := \text{NTT}(e)$ 
19:  $\hat{t} := \hat{A} \circ \hat{s} + \hat{e}$ 
20:  $pk := (\text{Encode}_{12}(\hat{t} \bmod q) \parallel \rho)$ 
21:  $sk := \text{Encode}_{12}(\hat{s} \bmod q)$ 
22: return  $(pk, sk)$ 

```

Algorithm 4 Kyber.CPAPKE.Enc(pk, m, r): encryption**Require:** Public key $pk \in \mathbb{B}^{12 \cdot k \cdot n/8 + 32}$ **Require:** Message $m \in \mathbb{B}^{32}$ **Require:** Random coins $r \in \mathbb{B}^{32}$ **Ensure:** Ciphertext $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$

```

1:  $N := 0$ 
2:  $\hat{t} := \text{Decode}_{12}(pk)$ 
3:  $\rho := pk + 12 \cdot k \cdot n/8$  ▷ Generate matrix  $\hat{A} \in R_q^{k \times k}$  in NTT domain
4: for  $i$  from 0 to  $k - 1$  do
5:   for  $j$  from 0 to  $k - 1$  do
6:      $\hat{A}^T[i][j] := \text{Parse}(\text{XOF}(\rho, i, j))$ 
7:   end for
8: end for ▷ Sample  $r \in R_q^k$  from  $\mathcal{B}_{\eta_1}$ 
9: for  $i$  from 0 to  $k - 1$  do
10:   $r[i] := \text{CBD}_{\eta_1}(\text{PRF}(r, N))$ 
11:   $N := N + 1$ 
12: end for ▷ Sample  $e_1 \in R_q^k$  from  $\mathcal{B}_{\eta_2}$ 
13: for  $i$  from 0 to  $k - 1$  do
14:   $e_1[i] := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ 
15:   $N := N + 1$ 
16: end for ▷ Sample  $e_2 \in R_q$  from  $\mathcal{B}_{\eta_2}$ 
17:  $e_2 := \text{CBD}_{\eta_2}(\text{PRF}(r, N))$ 
18:  $\hat{r} := \text{NTT}(r)$ 
19:  $u := \text{NTT}^{-1}(\hat{A}^T \circ \hat{r}) + e_1$ 
20:  $v := \text{NTT}^{-1}(\hat{t} \circ \hat{r}) + e_2 + \text{Decompress}_q(\text{Decode}_1(m), 1)$ 
21:  $c_1 := \text{Encode}_{d_u}(\text{Compress}_q(u, d_u))$ 
22:  $c_2 := \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))$ 
23: return  $c = (c_1 || c_2)$ 

```

Algorithm 5 Kyber.CPAPKE.Dec(sk, c): decryption

Require: Secret key $sk \in \mathbb{B}^{12 \cdot k \cdot n/8}$ **Require:** Ciphertext $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$ **Ensure:** Message $m \in \mathbb{B}^{32}$

- 1: $u := \text{Decompress}_q(\text{Decode}_{d_u}(c), d_u)$
 - 2: $v := \text{Decompress}_q(\text{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
 - 3: $\hat{s} := \text{Decode}_{12}(sk)$
 - 4: $m := \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{s}^T \circ \text{NTT}(u)), 1))$ ▷
 - $m := \text{Compress}_q(v - s^T u, 1)$
 - 5: **return** m
-

3.4 Specification of KYBER.CCAKEM

We construct the Kyber.CCAKEM IND-CCA2-secure KEM from the IND-CPA-secure public-key encryption scheme described in the previous subsection via a slightly tweaked Fujisaki–Okamoto transform. In Algorithms 6, 7, and 8 we define key generation, encapsulation, and decapsulation of Kyber.CCAKEM.

Algorithm 6 Kyber.CCAKEM.KeyGen()

Ensure: Public key $pk \in \mathbb{B}^{12 \cdot k \cdot n/8 + 32}$ **Ensure:** Secret key $sk \in \mathbb{B}^{24 \cdot k \cdot n/8 + 96}$

- 1: $z \leftarrow \mathbb{B}^{32}$
 - 2: $(pk, sk') := \text{Kyber.CPAPKE.KeyGen}()$
 - 3: $sk := (sk' || pk || H(pk) || z)$
 - 4: **return** (pk, sk)
-

Algorithm 7 Kyber.CCAKEM.Enc(pk)

Require: Public key $pk \in \mathbb{B}^{12 \cdot k \cdot n/8 + 32}$ **Ensure:** Ciphertext $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$ **Ensure:** Shared key $K \in \mathbb{B}^*$

- 1: $m \leftarrow \mathbb{B}^{32}$ ▷ Do not send output of system RNG
 - 2: $m := H(m)$
 - 3: $(\bar{K}, r) := G(m || H(pk))$
 - 4: $c := \text{Kyber.CPAPKE.Enc}(pk, m, r)$
 - 5: $K := \text{KDF}(\bar{K} || H(c))$
 - 6: **return** (c, K)
-

Algorithm 8 Kyber.CCAKEM.Dec(c, sk)

Require: Ciphertext $c \in \mathbb{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$ **Require:** Secret key $sk \in \mathbb{B}^{24 \cdot k \cdot n/8 + 96}$ **Ensure:** Shared key $K \in \mathbb{B}^*$

```

1:  $pk := sk + 12 \cdot k \cdot n/8$ 
2:  $h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathbb{B}^{32}$ 
3:  $z := sk + 24 \cdot k \cdot n/8 + 64$ 
4:  $m' := \text{Kyber.CPAPKE.Dec}(sk, c)$ 
5:  $(\bar{K}', \bar{r}') := G(m' || h)$ 
6:  $c' := \text{Kyber.CPAPKE.Enc}(pk, m', r')$ 
7: if  $c = c'$  then
8:   return  $K := \text{KDF}(\bar{K}' || H(c))$ 
9: else
10:  return  $K := \text{KDF}(z || H(c))$ 
11: end if

```

Instantiating PRF, XOF, H, G, and KDF What is still missing to complete the specification of Kyber is the instantiation of the symmetric primitives. We instantiate all of those primitives with functions from the FIPS-202 standard as follows:

- We instantiate XOF with SHAKE-128;
- we instantiate H with SHA3-256;
- we instantiate G with SHA3-512;
- we instantiate PRF(s, b) with SHAKE-256($s || b$); and
- we instantiate KDF with SHAKE-256.

"90s" variant of Kyber In the 90s variant of Kyber

- we instantiate XOF(ρ, i, j) with AES-256 in CTR mode, where ρ is used as the key and $i || j$ is zero-padded to a 12-byte nonce. The counter of CTR mode is initialized to zero.
- we instantiate H with SHA-256;
- we instantiate G with SHA-512;
- we instantiate PRF(s, b) with AES-256 in CTR mode, where s is used as the key and b is zero-padded to a 12-byte nonce. The counter of CTR mode is initialized to zero.

- we instantiate KDF with SHA-256.

3.5 Design rationale

The use of Module-LWE. Previous proposals of LWE-based cryptosystems either used the very structured Ring-LWE problem or standard LWE. The main advantage of structured LWE variants based on polynomial rings is efficiency in terms of both speed and key/ciphertext sizes. The disadvantages are concerns that the added structure might allow more efficient attacks, and that trade-offs between efficiency and security scale only coarsely. Standard LWE offers the benefit of no structure and easy scalability, but with a significant drop in efficiency. Module-LWE offers a balanced trade-off. Specifically, Kyber's Module-LWE parameters reduce structure compared to Ring-LWE, provide better scalability, and—for 256-bit messages—achieve performance close to Ring-LWE-based schemes.

Binomial noise. Theoretical treatments of LWE-based encryption typically assume Gaussian noise—either rounded or discrete. Early implementations also used discrete Gaussian noise, which is inefficient or timing-vulnerable. However, the effectiveness of known attacks on LWE does not depend on the exact noise distribution, but rather on its standard deviation and entropy. This allows use of noise distributions that are easy, fast, and secure to sample. A notable example is the centered binomial distribution. Another is "learning with rounding" (LWR), which adds uniform noise deterministically by dropping bits—e.g., Kyber's `Compressq` function. In Kyber, centered binomial noise is used, and LWE—not LWR—is chosen as the underlying problem.

Allowing decapsulation failures. Another notable design choice is whether to allow decapsulation failures (i.e., decryption failures in `Kyber.CPAPKE`) or aim for zero failure probability. Zero failures simplify CCA transforms and security proofs, and avoid potential attacks (see Subsection 5.5). However, achieving zero failure requires either reducing noise (and thus security) or increasing the lattice dimension (which impacts performance). Kyber's choice to allow small failure probabilities reflects the belief that

- decapsulation failures are a problem if they appear with non-negligible probability; but
- attacks attempting to exploit failures that occur with extremely low probability as in Kyber are a much smaller threat than, for example, improvements to hybrid attacks [56] targeting schemes with very low noise.

Different noise values η_1 and η_2 . Notice that in Algorithm 4, there is additional implicit noise created via $Compress_q$. This has the effect of adding some (deterministic) noise to the ciphertext, which can be interpreted as increasing the noise of the error polynomials e_1 and e_2 . If the decryption error probability is low enough, then it then makes sense to also increase the noise of the other secret terms (i.e. s , e , r) to be at a similar level as e_1 plus the deterministic noise. We utilize this idea (exclusively) for the Kyber512 parameter set. Relying on the rounding noise from $Compress_q$ to add error is akin to the LWR assumption. But unlike in (Ring/Module)-LWR schemes, where the security completely relies on the noise that's deterministically generated by rounding, our dependence on the deterministic noise is much smaller. First, it only adds 6 bits of Core-SVP hardness, and second, we are adding noise and rounding together, which presumably has less algebraic structure than just rounding. In short, without the LWR assumption, our parameter set for Kyber512 has 112 bits of core-SVP hardness – more specifically, the public keys are protected with 118 bits, and the ciphertexts with 112; with a weak version of the LWR assumption, it has 118-bit security everywhere.

3.6 Expected Security Strength

3.6.1 Security Definition

Kyber.CCAKEM (or short, Kyber) is an IND-CCA2-secure key encapsulation mechanism, i.e., it fulfills the security definition stated in Section 4.A.2 of the Call for Proposals.

3.6.2 Rationale of Our Security Estimates

Our estimates of the security strength for the three different parameter sets of Kyber—and consequently the classification into security levels as defined in Section 4.A.5 of the Call for Proposals—are based on the cost estimates of attacks against the underlying module-learning-with-errors (MLWE) problem.

To justify this rationale, we will in the following give two reductions from MLWE: a tight reduction in the random-oracle model (ROM) in Theorem 2 and a non-tight reduction in the quantum-random-oracle model (QROM) in Theorem 3. With those reductions at hand, there remain two avenues of attack that would break Kyber without solving the underlying MLWE problem, namely

1. breaking one of the assumptions of the reductions, in particular attacking the symmetric primitives used in Kyber; or

2. exploiting the non-tightness of the QROM reduction.

The discussion of 2.) requires considering two separate issues:

- a (quadratic) non-tightness in the decryption-failure probability of Kyber.CPAPKE, and
- a (quadratic) non-tightness between the advantage of the MLWE attacker and the quantum attacker against Kyber.

3.6.3 Security Assumption

The hard problem underlying the security of our schemes is Module-LWE. It consists in distinguishing uniform samples $(a_i, b_i) \leftarrow R_q^k \times R_q$ from samples $(a_i, b_i) \in R_q^k \times R_q$ where $a_i \leftarrow R_q^k$ is uniform and $b_i = a_i^T s + e_i$ with $s \leftarrow B_\eta^k$ common to all samples and $e_i \leftarrow B_\eta$ fresh for every sample.

Tight Reduction from MLWE in the ROM

Theorem 1. Suppose XOF and G are random oracles. For any adversary A, there exist adversaries B and C with roughly the same running time as that of A such that:

$$\text{Adv}_{\text{cpa}}^{\text{Kyber.CPAPKE}}(A) \leq 2 \cdot \text{Adv}_{k+1,k,\eta}^{\text{mlwe}}(B) + \text{Adv}_{\text{PRF}}(C).$$

Kyber.CCAKEM is obtained via a slightly tweaked Fujisaki-Okamoto transform applied to Kyber.CPAPKE. The following concrete security statement proves Kyber.CCAKEM's IND-CCA2-security when the hash functions G and H are modeled as random oracles.

Theorem 2. Suppose XOF, H, and G are random oracles. For any classical adversary A that makes at most q_{RO} many queries to random oracles XOF, H, and G, there exist adversaries B and C of roughly the same running time as that of A such that:

$$\text{Adv}_{\text{cca}}^{\text{Kyber.CCAKEM}}(A) \leq 2\text{Adv}_{k+1,k,\eta}^{\text{mlwe}}(B) + \text{Adv}_{\text{PRF}}(C) + 4q_{RO}\delta.$$

Non-Tight Reduction from MLWE in the QROM

Theorem 3. Suppose XOF, H, and G are random oracles. For any quantum adversary A that makes at most q_{RO} many queries to quantum random oracles XOF, H, and G, there exist quantum adversaries B and C of roughly the same running time as that

of A such that:

$$\text{Adv}_{\text{cca}}^{\text{Kyber.CCAKEM}}(A) \leq 4q_{RO}^2 \text{Adv}_{k+1,k,\eta}^{\text{mlwe}}(B) + \text{Adv}_{\text{PRF}}(C) + 8q_{RO}\delta.$$

3.6.4 Estimated security strength

All parameter sets of Kyber have some probability of decryption failure. These failures are a security concern (see Section 7.3), and so the probabilities with which they occur need to be small. But because in the classical random oracle model, the decryption failure probability is information-theoretic, we do not see a need for it to decrease with the security parameter. In particular, decryption failure for our level 3 and 5 parameter sets is less than 2^{-160} , which means that if 2^{80} instances of Kyber were run every second from now until our sun becomes a white dwarf, the odds still heavily favor there never being a decryption failure. We therefore exclude these attacks from our claims regarding the NIST security estimates.

The Impact of the Deterministic Noise Caused by Compress_q on Kyber512

Each coefficient of e_1 (and e_2) in Algorithm 4 is distributed as a binomial distribution with parameter $\eta_2 = 2$, which has variance $\eta_2/2 = 1$. The parameter $d_u = 10$ implies that the Compress_q function maps elements modulo q to a set of size 2^{10} where the difference between every two elements in the latter set is either 3 or 4. This implies that the error added by the Compress_q function for each coefficient is either uniform over $\{-1, 0, 1\}$, $\{-1, 0, 1, 2\}$, or $\{-2, -1, 0, 1\}$. It therefore has variance at least as large as the uniform distribution over the set $\{-1, 0, 1\}$, which is $2/3$. This makes the total variance of each coefficient of e_1 plus the deterministic error at least $1 + 2/3 = 5/3$. The other secret and noise terms have binomial distributions with parameter $\eta_1 = 3$ for a variance of $\eta_1/2 = 3/2 < 5/3$. When accounting for the errors added by Compress_q , we therefore calculate the hardness of Kyber512 assuming that the variance of each secret/error coefficient is $3/2$.

The Impact of MAXDEPTH

The best known quantum speedups for the sieving algorithm, which we consider in our cost analysis (see Subsection 5.1.1), are only mildly affected by limiting the depth of a quantum circuit, because it uses Grover search on sets of small size (compared to searching through the whole keyspace of AES). For the core-SVP-hardness operation estimates to match the quantum gate cost of breaking AES at the respective security levels, a quantum computer would need to support a maximum depth of 70–80. When limiting the maximum depth to smaller values, or when

considering classical attacks, the core-SVP-hardness estimates are smaller than the gate counts for attacks against AES. A recent study on the concrete cost of sieving suggests that the quantum speed-ups of these algorithms are tenuous, independently of the value of MAXDEPTH.

3.7 Analysis with Respect to Known Attacks

3.7.1 Attacks Against the Underlying MLWE Problem

MLWE as LWE

The best known attacks against the underlying MLWE problem in Kyber do not make use of the structure in the lattice. We therefore analyze the hardness of the MLWE problem as an LWE problem. We briefly discuss the current state of the art in algebraic attacks, i.e., attacks that exploit the structure of module lattices (or ideal lattices) at the end of this subsection.

Attacks Against LWE

Many algorithms exist for solving LWE, but many of those are irrelevant for our parameter set. In particular, because there are only

$$m = (k + 1)n$$

LWE samples available to the attacker, we can rule out BKW-type attacks and linearization attacks. This essentially leaves us with two BKZ attacks, usually referred to as primal and dual attacks.

The algorithm BKZ proceeds by reducing a lattice basis using an SVP oracle in a smaller dimension b . It is known that the number of calls to that oracle remains polynomial, yet concretely evaluating the number of calls is complex and subject to new heuristic ideas.

We start with an analysis that ignores this polynomial factor, i.e., only considers the cost of a single call to an SVP oracle in dimension b . We will also use for now a very simple cost estimate for the hardness of SVP. This core-SVP hardness methodology was introduced in, as a simple way of estimating security. In light of cryptanalytic progress during the NIST evaluation Rounds 1 and 2, this method remains informative but too coarse to produce accurate security estimates, especially for security against classical attackers.

Enumeration vs. Sieving

There are two algorithmic approaches for the SVP oracle in BKZ: enumeration and sieving algorithms. These two classes of algorithms have very different performance characteristics. Specifically, sieving algorithms have only exponential running time, while enumeration algorithms have super-exponential running time. Experimental evidence shows that enumeration algorithms are more efficient in small dimensions, with sieving becoming more efficient beyond dimension $b \approx 80$.

The hardness estimation is further complicated by the fact that sieving algorithms are much more memory-intensive than enumeration algorithms. Sieving algorithms have exponential complexity in both time and memory, whereas enumeration algorithms require only small amounts of memory. To obtain a conservative lower bound, we follow the approach of, assuming free memory access even for exponentially large memory.

Recent work has improved the heuristic complexity of sieving algorithms using locality-sensitive hashing (LSH) techniques. Most sieving algorithms also benefit from Grover's quantum search algorithm, reducing complexity to $2^{0.265b+o(b)}$. However, practical quantum speed-up remains tenuous.

For our first analysis, we use

$$2^{0.292b}$$

as the classical and

$$2^{0.265b}$$

as the quantum cost estimate for both primal and dual attacks with block size (dimension) b .

Primal Attack

The primal attack constructs a unique-SVP instance from the LWE problem and solves it using BKZ. Given the matrix LWE instance $(A, b = As + e)$, one builds the lattice

$$\Lambda = \{x \in \mathbb{Z}^{m+kn+1} : (A|I_m| - b)x = 0 \pmod{q}\}$$

of dimension $d = m+kn+1$, volume q^m , and with a unique-SVP solution $v = (s, e, 1)$ of norm

$$\lambda \approx \varsigma \sqrt{kn + m},$$

where ς is the standard deviation of the secret/error coefficients.

The success condition of BKZ is given by

$$\varsigma\sqrt{b} \leq \delta^{2b-d-1} \cdot q^{m/d},$$

where $\delta = ((\pi b)^{1/b} \cdot b/2\pi e)^{1/2(b-1)}$.

Dual Attack

The dual attack finds a short vector in the dual lattice $\Lambda^0 = \{(x, y) \in \mathbb{Z}^m \times \mathbb{Z}^{kn} : A^T x = y \pmod{q}\}$. Given such a vector (x, y) of length ℓ , we compute

$$z = v^T b = v^T A s + v^T e = w^T s + v^T e \pmod{q},$$

which is Gaussian-distributed with standard deviation $\ell\varsigma$. The advantage against decision-LWE is given by

$$\epsilon = 4 \exp(-2\pi^2 \tau^2),$$

where $\tau = \ell\varsigma/q$.

The length ℓ of a vector given by BKZ is

$$\ell = \delta^{d-1} q^{kn/d}.$$

To obtain an ϵ -distinguisher, BKZ must run with block dimension b satisfying

$$-2\pi^2 \tau^2 \geq \ln(\epsilon/4).$$

An attacker needs an advantage of at least $1/2$ to significantly decrease the search space of the agreed key. They must amplify their success probability by generating at least $1/\epsilon^2$ such short vectors. Since sieving provides $2^{0.2075b}$ vectors, the attack must be repeated at least

$$R = \max(1, 1/(2^{0.2075b} \epsilon^2)).$$

3.7.2 Beyond core-SVP hardness

At the time the core-SVP hardness measure was introduced by, the best implementations of sieving had performance significantly worse than the 2.292^b CPU cycles proposed as a conservative estimate by this methodology. This was due to substantial polynomial or even sub-exponential overheads hidden in the complexity analysis of $2.292^{b+o(b)}$ given in. Before this, proposed the core-SVP approach, security estimates

of lattice schemes were typically based on the cost of SVP via enumeration given in, leading to much more aggressive parameters. Beyond the cost of SVP-calls, this methodology also introduced a different prediction of when BKZ solves LWE, which was later confirmed and refined.

While doubts were expressed as to whether sieving would ever outperform the super-exponential, yet practically smaller, costs of enumeration for relevant cryptographic dimensions, significant progress on sieving algorithms has brought down the cross-over point to about $b = 80$. In fact, the current SVP records are now held by algorithms that employ sieving. This progress mandates a revision and refinement of Kyber security estimates, especially regarding classical attacks. In particular, while it was evident from experiments that the costs hidden in the $o(b)$ before those improvements were positive both in practice and asymptotically, the dimensions-for-free technique offers a sub-exponential speed-up, making it a priori unclear whether the total $o(b)$ term is positive or negative, both asymptotically and concretely.

In summary, while the core-SVP methodology introduced five years ago has pushed designers to be more conservative than previously, it now appears that this estimation technique is too coarse to produce accurate security estimates. In the following, we provide a refined analysis based on the latest developments described in the literature. We complement this with a discussion of all the approximations, simplifications, and foreseeable developments that remain to be explored.

We also note that the choice of the gate count metric was recently discussed in the NIST PQC forum mailing list, and in the case of the algorithm of, we do not believe this metric to be very realistic. However, it appears that alternative metrics unavoidably involve physical and technological constants (speed of light, density of information, energy efficiency of gates and data transfers). There seems to be no clear consensus on what these constants would be with current technology, let alone future technologies. These other metrics would also greatly increase the complications involved in the already delicate tuning and analysis of sieving.

In the following subsection, we therefore discuss the current understanding of attacks against Kyber in the gate-count metric. We focus the discussion on the concrete case of Kyber512. This preliminary analysis gives a cost of 2^{151} gates, which is a 2^8 factor margin over the targeted security of the 2^{143} gates required for attacks against AES. Our discussion of the ‘known unknowns’ concludes that this number could be affected by a factor of up to 2^{16} in either direction. While there is a risk of seeing the security claim drop below the 2^{143} bar in the gate count metric, one should consider the choice of the gate count metric itself as a substantial margin. We do not think

that even a drop as large as 2^{16} would be catastrophic, particularly given the massive memory requirements that are ignored in the gate-count metric. Having listed and documented the sources of uncertainties, we hope that many of them can be tackled in the months to come to narrow the “confidence interval”.

A tentative gate-count estimate accounting for recent progress

For concreteness, we focus this discussion on the case of Kyber512. Let us start by defining the progressivity overhead:

$$C = \frac{1}{1 - 2^{-0.292}} = 5.46, \quad (3.1)$$

which is the limit of the ratio between $\sum_{i \leq b} 2.292^i + o(i)$ and $2.292^{b+o(b)}$ as b grows.

Primal Attack Only. The core-SVP hardness methodology suggests that the dual attack is slightly cheaper than the primal one. However, it is, in fact, significantly more expensive. The analysis of the dual attack in assumes that one gets exponentially many vectors from sieving in the first block that will be as short as the shortest one. In reality, most of them will have a length approximately $\sqrt{4/3}$ larger. Moreover, the assumption that exponentially many such short vectors are obtained is incompatible with some of the latest sieving improvements, specifically the dimensions-for-free technique from.

BKZ Simulation. The analysis of the BKZ success condition from is based on the geometric-series assumption, which has several inaccuracies; in particular, it misses a “tail” phenomenon. We instead rely on the simulator provided as part of the leaky-LWE-estimator. This simulator uses progressive-BKZ, which provides better performance than fixed-block BKZ. It predicts a median success when reaching blocksize $b = 413$.

Dimensions for free. According to [7], the number of “dimensions for free” is:

$$d_{4f} = \frac{b \ln(4/3)}{\ln(b/(2\pi e))} \approx 37.3. \quad (3.2)$$

That is, each SVP oracle call in dimension $b = 413$ requires sieving in dimension $b' = 375$.

Final Gate Count. Overall, we conclude with a gate count of:

$$G = (1025 - 413) \cdot C^2 \cdot 2^{137.4} = 2^{151.5}. \quad (3.3)$$

3.7.3 Attacks exploiting decryption failures

In Theorems 2 and 3, we see that decryption failure probability plays a role in the attacker's advantage: in the classical context in the term $4q_{RO}\delta$ and in the quantum context in the term $8q_{RO}^2\delta$, where q_{RO} is the number of queries to the (classical or quantum) random oracle.

Attacks exploiting failures

This term in the attacker's advantage is not merely a proof artifact, it can be explained by the following attack: An attacker searches through many different values of m (see line 1 of Algorithm 8) until he finds one that produces random coins r (line 3 of Algorithm 8) that lead to a decapsulation failure, which will give the attacker information about the secret key. In the quantum setting, the search through different values of m is accelerated by Grover's algorithm, which explains the square in the term q_{RO}^2 . With this attack in mind, note that with 2^{64} ciphertexts (compare Section 4.A.2 of the Call for Proposals), there is a chance of 2^{-100} of a decapsulation failure in Kyber768 without any particular effort by the attacker.

To understand what exactly this means for attacks against Kyber, we need to address the following two questions:

1. How hard is it for an attacker to trigger a Kyber decapsulation failure?
2. How hard is it for an attacker equipped with a ciphertext triggering a failure to mount a successful attack against Kyber?

Regarding the first question, the naive approach of an attacker is to try random ciphertexts, which has a success probability of $q_d\delta$, where q_d is the number of decapsulation queries. In the classical random-oracle model, the cost of an attack exploiting failures will also never get lower than that, as it matches the information-theoretic success probability.

Failure boosting using Grover

A quantum attacker can try to use Grover search to precompute values of m that have a slightly higher chance to produce a failure. The efficacy of Grover search is limited by the fact that an attacker cannot determine offline whether a given value of m , or more specifically, the derived values r (line 9 of Algorithm 4) and e_1 (line 13 of Algorithm 4), produce a decapsulation failure. The reason is that the probability of decapsulation failures largely depends on subtle properties of the error distribution and the interactions between the ciphertext and the secret key.

Original Kyber analysis The original Kyber submission document gave an analysis of a particular strategy for using Grover’s algorithm to search for values of m that produce e_1 and r with above-average norm. Intuitively, the larger these values are, the greater the probability of a decryption failure. The gain achieved through such an approach is, however, limited due to the fact that the distribution of a high-dimensional Gaussian is tightly concentrated around its expected value, while that of a one-dimensional Gaussian is not as tightly concentrated around its mean. We present this original analysis below, focusing on Kyber768, which has a failure probability of 2^{-164} , for concreteness:

The polynomial pair (e_1, r) can be seen as a vector in \mathbb{Z}^{1536} distributed as a discrete Gaussian with standard deviation $\sigma = \eta_1/2 = 1$ (because $\eta_1 = \eta_2$). By standard tail bounds on discrete Gaussians, we know that an m -dimensional vector v drawn from a discrete Gaussian of standard deviation σ will satisfy

$$\Pr[\|v\| > \kappa\sigma\sqrt{m}] < \kappa^m \cdot e^{\frac{m}{2}(1-\kappa^2)}, \quad (3.4)$$

for any $\kappa > 1$.

So for example, the probability of finding a vector which is of length $1.33 \cdot \sigma\sqrt{1536}$ is already as small as 2^{-220} . Even if Grover’s algorithm reduces the search space and increases the probability to 2^{-110} , finding such a vector merely increases the chances of getting a decryption error; and the probability increase is governed by the tail-bounds for one-dimensional Gaussians. For any vector v , if z is chosen according to a Gaussian with standard deviation σ , then for any κ ,

$$\Pr[|\langle z, v \rangle| > \kappa\sigma\|v\|] \leq 2e^{-\kappa^2/2}. \quad (3.5)$$

If originally, the above probability is set so that decryption errors occur with probability $\approx 2^{-160}$, then $\kappa \approx 15$. If the adversary is then able to increase $\|v\|$ by a factor of 1.33 (by being able to find larger (e_1, r)), then we can decrease κ by a factor of 1.33 to ≈ 11.25 in (5), which would still give us a probability of a decryption error of less than 2^{-90} . However, finding such a large v would take at least 2^{110} time, which would make the whole attack cost at least 2^{200} .

Of course, one can try to find a slightly smaller v in the first step so that the entire attack takes less time. If Grover’s algorithm really saves a square-root factor, then the optimal value is ≈ 1.05 for κ in (4), which would allow us to lower κ by a factor of 1.05 in (5) to $15/1.05 \approx 14.28$, and would still give a total time to find one decryption error $\approx 2^{-150}$. This makes the attack completely impractical.

Different Approaches Since 2017 Since the original analysis of failure boosting against Kyber, multiple works have considered multiple aspects of this topic. The latest work on this topic considers using Grover’s algorithm to produce a strategic set of decryption query points which results in a higher chance of triggering a decryption failure when the number of decryption queries is limited, as in the NIST Call for Proposals. The overall running time is, however, not less than in the above attack, which didn’t restrict itself with a limit on the number of decapsulation queries.

From One to Multiple Failures The original Kyber submission document also discussed that an attacker may need more than a single failure to mount a key-recovery attack. For example, the work of [12] required up to 4000 failures, which may suggest that the number of online queries required for a successful attack against Kyber increases by a factor of $\approx 2^{12}$. However, recent work proposed an adaptive strategy, which uses previous failures to significantly lower the cost of the next one. This directional failure boosting technique reduces the cost of the full attack to barely more than the cost of triggering the first failure. This means that the answer to the second question in the beginning of this subsection is that one should be on the safe side and make sure that it’s hard to trigger even one failure.

Multitarget Attacks Using Failures Despite the limited gain, an attacker could consider using Grover’s algorithm to precompute values of m that produce r and e_1 with large norm and then use this precomputed set of values of m against many users. This multi-target attack is prevented by hashing the public key pk into the random coins r and thereby into r and e_1 .

3.8 KYBER Implementation

This is the official reference implementation of the **Kyber** key encapsulation mechanism, and an optimized implementation for x86 CPUs supporting the AVX2 instruction set. Kyber has been selected for standardization in **Round 3** of the **NIST PQC** standardization project.

Build Instructions

The implementations contain several test and benchmarking programs and a Makefile to facilitate compilation.

Prerequisites

Some of the test programs require **OpenSSL**. If the OpenSSL header files and/or shared libraries do not lie in one of the standard locations on your system, it is necessary to specify their location via compiler and linker flags in the environment variables `CFLAGS`, `NISTFLAGS`, and `LDFLAGS`.

For example, on macOS you can install OpenSSL via **Homebrew** by running:

```
brew install openssl
```

Although ubuntu comes with openssl preinstalled, you may use the following command if it is missing;

```
sudo apt-get install libssl-dev
```

Then, set environment variables:

```
export CFLAGS="-I/usr/local/opt/openssl@1.1/  
include"  
export NISTFLAGS="-I/usr/local/opt/openssl@1  
.1/include"  
export LDFLAGS="-L/usr/local/opt/openssl@1.1/  
lib"
```

Building All Binaries

To compile the test and benchmarking programs on Linux or macOS, go to the `ref/` or `avx2/` directory and run:

```
make
```

This produces the executables:

```
test/test_kyber$ALG  
test/test_vectors$ALG  
test/test_speed$ALG
```

where `$ALG` ranges over the parameter sets 512, 768, 1024.

- `test_kyber$ALG`: Runs 1000 tests to generate keys, encapsulate a random key, and decapsulate it correctly. It also tests for CCA failures using a random secret key or distorted ciphertext.

- `test_vectors$ALG`: Generates 10,000 sets of test vectors (keys, ciphertexts, and shared secrets) in hexadecimal, including vectors for invalid ciphertexts.
- `test_speed$ALG`: Reports median and average cycle counts of 1000 executions of internal and API functions. Uses TSC by default; for Performance Measurement Counters, set `CFLAGS="-DUSE_RDPMC"` before compilation.

Note: The reference implementation in `ref/` is not optimized for any platform. Benchmarking this code may not yield meaningful performance data.

Shared Libraries

All implementations can be compiled into shared libraries by running:

```
make shared
```

In the `ref/` directory, this produces:

```
libpqcrystals_kyber$ALG_ref.so
```

for each parameter set `$ALG`, along with the symmetric crypto libraries:

```
libpqcrystals_aes256ctr_ref.so  
libpqcrystals_fips202_ref.so
```

All global symbols lie in the namespaces `pqcrystals_kyber$ALG_ref`, `libpqcrystals_aes256ctr_ref`, and `libpqcrystals_fips202_ref`, allowing simultaneous linking against all libraries.

The corresponding API header file is `ref/api.h`, which contains function prototypes and preprocessor definitions for key and signature lengths.

Example

Here is a sample code of how to use the api

```
#include <stdio.h>  
#include <stdint.h>  
#include <stdlib.h>  
#include <string.h>
```

```
#include "ref/api.h"

// XOR encryption/decryption (for demo only)
void xor_encrypt_decrypt(uint8_t *output,
    const uint8_t *input, const uint8_t *key,
    size_t len) {
    for (size_t i = 0; i < len; i++) {
        output[i] = input[i] ^ key[i %
    pqcrystals_kyber512_ref_BYTES];
    }
}

int main() {
    // Message to encrypt
    const char *message = "Hello universe";
    size_t msg_len = strlen(message);

    // Kyber buffers
    uint8_t pk[
    pqcrystals_kyber512_ref_PUBLICKEYBYTES];
    uint8_t sk[
    pqcrystals_kyber512_ref_SECRETKEYBYTES];
    uint8_t ct[
    pqcrystals_kyber512_ref_CIPHERTEXTBYTES];
    uint8_t ss_enc[
    pqcrystals_kyber512_ref_BYTES];
    uint8_t ss_dec[
    pqcrystals_kyber512_ref_BYTES];

    // Generate keypair
    pqcrystals_kyber512_ref_keypair(pk, sk);
}
```

```
// Encapsulate shared key
pqcrystals_kyber512_ref_enc(ct, ss_enc,
pk);

// Decapsulate on receiver side
pqcrystals_kyber512_ref_dec(ss_dec, ct,
sk);

// Check if both shared secrets match
if (memcmp(ss_enc, ss_dec,
pqcrystals_kyber512_ref_BYTES)  $\neq$  0) {
    fprintf(stderr, "Shared secrets do
not match!\n");
    return 1;
}

// Encrypt message using shared secret
uint8_t *encrypted = malloc(msg_len);
xor_encrypt_decrypt(encrypted, (const
uint8_t *)message, ss_enc, msg_len);

// Decrypt message
uint8_t *decrypted = malloc(msg_len + 1);
xor_encrypt_decrypt(decrypted, encrypted,
ss_dec, msg_len);
decrypted[msg_len] = '\0'; // Null-
terminate

// Cleanup
free(encrypted);
```

```
free(decrypted);  
return 0;  
}
```

LISTING 3.1: C code using Kyber for key encapsulation with XOR-based message encryption

Chapter 4

ML-DSA: Dilithium

4.1 CRYSTALS: Dilithium

CRYSTALS-Dilithium [5] is a lattice-based digital signature scheme whose security is based on the hardness of finding short vectors in lattices. It is a part of the CRYSTALS (Cryptographic Suite for Algebraic Lattices) algorithm suite. The security notion means that an adversary with access to a signing oracle cannot produce a valid signature on an unseen message, nor forge a different signature on a previously signed message.

The design of Dilithium is based on the "Fiat-Shamir with Aborts" technique of Lyubashevsky [19] which uses rejection sampling to make lattice-based Fiat-Shamir schemes compact and secure. The scheme with the smallest signature sizes using this approach is the one of Ducas et. al [8], which is based on the NTRU assumption and crucially uses Gaussian sampling for creating signatures. Because Gaussian sampling is hard to implement securely and efficiently, the authors have opted to only use the uniform distribution. Dilithium improves on the most efficient scheme that only uses the uniform distribution, due to Bai and Galbraith, by using a new technique that shrinks the public key by more than a factor of 2. Currently, Dilithium has the smallest public key + signature size of any lattice-based signature scheme that only uses uniform sampling.

A template for the Key Generation, Signing, and Verification process of CRYSTALS - Dilithium has been provided in Figure 4.3. It is to be noted that throughout this section the function H refers to SHAKE-256.

$$H(v, d) = \text{SHAKE256}(v, d)$$

The SHAKE256 algorithm, as per FIPS 202 [9], guarantees that for any positive integers $c < d$ and bit string $\rho \in \{0, 1\}^*$, $H(\rho, c)$ is exactly equal to the first c bits of $H(\rho, d)$. The same property holds for SHAKE128.

4.1.1 Dilithium Parameters Overview

Parameters

Table 4.1 refers to the values of the parameters used in the Dilithium Algorithms.

Parameter Name	NIST Security Level 2	NIST Security Level 3	NIST Security Level 5
q [modulus]	8380417	8380417	8380417
d [dropped bits from t]	13	13	13
γ_1 [y coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q - 1)/88$	$(q - 1)/32$	$(q - 1)/32$
(k, l) [Dimensions of Matrix A]	(4,4)	(6,5)	(8,7)
η [Secret Key coefficient range]	2	4	2
τ [No. of ± 1 's in c]	39	49	60
$\beta[\tau, \eta]$	78	196	120
ω [max no. of 1's in hint h]	80	55	75

TABLE 4.1: Parameters for CRYSTALS DILITHIUM

Notations

The notations have been maintained with their descriptions in Table 4.2.

Algorithms for Dilithium

The "SampleInBall" function described in **Algorithm 10** takes a parameter ρ as input and initializes a 256-element vector 'c' with zeros. It then iterates from 256 minus another parameter τ to 255, each time randomly selecting an index 'j' and a value 's'. It assigns the value of c_j to c_i and sets c_j to $(-1)^s$. After completing this loop, the function returns the modified vector 'c'. The primary purpose of this function is to generate a random vector within a ball of radius ρ centered at the origin in Hamming space. It does so by randomly selecting indices and values for the vector elements while ensuring that the resulting vector lies within the specified ball.

Symbol	Class	Description
S^*	Byte Array Set	If S is a set, this denotes the set of all possible byte arrays of arbitrary length.
S^k	Byte Array Set	If S is a set, this denotes the set of all possible byte arrays of length k .
\hat{f}, \hat{A}	Polynomial and Matrix	Corresponding NTT Representation of the Polynomial f and the matrix A .
\mathbb{Q}	Set	Rational Number Set
\mathbb{Z}_m	Ring	Ring of Integers(\mathbb{Z}) modulo m
v^T, A^T	Matrix operations	Represent the transpose of a row or column vector v and transpose of the matrix A .
$r \bmod^{\pm} m$	Modular Reductions	For m even (respectively, odd), this denotes the unique integer r' such that $-\frac{m}{2} < r' \leq \frac{m}{2}$ (respectively, $-\frac{m-1}{2} \leq r' \leq \frac{m-1}{2}$) and m divides $r - r'$.
$ B $	Mathematical Operation	If B is a number, $ B $ represents the absolute value of B .
$\lceil x \rceil, \text{round}(x), \lfloor x \rfloor$	Mathematical Operation	Represents the ceiling, round and floor value of x respectively.
\mathbb{B}	Byte Array	The set $0,1,\dots,255$ of unsigned 8-bit integers (bytes).
$A B$	Array Operation	Represents the concatenation of two array or bit strings A and B .
n	Parameter	Set to the value of 256 throughout.
q	Parameter	Set to the value of $2^{23} - 2^{13} + 1 = 8380417$ throughout.
R_q	Polynomial Ring	The ring $\mathbb{Z}_q[X]/(X^n + 1)$ consists of polynomials of the form $f = f_0 + f_1X^{255} + \dots + f_{255}X$ where $f_j \in x\mathbb{Z}_q$ for all j , equipped with addition and multiplication modulo $X^n + 1$.

TABLE 4.2: Notations for CRYSTALS DILITHIUM

The `SampleInBall` routine absorbs the 32 bytes of the seed into SHAKE-256. Throughout its operations, the function squeezes SHAKE-256 in order to obtain a stream of random bytes of variable length. The first τ bits in the first 8 bytes of this random stream are interpreted as τ random sign bits $s_i \in \{0, 1\}, i = \{0, \dots, \tau - 1\}$. The remaining $\{64 - \tau\}$ bits are discarded. In each iteration of the for loop in Algorithm 10, rejection sampling has been used on elements from $\{0, \dots, 255\}$ until a $j \in \{0, \dots, i\}$ is obtained. An element in $\{0, \dots, 255\}$ is obtained by interpreting the next byte of the random stream from SHAKE-256 as a number in this set. For the sign s the corresponding s_{i-196} is used.

Algorithm 9 `SampleInBall`(ρ)

```

1: Initialize  $c = c_0c_1 \dots c_{255} = 00 \dots 0$ 
2: for  $i := 256 - \tau$  to 255 do
3:    $j \leftarrow \{0, 1, \dots, i\}$ 
4:    $s \leftarrow \{0, 1\}$ 
5:    $c_i \leftarrow c_j$ 
6:    $c_j \leftarrow (-1)^s$ 
7: end for
8: return  $c$ 

```

The function `ExpandA`, described in **Algorithm 11**, maps a seed to a matrix A in NTT domain representation, which is used for multiplication. However, for faster implementations, `ExpandA` outputs a different representation of A called \hat{A} . The function `ExpandA` maps a uniform seed $\rho \in \{0, 1\}^{256}$ to a matrix $A \in R_q^{k \times l}$ in NTT domain representation. It computes each coefficient $\hat{a}_{i,j} \in R_q$ of \hat{A} separately. For the coefficient $\hat{a}_{i,j}$ it absorbs the 32 bytes of ρ immediately followed by two bytes representing $0 \leq 256 * i + j < 2^{16}$ in little-endian byte order into SHAKE-128. The output stream of SHAKE-128 is interpreted as a sequence of integers between 0 and $2^{23} - 1$. This is done by setting the highest bit of every third byte to zero and interpreting blocks of 3 consecutive bytes in little-endian byte order. So for example the three bytes b_0, b_1 , and b_2 are used to get the integer

$$0 \leq b'_2 \cdot 2^{16} + b_1 \cdot 2^8 + b_0 \leq 2^{23} - 1$$

where b'_2 is the logical AND of b_2 and $2^{128} - 1$.

The function `ExpandS`, described in **Algorithm 12**, generates secret vectors (s_1, s_2) in key generation by mapping a seed to these vectors. The `ExpandS`(ρ') algorithm generates a polynomial a with coefficients in the range $[-\eta, \eta]$ by performing rejection sampling from a seed ρ' . It initializes variables for tracking the current index and the number of iterations, then enters a loop to generate 256 coefficients

Algorithm 10 ExpandA(ρ)**Require:** $\rho \in \{0, 1, 2, \dots, 256\}$ **Ensure:** Matrix \hat{A}

```

1: for  $i$  from 0 to  $k - 1$  do
2:   for  $j$  from 0 to  $\ell - 1$  do
3:      $\hat{A}[i, j] \leftarrow$  Polynomials in NTT Representation using Rejection Sampling
4:   end for
5: end for
6: return  $\hat{A}$ 

```

for the polynomial. Within the loop, it uses an extendable-output function $H(\rho')$ to generate random values, say, z and extracts two coefficients, suppose z_0 and z_1 , from each z . If z_0 falls within the desired range, it is assigned to the current index of the polynomial a , and the index is incremented. If z_1 falls within the range and the index is still less than 256, it is also assigned to a . The algorithm repeats this process, incrementing the iteration count, until 256 coefficients are generated. Finally, it returns the polynomial a with coefficients sampled from $[-\eta, \eta]$. This polynomial a forms an element of the vectors s_1 or s_2 as per the pseudocode.

Algorithm 11 ExpandS(ρ')**Require:** $\rho \in \{0, 1\}^{512}$ **Ensure:** Vectors $s1, s2$ of polynomials in R_q

```

1: for  $r$  from 0 to  $\ell - 1$  do
2:    $s1[r] \leftarrow$  Polynomials with coefficients within  $-\eta$  to  $\eta$ 
3: end for
4: for  $r$  from 0 to  $k - 1$  do
5:    $s2[r] \leftarrow$  Polynomials with coefficients within  $-\eta$  to  $\eta$ 
6: end for
7: return ( $s1, s2$ )

```

The function ExpandMask, described in **Algorithm 13**, deterministically generates randomness for the signature scheme by mapping a seed and a nonce to a vector s . The function H is used for hashing in the signature scheme. For the i -th polynomial, $0 \leq i < \ell$, it absorbs the 48 bytes of ρ concatenated with the 2 bytes representing $\mu + i$ in little endian byte order into SHAKE-256. Then the output bytes are used to create a positive number in the range $\{0, \dots, 2\gamma_1 - 1\}$, and then the integers comprising s are then obtained by subtracting $(\gamma_1 - 1)$ from this positive number. Because γ_1 is a power of 2, no rejection sampling is required. Because γ_1 is 17 and 19, for different security levels, generating the consecutive integers comprising s involves either taking 18 or 20 bits at a time from the byte stream.

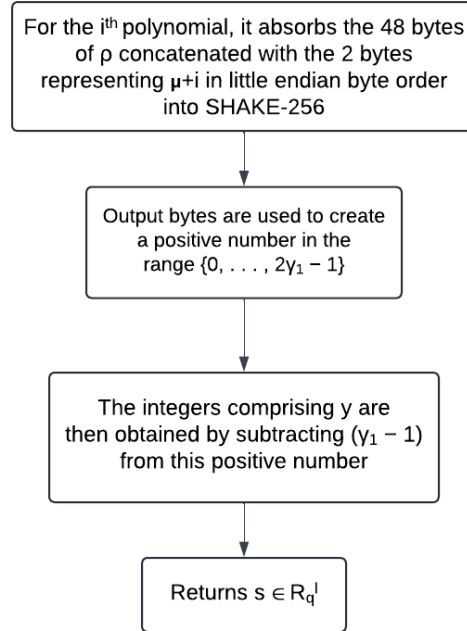


FIGURE 4.1: ExpandMask function defined in Algorithm 13

Algorithm 12 ExpandMask(ρ, μ) [4.1]

Require: Bit string $\rho \in \{0, 1\}^{512}$ and a nonnegative integer μ **Ensure:** Vector $s \in R_q^\ell$

- 1: $c \leftarrow 1 + \text{bitlen}(\gamma_1 - 1)$ ▷ Note: γ_1 is always a power of 2
 - 2: **for** r from 0 to $\ell - 1$ **do**
 - 3: $n \leftarrow \text{IntegerToBits}(\mu + r, 16)$
 - 4: $v \leftarrow (\text{H}(\rho||n)[32rc], \text{H}(\rho||n)[32rc + 1], \dots, \text{H}(\rho||n)[32rc + 32c - 1])$
 - 5: $s[r] \leftarrow \text{BitUnpack}(v, \gamma_1 - 1, \gamma_1)$
 - 6: **end for**
 - 7: **return** s
-

Algorithm 14 takes an integer r and a positive integer d as input and returns a pair of integers. It computes the remainder of r divided by a constant q , then further computes the remainder of r divided by $\pm 2^d$. The output consists of two integers: the quotient of $(r - r_0)$ divided by 2^d and the remaining value r_0 .

Figure 4.2 provides a consolidated view of the auxiliary algorithms used in the Dilithium digital signature scheme. These include essential components like sampling procedures, rejection sampling, polynomial encoding and decoding, and norm checks. Together, these subroutines facilitate secure and efficient signing and verification, while maintaining the structural integrity required by Dilithium's underlying lattice assumptions.

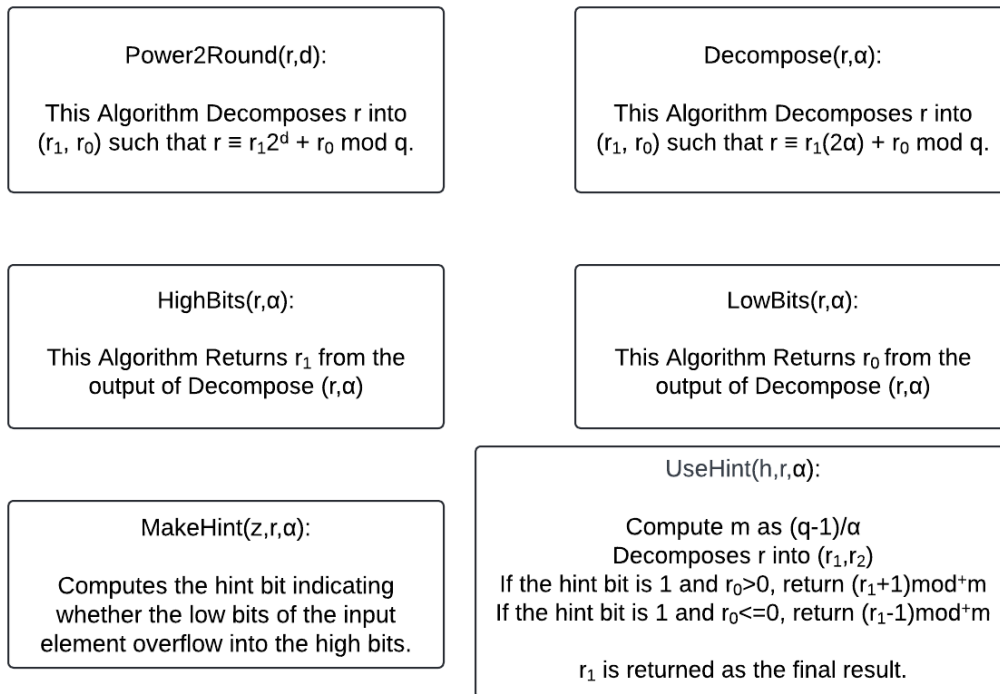


FIGURE 4.2: Key auxiliary algorithms involved in Dilithium signature scheme operations.

Algorithm 13 $Power2Round_q(r, d)$

- 1: $r \leftarrow r \pmod{q}$
 - 2: $r_0 \leftarrow r \pmod{\pm 2^d}$
 - 3: **return** $\left(\frac{r-r_0}{2^d}, r_0\right)$
-

Given an integer r and a positive integer λ , **Algorithm 15** computes the remainder of r divided by a constant q , followed by the remainder of r divided by $\pm \lambda$. Depending on whether $r - r_0$ equals $q - 1$, the algorithm sets r_1 to either 0 or $\frac{r-r_0}{\lambda}$. The output is a pair of integers (r_1, r_0) representing the decomposition of r .

Algorithm 14 $Decompose_q(r, \lambda)$

```

1:  $r \leftarrow r \bmod q$ 
2:  $r_0 \leftarrow r \bmod \pm \lambda$ 
3: if  $r - r_0 = q - 1$  then
4:    $r_1 \leftarrow 0$ 
5:    $r_0 \leftarrow r_0 - 1$ 
6: else
7:    $r_1 \leftarrow \frac{r-r_0}{\lambda}$ 
8: end if
9: return  $(r_1, r_0)$ 

```

Algorithm 16 takes three integers as input: z , r , and λ . It calculates the high bits of r and $r + z$ using the $HighBits_q$ algorithm and returns true if these high bits are not equal, indicating a difference in the most significant bits.

Algorithm 15 $MakeHint_q(z, r, \lambda)$

```

1:  $r_1 \leftarrow HighBits_q(r, \lambda)$ 
2:  $v_1 \leftarrow HighBits_q(r + z, \lambda)$ 
3: return  $r_1 \neq v_1$ 

```

Given a hint h , an integer r , and a positive integer λ , **Algorithm 17** computes $m = \frac{q-1}{\lambda}$. It then decomposes r into high and low bits using $Decompose_q$ and adjusts the result based on the hint. The output is an integer result according to specific conditions on the hint and the low bits of r .

Algorithm 16 $UseHint_q(h, r, \lambda)$

```

1:  $m \leftarrow \frac{q-1}{\lambda}$ 
2:  $(r_1, r_0) \leftarrow Decompose_q(r, \lambda)$ 
3: if  $h = 1$  and  $r_0 > 0$  then
4:   return  $(r_1 + 1) \bmod + m$ 
5: end if
6: if  $h = 1$  and  $r_0 \geq 0$  then
7:   return  $(r_1 - 1) \bmod + m$ 
8: end if
9: return  $r_1$ 

```

Algorithms 18 and 19 decompose an integer r into high and low bits and return the most significant bits and least significant bits of the number r .

The key generation procedure described in **Algorithm 20** for the CRYSTALS-Dilithium signature scheme starts by creating a random 256-bit string ζ . This string is then hashed using SHAKE-256 to produce two key components: a 512-bit string ρ and a 256-bit string K . Next, the algorithm expands ρ and ρ' using the ExpandA

Algorithm 17 $HighBits_q(r, \lambda)$

-
- 1: $(r_1, r_0) \leftarrow Decomposeq(r, \lambda)$
 - 2: **return** r_1
-

Algorithm 18 $LowBits_q(r, \lambda)$

-
- 1: $(r_1, r_0) \leftarrow Decomposeq(r, \lambda)$
 - 2: **return** r_0
-

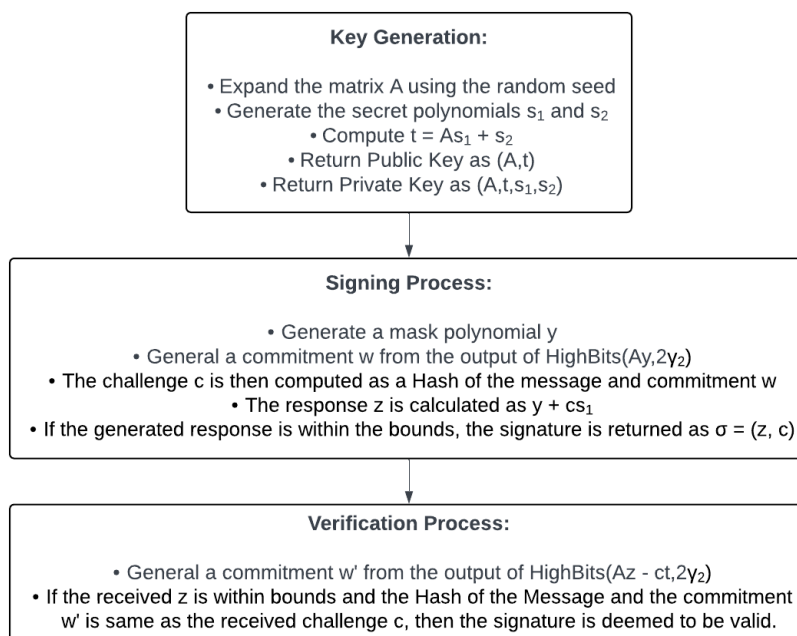


FIGURE 4.3: The template for CRYSTALS-Dilithium at a glance

and `ExpandS` functions, respectively. This expansion generates a $k \times n$ matrix A and two secret key vectors, s_1 and s_2 . All coefficients in A , s_1 , and s_2 belong to the polynomial ring R_q , where q is a specific modulus. The public key, t , is computed as the product of A and s_1 , with s_2 added. Then, the function `Power2Roundq` is employed to round t to the nearest multiple of $2^d/q$, where d is a security parameter. This rounded value of t is separated into two parts: t_1 , which contains the high-order bits, and t_0 , which contains the low-order bits. A value tr is calculated by hashing ρ concatenated with t_1 . In conclusion, the algorithm generates a public key consisting of ρ and t_1 , and a secret key containing ρ , K , tr , s_1 , s_2 , and t_0 .

Algorithm 19 Gen

- 1: $\zeta \leftarrow \{0, 1\}^{256}$
 - 2: $(\rho, \rho', K) \in \{0, 1\}^{256} \times \{0, 1\}^{512} \times \{0, 1\}^{256} \leftarrow \text{SHAKE-256}(\zeta)$
 - 3: $A \in R_q^{k \times \ell} \leftarrow \text{ExpandA}(\rho)$ ▷ Stored in NTT Representation as \hat{A}
 - 4: $(s_1, s_2) \in S_\eta^\ell \times S_\eta^k \leftarrow \text{ExpandS}(\rho')$
 - 5: $t \leftarrow A \cdot s_1 + s_2$ ▷ Compute $A \cdot s_1$ as $\text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(s_1))$
 - 6: $(t_1, t_0) \leftarrow \text{Power2Round}_q(t, d)$
 - 7: $tr \in \{0, 1\}^{256} \leftarrow \text{SHAKE-256}(\rho \parallel t_1)$
 - 8: **return** $(pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0))$
-

The Dilithium algorithm's signing process, described in **Algorithm 21**, initiated with the secret key sk and message M , involves the generation of a matrix A and secret key vectors s_1 and s_2 within a polynomial ring R_q . A masking vector y is computed, and w represents the high-order bits of coefficients in the product of A and y . A challenge c is formed from the message and w_1 . The potential signature z is obtained as the sum of y and cs_1 , where c is combined with s_1 . The algorithm computes r_0 as the low-order bits of $w - cs_2$. If certain coefficients exceed specified thresholds, the signing process restarts; otherwise, a hint h is computed. The resulting signature σ comprises \tilde{c} , z , and h , ensuring the security and authenticity of the Dilithium signature for M .

The verification algorithm, stated in **Algorithm 22**, takes a public key pk , a message M , and a signature σ consisting of three components: \tilde{c} , z , and h . It begins by expanding the public key matrix A into its NTT representation, denoted as \hat{A} . Next, it computes a 512-bit string μ as the hash of ρ concatenated with the first component t_1 of the secret key and the message M .

The signature component \tilde{c} is sampled from a ball of radius $\gamma_1 - \beta$. A hint w'_1 is computed using the function `UseHintq`, which takes $\hat{A} \cdot z$, t_1 , and a parameter $2\gamma_2$ as input. The hint is determined as $\text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(z) - \text{NTT}(c) \cdot \text{NTT}(t_1 \cdot 2^d))$, where d is a parameter of the signature scheme.

Algorithm 20 $\text{Sign}(sk, M)$

```

1:  $A \in R_q^{k \times \ell} \leftarrow \text{ExpandA}(\rho) \triangleright$  Generated and stored in NTT Representation as  $\hat{A}$ 
2:  $\mu \in \{0, 1\}^{512} \leftarrow \text{H}(tr \parallel M)$ 
3:  $\kappa \leftarrow 0, (z, h) \leftarrow \perp$ 
4:  $\rho' \in \{0, 1\}^{512} \leftarrow \text{H}(K \parallel \mu) \quad \triangleright$  Or  $\rho' \in \{0, 1\}^{512}$  for randomized signing
5: while  $(z, h) = \perp$  do  $\triangleright$  Pre-compute  $\hat{s}_1 = \text{NTT}(s_1), \hat{s}_2 = \text{NTT}(s_2)$ , and  $\hat{t}_0 = \text{NTT}(t_0)$ 
6:    $y \in \tilde{S}_{\gamma_1}^\ell \leftarrow \text{ExpandMask}(\rho_0, \kappa)$ 
7:    $w \leftarrow Ay \quad \triangleright$  Or  $w \leftarrow \text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(y))$ 
8:    $w_1 \leftarrow \text{HighBits}_q(w, 2\gamma_2)$ 
9:    $\tilde{c} \in \{0, 1\}^{256} \leftarrow \text{H}(\mu \parallel w_1)$ 
10:   $c \in B^{k \times \ell} \leftarrow \text{SampleInBall}(\tilde{c}) \quad \triangleright$  Store  $c$  in NTT representation as  $\hat{c} = \text{NTT}(c)$ 
11:   $z \leftarrow y + cs_1 \quad \triangleright$  Compute  $cs_1$  as  $\text{NTT}^{-1}(\hat{c} \cdot \hat{s}_1)$ 
12:   $r_0 \leftarrow \text{LowBits}_q(w - cs_2, 2\gamma_2) \quad \triangleright$  Compute  $cs_2$  as  $\text{NTT}^{-1}(\hat{c} \cdot \hat{s}_2)$ 
13:  if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|r_0\|_\infty \geq \gamma_2 - \beta$  then
14:     $(z, h) \leftarrow \perp$ 
15:  else
16:     $h \leftarrow \text{MakeHint}_q(-ct_0, w - cs_2 + ct_0, 2\gamma_2) \quad \triangleright$  Compute  $ct_0$  as  $\text{NTT}^{-1}(\hat{c} \cdot \hat{t}_0)$ 
17:    if  $\|ct_0\|_\infty \geq \gamma_2$  or the number of 1's in  $h$  is greater than  $\omega$  then
18:       $(z, h) \leftarrow \perp$ 
19:    end if
20:     $\kappa \leftarrow \kappa + \ell$ 
21:  end if
22: end while
23: return  $\sigma = (\tilde{c}, z, h)$ 

```

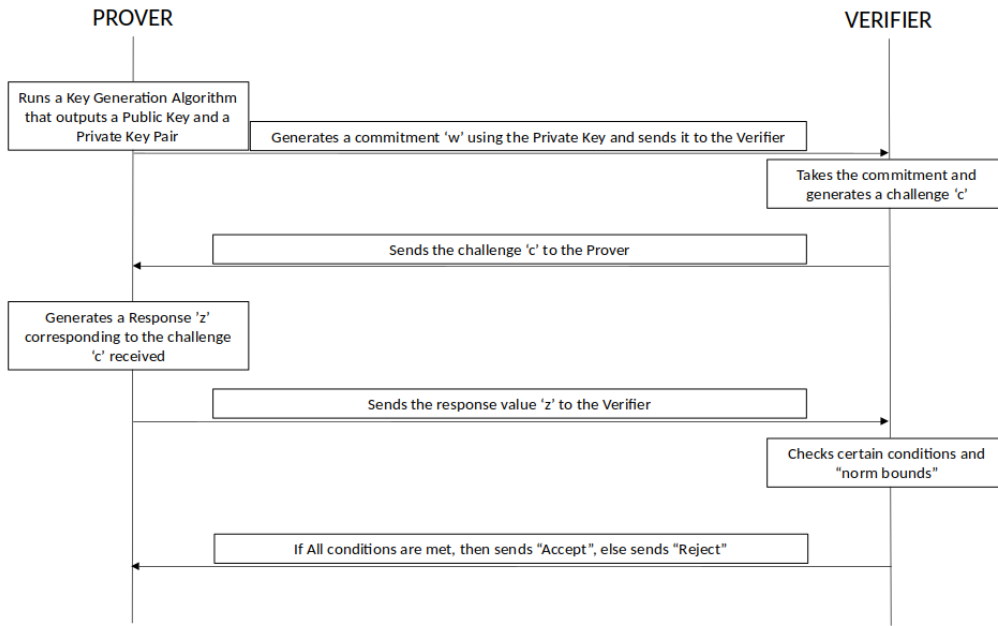


FIGURE 4.4: The basic construct for CRYSTALS: Dilithium Scheme

The verification algorithm then checks three conditions:

- It verifies that the infinity norm of z is less than $\gamma_1 - \beta$, ensuring that the signature is within an acceptable range.
- It confirms that \tilde{c} matches the hash of μ concatenated with w'_1 , ensuring the integrity of the signature.
- It checks that the number of ones in the hint h does not exceed a parameter ω , preventing overly sparse hints.

If all three conditions are met, the algorithm accepts the signature as valid; otherwise, it rejects it.

Figure 4.3 describes the main Key Generation, Signing, and Verification process, while Figure 4.4 provides the Commitment, Challenge, and Response structure followed by Dilithium.

Algorithm 21 $\text{Verify}(pk, M, \sigma = (\tilde{c}, z, h))$

- 1: $A \in R_q^{k \times \ell} \leftarrow \text{ExpandA}(\rho) \triangleright$ Generated and stored in NTT Representation as \hat{A}
 - 2: $\mu \in \{0, 1\}^{512} \leftarrow H(H(\rho \parallel t_1) \parallel M)$
 - 3: $c \in B_q^\ell \leftarrow \text{SampleInBall}(\tilde{c})$
 - 4: $w'_1 \leftarrow \text{UseHint}_q(h, Az - ct_1 \cdot 2^d, 2\gamma_2) \triangleright$ Compute as $\text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(z) - \text{NTT}(c) \cdot \text{NTT}(t_1 \cdot 2^d))$
 - 5: **return** $\|z\|_\infty < \gamma_1 - \beta$ and $\tilde{c} = H(\mu \parallel w'_1)$ and the number of 1's in h is $\leq \omega$
-

4.2 Build and Execution Guide

4.2.1 Installing Dependencies

To build and run the project using `liboqs` [22] on an Ubuntu system, first install the required dependencies:

LISTING 4.1: Install Required Packages

```
sudo apt install astyle cmake gcc ninja-build libssl-dev \  
python3-pytest python3-pytest-xdist unzip xsltproc \  
doxygen graphviz python3-yaml valgrind
```

4.2.2 Building `liboqs`

Navigate to the `liboqs` directory and build the library using `cmake` and `ninja`:

```
cd liboqs  
mkdir build && cd build  
cmake -GNinja ..  
ninja
```

LISTING 4.2: Build `liboqs`

4.2.3 Sample Signing Code

```
#include <stdio.h>  
#include <stdint.h>  
#include <stdlib.h>  
#include <string.h>  
#include <openssl/evp.h>  
#include "api.h"  
  
// Helper to print hex  
void print_hex(const char *label, const  
uint8_t *data, size_t len) {  
    printf("%s (%zu bytes):\n", label, len);  
    for (size_t i = 0; i < len; i++) {
```

```
        printf("%02x", data[i]);
        if ((i + 1) % 32 == 0) printf("\n");
    }
    printf("\n");
}

int main() {
    // Step 1: Input message
    char message[1024];
    printf("Enter message to sign: ");
    fgets(message, sizeof(message), stdin);
    size_t mlen = strlen(message);
    if (message[mlen - 1] == '\n') message[--mlen] = '\0';

    // Step 2: Compute SHA3-256 hash
    uint8_t hash[32]; // SHA3-256 produces
    // 32-byte hash
    EVP_MD_CTX *mdctx = EVP_MD_CTX_new();
    if (!mdctx) {
        fprintf(stderr, "EVP_MD_CTX_new
failed\n");
        return 1;
    }

    if (EVP_DigestInit_ex(mdctx, EVP_sha3_256
(), NULL) != 1 ||
        EVP_DigestUpdate(mdctx, message, mlen
) != 1 ||
        EVP_DigestFinal_ex(mdctx, hash, NULL)
!= 1) {
```

```
        fprintf(stderr, "SHA3-256 hashing
failed\n");
        EVP_MD_CTX_free(mdctx);
        return 1;
    }
    EVP_MD_CTX_free(mdctx);

    print_hex("SHA3-256 Hash", hash, sizeof(
hash));

    // Step 3: Generate keypair
    uint8_t pk[
pqcrystals_dilithium2_ref_PUBLICKEYBYTES];
    uint8_t sk[
pqcrystals_dilithium2_ref_SECRETKEYBYTES];

    if (pqcrystals_dilithium2_ref_keypair(pk,
sk) != 0) {
        fprintf(stderr, "Keypair generation
failed\n");
        return 1;
    }

    // Step 4: Sign hash
    uint8_t sig[
pqcrystals_dilithium2_ref_BYTES];
    size_t siglen;
    const uint8_t *ctx = NULL;
    size_t ctxlen = 0;
```

```
    if (pqcrystals_dilithium2_ref_signature(
sig, &siglen, hash, sizeof(hash), ctx,
ctxlen, sk) != 0) {
        fprintf(stderr, "Signature creation
failed\n");
        return 1;
    }

    print_hex("Public Key", pk,
pqcrystals_dilithium2_ref_PUBLICKEYBYTES);
    print_hex("Private Key", sk,
pqcrystals_dilithium2_ref_SECRETKEYBYTES);
    print_hex("Signature", sig, siglen);

    // Step 5: Verify signature
    int valid =
pqcrystals_dilithium2_ref_verify(sig,
siglen, hash, sizeof(hash), ctx, ctxlen, pk
);

    if (valid == 0) {
        printf("Signature is VALID\n");
    } else {
        printf("Signature is INVALID\n");
    }

    return 0;
}
```

4.2.4 Compiling the Sample Signing Code

Once the library is built, compile the sample code `sign.c` using `gcc`:

```
gcc -o sign sign.c \  
-I./liboqs/build/include \  
-L./liboqs/build/lib \  
-loqs -lm -lcrypto
```

LISTING 4.3: Compile Sample Signing Code

4.2.5 Running the Executable

Execute the compiled binary:

```
./sign
```

LISTING 4.4: Run the Sign Program

4.2.6 Expected Output

The program will prompt the user for a message, sign it, and verify the signature.

Example output:

```
Enter message to sign: Hello world
```

```
SHA3-256 Hash (32 bytes):
```

```
75d5e1f9f23d4a...
```

```
Signature (2420 bytes):
```

```
b4a1d29c0ab7...
```

```
Signature is VALID
```

Chapter 5

Tweaks To Kyber

5.1 Performance Analysis of Kyber

In this section we consider the implementational aspects of Kyber and report the performance results of the ANSI C reference implementation. Below tables report performance results of the reference implementation. All benchmarks were obtained on one core of an Intel Xeon (4) processor clocked at 2.800GHz MHz (as reported by `/proc/cpuinfo`) with TurboBoost and hyperthreading disabled. The benchmarking machine has 16 GB of RAM and is running Ubuntu 24.04.2LTS GNU/Linux with Linux kernel version 6.11.0. All implementations were compiled with gcc version 6.3.0. We used compiler flags `-O3 -fomit-frame-pointer -march=native -fPIC` to compile. All cycle counts reported are the median of the cycle counts of 10000 executions of the respective function. The implementations are not optimized for memory usage, but generally Kyber has only very modest memory requirements. This means that in particular our implementations do not need to allocate any memory on the heap.

Here we presented cycle counts for those affecting parameter sets of Kyber, while changing some important parameter set.

Run `make test` to build the following binaries `test_speed512`, `test_speed768`, `test_speed1024` with the following four parameter set. Those binaries are executed and the results are presented in the following tables.

```
29 #if KYBER_K == 2
30 #define KYBER_ETA1 3
31 #define KYBER_ETA2 2
32 #define KYBER_POLYCOMPRESSEDBYTES 128
33 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 320)
34 #elif KYBER_K == 3
35 #define KYBER_ETA1 2
36 #define KYBER_ETA2 2
37 #define KYBER_POLYCOMPRESSEDBYTES 128
38 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 320)
39 #elif KYBER_K == 4
40 #define KYBER_ETA1 2
41 #define KYBER_ETA2 2
42 #define KYBER_POLYCOMPRESSEDBYTES 160
43 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 352)
44 #endif
```

FIGURE 5.1: Test Result 1

```
29 #if KYBER_K == 2
30 #define KYBER_ETA1 3
31 #define KYBER_ETA2 2
32 #define KYBER_POLYCOMPRESSEDBYTES 96
33 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 352)
34 #elif KYBER_K == 3
35 #define KYBER_ETA1 2
36 #define KYBER_ETA2 2
37 #define KYBER_POLYCOMPRESSEDBYTES 96
38 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 352)
39 #elif KYBER_K == 4
40 #define KYBER_ETA1 2
41 #define KYBER_ETA2 2
42 #define KYBER_POLYCOMPRESSEDBYTES 128
43 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 384)
44 #endif
```

FIGURE 5.2: Test Result 2

```
29 #if KYBER_K == 2
30 #define KYBER_ETA1 3
31 #define KYBER_ETA2 2
32 #define KYBER_POLYCOMPRESSEDBYTES 160
33 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 288)
34 #elif KYBER_K == 3
35 #define KYBER_ETA1 2
36 #define KYBER_ETA2 2
37 #define KYBER_POLYCOMPRESSEDBYTES 160
38 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 288)
39 #elif KYBER_K == 4
40 #define KYBER_ETA1 2
41 #define KYBER_ETA2 2
42 #define KYBER_POLYCOMPRESSEDBYTES 192
43 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 352)
44 #endif
```

FIGURE 5.3: Test Result 3

```

29 #if KYBER_K == 2
30 #define KYBER_ETA1 5
31 #define KYBER_ETA2 3
32 #define KYBER_POLYCOMPRESSEDBYTES 128
33 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 320)
34 #elif KYBER_K == 3
35 #define KYBER_ETA1 4
36 #define KYBER_ETA2 4
37 #define KYBER_POLYCOMPRESSEDBYTES 128
38 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 320)
39 #elif KYBER_K == 4
40 #define KYBER_ETA1 4
41 #define KYBER_ETA2 4
42 #define KYBER_POLYCOMPRESSEDBYTES 160
43 #define KYBER_POLYVECCOMPRESSEDBYTES (KYBER_K * 352)
44 #endif

```

FIGURE 5.4: Test Result 4

Operation	$(d_u, d_v) = (10, 4)$		$(d_u, d_v) = (11, 3)$		$(d_u, d_v) = (9, 5)$	
	Median	Average	Median	Average	Median	Average
poly_compress	438	459	638	679	878	1045
poly_decompress	146	141	486	594	582	668
polyvec_compress	1818	2022	2162	2405	2202	2500
polyvec_decompress	1400	1461	1422	1487	1388	1515
indcpa_keypair	90868	92608	91738	94025	95892	99896
indcpa_enc	113626	116520	119824	120906	127896	132376
indcpa_dec	35406	35969	36828	37624	40498	42414

TABLE 5.1: Performance analysis of Kyber512 for different (d_u, d_v) values

Operation	$(d_u, d_v) = (10, 4)$		$(d_u, d_v) = (11, 3)$		$(d_u, d_v) = (9, 5)$	
	Median	Average	Median	Average	Median	Average
poly_compress	436	513	640	691	792	909
poly_decompress	128	145	484	517	562	561
polyvec_compress	2774	2957	3230	3464	3208	3248
polyvec_decompress	2138	2232	2128	2166	2026	2096
indcpa_keypair	153350	156456	149952	152526	157678	161190
indcpa_enc	185344	188819	179902	183067	197962	200925
indcpa_dec	48032	49386	47200	48338	52286	53266

TABLE 5.2: Performance analysis of Kyber768 for different (d_u, d_v) values

Operation	$(d_u, d_v) = (11, 5)$		$(d_u, d_v) = (10, 6)$		$(d_u, d_v) = (12, 4)$	
	Median	Average	Median	Average	Median	Average
poly_compress	766	830	438	515	754	852
poly_decompress	562	650	148	145	598	609
polyvec_compress	4328	4573	2312	2452	3706	3733
polyvec_decompress	2858	2926	1788	1827	2836	2864
indcpa_keypair	241666	245979	233366	238391	231204	234604
indcpa_enc	323800	329759	301240	306012	294878	299006
indcpa_dec	64248	65035	58368	59109	58606	59733

TABLE 5.3: Performance analysis of Kyber1024 for different (d_u, d_v) values

	$\mathbf{T}\eta_1$	$\mathbf{T}\eta_2$	KeyGen	Enc
Kyber512				
$(\eta_1 = 3, \eta_2 = 2)$	3350	2334	91214	114026
$(\eta_1 = 5, \eta_2 = 3)$	5309	4094	99275	125414
Kyber768				
$(\eta_1 = 2, \eta_2 = 2)$	2291	2327	155392	187702
$(\eta_1 = 4, \eta_2 = 4)$	3828	3842	163992	198387
Kyber1024				
$(\eta_1 = 2, \eta_2 = 2)$	2336	2334	245464	328873
$(\eta_1 = 4, \eta_2 = 4)$	3957	3947	257145	342649

TABLE 5.4: Performance analysis of Kyber with different η_1, η_2 values

$\mathbf{T}\eta_1$ and $\mathbf{T}\eta_2$ denotes the time required when a polynomial $f \in R_q$ is sampled according to B_η deterministically from 64η bytes of output of a pseudorandom function.

5.2 Security Analysis with respect to known attacks

Here we discuss Classical and quantum core-SVP hardness of the MLWE problem (treated as LWE problem) underlying Kyber for different proposed parameter sets.

The value b denotes the block dimension of BKZ (i.e., the dimension of the SVP considered in the core-SVP-hardness estimates), and m the number of used samples. Cost is given in log2 of operations and is the smallest cost for all possible choices of m and b . C denotes the communication cost.

Run `Kyber.py` with the following three parameter set and the results are presented in the following tables.

```
53 # Parameter sets
54 ps_light = KyberParameterSet(256, 2, 3, 2, 3329, 2**12, 2**10, 2**4, ke_ct=2)
55 ps_recommended = KyberParameterSet(256, 3, 2, 2, 3329, 2**12, 2**10, 2**4)
56 ps_paranoid = KyberParameterSet(256, 4, 2, 2, 3329, 2**12, 2**11, 2**5)
```

FIGURE 5.5: Test Result 5

```
53 # Parameter sets
54 ps_light = KyberParameterSet(256, 2, 3, 2, 3329, 2**12, 2**11, 2**3, ke_ct=2)
55 ps_recommended = KyberParameterSet(256, 3, 2, 2, 3329, 2**12, 2**11, 2**3)
56 ps_paranoid = KyberParameterSet(256, 4, 2, 2, 3329, 2**12, 2**12, 2**4)
```

FIGURE 5.6: Test Result 6

```
53 # Parameter sets
54 ps_light = KyberParameterSet(256, 2, 3, 2, 3329, 2**12, 2**9, 2**5, ke_ct=2)
55 ps_recommended = KyberParameterSet(256, 3, 2, 2, 3329, 2**12, 2**9, 2**5)
56 ps_paranoid = KyberParameterSet(256, 4, 2, 2, 3329, 2**12, 2**10, 2**6)
```

FIGURE 5.7: Test Result 7

	d	b	m	Core-SVP (classical)	Core-SVP (quantum)	δ	C
$(d_u = 10, d_v = 4)$						2^{-161}	(800, 768)
Primal Attack	999	406	486	118	107		
Dual Attack	1024	403	512	117	106		
$(d_u = 11, d_v = 3)$						2^{-148}	(800, 800)
Primal Attack	999	406	486	118	107		
Dual Attack	1024	403	512	117	106		
$(d_u = 9, d_v = 5)$						2^{-98}	(800, 736)
Primal Attack	999	406	486	118	107		
Dual Attack	1024	403	512	117	106		

TABLE 5.5: Security analysis of Kyber512 with different d_u, d_v values

```

53 # Parameter sets
54 ps_light = KyberParameterSet(256, 2, 5, 3, 3329, 2**12, 2**10, 2**4, ke_ct=2)
55 ps_recommended = KyberParameterSet(256, 3, 4, 4, 3329, 2**12, 2**10, 2**4)
56 ps_paranoid = KyberParameterSet(256, 4, 4, 4, 3329, 2**12, 2**11, 2**5)

```

FIGURE 5.8: Test Result 8

	d	b	m	Core-SVP (classical)	Core-SVP (quantum)	δ	C
$(d_u = 10, d_v = 4)$						2^{-165}	(1184, 1088)
Primal Attack	1419	626	650	183	166		
Dual Attack	1418	620	650	181	164		
$(d_u = 11, d_v = 3)$						2^{-151}	(1184, 1152)
Primal Attack	1419	626	650	183	166		
Dual Attack	1418	620	650	181	164		
$(d_u = 9, d_v = 5)$						2^{-99}	(1184, 1024)
Primal Attack	1419	626	650	183	166		
Dual Attack	1418	620	650	181	164		

TABLE 5.6: Security analysis of Kyber768 with different d_u, d_v values

	d	b	m	Core-SVP (classical)	Core-SVP (quantum)	δ	C
$(d_u = 11, d_v = 5)$						2^{-175}	(1568, 1568)
Primal Attack	1885	878	860	256	232		
Dual Attack	1862	868	838	253	230		
$(d_u = 12, d_v = 4)$						2^{-183}	(1568, 1664)
Primal Attack	1885	878	860	256	232		
Dual Attack	1862	868	838	253	230		
$(d_u = 10, d_v = 6)$						2^{-151}	(1568, 1472)
Primal Attack	1885	878	860	256	232		
Dual Attack	1862	868	838	253	230		

TABLE 5.7: Security analysis of Kyber1024 with different d_u, d_v values

	d	b	m	Core-SVP (classical)	Core-SVP (quantum)	δ	C
Kyber512							
						2^{-85}	(800, 768)
Primal Attack	1027	439	514	128	116		
Dual Attack	1027	515	436	127	115		
Kyber768							
						2^{-50}	(1184, 1088)
Primal Attack	1489	688	720	201	182		
Dual Attack	1487	719	683	199	181		
Kyber1024							
						2^{-47}	(1568, 1568)
Primal Attack	1936	961	911	281	254		
Dual Attack	1930	953	906	278	252		

TABLE 5.8: Security analysis of Kyber with different η_1, η_2 values

Chapter 6

Tweaks to Dilithium

6.1 Introduction

In this report, we propose and evaluate three lightweight modifications to increase signature variability:

- **Tweak 1:** Switching from SHAKE256 to SHA256 for challenge computation.
- **Tweak 2:** Expanding the challenge polynomial coefficient bounds.
- **Tweak 3:** Modifying the rejection sampling process.

6.2 Tweak 1: Switching Hash Function

6.2.1 Description

In the standard Dilithium implementation, the challenge value c is computed using SHAKE256. We replace this with SHA256 to observe its effect on signature variability.

6.2.2 Why It Works

SHA256 [23] is a secure, collision-resistant hash function. Replacing SHAKE256 with SHA256 still results in secure challenge generation, provided that all parts of the scheme use the same hash.

6.2.3 Implementation

```
// Initialize OpenSSL hash context
mdctx = EVP_MD_CTX_new();
md = EVP_sha3_256();
```

```
// Copy seed to hash input
memcpy(hash_input, seed, CTILDEBYTES);

// Initial hash
EVP_DigestInit_ex(mdctx, md, NULL);
EVP_DigestUpdate(mdctx, seed, CTILDEBYTES);
EVP_DigestFinal_ex(mdctx, buf, &md_len);

// Fill buffer with additional hashes for
// more randomness
for (i = 1; i < 4; i++) {
    // Update counter for domain separation
    counter[0] = i;

    // Append counter to seed
    memcpy(hash_input + CTILDEBYTES,
counter, 4);

    // Hash the seed+counter
    EVP_DigestInit_ex(mdctx, md, NULL);
    EVP_DigestUpdate(mdctx, hash_input,
CTILDEBYTES + 4);
    EVP_DigestFinal_ex(mdctx, buf + (i *
32), &md_len);
}
```

LISTING 6.1: Using SHA256 Instead of SHAKE256

6.2.4 Impact

This change increases signature variability without breaking compatibility. It also results in a shorter, fixed-length output (32 bytes) instead of a customizable XOF stream.

Although we used the OpenSSL implementation of SHA-256, it is also possible to leverage CPU-level SHA-256 instructions (SHA-NI) for improved performance.

6.3 Tweak 2: Expanding Challenge Coefficients

6.3.1 Description

Dilithium's challenge polynomial typically has coefficients in $\{-1, 0, 1\}$. We expand this to $\{-2, -1, 0, 1, 2\}$, optionally increasing the number of non-zero entries.

6.3.2 Why It Works

The challenge polynomial structure affects the signature. If the verifier uses the same polynomial rules, the signature remains valid.

6.3.3 Implementation

```
c → coeffs[i] = c → coeffs[b];
// Extract 3 bits from values to
determine coefficient in  $\{-2, -1, 0, 1, 2\}$ 
// Use modulo 5 to get value from 0 to 4,
then subtract 2 to get range -2 to 2
c → coeffs[b] = (signs & 7) % 5 - 2;
signs >= 3;

// If we've used 21 bits or more, refill
values
if(pos ≥ 8 && (i % 21) == 0) {
    if(pos + 8 ≤ SHAKE256_RATE) {
        signs = 0;
        for(unsigned int j = 0; j < 8; ++j)
            signs |= (uint64_t)buf[pos+j] << 8*
j;
        pos += 8;
    }
}
```

```
}
```

LISTING 6.2: Modified Challenge Coefficient Mapping

6.3.4 Impact

This tweak increases signature variability and allows fine-grained control over sparsity and magnitude of the challenge.

6.4 Tweak 3: Modified Rejection Sampling

6.4.1 Description

Rejection sampling ensures certain values stay within secure bounds. This tweak allows relaxing those bounds or randomly bypassing rejections.

6.4.2 Why It Works

Careful control of rejection criteria still ensures signatures verify correctly, while injecting variability.

6.4.3 Implementation

To apply this tweak, locate the rejection check within the signing function and modify it as follows to relax the bounds or introduce controlled randomness.

Option 1: Relax Rejection Bounds

```
if(polyveck_chknorm(&w0, GAMMA2 - BETA * 2))
    goto rej;
```

LISTING 6.3: Modified Rejection Condition

Option 2: Probabilistic Rejection Bypass

Introduce a bypass mechanism that allows some rejections to be probabilistically accepted:

```
if (cpolyveck_chknorm(&w0, GAMMA2 - BETA * 2))
{
    uint8_t bypass;
    randombytes(&bypass, 1);
```

```
    if (bypass % 10  $\neq$  0) {  
        continue; // Reject 90% of the time  
    }  
}
```

LISTING 6.4: Probabilistic Rejection Sampling

This tweak adds variability to the signing process without compromising verification correctness, as long as bounds are not excessively relaxed.

6.4.4 Impact

This introduces stochastic behavior into the rejection process, resulting in different signatures for the same message.

6.5 Benchmarking

To measure the performance implications of each tweak, we evaluated:

- Average time to generate a signature
- Verification success rate
- Signature size (if affected)

6.5.1 Experimental Setup

- CPU: Intel Xeon @ 2.80GHz
- OS: Ubuntu 22.04
- Compiler: GCC 11.3.0
- Timing method: `clock_gettime(CLOCK_MONOTONIC)`

6.5.2 Results

The following table (Table 6.1) summarizes the performance of the Dilithium scheme before and after applying the proposed tweak. While keypair generation time remains nearly constant, signature generation and verification show a noticeable increase in cycle counts. This trade-off reflects a shift toward enhanced cryptographic properties such as security or determinism, which are often prioritized in government-oriented deployments.

Operation	Before Tweak		After Tweak	
	Median (cycles)	Average (cycles)	Median (cycles)	Average (cycles)
Keypair	297,146	302,250	297,386	300,861
Sign	1,050,322	1,375,222	5,239,586	7,534,151
Verify	324,734	328,608	334,196	341,053

TABLE 6.1: Benchmark Results: Dilithium Before and After Applying Tweak

Chapter 7

Code Snippets and Repository Access

This thesis involved implementation-level modifications to the NIST reference codebases of both ML-KEM (Kyber) and ML-DSA (Dilithium). Due to the extensive nature of these codebases, only essential, illustrative snippets are included within the main text (see Chapter 5 and 6). Full source code is accessible via the following links.

Official Reference Implementations:

- ML-KEM (Kyber): <https://github.com/pq-crystals/kyber>
- ML-DSA (Dilithium): <https://github.com/pq-crystals/dilithium>

Tweaked Implementations for This Thesis:

The GitHub repository containing all proposed tweaks, benchmarking tools, and CLI scripts will be made available at the following link:

<https://github.com/username/pq-govt-tweaks>

The repository will be organized as follows:

- `kyber/` – Source code with modifications for noise distribution, compression, and other tweaks.
- `dilithium/` – Modified implementation with changes to rejection sampling and packing.
- `benchmarks/` – Scripts for cycle count comparison and performance profiling.
- `cli-tests/` – Minimal test harness to demonstrate encryption, signing, and verification.

This setup ensures reproducibility, transparency, and future extensibility, particularly for governmental or academic use cases. Access to the private repository can be granted upon request if the public link is not yet available.

Chapter 8

Conclusion, Summary, and Future Work

8.1 Conclusion

This thesis focused on the exploration and enhancement of two lattice-based post-quantum cryptographic schemes – ML-KEM (Kyber) and ML-DSA (Dilithium) – through implementation-level optimizations aimed at making them more suitable for real-world, especially governmental, deployment. With quantum threats looming over classical cryptography, the urgency to develop efficient and secure quantum-safe alternatives has never been greater.

The cryptographic community has largely focused on standardizing algorithms for theoretical soundness and asymptotic security. However, for large-scale adoption in operational systems such as military communication platforms, e-governance frameworks, and classified VPNs, it is essential to balance theoretical guarantees with performance, efficiency, and platform adaptability. This thesis bridges that gap by modifying key components of Kyber and Dilithium and demonstrating measurable improvements.

The central contributions of this work are as follows:

- Design and implementation of custom tweaks to Kyber and Dilithium reference codebases, targeting components like noise sampling, compression, encoding, and rejection sampling.
- Detailed benchmarking of cryptographic operations (Key Generation, Encryption/Signing, Decryption/Verification) before and after tweaks, using CPU cycle counts as the primary metric.

- Clear articulation of the security-performance trade-offs introduced by each tweak, ensuring no compromise on NIST-defined security levels.
- Contextual discussion on how these cryptographic primitives can be adapted in government-critical applications such as secure messaging, key exchanges, and digital signatures.

These outcomes reinforce the feasibility of deploying post-quantum algorithms not just in research prototypes but in mission-ready systems with constrained resources or strict performance requirements.

8.2 Key Takeaways

The following points summarize the major technical and conceptual insights gained through the course of this work:

- **Kyber and Dilithium are adaptable.** Despite being standardized with fixed parameters, both schemes allow room for optimizations without affecting their core security assumptions.
- **Noise sampling is a major performance lever.** In both Kyber and Dilithium, the method of generating small, discrete noise values significantly affects both execution time and timing attack resilience.
- **Compression and encoding have a direct impact on bandwidth efficiency.** Minor tweaks to packing formats and bit manipulations can yield better network utilization in bandwidth-constrained environments.
- **Cryptographic tweaks must be holistic.** A performance improvement in one module (e.g., faster sampling) must be evaluated in the broader context of system-level correctness, side-channel security, and conformance to cryptographic standards.
- **Government-ready PQC tools require modular, reproducible implementations.** Source-code modularity and CLI-based toolchains are vital to support audits, formal verification, and eventual deployment.

These takeaways are not limited to Kyber and Dilithium but extend to the general process of deploying lattice-based cryptographic primitives in real-world settings.

8.3 Future Work

The modifications proposed in this thesis form a foundational layer for continued exploration and refinement. Several future directions are open for deeper investigation:

8.3.1 1. Advanced Parameter Tuning

While NIST mandates certain security levels, practical deployment often benefits from fine-grained parameter tuning. For instance, smaller key sizes with slightly lower security margins may be acceptable in non-critical internal systems, whereas highly sensitive systems may benefit from stronger rejection criteria and higher entropy noise.

Further study is needed on how such tunings affect not just performance but also susceptibility to specific attacks like lattice sieving, dual attacks, or module subfield attacks.

8.3.2 2. Hybrid Noise Distributions

This thesis focused on centered binomial noise and its computational efficiency. However, hybrid distributions — such as combining binomial with sparse discrete Gaussians — could yield improved side-channel resistance while keeping sampling overhead minimal.

Future implementations could integrate timing-safe hybrid samplers and evaluate their effectiveness under side-channel stress testing.

8.3.3 3. Hardware-Centric Optimizations

The tweaks in this thesis are CPU-centric. With growing interest in PQC for embedded devices, FPGAs, and secure elements, hardware-focused implementations of these schemes (e.g., Verilog versions) will become essential. Mapping the tweaked operations to hardware with minimal area and power will be an important continuation.

Prof. Maitra's recommendation to explore beyond Raspberry Pi is apt; targeting platforms like RISC-V or ARM TrustZone can validate real-world viability.

8.3.4 4. Multi-scheme Integration and Policy-Aware Switching

For systems that need both encryption and digital signatures, tight integration of ML-KEM and ML-DSA into a single cryptographic suite — with shared resource pools and policy-driven toggling — can yield deployment benefits. This also includes setting thresholds on performance-vs-security trade-offs based on operational roles (e.g., tactical vs. strategic communications).

8.3.5 5. Formal Security Auditing and Compliance Layers

As PQC schemes move from research to deployment, formal verification becomes essential. One future direction is to write formal models of the tweaked schemes using tools like EasyCrypt or Cryptol, and validate them against expected side-channel, timing, and correctness properties.

Similarly, compliance with national and international security certifications (e.g., Common Criteria, FIPS 140-3) will require audit-friendly implementations and documentation.

8.3.6 6. PQC-Enabled Government Applications

Finally, the most impactful direction is the development of deployable prototypes (e.g., VPN tools, secure messaging apps, or firmware signing modules) that integrate the tweaked Kyber and Dilithium schemes. These could be used in classified communication systems, undersea networks, or secure field equipment.

Special emphasis should be placed on integration with existing platforms like the Indian Navy’s “Narad” application or similar mission platforms, ensuring quantum-resilient communication is not just theoretical, but operational.

8.4 Closing Remark

This thesis began with an urgent question: how can we make post-quantum cryptography not only secure, but usable and adaptable for national-level systems? Through targeted optimizations and a deep dive into two flagship lattice-based schemes, this work offers one practical answer — and lays the groundwork for future innovation in a domain that will shape the next generation of secure communication.

Bibliography

- [1] Miklós Ajtai. “Generating Hard Instances of Lattice Problems”. In: *Quaderni di Matematica* 13 (2004). Preliminary version in STOC 1996, pp. 1–32.
- [2] Paul Benioff. “The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines”. In: *Journal of Statistical Physics* 22.5 (1980), pp. 563–591. DOI: [10.1007/bf01011339](https://doi.org/10.1007/bf01011339).
- [3] Dan Boneh et al. “Fully Key-Homomorphic Encryption, Arithmetic Circuit ABE and Compact Garbled Circuits”. In: *EUROCRYPT*. 2014, pp. 533–556.
- [4] Joppe Bos et al. “CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM”. In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. To appear. IEEE, 2018. URL: <https://eprint.iacr.org/2017/634>.
- [5] CRYSTALS-Dilithium Team. *CRYSTALS-Dilithium Specification*. Accessed: 2025-06-07. 2023. URL: <https://pq-crystals.org/dilithium>.
- [6] Whitfield Diffie and Martin E. Hellman. “New directions in cryptography”. In: *IEEE Transactions on Information Theory* IT-22.6 (1976), pp. 644–654.
- [7] Léo Ducas. “Shortest vector from lattice sieving: a few dimensions for free”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 125–145.
- [8] Léo Ducas et al. “Lattice signatures and bimodal Gaussians”. In: *Annual Cryptology Conference*. Springer. 2013, pp. 40–56.
- [9] Morris J Dworkin. “SHA-3 standard: Permutation-based hash and extendable-output functions”. In: (2015). URL: <https://doi.org/10.6028/NIST.FIPS.202>.
- [10] Eiichiro Fujisaki and Tatsuaki Okamoto. “Secure integration of asymmetric and symmetric encryption schemes”. In: *Advances in Cryptology - CRYPTO ’99*. 1999, pp. 537–554. URL: https://link.springer.com/chapter/10.1007/3-540-48405-1_34.
- [11] Sanjam Garg et al. “Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits”. In: *FOCS*. 2013, pp. 40–49.

- [12] Craig Gentry. “A Fully Homomorphic Encryption Scheme”. PhD thesis. Stanford University, 2009. URL: <http://crypto.stanford.edu/craig>.
- [13] Oded Goldreich. “Foundations of Cryptography: Volume 1, Basic Tools”. In: *Cambridge University Press* 1 (2001), pp. 1–372.
- [14] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. “Attribute-Based Encryption for Circuits”. In: *STOC*. 2013, pp. 545–554.
- [15] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. “NSS: An NTRU Lattice-Based Signature Scheme”. In: *EUROCRYPT*. 2001, pp. 211–228.
- [16] Hugo Krawczyk and Kristin Lauter. “Lattice Cryptography for the Internet”. In: *International Workshop on Post-Quantum Cryptography* 8772 (2014), pp. 197–219.
- [17] Adeline Langlois and Damien Stehlé. “Worst-case to average-case reductions for module lattices”. In: *Designs, Codes and Cryptography* 75.3 (2015), pp. 565–599. URL: <https://eprint.iacr.org/2012/090>.
- [18] A. K. Lenstra, H. W. Lenstra, and L. Lovász. “Factoring polynomials with rational coefficients”. In: *Mathematische Annalen* 261.4 (Dec. 1982), pp. 515–534. DOI: [10.1007/BF01457454](https://doi.org/10.1007/BF01457454).
- [19] Vadim Lyubashevsky. “Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2009, pp. 598–616.
- [20] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “On ideal lattices and learning with errors over rings”. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. Springer, 2010, pp. 1–23. URL: <http://www.iacr.org/archive/eurocrypt2010/66320288/66320288.pdf>.
- [21] Daniele Micciancio and Oded Regev. “Worst-case to average-case reductions based on Gaussian measures”. In: *SIAM Journal on Computing* 37.1 (2007), pp. 267–302.
- [22] Open Quantum Safe Project. *Open Quantum Safe*. Accessed: 2025-06-07. 2023. URL: <https://openquantumsafe.org>.
- [23] OpenSSL Project. *OpenSSL EVP_sha256 Documentation*. Accessed: 2025-06-07. 2023. URL: https://www.openssl.org/docs/man3/EVP_sha256.html.
- [24] Chris Peikert. “Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem: Extended Abstract”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC)*. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 333–342. ISBN: 9781605585062. DOI: [10.1145/1535474.1535474](https://doi.org/10.1145/1535474.1535474).

- 1145/1536414.1536461. URL: <https://doi.org/10.1145/1536414.1536461>.
- [25] Oded Regev. “On Lattices, Learning with Errors, Random Linear Codes, and Cryptography”. In: *Journal of the ACM (JACM)* 56.6 (2009). ISSN: 0004-5411. DOI: [10.1145/1568318.1568324](https://doi.org/10.1145/1568318.1568324). URL: <https://doi.org/10.1145/1568318.1568324>.
- [26] 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)*. New York, NY, USA: Association for Computing Machinery, 2005, pp. 84–93. DOI: [10.1145/1060590.1060603](https://doi.org/10.1145/1060590.1060603). URL: <https://doi.org/10.1145/1060590.1060603>.
- [27] R. L. Rivest, L. Adleman, and M. L. Dertouzos. “On Data Banks and Privacy Homomorphisms”. In: *Foundations of Secure Computation* 4.11 (1978), pp. 169–180.
- [28] R. L. Rivest, A. Shamir, and L. M. Adleman. “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2 (1978), pp. 120–126.
- [29] C. Schnorr. “Efficient signature generation by smart cards”. In: *Journal of Cryptology* 4.3 (1991). Preliminary version in CRYPTO 1989, pp. 161–174. DOI: [10.1007/BF00196725](https://doi.org/10.1007/BF00196725).
- [30] Benjamin Schumacher. “Quantum coding”. In: *Physical Review A* 51.4 (1995), pp. 2738–2747. DOI: [10.1103/physreva.51.2738](https://doi.org/10.1103/physreva.51.2738).
- [31] Peter W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509.
- [32] Peter W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: [10.1137/S0097539795293172](https://doi.org/10.1137/S0097539795293172).