Master Degree Project

Lattice based cryptographic hash functions

Author: Christos Savvidis
Supervisor: Marcus Nilsson
Examiner: Stefan Wagner
Examiner: Andrei Khrennikov
Semester: Spring 2026
Subject: Mathematics

Abstract

This thesis investigates the security of lattice-based cryptographic hash functions, with a specific focus on the Ring-Short Integer Solution (Ring-SIS) problem in the context of Post-Quantum Cryptography (PQC). As modern cryptographic standards increasingly rely on lattice-based assumptions, it becomes important to understand how the algebraic structure of the underlying polynomial rings affects security.

The main contribution of this thesis is a cryptanalysis of the cyclic quotient ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$. We show that the factorization of $x^n - 1$, in particular the presence of the linear factor $x - 1$, introduces a structural weakness. The all-ones polynomial $\widetilde{e}(x) = 1 + x + \cdots + x^{n-1}$ generates a rank-one invariant submodule, which causes a dimension collapse in the Ring-SIS relation. As a result, the high-dimensional collision search is reduced to a one-dimensional modular relation over $\mathbb{Z}_q$.

To exploit this collapse, the thesis develops an explicit collision-generation pipeline based on birthday-style search methods, including meet-in-the-middle techniques and Wagner's generalized birthday framework. These theoretical constructs are translated into practice by formulating detailed pseudocode algorithms and developing a custom Python implementation. The analysis shows that, in the cyclic setting, the main collision-search cost is governed primarily by the number of ring blocks rather than by the ring dimension n.

The thesis also contrasts the cyclic construction with the negacyclic quotient ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$. When n is a power of two, this modulus avoids the specific evaluation-at-one collapse caused by the factor $x - 1$, while still supporting efficient polynomial arithmetic through negacyclic convolution and the Number Theoretic Transform (NTT).

Finally, experimental evaluations of the scalar meet-in-the-middle search support the theoretical analysis. The practical results support the conclusion that the cyclic modulus $x^n - 1$ is unsuitable for collision-resistant ring-based hashing, while the negacyclic setting avoids this particular structural vulnerability.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Marcus Nilsson, for his guidance, support, and valuable feedback throughout the development of this thesis.

I would also like to thank my examiners, Stefan Wagner and Andrei Khrennikov, for their time, comments, and evaluation of this work.

Finally, I am grateful to my family and friends for their encouragement and support during my studies.

Contents

1 Introduction **1**

2 Mathematical and Cryptographic Background **2**
2.1 Notation and Complexity Conventions 2
2.2 Number-Theoretic Transform 3
2.2.1 Primitive n-th Roots of Unity 3
2.2.2 Definition and Properties of the Standard NTT (NTT_ω) 4
2.2.3 Primitive $2n$-th Roots of Unity 5
2.2.4 Definition and Properties of the Negative-Wrapped NTT (NTT_ψ) . 5
2.3 Lattices and Related Problems 6
2.3.1 Lattices 6
2.3.2 The Shortest Vector Problem (SVP) 9
2.3.3 Definition of the SIS Problem 10
2.4 Cryptographic Hash Functions 11
2.4.1 Definition of a Cryptographic Hash Function 11
2.4.2 The Compression Function 12

3 Security Analysis of Cyclic and Negacyclic Ring-Based Compression **12**
3.1 Foundations of Lattice-Based Hashing 12
3.1.1 The SIS Problem and Average-Case Hardness 12
3.1.2 Complexity Bottlenecks of Unstructured Lattices 13
3.2 Ring-SIS: Structural Efficiency and Isomorphisms 14
3.2.1 Circulant Matrices and Algebraic Structure 14
3.2.2 Polynomial Quotient Rings and the Isomorphism ψ 15
3.2.3 Computational Optimization via NTT 17
3.3 Cryptanalysis of the Cyclic Modulus $x^n - 1$ 18
3.3.1 Reducibility and the Presence of Zero Divisors 18
3.3.2 The Scalar Reduction Lemma and Dimension Collapse 19
3.3.3 The Invariant Submodule and the Rank-One Collapse 19
3.3.4 Norm Constraints and Collision Existence Guarantee 20
3.4 Algorithmic Production of Collisions 23
3.4.1 Step-by-Step Collision Generation Methodology 23
3.5 The Transition to $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and the Mitigation of Structural Vulnerabilities 25
3.5.1 The Geometry of the Solution Space and Ideal Lattices 27

4 The Generalized Birthday Problem **28**
4.1 The Generalized Birthday Paradigm 28
4.2 Join-Based Search for the Two-List Birthday Problem 31
4.3 The Four-List Birthday Algorithm 31
4.4 The k-Tree Algorithm for Modular Arithmetic 33
4.5 The k-Tree Attack on the Scalar Problem 34
4.6 The k-Tree Attack on the Full Negacyclic Problem 35

5 Implementation and Experimental Evaluation **43**
5.1 Experimental Setup 43
5.2 Example over the Cyclic Ring $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$ 43
5.3 Example over the Negacyclic Ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$ 47

5.4 Running Time Analysis 51
5.5 Memory Usage Analysis 54
5.5.1 Cyclic Rank-One Collapse Attack 55
5.5.2 Negacyclic Wagner Attack 57
5.6 Comparison of the Cyclic and Negacyclic Cases 59

6 Discussion **60**

References **61**

A Appendix 1 - Algorithms **63**

B Appendix 2 - Source Code **74**

1 Introduction

For several decades, the security of digital communication has relied heavily on public-key cryptographic systems such as RSA and Elliptic Curve Cryptography (ECC). These systems are based on the assumed hardness of classical number theoretic problems, mainly integer factorization and the discrete logarithm problem. However, Shor's quantum algorithm showed that these problems can be solved efficiently on a sufficiently powerful quantum computer. This creates the need for cryptographic constructions whose security is not based on these classical assumptions.

One of the main directions in post-quantum cryptography is lattice-based cryptography. Lattice-based schemes are important because they are efficient and because their security can often be related to well-studied hard problems on lattices. A central example is the Short Integer Solution problem (SIS), which was introduced by Ajtai. Ajtai's result showed that solving certain average-case SIS instances is at least as hard as solving worst-case lattice problems, such as approximate versions of the Shortest Vector Problem. This gives a strong theoretical justification for using SIS-based assumptions in cryptography [1].

In practice, however, standard SIS constructions are inefficient because they use large unstructured matrices. Ring-SIS improves this by replacing ordinary matrix multiplication with polynomial arithmetic in quotient rings. This gives smaller public keys and faster evaluation, especially when polynomial multiplication is implemented using convolution techniques such as the Number Theoretic Transform. This efficiency is one of the main reasons why ring-based and module-based lattice constructions are widely used in modern post-quantum cryptography.

The additional algebraic structure must nevertheless be treated carefully. While it improves efficiency, it can also introduce weaknesses. This thesis studies this issue for Ring-SIS-style compression functions. The main question is how the choice of polynomial modulus affects collision resistance. In particular, we compare the cyclic ring

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle$$

with the negacyclic ring

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle.$$

The cyclic modulus $x^n - 1$ has the factorization

$$x^n - 1 = (x - 1)(1 + x + \cdots + x^{n-1}).$$

This creates the all-ones polynomial

$$\widetilde{e}(x) = 1 + x + \cdots + x^{n-1},$$

which generates a rank-one invariant submodule. As a consequence, the Ring-SIS collision search over the cyclic ring collapses from a vector problem in $\mathbb{Z}_q^n$ to a one-dimensional modular relation over $\mathbb{Z}_q$. This structural collapse gives an efficient way to construct collisions.

The negacyclic modulus $x^n + 1$, in contrast, avoids this particular evaluation-at-one collapse when n is a power of two. Therefore, the same rank-one attack is not available. In the negacyclic case, the collision search must be performed in the full vector space $\mathbb{Z}_q^n$. To study this case experimentally, this thesis uses a Wagner-style generalized birthday approach [2].

The aim of this thesis is to show that the cyclic modulus $x^n - 1$ is structurally unsuitable for collision-resistant Ring-SIS hashing, while the negacyclic modulus $x^n + 1$ avoids this

specific weakness. The analysis is both theoretical and experimental: we prove the rank-one collapse in the cyclic case, describe explicit collision-generation algorithms, and compare their running time, memory usage, and success rate.

The thesis is organized as follows. Chapter 1 introduces the motivation and the main problem studied in the thesis. Chapter 2 presents the necessary mathematical and cryptographic background, including asymptotic notation, the Number Theoretic Transform, lattices, SIS, Ring-SIS, and cryptographic hash functions. Chapter 3 gives the theoretical security analysis of cyclic and negacyclic ring-based compression, with emphasis on the rank-one collapse caused by the factor $x - 1$. Chapter 4 presents the generalized birthday problem and Wagner's tree-based method, which are used for the collision-generation algorithms. Chapter 5 contains the implementation and experimental evaluation, including examples, running-time measurements, memory-usage measurements, and a comparison of the cyclic and negacyclic cases. Chapter 6 discusses their significance and limitations.

Finally, the Appendices provide detailed pseudocode for the implemented algorithms alongside the complete Python source code developed for this research. It should be explicitly noted that this Python implementation was constructed strictly for demonstration and proof-of-concept purposes. While it is mathematically faithful to the theoretical cryptanalysis and correctly models the underlying algebraic structures, it is not engineered as a highly optimized, production-ready library. Consequently, execution speed and memory optimization were not the primary design objectives, and the provided code should be viewed with all associated reservations regarding large-scale performance.

2 Mathematical and Cryptographic Background

In this chapter, we review relevant literature and theoretical background.

2.1 Notation and Complexity Conventions

Definition 2.1 (Asymptotic Notation)**.** Let $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be two non-negative functions.

We write

$$f(n) = O(g(n))$$

if there exist constants $C > 0$ and $n_0 \in \mathbb{N}$ such that

$$f(n) \leq Cg(n)$$

for every $n \geq n_0$. Thus, $O(g(n))$ denotes an asymptotic upper bound.

We write

$$f(n) = \Omega(g(n))$$

if there exist constants $c > 0$ and $n_0 \in \mathbb{N}$ such that

$$f(n) \geq cg(n)$$

for every $n \geq n_0$. Thus, $\Omega(g(n))$ denotes an asymptotic lower bound.

We write

$$f(n) = \Theta(g(n))$$

if there exist constants $c_1, c_2 > 0$ and $n_0 \in \mathbb{N}$ such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

for every $n \geq n_0$. Thus, $\Theta(g(n))$ denotes an asymptotically tight bound.

Definition 2.2 (Soft Asymptotic Notation)**.** The notation

$$\widetilde{O}(g(n))$$

suppresses polylogarithmic factors in an asymptotic upper bound. More precisely,

$$f(n) = \widetilde{O}(g(n))$$

means that there exists a constant $c > 0$ such that

$$f(n) = O(g(n)(\log n)^c)\,.$$

Similarly,

$$f(n) = \widetilde{\Omega}(g(n))$$

means that there exists a constant $c > 0$ such that

$$f(n) = \Omega\left(\frac{g(n)}{(\log n)^c}\right).$$

Finally,

$$f(n) = \widetilde{\Theta}(g(n))$$

means that

$$f(n) = \widetilde{O}(g(n)) \quad \text{and} \quad f(n) = \widetilde{\Omega}(g(n)).$$

The base of the logarithm is irrelevant for asymptotic estimates.

2.2 Number-Theoretic Transform

2.2.1 Primitive n-th Roots of Unity

Definition 2.3 (Primitive n-th Root of Unity Modulo q [3])**.** Let $q \in \mathbb{N}$ with $q \geq 2$, and let

$$\mathbb{Z}_q := \mathbb{Z}/q\mathbb{Z}$$

be the corresponding ring of integers modulo q. Let $n \in \mathbb{N}$ with $n \geq 2$.

An element ω of the multiplicative group $(\mathbb{Z}_q^\times, \cdot)$, where

$$\mathbb{Z}_q^\times = \{a \in \mathbb{Z}_q \mid \gcd(a, q) = 1\},$$

is called a primitive n-th root of unity modulo q if the following conditions hold

$$\omega^n \equiv 1 \pmod{q},$$

and

$$\omega^k \not\equiv 1 \pmod{q} \quad \text{for every integer } 1 \leq k < n.$$

Equivalently, the element ω has order exactly n in the multiplicative group $\mathbb{Z}_q^\times$, and its powers correspond to the roots of the polynomial $x^n - 1$ in $\mathbb{Z}_q$.

2.2.2 Definition and Properties of the Standard NTT (NTT_ω)

Definition 2.4 (Number Theoretic Transform (NTT))**.** Let $q \in \mathbb{N}$ with $q \geq 2$ and consider the ring $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Let $n \in \mathbb{N}$ with $n \geq 2$, and let $\omega \in \mathbb{Z}_q$ be a primitive n-th root of unity modulo q.

The Number Theoretic Transform (NTT) is defined as the linear mapping (a $\mathbb{Z}_q$-module homomorphism)

$$\mathrm{NTT}_\omega : \mathbb{Z}_q^n \to \mathbb{Z}_q^n,$$

which maps the coefficient vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^\top \in \mathbb{Z}_q^n$ to the vector $\hat{\mathbf{a}} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})^\top \in \mathbb{Z}_q^n$, where

$$\hat{a}_j = \sum_{i=0}^{n-1} a_i \omega^{ij} \pmod{q}, \qquad j = 0, 1, \ldots, n-1.$$

In matrix form, the transformation can be written as

$$\hat{\mathbf{a}} = \mathbf{W}\mathbf{a},$$

where $\mathbf{W} \in \mathbb{Z}_q^{n \times n}$ is the transformation matrix with entries $W_{j,i} = \omega^{ij} \pmod{q}$.

Definition 2.5 (Inverse Number-Theoretic Transform (INTT)[3])**.** Let $q \in \mathbb{Z}$ with $q \geq 2$, and consider the ring $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Let $n \in \mathbb{N}$, and let $\omega \in \mathbb{Z}_q$ be a **principal n-th root of unity** modulo q.

The **Inverse Number-Theoretic Transform (INTT)** is defined as the linear mapping (a $\mathbb{Z}_q$-module homomorphism)

$$\mathrm{INTT}_\omega : \mathbb{Z}_q^n \to \mathbb{Z}_q^n,$$

which maps the transformed vector $\hat{\mathbf{a}} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})^\top \in \mathbb{Z}_q^n$ to the coefficient vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^\top \in \mathbb{Z}_q^n$, which additionally requires the condition $\gcd(n, q) = 1$ (so that $n^{-1} \in \mathbb{Z}_q$ exists), and is given explicitly by

$$a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega^{-ij} \pmod{q}, \qquad i = 0, 1, \ldots, n-1.$$

In matrix form, the transformation can be written as

$$\mathbf{a} = n^{-1}\mathbf{W}^*\hat{\mathbf{a}},$$

where $\mathbf{W}^* \in \mathbb{Z}_q^{n \times n}$ is the matrix with entries

$$(\mathbf{W}^*)_{i,j} = \omega^{-ij} \pmod{q}.$$

Theorem 2.1 (Cyclic Convolution Theorem [3])**.** *Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$ be the coefficient vectors of two polynomials $a(x), b(x) \in \mathbb{Z}_q[x]$. The cyclic convolution (or positive-wrapped convolution) of $\mathbf{a}$ and $\mathbf{b}$ corresponds to the product in the ring $R = \mathbb{Z}_q[x]/(x^n - 1)$. This can be calculated efficiently using the standard Number Theoretic Transform (NTT_ω) as follows:*

$$\mathbf{c} = \mathrm{INTT}_\omega(\mathrm{NTT}_\omega(\mathbf{a}) \circ \mathrm{NTT}_\omega(\mathbf{b})),$$

where $\circ$ denotes component-wise multiplication in $\mathbb{Z}_q$.

Remark 2.1. The vector $\mathbf{c}$ computed above represents the coefficients of the polynomial $c(x) = a(x) \cdot b(x) \pmod{x^n - 1}$. This property assumes that the transform is defined using a primitive n-th root of unity ω.

2.2.3 Primitive $2n$-th Roots of Unity

Definition 2.6 (Primitive $2n$-th Root of Unity for Negative-Wrapped NTT [3])**.** Let q, n and $\mathbb{Z}_q$ be as defined above, and let ω be a primitive n-th root of unity modulo q.

An element ψ of the multiplicative group $\mathbb{Z}_q^\times$ is called a **primitive $2n$-th root of unity** (associated with ω) if the following conditions hold:

$$\psi^2 \equiv \omega \pmod{q},$$

and

$$\psi^n \equiv -1 \pmod{q}.$$

Equivalently, the element ψ has order exactly $2n$ in the multiplicative group $\mathbb{Z}_q^\times$, and its powers correspond to the roots of the polynomial $x^n + 1$ in $\mathbb{Z}_q$.

2.2.4 Definition and Properties of the Negative-Wrapped NTT (NTT_ψ)

Definition 2.7 (Negative-Wrapped Number Theoretic Transform (NTT_ψ)[3])**.** Let q, n and $\mathbb{Z}_q$ be as defined previously. Let $\psi \in \mathbb{Z}_q$ be a primitive $2n$-th root of unity modulo q (such that $\psi^2 = \omega$).

The Negative-Wrapped Number Theoretic Transform (NTT_ψ) is defined as the linear mapping

$$\mathrm{NTT}_\psi : \mathbb{Z}_q^n \to \mathbb{Z}_q^n,$$

which maps the coefficient vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^\top \in \mathbb{Z}_q^n$ to the vector $\hat{\mathbf{a}} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})^\top \in \mathbb{Z}_q^n$, where

$$\hat{a}_j = \sum_{i=0}^{n-1} a_i \psi^{2ij+i} \pmod{q}, \qquad j = 0, 1, \ldots, n-1.$$

In matrix form, the transformation can be written as

$$\hat{\mathbf{a}} = \mathbf{W}_\psi \mathbf{a},$$

where $\mathbf{W}_\psi \in \mathbb{Z}_q^{n \times n}$ is the transformation matrix with entries $(\mathbf{W}_\psi)_{j,i} = \psi^{2ij+i} \pmod{q}$.

Definition 2.8 (Negative-Wrapped Inverse Number-Theoretic Transform (INTT_ψ)[3])**.** Let q, n and $\mathbb{Z}_q$ be as defined previously. Let $\psi \in \mathbb{Z}_q$ be a **primitive $2n$-th root of unity** modulo q (such that $\psi^2 = \omega$).

The **Negative-Wrapped Inverse Number-Theoretic Transform** (INTT_ψ) is defined as the linear mapping (a $\mathbb{Z}_q$-module homomorphism)

$$\mathrm{INTT}_\psi : \mathbb{Z}_q^n \to \mathbb{Z}_q^n,$$

which maps the transformed vector $\hat{\mathbf{a}} = (\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1})^\top \in \mathbb{Z}_q^n$ to the coefficient vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^\top \in \mathbb{Z}_q^n$. This transformation requires the condition $\gcd(n, q) = 1$ (so that $n^{-1} \in \mathbb{Z}_q$ exists) and uses the substitution $\omega = \psi^2$ to yield the explicit formula:

$$a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \psi^{-(2ij+i)} \pmod{q}, \qquad i = 0, 1, \ldots, n-1.$$

In matrix form, the transformation can be written as

$$\mathbf{a} = n^{-1}\mathbf{W}_{\psi}^{*}\hat{\mathbf{a}},$$

where $\mathbf{W}_{\psi}^{*} \in \mathbb{Z}_q^{n\times n}$ is the matrix with entries

$$(\mathbf{W}_{\psi}^{*})_{i,j} = \psi^{-(2ij+i)} \pmod{q}.$$

Theorem 2.2 (Negacyclic Convolution Theorem [3])**.** *Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$ be the coefficient vectors of two polynomials $a(x), b(x) \in \mathbb{Z}_q[x]$. The negacyclic convolution (or negative-wrapped convolution) of $\mathbf{a}$ and $\mathbf{b}$ corresponds to the product in the ring $R = \mathbb{Z}_q[x]/(x^n+1)$. This can be calculated efficiently using the Negative-Wrapped Number Theoretic Transform (NTT_{ψ}) as follows:*

$$\mathbf{c} = \mathrm{INTT}_{\psi}(\mathrm{NTT}_{\psi}(\mathbf{a}) \circ \mathrm{NTT}_{\psi}(\mathbf{b})),$$

where $\circ$ denotes component-wise multiplication in $\mathbb{Z}_q$.

Remark 2.2. The vector $\mathbf{c}$ computed above represents the coefficients of the polynomial $c(x) = a(x) \cdot b(x) \pmod{x^n + 1}$. This property assumes that the transform is defined using a primitive $2n$-th root of unity ψ.

2.3 Lattices and Related Problems

2.3.1 Lattices

Definition 2.9 (Euclidean Norm)**.** For a vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$, the **Euclidean norm** (or ℓ_2-norm) is defined as:

$$\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}.$$

Unless otherwise specified, any reference to the length of a vector implies the Euclidean norm.

Definition 2.10 (Lattice [4])**.** Let $m, n \in \mathbb{N}$ with $m \leq n$. A **lattice** $\mathcal{L} \subseteq \mathbb{R}^n$ is the set of all integer linear combinations of m linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_m \in \mathbb{R}^n$:

$$\mathcal{L} = \left\{ \sum_{i=1}^{m} z_i \mathbf{b}_i : z_i \in \mathbb{Z} \right\}.$$

The set of vectors $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ is called a **basis** of the lattice. The integer m is called the **rank** of the lattice, and n is the **dimension**. (If $m = n$, the lattice is called *full-rank*).

Remark 2.3 (Integer Lattices and Matrix Notation). While the general definition allows the basis vectors to have real coordinates ($\mathbf{b}_i \in \mathbb{R}^n$), in the context of lattice-based cryptography (and specifically for SWIFFT), we restrict our attention to **integer lattices**, where the basis vectors belong to $\mathbb{Z}^n$.

Furthermore, it is standard to represent the basis as a matrix $\mathbf{B} \in \mathbb{Z}^{n\times m}$, formed by taking the basis vectors as its columns:

$$\mathbf{B} = [\mathbf{b}_1 \mid \mathbf{b}_2 \mid \cdots \mid \mathbf{b}_m].$$

Using this notation, the lattice can be compactly defined using matrix-vector multiplication:

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{Bz} : \mathbf{z} \in \mathbb{Z}^m\}.$$

Definition 2.11 (Sublattice [4])**.** Let $\mathcal{L}$ and $\mathcal{L}'$ be lattices in $\mathbb{R}^n$. We say that $\mathcal{L}'$ is a **sublattice** of $\mathcal{L}$ if $\mathcal{L}' \subseteq \mathcal{L}$.

Remark 2.4 (Notation and Conventions). Henceforth, unless otherwise stated, we will assume that all lattices discussed in this thesis are **full-rank integer lattices** ($\mathcal{L} \subseteq \mathbb{Z}^n$ and $m = n$). Consequently, their basis matrices $\mathbf{B}$ will be square matrices in $\mathbb{Z}^{n\times n}$.

Definition 2.12 (Fundamental Parallelepiped [4])**.** Let $\mathcal{L}$ be a lattice of rank n with basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$. The **fundamental parallelepiped** associated with $\mathbf{B}$ is the set of points formed by linear combinations of the basis vectors with coefficients in the half-open interval $[0, 1)$:

$$\mathcal{P}(\mathbf{B}) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : 0 \le x_i < 1 \right\} = \{\mathbf{Bx} : \mathbf{x} \in [0, 1)^n\}.$$

Remark 2.5. The fundamental parallelepiped $\mathcal{P}(\mathbf{B})$ does not depend solely on the lattice $\mathcal{L}$, but specifically on the chosen basis $\mathbf{B}$. However, its volume is an invariant of the lattice. Geometrically, the translates of $\mathcal{P}(\mathbf{B})$ by lattice vectors partition (tile) the entire space $\mathbb{R}^n$.

Theorem 2.3 (Characterization of Lattice Bases [4])**.** *Let $\mathcal{L} \subseteq \mathbb{Z}^n$ be a full-rank lattice. Let $\mathbf{B}_1 \in \mathbb{Z}^{n\times n}$ be a basis of $\mathcal{L}$. A matrix $\mathbf{B}_2 \in \mathbb{Z}^{n\times n}$ is also a basis for the same lattice $\mathcal{L}$ if and only if there exists a **unimodular matrix** $\mathbf{U} \in \mathbb{Z}^{n\times n}$ such that:*

$$\mathbf{B}_2 = \mathbf{B}_1\mathbf{U}.$$

A matrix $\mathbf{U}$ is called unimodular if it has integer entries and its determinant is $\det(\mathbf{U}) = \pm 1$.

Remark 2.6 (The Basis Reduction Problem). This equivalence of bases is the foundation of lattice-based cryptography. Although $\mathbf{B}_1$ and $\mathbf{B}_2$ generate the same set of points, they reveal different amounts of geometric information:

- A **"Good" Basis** consists of short, nearly orthogonal vectors. It allows for efficient solving of lattice problems (like the Closest Vector Problem). In cryptosystems (e.g., GGH, NTRU), this serves as the **private key**.
- A **"Bad" Basis** consists of long, highly skewed vectors. It makes lattice problems computationally infeasible to solve directly. This serves as the **public key**.

The security relies on the asymmetry of this transformation: while it is trivial to transform a good basis into a bad one (by multiplying with a random unimodular matrix $\mathbf{U}$), the reverse process—recovering a good basis given only a bad one—is an computationally hard problem known as **Lattice Basis Reduction**.

Definition 2.13 (Volume of a Lattice [4])**.** Let $\mathcal{L}$ be a lattice of rank m in $\mathbb{R}^n$ with basis $\mathbf{B} \in \mathbb{R}^{n\times m}$. The **volume** (or determinant) of the lattice, denoted by $\mathrm{vol}(\mathcal{L})$, is defined as:

$$\mathrm{vol}(\mathcal{L}) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}.$$

Geometrically, this value corresponds to the m-dimensional volume of the **fundamental parallelepiped** $\mathcal{P}(\mathbf{B})$ associated with the basis.

Ideally, in lattice-based cryptography (including the analysis of the SWIFFT hash function), we primarily deal with **full-rank lattices**, where the rank equals the dimension ($m = n$). In this specific case, the basis matrix $\mathbf{B}$ is a square $n \times n$ matrix, and the volume formula simplifies to the absolute value of its determinant:

$$\mathrm{vol}(\mathcal{L}) = |\det(\mathbf{B})|.$$

Proposition 2.1 (Invariance of Lattice Volume)**.** *The volume of a lattice $\mathcal{L}$ is an invariant quantity, meaning it is independent of the specific basis chosen to represent the lattice. Specifically, if $\mathbf{B}_1$ and $\mathbf{B}_2$ are two arbitrary bases for the same lattice $\mathcal{L}$, then:*

$$|\det(\mathbf{B}_1)| = |\det(\mathbf{B}_2)| = \mathrm{vol}(\mathcal{L}).$$

Proof. From the characterization of lattice bases, we know that if $\mathbf{B}_1$ and $\mathbf{B}_2$ generate the same lattice, they are related by a unimodular matrix $\mathbf{U}$ such that $\mathbf{B}_2 = \mathbf{B}_1\mathbf{U}$, where $\det(\mathbf{U}) = \pm 1$. Using the multiplicative property of determinants, we have:

$$\det(\mathbf{B}_2) = \det(\mathbf{B}_1\mathbf{U}) = \det(\mathbf{B}_1) \cdot \det(\mathbf{U}).$$

Taking absolute values on both sides:

$$|\det(\mathbf{B}_2)| = |\det(\mathbf{B}_1)| \cdot |\det(\mathbf{U})| = |\det(\mathbf{B}_1)| \cdot 1 = |\det(\mathbf{B}_1)|.$$

Thus, the volume remains constant regardless of the basis choice. □

Proposition 2.2 (Volume of Sublattices)**.** *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be full-rank lattices in $\mathbb{R}^n$. If $\mathcal{L}_1$ is a sublattice of $\mathcal{L}_2$ (i.e., $\mathcal{L}_1 \subseteq \mathcal{L}_2$), then the volume of the sublattice is at least as large as the volume of the superlattice:*

$$\mathrm{vol}(\mathcal{L}_1) \geq \mathrm{vol}(\mathcal{L}_2).$$

*More precisely, $\mathrm{vol}(\mathcal{L}_1) = k \cdot \mathrm{vol}(\mathcal{L}_2)$, where $k = |\det(\mathbf{M})|$ is a positive integer known as the **index** of $\mathcal{L}_1$ in $\mathcal{L}_2$.*

Proof. Let $\mathbf{B}_1$ and $\mathbf{B}_2$ be basis matrices for $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. Since $\mathcal{L}_1 \subseteq \mathcal{L}_2$, every basis vector of $\mathcal{L}_1$ belongs to $\mathcal{L}_2$, meaning it can be expressed as an integer linear combination of the columns of $\mathbf{B}_2$. Algebraically, this implies the existence of an integer matrix $\mathbf{M} \in \mathbb{Z}^{n\times n}$ such that:

$$\mathbf{B}_1 = \mathbf{B}_2\mathbf{M}.$$

Taking determinants on both sides, we get:

$$\det(\mathbf{B}_1) = \det(\mathbf{B}_2)\det(\mathbf{M}) \implies \mathrm{vol}(\mathcal{L}_1) = \mathrm{vol}(\mathcal{L}_2) \cdot |\det(\mathbf{M})|.$$

Since both lattices are full-rank, $\det(\mathbf{M})$ must be a non-zero integer, so $|\det(\mathbf{M})| \geq 1$. Consequently, $\mathrm{vol}(\mathcal{L}_1) \geq \mathrm{vol}(\mathcal{L}_2)$. □

Definition 2.14 (Successive Minima [4])**.** Let $\mathcal{L} \subseteq \mathbb{R}^n$ be a lattice of rank n. For each $i \in \{1, \ldots, n\}$, the i-th **successive minimum** $\lambda_i(\mathcal{L})$ is defined as the smallest real number r such that $\mathcal{L}$ contains at least i linearly independent vectors of length at most r:

$$\lambda_i(\mathcal{L}) = \inf\{r > 0 : \dim(\mathrm{span}(\mathcal{L} \cap \bar{B}(\mathbf{0}, r))) \geq i\}.$$

Proposition 2.3 (Properties of Successive Minima)**.** *For any lattice $\mathcal{L}$ of rank n, the successive minima satisfy the following chain of inequalities:*

$$0 < \lambda_1(\mathcal{L}) \leq \lambda_2(\mathcal{L}) \leq \cdots \leq \lambda_n(\mathcal{L}).$$

Here, $\lambda_1(\mathcal{L})$ represents the length of the shortest non-zero vector in the lattice, which is the solution to the Shortest Vector Problem (SVP).

Remark 2.7 (Estimating λ_1). The magnitude of $\lambda_1(\mathcal{L})$ is bounded theoretically by Minkowski's Theorem and estimated practically by the Gaussian Heuristic:

- **Upper Bound (Worst-Case):** $\lambda_1(\mathcal{L}) \leq \sqrt{n}\det(\mathcal{L})^{1/n}$ (Minkowski).
- **Approximation (Average-Case):** For random lattices used in cryptography, $\lambda_1(\mathcal{L}) \approx \sqrt{\frac{n}{2\pi e}}\det(\mathcal{L})^{1/n}$ (Gaussian Heuristic).

Cryptographic parameters are typically chosen such that this predicted λ_1 is large enough to resist lattice reduction attacks.

Proposition 2.4 (Volume of Full-Rank Lattices)**.** *If $\mathcal{L}$ is a full-rank lattice (i.e., $\mathbf{B}$ is a square $n \times n$ matrix), the volume formula simplifies to the absolute value of the determinant of the basis matrix:*

$$\mathrm{vol}(\mathcal{L}) = |\det(\mathbf{B})|.$$

Remark 2.8. Geometrically, the volume represents the inverse of the lattice density. A small volume implies that lattice points are densely packed, whereas a large volume indicates a sparse lattice.

2.3.2 The Shortest Vector Problem (SVP)

The security of lattice-based cryptosystems relies on the computational difficulty of finding short vectors in high-dimensional lattices. The most fundamental of these problems is the Shortest Vector Problem.

Definition 2.15 (Shortest Vector Problem (SVP) [4])**.** Given a basis $\mathbf{B}$ of a full-rank lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, the **Shortest Vector Problem (SVP)** asks to find a non-zero lattice vector $\mathbf{v} \in \mathcal{L}$ such that its length is minimal, i.e., $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$.

Solving the exact SVP is known to be NP-hard under randomized reductions. However, in cryptography, we often consider a relaxed version of the problem.

Definition 2.16 (Approximate SVP (SVP_γ))**.** Let $\gamma \geq 1$ be an approximation factor. The **Approximate Shortest Vector Problem (SVP_γ)** asks to find a non-zero lattice vector $\mathbf{v} \in \mathcal{L}$ such that:

$$\|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\mathcal{L}).$$

Remark 2.9 (Hardness and Approximation Factor). The hardness of SVP_γ depends crucially on the factor γ:

- For small γ (e.g., constant or polynomial in n), the problem remains computationally hard, serving as the basis for cryptographic security.
- For large γ (e.g., exponential in n, specifically $\gamma \approx 2^n$), the problem becomes easier and can be solved efficiently using lattice reduction algorithms like LLL.

Cryptanalysis involves finding the smallest γ for which the problem can be solved in practical time.

2.3.3 Definition of the SIS Problem

Definition 2.17 (Short Integer Solution (SIS) [1])**.** Let $n, m, q \in \mathbb{N}$ with $q \geq 2$, and let $\beta > 0$ be a real bound. Given a uniformly random matrix $A \in \mathbb{Z}_q^{n \times m}$, the average-case SIS problem $\mathrm{SIS}_{n,m,q,\beta}$ asks to find a nonzero integer vector $\mathbf{z} \in \mathbb{Z}^m$ of Euclidean norm $\|\mathbf{z}\|_2 \leq \beta$ such that

$$A\mathbf{z} \equiv \mathbf{0} \pmod{q}.$$

(Usually one assumes $\beta < q$ to avoid trivial solutions such as $\mathbf{z} = (q, 0, \ldots, 0)$.) In other words, SIS asks for a "short" vector in the kernel of A modulo q. The SIS problem was originally introduced by Ajtai (1996), who proved that solving SIS on average is at least as hard as certain worst-case lattice problems (e.g., approximating the shortest vector).

Definition 2.18 (q-ary Kernel Lattice)**.** Let $q \geq 2$ and let $A \in \mathbb{Z}_q^{n \times m}$. The **$q$-ary kernel lattice** associated with A is defined as

$$\Lambda_q^{\perp}(A) = \{\mathbf{z} \in \mathbb{Z}^m : A\mathbf{z} \equiv \mathbf{0} \pmod{q}\}.$$

Thus, $\Lambda_q^{\perp}(A)$ is the set of all integer vectors that vanish under multiplication by A modulo q.

Definition 2.19 (q-ary Row Lattice Associated with A)**.** Let $q \geq 2$ and let $A \in \mathbb{Z}_q^{n \times m}$. The **$q$-ary row lattice** associated with A is defined as

$$\Lambda_q(A) = \{\mathbf{y} \in \mathbb{Z}^m : \mathbf{y} \equiv A^T\mathbf{s} \pmod{q} \text{ for some } \mathbf{s} \in \mathbb{Z}_q^n\}.$$

Equivalently, $\Lambda_q(A)$ consists of all integer vectors whose reduction modulo q lies in the row span of A over $\mathbb{Z}_q$.

Remark 2.10 (Duality Relation). The two q-ary lattices above are related by duality. In particular, for the standard q-ary lattices associated with A, one has

$$\Lambda_q^{\perp}(A)^* = \frac{1}{q}\Lambda_q(A).$$

This relation explains why SIS, which is formulated using $\Lambda_q^{\perp}(A)$, is closely connected to dual lattice geometry.

Definition 2.20 (Ring-SIS for Power-of-Two Cyclotomics [5])**.** Let n be a power of two, $q \geq 2$ an integer modulus, and let $R = \mathbb{Z}[x]/\langle x^n + 1\rangle$. We denote by $R_q = R/qR$ the quotient ring.

Let $l \geq 1$. We define a matrix $\mathbf{A} \in R_q^{1 \times l}$ as a row vector of l polynomials $\mathbf{A} = [a_1, \ldots, a_l]$, where each a_i is sampled uniformly at random from R_q.

The *Ring-SIS*$_{n,l,q,\beta}$ problem asks to find a non-zero vector of polynomials $\mathbf{z} \in R^l \setminus \{\mathbf{0}\}$ (interpreted as a column vector) such that:

$$\mathbf{A}\mathbf{z} \equiv 0 \pmod{q}$$

and

$$\|\mathbf{z}\| \leq \beta,$$

where $\|\mathbf{z}\|$ denotes the Euclidean norm of the coefficient vector of $\mathbf{z}$ in $\mathbb{Z}^{ln}$.

Definition 2.21 (Module-SIS Problem [6])**.** Let n be a power of two and $R = \mathbb{Z}[x]/\langle x^n + 1\rangle$ be the ring of integers. We denote by $R_q = R/qR = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ the quotient ring for an integer modulus $q \geq 2$.

Let $k, m > 0$ be integers (typically with $m > k$). Given a matrix $\mathbf{A} \in R_q^{k\times m}$ consisting of uniformly random polynomials, the *Module Short Integer Solution* problem (M-SIS$_{n,k,m,q,\beta}$) asks to find a non-zero vector of polynomials $\mathbf{z} \in R^m \setminus \{\mathbf{0}\}$ such that:

$$\mathbf{Az} \equiv \mathbf{0} \pmod{q}$$

and

$$0 < \|\mathbf{z}\| \leq \beta,$$

where $\|\mathbf{z}\|$ denotes the Euclidean norm induced by the coefficient embedding of R^m into $\mathbb{Z}^{mn}$.

2.4 Cryptographic Hash Functions

2.4.1 Definition of a Cryptographic Hash Function

Definition 2.22 (Cryptographic Hash Function [7])**.** A *cryptographic hash function* is a deterministic and efficiently computable function

$$H : \{0,1\}^* \to \{0,1\}^n,$$

that maps inputs of arbitrary length (bit strings) to fixed-length outputs of n bits. Such functions are used to generate compact representations, or *digests*, of data. A cryptographic hash function must satisfy the following essential properties:

- **Fixed-Length Output (Compression):** For any input $x \in \{0,1\}^*$, the output $H(x)$ always has length n bits, where n is a fixed parameter of the function. This means that even large inputs are "compressed" into a short, fixed-size output.
- **Efficient Computability:** The function $H(x)$ can be computed in time polynomial in the length of x. That is, for any input, the hash value can be calculated quickly and reliably.
- **Preimage Resistance:** Given a hash output $y \in \{0,1\}^n$, it is computationally infeasible to find any input $x \in \{0,1\}^*$ such that $H(x) = y$.
- **Second Preimage Resistance:** Given an input $x \in \{0,1\}^*$, it is computationally infeasible to find a different input $x' \neq x$ such that $H(x') = H(x)$.
- **Collision Resistance:** It is computationally infeasible to find any two distinct inputs $x, x' \in \{0,1\}^*$, with $x \neq x'$, such that $H(x) = H(x')$.

These properties ensure that a cryptographic hash function acts as a secure and compact fingerprint of its input: it is easy to compute but computationally hard to reverse or to find two inputs that produce the same output.

2.4.2 The Compression Function

Definition 2.23 (Compression Function [7])**.** A *compression function* is a deterministic and efficiently computable mapping

$$h : \{0,1\}^m \to \{0,1\}^n,$$

where m and n are fixed positive integers representing the bit-lengths of the input and output respectively, such that

$$m > n.$$

The requirement $m > n$ ensures that the function maps a larger, fixed-length domain to a smaller, fixed-length range, effectively "compressing" the input data. Unlike a general cryptographic hash function, a compression function is defined strictly for inputs of a **fixed length** m.

Remark 2.11 (Relation to General Hash Functions). The compression function serves as the fundamental building block for constructing cryptographic hash functions that accept inputs of arbitrary length. By employing a domain extension transform, such as the *Merkle-Damgård construction*, a collision-resistant compression function can be used to yield a collision-resistant hash function $H : \{0,1\}^* \to \{0,1\}^n$.

From this point onward, when we refer to the compression function studied in this thesis, we mean the fixed-input linear map

$$h_A(z) = Az \bmod q,$$

where

$$A \in \mathbb{Z}_q^{n \times m} \qquad \text{and} \qquad z \in \{0,1\}^m.$$

3 Security Analysis of Cyclic and Negacyclic Ring-Based Compression

3.1 Foundations of Lattice-Based Hashing

3.1.1 The SIS Problem and Average-Case Hardness

The reason the SIS problem is considered fundamental in the security theory of lattice-based cryptography lies in the rigorous mathematical connection between its average-case hardness and the worst-case hardness of certain worst-case lattice problems, established through worst-case to average-case reductions.

Specifically, let $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ be a uniformly random matrix, $q \in \mathbb{Z}_{\geq 2}$. Then the condition $\mathbf{Az} \equiv \mathbf{0} \pmod q$ for $\mathbf{z} \in \mathbb{Z}^m$ is equivalent to the existence of a vector $\mathbf{y} \in \mathbb{Z}^n$ such that $\mathbf{Az} = q\mathbf{y}$. Therefore, the set of all integer solutions

$$\Lambda_q^{\perp}(\mathbf{A}) = \{\mathbf{z} \in \mathbb{Z}^m : \mathbf{Az} \in q\mathbb{Z}^n\}$$

forms an additive subgroup of $\mathbb{Z}^m$, and hence a lattice in $\mathbb{R}^m$, called the q-ary lattice (or kernel lattice) associated with $\mathbf{A}$.

Since $\Lambda_q^{\perp}(\mathbf{A}) \subseteq \mathbb{Z}^m$, it is discrete in $\mathbb{R}^m$; moreover, it contains $q\mathbb{Z}^m$, and therefore has full rank. Furthermore, this structure is intimately connected to the geometry of dual lattices. In lattice theory, the dual of $\Lambda_q^{\perp}(\mathbf{A})$ satisfies the relation $\Lambda_q^{\perp}(\mathbf{A})^* = \frac{1}{q}\Lambda_q(\mathbf{A})$, where

$$\Lambda_q(A) = \{\mathbf{y} \in \mathbb{Z}^m : \mathbf{y} \equiv A^T\mathbf{s} \pmod q \text{ for some } \mathbf{s} \in \mathbb{Z}_q^n\}$$

and

$$\Lambda^* = \{x \in \mathrm{span}(\Lambda) : \langle x, y\rangle \in \mathbb{Z}, \forall y \in \Lambda\}.$$

This is the standard duality relation for q-ary lattices; its module-lattice analogue is used by Langlois and Stehlé in their converse reduction for M-SIS [6, Section 5]. This duality provides a link between SIS and the Learning With Errors (LWE) problem; while SIS is defined over the perpendicular lattice $\Lambda_q^{\perp}(\mathbf{A})$, LWE is closely related to the dual geometry of the corresponding q-ary lattices. Thus, SIS and LWE are naturally connected through the geometry of associated q-ary lattices.

In this sense, SIS transcends the form of a simple congruence and becomes a lattice problem: Given $\mathbf{A}$, find a non-zero vector $\mathbf{z} \in \Lambda_q^{\perp}(\mathbf{A})$ of Euclidean norm at most β, i.e., $\|\mathbf{z}\|_2 \leq \beta$. We use the ℓ_2-norm formulation of SIS; equivalent formulations with bounded ℓ_∞-norm are also common in the literature.

This lattice interpretation also yields a direct cryptographic application. Consider the keyed linear compression function

$$h_{\mathbf{A}}(\mathbf{z}) = \mathbf{A}\mathbf{z} \bmod q,$$

where the public key is the uniformly random matrix $\mathbf{A}$ (or a seed generating it), and the input is restricted to binary vectors $\mathbf{z} \in \{0,1\}^m$.

If an adversary finds a collision, namely two distinct inputs $\mathbf{z}_1 \neq \mathbf{z}_2$ such that

$$h_{\mathbf{A}}(\mathbf{z}_1) = h_{\mathbf{A}}(\mathbf{z}_2),$$

then

$$\mathbf{A}(\mathbf{z}_1 - \mathbf{z}_2) \equiv \mathbf{0} \pmod{q}.$$

Setting $\mathbf{z} = \mathbf{z}_1 - \mathbf{z}_2$, we obtain a non-zero vector

$$\mathbf{z} \in \{-1, 0, 1\}^m \subset \mathbb{Z}^m$$

satisfying

$$\mathbf{A}\mathbf{z} \equiv \mathbf{0} \pmod{q}.$$

Moreover,

$$\|\mathbf{z}\|_2 \leq \sqrt{m},$$

since each coordinate of $\mathbf{z}$ belongs to $\{-1, 0, 1\}$. Therefore, every collision yields a valid SIS solution for any parameter $\beta \geq \sqrt{m}$.

The converse implication does not hold for an arbitrary SIS solution. A generic SIS solver may output a short vector in $\mathbb{Z}^m$ whose coordinates are not restricted to $\{-1, 0, 1\}$. Such a vector need not be representable as the difference of two binary vectors. However, if the SIS solution is ternary, that is $z \in \{-1, 0, 1\}^m$, then it can be decomposed as $z = z^+ - z^-$, with $z^+, z^- \in \{0, 1\}^m$, and therefore it yields a valid collision.

3.1.2 Complexity Bottlenecks of Unstructured Lattices

Although the SIS framework is theoretically well motivated, it is computationally inefficient in practice. For the map $h_{\mathbf{A}}(\mathbf{z}) = \mathbf{A}\mathbf{z} \bmod q$ to serve as a compression function, the input length must exceed the output length when both are measured in bits. Since the input vector $\mathbf{z} \in \{0,1\}^m$ contains exactly m bits, while the output lies in $\mathbb{Z}_q^n$ and therefore requires $n\lceil \log_2 q \rceil$ bits under a fixed-length encoding, compression requires

$$m > n\lceil \log_2 q \rceil.$$

This immediately leads to a substantial public-key size. Indeed, the matrix $A \in \mathbb{Z}_q^{n \times m}$ contains nm entries, so an explicit representation requires $nm\lceil \log_2 q \rceil$ bits. We assume that the modulus q is polynomially bounded in the dimension n, meaning that there exists a constant $c > 0$ such that

$$q \leq n^c.$$

Hence,

$$\log_2 q \leq c \log_2 n,$$

and therefore

$$\lceil \log_2 q \rceil = O(\log n).$$

In the minimal compression regime, where $m \gtrsim n\lceil \log_2 q \rceil$, the public key therefore has size on the order of

$$n^2 \lceil \log_2 q \rceil^2$$

bits.

Consequently, in the minimal compression regime $m \approx n \log_2 q$, the explicit public-key size in bits satisfies

$$nm\lceil \log_2 q \rceil = O\Big(n^2 (\log q)^2\Big) = \widetilde{O}(n^2).$$

Thus, the key size is quasi-quadratic in the dimension n.

The evaluation cost is also quasi-quadratic under the same parameter regime. Computing $h_{\mathbf{A}}(\mathbf{z})$ amounts to summing the columns of $\mathbf{A}$ indexed by the nonzero entries of $\mathbf{z}$ and reducing modulo q. In the worst case, this requires

$$\Theta(nm)$$

modular additions and reductions, which is again quasi-quadratic under the same parameter regime.

This practical bottleneck motivates the transition to structured variants, such as Ring-SIS and Module-SIS, in which $\mathbf{A}$ admits a compact representation together with fast structured multiplication, typically via polynomial arithmetic and the Number Theoretic Transform (NTT). As a result, both storage and evaluation complexity can be reduced to quasi-linear in the ring dimension n, up to polylogarithmic factors and the number of ring/module components.

The structure of this chapter, as well as the formulation of several theorems and propositions, follows the exposition of [8]. Additional analysis, detailed proofs, and supplementary explanations are included to extend and clarify the original presentation.

3.2 Ring-SIS: Structural Efficiency and Isomorphisms

3.2.1 Circulant Matrices and Algebraic Structure

The transition from standard Short Integer Solution (SIS) to Ring-SIS is motivated primarily by efficiency considerations. In the unstructured setting, the public matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ must be stored explicitly, which requires $nm\lceil \log_2 q \rceil$ bits. Likewise, evaluating

$$h_{\mathbf{A}}(\mathbf{z}) = \mathbf{A}\mathbf{z} \bmod q$$

requires $O(nm)$ modular operations. Under the standard cryptographic scaling, both storage and evaluation therefore become quasi-quadratic in the dimension.

Ring-SIS addresses this bottleneck by replacing the fully random matrix $\mathbf{A}$ with a structured matrix admitting a compact description and fast multiplication. This improvement

is typically achieved by imposing a ring structure and interpreting matrix–vector multiplication in terms of polynomial multiplication, which can be accelerated by convolution techniques such as the Number Theoretic Transform.

This gain in efficiency changes the mathematical structure of the problem. In standard SIS, the public matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ is sampled uniformly at random, and the corresponding hardness assumption is related to general lattices. In Ring-SIS, the matrix is no longer arbitrary: it is generated from a small number of ring elements and has a block-circulant structure. Therefore, the associated lattice problem is restricted to lattices with additional algebraic structure, commonly described as ideal lattices. Thus, Ring-SIS should be treated as a structured variant of SIS, rather than simply as a faster implementation of the same unstructured problem.

In the construction considered here, the public matrix is generated from a compact set of ℓ independent random vectors $\mathbf{a}_i \leftarrow \mathbb{Z}_q^n$. These vectors serve as generators of a block-circulant matrix, thereby providing the algebraic basis of the Ring-SIS framework. The next step is to make this structured representation explicit and to show how it corresponds naturally to multiplication in a suitable quotient ring.

The fundamental building block of the cyclic structure is the cyclic shift matrix $\mathbf{X} \in \{0,1\}^{n \times n}$, which satisfies the identity $\mathbf{X}^n = \mathbf{I}_n$

$$\mathbf{X} := \begin{pmatrix} 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \end{pmatrix}$$

For every vector $\mathbf{a} = (a_0, \dots, a_{n-1})^\top \in \mathbb{Z}_q^n$, we define the circulant matrix $\mathrm{Rot}(\mathbf{a}) \in \mathbb{Z}_q^{n \times n}$, whose columns are generated by successive applications of $\mathbf{X}$

$$\mathrm{Rot}(\mathbf{a}) := \left(\mathbf{a}, \mathbf{X}\mathbf{a}, \dots, \mathbf{X}^{n-1}\mathbf{a}\right) = \begin{pmatrix} a_0 & a_{n-1} & \dots & a_1 \\ a_1 & a_0 & \dots & a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & \dots & a_0 \end{pmatrix}$$

By setting $m = n\ell$, the public key matrix $\mathbf{A}$ is obtained by horizontally concatenating ℓ such blocks, thereby reducing the storage requirement to $m\lceil \log_2 q \rceil$ bits.

The block-circulant structure admits a natural algebraic interpretation. Indeed, the cyclic shift matrix $\mathbf{X}$ satisfies $\mathbf{X}^n = \mathbf{I}_n$, and every circulant matrix generated by a vector $\mathbf{a} = (a_0, \dots, a_{n-1})^\top$ can be written as a polynomial in $\mathbf{X}$

$$\mathrm{Rot}(\mathbf{a}) = \sum_{k=0}^{n-1} a_k \mathbf{X}^k.$$

This suggests identifying circulant matrices with polynomials modulo the relation $x^n - 1 = 0$ [8].

3.2.2 Polynomial Quotient Rings and the Isomorphism ψ

Proposition 3.1. *Let*

$$\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1 \rangle$$

and let $\widetilde{\mathcal{R}}_q \subseteq \mathbb{Z}_q^{n\times n}$ *denote the ring of* $n \times n$ *circulant matrices over* $\mathbb{Z}_q$*. If* $\mathbf{X} \in \{0,1\}^{n\times n}$ *is the cyclic shift matrix satisfying* $\mathbf{X}^n = \mathbf{I}_n$*, then the map*

$$\psi : \mathcal{R}_q \to \widetilde{\mathcal{R}}_q, \qquad \psi\left(\sum_{k=0}^{n-1} a_k x^k\right) = \sum_{k=0}^{n-1} a_k \mathbf{X}^k,$$

is a well-defined ring isomorphism.

Proof. We first show that ψ is a ring homomorphism. Let

$$a(x) = \sum_{i=0}^{n-1} a_i x^i, \qquad b(x) = \sum_{i=0}^{n-1} b_i x^i$$

be elements of $\mathcal{R}_q$. By linearity,

$$\psi(a+b) = \sum_{i=0}^{n-1} (a_i + b_i)\mathbf{X}^i = \sum_{i=0}^{n-1} a_i \mathbf{X}^i + \sum_{i=0}^{n-1} b_i \mathbf{X}^i = \psi(a) + \psi(b).$$

Moreover, if

$$a(x)b(x) = \sum_{k=0}^{n-1} c_k x^k \qquad \text{in } \mathcal{R}_q,$$

then the coefficients c_k are given by the cyclic convolution

$$c_k = \sum_{i+j \equiv k \pmod n} a_i b_j.$$

Hence

$$\psi(a)\psi(b) = \left(\sum_{i=0}^{n-1} a_i \mathbf{X}^i\right)\left(\sum_{j=0}^{n-1} b_j \mathbf{X}^j\right) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j \mathbf{X}^{i+j}.$$

Since $\mathbf{X}^n = \mathbf{I}_n$, each power $\mathbf{X}^{i+j}$ reduces to

$$\mathbf{X}^{(i+j) \bmod n},$$

and therefore

$$\psi(a)\psi(b) = \sum_{k=0}^{n-1}\left(\sum_{i+j\equiv k \pmod n} a_i b_j\right)\mathbf{X}^k = \sum_{k=0}^{n-1} c_k \mathbf{X}^k = \psi(ab).$$

Also,

$$\psi(1) = \mathbf{I}_n,$$

so ψ preserves the multiplicative identity.

We next verify that ψ is well defined. Suppose that $p_1(x), p_2(x) \in \mathbb{Z}_q[x]$ represent the same residue class in $\mathcal{R}_q$. Then

$$p_1(x) - p_2(x) = s(x)(x^n - 1)$$

for some $s(x) \in \mathbb{Z}_q[x]$. Substituting $\mathbf{X}$ for x gives

$$\psi(p_1) - \psi(p_2) = s(\mathbf{X})(\mathbf{X}^n - \mathbf{I}_n) = \mathbf{0},$$

since $\mathbf{X}^n = \mathbf{I}_n$. Thus $\psi(p_1) = \psi(p_2)$, and so ψ depends only on the residue class modulo $x^n - 1$.

Surjectivity is immediate. Every matrix in $\widetilde{\mathcal{R}}_q$ is circulant and therefore has the form

$$\sum_{k=0}^{n-1} a_k \mathbf{X}^k$$

for uniquely determined coefficients $a_k \in \mathbb{Z}_q$. It follows that every element of $\widetilde{\mathcal{R}}_q$ lies in the image of ψ.

To prove injectivity, let $p(x) \in \mathcal{R}_q$ satisfy $\psi(p) = \mathbf{0}$. Choosing the unique representative of degree $< n$, write

$$p(x) = \sum_{k=0}^{n-1} c_k x^k.$$

Then

$$\psi(p) = \sum_{k=0}^{n-1} c_k \mathbf{X}^k = \mathbf{0}.$$

Applying this matrix to the first standard basis vector $\mathbf{e}_1 = (1, 0, \ldots, 0)^\top$, we obtain

$$\psi(p)\mathbf{e}_1 = \sum_{k=0}^{n-1} c_k \mathbf{X}^k \mathbf{e}_1 = \sum_{k=0}^{n-1} c_k \mathbf{e}_{k+1} = (c_0, c_1, \ldots, c_{n-1})^\top.$$

Since $\psi(p)\mathbf{e}_1 = \mathbf{0}$, all coefficients c_k must vanish. Hence $p(x) = 0$ in $\mathcal{R}_q$, so $\ker(\psi) = \{0\}$ and ψ is injective.

Therefore, ψ is a well-defined ring isomorphism. □

3.2.3 Computational Optimization via NTT

Through the ring isomorphism described above, the evaluation of the Ring-SIS map reduces to the computation of polynomial convolutions. In the cyclic case, multiplication in

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle$$

corresponds to cyclic convolution. Such convolutions can be performed efficiently by means of the Number Theoretic Transform (NTT), which may be viewed as the analogue of the Discrete Fourier Transform over the finite ring $\mathbb{Z}_q$.

For cyclic convolution modulo $x^n - 1$, an NTT of length n exists whenever $\mathbb{Z}_q$ contains a primitive n-th root of unity, equivalently, an element of multiplicative order n in $\mathbb{Z}_q^\times$. In the important special case where q is prime, this is ensured by the divisibility condition

$$n \mid (q - 1).$$

For negacyclic convolution modulo $x^n + 1$, the required condition is stronger. In this case, one needs a primitive $2n$-th root of unity $\psi \in \mathbb{Z}_q$ satisfying

$$\psi^n \equiv -1 \pmod{q}.$$

When q is prime, this is ensured by

$$2n \mid (q - 1).$$

Under the appropriate root-of-unity condition, polynomial multiplication modulo $x^n - 1$ or modulo $x^n + 1$ can be carried out by transforming to the frequency domain, performing point-wise multiplication, and then applying the inverse transform. Consequently, the cost of evaluating one block drops to

$$O(n \log n).$$

Since the input consists of ℓ blocks with $m = n\ell$, the total evaluation cost becomes

$$O(\ell n \log n) = O(m \log n)$$

modular operations.

A further advantage of this approach is that all arithmetic is performed modulo q. As a result, the bit-length of the intermediate coefficients remains bounded by $O(\log q)$, in contrast to naive polynomial multiplication, where coefficient growth can become polynomial in n prior to reduction. Thus, both the arithmetic complexity and the bit complexity remain well controlled.

3.3 Cryptanalysis of the Cyclic Modulus $x^n - 1$

3.3.1 Reducibility and the Presence of Zero Divisors

The choice of the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ for any integer modulus $q \geq 2$, while computationally attractive due to fast convolution, introduces a critical structural weakness that renders the hash function insecure against collisions. The primary cause is that the polynomial modulus $x^n - 1$ is always reducible and factors over the integers as $(x-1)(1 + x + \cdots + x^{n-1})$.

This factorization leads to the existence of non-trivial zero divisors within the ring , allowing for an efficient attack where the polynomial $\widetilde{e}(x) = 1+x+\cdots+x^{n-1}$ is annihilated by $x - 1$, and therefore it is a non-trivial zero-divisor.

As a result, the Ring-SIS problem over this ring suffers a rank-one collapse, the collision search inside the submodule generated by $\widetilde{e}(x)$ is reduced to a one-dimensional modular relation over $\mathbb{Z}_q$. This effectively degenerates into a simple one-dimensional parity relation over $\mathbb{Z}_q$. This structural collapse makes the cyclic choice unsuitable for collision-resistant hashing and motivates the transition to moduli without the linear factor $x - 1$, such as the negacyclic modulus $x^n + 1$ when n is a power of two.

Proposition 3.2. *In the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ with $n \geq 2$, the all-ones polynomial $\bar{e}(x) := \sum_{i=0}^{n-1} x^i$ constitutes a non-trivial zero-divisor. Specifically, the following relation holds*

$$(x-1)\tilde{e}(x) = 0 \qquad \text{in } \mathcal{R}_q$$

Proof. Consider the standard algebraic identity for the sum of a geometric progression, $(x-1)\sum_{i=0}^{n-1} x^i = x^n - 1$, which holds in the base polynomial ring $\mathbb{Z}_q[x]$. Upon passing to the quotient ring by the ideal $\langle x^n - 1\rangle$, the right-hand side of the equation vanishes, yielding the relation $(x-1)\bar{e}(x) \equiv 0 \pmod{x^n - 1}$.

For $n \geq 2$, the polynomials $(x-1)$ and $\bar{e}(x)$ have degrees strictly less than n and are non-zero in $\mathbb{Z}_q[x]$; consequently, their corresponding residue classes are non-zero in $\mathcal{R}_q$. This confirms that $\bar{e}(x)$ is a non-trivial zero-divisor, as it is a non-zero element whose product with another non-zero element $(x-1)$ results in the zero element of the ring. □

To make this phenomenon completely explicit and mathematically computable, we define the all-ones polynomial, often referred to as the geometric series element:

$$\tilde{e}(x) := \sum_{i=0}^{n-1} x^i \in \mathcal{R}_q$$

3.3.2 The Scalar Reduction Lemma and Dimension Collapse

Lemma 3.1 (Scalar Reduction)**.** *For any polynomial $a(x) \in \mathcal{R}_q$, where $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$, the product with the all-ones polynomial $\tilde{e}(x) = \sum_{i=0}^{n-1} x^i$ satisfies the following identity*

$$a(x)\tilde{e}(x) = a(1)\tilde{e}(x) \pmod{\langle x^n - 1, q\rangle} \tag{1}$$

where $a(1) = \sum_{i=0}^{n-1} a_i \pmod{q}$ denotes the sum of the coefficients of $a(x)$.

Proof. Let $a(x) \in \mathbb{Z}_q[x]$ be an arbitrary representative of a residue class in $\mathcal{R}_q$. By the polynomial remainder theorem, any polynomial $a(x)$ can be expressed as $a(x) = (x-1)b(x) + r$, where $b(x)$ is the quotient polynomial and r is a constant residue. Evaluating this expression at $x = 1$, we observe that $a(1) = (1-1)b(1) + r$, which implies that the residue r is exactly the sum of the coefficients $a(1)$ modulo q.

By multiplying both sides of the identity $a(x) - a(1) = (x-1)b(x)$ by the all-ones polynomial $\tilde{e}(x)$, we obtain

$$(a(x) - a(1))\tilde{e}(x) = b(x)(x-1)\tilde{e}(x)$$

Recalling the fundamental identity $(x-1)\tilde{e}(x) = x^n - 1$, and given that $x^n - 1 \equiv 0$ within the quotient ring $\mathcal{R}_q$, the right-hand side of the equation vanishes. Consequently, it holds that $(a(x) - a(1))\tilde{e}(x) = 0$ in $\mathcal{R}_q$, which directly yields $a(x)\tilde{e}(x) = a(1)\tilde{e}(x)$. □

3.3.3 The Invariant Submodule and the Rank-One Collapse

The algebraic properties of the all-ones polynomial established in the previous section imply that the $\mathbb{Z}_q$-submodule generated by $\tilde{e}(x)$ remains invariant under multiplication by arbitrary elements of the ring $\mathcal{R}_q$. We formally define this submodule E as follows:

$$E := \mathrm{Span}_{\mathbb{Z}_q}\{\tilde{e}(x)\} = \{t\tilde{e}(x) : t \in \mathbb{Z}_q\} \subseteq \mathcal{R}_q$$

For every polynomial $a(x) \in \mathcal{R}_q$ and for every scalar $t \in \mathbb{Z}_q$, the Scalar Reduction Lemma ensures that the following relation holds:

$$a(x) \cdot (t\tilde{e}(x)) = t(a(x)\tilde{e}(x)) = ta(1)\tilde{e}(x) \in E \tag{2}$$

This confirms that E constitutes a rank-one invariant submodule, a property that represents the core of the cryptographic collapse in the cyclic setting.

Geometrically, this invariance signifies that any high-dimensional operation within $\mathcal{R}_q$ involving the all-ones polynomial effectively degenerates into a one-dimensional scalar multiplication governed by $a(1) \in \mathbb{Z}_q$. Consequently, finding a collision in $\mathcal{R}_q$ for any $n > 1$ becomes no more difficult than solving a simple instance of the SIS problem for $n = 1$, rendering the structure useless for ensuring collision resistance.

Theorem 3.1 (Rank-One Structural Collapse)**.** *Let $q \geq 2$ and $n \geq 2$ be integers, and consider the quotient ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$. Let $\mathbf{a}(x) = (a_1(x), \ldots, a_\ell(x)) \in \mathcal{R}_q^\ell$ be a public key vector consisting of independent and uniformly random ring elements. For an input vector $\mathbf{z}(x) = (z_1(x), \ldots, z_\ell(x)) \in \mathcal{R}_q^\ell$ restricted to the invariant submodule E^ℓ, where each component takes the form $z_i(x) = \epsilon_i \tilde{e}(x)$ with $\epsilon_i \in \{-1, 0, 1\}$ and $\tilde{e}(x) = \sum_{j=0}^{n-1} x^j$, the Ring-SIS hash evaluation*

$$H_{\mathbf{a}}(\mathbf{z}) = \sum_{i=1}^{\ell} a_i(x) z_i(x) \in \mathcal{R}_q \tag{3}$$

is equal to zero if and only if the following one-dimensional linear congruence is satisfied

$$\sum_{i=1}^{\ell} \epsilon_i a_i(1) \equiv 0 \pmod{q} \tag{4}$$

where $a_i(1) = \sum_{j=0}^{n-1} a_{i,j} \in \mathbb{Z}_q$ represents the sum of the coefficients of $a_i(x)$. Consequently, any non-zero vector $\boldsymbol{\epsilon} \in \{-1, 0, 1\}^\ell$ satisfying this parity condition yields a non-zero ternary kernel vector $\mathbf{z} \in \{-1, 0, 1\}^m$. By decomposing $\mathbf{z} = \mathbf{z}^+ - \mathbf{z}^-$, with $\mathbf{z}^+, \mathbf{z}^- \in \{0, 1\}^m$, one obtains a non-trivial binary collision for $H_{\mathbf{a}}$.

Proof. By substituting the restricted form $z_i(x) = \epsilon_i \tilde{e}(x)$ into the hash mapping, the expression becomes

$$\sum_{i=1}^{\ell} a_i(x)(\epsilon_i \tilde{e}(x)) = \left(\sum_{i=1}^{\ell} \epsilon_i a_i(x) \right) \tilde{e}(x) \in \mathcal{R}_q. \tag{5}$$

Applying the Scalar Reduction Lemma to the accumulated polynomial $A(x) = \sum \epsilon_i a_i(x)$, we obtain the identity $A(x)\tilde{e}(x) = A(1)\tilde{e}(x)$ within $\mathcal{R}_q$. Thus, the sum reduces to

$$H_{\mathbf{a}}(\mathbf{z}) = \left(\sum_{i=1}^{\ell} \epsilon_i a_i(1) \right) \tilde{e}(x). \tag{6}$$

Since $\tilde{e}(x)$ is a non-zero element of $\mathcal{R}_q$ with coefficient vector $(1, 1, \ldots, 1)$, the product vanishes in the quotient ring if and only if the scalar multiplier $c = \sum \epsilon_i a_i(1)$ is zero modulo q. This confirms that finding a collision in the n-dimensional ring $\mathcal{R}_q$ collapses to solving a one-dimensional SIS instance over $\mathbb{Z}_q$. □

3.3.4 Norm Constraints and Collision Existence Guarantee

For the algebraic reduction to be practically applicable, the resulting vector $\mathbf{z}$ must satisfy the geometric constraint $\|\mathbf{z}\|_2 \leq \beta$ inherent to the Ring-SIS problem. The Euclidean norm of $\mathbf{z}$ is defined as the square root of the sum of the squares of all its coefficients across the ℓ polynomials.

Since each component $z_i(x)$ is constructed as $\epsilon_i \tilde{e}(x)$, where $\tilde{e}(x)$ is the all-ones polynomial of degree $n - 1$, the individual contribution of each component to the total norm is determined by the selection coefficient $\epsilon_i \in \{-1, 0, 1\}$.

To formalize this contribution, we utilize the Hamming weight, denoted as $\mathrm{wt}(\epsilon_i)$, which serves as an indicator function that equals 1 if $\epsilon_i \neq 0$ and 0 otherwise. Given that each non-zero component $z_i(x)$ contains exactly n coefficients whose squares equal 1, it

contributes exactly n to the total sum of squares. Consequently, the norm of the concatenated vector is expressed as the square root of the sum of these individual contributions, leading to the formula $\|\mathbf{z}\|_2 = \sqrt{\sum_{i=1}^{\ell} \mathrm{wt}(\epsilon_i) \cdot n}$.

In the maximum case where all components are non-zero, this sum simplifies to the upper bound $\sqrt{\ell \cdot n}$. Thus, the lifted vector satisfies the Ring-SIS norm bound whenever

$$\beta \geq \sqrt{\ell n}.$$

Corollary 3.2 (Existence of a Non-trivial Signed Relation)**.** *Let $a_1(1), \ldots, a_\ell(1) \in \mathbb{Z}_q$ be the scalar coefficients derived from the public key. If the condition $2^\ell > q$ is satisfied, there is a guaranteed existence of a non-zero vector $\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_\ell) \in \{-1, 0, 1\}^\ell \setminus \{\mathbf{0}\}$ such that:*

$$\sum_{i=1}^{\ell} \epsilon_i a_i(1) \equiv 0 \pmod{q} \tag{7}$$

Consequently, a non-trivial collision within the restricted submodule E^ℓ is mathematically certain.

Proof. To establish the existence of such a relation, we consider the set of all 2^ℓ possible binary sub-sums $S(\mathbf{u}) = \sum_{i=1}^{\ell} u_i a_i(1) \in \mathbb{Z}_q$, where $\mathbf{u} \in \{0, 1\}^\ell$. Since the range of these sums is restricted to the q possible remainders in $\mathbb{Z}_q$, the pigeonhole principle dictates that if $2^\ell > q$, there must exist at least two distinct binary vectors $\mathbf{u}, \mathbf{v} \in \{0, 1\}^\ell$ (where $\mathbf{u} \neq \mathbf{v}$) that result in the exact same sum, such that $S(\mathbf{u}) \equiv S(\mathbf{v}) \pmod{q}$.

By rearranging this congruence, we obtain $S(\mathbf{u}) - S(\mathbf{v}) \equiv 0 \pmod{q}$, which can be rewritten as:

$$\sum_{i=1}^{\ell} (u_i - v_i) a_i(1) \equiv 0 \pmod{q} \tag{8}$$

Defining the components of the relation vector as $\epsilon_i := u_i - v_i$, it follows from the binary nature of u_i and v_i that each $\epsilon_i \in \{-1, 0, 1\}$. Furthermore, the condition $\mathbf{u} \neq \mathbf{v}$ ensures that the resulting vector $\boldsymbol{\epsilon}$ is non-zero. This scalar solution is then lifted to a valid collision in the ring $\mathcal{R}_q$ via the Rank-One Collapse theorem, completing the proof. □

We now quantify the consequences of the rank-one collapse in

$$R_q = \mathbb{Z}_q[x]/\langle x^n - 1 \rangle.$$

First, we analyze the success probability of a single fixed candidate vector inside the invariant submodule generated by $\widetilde{e}(x)$. For any fixed non-zero choice of

$$\boldsymbol{\varepsilon} \in \{-1, 0, 1\}^\ell,$$

the corresponding lifted vector satisfies the Ring-SIS relation with probability exactly $1/q$ over the random choice of the public key. This describes only the success probability of one fixed candidate. It does not rule out collisions, because the attacker can search over many possible signed vectors $\boldsymbol{\varepsilon}$.

To fully expose the vulnerability of this ring structure, we move from the probabilistic analysis of a fixed candidate to a deterministic collision-search framework. By exploiting the pigeonhole principle, we prove that whenever

$$2^\ell > q,$$

there exists a non-trivial signed relation

$$\sum_{i=1}^{\ell} \varepsilon_i a_i(1) \equiv 0 \pmod{q}, \qquad \varepsilon_i \in \{-1, 0, 1\}.$$

This relation can then be lifted to a non-zero ternary Ring-SIS solution inside the restricted submodule generated by $\widetilde{e}(x)$, and hence to a binary collision.

Algorithmically, the reduced problem is a one-dimensional modular subset-sum instance. It can be solved by a meet-in-the-middle strategy in time and memory

$$O(2^{\ell/2}).$$

In the parameter regime where $2^{\ell} \approx q$, this becomes approximately

$$O(\sqrt{q}).$$

Thus, the cyclic modulus $x^n - 1$ provides no meaningful resistance against this targeted structural attack.

Theorem 3.3 (Collision Probability of a Fixed Candidate Vector)**.** *Let $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ be a polynomial quotient ring and let $\tilde{e}(x) = \sum_{j=0}^{n-1} x^j \in \mathcal{R}_q$ denote the all-ones polynomial.*

Let $\mathbf{z} = (z_1, \ldots, z_\ell) \in \mathcal{R}_q^{\ell}$ be a vector of polynomials defined by $z_i(x) = \varepsilon_i \tilde{e}(x)$, where $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_\ell) \in \{-1, 0, 1\}^{\ell} \setminus \{\mathbf{0}\}$ is a fixed, pre-selected non-zero coefficient vector.

If the key polynomials $a_1, \ldots, a_\ell$ are sampled independently and uniformly at random from $\mathcal{R}_q$, the probability that $\mathbf{z}$ satisfies the Ring-SIS relation

$$\sum_{i=1}^{\ell} a_i(x) z_i(x) \equiv 0 \pmod{\mathcal{R}_q}$$

is exactly $1/q$ [8].

Proof. By substituting the structure $z_i(x) = \varepsilon_i \tilde{e}(x)$ into the compression function, we apply the **Scalar Reduction Lemma** for the $x^n - 1$ modulus. The lemma states that $a(x)\tilde{e}(x) \equiv a(1)\tilde{e}(x) \pmod{x^n - 1}$, where $a(1)$ is the scalar evaluation of the polynomial at $x = 1$.

Thus

$$\sum_{i=1}^{\ell} a_i(x)(\varepsilon_i \tilde{e}(x)) = \left(\sum_{i=1}^{\ell} \varepsilon_i a_i(1) \right) \tilde{e}(x) \pmod{x^n - 1}$$

Since $\tilde{e}(x)$ is a non-zero element in $\mathcal{R}_q$, the product vanishes if and only if the scalar sum $S = \sum \varepsilon_i a_i(1)$ satisfies $S \equiv 0 \pmod{q}$.

The key polynomials $a_i(x)$ are sampled uniformly at random from $\mathcal{R}_q$. Consequently, their scalar evaluations $a_i(1) = \sum_{j=0}^{n-1} a_{i,j} \pmod{q}$ (the sum of their coefficients) are independent and identically distributed (i.i.d.) uniform random variables over the finite ring $\mathbb{Z}_q$.

The sum $S = \sum_{i=1}^{\ell} \varepsilon_i a_i(1)$ is a linear combination of these i.i.d. uniform variables. In any finite cyclic group $\mathbb{Z}_q$, the sum of a uniform random variable and any independent random variable is itself uniformly distributed. Since ε is fixed and non-zero, at least one coefficient $\varepsilon_j \in \{-1, 1\}$ exists. Given that ε_j is coprime to q, the term $\varepsilon_j a_j(1)$ is uniformly distributed. By induction, the total sum S follows a perfect uniform distribution over the set $\{0, 1, \ldots, q - 1\}$.

Since S is uniformly distributed over the q possible values of $\mathbb{Z}_q$, the probability that it equals any specific value, including 0, is exactly $1/q$. Therefore:

$$P\left(\sum_{i=1}^{\ell} \varepsilon_i a_i(1) \equiv 0 \pmod{q}\right) = \frac{1}{q}$$

This concludes the proof that any single fixed candidate vector $\mathbf{z}$ within the targeted submodule succeeds as a collision with probability $1/q$. □

3.4 Algorithmic Production of Collisions

At this point, recall from Definition 2.20 that, in the Ring-SIS setting considered here, the ring dimension n is assumed to be a power of two. All the pseudocode related to the Algorithms in the document can be found in Appendix 1.

3.4.1 Step-by-Step Collision Generation Methodology

Building upon the theoretical framework of the rank-one collapse, this section provides an explicit deterministic pipeline for generating collisions. We consider the public key matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, which, in the context of Ring-SIS over $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n - 1\rangle$, is structured as a block-circulant matrix. Specifically, $\mathbf{A}$ is formed by the horizontal concatenation of $\ell = m/n$ circulant blocks

$$\mathbf{A} = [\mathrm{Rot}(\mathbf{a}_1) \mid \mathrm{Rot}(\mathbf{a}_2) \mid \cdots \mid \mathrm{Rot}(\mathbf{a}_\ell)] \in \mathbb{Z}_q^{n \times m}$$

where each block $\mathrm{Rot}(\mathbf{a}_i)$ is the circulant matrix generated by the coefficient vector of the public polynomial $a_i(x) \in \mathcal{R}_q$.

The objective of the attack is first to construct a non-zero ternary vector

$$z \in \{-1, 0, 1\}^m$$

such that

$$Az \equiv 0 \pmod{q}.$$

Such a vector can then be decomposed as

$$z = z^+ - z^-, \qquad z^+, z^- \in \{0, 1\}^m,$$

producing a binary collision for h_A. The generation process exploits the invariant submodule

$$E = \langle \widetilde{e}(x) \rangle$$

to reduce the n-dimensional Ring-SIS relation to a rank-one modular search over $\mathbb{Z}_q$, as detailed in the following steps.

Step 1: Scalar mapping and dimensional reduction. The first stage of the attack involves projecting the structured n-dimensional public key matrix $\mathbf{A}$ onto the one-dimensional invariant submodule E. For each circulant block $\mathrm{Rot}(\mathbf{a}_i)$ within the public matrix, we compute its induced scalar $\alpha_i \in \mathbb{Z}_q$. This value corresponds to the evaluation of the public polynomial $a_i(x)$ at $x = 1$, or equivalently, the sum of its scalar coefficients

$$\alpha_i := a_i(1) = \sum_{j=0}^{n-1} a_{i,j} \pmod{q}, \quad \text{for } i = 1, \ldots, \ell.$$

By the Scalar Reduction Lemma, this operation effectively maps the algebraic action of each circulant block within the submodule E to a simple scalar multiplication. Consequently, the high-dimensional Ring-SIS instance over $\mathcal{R}_q^\ell$ is reduced to a one-dimensional modular instance over $\mathbb{Z}_q$, facilitating the subsequent combinatorial search.

Step 2: Combinatorial search and meet-in-the-middle strategy. With the structured instance reduced to a one-dimensional modular problem, the objective is to find a non-zero vector $\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_\ell) \in \{-1, 0, 1\}^\ell \setminus \{\mathbf{0}\}$ that satisfies the linear congruence

$$\sum_{i=1}^{\ell} \epsilon_i \alpha_i \equiv 0 \pmod{q}.$$

The existence of such a vector is mathematically guaranteed by the pigeonhole principle. Specifically, if we consider the 2^ℓ possible binary subset sums $s(S) = \sum_{i \in S} \alpha_i \pmod{q}$ for $S \subseteq \{1, \ldots, \ell\}$, the condition $2^\ell > q$ (or equivalently $\ell > \log_2 q$) ensures that at least two distinct subsets S, T must yield the same sum, $s(S) = s(T)$. Their difference yields a non-zero vector $\boldsymbol{\epsilon}$ with components in $\{-1, 0, 1\}$ that satisfies the target condition.

To compute such a vector efficiently, we use a birthday-style meet-in-the-middle strategy. The index set is partitioned into two halves of size approximately $\ell/2$. We enumerate subset sums for the first half and store them in a hash table. Then we enumerate subset sums for the second half and search for a matching value modulo q. Under the usual randomness heuristic for the residues α_i, this approach has expected time and memory complexity

$$O(2^{\ell/2}).$$

In the parameter regime where $2^\ell \approx q$, this becomes approximately

$$O(\sqrt{q}).$$

Thus, the security of the $x^n - 1$ ring does not scale with the dimension n, because the attack operates through the one-dimensional relation over $\mathbb{Z}_q$.

Step 3: Polynomial lifting and ternary kernel construction. Once a valid relation vector $\boldsymbol{\epsilon}$ has been identified in the scalar domain, the solution is lifted back to the ring $\mathcal{R}_q^\ell$ to construct the target collision vector $\mathbf{z}$. This is achieved by mapping each component ϵ_i to a polynomial $z_i(x) \in \mathcal{R}_q$ through the all-ones polynomial:

$$z_i(x) := \epsilon_i \tilde{e}(x), \quad \text{for } i = 1, \ldots, \ell. \tag{9}$$

By invoking the **Scalar Reduction Lemma**, the evaluation of the Ring-SIS compression function on the constructed vector $\mathbf{z}$ is transformed as follows:

$$\sum_{i=1}^{\ell} a_i(x) z_i(x) = \sum_{i=1}^{\ell} a_i(x)(\epsilon_i \tilde{e}(x)) = \left(\sum_{i=1}^{\ell} \epsilon_i a_i(1) \right) \tilde{e}(x) \pmod{\mathcal{R}_q}. \tag{10}$$

Since the relation $\sum_{i=1}^{\ell} \epsilon_i a_i(1) \equiv 0 \pmod{q}$ is satisfied by construction in Step 2, the scalar multiplier of the all-ones polynomial vanishes. Consequently, the entire sum yields the zero element in $\mathcal{R}_q$, confirming that the lifted vector $\mathbf{z}$ constitutes a valid, non-trivial ternary kernel solution to the Ring-SIS instance over the reducible modulus $x^n - 1$.

Step 4: Binary decomposition and collision extraction. The final stage of the pipeline extracts a valid binary collision pair from the ternary kernel solution $\mathbf{z}$. We first interpret the polynomial vector

$$\mathbf{z} \in R_q^\ell$$

as a concatenated coefficient vector

$$\mathbf{z} \in \{-1, 0, 1\}^m, \qquad m = n\ell.$$

To satisfy the binary input restriction of the compression function h_A, we define two binary vectors

$$\mathbf{z}^+, \mathbf{z}^- \in \{0, 1\}^m$$

coordinate-wise as follows:

$$\begin{cases} z_j = 1 & \Rightarrow (z_j^+, z_j^-) = (1, 0), \\ z_j = -1 & \Rightarrow (z_j^+, z_j^-) = (0, 1), \\ z_j = 0 & \Rightarrow (z_j^+, z_j^-) = (0, 0), \end{cases} \qquad j = 1, \ldots, m.$$

By construction, this gives

$$\mathbf{z} = \mathbf{z}^+ - \mathbf{z}^-.$$

Substituting this relation into the SIS congruence, we obtain

$$A\mathbf{z}^+ - A\mathbf{z}^- = A(\mathbf{z}^+ - \mathbf{z}^-) = A\mathbf{z} \equiv \mathbf{0} \pmod{q}.$$

Therefore,

$$A\mathbf{z}^+ \equiv A\mathbf{z}^- \pmod{q},$$

and hence

$$h_A(\mathbf{z}^+) = h_A(\mathbf{z}^-).$$

Thus, $(\mathbf{z}^+, \mathbf{z}^-)$ is a valid binary collision for the Ring-SIS-style compression function instantiated over the reducible modulus $x^n - 1$.

This explains why the cyclic modulus $x^n - 1$ is unsuitable for collision-resistant hashing and motivates the transition to the negacyclic modulus $x^n + 1$, which avoids the dimension collapse induced by evaluation at $x = 1$.

3.5 The Transition to $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and the Mitigation of Structural Vulnerabilities

As shown in the preceding analysis, the weakness of the cyclic ring

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle$$

comes from the factorization

$$x^n - 1 = (x - 1)(1 + x + \cdots + x^{n-1}).$$

This factorization produces the all-ones polynomial

$$\widetilde{e}(x) = 1 + x + \cdots + x^{n-1},$$

which satisfies

$$(x - 1)\widetilde{e}(x) = 0$$

in the quotient ring. Hence, $\widetilde{e}(x)$ is a non-trivial zero-divisor. This is exactly the algebraic property that enables the rank-one collapse described above.

To avoid this particular collapse, one replaces the cyclic modulus $x^n - 1$ by a modulus that does not contain the linear factor $x - 1$. A standard choice in Ring-SIS constructions is the negacyclic modulus

$$x^n + 1,$$

especially when n is a power of two. In this case, $x^n + 1$ is the $2n$-th cyclotomic polynomial and is irreducible over $\mathbb{Z}$. Thus, the simple evaluation attack at $x = 1$ no longer applies.

At the same time, the negacyclic modulus still supports efficient polynomial arithmetic. The relation

$$x^n \equiv -1 \pmod{x^n + 1}$$

gives rise to negacyclic convolution, which can be implemented efficiently using the Number Theoretic Transform when suitable roots of unity exist modulo q.

The adoption of the negacyclic ring

$$R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$$

changes the polynomial multiplication rule. Since

$$x^n \equiv -1 \pmod{x^n + 1},$$

terms that pass degree $n - 1$ wrap around with a sign change. Thus, the structured public-key blocks are no longer circulant, but negacyclic.

The basic linear operator is the negacyclic shift matrix

$$X_{\text{neg}} \in \{-1, 0, 1\}^{n\times n},$$

which satisfies

$$X_{\text{neg}}^n = -I_n.$$

It is defined by

$$X_{\text{neg}} := \begin{pmatrix} 0 & 0 & \cdots & 0 & -1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}.[8]$$

For every coefficient vector

$$\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})^T \in \mathbb{Z}_q^n,$$

we define the negacyclic rotation matrix

$$\text{Rot}_{\text{neg}}(\mathbf{a}) \in \mathbb{Z}_q^{n\times n}$$

by

$$\text{Rot}_{\text{neg}}(\mathbf{a}) := \left(\mathbf{a}, X_{\text{neg}}\mathbf{a}, \ldots, X_{\text{neg}}^{n-1}\mathbf{a}\right).$$

Explicitly,

$$\text{Rot}_{\text{neg}}(\mathbf{a}) = \begin{pmatrix} a_0 & -a_{n-1} & \cdots & -a_1 \\ a_1 & a_0 & \cdots & -a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1} & a_{n-2} & \cdots & a_0 \end{pmatrix}.$$

Although the compression function keeps the same formal shape,

$$h_A(\mathbf{z}) = A\mathbf{z} \bmod q,$$

its algebraic structure changes. A binary collision still gives a short ternary solution to the corresponding Ring-SIS relation over

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle.$$

However, the rank-one collapse from the cyclic case no longer applies, since, for odd q, the polynomial $x^n + 1$ does not vanish at $x = 1$ and therefore does not contain the linear factor $x - 1$ over $\mathbb{Z}_q$. Thus, the transition to the negacyclic modulus prevents the specific dimension-collapse attack based on evaluation at $x = 1$, while preserving efficient polynomial arithmetic through negacyclic convolution.

3.5.1 The Geometry of the Solution Space and Ideal Lattices

To explain the geometric meaning of the negacyclic Ring-SIS construction, we consider the solution space of the congruence

$$h_A(\mathbf{y}) = A\mathbf{y} \equiv 0 \pmod q$$

over the ring

$$R = \mathbb{Z}[x]/\langle x^n + 1\rangle.$$

Here

$$\mathbf{y} = (y_1(x), \ldots, y_\ell(x)) \in R^\ell$$

is a vector of ring elements, and its coefficient embedding lies in

$$\mathbb{Z}^{\ell n}.$$

This solution space is not an arbitrary lattice. Because the map h_A is R-linear, the set of solutions is closed under multiplication by ring elements. In particular, if

$$\mathbf{y} \in R^\ell$$

satisfies

$$A\mathbf{y} \equiv 0 \pmod q,$$

then for every $r(x) \in R$,

$$A(r\mathbf{y}) = r(A\mathbf{y}) \equiv 0 \pmod q.$$

Indeed, writing the Ring-SIS relation as

$$Ay = \sum_{i=1}^{\ell} a_i(x) y_i(x),$$

we obtain, for any $r(x) \in R$,

$$A(ry) = \sum_{i=1}^{\ell} a_i(x)\big(r(x)y_i(x)\big) = r(x)\sum_{i=1}^{\ell} a_i(x)y_i(x) = r(x)Ay.$$

Therefore, if $Ay \equiv 0 \pmod{q}$, then

$$A(ry) = r(x)Ay \equiv 0 \pmod{q}.$$

Thus, the solution set is closed under multiplication by elements of R, and hence it has an R-module structure.

Under the coefficient embedding, this gives a structured lattice, commonly referred to as an ideal or module lattice depending on the number of ring components.

In particular, multiplication by x maps

$$\mathbf{y} = (y_1(x), \ldots, y_\ell(x))$$

to

$$x\mathbf{y} = (xy_1(x), \ldots, xy_\ell(x)).$$

Since we work modulo $x^n + 1$, multiplication by x corresponds in each component to the negacyclic shift

$$X_{\text{neg}}.$$

This shift only permutes coefficients and changes signs. Therefore it preserves the Euclidean norm:

$$\|\mathbf{y}\|_2 = \|x\mathbf{y}\|_2 = \|x^2\mathbf{y}\|_2 = \cdots = \|x^{n-1}\mathbf{y}\|_2.$$

Consequently, if a non-zero short solution

$$\mathbf{y} = (y_1(x), \ldots, y_\ell(x)) \in R^\ell$$

satisfies the Ring-SIS relation

$$A\mathbf{y} \equiv 0 \pmod{q},$$

then the R-module structure gives the shifted solutions

$$\mathbf{y},\ x\mathbf{y},\ x^2\mathbf{y},\ \ldots,\ x^{n-1}\mathbf{y}.$$

4 The Generalized Birthday Problem

4.1 The Generalized Birthday Paradigm

The security of modern cryptographic systems, particularly hash functions, relies heavily on their resistance to collision attacks. The classic Birthday Problem serves as the theoretical foundation for understanding these attacks. In its traditional formulation, the problem is modeled as the search for two elements from two distinct lists,

$$x_1 \in L_1 \qquad \text{and} \qquad x_2 \in L_2,$$

that XOR to zero,

$$x_1 \oplus x_2 = 0.$$

It is mathematically proven that, when applying optimal search algorithms, the time complexity of solving this problem is

$$\Theta(2^{N/2}),$$

where N denotes the bit-length of the search space. This establishes what has served for years as a de facto speed limit in cryptanalysis.

Definition 4.1 (The Birthday Problem [2])**.** Given two lists L_1, L_2 of elements drawn uniformly and independently at random from $\{0,1\}^N$, find

$$x_1 \in L_1 \qquad \text{and} \qquad x_2 \in L_2$$

such that

$$x_1 \oplus x_2 = 0.$$

The above definition does not require the lists L_1 and L_2 to contain the same number of elements. The only mathematical condition for a solution to exist with high probability is that the product of their sizes greatly exceed the total space of possible values, that is,

$$|L_1| \cdot |L_2| \gg 2^N.$$

The first list contains $|L_1|$ elements and the second contains $|L_2|$ elements. By combining them, we can form a total of

$$|L_1| \cdot |L_2|$$

possible pairs. For each individual pair, the probability that the two values match is

$$\frac{1}{2^N}.$$

Therefore, to ensure a good probability of finding at least one match, our total attempts—that is, the possible pairs—must be much greater than the total space of possible digital values. This is expressed mathematically by the following condition:

$$|L_1| \cdot |L_2| \gg 2^N.$$

If the list sizes are favorably chosen, the optimal balance is obtained when

$$|L_1| \approx |L_2| \approx 2^{N/2}.$$

In this balanced case, the complexity of the optimal birthday search algorithm is

$$\Theta(2^{N/2}).$$

However, David Wagner's groundbreaking work shifted this paradigm by introducing a multidimensional generalization known as the k-sum problem. Within this expanded framework, an attacker has access to multiple data sources, namely lists

$$L_1, \ldots, L_k,$$

and seeks a combination of elements whose total XOR sum is equal to zero:

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = 0.$$

While probabilistic reasoning shows that such a solution exists with high probability provided that

$$|L_1| \cdot |L_2| \cdots |L_k| \gg 2^N,$$

the main algorithmic challenge is to find such a solution explicitly and with high computational efficiency.

Definition 4.2 (The k-Sum Problem [2])**.** Given k lists $L_1, \ldots, L_k$ of elements drawn uniformly and independently at random from $\{0,1\}^N$, find

$$x_1 \in L_1, \ldots, x_k \in L_k$$

such that

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = 0.$$

The lists can be expanded to any arbitrary size. To conceptualize this, one may imagine that the entries of each list are produced by a random or pseudorandom oracle R_i, meaning that the j-th item in L_i is given by

$$R_i(j).$$

Following the same probabilistic reasoning as in the classic birthday problem, if we select one element from each of the k lists, we can form a total of

$$|L_1| \cdot |L_2| \cdots |L_k|$$

possible k-tuples. Assuming that the elements are uniformly distributed, the probability that the XOR sum of any fixed k-tuple equals zero is exactly

$$\frac{1}{2^N}.$$

To see this, consider an arbitrary k-tuple

$$(x_1, x_2, \ldots, x_k)$$

and define S as the XOR sum of the first $k-1$ elements:

$$S = x_1 \oplus x_2 \oplus \cdots \oplus x_{k-1}.$$

The condition that the entire tuple sums to zero can then be written as

$$S \oplus x_k = 0.$$

Since XOR is its own inverse, this equality holds if and only if

$$x_k = S.$$

Because x_k is drawn uniformly from the N-bit space $\{0,1\}^N$, the probability that it takes this exact value S, among the 2^N possible values, is precisely

$$\frac{1}{2^N}.$$

Consequently, the expected number of zero-sum k-tuples is

$$\frac{|L_1| \cdot |L_2| \cdots |L_k|}{2^N}.$$

Thus, to guarantee that at least one such k-tuple exists with high probability, the total number of combinations must greatly exceed the size of the search space. This statistical prerequisite is expressed by the condition

$$|L_1| \cdot |L_2| \cdots |L_k| \gg 2^N.$$

Therefore, the primary difficulty does not lie in proving that a solution exists, but rather in designing an algorithm that discovers the exact values

$$x_1, \ldots, x_k$$

explicitly and with high computational efficiency.

4.2 Join-Based Search for the Two-List Birthday Problem

The classical birthday problem can be interpreted as a two-list matching problem. Given two lists L_1 and L_2 of N-bit values, the goal is to find

$$x_1 \in L_1, \qquad x_2 \in L_2,$$

such that

$$x_1 \oplus x_2 = 0.$$

Since

$$x_1 \oplus x_2 = 0 \quad \Longleftrightarrow \quad x_1 = x_2,$$

the problem reduces to finding a common element of the two lists.

This can be done using a join operation between the two lists. In practice, the join may be implemented in different ways. If enough memory is available, one can store one list in a hash table and scan the other list for matches. This gives a hash-join implementation with essentially linear time in the total list size. If memory is more restricted, one may instead sort both lists and then scan them simultaneously, obtaining a merge-join implementation.

In collision-search algorithms, the join must also preserve witnesses. A witness records which original list element produced a given value. Therefore, when two values are matched, their witnesses are merged as well. This allows the final collision, or the corresponding modular relation, to be reconstructed from the matched list elements.

The expected number of matches between the two lists is

$$\frac{|L_1| \cdot |L_2|}{2^N}.$$

Thus, when both lists have size approximately

$$|L_1| \approx |L_2| \approx 2^{N/2},$$

the expected number of matches is approximately one. Equivalently, a collision is found with constant probability.

The corresponding running time of the standard two-list birthday search is

$$O(2^{N/2})$$

for the hash-join implementation, assuming sufficient memory. More precisely, in the balanced setting, the classical birthday search has asymptotic complexity

$$\Theta(2^{N/2}).$$

This two-list join viewpoint is the basic idea that Wagner later generalizes to the k-list setting.

4.3 The Four-List Birthday Algorithm

The first non-trivial improvement beyond the classical birthday search appears in the case of four lists. Suppose that we are given four lists

$$L_1,\ L_2,\ L_3,\ L_4$$

of N-bit values, and we want to find

$$x_i \in L_i \qquad (1 \le i \le 4)$$

such that

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0.$$

A direct birthday-style approach would still require about

$$O(2^{N/2})$$

operations. Wagner's idea is to avoid searching over all quadruples directly. Instead, the four lists are first paired as

$$(L_1, L_2) \qquad \text{and} \qquad (L_3, L_4).$$

For a filtering parameter τ, we keep only pairs whose XOR is zero in the lowest τ bits. Thus, from L_1 and L_2, we form an intermediate list L_{12} containing values of the form

$$x_1 \oplus x_2,$$

where

$$\mathrm{low}_\tau(x_1 \oplus x_2) = 0.$$

Similarly, from L_3 and L_4, we form a second intermediate list L_{34} containing values of the form

$$x_3 \oplus x_4,$$

with

$$\mathrm{low}_\tau(x_3 \oplus x_4) = 0.$$

The final step is to search for a match between the two intermediate lists. Indeed, if

$$x_1 \oplus x_2 = x_3 \oplus x_4,$$

then

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0.$$

Therefore, a match between L_{12} and L_{34} gives a solution to the four-list birthday problem.

To analyze the complexity, suppose that each original list has size approximately

$$2^\tau.$$

For a random pair $(x_1, x_2) \in L_1 \times L_2$, the probability that

$$\mathrm{low}_\tau(x_1 \oplus x_2) = 0$$

is

$$2^{-\tau}.$$

Hence the expected size of the intermediate list L_{12} is

$$\frac{2^\tau \cdot 2^\tau}{2^\tau} = 2^\tau.$$

The same estimate holds for L_{34}.

Since all elements in L_{12} and L_{34} already have their lowest τ bits equal to zero, the final matching step takes place over the remaining $N - \tau$ bits. Thus, the expected number of final matches is approximately

$$\frac{|L_{12}| \cdot |L_{34}|}{2^{N-\tau}} \approx \frac{2^\tau \cdot 2^\tau}{2^{N-\tau}} = 2^{3\tau - N}.$$

To obtain a constant probability of success, we choose

$$\tau = \frac{N}{3}.$$

Consequently, all lists involved have size about

$$2^{N/3},$$

and the four-list birthday problem can be solved in time and memory

$$O(2^{N/3}).$$

This is the key improvement over the classical

$$O(2^{N/2})$$

birthday bound. The same tree-based idea is later generalized by Wagner to the k-list setting.

Wagner also notes that the zero target is not essential. An equation of the form

$$x_1 \oplus \cdots \oplus x_k = c$$

can be reduced to the zero-sum case by shifting one of the lists by c.

Moreover, when k is a power of two, the same idea extends to a complete binary tree of joins. At each internal level, partial sums are merged while forcing progressively more bits to match, and each intermediate element stores back-pointers to the original list elements. This yields Wagner's k-tree algorithm, with time and space complexity

$$O\left(k \cdot 2^{N/(1+\log_2 k)}\right)$$

under the standard random-list heuristic.

4.4 The k-Tree Algorithm for Modular Arithmetic

We can also apply the tree algorithm to the group $(\mathbb{Z}/m\mathbb{Z}, +)$, where m is arbitrary. Let

$$[a, b]=\{x \in \mathbb{Z}/m\mathbb{Z} : a \le x \le b\}$$

denote the interval of elements between a and b, with wrapping modulo m. We define the join operation

$$L_1 \bowtie_{[a,b]} L_2$$

to represent the solutions to

$$x_1 + x_2 \in [a, b], \qquad x_i \in L_i.$$

Then we may solve a 4-sum problem over $\mathbb{Z}/m\mathbb{Z}$ by computing

$$(L_1 \bowtie_{[a,b]} L_2) \bowtie (L_3 \bowtie_{[a,b]} L_4),$$

where

$$[a, b] = \left[-\frac{m}{2^{N+1}}, \frac{m}{2^{N+1}} - 1\right].$$

In its practical algorithmic implementation, this model generalizes to solve the k-sum problem, where

$$k = 2^D$$

for some integer D, so that the computation forms a complete binary tree. This is done by employing a dual join strategy. At the intermediate, non-root levels of the tree, the algorithm uses the aforementioned interval join, or interval filtering, dynamically adjusting the radius of the allowed interval at each height in order to progressively restrict the bounds.

However, at the root of the tree, namely the final join step, the strategy shifts to exact modular complement matching. The algorithm negates the elements of one list, computing

$$-y \pmod{m},$$

and searches for exact equalities via a hash join, ensuring that the final sum is strictly zero modulo m.

Throughout this process, each generated element maintains a witness, or back-pointer, which allows the algorithm, upon finding the zero sum at the root, to trace the path backward and accurately recover the k original elements

$$x_0, \ldots, x_{k-1}$$

that form the complete solution.

4.5 The k-Tree Attack on the Scalar Problem

To find a non-zero vector

$$\varepsilon \in \{-1, 0, 1\}^\ell \setminus \{0\}$$

satisfying the scalar relation derived from the rank-one collapse, namely

$$\sum_{i=1}^{\ell} \varepsilon_i a_i(1) \equiv 0 \pmod{q},$$

a brute-force exhaustive search is practically infeasible. Since each coefficient ε_i takes values in the set $\{-1, 0, 1\}$, the total search space has size

$$3^\ell,$$

which grows exponentially in ℓ.

Furthermore, even a classical meet-in-the-middle strategy, which partitions the problem into two halves, is bottlenecked by a complexity of approximately

$$O(3^{\ell/2}).$$

Equivalently, when

$$3^\ell \approx q,$$

this corresponds to the classical birthday-type complexity

$$O(\sqrt{q}).$$

This remains inadequate for large values of ℓ or large modulus q.

To overcome these computational limitations, we reduce the scalar search task to a Generalized Birthday Problem, or k-sum problem. Specifically, we partition the initial set of ℓ scalar coefficients

$$a_1(1), \ldots, a_\ell(1)$$

into k disjoint subsets. For each subset, we compute all possible partial sums using the allowed ternary coefficients $\{-1, 0, 1\}$. In this way, we generate k initial base lists.

These k lists are then given as input to a customized generalized k-tree modular solver. Through hierarchical interval filtering at the intermediate tree levels, the algorithm systematically discards partial sums that fall outside the permitted target interval, thereby pruning the search space. Upon reaching the root of the tree, the algorithm applies an exact modular hash join to identify a combination of intermediate values whose total sum is precisely

$$0 \pmod{q}.$$

Finally, by tracing back through the stored witnesses, or back-pointers, the algorithm recovers the specific individual coefficients

$$\varepsilon_1, \ldots, \varepsilon_\ell$$

that compose the final zero sum. Thus, the method produces a non-zero ternary solution to

$$\sum_{i=1}^{\ell} \varepsilon_i a_i(1) \equiv 0 \pmod{q},$$

with substantially better efficiency than naive exhaustive search.

4.6 The k-Tree Attack on the Full Negacyclic Problem

Unlike the scalar rank-one collapse, the full negacyclic attack does not reduce the problem to a one-dimensional congruence. The algorithm works directly in $\mathbb{Z}_q^n$, and its effectiveness depends on the behavior of the intermediate vector lists. Since the negacyclic columns have algebraic correlations, the standard Wagner complexity estimates apply only under a random-list heuristic. Nevertheless, the procedure provides a natural tree-based framework for searching for short ternary kernel vectors in the full negacyclic Ring-SIS instance.

We now describe how the k-tree methodology can be applied to the full negacyclic Ring-SIS instance. In contrast to the scalar collapse attack, which reduces the problem to a one-dimensional congruence modulo q, the full negacyclic problem works directly with vectors in $\mathbb{Z}_q^n$.

Let

$$A \in \mathbb{Z}_q^{n \times m}$$

be the structured public matrix obtained from the negacyclic Ring-SIS instance, and write its columns as

$$A = \left[\mathbf{c}_0 \mid \mathbf{c}_1 \mid \cdots \mid \mathbf{c}_{m-1}\right], \qquad \mathbf{c}_j \in \mathbb{Z}_q^n.$$

The goal is to find a non-zero ternary vector

$$z = (z_0, \ldots, z_{m-1}) \in \{-1, 0, 1\}^m \setminus \{0\}$$

such that

$$Az = \sum_{j=0}^{m-1} z_j \mathbf{c}_j \equiv 0 \pmod{q}.$$

This is a vector-valued modular k-sum problem over the additive group

$$(\mathbb{Z}_q^n, +).$$

Step 1: Extract the negacyclic column vectors. The algorithm begins by extracting the individual column vectors

$$\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{m-1} \in \mathbb{Z}_q^n$$

from the structured negacyclic public matrix A. These columns are not arbitrary independent random vectors: within each negacyclic block, they are signed negacyclic shifts of the same underlying polynomial coefficient vector. Therefore, the application of Wagner's k-tree method to the full vector problem should be understood under a random-list heuristic.

Step 2: Partition the columns into Wagner groups. Choose

$$k = 2^D$$

for some integer D, so that the merging process can be organized as a complete binary tree of depth D. Let

$$\sigma$$

be a random permutation of the column indices

$$\{0, 1, \ldots, m-1\}.$$

The permuted column indices are partitioned into k disjoint groups

$$G_0, G_1, \ldots, G_{k-1}.$$

Ideally, the partition is balanced, so that each group contains approximately

$$s = \frac{m}{k}$$

indices. In particular, when $k \mid m$, we set $s = m/k$ and define

$$G_r = \{\sigma(rs), \sigma(rs+1), \ldots, \sigma((r+1)s-1)\}, \qquad r = 0, \ldots, k-1.$$

This random permutation, or alternatively a balanced assignment across negacyclic blocks, is used in order to avoid placing too many highly correlated shifted columns in the same group.

Step 3: Build the initial partial-sum lists. For each group G_r, the algorithm generates local ternary combinations of the columns indexed by G_r. For a local coefficient vector

$$\varepsilon^{(r)} = \left(\varepsilon_j\right)_{j \in G_r} \in \{-1, 0, 1\}^{G_r},$$

we compute the partial sum

$$\mathbf{S}_r(\varepsilon^{(r)}) = \sum_{j \in G_r} \varepsilon_j \mathbf{c}_j \pmod q.$$

Each generated element stores both the vector

$$\mathbf{S}_r(\varepsilon^{(r)}) \in \mathbb{Z}_q^n$$

and a witness, namely the local ternary coefficient choices

$$\varepsilon^{(r)}.$$

Thus, the initial list associated with the group G_r is

$$\mathcal{L}_r = \left\{ \left(\mathbf{S}_r(\varepsilon^{(r)}), \varepsilon^{(r)} \right) : \varepsilon^{(r)} \in \{-1, 0, 1\}^{G_r} \right\}.$$

In the exhaustive version, the initial list for G_r has size

$$3^{|G_r|}.$$

However, this size grows exponentially. Therefore, in practical implementations one may retain only up to $L_{\max}$ candidates per group. In that case, the algorithm becomes a truncated, heuristic search rather than a fully exhaustive enumeration.

We denote the initial lists by

$$\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{k-1}.$$

Step 4: Perform hierarchical vector merging. The core of the attack is a Wagner-style hierarchical merging procedure. At each internal level of the binary tree, pairs of lists are merged. Suppose two intermediate lists contain elements of the form

$$(\mathbf{S}, \omega) \qquad \text{and} \qquad (\mathbf{T}, \nu),$$

where

$$\mathbf{S}, \mathbf{T} \in \mathbb{Z}_q^n$$

are partial vector sums and ω, ν are the corresponding witnesses.

At an intermediate level, the algorithm imposes filtering conditions on selected coordinate positions. Let

$$I_h \subseteq \{1, \ldots, n\}$$

denote the set of coordinates filtered at level h. The merge keeps only pairs satisfying

$$\pi_{I_h}(\mathbf{S} + \mathbf{T}) \equiv 0 \pmod{q},$$

where π_{I_h} denotes projection onto the coordinates indexed by I_h. For each accepted pair, the algorithm forms the new partial sum

$$\mathbf{U} = \mathbf{S} + \mathbf{T} \pmod{q}$$

and concatenates the witnesses to obtain the combined ternary choices.

As the tree levels increase, the coordinate constraints become progressively stronger. This does not reduce the ambient vector dimension: all vectors remain elements of $\mathbb{Z}_q^n$. Rather, it reduces the number of admissible candidate sums by forcing more coordinates to satisfy the target congruence.

If an intermediate list becomes too large, it may again be truncated to at most $L_{\max}$ elements. This is a practical memory-control step and should be understood as part of the heuristic implementation.

Step 5: Apply exact zero matching at the root. Upon reaching the root of the tree, exactly two lists remain. At this final stage, no partial coordinate filtering is used. Instead, the algorithm performs an exact modular complement match over the full vector space

$$\mathbb{Z}_q^n.$$

That is, it searches for two remaining elements

$$(\mathbf{S}_1, \omega_1) \qquad \text{and} \qquad (\mathbf{S}_2, \omega_2)$$

such that

$$\mathbf{S}_1 + \mathbf{S}_2 \equiv 0 \pmod{q}.$$

Equivalently, one may store one list in a hash table and search for exact matches against the negated values of the other list:

$$\mathbf{S}_1 \equiv -\mathbf{S}_2 \pmod{q}.$$

When such a match is found, the stored witnesses are traced backward through the tree. This reconstructs a full ternary vector

$$z \in \{-1, 0, 1\}^m$$

satisfying

$$Az \equiv 0 \pmod{q}.$$

If the reconstructed vector is the zero vector, it is discarded; otherwise, it is a valid non-zero Ring-SIS solution.

Step 6: Extract a binary collision. The standard Ring-SIS collision requirement is a pair of distinct binary input vectors. Given a non-zero ternary solution

$$z \in \{-1, 0, 1\}^m$$

satisfying

$$Az \equiv 0 \pmod{q},$$

we decompose it into its positive and negative parts:

$$z = z^+ - z^-,$$

where

$$z_j^+ = \begin{cases} 1, & z_j = 1, \\ 0, & \text{otherwise}, \end{cases} \qquad z_j^- = \begin{cases} 1, & z_j = -1, \\ 0, & \text{otherwise}. \end{cases}$$

Thus,

$$z^+, z^- \in \{0, 1\}^m.$$

Since $z \neq 0$, the two binary vectors z^+ and z^- are distinct. Moreover,

$$A(z^+ - z^-) \equiv 0 \pmod{q},$$

and therefore

$$Az^+ \equiv Az^- \pmod{q}.$$

Hence,

$$h_A(z^+) = h_A(z^-),$$

so the pair

$$(z^+, z^-)$$

is a valid binary collision for the compression function.

Proposition 4.1 (Heuristic non-starvation condition)**.** *Consider the full negacyclic Wagner attack over $\mathbb{Z}_q^n$. Assume that, at an intermediate merge level, two input lists*

$$\mathcal{L}_1, \mathcal{L}_2$$

have sizes

$$|\mathcal{L}_1| = L_1, \qquad |\mathcal{L}_2| = L_2.$$

Assume also that the partial sums in these lists behave as independent uniform random vectors in $\mathbb{Z}_q^n$. If the merge filters r coordinates, then the expected size of the merged list is

$$\mathbb{E}\big[|\mathcal{L}_{\text{next}}|\big] = \frac{L_1 L_2}{q^r}.$$

Therefore, a heuristic condition for avoiding list starvation at this merge level is

$$\frac{L_1 L_2}{q^r} \geq 1.$$

More safely, one should require

$$\frac{L_1 L_2}{q^r} \gg 1.$$

Proof. At this merge level, the algorithm considers pairs

$$(\mathbf{S}, \mathbf{T}) \in \mathcal{L}_1 \times \mathcal{L}_2,$$

where

$$\mathbf{S}, \mathbf{T} \in \mathbb{Z}_q^n.$$

The merge keeps a pair only if

$$\pi_I(\mathbf{S} + \mathbf{T}) \equiv 0 \pmod{q},$$

where

$$I \subseteq \{0, \ldots, n-1\}, \qquad |I| = r,$$

and π_I denotes projection onto the coordinates indexed by I.

Under the random-list heuristic, the projected sum

$$\pi_I(\mathbf{S} + \mathbf{T})$$

is uniformly distributed in

$$\mathbb{Z}_q^r.$$

Since

$$|\mathbb{Z}_q^r| = q^r,$$

the probability that this projected sum is zero is

$$\Pr\big[\pi_I(\mathbf{S} + \mathbf{T}) = 0\big] = \frac{1}{q^r}.$$

There are

$$L_1 L_2$$

candidate pairs. Hence, by linearity of expectation,

$$\mathbb{E}\big[|\mathcal{L}_{\text{next}}|\big] = L_1 L_2 \cdot \frac{1}{q^r} = \frac{L_1 L_2}{q^r}.$$

If this expected value is below 1, then the new list is expected to contain less than one surviving element. In that case, the list is likely to become empty, which is precisely list starvation. □

Proposition 4.2 (Heuristic final matching condition)**.** *At the root of the full negacyclic Wagner tree, suppose that two final lists remain,*

$$\mathcal{L}_{\text{left}}, \qquad \mathcal{L}_{\text{right}},$$

with sizes

$$|\mathcal{L}_{\text{left}}| = L_{\text{left}}, \qquad |\mathcal{L}_{\text{right}}| = L_{\text{right}}.$$

Assume that the previous merge levels have already imposed zero constraints on a set of coordinates

$$J \subseteq \{0, \dots, n-1\},$$

where

$$|J| = R.$$

Assume further that, conditioned on these previous constraints, the remaining unconstrained coordinates behave as independent uniform elements of $\mathbb{Z}_q$. Then the expected number of exact root matches is

$$\mathbb{E}[N_{\text{root}}] = \frac{L_{\text{left}} L_{\text{right}}}{q^{n-R}}.$$

Therefore, a heuristic condition for obtaining a root match with non-negligible probability is

$$\frac{L_{\text{left}} L_{\text{right}}}{q^{n-R}} \geq 1.$$

More generally, if one wants at least $\lambda > 0$ expected root matches, one should require

$$\frac{L_{\text{left}} L_{\text{right}}}{q^{n-R}} \geq \lambda.$$

Proof. At the root, the algorithm searches for

$$\mathbf{S} + \mathbf{T} \equiv 0 \pmod{q},$$

where

$$\mathbf{S} \in \mathcal{L}_{\text{left}}, \qquad \mathbf{T} \in \mathcal{L}_{\text{right}}.$$

Equivalently, it searches for

$$\mathbf{S} \equiv -\mathbf{T} \pmod{q}.$$

However, the vectors in the root lists are not arbitrary fresh elements of $\mathbb{Z}_q^n$. They have already passed the intermediate Wagner filters. Let

$$J \subseteq \{0, \dots, n-1\}$$

be the set of coordinates that have already been forced to be zero through the previous merge levels, and let

$$|J| = R.$$

Therefore, the root match only has to enforce the remaining

$$n - R$$

coordinates.

Under the random-list heuristic, conditioned on the previous constraints, the unconstrained part of

$$\mathbf{S} + \mathbf{T}$$

is uniformly distributed in

$$\mathbb{Z}_q^{n-R}.$$

Since

$$|\mathbb{Z}_q^{n-R}| = q^{n-R},$$

the probability that a fixed pair $(\mathbf{S}, \mathbf{T})$ satisfies all the remaining root constraints is

$$\frac{1}{q^{n-R}}.$$

There are

$$L_{\text{left}} L_{\text{right}}$$

candidate pairs. Hence, by linearity of expectation,

$$\mathbb{E}[N_{\text{root}}] = L_{\text{left}} L_{\text{right}} \cdot \frac{1}{q^{n-R}} = \frac{L_{\text{left}} L_{\text{right}}}{q^{n-R}}.$$

Thus, if this expected value is at least 1, then the random-list heuristic predicts at least one exact root match on average. □

Proposition 4.3 (Heuristic initial list size requirement)**.** *Let $k = 2^D$ be the number of Wagner lists in the full negacyclic Wagner attack. Suppose that the intermediate filtering levels are*

$$h = 1, \ldots, D-1,$$

and that at level h the merge filters r_h coordinates. Let

$$J = \bigcup_{h=1}^{D-1} I_h, \qquad R = |J|,$$

be the set of coordinates already constrained before the root. Let $\lambda \geq 1$ be a desired expected-match safety factor.

Define

$$T_{D-1} = \sqrt{\lambda q^{\,n-R}},$$

and recursively, for

$$h = D-1, D-2, \ldots, 1,$$

define

$$T_{h-1} = \sqrt{\lambda T_h q^{r_h}}.$$

Then, under the random-list heuristic, an initial list size satisfying

$$L_0 \geq T_0$$

is sufficient heuristically to avoid starvation at the intermediate levels and to expect at least λ final root matches.

Proof. At an intermediate merge level h, two lists of size approximately L_{h-1} produce approximately

$$L_{h-1}^2$$

candidate pairs. If the merge filters r_h coordinates, then under the random-list heuristic a fixed pair survives with probability

$$q^{-r_h}.$$

Therefore, the expected size of the next list is

$$L_h \approx \frac{L_{h-1}^2}{q^{r_h}}.$$

To obtain a next list of expected size at least T_h, with safety factor λ, it is enough to require

$$\frac{L_{h-1}^2}{q^{r_h}} \geq \lambda T_h.$$

Equivalently,

$$L_{h-1} \geq \sqrt{\lambda T_h q^{r_h}}.$$

This gives the backward recursion

$$T_{h-1} = \sqrt{\lambda T_h q^{r_h}}.$$

At the root, the previous intermediate filters have already constrained the coordinates in

$$J = \bigcup_{h=1}^{D-1} I_h,$$

where $R = |J|$. Hence only $n - R$ coordinates remain to be matched. If the two root lists have size approximately L_{D-1}, then the expected number of exact root matches is

$$\frac{L_{D-1}^2}{q^{n-R}}.$$

To expect at least λ root matches, it is enough to require

$$\frac{L_{D-1}^2}{q^{n-R}} \geq \lambda.$$

Equivalently,

$$L_{D-1} \geq \sqrt{\lambda q^{n-R}}.$$

Thus we set

$$T_{D-1} = \sqrt{\lambda q^{n-R}}.$$

Applying the intermediate-level recursion backwards gives T_0. Therefore, if

$$L_0 \geq T_0,$$

then, under the random-list heuristic, every intermediate level has sufficient expected list size, and the root has at least λ expected exact zero matches. □

5 Implementation and Experimental Evaluation

5.1 Experimental Setup

The experiments were implemented in Python and were designed to compare the behavior of the collision-generation procedures for the cyclic ring

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle$$

and the negacyclic ring

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle.$$

For each experiment, we fixed the ring dimension n, the modulus q, the number of blocks ℓ, and hence the total input dimension

$$m = n\ell.$$

In the cyclic case, the implementation first computes the scalar values

$$\alpha_i = a_i(1) \pmod{q}, \qquad i = 1, \ldots, \ell,$$

and then searches for a non-zero ternary relation

$$\sum_{i=1}^{\ell} \varepsilon_i \alpha_i \equiv 0 \pmod{q}.$$

The resulting relation is lifted to a ternary Ring-SIS solution and then decomposed into a binary collision pair.

In the negacyclic case, the implementation works directly with the full negacyclic matrix

$$A \in \mathbb{Z}_q^{n \times m}$$

and searches for a non-zero ternary vector

$$z \in \{-1, 0, 1\}^m$$

such that

$$Az \equiv 0 \pmod{q}.$$

This is done using the full vector-valued Wagner-style procedure described in the previous section.

For both cases, we record the running time, memory usage, and whether a valid collision was found. All reported collisions are verified by checking that

$$Az^+ \equiv Az^- \pmod{q}.$$

5.2 Example over the Cyclic Ring $\mathbb{Z}_q[x]/\langle x^n - 1\rangle$

We now illustrate the collision-generation pipeline with a small example that can be verified by hand. Let

$$n = 2, \qquad q = 5, \qquad \ell = 3.$$

Then

$$m = n\ell = 2 \cdot 3 = 6.$$

We work over the cyclic quotient ring

$$\mathcal{R}_q = \mathbb{Z}_5[x]/\langle x^2 - 1\rangle.$$

The public matrix has the block-circulant Ring-SIS form

$$A = [\mathrm{Rot}(a_1) \mid \mathrm{Rot}(a_2) \mid \mathrm{Rot}(a_3)] \in \mathbb{Z}_5^{2\times 6}.$$

This is the standard structured Ring-SIS representation, where the full public matrix is generated from cyclic rotations of short public blocks.

Let the public blocks be

$$a_1 = (2, 4), \qquad a_2 = (1, 1), \qquad a_3 = (3, 3).$$

Thus,

$$\mathrm{Rot}(a_1) = \begin{pmatrix} 2 & 4 \\ 4 & 2 \end{pmatrix},$$

$$\mathrm{Rot}(a_2) = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

and

$$\mathrm{Rot}(a_3) = \begin{pmatrix} 3 & 3 \\ 3 & 3 \end{pmatrix}.$$

Therefore, the full public matrix is

$$A = \begin{pmatrix} 2 & 4 & 1 & 1 & 3 & 3 \\ 4 & 2 & 1 & 1 & 3 & 3 \end{pmatrix} \pmod 5.$$

The first step is to compute the scalar values induced by evaluation at $x = 1$. For each block,

$$\alpha_i = a_i(1) = \sum_{j=0}^{n-1} a_{i,j} \pmod q.$$

Hence,

$$\alpha_1 = a_1(1) = 2 + 4 = 6 \equiv 1 \pmod 5,$$

$$\alpha_2 = a_2(1) = 1 + 1 = 2 \pmod 5,$$

and

$$\alpha_3 = a_3(1) = 3 + 3 = 6 \equiv 1 \pmod 5.$$

Thus,

$$(\alpha_1, \alpha_2, \alpha_3) = (1, 2, 1).$$

We now search for two different binary subset vectors that give the same subset sum modulo 5. Consider

$$u = (0, 0, 1), \qquad v = (1, 0, 0).$$

Then

$$\sum_{i=1}^{3} u_i \alpha_i = 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 1 = 1 \pmod 5,$$

while

$$\sum_{i=1}^{3} v_i \alpha_i = 1 \cdot 1 + 0 \cdot 2 + 0 \cdot 1 = 1 \pmod 5.$$

Therefore, the two subset sums collide modulo 5. Taking the difference gives

$$\epsilon = u - v.$$

Hence,

$$\epsilon = (0, 0, 1) - (1, 0, 0) = (-1, 0, 1).$$

We verify that this is a signed modular relation:

$$\sum_{i=1}^{3} \epsilon_i \alpha_i = (-1) \cdot 1 + 0 \cdot 2 + 1 \cdot 1 = 0 \equiv 0 \pmod 5.$$

We now lift this scalar relation back to the ring. Since $n = 2$, the all-ones polynomial is

$$\widetilde{e}(x) = 1 + x.$$

For each $i = 1, 2, 3$, define

$$z_i(x) = \epsilon_i \widetilde{e}(x).$$

Using

$$\epsilon = (-1, 0, 1),$$

we obtain

$$z_1(x) = -(1 + x),$$

$$z_2(x) = 0,$$

and

$$z_3(x) = 1 + x.$$

In coefficient-vector form, this gives

$$z_1 = (-1, -1), \qquad z_2 = (0, 0), \qquad z_3 = (1, 1).$$

Thus, the lifted ternary vector is

$$z = \begin{pmatrix} -1 \\ -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \in \{-1, 0, 1\}^6.$$

We now verify directly that z lies in the kernel of A modulo 5. We compute

$$Az = \begin{pmatrix} 2 & 4 & 1 & 1 & 3 & 3 \\ 4 & 2 & 1 & 1 & 3 & 3 \end{pmatrix} \begin{pmatrix} -1 \\ -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \pmod 5.$$

The first coordinate is

$$2(-1) + 4(-1) + 1 \cdot 0 + 1 \cdot 0 + 3 \cdot 1 + 3 \cdot 1 = -2 - 4 + 3 + 3 = 0.$$

The second coordinate is

$$4(-1) + 2(-1) + 1 \cdot 0 + 1 \cdot 0 + 3 \cdot 1 + 3 \cdot 1 = -4 - 2 + 3 + 3 = 0.$$

Therefore,

$$Az \equiv \begin{pmatrix} 0 \\ 0 \end{pmatrix} \pmod 5.$$

So z is a non-zero ternary kernel vector.
The final step is to decompose z into two binary vectors. We define

$$z^+ = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \qquad z^- = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

Then

$$z^+ - z^- = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = z.$$

Since $Az \equiv 0 \pmod 5$, it follows that

$$A(z^+ - z^-) \equiv 0 \pmod 5.$$

Equivalently,

$$Az^+ \equiv Az^- \pmod 5.$$

We can verify this directly. First,

$$Az^+ = \begin{pmatrix} 2 & 4 & 1 & 1 & 3 & 3 \\ 4 & 2 & 1 & 1 & 3 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3+3 \\ 3+3 \end{pmatrix} = \begin{pmatrix} 6 \\ 6 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 1 \end{pmatrix} \pmod 5.$$

Similarly,

$$Az^- = \begin{pmatrix} 2 & 4 & 1 & 1 & 3 & 3 \\ 4 & 2 & 1 & 1 & 3 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2+4 \\ 4+2 \end{pmatrix} = \begin{pmatrix} 6 \\ 6 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 1 \end{pmatrix} \pmod 5.$$

Hence,

$$Az^+ \equiv Az^- \pmod 5.$$

Therefore,

$$h_A(z^+) = h_A(z^-).$$

The two inputs are distinct and non-zero:

$$z^+ \neq z^-, \qquad z^+ \neq 0, \qquad z^- \neq 0.$$

Thus,

$$\left(z^+, z^-\right)$$

is a non-trivial binary collision for the cyclic Ring-SIS compression function.

5.3 Example over the Negacyclic Ring $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$

We now illustrate the negacyclic Wagner collision-generation procedure with a small example produced by the Python implementation. Let

$$n = 2, \qquad q = 5, \qquad \ell = 3.$$

Then

$$m = n\ell = 2 \cdot 3 = 6.$$

We work over the negacyclic quotient ring

$$R_q = \mathbb{Z}_5[x]/\langle x^2 + 1\rangle.$$

The public matrix has the block-negacyclic Ring-SIS form

$$A = \left[\mathrm{Rot}_{\mathrm{neg}}(a_1) \mid \mathrm{Rot}_{\mathrm{neg}}(a_2) \mid \mathrm{Rot}_{\mathrm{neg}}(a_3)\right] \in \mathbb{Z}_5^{2\times 6}.$$

Let the public blocks be

$$a_1 = (1, 1), \qquad a_2 = (1, 2), \qquad a_3 = (1, 3).$$

Since we work modulo $x^2 + 1$, we have

$$x^2 \equiv -1 \pmod{x^2 + 1}.$$

Thus, for a block $a = (a_0, a_1)$, the negacyclic rotation matrix is

$$\mathrm{Rot}_{\mathrm{neg}}(a) = \begin{pmatrix} a_0 & -a_1 \\ a_1 & a_0 \end{pmatrix} \pmod 5.$$

Therefore,

$$\mathrm{Rot}_{\mathrm{neg}}(a_1) = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \equiv \begin{pmatrix} 1 & 4 \\ 1 & 1 \end{pmatrix} \pmod 5,$$

$$\mathrm{Rot}_{\mathrm{neg}}(a_2) = \begin{pmatrix} 1 & -2 \\ 2 & 1 \end{pmatrix} \equiv \begin{pmatrix} 1 & 3 \\ 2 & 1 \end{pmatrix} \pmod 5,$$

and

$$\mathrm{Rot}_{\mathrm{neg}}(a_3) = \begin{pmatrix} 1 & -3 \\ 3 & 1 \end{pmatrix} \equiv \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix} \pmod 5.$$

Hence, the full public matrix is

$$A = \begin{pmatrix} 1 & 4 & 1 & 3 & 1 & 2 \\ 1 & 1 & 2 & 1 & 3 & 1 \end{pmatrix} \pmod 5.$$

Unlike the cyclic example, we do not reduce the problem to the scalar evaluation at $x = 1$. Instead, the negacyclic Wagner procedure searches directly for a ternary vector

$$z \in \{-1, 0, 1\}^6$$

such that

$$Az \equiv 0 \pmod 5.$$

The Python implementation returns the ternary vector

$$z = \begin{pmatrix} -1 \\ -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{pmatrix}.$$

We now explain how the Wagner step produces the ternary vector in this small negacyclic example. Since

$$k = 2 = 2^1,$$

the Wagner tree has depth $D = 1$. Therefore, there are no intermediate filtering levels in this toy example. The algorithm only builds two initial partial-sum lists and then performs the exact root matching step.

The columns of the public matrix are

$$c_0 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad c_1 = \begin{pmatrix} 4 \\ 1 \end{pmatrix}, \quad c_2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad c_3 = \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \quad c_4 = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad c_5 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

We split the columns consecutively into two Wagner groups:

$$G_0 = \{0, 1, 2\}, \qquad G_1 = \{3, 4, 5\}.$$

For the first group, the algorithm considers ternary combinations of c_0, c_1, c_2. One such local choice is

$$\varepsilon^{(0)} = (-1, -1, 1).$$

This gives the partial sum

$$S_0 = -c_0 - c_1 + c_2 = -\begin{pmatrix}1\\1\end{pmatrix} - \begin{pmatrix}4\\1\end{pmatrix} + \begin{pmatrix}1\\2\end{pmatrix} = \begin{pmatrix}-4\\0\end{pmatrix} \equiv \begin{pmatrix}1\\0\end{pmatrix} \pmod 5.$$

For the second group, the algorithm considers ternary combinations of c_3, c_4, c_5. One such local choice is

$$\varepsilon^{(1)} = (-1, -1, -1).$$

This gives

$$S_1 = -c_3 - c_4 - c_5 = -\begin{pmatrix}3\\1\end{pmatrix} - \begin{pmatrix}1\\3\end{pmatrix} - \begin{pmatrix}2\\1\end{pmatrix} = \begin{pmatrix}-6\\-5\end{pmatrix} \equiv \begin{pmatrix}4\\0\end{pmatrix} \pmod 5.$$

At the root, the algorithm searches for two partial sums satisfying

$$S_0 + S_1 \equiv 0 \pmod 5.$$

Indeed,

$$S_0 + S_1 \equiv \begin{pmatrix}1\\0\end{pmatrix} + \begin{pmatrix}4\\0\end{pmatrix} = \begin{pmatrix}5\\0\end{pmatrix} \equiv \begin{pmatrix}0\\0\end{pmatrix} \pmod 5.$$

Thus, the root match combines the two local witnesses

$$\varepsilon^{(0)} = (-1, -1, 1), \qquad \varepsilon^{(1)} = (-1, -1, -1).$$

Concatenating them gives the full ternary vector

$$z = (-1, -1, 1, -1, -1, -1).$$

Therefore, the vector z was obtained by Wagner's two-list root matching:

$$-c_0 - c_1 + c_2 - c_3 - c_4 - c_5 \equiv 0 \pmod 5.$$

Equivalently,

$$Az \equiv 0 \pmod 5.$$

We verify directly that z lies in the kernel of A modulo 5. We compute

$$Az = \begin{pmatrix}1 & 4 & 1 & 3 & 1 & 2\\ 1 & 1 & 2 & 1 & 3 & 1\end{pmatrix}\begin{pmatrix}-1\\-1\\1\\-1\\-1\\-1\end{pmatrix} \pmod 5.$$

The first coordinate is

$$1(-1)+4(-1)+1(1)+3(-1)+1(-1)+2(-1) = -1-4+1-3-1-2 = -10 \equiv 0 \pmod 5.$$

The second coordinate is

$$1(-1)+1(-1)+2(1)+1(-1)+3(-1)+1(-1) = -1-1+2-1-3-1 = -5 \equiv 0 \pmod 5.$$

Therefore,

$$Az \equiv \begin{pmatrix} 0 \\ 0 \end{pmatrix} \pmod 5.$$

So z is a non-zero ternary kernel vector for the negacyclic Ring-SIS instance.
The final step is to decompose z into two binary vectors. We define

$$z^+ = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \qquad z^- = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Then

$$z^+ - z^- = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{pmatrix} = z.$$

Since

$$Az \equiv 0 \pmod 5,$$

we obtain

$$A(z^+ - z^-) \equiv 0 \pmod 5.$$

Equivalently,

$$Az^+ \equiv Az^- \pmod 5.$$

We verify this directly. First,

$$Az^+ = \begin{pmatrix} 1 & 4 & 1 & 3 & 1 & 2 \\ 1 & 1 & 2 & 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \pmod 5.$$

Similarly,

$$Az^- = \begin{pmatrix} 1 & 4 & 1 & 3 & 1 & 2 \\ 1 & 1 & 2 & 1 & 3 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1+4+3+1+2 \\ 1+1+1+3+1 \end{pmatrix} = \begin{pmatrix} 11 \\ 7 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 2 \end{pmatrix} \pmod 5.$$

Hence,

$$Az^+ \equiv Az^- \pmod 5.$$

Therefore,

$$h_A(z^+) = h_A(z^-).$$

The two binary inputs are distinct:

$$z^+ \neq z^-.$$

Thus,

$$\left(z^+, z^-\right)$$

is a binary collision for this small negacyclic Ring-SIS compression instance.

5.4 Running Time Analysis

We now compare the running time of the two implemented collision-generation methods. The experiments were performed for fixed modulus $q = 17$ and fixed ring dimension $n = 4$. The number of ring blocks ℓ was varied, and hence the total input dimension

$$m = n\ell$$

also increased linearly with ℓ.

Figures 5.1 and 5.2 show the running time of the full negacyclic Wagner collision generator. The running time increases as the number of blocks grows. For small values of ℓ, the running time is below one second, while for larger values it reaches a few seconds. The curve is not perfectly monotone, which is expected because the algorithm uses randomized grouping and heuristic list truncation. Nevertheless, the general trend shows that the full negacyclic vector attack becomes more expensive as the input dimension increases.

Figures 5.4 and 5.5 show the running time of the cyclic rank-one collapse attack. In this case, the running time is much smaller. Even for the largest tested value of ℓ, the average time remains below 0.03 seconds. This confirms the theoretical analysis: in the cyclic ring

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle,$$

the collision search collapses to a one-dimensional modular relation over $\mathbb{Z}_q$, and therefore the attack is much easier.

The success-rate plots show that both implementations found a collision in all tested cases. Thus, in these experiments, the success rate was

$$1.00$$

for every tested value of ℓ. However, the running times are very different. The cyclic rank-one attack is faster by several orders of magnitude than the full negacyclic Wagner attack. This experimental behavior supports the main theoretical conclusion: the cyclic modulus $x^n - 1$ introduces a structural weakness, while the negacyclic modulus $x^n + 1$ avoids this specific rank-one collapse and forces the attack to work in the full vector space $\mathbb{Z}_q^n$.

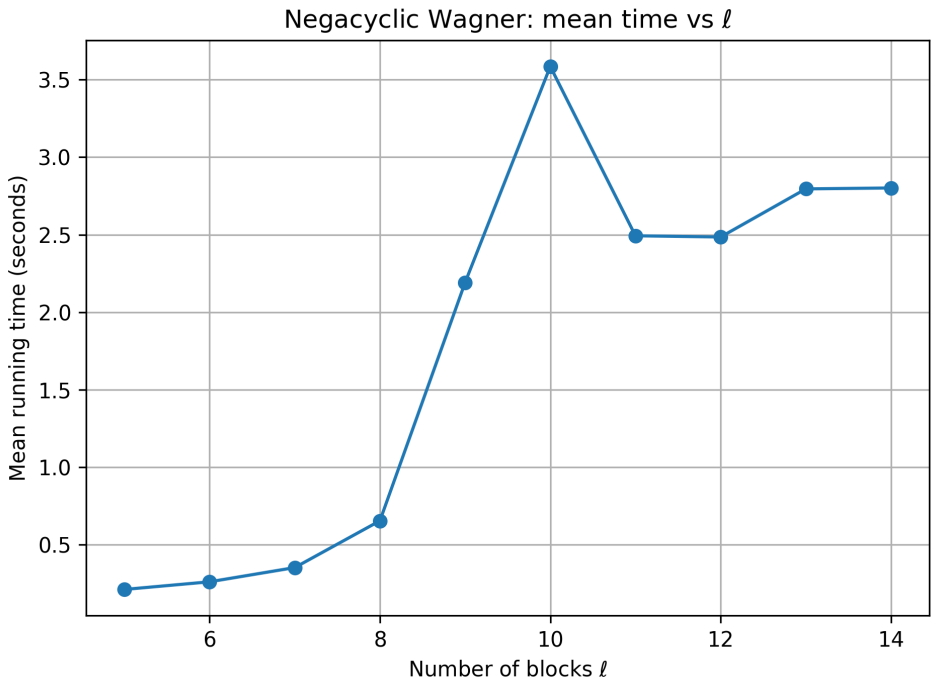

Figure 5.1: Mean running time of the negacyclic Wagner attack as a function of the number of blocks ℓ.

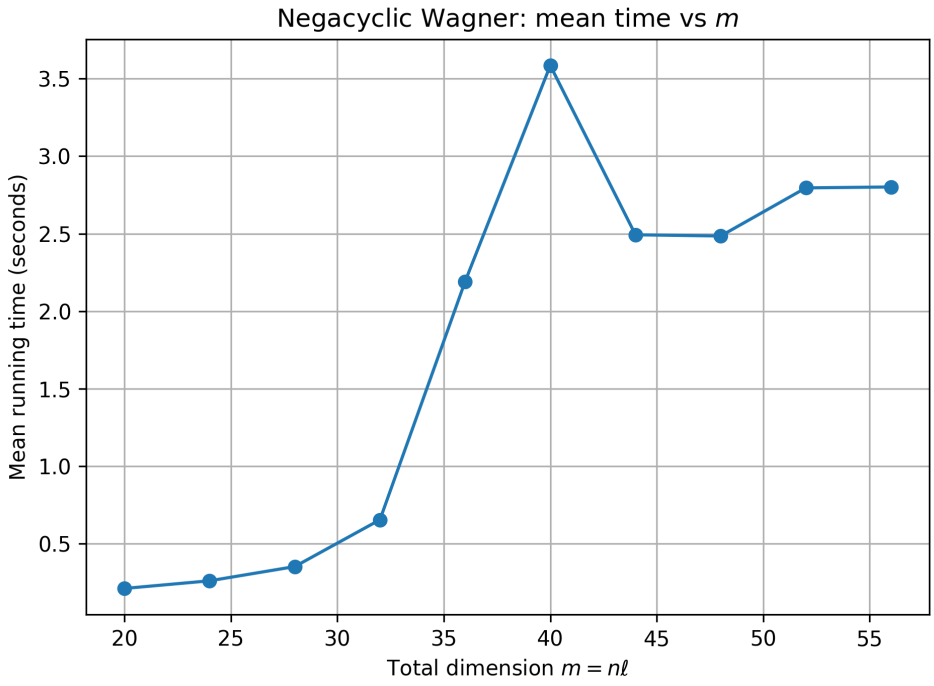

Figure 5.2: Mean running time of the negacyclic Wagner attack as a function of the total dimension $m = n\ell$.

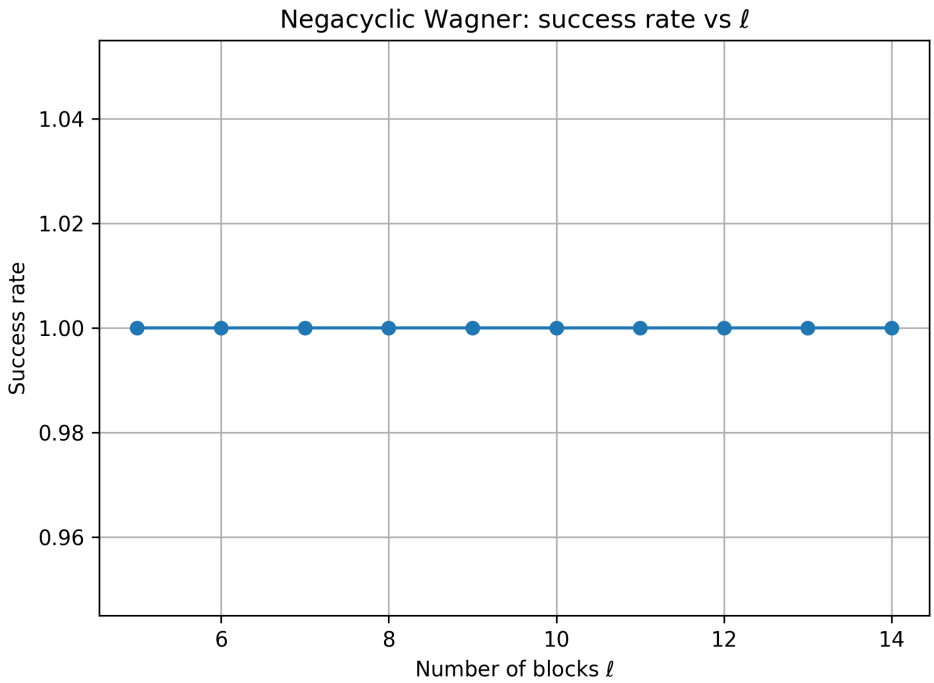

Figure 5.3: Success rate of the negacyclic Wagner attack.

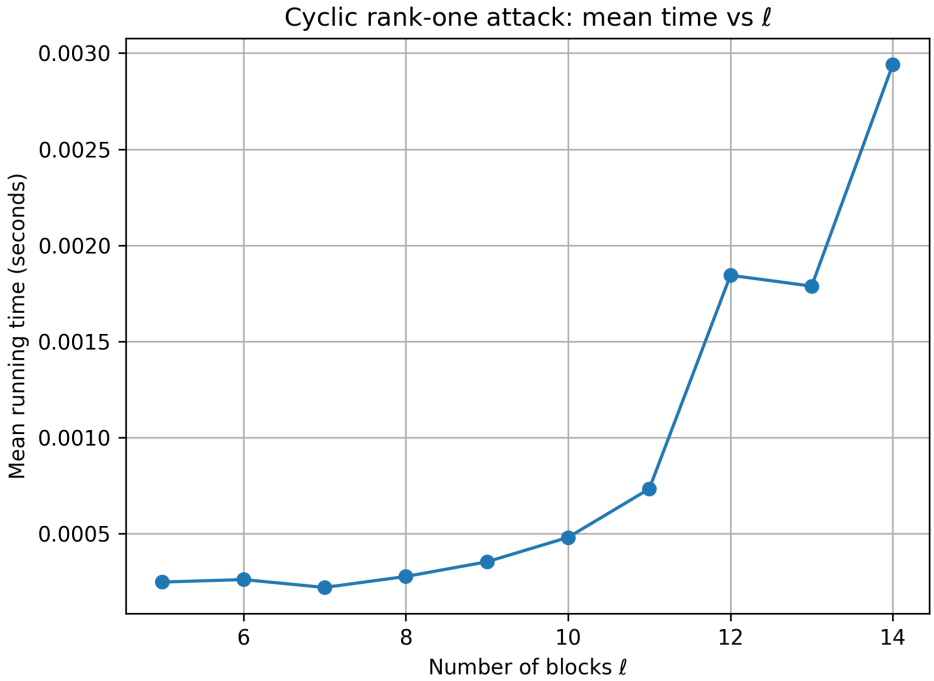

Figure 5.4: Mean running time of the cyclic rank-one collapse attack as a function of the number of blocks ℓ.

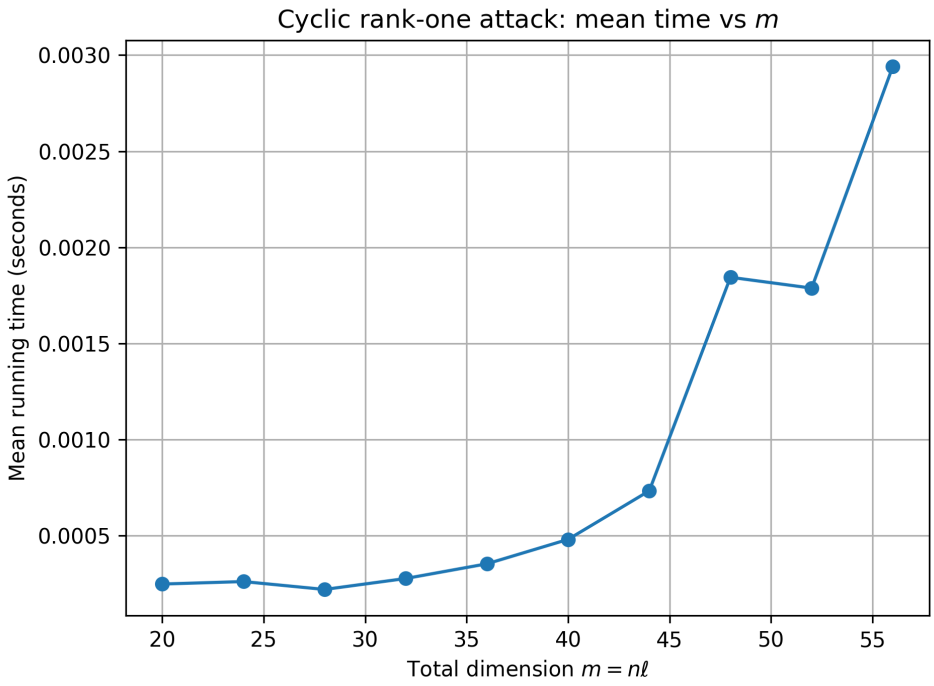

Figure 5.5: Mean running time of the cyclic rank-one collapse attack as a function of the total dimension $m = n\ell$.

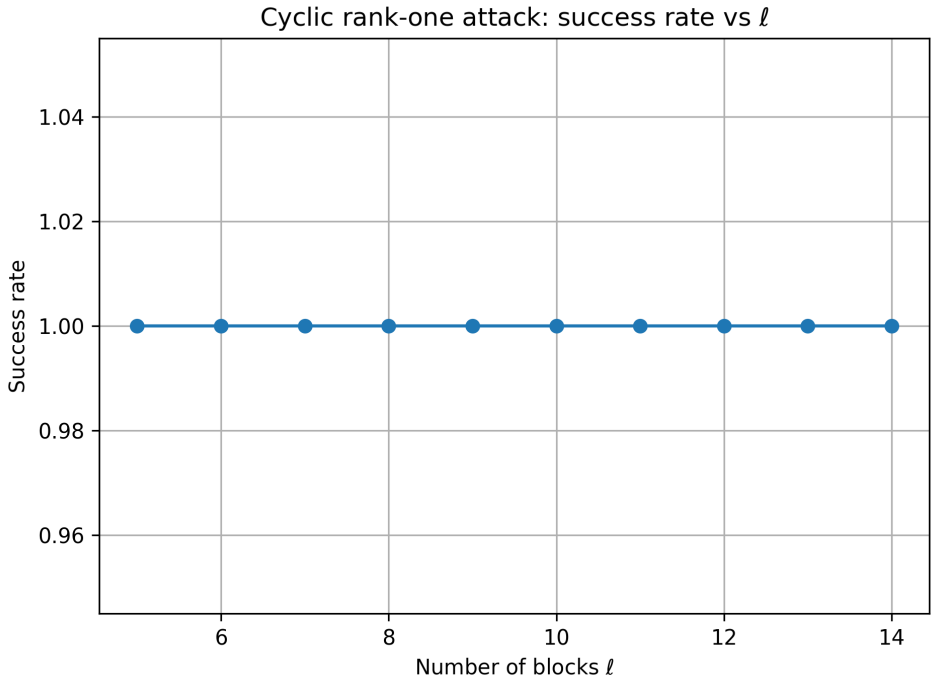

Figure 5.6: Success rate of the cyclic rank-one collapse attack.

5.5 Memory Usage Analysis

In this section, we compare the memory usage of the two collision-generation procedures implemented in Python. The first procedure is the cyclic rank-one collapse attack over

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle,$$

while the second procedure is the full negacyclic Wagner attack over

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle.$$

For both experiments, we used the same fixed parameters

$$q = 17, \qquad n = 4,$$

and varied the number of ring blocks ℓ. Since the total input dimension is

$$m = n\ell,$$

increasing ℓ also increases the total number of columns of the public matrix. The memory usage was measured during the execution of each algorithm, and the reported value is the mean peak memory usage over repeated runs.

5.5.1 Cyclic Rank-One Collapse Attack

Figure 5.7 shows the memory usage of the cyclic rank-one collapse attack as a function of the number of blocks ℓ.

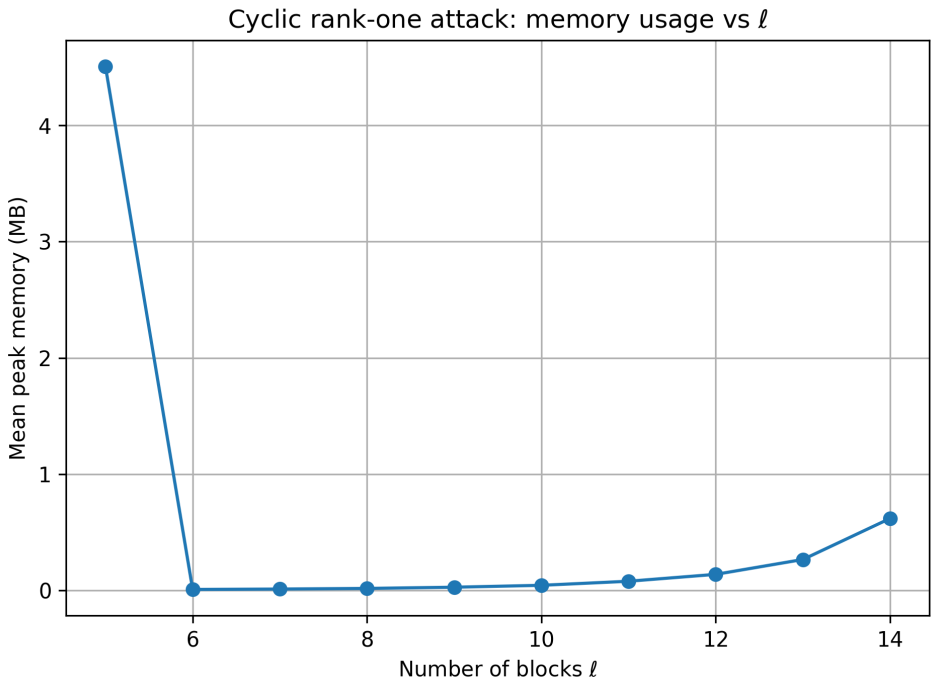

Figure 5.7: Mean peak memory usage of the cyclic rank-one attack as a function of ℓ.

Figure 5.8 shows the same memory measurements as a function of the total dimension $m = n\ell$.

The cyclic attack uses the rank-one collapse to reduce the original Ring-SIS relation to a one-dimensional modular relation over $\mathbb{Z}_q$. Therefore, the algorithm does not need to store large vector-valued Wagner lists in $\mathbb{Z}_q^n$. This explains why the memory usage remains relatively small for most tested values of ℓ.

The first measured point is higher than the following points. This is most likely caused by initialization overhead in Python, NumPy, and the memory measurement tool. After this initial point, the memory usage grows gradually as ℓ increases.

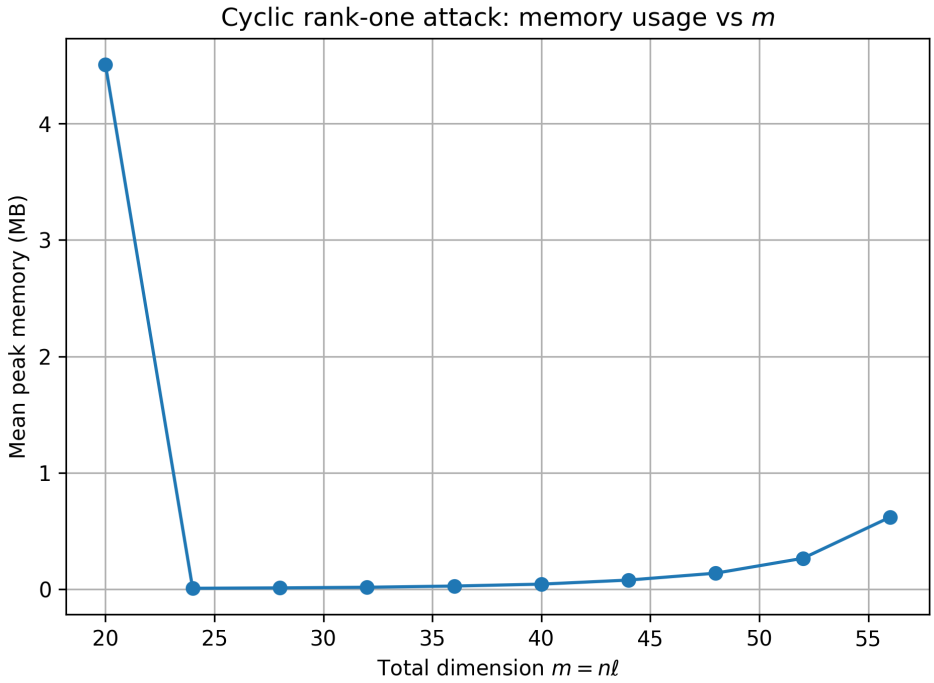

Figure 5.8: Mean peak memory usage of the cyclic rank-one attack as a function of $m = n\ell$.

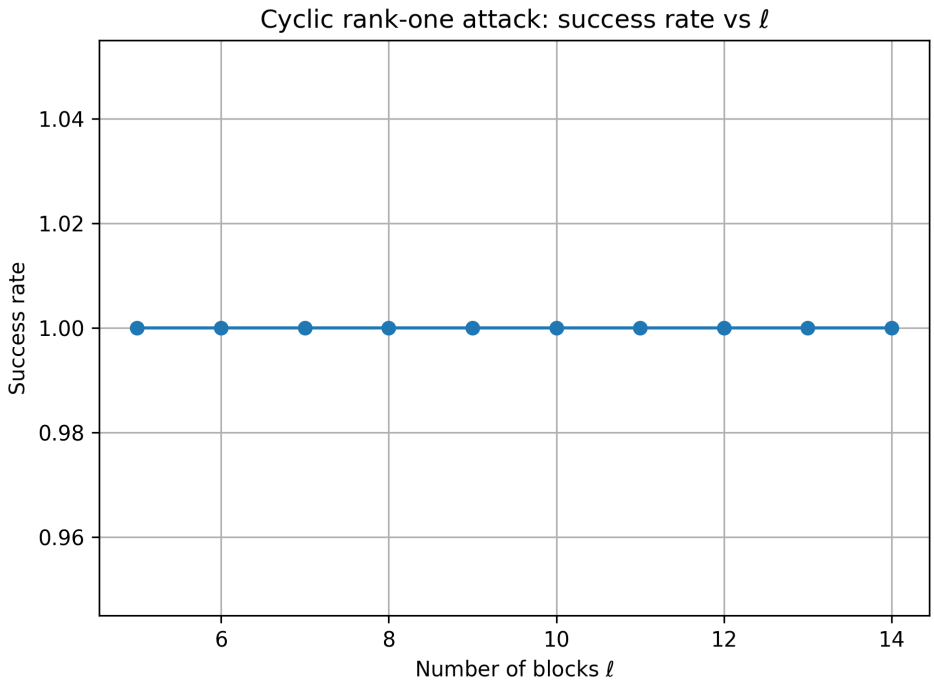

Figure 5.9: Success rate of the cyclic rank-one attack as a function of ℓ.

The success rate of the cyclic experiment is shown in Figure 5.9.

In the tested parameter range, the cyclic rank-one attack succeeds for all tested values of ℓ. This agrees with the theoretical analysis: the cyclic modulus $x^n - 1$ admits the evaluation-at-one collapse, and the collision search is reduced to a much simpler scalar modular problem.

5.5.2 Negacyclic Wagner Attack

We now consider the negacyclic Wagner attack over

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle.$$

In this case, the rank-one collapse is not available. The algorithm must work directly with vector sums in $\mathbb{Z}_q^n$, and therefore it stores vector-valued partial-sum lists.

Figure 5.10 shows the memory usage of the negacyclic Wagner attack as a function of ℓ.

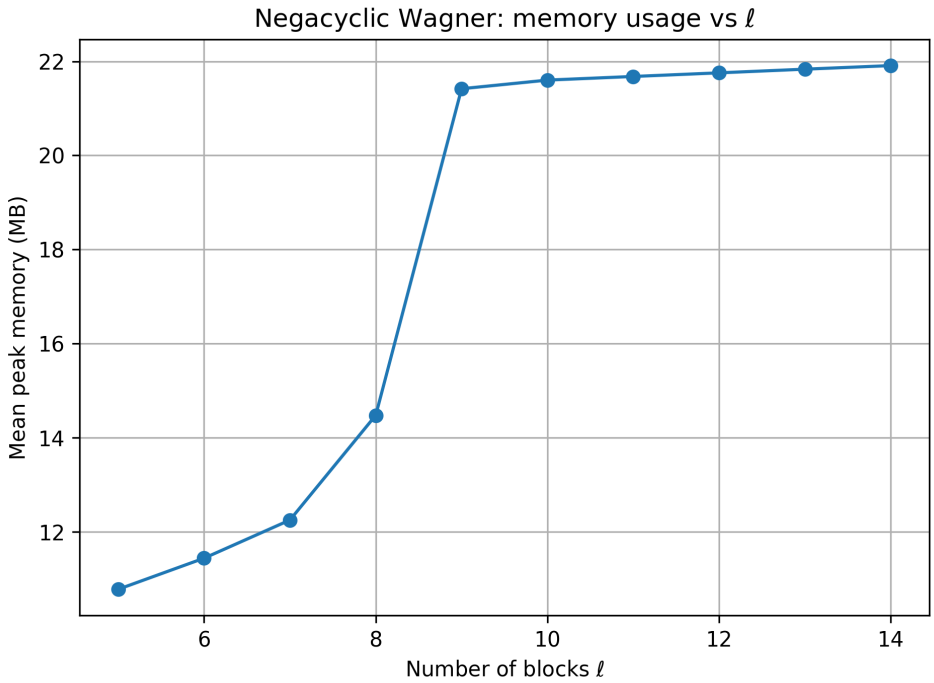

Figure 5.10: Mean peak memory usage of the negacyclic Wagner attack as a function of ℓ.

Figure 5.11 shows the same memory measurements as a function of the total dimension $m = n\ell$.

The negacyclic Wagner attack requires substantially more memory than the cyclic rank-one attack. This is expected, because the algorithm constructs and merges lists of vectors in $\mathbb{Z}_q^n$, rather than lists of scalar residues in $\mathbb{Z}_q$. As ℓ increases, the number of possible ternary combinations also increases, and the algorithm needs to store more candidate partial sums.

The memory usage increases sharply around $\ell = 9$. After that point, the growth becomes slower and almost flat. This behaviour is consistent with the implementation, because the intermediate lists are truncated by the parameter $L_{\max}$. Once this bound is reached, the memory usage cannot continue to grow freely with the number of candidates.

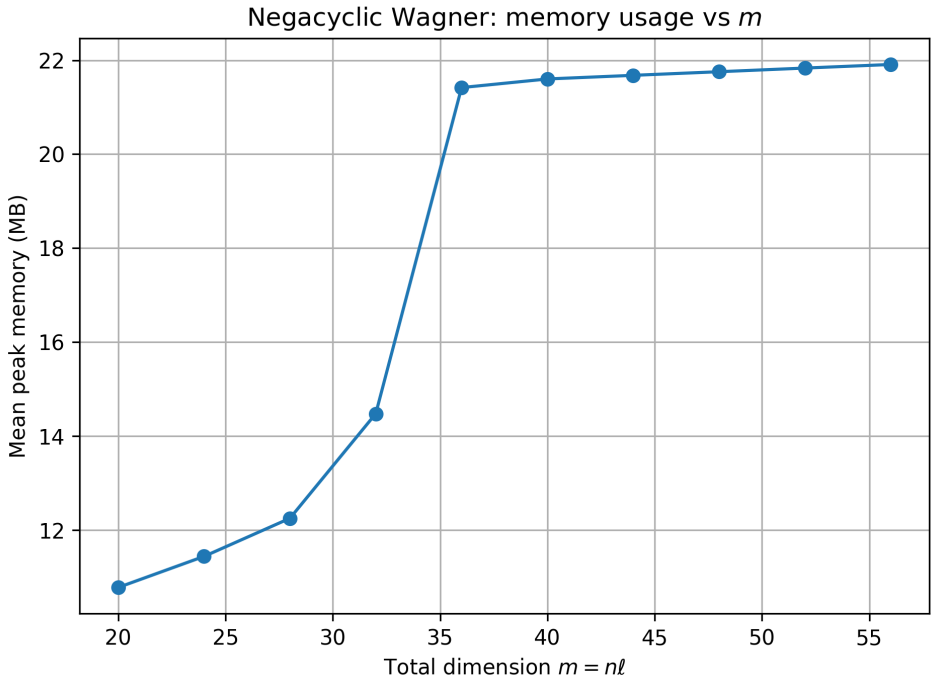

Figure 5.11: Mean peak memory usage of the negacyclic Wagner attack as a function of $m = n\ell$.

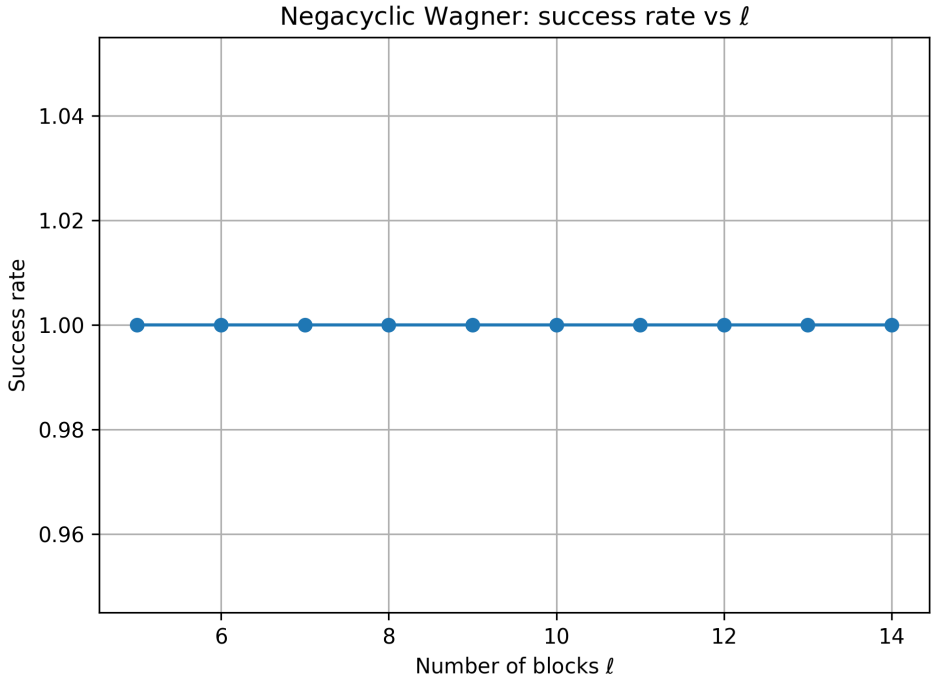

Figure 5.12: Success rate of the negacyclic Wagner attack as a function of ℓ.

The success rate of the negacyclic experiment is shown in Figure 5.12.

In the tested parameter range, the negacyclic Wagner attack also succeeds for all tested values of ℓ. However, this should not be interpreted as evidence of a structural weakness comparable to the cyclic case. These are small experimental parameters, and the search is performed by a heuristic Wagner-style algorithm. The important distinction is that the negacyclic attack works in the full vector space $\mathbb{Z}_q^n$, while the cyclic attack collapses to a scalar problem.

Overall, the experiments show a clear difference between the two settings. The cyclic rank-one attack has very small memory requirements because of the dimension collapse caused by the factor $x - 1$. In contrast, the negacyclic Wagner attack requires significantly more memory, because it must store and merge vector-valued partial sums in the full space $\mathbb{Z}_q^n$.

5.6 Comparison of the Cyclic and Negacyclic Cases

The experiments confirm that the two quotient rings lead to qualitatively different collision-search problems. In the cyclic case,

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle,$$

the factorization

$$x^n - 1 = (x - 1)(1 + x + \cdots + x^{n-1})$$

creates the all-ones element $\widetilde{e}(x)$, which generates a rank-one invariant submodule. Hence the Ring-SIS relation collapses from a vector problem over $\mathbb{Z}_q^n$ to the scalar congruence

$$\sum_{i=1}^{\ell} \varepsilon_i a_i(1) \equiv 0 \pmod{q}.$$

This explains both the very small running times in Figures 5.4–5.5 and the low memory usage in Figures 5.7–5.8: the algorithm stores and compares scalar residues rather than vector-valued partial sums.

In the negacyclic case,

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle,$$

with n a power of two, this evaluation-at-one collapse is not available. The attack must search directly for

$$z \in \{-1, 0, 1\}^m \qquad \text{such that} \qquad Az \equiv 0 \pmod{q}$$

inside the full space $\mathbb{Z}_q^n$. This makes the negacyclic Wagner attack substantially more expensive in both time and memory, as shown in Figures 5.1–5.2 and Figures 5.10–5.11. The higher cost is expected, because the algorithm must construct, filter, and merge lists of vector-valued partial sums.

The success-rate plots should not be interpreted as a direct security measure. Their role is mainly diagnostic: they indicate whether the tested parameters are large enough for the heuristic non-starvation and final matching conditions to be satisfied. If the success rate were low, this would suggest that the generated lists are too small, or that the root matching condition is not being met with sufficient probability. For this reason, the experiments are deliberately kept in a small parameter range, where the algorithms can run reliably and the comparison remains visible.

Overall, the cyclic attack succeeds because of an algebraic collapse caused by the factor $x - 1$. The negacyclic attack, in contrast, does not exploit such a collapse; it succeeds only by carrying out a heuristic Wagner-style search in the full vector space. This supports the main conclusion of the thesis: the cyclic modulus $x^n - 1$ is structurally unsuitable for collision-resistant Ring-SIS hashing, while the negacyclic modulus $x^n + 1$ avoids this particular rank-one vulnerability.

6 Discussion

The main message of this thesis is that, in lattice-based hashing, algebraic structure is not a secondary implementation detail. It is part of the security assumption itself. The same Ring-SIS relation

$$Az \equiv 0 \pmod{q}$$

can lead to very different cryptanalytic behaviour depending on the polynomial ring in which it is instantiated.

The cyclic ring

$$\mathbb{Z}_q[x]/\langle x^n - 1\rangle$$

illustrates this point clearly. Although it supports efficient cyclic convolution, its modulus contains the factor $x - 1$. This gives rise to the all-ones polynomial

$$\widetilde{e}(x) = 1 + x + \cdots + x^{n-1},$$

and hence to a rank-one invariant submodule. The consequence is not merely a small loss of security. It is a structural collapse: a high-dimensional Ring-SIS collision problem over $\mathbb{Z}_q^n$ becomes a one-dimensional modular relation over $\mathbb{Z}_q$. In this setting, the attacker no longer has to solve the full lattice problem. The ring itself exposes a much easier problem.

This explains why the cyclic construction is unsuitable for collision-resistant Ring-SIS hashing. The weakness does not come from poor parameters or from an inefficient implementation. It comes from the algebraic form of the modulus. Once the factor $x - 1$ is present, the compression function admits a natural low-dimensional attack surface.

The negacyclic ring

$$\mathbb{Z}_q[x]/\langle x^n + 1\rangle$$

does not suffer from this particular weakness when n is a power of two. The evaluation-at-one collapse is no longer available, and the all-ones direction no longer produces the same rank-one reduction. This is the key reason why SWIFFT-style constructions use the negacyclic cyclotomic setting. The modulus $x^n + 1$ preserves efficient arithmetic through FFT/NTT methods, but avoids the specific zero-divisor structure that makes the cyclic case insecure [5].

The experiments support this theoretical distinction. In the cyclic case, the attack is fast because it searches only for a scalar modular relation. In the negacyclic case, the Wagner-style method must work with full vector sums in $\mathbb{Z}_q^n$. The difference in running time and memory usage is therefore not accidental. It reflects the difference between an algebraic collapse and a genuine high-dimensional search problem.

This also clarifies the role of Wagner's generalized birthday method in the analysis. Wagner's algorithm is not simply a faster way to search for collisions. It provides a combinatorial lens through which the Ring-SIS kernel can be tested. In the cyclic case, this lens immediately exposes the scalar weakness. In the negacyclic case, it becomes a full vector-valued k-sum search. Thus, Wagner's method helps separate weak algebraic structure from genuinely harder high-dimensional structure [2].

At the same time, Wagner's method is only one side of the cryptanalytic picture. The other side is geometric. The kernel

$$\Lambda_q^{\perp}(A) = \{z \in \mathbb{Z}^m : Az \equiv 0 \pmod{q}\}$$

is a q-ary lattice, so collision search can also be studied through lattice basis reduction. Algorithms such as LLL and BKZ attempt to find short vectors in this kernel. If such a vector is ternary, then it gives a true binary collision. If it is short but not ternary, then it gives only a pseudo-collision. This distinction is important: geometric reduction may reveal weakness in the lattice, but it does not automatically break the binary compression function.

The natural next step is therefore a direct comparison between the two attack families. Wagner's method measures the strength of the construction against combinatorial k-sum attacks. LLL and BKZ measure its resistance against geometric short-vector attacks. The central concrete-security question is where the crossover occurs: for which dimensions and parameters does the combinatorial approach become more effective than lattice reduction, and where does geometric reduction remain competitive?

This thesis should therefore be understood as a foundation for that broader cryptanalytic program. It establishes the algebraic reason why the cyclic modulus fails, explains why the negacyclic modulus avoids this particular failure, and provides an experimental framework for testing collision-generation methods on scaled-down Ring-SIS instances. This is exactly the kind of analysis needed before studying stronger attacks on practical lattice-based hash functions such as SWIFFT.

The broader lesson is that provable security and concrete cryptanalysis must be read together. Worst-case to average-case reductions explain why Ring-SIS-based hashing can be secure in principle. Concrete attacks such as Wagner, LLL, and BKZ test how much security remains for a specific algebraic choice, parameter set, and implementation. A secure design must survive both levels of analysis.

In conclusion, the cyclic modulus $x^n - 1$ fails because it contains an algebraic shortcut. The negacyclic modulus $x^n + 1$ avoids that shortcut and keeps the attack in the intended high-dimensional space. This does not end the cryptanalysis of SWIFFT-like hashing. Rather, it identifies the correct ground on which that cryptanalysis should continue: the comparison of combinatorial and geometric attacks against the full Ring-SIS structure.

References

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

[2] D. Wagner, "A generalized birthday problem," in *Annual International Cryptology Conference*. Springer, 2002, pp. 288–304.

[3] A. Satriawan, R. Mareta, and H. Lee, "A complete beginner guide to the number theoretic transform (ntt)," *Cryptology ePrint Archive*, 2024.

[4] J. H. Silverman, J. Pipher, and J. Hoffstein, *An introduction to mathematical cryptography*. Springer, 2008, vol. 1.

[5] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, "Swifft: A modest proposal for fft hashing," in *International workshop on fast software encryption*. Springer, 2008, pp. 54–72.

[6] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Designs, Codes and Cryptography*, vol. 75, no. 3, pp. 565–599, 2015.

[7] M. Stamp and R. M. Low, *Applied Cryptanalysis: Breaking Ciphers in the Real World*. John Wiley & Sons, 2007.

[8] N. Stephens-Davidowitz, "Ring-sis and ideal lattices," Lecture notes for Vinod Vaikuntanathan's course 6.876 (Advanced Topics in Cryptography), MIT, Fall 2018, accessed: 2026-03-06. [Online]. Available: https://people.csail.mit.edu/vinodv/6876-Fall2018/RingSISclass.pdf

A Appendix 1 - Algorithms

Algorithm 1 Bowtie-Hash-Join

Require: Two lists

$$L_{\text{left}} = [(x_{\text{val}}, x_{\text{wit}})] \quad \text{and} \quad L_{\text{right}} = [(y_{\text{val}}, y_{\text{wit}})],$$

where $x_{\text{val}}, y_{\text{val}}$ are values and $x_{\text{wit}}, y_{\text{wit}}$ are the corresponding witnesses. Let $L_{\max} \in \mathbb{Z}_{\geq 0}$ be the maximum allowed output size.

Ensure: A list L_{out} containing at most $L_{\max}$ matching elements with merged witnesses.

Initialize an empty hash table H ▷ Used to index values from the left list for fast lookup
Initialize an empty list L_{out} ▷ Output list
for all $(x_{\text{val}}, x_{\text{wit}}) \in L_{\text{left}}$ **do** ▷ Build phase: insert all elements of the left list into the hash table
    if x_{val} is not a key of H **then**
        $H[x_{\text{val}}] \leftarrow [\,]$ ▷ Create a new bucket for this value
    end if
    Append $(x_{\text{val}}, x_{\text{wit}})$ to $H[x_{\text{val}}]$ ▷ Store all entries (handles duplicates correctly)
end for
for all $(y_{\text{val}}, y_{\text{wit}}) \in L_{\text{right}}$ **do** ▷ Probe phase: search for matches in the hash table
    if y_{val} is a key of H **then** ▷ A match is found: $x_{\text{val}} = y_{\text{val}}$
        for all $(x_{\text{val}}, x_{\text{wit}}) \in H[y_{\text{val}}]$ **do**
            if $|L_{\text{out}}| \geq L_{\max}$ **then**
                return L_{out} ▷ Stop if output size limit is reached
            end if
            $v_{\text{new}} \leftarrow y_{\text{val}}$ ▷ Matched value (since $x = y$)
            $w_{\text{new}} \leftarrow \text{Concatenate}(x_{\text{wit}}, y_{\text{wit}})$ ▷ Combine witnesses for backtracking
            Append $(v_{\text{new}}, w_{\text{new}})$ to L_{out} ▷ Store result
        end for
    end if
end for
return L_{out}

Algorithm 2 Bowtie-Interval-Join

Require: Two lists

$$L_{\text{left}} = [(x_{\text{val}}, x_{\text{wit}})] \quad \text{and} \quad L_{\text{right}} = [(y_{\text{val}}, y_{\text{wit}})],$$

where

$$x_{\text{val}}, y_{\text{val}} \in \{0, 1, \ldots, q-1\}$$

are canonical representatives of elements of $\mathbb{Z}_q$, and $x_{\text{wit}}, y_{\text{wit}}$ are the corresponding witnesses. Let $q \geq 2$ be the modulus, let $r \in \mathbb{Z}_{\geq 0}$ be the filtering parameter, and let $L_{\max} \in \mathbb{Z}_{\geq 0}$ be the maximum allowed size of the output list. Assume that

$$R = \left\lfloor \frac{q}{2^{r+1}} \right\rfloor \geq 1.$$

Ensure: A list L_{out} containing at most $L_{\max}$ merged elements $(v_{\text{new}}, w_{\text{new}})$, where

$$v_{\text{new}} = (x_{\text{val}} + y_{\text{val}}) \bmod q$$

and

$$v_{\text{new}} \in [-R, R-1] \pmod q.$$

Initialize an empty list L_{out} ▷ This will store the merged output elements
$R \leftarrow \lfloor q/2^{r+1} \rfloor$ ▷ Radius of the accepted interval around 0 modulo q
Sort L_{right} increasingly by y_{val}, obtaining $\widetilde{L}_{\text{right}}$ ▷ Sorting enables efficient range queries
Let B be the sorted array of values from $\widetilde{L}_{\text{right}}$ ▷ $B[j]$ is the y_{val} of $\widetilde{L}_{\text{right}}[j]$

Binary-search convention:
LowerBound(B, z) returns the first index j such that $B[j] \geq z$.
UpperBound(B, z) returns the first index j such that $B[j] > z$.
Thus, all entries with $a \leq y_{\text{val}} \leq b$ are exactly the entries with

$$j \in [\text{LowerBound}(B, a), \text{UpperBound}(B, b) - 1].$$

for all $(x_{\text{val}}, x_{\text{wit}}) \in L_{\text{left}}$ **do** ▷ For this x_{val}, compute the admissible interval for y_{val}
    $\alpha \leftarrow (-R - x_{\text{val}}) \bmod q$ ▷ Lower endpoint of the admissible interval for y_{val}
    $\beta \leftarrow (R - 1 - x_{\text{val}}) \bmod q$ ▷ Upper endpoint of the admissible interval for y_{val}
    if $\alpha \leq \beta$ **then** ▷ The admissible interval does not wrap around modulo q
        $s \leftarrow \text{LowerBound}(B, \alpha)$
        $t \leftarrow \text{UpperBound}(B, \beta)$
        for $j = s$ to $t - 1$ **do**
            if $|L_{\text{out}}| \geq L_{\max}$ **then**
                return L_{out} ▷ Stop before exceeding the output bound
            end if
            $(y_{\text{val}}, y_{\text{wit}}) \leftarrow \widetilde{L}_{\text{right}}[j]$
            $v_{\text{new}} \leftarrow (x_{\text{val}} + y_{\text{val}}) \bmod q$ ▷ This lies in $[-R, R-1] \pmod q$
            $w_{\text{new}} \leftarrow \text{Concatenate}(x_{\text{wit}}, y_{\text{wit}})$ ▷ Merge witnesses to preserve provenance
            Append $(v_{\text{new}}, w_{\text{new}})$ to L_{out}
        end for
    else ▷ The admissible interval wraps around modulo q
        $s_1 \leftarrow \text{LowerBound}(B, \alpha)$
        $t_1 \leftarrow \text{UpperBound}(B, q - 1)$
        for $j = s_1$ to $t_1 - 1$ **do**
            if $|L_{\text{out}}| \geq L_{\max}$ **then**
                return L_{out} ▷ Stop before exceeding the output bound
            end if
            $(y_{\text{val}}, y_{\text{wit}}) \leftarrow \widetilde{L}_{\text{right}}[j]$
            $v_{\text{new}} \leftarrow (x_{\text{val}} + y_{\text{val}}) \bmod q$ ▷ This lies in the wrapped part of $[-R, R-1] \pmod q$
            $w_{\text{new}} \leftarrow \text{Concatenate}(x_{\text{wit}}, y_{\text{wit}})$ ▷ Merge witnesses to preserve provenance
            Append $(v_{\text{new}}, w_{\text{new}})$ to L_{out}
        end for
        $s_2 \leftarrow \text{LowerBound}(B, 0)$

$t_2 \leftarrow \text{UpperBound}(B, \beta)$
for $j = s_2$ to $t_2 - 1$ **do**
if $|L_{\text{out}}| \geq L_{\max}$ **then**
return L_{out} ▷ Stop before exceeding the output bound
end if
$(y_{\text{val}}, y_{\text{wit}}) \leftarrow \widetilde{L}_{\text{right}}[j]$
$v_{\text{new}} \leftarrow (x_{\text{val}} + y_{\text{val}}) \bmod q$ ▷ This lies in the wrapped part of $[-R, R-1] \pmod q$
$w_{\text{new}} \leftarrow \text{Concatenate}(x_{\text{wit}}, y_{\text{wit}})$ ▷ Merge witnesses to preserve provenance
Append $(v_{\text{new}}, w_{\text{new}})$ to L_{out}
end for
end if
end for
return L_{out}

Algorithm 3 GENERALIZED-K-TREE-MODULAR-SOLVER

Require: A sequence of k input lists

$$\mathcal{L} = [L_0, L_1, \ldots, L_{k-1}],$$

where $k = 2^D$ for some integer $D \geq 1$. Each list element has the form

$$(v_{\text{val}}, v_{\text{wit}}),$$

where

$$v_{\text{val}} \in \{0, 1, \ldots, q-1\}$$

is a canonical representative modulo q, and v_{wit} is the corresponding witness. Let $q \geq 2$ be the modulus. Let $L_{\max} \in \mathbb{Z}_{\geq 0}$ be the maximum allowed size of every intermediate list. Let $S_{\max} \in \mathbb{Z}_{\geq 0}$ be the maximum number of final solutions returned.

Ensure: A list Sol of elements $(0, w_{\text{root}})$, where each w_{root} identifies one element

$$x_i \in L_i \qquad (0 \leq i < k)$$

such that

$$x_0 + x_1 + \cdots + x_{k-1} \equiv 0 \pmod q.$$

Subroutines.
This algorithm uses

BOWTIE-INTERVAL-JOIN

from Algorithm 2 at every non-root internal node, and

BOWTIE-HASH-JOIN

from Algorithm 1 at the root.

Conventions.
All arrays and lists are indexed from 0.
$\text{Canon}_q(z)$ denotes the canonical representative of z mod q in $\{0, 1, \ldots, q-1\}$.
Witnesses are stored as back-pointers to child witnesses.
The non-root internal nodes use interval filtering.
The root uses exact modular complement matching.

Input validation.

if $k < 2$ **then**
  return [ ]
end if
if k is not a power of two **then**
  return [ ] ▷ This core version assumes $k = 2^D$
end if
for $i = 0$ to $k - 1$ **do**
  if $L_i = [\,]$ **then**
    return [ ] ▷ No solution can use one element from every list
  end if
end for

Initial parameter setup.

$$D \leftarrow \log_2 k$$

▷ Depth of the complete binary tree

$$n_q \leftarrow \lceil \log_2 q \rceil$$

▷ Bit-length scale of the modulus

$$r_{\max} \leftarrow \max\{0, \lfloor \log_2 q \rfloor - 1\}$$

▷ Ensures that the interval radius in Algorithm 2 is nonzero

$$\text{Current} \leftarrow [L_0, L_1, \ldots, L_{k-1}]$$

▷ Current level of the binary tree

Process all non-root internal levels.

for $h = 1$ to $D - 1$ **do**

$$r_h \leftarrow \min\left\{ \left\lfloor \frac{h\, n_q}{D+1} \right\rfloor, r_{\max} \right\}$$

▷ Height-dependent modular filtering parameter

$$\text{Next} \leftarrow [\,]$$

▷ Lists produced at the next tree level

  for $t = 0$ to $\text{Length}(\text{Current})/2 - 1$ **do**

$$L_{\text{left}} \leftarrow \text{Current}[2t]$$

$$L_{\text{right}} \leftarrow \text{Current}[2t+1]$$

$$L_{\text{merged}} \leftarrow \textsc{Bowtie-Interval-Join}(L_{\text{left}}, L_{\text{right}}, q, r_h, L_{\max})$$

▷ Algorithm 2

    Append L_{merged} to Next
  end for

$$\text{Current} \leftarrow \text{Next}$$

▷ Move one level upward in the tree

end for

Process the root by exact modular complement matching.

$$L_{\text{root,left}} \leftarrow \text{Current}[0]$$

$$L_{\text{root,right}} \leftarrow \text{Current}[1]$$

$$L^{-}_{\text{root,right}} \leftarrow [\,]$$

▷ Negated right root list

for all $(y_{\text{val}}, y_{\text{wit}}) \in L_{\text{root,right}}$ **do**

$$\overline{y}_{\text{val}} \leftarrow \text{Canon}_q(-y_{\text{val}})$$

▷ Need $x_{\text{val}} = -y_{\text{val}} \pmod q$

Append

$$(\overline{y}_{\text{val}}, y_{\text{wit}})$$

to $L^{-}_{\text{root,right}}$

end for

$$J \leftarrow \textsc{Bowtie-Hash-Join}(L_{\text{root,left}}, L^{-}_{\text{root,right}}, S_{\max})$$

▷ Algorithm 1

$$\text{Sol} \leftarrow [\,]$$

▷ Final solutions

for all $(z_{\text{val}}, z_{\text{wit}}) \in J$ **do**
    if $|\text{Sol}| \geq S_{\max}$ **then**
        return Sol
    end if

$$w_{\text{root}} \leftarrow z_{\text{wit}}$$

▷ Back-pointers to the two root children

Append

$$(0, w_{\text{root}})$$

to Sol ▷ The complete modular sum is exactly 0

end for

return Sol

Algorithm 4 Rank-One-Collapse-Collision-Generator

Require: A modulus $q \geq 2$, integers n, m, ℓ with

$$m = \ell n,$$

and public block vectors

$$a_1, a_2, \ldots, a_\ell \in \mathbb{Z}_q^n,$$

where

$$a_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,n-1}) \qquad (1 \leq i \leq \ell).$$

These blocks define the public matrix

$$A = [\text{Rot}(a_1) \mid \text{Rot}(a_2) \mid \cdots \mid \text{Rot}(a_\ell)] \in \mathbb{Z}_q^{n \times m}.$$

Let $L_{\max} \in \mathbb{Z}_{\geq 0}$ be the maximum allowed size of every intermediate list used by the modular solver, and let $S_{\max} \in \mathbb{Z}_{\geq 0}$ be the maximum number of modular relations returned per attempt.

Ensure: Either $\perp$, if no suitable relation is found, or a collision pair

$$(z^+, z^-) \in \{0,1\}^m \times \{0,1\}^m$$

such that

$$Az^+ \equiv Az^- \pmod{q}.$$

Subroutines.
This algorithm uses a modular zero-sum solver

MODULAR-ZERO-SUM-SOLVER

to find one element from each input list whose sum is $0 \pmod{q}$. This subroutine may be implemented either by the filtered

GENERALIZED-K-TREE-MODULAR-SOLVER

or by an exact modular merge solver.

Input validation.
if $q < 2$ **then**
    return $\perp$
end if
if $n < 1$ or $\ell < 1$ **then**
    return $\perp$
end if
if $m \neq \ell n$ **then**
    return $\perp$
end if
for $i = 1$ to ℓ **do**
    if Length$(a_i) \neq n$ **then**
        return $\perp$
    end if
    for $j = 0$ to $n - 1$ **do**
        if $a_{i,j} \notin \{0, 1, \ldots, q-1\}$ **then**
            return $\perp$
        end if
    end for
end for

Step 1: Scalar mapping and dimensional reduction.
for $i = 1$ to ℓ **do**

$$\alpha_i \leftarrow \left(\sum_{j=0}^{n-1} a_{i,j} \right) \bmod q$$

▷ Evaluate $a_i(x)$ at $x = 1$

end for

Step 2: Build a non-trivial ternary modular relation.

$$\varepsilon \leftarrow \perp$$

for $p = 1$ to ℓ **do** ▷ p is forced to be the first nonzero ternary position

$$\mathcal{T} \leftarrow [\,]$$

▷ Lists for the modular zero-sum solver

for $i = 1$ to ℓ **do**
  if $i < p$ **then**

$$T_i \leftarrow [(0, (i, 0))]$$

▷ Positions before p are forced to zero

  else if $i = p$ **then**

$$T_i \leftarrow \left[(\alpha_i, (i, +1)), \left(\mathrm{Canon}_q(-\alpha_i), (i, -1)\right)\right]$$

▷ Position p is forced to be nonzero

  else

$$T_i \leftarrow \left[(0, (i, 0)), (\alpha_i, (i, +1)), \left(\mathrm{Canon}_q(-\alpha_i), (i, -1)\right)\right]$$

▷ Later positions may be $0, +1$, or -1

  end if
  Append T_i to $\mathcal{T}$
end for

$$\mathcal{T}' \leftarrow \mathcal{T}$$

while Length($\mathcal{T}'$) is not a power of two **do**
  Append

$$[(0, (\mathrm{dummy}, 0))]$$

to $\mathcal{T}'$ ▷ Padding by zero lists for a power-of-two solver

end while

$$\mathrm{Rel} \leftarrow \textsc{Modular-Zero-Sum-Solver}(\mathcal{T}', q, L_{\max}, S_{\max})$$

if Rel $\neq [\,]$ **then**
  Let w_{root} be the witness of the first relation in Rel
  Recover from w_{root} the chosen signs

$$\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_\ell \in \{-1, 0, 1\}$$

using the witnesses (i, ε_i)

$$\varepsilon \leftarrow (\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_\ell)$$

  break
end if
end for
if $\varepsilon = \perp$ **then**
  return $\perp$
end if

Step 3: Polynomial lifting.

$$z \leftarrow [\,]$$

▷ This will become a vector in $\{-1, 0, 1\}^m$

for $i = 1$ to ℓ **do**

for $j = 0$ to $n - 1$ **do**
    Append ε_i to z
end for
end for

Step 4: Binary decomposition and collision extraction.

$$z^+ \leftarrow [\,], \qquad z^- \leftarrow [\,]$$

for $t = 0$ to $m - 1$ **do**
if $z_t = 1$ **then**
    Append 1 to z^+
    Append 0 to z^-
else if $z_t = -1$ **then**
    Append 0 to z^+
    Append 1 to z^-
else
    Append 0 to z^+
    Append 0 to z^-
end if
end for
return (z^+, z^-)

Algorithm 5 Negacyclic-Wagner-Collision-Generator

Require: A modulus $q \geq 2$, integers n, m, ℓ with

$$m = \ell n,$$

and public block vectors

$$a_1, a_2, \ldots, a_\ell \in \mathbb{Z}_q^n,$$

where

$$a_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,n-1}) \qquad (1 \leq i \leq \ell).$$

These blocks define the negacyclic public matrix

$$A = \left[\mathrm{Rot}_{\mathrm{neg}}(a_1) \mid \mathrm{Rot}_{\mathrm{neg}}(a_2) \mid \cdots \mid \mathrm{Rot}_{\mathrm{neg}}(a_\ell)\right] \in \mathbb{Z}_q^{n \times m},$$

where $\mathrm{Rot}_{\mathrm{neg}}(a_i)$ is generated by multiplication by x in

$$R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle.$$

Let $k = 2^D$ be the number of Wagner lists, for some integer $D \geq 1$, with

$$k \mid m.$$

Let $L_{\max} \in \mathbb{Z}_{\geq 1}$ be the maximum allowed size of every intermediate list.

Ensure: Either $\perp$, if no suitable relation is found, or a collision pair

$$(z^+, z^-) \in \{0,1\}^m \times \{0,1\}^m$$

such that

$$Az^+ \equiv Az^- \pmod{q}.$$

Conventions.
All lists are indexed from 0.
All vector additions are performed in $\mathbb{Z}_q^n$.

A list element has the form

$$(S, w),$$

where

$$S \in \mathbb{Z}_q^n$$

is a partial sum and w is the corresponding ternary witness.

Input validation.

if $q < 2$ **then**
    return $\perp$
end if
if $n < 1$ or $\ell < 1$ **then**
    return $\perp$
end if
if $m \neq \ell n$ **then**
    return $\perp$
end if
if $k < 2$ or k is not a power of two **then**
    return $\perp$
end if
if $k \nmid m$ **then**
    return $\perp$
end if
if $L_{\max} < 1$ **then**
    return $\perp$
end if
for $i = 1$ to ℓ **do**
    if Length$(a_i) \neq n$ **then**
        return $\perp$
    end if
    for $j = 0$ to $n - 1$ **do**
        if $a_{i,j} \notin \{0, 1, \ldots, q - 1\}$ **then**
            return $\perp$
        end if
    end for
end for

Step 1: Extract the negacyclic column vectors.

Let

$$C = (c_0, c_1, \ldots, c_{m-1})$$

be the ordered list of the columns of A, where

$$c_t \in \mathbb{Z}_q^n \qquad (0 \leq t < m).$$

Step 2: Split the columns into Wagner groups.

$s \leftarrow m/k$ ▷ Number of columns per Wagner group
Let σ be a random permutation of $\{0, 1, \ldots, m - 1\}$
for $r = 0$ to $k - 1$ **do**
    Define

$$G_r \leftarrow \{\sigma(rs), \sigma(rs + 1), \ldots, \sigma((r + 1)s - 1)\}.$$

end for

Step 3: Build the initial partial-sum lists.

$$\mathcal{L} \leftarrow [\,]$$

for $r = 0$ to $k - 1$ **do**

$$L_r \leftarrow [\,]$$

  for all $\varepsilon = (\varepsilon_t)_{t \in G_r} \in \{-1, 0, 1\}^b$ **do**

$$S \leftarrow \sum_{t \in G_r} \varepsilon_t c_t \pmod{q}$$

$$w \leftarrow \big((t, \varepsilon_t) : t \in G_r\big)$$

▷ Witness storing the local ternary choices

    Append

$$(S, w)$$

to L_r

    if $|L_r| \geq L_{\max}$ **then**

      break ▷ Keep the list size bounded

    end if

  end for

  Append L_r to $\mathcal{L}$

end for

Step 4: Wagner hierarchical vector merging.

for $h = 1$ to $D - 1$ **do**

  Choose a set of coefficient positions

$$I_h \subseteq \{0, 1, \ldots, n - 1\}$$

to be forced to zero at level h.

$$\mathcal{L}_{\text{new}} \leftarrow [\,]$$

  for $r = 0$ to $\text{Length}(\mathcal{L})/2 - 1$ **do**

$$L_{\text{left}} \leftarrow \mathcal{L}[2r]$$

$$L_{\text{right}} \leftarrow \mathcal{L}[2r + 1]$$

    Initialize an empty hash table H

    for all $(S_1, w_1) \in L_{\text{left}}$ **do**

$$\kappa_1 \leftarrow (S_1[j])_{j \in I_h}$$

▷ Projection of S_1 on the filtered coordinates

      Append (S_1, w_1) to $H[\kappa_1]$

    end for

$$L_{\text{merged}} \leftarrow [\,]$$

    for all $(S_2, w_2) \in L_{\text{right}}$ **do**

61: $\kappa_2 \leftarrow (-S_2[j])_{j \in I_h} \pmod q$

▷ Complementary key

62: **if** κ_2 is a key of H **then**
63: **for all** $(S_1, w_1) \in H[\kappa_2]$ **do**
64: $S \leftarrow S_1 + S_2 \pmod q$
65: $w \leftarrow \text{Concatenate}(w_1, w_2)$
66: Append (S, w) to L_{merged}
67: **if** $|L_{\text{merged}}| \geq L_{\max}$ **then**
68: **break**
69: **end if**
70: **end for**
71: **end if**
72: **if** $|L_{\text{merged}}| \geq L_{\max}$ **then**
73: **break**
74: **end if**
75: **end for**
76: Append L_{merged} to $\mathcal{L}_{\text{new}}$
77: **end for**
78: $\mathcal{L} \leftarrow \mathcal{L}_{\text{new}}$
79: **end for**

Step 5: Final exact zero matching.

80: **if** $\text{Length}(\mathcal{L}) \neq 2$ **then**
81: **return** $\perp$
82: **end if**
83: $L_{\text{left}} \leftarrow \mathcal{L}[0]$
84: $L_{\text{right}} \leftarrow \mathcal{L}[1]$
85: Initialize an empty hash table H_{root}
86: **for all** $(S_1, w_1) \in L_{\text{left}}$ **do**
87: Append (S_1, w_1) to $H_{\text{root}}[S_1]$
88: **end for**
89: $z \leftarrow \perp$
90: **for all** $(S_2, w_2) \in L_{\text{right}}$ **do**
91: $T \leftarrow -S_2 \pmod q$

▷ Need $S_1 + S_2 \equiv 0 \pmod q$

92: **if** T is a key of H_{root} **then**
93: Let (S_1, w_1) be the first element of $H_{\text{root}}[T]$
94: $w_{\text{root}} \leftarrow \text{Concatenate}(w_1, w_2)$

$z \leftarrow \mathbf{0}_m$

for all $(t, \varepsilon_t) \in w_{\text{root}}$ **do**

$z_t \leftarrow \varepsilon_t$

end for
break
end if
end for
if $z = \perp$ **then**
return $\perp$
end if
if $z = \mathbf{0}_m$ **then**
return $\perp$
end if
if $Az \not\equiv \mathbf{0} \pmod{q}$ **then**
return $\perp$ ▷ Final verification
end if

Step 6: Binary decomposition and collision extraction.

$z^+ \leftarrow [\,], \qquad z^- \leftarrow [\,]$

for $t = 0$ to $m - 1$ **do**
if $z_t = 1$ **then**
Append 1 to z^+
Append 0 to z^-
else if $z_t = -1$ **then**
Append 0 to z^+
Append 1 to z^-
else
Append 0 to z^+
Append 0 to z^-
end if
end for
return (z^+, z^-)

B Appendix 2 - Source Code

```
from __future__ import annotations

from collections.abc import Hashable, Sequence
from dataclasses import dataclass
from typing import TypeVar

import numpy as np
from numba import njit, types
from numba.typed import Dict

import math

```

```
ValueT = TypeVar("ValueT", bound=Hashable)
WitnessT = TypeVar("WitnessT")

MAX_SAFE_INT64_MODULUS = 1 << 62


@njit(cache=True)
def _bowtie_hash_join_int64_core(
    left_values: np.ndarray,
    left_witness_ids: np.ndarray,
    right_values: np.ndarray,
    right_witness_ids: np.ndarray,
    max_output_size: int,
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Run Bowtie-Hash-Join for int64 arrays.

    Args:
        left_values: Values of the left list.
        left_witness_ids: Witness identifiers of the left list.
        right_values: Values of the right list.
        right_witness_ids: Witness identifiers of the right
           list.
        max_output_size: Maximum number of output matches.

    Returns:
        A tuple with three arrays:
        - output values,
        - left witness identifiers,
        - right witness identifiers.

    Notes:
        The witness of each output element is represented by
           the pair
        (left_witness_id, right_witness_id). This preserves all
        backtracking information.
    """
    output_values = np.empty(max_output_size, dtype=np.int64)
    output_left_witness_ids = np.empty(max_output_size,
       dtype=np.int64)
    output_right_witness_ids = np.empty(max_output_size,
       dtype=np.int64)

    if max_output_size == 0:
        return (
            output_values[:0],
            output_left_witness_ids[:0],
            output_right_witness_ids[:0],
        )

    heads = Dict.empty(key_type=types.int64,
       value_type=types.int64)
```

```
    tails = Dict.empty(key_type=types.int64,
       value_type=types.int64)

    next_left_index = np.full(left_values.shape[0], -1,
       dtype=np.int64)

    for left_index in range(left_values.shape[0]):
        x_val = left_values[left_index]

        if x_val in heads:
            previous_tail = tails[x_val]
            next_left_index[previous_tail] = left_index
            tails[x_val] = left_index
        else:
            heads[x_val] = left_index
            tails[x_val] = left_index

    output_size = 0

    for right_index in range(right_values.shape[0]):
        y_val = right_values[right_index]

        if y_val not in heads:
            continue

        left_index = heads[y_val]

        while left_index != -1:
            if output_size >= max_output_size:
                return (
                    output_values[:output_size],
                    output_left_witness_ids[:output_size],
                    output_right_witness_ids[:output_size],
                )

            output_values[output_size] = y_val
            output_left_witness_ids[output_size] =
               left_witness_ids[left_index]
            output_right_witness_ids[output_size] =
               right_witness_ids[
                right_index
            ]

            output_size += 1
            left_index = next_left_index[left_index]

    return (
        output_values[:output_size],
        output_left_witness_ids[:output_size],
        output_right_witness_ids[:output_size],
    )

```

```
def bowtie_hash_join_int64(
    left_values: Sequence[int],
    left_witness_ids: Sequence[int],
    right_values: Sequence[int],
    right_witness_ids: Sequence[int],
    max_output_size: int,
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Run a fast Bowtie-Hash-Join on integer data.

    Args:
        left_values: Values of the left list.
        left_witness_ids: Witness identifiers of the left list.
        right_values: Values of the right list.
        right_witness_ids: Witness identifiers of the right
           list.
        max_output_size: Maximum number of output matches.

    Returns:
        A tuple ``(values, left_ids, right_ids)`` where every
           position
        represents one joined output element:
        ``(values[i], (left_ids[i], right_ids[i]))``.

    Raises:
        TypeError: If ``max_output_size`` is not an integer.
        ValueError: If input lengths do not match or if
            ``max_output_size`` is negative.
    """
    if not isinstance(max_output_size, int):
        raise TypeError("max_output_size must be an integer.")

    if max_output_size < 0:
        raise ValueError("max_output_size must be
           non-negative.")

    if len(left_values) != len(left_witness_ids):
        raise ValueError(
            "left_values and left_witness_ids must have the
               same length."
        )

    if len(right_values) != len(right_witness_ids):
        raise ValueError(
            "right_values and right_witness_ids must have the
               same length."
        )

    left_values_array = np.asarray(left_values, dtype=np.int64)
    left_witness_ids_array = np.asarray(left_witness_ids,
       dtype=np.int64)
```

```
    right_values_array = np.asarray(right_values, dtype=np.int64)
    right_witness_ids_array = np.asarray(right_witness_ids, dtype=np.int64)

    return _bowtie_hash_join_int64_core(
        left_values_array,
        left_witness_ids_array,
        right_values_array,
        right_witness_ids_array,
        max_output_size,
    )


@njit(cache=True)
def _lower_bound_int64(values: np.ndarray, target: int) -> int:
    """Return the first index j such that values[j] >= target."""
    left = 0
    right = values.shape[0]

    while left < right:
        middle = (left + right) // 2

        if values[middle] < target:
            left = middle + 1
        else:
            right = middle

    return left


@njit(cache=True)
def _upper_bound_int64(values: np.ndarray, target: int) -> int:
    """Return the first index j such that values[j] > target."""
    left = 0
    right = values.shape[0]

    while left < right:
        middle = (left + right) // 2

        if values[middle] <= target:
            left = middle + 1
        else:
            right = middle

    return left


@njit(cache=True)
def _mod_add_int64(left_value: int, right_value: int, modulus: int) -> int:
```

```
    """Return (left_value + right_value) mod modulus.

    This avoids overflow under the wrapper guarantee modulus <=
       2**62.
    """
    total = left_value + right_value

    if total >= modulus:
        total -= modulus

    return total


@njit(cache=True)
def _bowtie_interval_join_int64_core(
    left_values: np.ndarray,
    left_witness_ids: np.ndarray,
    sorted_right_values: np.ndarray,
    sorted_right_witness_ids: np.ndarray,
    modulus: int,
    radius: int,
    max_output_size: int,
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Run Bowtie-Interval-Join for sorted int64 arrays.

    Args:
        left_values: Values of the left list.
        left_witness_ids: Witness identifiers of the left list.
        sorted_right_values: Sorted values of the right list.
        sorted_right_witness_ids: Witness identifiers sorted in
           the same
            order as ``sorted_right_values``.
        modulus: The modulus q.
        radius: The interval radius R = floor(q / 2**(r + 1)).
        max_output_size: Maximum number of output elements.

    Returns:
        A tuple ``(values, left_ids, right_ids)``. Every
           position
        represents one output element:
        ``(values[i], (left_ids[i], right_ids[i]))``.
    """
    output_values = np.empty(max_output_size, dtype=np.int64)
    output_left_witness_ids = np.empty(max_output_size,
       dtype=np.int64)
    output_right_witness_ids = np.empty(max_output_size,
       dtype=np.int64)

    if max_output_size == 0:
        return (
            output_values[:0],
            output_left_witness_ids[:0],
```

```
            output_right_witness_ids[:0],
        )

    output_size = 0
    upper_interval_endpoint = radius - 1

    for left_index in range(left_values.shape[0]):
        x_val = left_values[left_index]
        x_wit = left_witness_ids[left_index]

        # alpha = (-R - x_val) mod q.
        shifted = x_val + radius
        if shifted >= modulus:
            shifted -= modulus

        if shifted == 0:
            alpha = 0
        else:
            alpha = modulus - shifted

        # beta = (R - 1 - x_val) mod q.
        if x_val <= upper_interval_endpoint:
            beta = upper_interval_endpoint - x_val
        else:
            beta = modulus - (x_val - upper_interval_endpoint)

        if alpha <= beta:
            # Non-wrapping interval: [alpha, beta].
            start = _lower_bound_int64(sorted_right_values,
               alpha)
            stop = _upper_bound_int64(sorted_right_values, beta)

            for right_index in range(start, stop):
                if output_size >= max_output_size:
                    return (
                        output_values[:output_size],
                        output_left_witness_ids[:output_size],
                        output_right_witness_ids[:output_size],
                    )

                y_val = sorted_right_values[right_index]

                output_values[output_size] = _mod_add_int64(
                    x_val,
                    y_val,
                    modulus,
                )
                output_left_witness_ids[output_size] = x_wit
                output_right_witness_ids[output_size] = (
                    sorted_right_witness_ids[right_index]
                )
                output_size += 1
```

```
299
300         else:
301             # Wrapping interval: [alpha, q - 1] union [0, beta].
302             start_1 = _lower_bound_int64(sorted_right_values,
                    alpha)
303             stop_1 = _upper_bound_int64(sorted_right_values,
                    modulus - 1)
304
305             for right_index in range(start_1, stop_1):
306                 if output_size >= max_output_size:
307                     return (
308                         output_values[:output_size],
309                         output_left_witness_ids[:output_size],
310                         output_right_witness_ids[:output_size],
311                     )
312
313                 y_val = sorted_right_values[right_index]
314
315                 output_values[output_size] = _mod_add_int64(
316                     x_val,
317                     y_val,
318                     modulus,
319                 )
320                 output_left_witness_ids[output_size] = x_wit
321                 output_right_witness_ids[output_size] = (
322                     sorted_right_witness_ids[right_index]
323                 )
324                 output_size += 1
325
326             start_2 = _lower_bound_int64(sorted_right_values, 0)
327             stop_2 = _upper_bound_int64(sorted_right_values,
                    beta)
328
329             for right_index in range(start_2, stop_2):
330                 if output_size >= max_output_size:
331                     return (
332                         output_values[:output_size],
333                         output_left_witness_ids[:output_size],
334                         output_right_witness_ids[:output_size],
335                     )
336
337                 y_val = sorted_right_values[right_index]
338
339                 output_values[output_size] = _mod_add_int64(
340                     x_val,
341                     y_val,
342                     modulus,
343                 )
344                 output_left_witness_ids[output_size] = x_wit
345                 output_right_witness_ids[output_size] = (
346                     sorted_right_witness_ids[right_index]
347                 )
```

```
                output_size += 1

    return (
        output_values[:output_size],
        output_left_witness_ids[:output_size],
        output_right_witness_ids[:output_size],
    )


def bowtie_interval_join_int64(
    left_values: Sequence[int],
    left_witness_ids: Sequence[int],
    right_values: Sequence[int],
    right_witness_ids: Sequence[int],
    modulus: int,
    filtering_parameter: int,
    max_output_size: int,
) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Run a fast Bowtie-Interval-Join on integer data.

    Args:
        left_values: Values of the left list. Each value must
           be in
            ``{0, 1, ..., modulus - 1}``.
        left_witness_ids: Witness identifiers of the left list.
        right_values: Values of the right list. Each value must
           be in
            ``{0, 1, ..., modulus - 1}``.
        right_witness_ids: Witness identifiers of the right
           list.
        modulus: The modulus q.
        filtering_parameter: The filtering parameter r.
        max_output_size: Maximum allowed output size.

    Returns:
        A tuple ``(values, left_ids, right_ids)`` where every
           position
        represents one merged element:
        ``(values[i], (left_ids[i], right_ids[i]))``.

    Raises:
        TypeError: If integer parameters are not integers.
        ValueError: If lengths do not match, parameters are
           invalid, or
            values are not canonical modulo ``modulus``.
    """
    if not isinstance(modulus, int):
        raise TypeError("modulus must be an integer.")

    if not isinstance(filtering_parameter, int):
        raise TypeError("filtering_parameter must be an
           integer.")
```

```
394
395     if not isinstance(max_output_size, int):
396         raise TypeError("max_output_size must be an integer.")
397
398     if modulus < 2:
399         raise ValueError("modulus must be at least 2.")
400
401     if modulus > MAX_SAFE_INT64_MODULUS:
402         raise ValueError(
403             "modulus is too large for this int64
                implementation. "
404             "Use modulus <= 2**62."
405         )
406
407     if filtering_parameter < 0:
408         raise ValueError("filtering_parameter must be
            non-negative.")
409
410     if max_output_size < 0:
411         raise ValueError("max_output_size must be
            non-negative.")
412
413     if len(left_values) != len(left_witness_ids):
414         raise ValueError(
415             "left_values and left_witness_ids must have the
                same length."
416         )
417
418     if len(right_values) != len(right_witness_ids):
419         raise ValueError(
420             "right_values and right_witness_ids must have the
                same length."
421         )
422
423     if filtering_parameter + 1 > modulus.bit_length():
424         radius = 0
425     else:
426         radius = modulus // (1 << (filtering_parameter + 1))
427
428     if radius < 1:
429         raise ValueError(
430             "Invalid filtering parameter: "
431             "floor(modulus / 2**(filtering_parameter + 1)) must
                be >= 1."
432         )
433
434     left_values_array = np.asarray(left_values, dtype=np.int64)
435     left_witness_ids_array = np.asarray(left_witness_ids,
        dtype=np.int64)
436     right_values_array = np.asarray(right_values,
        dtype=np.int64)
```

```
    right_witness_ids_array = np.asarray(right_witness_ids,
        dtype=np.int64)

    if left_values_array.ndim != 1:
        raise ValueError("left_values must be one-dimensional.")

    if left_witness_ids_array.ndim != 1:
        raise ValueError("left_witness_ids must be
            one-dimensional.")

    if right_values_array.ndim != 1:
        raise ValueError("right_values must be
            one-dimensional.")

    if right_witness_ids_array.ndim != 1:
        raise ValueError("right_witness_ids must be
            one-dimensional.")

    if np.any(left_values_array < 0) or
        np.any(left_values_array >= modulus):
        raise ValueError(
            "All left values must be canonical representatives
                modulo modulus."
        )

    if np.any(right_values_array < 0) or
        np.any(right_values_array >= modulus):
        raise ValueError(
            "All right values must be canonical representatives
                modulo modulus."
        )

    if np.any(left_witness_ids_array < 0):
        raise ValueError("left_witness_ids must be
            non-negative.")

    if np.any(right_witness_ids_array < 0):
        raise ValueError("right_witness_ids must be
            non-negative.")

    right_order = np.argsort(right_values_array, kind="stable")
    sorted_right_values = right_values_array[right_order]
    sorted_right_witness_ids =
        right_witness_ids_array[right_order]

    return _bowtie_interval_join_int64_core(
        left_values=left_values_array,
        left_witness_ids=left_witness_ids_array,
        sorted_right_values=sorted_right_values,
        sorted_right_witness_ids=sorted_right_witness_ids,
        modulus=modulus,
        radius=radius,
```

```
        max_output_size=max_output_size,
    )


@dataclass(slots=True)
class WitnessStore:
    """Store witness back-pointers for the K-tree solver.

    Each witness id is an integer.

    A witness is either:
    - a leaf witness, pointing to an original list element, or
    - an internal witness, pointing to two child witnesses.
    """

    left_parent: list[int]
    right_parent: list[int]
    leaf_list_id: list[int]
    leaf_index: list[int]

    @classmethod
    def empty(cls) -> "WitnessStore":
        """Create an empty witness store."""
        return cls(
            left_parent=[],
            right_parent=[],
            leaf_list_id=[],
            leaf_index=[],
        )

    def add_leaf(self, list_id: int, index: int) -> int:
        """Add a leaf witness.

        Args:
            list_id: Index of the original input list.
            index: Index of the element inside that list.

        Returns:
            The new witness id.
        """
        witness_id = len(self.left_parent)

        self.left_parent.append(-1)
        self.right_parent.append(-1)
        self.leaf_list_id.append(list_id)
        self.leaf_index.append(index)

        return witness_id

    def add_leaves(self, list_id: int, count: int) ->
       np.ndarray:
        """Add many leaf witnesses for one input list.
```

```
529
530         Args:
531             list_id: Index of the original input list.
532             count: Number of elements in the list.
533
534         Returns:
535             Array of new witness ids.
536         """
537         witness_ids = np.empty(count, dtype=np.int64)
538
539         for index in range(count):
540             witness_ids[index] = self.add_leaf(list_id, index)
541
542         return witness_ids
543
544     def add_internal(self, left_witness: int, right_witness:
        int) -> int:
545         """Add an internal witness.
546
547         Args:
548             left_witness: Witness id of the left child.
549             right_witness: Witness id of the right child.
550
551         Returns:
552             The new witness id.
553         """
554         witness_id = len(self.left_parent)
555
556         self.left_parent.append(left_witness)
557         self.right_parent.append(right_witness)
558         self.leaf_list_id.append(-1)
559         self.leaf_index.append(-1)
560
561         return witness_id
562
563     def add_internal_many(
564         self,
565         left_witness_ids: np.ndarray,
566         right_witness_ids: np.ndarray,
567     ) -> np.ndarray:
568         """Add many internal witnesses.
569
570         Args:
571             left_witness_ids: Left child witness ids.
572             right_witness_ids: Right child witness ids.
573
574         Returns:
575             Array of new witness ids.
576
577         Raises:
578             ValueError: If the input arrays have different
                lengths.
```

```
        """
        if left_witness_ids.shape[0] !=
           right_witness_ids.shape[0]:
            raise ValueError(
                "left_witness_ids and right_witness_ids must
                   have "
                "the same length."
            )

        count = left_witness_ids.shape[0]
        new_witness_ids = np.empty(count, dtype=np.int64)

        for index in range(count):
            new_witness_ids[index] = self.add_internal(
                int(left_witness_ids[index]),
                int(right_witness_ids[index]),
            )

        return new_witness_ids

    def recover_leaf_indices(self, witness_id: int) ->
       list[tuple[int, int]]:
        """Recover all original leaves used by a witness.

        Args:
            witness_id: Root witness id.

        Returns:
            A list of pairs ``(list_id, element_index)``.
        """
        if witness_id < 0 or witness_id >=
           len(self.left_parent):
            raise ValueError("Invalid witness_id.")

        left = self.left_parent[witness_id]
        right = self.right_parent[witness_id]

        if left == -1 and right == -1:
            return [
                (
                    self.leaf_list_id[witness_id],
                    self.leaf_index[witness_id],
                )
            ]

        leaves: list[tuple[int, int]] = []
        leaves.extend(self.recover_leaf_indices(left))
        leaves.extend(self.recover_leaf_indices(right))

        return leaves
```

```python
@dataclass(slots=True)
class KTreeResult:
    """Result of the Generalized-K-Tree-Modular-Solver."""

    values: np.ndarray
    root_witness_ids: np.ndarray
    witness_store: WitnessStore


def _is_power_of_two(value: int) -> bool:
    """Return True if value is a positive power of two."""
    return value > 0 and (value & (value - 1)) == 0


def _ceil_log2(value: int) -> int:
    """Return ceil(log2(value)) for a positive integer."""
    if value <= 0:
        raise ValueError("value must be positive.")

    return (value - 1).bit_length()


def _canonical_negatives(values: np.ndarray, modulus: int) ->
   np.ndarray:
    """Return Canon_q(-values) for canonical values modulo q."""
    return np.where(values == 0, 0, modulus -
       values).astype(np.int64)


def generalized_k_tree_modular_solver_int64(
    input_lists: Sequence[Sequence[int]],
    modulus: int,
    max_intermediate_size: int,
    max_solution_count: int,
) -> KTreeResult:
    """Run the Generalized-K-Tree-Modular-Solver.

    Args:
        input_lists: Sequence of k input lists. Each element
           must be a
            canonical representative in ``{0, ..., modulus -
               1}``.
        modulus: The modulus q.
        max_intermediate_size: Maximum size of every
           intermediate list.
        max_solution_count: Maximum number of final solutions
           returned.

    Returns:
        A ``KTreeResult`` containing:
        - final values, always zero modulo q,
        - root witness ids,
```

```
        - witness store for backtracking.

    Raises:
        TypeError: If integer parameters are not integers.
        ValueError: If parameters or input lists are invalid.
    """
    if not isinstance(modulus, int):
        raise TypeError("modulus must be an integer.")

    if not isinstance(max_intermediate_size, int):
        raise TypeError("max_intermediate_size must be an
            integer.")

    if not isinstance(max_solution_count, int):
        raise TypeError("max_solution_count must be an
            integer.")

    if modulus < 2:
        raise ValueError("modulus must be at least 2.")

    if max_intermediate_size < 0:
        raise ValueError("max_intermediate_size must be
            non-negative.")

    if max_solution_count < 0:
        raise ValueError("max_solution_count must be
            non-negative.")

    k = len(input_lists)

    if k < 2:
        return KTreeResult(
            values=np.empty(0, dtype=np.int64),
            root_witness_ids=np.empty(0, dtype=np.int64),
            witness_store=WitnessStore.empty(),
        )

    if not _is_power_of_two(k):
        return KTreeResult(
            values=np.empty(0, dtype=np.int64),
            root_witness_ids=np.empty(0, dtype=np.int64),
            witness_store=WitnessStore.empty(),
        )

    depth = k.bit_length() - 1
    bit_length_scale = _ceil_log2(modulus)
    max_filtering_parameter = max(0, modulus.bit_length() - 2)

    witness_store = WitnessStore.empty()
    current: list[tuple[np.ndarray, np.ndarray]] = []

    for list_id, values in enumerate(input_lists):
```

```
        values_array = np.asarray(values, dtype=np.int64)

        if values_array.ndim != 1:
            raise ValueError("Each input list must be
               one-dimensional.")

        if values_array.shape[0] == 0:
            return KTreeResult(
                values=np.empty(0, dtype=np.int64),
                root_witness_ids=np.empty(0, dtype=np.int64),
                witness_store=witness_store,
            )

        if np.any(values_array < 0) or np.any(values_array >=
           modulus):
            raise ValueError(
                "All input values must be canonical
                   representatives "
                "modulo modulus."
            )

        witness_ids = witness_store.add_leaves(
            list_id=list_id,
            count=values_array.shape[0],
        )

        current.append((values_array, witness_ids))

    for height in range(1, depth):
        filtering_parameter = min(
            (height * bit_length_scale) // (depth + 1),
            max_filtering_parameter,
        )

        next_level: list[tuple[np.ndarray, np.ndarray]] = []

        for tree_index in range(len(current) // 2):
            left_values, left_witness_ids = current[2 *
               tree_index]
            right_values, right_witness_ids = current[2 *
               tree_index + 1]

            (
                merged_values,
                merged_left_witness_ids,
                merged_right_witness_ids,
            ) = bowtie_interval_join_int64(
                left_values=left_values,
                left_witness_ids=left_witness_ids,
                right_values=right_values,
                right_witness_ids=right_witness_ids,
                modulus=modulus,
```

```
                filtering_parameter=filtering_parameter,
                max_output_size=max_intermediate_size,
            )

            merged_witness_ids =
               witness_store.add_internal_many(
                left_witness_ids=merged_left_witness_ids,
                right_witness_ids=merged_right_witness_ids,
            )

            next_level.append((merged_values,
               merged_witness_ids))

        current = next_level

    root_left_values, root_left_witness_ids = current[0]
    root_right_values, root_right_witness_ids = current[1]

    negated_root_right_values = _canonical_negatives(
        root_right_values,
        modulus,
    )

    (
        _matched_values,
        final_left_witness_ids,
        final_right_witness_ids,
    ) = bowtie_hash_join_int64(
        left_values=root_left_values,
        left_witness_ids=root_left_witness_ids,
        right_values=negated_root_right_values,
        right_witness_ids=root_right_witness_ids,
        max_output_size=max_solution_count,
    )

    root_witness_ids = witness_store.add_internal_many(
        left_witness_ids=final_left_witness_ids,
        right_witness_ids=final_right_witness_ids,
    )

    final_values = np.zeros(root_witness_ids.shape[0],
       dtype=np.int64)

    return KTreeResult(
        values=final_values,
        root_witness_ids=root_witness_ids,
        witness_store=witness_store,
    )


@dataclass(slots=True)
class GroupedWagnerCollisionResult:
```

```
    """Result of the grouped Wagner rank-one collision
       generator."""

    alphas: np.ndarray
    epsilon: np.ndarray
    z: np.ndarray
    z_plus: np.ndarray
    z_minus: np.ndarray
    root_witness_id: int
    group_count: int


def _validate_grouped_rank_one_inputs(
    block_vectors: Sequence[Sequence[int]],
    modulus: int,
    block_size: int,
    total_dimension: int,
    number_of_blocks: int,
    max_intermediate_size: int,
    max_relation_count: int,
) -> list[np.ndarray]:
    """Validate inputs for the grouped Wagner rank-one
       generator."""
    if not isinstance(modulus, int):
        raise TypeError("modulus must be an integer.")

    if not isinstance(block_size, int):
        raise TypeError("block_size must be an integer.")

    if not isinstance(total_dimension, int):
        raise TypeError("total_dimension must be an integer.")

    if not isinstance(number_of_blocks, int):
        raise TypeError("number_of_blocks must be an integer.")

    if not isinstance(max_intermediate_size, int):
        raise TypeError("max_intermediate_size must be an
           integer.")

    if not isinstance(max_relation_count, int):
        raise TypeError("max_relation_count must be an
           integer.")

    if modulus < 2:
        raise ValueError("modulus must be at least 2.")

    if modulus > MAX_SAFE_INT64_MODULUS:
        raise ValueError("modulus must be at most 2**62.")

    if block_size < 1:
        raise ValueError("block_size must be positive.")

```

```
    if number_of_blocks < 2:
        raise ValueError("number_of_blocks must be at least 2.")

    if total_dimension != number_of_blocks * block_size:
        raise ValueError(
            "total_dimension must equal number_of_blocks *
               block_size."
        )

    if len(block_vectors) != number_of_blocks:
        raise ValueError(
            "block_vectors must contain exactly
               number_of_blocks blocks."
        )

    if max_intermediate_size < 1:
        raise ValueError("max_intermediate_size must be
           positive.")

    if max_relation_count < 1:
        raise ValueError("max_relation_count must be positive.")

    normalized_blocks: list[np.ndarray] = []

    for block_index, block in enumerate(block_vectors):
        if len(block) != block_size:
            raise ValueError(
                f"block_vectors[{block_index}] must have length
                   block_size."
            )

        normalized_block = np.empty(block_size, dtype=np.int64)

        for coefficient_index, coefficient in enumerate(block):
            if not isinstance(coefficient, (int, np.integer)):
                raise TypeError("All block coefficients must be
                   integers.")

            coefficient_int = int(coefficient) if False else
               int(coefficient)

            if coefficient_int < 0 or coefficient_int >=
               modulus:
                raise ValueError(
                    "All block coefficients must be canonical "
                    "representatives modulo modulus."
                )

            normalized_block[coefficient_index] =
               coefficient_int

        normalized_blocks.append(normalized_block)
```

```
    return normalized_blocks


def _compute_alphas_from_blocks(
    block_vectors: Sequence[np.ndarray],
    modulus: int,
) -> np.ndarray:
    """Compute alpha_i = sum_j a_{i,j} mod q."""
    alphas = np.empty(len(block_vectors), dtype=np.int64)

    for block_index, block in enumerate(block_vectors):
        total = 0

        for coefficient in block:
            total += int(coefficient)

        alphas[block_index] = total % modulus

    return alphas


def _power_of_three(exponent: int) -> int:
    """Return 3**exponent."""
    if exponent < 0:
        raise ValueError("exponent must be non-negative.")

    result = 1

    for _ in range(exponent):
        result *= 3

    return result


def _select_wagner_group_count(
    number_of_blocks: int,
    modulus: int,
    max_group_list_size: int,
) -> int:
    """Choose a power-of-two number of grouped Wagner lists."""
    if max_group_list_size < 1:
        raise ValueError("max_group_list_size must be
           positive.")

    candidates: list[tuple[float, int]] = []

    group_count = 2

    while group_count <= number_of_blocks:
        max_group_size = math.ceil(number_of_blocks /
           group_count)
```

```
        estimated_list_size = _power_of_three(max_group_size)

        if estimated_list_size <= max_group_list_size:
            target_size = modulus ** (1.0 / (1.0 +
               math.log2(group_count)))
            score = estimated_list_size / target_size
            candidates.append((score, group_count))

        group_count *= 2

    if not candidates:
        raise ValueError(
            "No feasible group_count found. Increase
               max_group_list_size."
        )

    candidates.sort(reverse=True)
    return candidates[0][1]


def _split_indices_evenly(
    number_of_blocks: int,
    group_count: int,
) -> list[list[int]]:
    """Split alpha indices into group_count balanced groups."""
    if group_count < 2:
        raise ValueError("group_count must be at least 2.")

    if group_count > number_of_blocks:
        raise ValueError("group_count must not exceed
           number_of_blocks.")

    if not _is_power_of_two(group_count):
        raise ValueError("group_count must be a power of two.")

    groups: list[list[int]] = []
    base_size = number_of_blocks // group_count
    remainder = number_of_blocks % group_count
    start = 0

    for group_index in range(group_count):
        extra = 1 if group_index < remainder else 0
        stop = start + base_size + extra
        groups.append(list(range(start, stop)))
        start = stop

    return groups


def _epsilon_choices_for_position(
    index: int,
    first_nonzero_position: int,
```

```
) -> tuple[int, ...]:
    """Return allowed epsilon choices for one position."""
    if index < first_nonzero_position:
        return (0,)

    if index == first_nonzero_position:
        return (1, -1)

    return (0, 1, -1)


def _build_one_group_signed_sum_list(
    alphas: np.ndarray,
    modulus: int,
    group_indices: Sequence[int],
    first_nonzero_position: int,
    max_group_list_size: int,
) -> tuple[list[int], list[tuple[int, ...]]]:
    """Build one grouped signed-sum list."""
    states: list[tuple[int, tuple[int, ...]]] = [(0, ())]

    for alpha_index in group_indices:
        alpha = int(alphas[alpha_index])
        choices = _epsilon_choices_for_position(
            index=alpha_index,
            first_nonzero_position=first_nonzero_position,
        )

        next_states: list[tuple[int, tuple[int, ...]]] = []

        for residue, signs in states:
            for sign in choices:
                new_residue = (residue + sign * alpha) % modulus
                next_states.append((new_residue, signs +
                    (sign,)))

        states = next_states

        if len(states) > max_group_list_size:
            raise ValueError(
                "A grouped signed-sum list exceeded
                    max_group_list_size."
            )

    values = [residue for residue, _signs in states]
    sign_patterns = [signs for _residue, signs in states]

    return values, sign_patterns


def _build_grouped_signed_sum_lists(
    alphas: np.ndarray,
```

```
    modulus: int,
    groups: Sequence[Sequence[int]],
    first_nonzero_position: int,
    max_group_list_size: int,
) -> tuple[list[list[int]], list[list[tuple[int, ...]]]]:
    """Build all grouped signed-sum lists for one first nonzero
       position."""
    value_lists: list[list[int]] = []
    sign_maps: list[list[tuple[int, ...]]] = []

    for group_indices in groups:
        values, signs = _build_one_group_signed_sum_list(
            alphas=alphas,
            modulus=modulus,
            group_indices=group_indices,
            first_nonzero_position=first_nonzero_position,
            max_group_list_size=max_group_list_size,
        )

        value_lists.append(values)
        sign_maps.append(signs)

    return value_lists, sign_maps


def _make_zero_sum_offsets(
    group_count: int,
    modulus: int,
    rng: np.random.Generator,
    attempt_index: int,
) -> list[int]:
    """Make random offsets whose total is zero modulo q."""
    offsets = [0 for _ in range(group_count)]

    if attempt_index == 0:
        return offsets

    partial_sum = 0

    for index in range(group_count - 1):
        offset = int(
            rng.integers(
                low=0,
                high=modulus,
                dtype=np.int64,
            )
        )
        offsets[index] = offset
        partial_sum = (partial_sum + offset) % modulus

    offsets[-1] = (-partial_sum) % modulus

```

```
    return offsets


def _shift_value_lists(
    value_lists: Sequence[Sequence[int]],
    offsets: Sequence[int],
    modulus: int,
) -> list[list[int]]:
    """Shift grouped lists without changing the total zero-sum
       condition."""
    shifted_lists: list[list[int]] = []

    for values, offset in zip(value_lists, offsets,
       strict=True):
        shifted_values = [
            (int(value) + int(offset)) % modulus
            for value in values
        ]
        shifted_lists.append(shifted_values)

    return shifted_lists


def _recover_grouped_epsilon(
    result: KTreeResult,
    groups: Sequence[Sequence[int]],
    sign_maps: Sequence[Sequence[tuple[int, ...]]],
    number_of_blocks: int,
    root_witness_id: int,
) -> np.ndarray:
    """Recover epsilon from grouped signed-sum witnesses."""
    epsilon = np.zeros(number_of_blocks, dtype=np.int64)
    leaves =
       result.witness_store.recover_leaf_indices(root_witness_id)

    for group_id, element_index in leaves:
        if group_id >= len(groups):
            continue

        group_indices = groups[group_id]
        signs = sign_maps[group_id][element_index]

        for local_index, alpha_index in
           enumerate(group_indices):
            epsilon[alpha_index] = int(signs[local_index])

    return epsilon


def _is_valid_epsilon_relation(
    epsilon: np.ndarray,
    alphas: np.ndarray,
```

```
    modulus: int,
) -> bool:
    """Return True if sum_i epsilon_i alpha_i is zero modulo
       q."""
    total = 0

    for index in range(epsilon.shape[0]):
        total += int(epsilon[index]) * int(alphas[index])

    return total % modulus == 0


def _lift_epsilon_to_z(
    epsilon: np.ndarray,
    block_size: int,
) -> np.ndarray:
    """Lift epsilon to z by repeating each coordinate
       block_size times."""
    return np.repeat(epsilon.astype(np.int64), block_size)


def _split_ternary_vector_to_binary_pair(
    z: np.ndarray,
) -> tuple[np.ndarray, np.ndarray]:
    """Write z as z_plus - z_minus."""
    z_plus = (z == 1).astype(np.int64)
    z_minus = (z == -1).astype(np.int64)

    return z_plus, z_minus


def rank_one_collapse_collision_generator_grouped_wagner_int64(
    block_vectors: Sequence[Sequence[int]],
    modulus: int,
    block_size: int,
    total_dimension: int,
    number_of_blocks: int,
    max_intermediate_size: int,
    max_relation_count: int = 1,
    max_group_list_size: int = 100_000,
    random_shift_attempts: int = 30,
    seed: int = 12345,
) -> GroupedWagnerCollisionResult | None:
    """Run rank-one collapse using grouped signed sums and
       Wagner K-tree.

    This keeps the Wagner solver, but gives it larger input
       lists by
    grouping several alpha_i values into each signed-sum list.
    """
    if random_shift_attempts < 1:
```

```
        raise ValueError("random_shift_attempts must be positive.")

    normalized_blocks = _validate_grouped_rank_one_inputs(
        block_vectors=block_vectors,
        modulus=modulus,
        block_size=block_size,
        total_dimension=total_dimension,
        number_of_blocks=number_of_blocks,
        max_intermediate_size=max_intermediate_size,
        max_relation_count=max_relation_count,
    )

    alphas = _compute_alphas_from_blocks(
        block_vectors=normalized_blocks,
        modulus=modulus,
    )

    group_count = _select_wagner_group_count(
        number_of_blocks=number_of_blocks,
        modulus=modulus,
        max_group_list_size=max_group_list_size,
    )

    groups = _split_indices_evenly(
        number_of_blocks=number_of_blocks,
        group_count=group_count,
    )

    rng = np.random.default_rng(seed)

    for first_nonzero_position in range(number_of_blocks):
        value_lists, sign_maps = _build_grouped_signed_sum_lists(
            alphas=alphas,
            modulus=modulus,
            groups=groups,
            first_nonzero_position=first_nonzero_position,
            max_group_list_size=max_group_list_size,
        )

        for attempt_index in range(random_shift_attempts):
            offsets = _make_zero_sum_offsets(
                group_count=group_count,
                modulus=modulus,
                rng=rng,
                attempt_index=attempt_index,
            )

            shifted_value_lists = _shift_value_lists(
                value_lists=value_lists,
                offsets=offsets,
```

```
                modulus=modulus,
            )

            result = generalized_k_tree_modular_solver_int64(
                input_lists=shifted_value_lists,
                modulus=modulus,
                max_intermediate_size=max_intermediate_size,
                max_solution_count=max_relation_count,
            )

            if result.root_witness_ids.shape[0] == 0:
                continue

            root_witness_id = int(result.root_witness_ids[0])

            epsilon = _recover_grouped_epsilon(
                result=result,
                groups=groups,
                sign_maps=sign_maps,
                number_of_blocks=number_of_blocks,
                root_witness_id=root_witness_id,
            )

            if not np.any(epsilon):
                continue

            if not _is_valid_epsilon_relation(
                epsilon=epsilon,
                alphas=alphas,
                modulus=modulus,
            ):
                continue

            z = _lift_epsilon_to_z(
                epsilon=epsilon,
                block_size=block_size,
            )

            z_plus, z_minus =
               _split_ternary_vector_to_binary_pair(z)

            return GroupedWagnerCollisionResult(
                alphas=alphas,
                epsilon=epsilon,
                z=z,
                z_plus=z_plus,
                z_minus=z_minus,
                root_witness_id=root_witness_id,
                group_count=group_count,
            )

    return None
```

```python


# HELPER FUNCTIONS


def build_circulant_block_from_definition(
    block: Sequence[int],
    modulus: int,
) -> np.ndarray:
    """Build Rot(a) with columns a, Xa, ..., X^(n-1)a."""
    block_array = np.asarray(block, dtype=np.int64)
    block_size = block_array.shape[0]

    rot_block = np.empty((block_size, block_size),
       dtype=np.int64)

    for row in range(block_size):
        for col in range(block_size):
            rot_block[row, col] = block_array[(row - col) %
               block_size] % modulus

    return rot_block


def build_public_matrix_a(
    block_vectors: Sequence[Sequence[int]],
    modulus: int,
) -> np.ndarray:
    """Build A = [Rot(a_1) | ... | Rot(a_ell)]."""
    circulant_blocks = [
        build_circulant_block_from_definition(block, modulus)
        for block in block_vectors
    ]

    return np.hstack(circulant_blocks)


def matrix_vector_mod(
    matrix: np.ndarray,
    vector: np.ndarray,
    modulus: int,
) -> np.ndarray:
    """Compute matrix @ vector modulo modulus without int64
       overflow."""
    if matrix.shape[1] != vector.shape[0]:
        raise ValueError("Matrix columns must match vector
           length.")

    output = np.empty(matrix.shape[0], dtype=np.int64)

    for row in range(matrix.shape[0]):
        total = 0
```

```
        for col in range(matrix.shape[1]):
            total += int(matrix[row, col]) * int(vector[col])
            total %= modulus

        output[row] = total

    return output
```

Listing 1: Python implementation of the collision-generation algorithms

```
from __future__ import annotations

from dataclasses import dataclass
from typing import Literal, Sequence

import numpy as np

MAX_SAFE_INT64_MODULUS = 1 << 62


@dataclass(slots=True)
class WitnessStore:
    """Store witness back-pointers for Wagner tree
       reconstruction."""

    left_parent: list[int]
    right_parent: list[int]
    leaf_list_id: list[int]
    leaf_index: list[int]

    @classmethod
    def empty(cls) -> "WitnessStore":
        """Create an empty witness store."""
        return cls(
            left_parent=[],
            right_parent=[],
            leaf_list_id=[],
            leaf_index=[],
        )

    def add_leaf(self, list_id: int, index: int) -> int:
        """Add a leaf witness."""
        witness_id = len(self.left_parent)

        self.left_parent.append(-1)
        self.right_parent.append(-1)
        self.leaf_list_id.append(list_id)
        self.leaf_index.append(index)

        return witness_id

    def add_leaves(self, list_id: int, count: int) ->
```

```
        np.ndarray:
        """Add many leaf witnesses for one list."""
        witness_ids = np.empty(count, dtype=np.int64)

        for index in range(count):
            witness_ids[index] = self.add_leaf(list_id, index)

        return witness_ids

    def add_internal(self, left_witness: int, right_witness:
        int) -> int:
        """Add an internal witness."""
        witness_id = len(self.left_parent)

        self.left_parent.append(left_witness)
        self.right_parent.append(right_witness)
        self.leaf_list_id.append(-1)
        self.leaf_index.append(-1)

        return witness_id

    def recover_leaf_indices(self, witness_id: int) ->
        list[tuple[int, int]]:
        """Recover all original leaves used by a root
            witness."""
        if witness_id < 0 or witness_id >=
            len(self.left_parent):
            raise ValueError("Invalid witness_id.")

        left = self.left_parent[witness_id]
        right = self.right_parent[witness_id]

        if left == -1 and right == -1:
            return [
                (
                    self.leaf_list_id[witness_id],
                    self.leaf_index[witness_id],
                )
            ]

        leaves: list[tuple[int, int]] = []
        leaves.extend(self.recover_leaf_indices(left))
        leaves.extend(self.recover_leaf_indices(right))

        return leaves


@dataclass(slots=True)
class VectorWagnerList:
    """Store one vector-valued Wagner list."""

    values: np.ndarray
```

```
    witness_ids: np.ndarray


@dataclass(slots=True)
class NegacyclicWagnerCollisionResult:
    """Result of the Negacyclic-Wagner-Collision-Generator."""

    z: np.ndarray
    z_plus: np.ndarray
    z_minus: np.ndarray
    group_count: int
    root_witness_id: int


def _is_power_of_two(value: int) -> bool:
    """Return True if value is a positive power of two."""
    return value > 0 and (value & (value - 1)) == 0


def _validate_negacyclic_wagner_inputs(
    block_vectors: Sequence[Sequence[int]],
    modulus: int,
    block_size: int,
    total_dimension: int,
    number_of_blocks: int,
    list_count: int,
    max_initial_list_size: int,
    max_intermediate_size: int,
) -> list[np.ndarray]:
    """Validate inputs for the negacyclic Wagner collision
       generator."""
    if not isinstance(modulus, int):
        raise TypeError("modulus must be an integer.")

    if not isinstance(block_size, int):
        raise TypeError("block_size must be an integer.")

    if not isinstance(total_dimension, int):
        raise TypeError("total_dimension must be an integer.")

    if not isinstance(number_of_blocks, int):
        raise TypeError("number_of_blocks must be an integer.")

    if not isinstance(list_count, int):
        raise TypeError("list_count must be an integer.")

    if not isinstance(max_initial_list_size, int):
        raise TypeError("max_initial_list_size must be an
           integer.")

    if not isinstance(max_intermediate_size, int):
        raise TypeError("max_intermediate_size must be an
```

```
            integer.")

    if modulus < 2:
        raise ValueError("modulus must be at least 2.")

    if modulus > MAX_SAFE_INT64_MODULUS:
        raise ValueError("modulus must be at most 2**62.")

    if block_size < 1:
        raise ValueError("block_size must be positive.")

    if number_of_blocks < 1:
        raise ValueError("number_of_blocks must be positive.")

    if total_dimension != number_of_blocks * block_size:
        raise ValueError(
            "total_dimension must equal number_of_blocks *
               block_size."
        )

    if list_count < 2:
        raise ValueError("list_count must be at least 2.")

    if not _is_power_of_two(list_count):
        raise ValueError("list_count must be a power of two.")

    if list_count > total_dimension:
        raise ValueError("list_count must not exceed
            total_dimension.")

    if total_dimension % list_count != 0:
        raise ValueError("list_count must divide
            total_dimension.")

    if max_initial_list_size < 1:
        raise ValueError("max_initial_list_size must be
            positive.")

    if max_intermediate_size < 1:
        raise ValueError("max_intermediate_size must be
            positive.")

    if len(block_vectors) != number_of_blocks:
        raise ValueError(
            "block_vectors must contain exactly
               number_of_blocks blocks."
        )

    normalized_blocks: list[np.ndarray] = []

    for block_index, block in enumerate(block_vectors):
        if len(block) != block_size:
```

```
            raise ValueError(
                f"block_vectors[{block_index}] must have length block_size."
            )

        normalized_block = np.empty(block_size, dtype=np.int64)

        for coefficient_index, coefficient in enumerate(block):
            if not isinstance(coefficient, (int, np.integer)):
                raise TypeError("All block coefficients must be integers.")

            coefficient_int = int(coefficient) if False else int(coefficient)

            if coefficient_int < 0 or coefficient_int >= modulus:
                raise ValueError(
                    "All block coefficients must be canonical "
                    "representatives modulo modulus."
                )

            normalized_block[coefficient_index] = coefficient_int

        normalized_blocks.append(normalized_block)

    return normalized_blocks


def build_negacyclic_block_from_definition(
    block: Sequence[int],
    modulus: int,
) -> np.ndarray:
    """Build Rot_neg(a) with columns a, xa, ..., x^(n-1)a.

    The multiplication is performed in Z_q[x] / (x^n + 1).
    """
    block_array = np.asarray(block, dtype=np.int64)

    if block_array.ndim != 1:
        raise ValueError("block must be one-dimensional.")

    block_size = block_array.shape[0]
    rot_block = np.empty((block_size, block_size), dtype=np.int64)

    for column in range(block_size):
        for coefficient_index in range(block_size):
            row = coefficient_index + column

            if row < block_size:
```

```
                rot_block[row, column] =
                   block_array[coefficient_index] % modulus
            else:
                row -= block_size
                rot_block[row, column] = (
                    modulus - (block_array[coefficient_index] %
                       modulus)
                ) % modulus

    return rot_block


def build_negacyclic_public_matrix_a(
    block_vectors: Sequence[Sequence[int]],
    modulus: int,
) -> np.ndarray:
    """Build A = [Rot_neg(a_1) | ... | Rot_neg(a_ell)]."""
    negacyclic_blocks = [
        build_negacyclic_block_from_definition(block, modulus)
        for block in block_vectors
    ]

    return np.hstack(negacyclic_blocks)


def matrix_vector_mod(
    matrix: np.ndarray,
    vector: np.ndarray,
    modulus: int,
) -> np.ndarray:
    """Compute matrix @ vector modulo modulus."""
    if matrix.shape[1] != vector.shape[0]:
        raise ValueError("Matrix columns must match vector
           length.")

    output = np.empty(matrix.shape[0], dtype=np.int64)

    for row in range(matrix.shape[0]):
        total = 0

        for column in range(matrix.shape[1]):
            total += int(matrix[row, column]) *
               int(vector[column])
            total %= modulus

        output[row] = total

    return output


def _split_ternary_vector_to_binary_pair(
    z: np.ndarray,
```

```
) -> tuple[np.ndarray, np.ndarray]:
    """Write z as z_plus - z_minus."""
    z_plus = (z == 1).astype(np.int64)
    z_minus = (z == -1).astype(np.int64)

    return z_plus, z_minus


def _make_wagner_column_groups(
    total_dimension: int,
    list_count: int,
    rng: np.random.Generator,
    group_strategy: Literal["random", "consecutive"] = "random",
) -> list[np.ndarray]:
    """Split column indices into balanced Wagner groups.

    The default strategy is random permutation, because
       consecutive grouping
    may place many correlated negacyclic shifts inside the same
       group.
    """
    group_size = total_dimension // list_count

    if group_strategy == "consecutive":
        ordered_indices = np.arange(total_dimension,
           dtype=np.int64)
    elif group_strategy == "random":
        ordered_indices =
           rng.permutation(total_dimension).astype(np.int64)
    else:
        raise ValueError("group_strategy must be 'random' or
           'consecutive'.")

    groups: list[np.ndarray] = []

    for group_index in range(list_count):
        start = group_index * group_size
        stop = (group_index + 1) * group_size
        groups.append(ordered_indices[start:stop])

    return groups


def _bounded_power_of_three(exponent: int, bound: int) -> int:
    """Return min(3**exponent, bound + 1)."""
    result = 1

    for _ in range(exponent):
        result *= 3

        if result > bound:
            return bound + 1
```

```
    return result


def _code_to_ternary_signs(code: int, length: int) -> np.ndarray:
    """Convert a base-3 code to signs in {-1, 0, 1}."""
    signs = np.empty(length, dtype=np.int8)

    for index in range(length):
        digit = code % 3
        code //= 3
        signs[index] = np.int8(digit - 1)

    return signs


def _random_ternary_signs(
    length: int,
    count: int,
    rng: np.random.Generator,
) -> np.ndarray:
    """Generate distinct random ternary sign patterns."""
    signs_array = np.empty((count, length), dtype=np.int8)
    seen: set[bytes] = set()
    filled = 0
    attempts = 0
    max_attempts = max(10_000, 20 * count)

    while filled < count and attempts < max_attempts:
        signs = rng.integers(
            low=-1,
            high=2,
            size=length,
            dtype=np.int8,
        )

        key = signs.tobytes()

        if key not in seen:
            seen.add(key)
            signs_array[filled] = signs
            filled += 1

        attempts += 1

    if filled < count:
        raise RuntimeError(
            "Could not generate enough distinct ternary sign patterns."
        )

```

```
    return signs_array


def _enumerate_or_sample_signs(
    group_size: int,
    max_list_size: int,
    rng: np.random.Generator,
) -> np.ndarray:
    """Enumerate all signs if possible; otherwise sample random signs."""
    total_patterns = _bounded_power_of_three(
        exponent=group_size,
        bound=max_list_size,
    )

    if total_patterns <= max_list_size:
        signs = np.empty((total_patterns, group_size), dtype=np.int8)

        for code in range(total_patterns):
            signs[code] = _code_to_ternary_signs(code, group_size)

        return signs

    return _random_ternary_signs(
        length=group_size,
        count=max_list_size,
        rng=rng,
    )


def _partial_sum_from_signs(
    columns: np.ndarray,
    group_indices: np.ndarray,
    signs: np.ndarray,
    modulus: int,
) -> np.ndarray:
    """Compute sum_j signs[j] * columns[group_indices[j]] modulo modulus."""
    vector_dimension = columns.shape[1]
    total = np.zeros(vector_dimension, dtype=np.int64)

    for local_index, column_index in enumerate(group_indices):
        sign = int(signs[local_index])

        if sign == 0:
            continue

        column = columns[int(column_index)]

        if sign == 1:
```

```
            for coordinate in range(vector_dimension):
                value = int(total[coordinate]) +
                    int(column[coordinate])
                total[coordinate] = value % modulus
        else:
            for coordinate in range(vector_dimension):
                value = int(total[coordinate]) -
                    int(column[coordinate])
                total[coordinate] = value % modulus

    return total


def _build_initial_vector_wagner_list(
    columns: np.ndarray,
    group_indices: np.ndarray,
    modulus: int,
    max_list_size: int,
    list_id: int,
    witness_store: WitnessStore,
    rng: np.random.Generator,
) -> tuple[VectorWagnerList, np.ndarray]:
    """Build one initial vector-valued Wagner list."""
    group_size = group_indices.shape[0]
    vector_dimension = columns.shape[1]

    signs = _enumerate_or_sample_signs(
        group_size=group_size,
        max_list_size=max_list_size,
        rng=rng,
    )

    values = np.empty((signs.shape[0], vector_dimension),
       dtype=np.int64)

    for index in range(signs.shape[0]):
        values[index] = _partial_sum_from_signs(
            columns=columns,
            group_indices=group_indices,
            signs=signs[index],
            modulus=modulus,
        )

    witness_ids = witness_store.add_leaves(
        list_id=list_id,
        count=values.shape[0],
    )

    return VectorWagnerList(values=values,
       witness_ids=witness_ids), signs
```

```
def _projection_key(
    value: np.ndarray,
    coordinate_indices: np.ndarray,
) -> tuple[int, ...]:
    """Return a hashable projection key."""
    return tuple(int(value[index]) for index in
       coordinate_indices)


def _complement_projection_key(
    value: np.ndarray,
    coordinate_indices: np.ndarray,
    modulus: int,
) -> tuple[int, ...]:
    """Return a hashable modular-complement projection key."""
    return tuple(
        0 if int(value[index]) == 0 else modulus -
           int(value[index])
        for index in coordinate_indices
    )


def _full_vector_key(value: np.ndarray) -> tuple[int, ...]:
    """Return a hashable key for a full vector."""
    return tuple(int(coordinate) for coordinate in value)


def _full_complement_vector_key(
    value: np.ndarray,
    modulus: int,
) -> tuple[int, ...]:
    """Return a hashable key for the modular complement of a
       full vector."""
    return tuple(
        0 if int(coordinate) == 0 else modulus - int(coordinate)
        for coordinate in value
    )


def _choose_filter_size(
    left_size: int,
    right_size: int,
    modulus: int,
    vector_dimension: int,
    max_output_size: int,
) -> int:
    """Choose a filter size that avoids obvious list
       starvation."""
    pair_count = left_size * right_size

    if pair_count <= max_output_size:
        return 0
```

```
519
520     filter_size = 0
521     expected_after_filter = float(pair_count)
522     target = float(max_output_size)
523
524     while filter_size < vector_dimension:
525         next_expected = expected_after_filter / float(modulus)
526
527         if next_expected < target:
528             break
529
530         expected_after_filter = next_expected
531         filter_size += 1
532
533     return filter_size
534
535
536 def _choose_coordinate_indices(
537     vector_dimension: int,
538     filter_size: int,
539     rng: np.random.Generator,
540 ) -> np.ndarray:
541     """Choose coordinate positions for one Wagner merge."""
542     if filter_size < 0:
543         raise ValueError("filter_size must be non-negative.")
544
545     if filter_size > vector_dimension:
546         raise ValueError("filter_size cannot exceed
            vector_dimension.")
547
548     if filter_size == 0:
549         return np.empty(0, dtype=np.int64)
550
551     return np.sort(
552         rng.choice(
553             vector_dimension,
554             size=filter_size,
555             replace=False,
556         ).astype(np.int64)
557     )
558
559
560 def _merge_vector_wagner_lists(
561     left_list: VectorWagnerList,
562     right_list: VectorWagnerList,
563     coordinate_indices: np.ndarray,
564     modulus: int,
565     max_output_size: int,
566     witness_store: WitnessStore,
567 ) -> VectorWagnerList:
568     """Merge two vector-valued Wagner lists using projected
        hash matching."""
```

```
    if left_list.values.shape[0] !=
       left_list.witness_ids.shape[0]:
        raise ValueError("left_list values and witness_ids
           mismatch.")

    if right_list.values.shape[0] !=
       right_list.witness_ids.shape[0]:
        raise ValueError("right_list values and witness_ids
           mismatch.")

    vector_dimension = left_list.values.shape[1]
    hash_table: dict[tuple[int, ...], list[int]] = {}

    for left_index in range(left_list.values.shape[0]):
        key = _projection_key(
            value=left_list.values[left_index],
            coordinate_indices=coordinate_indices,
        )

        if key not in hash_table:
            hash_table[key] = []

        hash_table[key].append(left_index)

    merged_values: list[np.ndarray] = []
    merged_left_ids: list[int] = []
    merged_right_ids: list[int] = []

    for right_index in range(right_list.values.shape[0]):
        key = _complement_projection_key(
            value=right_list.values[right_index],
            coordinate_indices=coordinate_indices,
            modulus=modulus,
        )

        if key not in hash_table:
            continue

        for left_index in hash_table[key]:
            merged_value = (
                left_list.values[left_index]
                + right_list.values[right_index]
            ) % modulus

            merged_values.append(merged_value)
            merged_left_ids.append(int(left_list.witness_ids[left_index]))
            merged_right_ids.append(int(right_list.witness_ids[right_index]))

            if len(merged_values) >= max_output_size:
                break

        if len(merged_values) >= max_output_size:
```

```
            break

    if not merged_values:
        return VectorWagnerList(
            values=np.empty((0, vector_dimension),
               dtype=np.int64),
            witness_ids=np.empty(0, dtype=np.int64),
        )

    merged_witness_ids = np.empty(len(merged_values),
       dtype=np.int64)

    for index, (left_id, right_id) in enumerate(
        zip(merged_left_ids, merged_right_ids, strict=True)
    ):
        merged_witness_ids[index] = witness_store.add_internal(
            left_witness=left_id,
            right_witness=right_id,
        )

    return VectorWagnerList(
        values=np.vstack(merged_values).astype(np.int64),
        witness_ids=merged_witness_ids,
    )


def _recover_z_from_vector_witness(
    witness_store: WitnessStore,
    root_witness_id: int,
    groups: Sequence[np.ndarray],
    sign_maps: Sequence[np.ndarray],
    total_dimension: int,
) -> np.ndarray:
    """Recover a ternary vector z from a root witness."""
    z = np.zeros(total_dimension, dtype=np.int64)
    leaves = witness_store.recover_leaf_indices(root_witness_id)

    for list_id, element_index in leaves:
        group_indices = groups[list_id]
        local_signs = sign_maps[list_id][element_index]

        for local_index, column_index in
           enumerate(group_indices):
            z[int(column_index)] = int(local_signs[local_index])

    return z


def _final_vector_zero_match(
    left_list: VectorWagnerList,
    right_list: VectorWagnerList,
    public_matrix: np.ndarray,
```

```
    modulus: int,
    groups: Sequence[np.ndarray],
    sign_maps: Sequence[np.ndarray],
    total_dimension: int,
    witness_store: WitnessStore,
) -> tuple[np.ndarray, int] | None:
    """Find a final exact zero match and recover the ternary
       vector."""
    hash_table: dict[tuple[int, ...], list[int]] = {}

    for left_index in range(left_list.values.shape[0]):
        key = _full_vector_key(left_list.values[left_index])

        if key not in hash_table:
            hash_table[key] = []

        hash_table[key].append(left_index)

    for right_index in range(right_list.values.shape[0]):
        target_key = _full_complement_vector_key(
            value=right_list.values[right_index],
            modulus=modulus,
        )

        if target_key not in hash_table:
            continue

        for left_index in hash_table[target_key]:
            root_witness_id = witness_store.add_internal(
                left_witness=int(left_list.witness_ids[left_index]),
                right_witness=int(right_list.witness_ids[right_index]),
            )

            z = _recover_z_from_vector_witness(
                witness_store=witness_store,
                root_witness_id=root_witness_id,
                groups=groups,
                sign_maps=sign_maps,
                total_dimension=total_dimension,
            )

            if not np.any(z):
                continue

            kernel_value = matrix_vector_mod(
                matrix=public_matrix,
                vector=z,
                modulus=modulus,
            )

            if np.all(kernel_value == 0):
                return z, root_witness_id
```

```
    return None


def negacyclic_wagner_collision_generator_int64(
    block_vectors: Sequence[Sequence[int]],
    modulus: int,
    block_size: int,
    total_dimension: int,
    number_of_blocks: int,
    list_count: int,
    max_intermediate_size: int,
    max_initial_list_size: int | None = None,
    group_strategy: Literal["random", "consecutive"] = "random",
    seed: int = 12345,
) -> NegacyclicWagnerCollisionResult | None:
    """Run the Negacyclic-Wagner-Collision-Generator.

    Args:
        block_vectors: Public block vectors a_1, ..., a_ell.
        modulus: The modulus q.
        block_size: Ring dimension n.
        total_dimension: Total dimension m = ell * n.
        number_of_blocks: Number of public blocks ell.
        list_count: Number k = 2**D of Wagner lists.
        max_intermediate_size: Maximum size of every
           intermediate list.
        max_initial_list_size: Maximum size of every initial
           list. If None,
            it is set equal to max_intermediate_size.
        group_strategy: Either "random" or "consecutive".
        seed: Random seed used for grouping and truncated
           sampling.

    Returns:
        A collision result, or None if no suitable relation is
           found.
    """
    if max_initial_list_size is None:
        max_initial_list_size = max_intermediate_size

    rng = np.random.default_rng(seed)

    normalized_blocks = _validate_negacyclic_wagner_inputs(
        block_vectors=block_vectors,
        modulus=modulus,
        block_size=block_size,
        total_dimension=total_dimension,
        number_of_blocks=number_of_blocks,
        list_count=list_count,
        max_initial_list_size=max_initial_list_size,
        max_intermediate_size=max_intermediate_size,
```

```
    )

    public_matrix = build_negacyclic_public_matrix_a(
        block_vectors=normalized_blocks,
        modulus=modulus,
    )

    columns = public_matrix.T.copy()

    groups = _make_wagner_column_groups(
        total_dimension=total_dimension,
        list_count=list_count,
        rng=rng,
        group_strategy=group_strategy,
    )

    witness_store = WitnessStore.empty()
    current: list[VectorWagnerList] = []
    sign_maps: list[np.ndarray] = []

    for list_id, group_indices in enumerate(groups):
        initial_list, signs = _build_initial_vector_wagner_list(
            columns=columns,
            group_indices=group_indices,
            modulus=modulus,
            max_list_size=max_initial_list_size,
            list_id=list_id,
            witness_store=witness_store,
            rng=rng,
        )

        if initial_list.values.shape[0] == 0:
            return None

        current.append(initial_list)
        sign_maps.append(signs)

    depth = list_count.bit_length() - 1
    vector_dimension = block_size

    for _level in range(1, depth):
        next_level: list[VectorWagnerList] = []

        for tree_index in range(len(current) // 2):
            left_list = current[2 * tree_index]
            right_list = current[2 * tree_index + 1]

            filter_size = _choose_filter_size(
                left_size=left_list.values.shape[0],
                right_size=right_list.values.shape[0],
                modulus=modulus,
                vector_dimension=vector_dimension,
```

```
                max_output_size=max_intermediate_size,
            )

            coordinate_indices = _choose_coordinate_indices(
                vector_dimension=vector_dimension,
                filter_size=filter_size,
                rng=rng,
            )

            merged_list = _merge_vector_wagner_lists(
                left_list=left_list,
                right_list=right_list,
                coordinate_indices=coordinate_indices,
                modulus=modulus,
                max_output_size=max_intermediate_size,
                witness_store=witness_store,
            )

            if merged_list.values.shape[0] == 0:
                return None

            next_level.append(merged_list)

        current = next_level

    if len(current) != 2:
        return None

    final_match = _final_vector_zero_match(
        left_list=current[0],
        right_list=current[1],
        public_matrix=public_matrix,
        modulus=modulus,
        groups=groups,
        sign_maps=sign_maps,
        total_dimension=total_dimension,
        witness_store=witness_store,
    )

    if final_match is None:
        return None

    z, root_witness_id = final_match
    z_plus, z_minus = _split_ternary_vector_to_binary_pair(z)

    return NegacyclicWagnerCollisionResult(
        z=z,
        z_plus=z_plus,
        z_minus=z_minus,
        group_count=list_count,
        root_witness_id=root_witness_id,
    )
```

Listing 2: Python implementation of the Negacyclic-Wagner-Collision-Generator