POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# C++ redesign of attack algorithms to code-based cryptography

Tesi di Laurea Magistrale in
Ingegneria Informatica
Computer Science and Engineering

Author: **Alessandro Conti**

Student ID: 10710583
Advisor: Prof. Alessandro Barenghi
Co-advisors: Prof. Gerardo Pelosi, Dr. Simone Perriello
Academic Year: 2025-26

# Abstract

The rise of quantum computing poses a threat to the most widely used public-key cryptographic systems, such as RSA and Elliptic Curve Cryptography, whose security can be compromised by quantum algorithms such as Shor's algorithm. In this context, Post-Quantum Cryptography aims to develop cryptographic primitives that are resistant to both classical and quantum attacks. Among the various families of post-quantum solutions, code-based cryptography bases its security on the computational difficulty of decoding linear codes, such as General Decoding Problem and Syndrome Decoding Problem. The Information Set Decoding algorithms represent the main class of attacks used to evaluate the practical security of such systems. However, existing analyses primarily consider the serial case, ignoring the possibility that an adversary might exploit parallel computing resources to accelerate the attack.

This thesis addresses this gap by designing and integrating the parallelization of the well-known Information Set Decoding algorithms within the CryptAttackTester framework, which provides a formalized environment for the quantitative analysis of cryptographic attacks based on a Boolean circuit computational model. In particular, a parallel variant of the `isd1` algorithm, called `isd1_parallel`, has been designed and implemented, which distributes the computation across a network of concurrent machines following Bernstein's parallel brute-force model.

An experimental campaign conducted over sets of parameters compares `isd1_parallel` against its serial counterpart in terms of depth speedup, area-time product, and security margin. The results show that, while parallelism can significantly reduce the circuit depth of an Information Set Decoding attack, the parameters maintain a robust security margin compliant with the required levels, confirming that the attack complexity remains exponential even under the parallel circuit model.

**Keywords:** post-quantum cryptography; code-based cryptography; information set decoding; parallelization; CryptAttackTester

# Abstract in lingua italiana

L'avvento dell'informatica quantistica rappresenta una minaccia per i sistemi crittografici a chiave pubblica più diffusi, come RSA e Elliptic Curve Cryptography, la cui sicurezza può essere compromessa da algoritmi quantistici quali l'algoritmo di Shor. In questo contesto, la Post-Quantum Cryptography mira a sviluppare primitive crittografiche resistenti ad attacchi sia classici che quantistici. Tra le varie famiglie di soluzioni post-quantistiche, la crittografia basata su codici fonda la propria sicurezza sulla difficoltà computazionale di decodificare codici lineari, quali General Decoding Problem e Syndrome Decoding Problem. Gli algoritmi Information Set Decoding rappresentano la principale classe di attacchi per valutare la sicurezza pratica di tali sistemi. Tuttavia, le analisi esistenti considerano prevalentemente il caso seriale, ignorando la possibilità che un avversario sfrutti risorse di calcolo parallele per accelerare l'attacco.

La presente tesi colma questa lacuna progettando e integrando la parallelizzazione dei ben noti algoritmi Information Set Decoding all'interno del framework CryptAttackTester, che fornisce un ambiente formalizzato per l'analisi quantitativa degli attacchi crittografici basata su un modello computazionale a circuiti booleani. In particolare, è stata progettata e implementata una variante parallela dell'algoritmo `isd1`, denominata `isd1_parallel`, che distribuisce il calcolo su una rete di macchine concorrenti seguendo il modello di forza bruta parallela di Bernstein.

Una campagna sperimentale condotta su diverse combinazioni di parametri mette a confronto `isd1_parallel` con la sua controparte seriale in termini di accelerazione della profondità, prodotto area-tempo e margine di sicurezza. I risultati mostrano che, sebbene il parallelismo possa ridurre significativamente la profondità del circuito di un attacco Information Set Decoding, i parametri mantengono un margine di sicurezza robusto e conforme ai livelli richiesti, confermando che la complessità dell'attacco rimane esponenziale anche nel modello di circuito parallelo.

**Parole chiave:** crittografia post-quantistica; crittografia basata su codici; information set decoding; parallelizzazione; CryptAttackTester

# Contents

# Introduction

In recent years, the rapid advances in the field of quantum computing have posed a real and imminent threat to the security of current cryptographic infrastructures. The most widely used public-key systems today, such as Rivest, Shamir, Adleman (RSA) [39] and Elliptic Curve Cryptography (ECC) [33], base their security on the computational difficulty of factoring large integers or computing discrete logarithms over finite groups. However, there are quantum algorithms capable of solving these problems in polynomial time, the best known of which is Shor's algorithm [44], effectively rendering these systems vulnerable in the event of an adversary possessing a sufficiently powerful quantum computer. In addition to this direct threat, there is the paradigm known as harvest now, decrypt later, whereby an adversary with substantial storage capacity can intercept and store communications encrypted with classical algorithms today, with the aim of decrypting them retrospectively in the future as soon as they have access to suitable quantum hardware. This prospect makes the immediate adoption of quantum-resistant cryptosystems a matter of urgency.

In this scenario, the scientific community has turned its attention to Post-Quantum Cryptography (PQC), that is, the study of cryptographic systems resistant to attacks from both classical and quantum computers. Among the families of post-quantum cryptosystems, code-based cryptography is one of the most promising and widely studied. The security of these cryptosystems, the forerunner of which is the McEliece algorithm [32], is based on the computational difficulty of solving the General Decoding Problem (GDP) or the Syndrome Decoding Problem (SDP) [6], problems for which no quantum algorithms are yet known that can significantly reduce the complexity of an attack. Although the keys used by these cryptosystems are of large size, these systems are among the strongest candidates in the standardization process initiated by National Institute of Standards and Technology (NIST) for PQC.

The security of a cryptosystem does not depend solely on the theoretical difficulty of the underlying problem, but also on its practical resistance to the best-known attack algorithms. The practical security of the parameters chosen for these cryptosystems is assessed by measuring the effectiveness of the best-known attack algorithms currently

available, which belong to the Information Set Decoding (ISD) family. These algorithms were introduced by Prange [38] and progressively refined over the decades; they aim to solve the SDP more efficiently than exhaustive search, and their analysis is fundamental to determining the security parameters of code-based cryptosystems. However, most existing analyses of these algorithms consider only the serial case, ignoring the possibility that an adversary may have distributed computational resources and could exploit parallelism to accelerate the attack.

This thesis forms part of this broader context of security evaluation, with the aim of assessing whether, and to what extent, the introduction of parallelism can reduce the execution times of ISD algorithms. In particular, the aim is to integrate parallelism within the CryptAttackTester (CAT) [9] framework, which provides a formalized environment for the quantitative analysis of cryptographic attacks based on a Boolean circuit computation model. Within this framework, a parallel version of the serial algorithm `isd1`, named `isd1_parallel`, has been designed and integrated; this attack algorithm distributes the computation of certain phases across a network of concurrent machines, following the approach proposed by Bernstein in [8]. To compare the implemented parallel algorithm with its serial counterpart in CAT, an experimental campaign was conducted using five sets of candidate parameters from the NIST, which allowed for a direct comparison of the performance of these two algorithms.

The thesis is organized into five chapters, structured as follows. Chapter 1 provides the theoretical foundations necessary for understanding the work: the fundamentals of cryptography, the family of two-dimensional sorting networks used in the parallel implementation, the concept of [8], coding theory, code-based cryptosystems and their main applications (the cryptosystems of McEliece [32] and Niederreiter [34]), and the CAT framework with its computational model and architecture. Chapter 2 describes the ISD algorithms implemented in CAT. It describes the high-level structure common to all variants, and the three variants present in the framework (`isd0`, `isd1` and `isd2`), which differ from one another in the search phase. These three algorithms implement variants of the ISD attacks by Prange [38], Lee-Brickell [27], Leon [28], Ball-Collision [12], Dumer [17], Stern [46], May-Meurer-Thomae (MMT) [31] and Becker-Joux-May-Meurer (BJMM) [5]. Chapter 3 presents the main contribution of the thesis: the design and implementation of the parallel version of the `isd1` attack, named `isd1_parallel`, within CAT. The chapter analyses the opportunities for parallelization in the ISD algorithms, describing the parallelization strategy adopted and detailing the implementation of the three phases that make up the parallel version of the algorithm. Chapter 4 presents the experimental campaign for `isd1_parallel`, comparing it with its original serial counterpart, `isd1`. The chapter

describes the methodology and approach adopted during the experiment, the strategy for selecting the parameters to be tested, and the results obtained for each configuration of the tested parameters. Chapter 5 draws the conclusions of the work, summarizing the main results and outlining possible directions for future research.

# 1 | Background and State-of-the-Art

This chapter will provide the theoretical foundations necessary for understanding this work. First, the notation and typographical conventions used in this work will be presented. The cryptographic fundamentals will then be described, covering both private-key and public-key systems. Next, we will cover the essential concepts of coding theory that underpin code-based cryptosystems, code-based cryptographic systems and their main examples, namely the McEliece [32] and Niederreiter [34] cryptosystems. Attention will then shift to the family of two-dimensional sorting networks relevant to the parallel implementation developed in this thesis. Finally, the CAT framework [9] will be described, along with its computational model, architecture and the problem underlying ISD attacks in this framework.

## 1.1. Notation

This section defines the symbols and typographical conventions used in this work. Unless otherwise specified, the rules listed below apply.

Scalar variables are denoted by lowercase roman letters (e.g., $a$), while random variable are denoted by calligraphic-faced uppercase roman letters (e.g., $\mathcal{A}$).

Vectors are lowercase boldface roman letters (e.g., $\mathbf{v}$), a subvector is denoted using the same notation as the vector, with the indices specifying the selected elements written as subscripts (e.g., $\mathbf{v}_{\{i,\dots,j\}}$, $\mathbf{v}_{\{i,j,v\}}$), while an element of the vector is represented by lowercase roman letters with the index written as a subscript (e.g., $v_i$). Unless otherwise stated, vectors are assumed to be row vectors; consequently, column vectors are denoted by the use of the transpose operator (e.g., $\mathbf{v}^\intercal$).

Matrices are uppercase roman letters (e.g., $M$), a row of the matrix is represented as a vector, written in lowercase boldface roman letters, with the row index as a subscript followed by a colon indicating all the columns in the row (e.g., $\mathbf{m}_{(i,:)}$), a column of the

matrix is represented as a vector, written in lowercase boldface roman letters, with the column index as a subscript followed by a colon indicating all the rows in the column (e.g., $\mathbf{m}^{\intercal}_{(:,j)}$), a submatrix is denoted using the same notation as the matrix, with indices specifying the selected rows and columns (e.g., $M_{(\{i,\dots,j\},\{k,\dots,w\})}$, $M_{(\{i,j\},k)}$), while an element of the matrix is represented by lowercase roman letters with the index written as a subscript (e.g., $m_{(i,j)}$). Unless otherwise specified, the identity matrix is denoted by the uppercase letter I with the dimension as a subscript (e.g., $I_m$).

Sets are denoted by boldface uppercase roman letters (e.g., $\mathbf{S}$), while notable sets are employed as blackboard-bold typeface (e.g., $\mathbb{Z}$, $\mathbb{F}_q$).

Functions are lowercase roman letters (e.g., $f$).

## 1.2. Fundamentals of Cryptography

Cryptography is the practice of using techniques to ensure the security of communications between two parties over an insecure channel, a channel that can be intercepted and tampered with by a malicious party. It provides the mathematical and algorithmic foundations necessary to ensure the confidentiality, integrity, authentication, and non-repudiation of messages.

To this end, cryptographic schemes transform a plaintext message, readable by anyone, into a ciphertext message, which is unreadable to anyone not authorized to read it. These schemes fall into two categories: *private-key systems*, also known as symmetric-key systems, and *public-key systems*, also known as asymmetric-key systems.

### 1.2.1. Private-Key Cryptosystem

A private-key cryptosystem, also known as a symmetric-key cryptosystem, is based on the assumption that the two parties involved in the communication, traditionally referred to as Alice and Bob, have previously agreed on a secret key, denoted as $k \in \mathbf{K}$. This key is the fundamental element used to encrypt and decrypt the messages exchanged between the two parties.

When Alice needs to send a message, $m \in \mathbf{M}$, to Bob, she must encrypt the plaintext message. To do so, she uses the encryption function (`encryption`), which, using the shared key $k$, transforms the plaintext message $m$ into ciphertext $c = \texttt{encryption}(m,k) \in \mathbf{C}$. Now that the message $m$ has been transformed into the ciphertext $c$, it can be transmitted over an insecure channel.

Once Bob receives a message from Alice, he must recover the plaintext message, $m$, from the received ciphertext, $c$. To derive the plaintext from the ciphertext, Bob must apply the decryption function (`decryption`), which, using the same key $k$, transforms the ciphertext into the plaintext sent by Alice, $m = \texttt{decryption}(c, k) \in \mathbf{M}$.

For the cryptosystem to be correct, the *consistency property* must hold for every key $k$ and message $m$, $\texttt{decryption}(\texttt{encryption}(m, k),\ k) = m$.

The security level of private-key cryptosystems depends strictly on the bit length of the secret key $k \in \mathbf{K}$ shared between the parties. In modern contexts, to ensure an adequate level of security against brute-force attacks, it is recommended to use keys with a length of $\lambda \in \{128, 192, 256\}$ bits. This implies that an attacker would need to make an average of $2^{\lambda}$ attempts before recovering the correct key.

Among the most efficient and secure private-key cryptosystems currently available are Advanced Encryption Standard (AES) [36] and ChaCha20 [7]. Over the years, both of these cryptosystems have been subjected to numerous cryptanalytic attacks and have demonstrated excellent security levels; for this reason, they are widely used in numerous practical applications.

### 1.2.2. Public-Key Cryptosystem

A public-key cryptosystem, also known as an asymmetric cryptosystem, is based on the use of a pair of distinct keys for each party involved in the communication. Unlike private-key cryptosystems, this type of cryptosystem allows for secure communication between parties without the need to have previously agreed upon and shared a secret key.

The two keys required for public-key cryptosystems are the private-key $k_{private}$ and the public-key $k_{public}$. The key pair must be generated independently by each party. The private-key is used exclusively by its owner and must not be disclosed to anyone, while the public-key must be made available to the other participants in the communication.

When Alice needs to send a message, $m \in \mathbf{M}$, to Bob, she must first obtain Bob's public-key, $k^B_{public}$, and use it to encrypt the message. To encrypt the message $m$, Alice uses the encryption function (`encryption`), which transforms the plaintext message into ciphertext $c = \texttt{encryption}(m, k^B_{public}) \in \mathbf{C}$. Now that the message $m$ has been transformed into the ciphertext $c$, it can be transmitted over an insecure channel.

Once Bob receives a message from Alice, he must recover the plaintext message, $m$, from the received ciphertext, $c$. To derive the plaintext from the ciphertext, Bob must apply the decryption function (`decryption`), which, using his private-key, transforms the ciphertext

into the plaintext message sent by Alice, $m = \texttt{decryption}(c, k^B_{private}) \in \mathbf{M}$.

For the cryptosystem to be correct, the *consistency property* must hold for every key pair $(k_{private}, k_{public})$ and message $m$, $\texttt{decryption}(\texttt{encryption}(m, k_{public}),\ k_{private}) = m$.

The security of public-key cryptosystems does not depend exclusively on the length of the key, but rather on the computational difficulty of specific mathematical problems. In particular, this family of cryptosystems uses *one-way trapdoor functions*, that is, functions that are easy to compute in one direction but computationally difficult to invert without additional information, represented by the private-key. Among the main mathematical problems used as the basis for public-key cryptosystems are the factorization of large integers, on which the RSA [39] cryptosystem is based, the discrete logarithm problem over finite groups used by the Diffie-Hellman [15] and ElGamal [20] cryptosystems, or the discrete logarithm problem over elliptic curves used by ECC [33], which allows for high levels of security with small key sizes.

The security of these cryptosystems stems from the computational complexity required to solve the problems listed above, for which no efficient polynomial-time algorithms are yet known on classical computers. Consequently, an attacker attempting to derive the private-key from the public-key faces a computationally infeasible problem with current technologies.

Public-key cryptosystems are widely used in numerous modern security protocols; however, due to their greater computational complexity compared to private-key cryptosystems, they are often used in combination with the latter in hybrid schemes. In such schemes, public-key cryptography is employed to securely establish a shared symmetric key, which is subsequently used to efficiently encrypt messages.

**Security Notions** In modern cryptography, the security of a public-key cryptosystem is formally evaluated against specific adversary models and goals. Among these, the capability of an adversary to withstand inversion attacks under active monitoring is fundamental.

**Definition 1.2.1** (One-Wayness under Chosen Plaintext Attack). *The One-Wayness under Chosen Plaintext Attack (OW-CPA) is a security concept relating to public-key cryptosystems. An attacker knows the public-key and the ciphertext $c \in \mathbf{C}$ of a message $m \in \mathbf{M}$ generated randomly using the public-key. The attacker's objective is to find the message $m$ that generated the ciphertext $c$.*

## 1.3. Sorting Network

A sorting network is a fixed-structure comparison network that sorts a sequence of $n$ elements using a predetermined set of Compare-and-Swap (CAS) operations, regardless of the values of the input elements. A CAS operation consists of comparing two elements, $a$ and $b$, located respectively at the $i$-th and $j$-th positions in the input sequence, and swapping the two elements if the comparison is positive. Unlike traditional sorting algorithms, where the execution flow of the algorithm depends on the results of comparisons between elements, sorting networks always execute the same predetermined sequence of CAS operations, without ever skipping one based on the value of the elements.

Formally, a sorting network operates on $n$ wires, each of which carries an element, and uses a CAS operations between two wires. The sequence of CAS is fixed, determined in advance for a sequence of size $n$, and completely defines the behavior of the network. A sorting network is correct if, for every permutation of the $n$ input elements, it produces an ordered sequence as output. To verify the correctness of a sorting network, the 0-1 *Principle* [23] is used: it suffices to verify that the network can correctly sort all $2^n$ binary sequences of length $n$ to determine that it correctly sorts any sequence of $n$ elements.

The two fundamental metrics for evaluating a sorting network are: the total number of CAS required to implement the network, $C(n)$, and the depth of the network, i.e. the maximum number of sequential CAS operations along any path from the first to the last operation, $T(n)$.

This work focuses on *two-dimensional sorting networks*, designed to operate on a mesh of $n \times n$ concurrent machines, where each machine communicates exclusively with its four immediate neighbors (north, south, west and east). This structure lends itself naturally to exploiting the intrinsic parallelism of mesh architectures, in which CAS operations between adjacent machines can be executed simultaneously.

For these architectures, the theoretical lower bound on depth is $T(n^2) \geq 2n - 2$ steps, i.e. the minimum number of moves required to move an element from the bottom-right corner to the top-left corner of the mesh. This number was subsequently improved by Kunde [25], who that any sorting network on an $n \times n$ mesh must satisfy

$$T\left(n^2\right) \geq 3n - \sqrt{2n} - 3 \tag{1.1}$$

meaning that no algorithm, however optimized, can sort an $n \times n$ mesh in fewer steps than this quantity establishes.

Table 1.1: Complexity of two-dimensional sorting networks

| **Sorting Network** | $C(n^2)$ | $T(n^2)$ |
|---|---|---|
| 2D odd-even transposition sort | $\mathcal{O}(n^4)$ | $\mathcal{O}(n^2)$ |
| Shearsort | $\frac{n^3 \log_2 n}{2} + \frac{3}{2}n^3 - \frac{n^2 \log_2 n}{2} - \frac{5}{2}n^2 + n$ | $n \log_2 n + 3n - 2$ |
| Rotatesort | $\frac{9}{2}n^3 + \frac{3}{2}n^{\frac{5}{2}} - \frac{9}{2}n^2 - \frac{3}{2}n^{\frac{3}{2}} + 19n$ | $10n + 5\sqrt{n} + 11$ |
| 4-way mergesort | $4n^3 - 2n^2 \log_2 n - 4n^2$ | $7n - 6$ |
| LS3-sort | $4n^3 - \frac{1}{2}n^2 \log_2 n - 4n^2$ | $9n - 9$ |
| 3n-sort of Schnorr and Shamir | $n^3 + \frac{25}{2}n^{\frac{11}{4}} - \frac{9}{2}n^2 \log_2 n - 13n^2$ | $3n + 22n^{\frac{3}{4}} - 18$ |

Table 1.1 provides an overview of the computational complexity of all sorting network discussed in this section, in terms of number of CAS $C(\cdot)$ and depth $T(\cdot)$.

### 1.3.1. 2D Odd-Even Transposition Sort

The 2D odd-even transposition sort is the simplest algorithm for sorting two-dimensional arrays, but it is very slow, $T(n^2) = \mathcal{O}(n^2)$. The idea of Odd-Even Transposition Sort (OETS) [23] is generalized to two-dimensional arrays in a straightforward way. The elements are not just compared with their right and left neighbors, but also with their upper and lower neighbors.

In the one-dimensional algorithm, $m$ elements are arranged across $m$ comparator stages. In each comparator stage, all inputs in odd-numbered positions, or all those in even-numbered positions, are compared with their neighbors; the odd and even comparator stages alternate, requiring $C(m) = \frac{m(m-1)}{2}$ comparators and $T(m) = \frac{m(m-1)}{2}$ steps.

In the two-dimensional version, the algorithm operates on a $n \times n$ mesh and executes a fixed sequence of phases, alternating between row steps and column steps. In a row step, each row independently performs a stage of OETS between horizontally adjacent elements; whereas in a column step, each column independently performs a stage of OETS between vertically adjacent elements. Within each row or column step, odd and even OETS step alternate as in one-dimensional case. Applying the one-dimensional complexity to each

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the sorted matrix

$M' \leftarrow M$
**for** $r \leftarrow 0$ **to** $n^2 - 1$ **do**
  **for** $i \leftarrow 0$ **to** $n - 1$ **do**
    **if** $i \bmod 2 = 0$ **then**
      $\mathbf{m'}_{(i,:)} \leftarrow$ `oetsOddStep(`$\mathbf{m'}_{(i,:)}$`, ASC) // apply OETS to the row in ascending order`
      $\mathbf{m'}_{(i,:)} \leftarrow$ `oetsEvenStep(`$\mathbf{m'}_{(i,:)}$`, ASC) // apply OETS to the row in ascending order`
    **else**
      $\mathbf{m'}_{(i,:)} \leftarrow$ `oetsOddStep(`$\mathbf{m'}_{(i,:)}$`, DESC) // apply OETS to the row in descending order`
      $\mathbf{m'}_{(i,:)} \leftarrow$ `oetsEvenStep(`$\mathbf{m'}_{(i,:)}$`, DESC) // apply OETS to the row in descending order`
  **for** $j \leftarrow 0$ **to** $n - 1$ **do**
    $\mathbf{m'}^{\mathsf{T}}_{(:,j)} \leftarrow$ `oetsOddStep(`$\mathbf{m'}^{\mathsf{T}}_{(:,j)}$`, ASC)` $\mathbf{m'}^{\mathsf{T}}_{(:,j)} \leftarrow$ `oetsEvenStep(`$\mathbf{m'}^{\mathsf{T}}_{(:,j)}$`, ASC)`
**return** $M'$

**Algorithm 1.1:** 2D odd-even transposition sort algorithm

of the $n$ rows and $n$ columns, the total number of comparators is $C(n^2) = \mathcal{O}(n^4)$, while the depth is $T(n^2) = \mathcal{O}(n^2)$. The full procedure is detailed in Algorithm 1.1.

### 1.3.2. Shearsort

Shearsort [40] is a two-dimensional sorting network that alternates between row-sorting and column-sorting phases, requiring a total of $\log_2 n$ iterations. Each iteration has two phases: the row sorting phase followed by the column sorting phase. Each row is sorted independently using OETS [23]; even-numbered rows are sorted in ascending order (from left to right) whilst odd-numbered rows are sorted in descending order (from right to left); conversely, each column is sorted independently using OETS in ascending order. After $\log_2 n$ such iterations, at most one dirty row remains, which is resolved by a final additional row-sorting step. The full procedure is detailed in Algorithm 1.2.

After sorting the rows in the first stage, we obtain $\frac{n}{2}$ rows sorted left-to-right and $\frac{n}{2}$ rows sorted right-to-left. In the column-sorting stage, each pair of these sorted rows is merged into at least one correct row. Therefore, after the first iteration, the mesh consists of some correct rows and an unsorted region comprising at most $\frac{n}{2}$ rows. In the second iteration, every two rows of the unsorted area are again combined into at least one correct row. Therefore, after the second iteration, the unsorted area consists of at most $\frac{n}{4}$ rows, and so on. After $\log_2 n$ iterations, at most one row may remain that is not completely ordered.

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the sorted matrix

$M' \leftarrow M$
**for** $k \leftarrow 0$ **to** $\log_2 n - 1$ **do**
  **for** $i \leftarrow 0$ **to** $n - 1$ **do**
    **if** $i \bmod 2 = 0$ **then**
      $\mathbf{m'}_{(i,:)} \leftarrow$ `sort(`$\mathbf{m'}_{(i,:)}$`, ASC) // sort the row in ascending order`
    **else**
      $\mathbf{m'}_{(i,:)} \leftarrow$ `sort(`$\mathbf{m'}_{(i,:)}$`, DESC) // sort the row in descending order`
  **for** $j \leftarrow 0$ **to** $n - 1$ **do**
    $\mathbf{m'}^{\mathsf{T}}_{(:,j)} \leftarrow$ `sort(`$\mathbf{m'}^{\mathsf{T}}_{(:,j)}$`, ASC)`
**for** $i \leftarrow 0$ **to** $n - 1$ **do**
  **if** $i \bmod 2 = 0$ **then**
    $\mathbf{m'}_{(i,:)} \leftarrow$ `sort(`$\mathbf{m'}_{(i,:)}$`, ASC) // sort the row in ascending order`
  **else**
    $\mathbf{m'}_{(i,:)} \leftarrow$ `sort(`$\mathbf{m'}_{(i,:)}$`, DESC) // sort the row in descending order`
**return** $M'$

**Algorithm 1.2:** Shearsort algorithm

In the final additional step, the last remaining row is ordered, and the entire mesh is then ordered.

In each iteration, the height of the unsorted region is halved. This means that sorting a column containing an unsorted region of length $k$ requires $k$ steps of the OETS. Summing over all $\log_2 n$ iterations, the total number of steps for column sorting is $n + \frac{n}{2} + \cdots + 2 = 2n - 2$. Adding the $n \log_2 n$ steps for row sorting and the final $n$ steps yields Equation 1.2.

$$T(n^2) = n \log_2 n + 3n - 2 \tag{1.2}$$

and the number of comparators required:

$$C(n^2) = \frac{n^3 \log_2 n}{2} + \frac{3}{2}n^3 - \frac{n^2 \log_2 n}{2} - \frac{5}{2}n^2 + n \tag{1.3}$$

which is given by the product $T(n^2)\frac{n(n-1)}{2}$.

### 1.3.3. Rotatesort

Rotatesort [30] is a two-dimensional sorting network based on three basic operations: `balance`, `unblock` and `shear`. The $n \times n$ mesh is divided into vertical slices of size $n \times \sqrt{n}$, horizontal slices of size $\sqrt{n} \times n$ and blocks of size $\sqrt{n} \times \sqrt{n}$. Unless otherwise specified, all sorting of rows and columns uses OETS [23], which requires $\frac{m(m-1)}{2}$ comparators to

**Input:** $S \in \mathbb{Z}^{k \times \sqrt{k}}$: the unsorted vertical slice
**Output:** $S' \in \mathbb{Z}^{k \times \sqrt{k}}$: the sorted vertical slice
**Data:** $T \in \mathbb{Z}^{k \times \sqrt{k}}$: the temporary matrix used to rotate the rows
$j_{orig} \in \mathbb{Z}$: the final position in the column where the element is to be inserted during the row sorting phase

$S' \leftarrow S$
**for** $j \leftarrow 0$ **to** $\sqrt{k} - 1$ **do**
  $\mathbf{s'}^{\intercal}_{(:,j)} \leftarrow \texttt{sort}(\mathbf{s'}^{\intercal}_{(:,j)})$
$T \leftarrow S'$
**for** $i \leftarrow 0$ **to** $k - 1$ **do**
  **for** $j \leftarrow 0$ **to** $\sqrt{k} - 1$ **do**
    $j_{orig} \leftarrow \left(j - (i \bmod \sqrt{k}) + \sqrt{k}\right) \bmod \sqrt{k}$
    $s'_{(i,j)} \leftarrow t_{(i,j_{orig})}$
**for** $j \leftarrow 0$ **to** $\sqrt{k} - 1$ **do**
  $\mathbf{s'}^{\intercal}_{(:,j)} \leftarrow \texttt{sort}(\mathbf{s'}^{\intercal}_{(:,j)})$
**return** $S'$

**Algorithm 1.3:** Balance algorithm

sort an array of size $m$.

The `balance` operation is applied to the vertical slices, $n \times \sqrt{n}$, and consists of three steps: sorting the columns, rotating the rows (i.e. each row $i$ is rotated by $i \bmod \sqrt{n}$ positions), and sorting the columns again. Sorting the columns takes $n$ steps and requires $\sqrt{n}\frac{n(n-1)}{2}$ comparators to sort a vertical slice, but as there are $\sqrt{n}$ vertical slices, so $\sqrt{n}\left(\sqrt{n}\frac{n(n-1)}{2}\right)$ are required. Rotating the rows takes $\sqrt{n}$ steps to complete and requires no comparators, as it is simply a predefined permutation. Finally, sorting the columns takes $n$ steps and requires a total of $\sqrt{n}$ rows, $\sqrt{n}\left(\sqrt{n}\frac{n(n-1)}{2}\right)$ comparators, as before. The function is shown in detail in Algorithm 1.3.

The `unblock` operation is applied to the entire mesh, $n \times n$, and serves to distribute the elements of each block across the full width of the mesh; it consists of two steps: rotating the rows, where each row $i$ is rotated by $i\sqrt{n} \bmod n$ positions, and sorting the columns. Rotating the rows requires $\frac{n}{2}$ steps but no comparators, as it involves applying a permutation, whereas sorting the columns requires $n$ steps and $n\frac{n(n-1)}{2}$ comparisons. The function is shown in detail in Algorithm 1.4.

The final operation, `shear`, is also applied to the entire mesh, and first sorts the rows in alternating directions (i.e. even-numbered rows in ascending order and odd-numbered rows in descending order), and then sorts the columns, as shown in Algorithm 1.5.

Rotatesort uses the three basic operations to sort the entire mesh: it applies `balance` to

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the matrix after unblock operation
**Data:** $T \in \mathbb{Z}^{n \times n}$: the temporary matrix used to rotate the rows
$j_{orig} \in \mathbb{Z}$: the final position in the column where the element is to be inserted during the row sorting phase

$M' \leftarrow M$
$T \leftarrow M'$
**for** $i \leftarrow 0$ **to** $n-1$ **do**
  **for** $j \leftarrow 0$ **to** $n-1$ **do**
    $j_{orig} \leftarrow (j - (i\sqrt{n} \bmod n) + n) \bmod n$
    $m'_{(i,j)} \leftarrow t_{(i, j_{orig})}$
**for** $j \leftarrow 0$ **to** $n-1$ **do**
  $\mathbf{m'}^{\intercal}_{(:,j)} \leftarrow \texttt{sort}(\mathbf{m'}^{\intercal}_{(:,j)})$
**return** $M'$

**Algorithm 1.4:** Unblock algorithm

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the matrix after shear operation

$M' \leftarrow M$
**for** $i \leftarrow 0$ **to** $n-1$ **do**
  **if** $i \bmod 2 = 0$ **then**
    $\mathbf{m'}_{(i,:)} \leftarrow \texttt{sort}(\mathbf{m'}_{(i,:)}, \textit{ASC})$ `// sort the row in ascending order`
  **else**
    $\mathbf{m'}_{(i,:)} \leftarrow \texttt{sort}(\mathbf{m'}_{(i,:)}, \textit{DESC})$ `// sort the row in descending order`
**for** $j \leftarrow 0$ **to** $n-1$ **do**
  $\mathbf{m'}^{\intercal}_{(:,j)} \leftarrow \texttt{sort}(\mathbf{m'}^{\intercal}_{(:,j)}, \textit{ASC})$
**return** $M'$

**Algorithm 1.5:** Shear algorithm

the vertical slices, applies `unblock`, applies `balance` to the horizontal slices (treating it as if it were a vertical slice, i.e., operating on its transposition), applies `unblock` again, performs the `shear` operation three times. The full procedure is detailed in Algorithm 1.6.

Applying the `shear` operation three times requires $3n$ steps and $3n\frac{n(n-1)}{2}$ comparators to sort the rows and columns; since only a constant number of unsorted rows remain, the number of steps required is $6 + 3 + 2$, which corresponds to $n\left(\frac{6(6-1)}{2} + \frac{3(3-1)}{2} + \frac{2(2-1)}{2}\right)$ comparators. Finally sorts all remaining rows that requires $n$ steps and $n\frac{n(n-1)}{2}$ comparators.

The number of steps required to fully sort an $n \times n$ mesh using this sorting network is

$$T\left(n^2\right) = 10n + 5\sqrt{n} + 11 \tag{1.4}$$

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the sorted matrix
**Data:** $s \in \mathbb{Z}$: the first column of the slice
$e \in \mathbb{Z}$: the last column of the slice
$T \in \mathbb{Z}^{n \times \sqrt{n}}$: the temporary matrix used to store the result of the `balance` operation on the rows (`balance` operates on vertical slices, so horizontal slices must be transposed)

$M' \leftarrow M$
**for** $f \leftarrow 0$ **to** $\sqrt{n} - 1$ **do**
  $s \leftarrow f \cdot \sqrt{n}$
  $e \leftarrow s + \sqrt{n} - 1$
  $M'_{(:,\, s,\ldots,e)} \leftarrow \texttt{balance}(M'_{(:,\, s,\ldots,e)})$
$M' \leftarrow \texttt{unblock}(M')$
**for** $f \leftarrow 0$ **to** $\sqrt{n} - 1$ **do**
  $s \leftarrow f \cdot \sqrt{n}$
  $e \leftarrow s + \sqrt{n} - 1$
  $T \leftarrow \texttt{balance}(M'^{\intercal}_{(s,\ldots,e,\,:)})$
  $M'_{(s,\ldots,e,\,:)} \leftarrow T^{\intercal}$
$M' \leftarrow \texttt{unblock}(M')$
$M' \leftarrow \texttt{shear}(M')$
$M' \leftarrow \texttt{shear}(M')$
$M' \leftarrow \texttt{shear}(M')$
**for** $i \leftarrow 0$ **to** $n - 1$ **do**
  $\mathbf{m}'_{(i,\,:)} \leftarrow \texttt{sort}(\mathbf{m}'_{(i,\,:)})$
**return** $M'$

**Algorithm 1.6:** Rotatesort algorithm

The number of comparators required by this sorting network, using OETS as the underlying sorting primitive for both rows and columns, is

$$C(n^2) = \frac{9}{2}n^3 + \frac{3}{2}n^{\frac{5}{2}} - \frac{9}{2}n^2 - \frac{3}{2}n^{\frac{3}{2}} + 19n \tag{1.5}$$

### 1.3.4. 4-Way Mergesort

The 4-way mergesort [41] is a two-dimensional sorting network that recursively uses a `4wayMerge` procedure to sort an $n \times n$ mesh. The network is based on the concept of roughly sorted mesh.

**Definition 1.3.1** (Roughly Sorted Array). *An $n \times n$ array is considered roughly sorted if it is sufficient to sort the rows to obtain a fully sorted mesh, i.e. if each element of the mesh is already in the correct row within the array.*

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the sorted matrix

$M' \leftarrow M$
$M' \leftarrow$ `roughsort`$(M')$
**for** $i \leftarrow 0$ **to** $n-1$ **do**
  $\mathbf{m'}_{(i,:)} \leftarrow$ `sort`$(\mathbf{m'}_{(i,:)})$
**return** $M'$

**Algorithm 1.7:** 4-way mergesort algorithm

**Input:** $M \in \mathbb{Z}^{k \times k}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{k \times k}$: the roughly sorted matrix

$M' \leftarrow M$
**if** $k > 1$ **then**
  $M'_{\left(0,\ldots,\frac{k}{2}-1,\,0,\ldots,\frac{k}{2}-1\right)} \leftarrow$ `roughsort`$(M'_{\left(0,\ldots,\frac{k}{2}-1,\,0,\ldots,\frac{k}{2}-1\right)})$
  $M'_{\left(0,\ldots,\frac{k}{2}-1,\,\frac{k}{2},\ldots,k-1\right)} \leftarrow$ `roughsort`$(M'_{\left(0,\ldots,\frac{k}{2}-1,\,\frac{k}{2},\ldots,k-1\right)})$
  $M'_{\left(\frac{k}{2},\ldots,k-1,\,0,\ldots,\frac{k}{2}-1\right)} \leftarrow$ `roughsort`$(M'_{\left(\frac{k}{2},\ldots,k-1,\,0,\ldots,\frac{k}{2}-1\right)})$
  $M'_{\left(\frac{k}{2},\ldots,k-1,\,\frac{k}{2},\ldots,k-1\right)} \leftarrow$ `roughsort`$(M'_{\left(\frac{k}{2},\ldots,k-1,\,\frac{k}{2},\ldots,k-1\right)})$
  $M' \leftarrow$ `4wayMerge`$(M')$
**return** $M'$

**Algorithm 1.8:** Roughsort algorithm

The 4-way mergesort algorithm uses the `roughsort` function to bring the mesh into a roughly sorted state, and then simply sorts the rows of the mesh to achieve the complete sort, as shown in Algorithm 1.7.

The `roughsort` function, shown in Algorithm 1.8, brings an $n \times n$ mesh to a roughly sorted state by recursively calling itself on the four sub-blocks, and then merging the roughly sorted sub-blocks using the `4wayMerge` function.

Given an $n \times n$ mesh with the four $\frac{n}{2} \times \frac{n}{2}$ sub-quadrants roughly sorted, the `4wayMerge` function produces a fully roughly sorted $n \times n$ mesh by applying the following steps in sequence: it sorts the rows of each sub-quadrant (in ascending order for the top two quadrants and descending order for the bottom two), sorts the columns of the entire mesh, sorts the rows (odd-numbered rows in ascending order and even-numbered rows in descending order), and finally sorts the columns again, as shown in Algorithm 1.9.

Here are the details of the number of steps and comparators required to implement the `4wayMerge` function for a mesh $k \times k$. The first step is to sort the rows of each sub-block; a total of $\frac{k}{2}$ steps and $2k\frac{\frac{k}{2}\left(\frac{k}{2}-1\right)}{2}$ comparators are required; to sort the columns, $k$ steps and $k\frac{k(k-1)}{2}$ comparators are required; to sort the rows alternately, $k$ steps and $k\frac{k(k-1)}{2}$ comparators are required; finally, to sort the columns again, only $\frac{n}{2}$ steps are required,

**Input:** $M \in \mathbb{Z}^{k \times k}$: the matrix, whose four $\frac{k}{2} \times \frac{k}{2}$-submatrices are roughly sorted
**Output:** $M' \in \mathbb{Z}^{k \times k}$: the roughly sorted matrix

```
1  M' ← M
2  for i ← 0 to k − 1 do
3  |  if i < k/2 then
4  |  |  M'_(i,0,...,k/2−1) ← sort(M'_(i,0,...,k/2−1), ASC) // sort the rows of the
   |  |     submatrix in the top-left corner in ascending order
5  |  |  M'_(i,k/2,...,k−1) ← sort(M'_(i,k/2,...,k−1), ASC) // sort the rows of the
   |  |     submatrix in the top-right corner in ascending order
6  |  else
7  |  |  M'_(i,0,...,k/2−1) ← sort(M'_(i,0,...,k/2−1), DESC) // sort the rows of the
   |  |     submatrix in the bottom-left corner in descending order
8  |  |  M'_(i,k/2,...,k−1) ← sort(M'_(i,k/2,...,k−1), DESC) // sort the rows of the
   |  |     submatrix in the bottom-right corner in descending order
9  for j ← 0 to k − 1 do
10 |  m'^T_(:,j) ← sort(m'^T_(:,j))
11 for i ← 0 to k − 1 do
12 |  if i mod 2 ≠ 0 then
13 |  |  m'_(i,:) ← sort(m'_(i,:), ASC) // sort the row in ascending order
14 |  else
15 |  |  m'_(i,:) ← sort(m'_(i,:), DESC) // sort the row in descending order
16 for j ← 0 to k − 1 do
17 |  m'^T_(:,j) ← sort(m'^T_(:,j))
18 return M'
```

**Algorithm 1.9:** 4-way merge algorithm

because each column consists of two sorted sequences intermixed with one another, whilst the number of comparators required is $k\frac{k(k-1)}{2}$.

The `roughsort` function is a recursive function which, at the end of the recursion, uses the `4wayMerge` function; it therefore requires $\sum_{i=0}^{\log_2 n-1} 3\frac{n}{2^i}$ steps and $\sum_{i=0}^{\log_2 n-1} 4^i \frac{\left(\frac{n}{2^i}\right)^2\left(7\frac{n}{2^i}-8\right)}{4}$ comparators. Finally, 4-way mergesort applies a sort to the rows of the mesh after calling `roughsort`, which requires $n$ steps and $n\frac{n(n-1)}{2}$ additional comparators.

The number of steps required to sort an $n \times n$ mesh is:

$$T\left(n^2\right) = 7n - 6 \tag{1.6}$$

The number of comparators required to implement the sorting network, assuming that

each row and column is sorted using OETS [23], is:

$$C(n^2) = 4n^3 - 2n^2 \log_2 n - 4n^2 \tag{1.7}$$

### 1.3.5. LS3-Sort

LS3-sort [26] is a two-dimensional sorting network based on the `LS3Merge` algorithm, which merges four ordered sub-meshes into a single globally ordered mesh.

**Definition 1.3.2** (Perfect Shuffle). *A perfect shuffle is a deterministic permutation in which a sequence is divided into two exactly equal halves and shuffled perfectly, alternating one element from each half.*

The `LS3Merge` algorithm globally sorts a $k \times k$ mesh that has four sub-blocks, $\frac{k}{2} \times \frac{k}{2}$, which are already sorted, by applying the following steps: the perfect shuffle is applied along all rows, the double columns, each containing $2k$ elements, are sorted as a single sequence in snake-like order, and finally $2k$ OETS [23] steps are applied to the entire mesh in snake-like order. The full procedure is shown in Algorithm 1.10.

The LS3-sort algorithm recursively calls itself on the four sub-blocks that make up the mesh to sort them locally and finally, via the `LS3Merge` function, sorts the mesh globally, as shown in Algorithm 1.11.

Here are the details of the number of steps and comparators required to implement the `LS3Merge` function for a mesh $k \times k$. The first step is the shuffle along the rows that requires $\frac{k}{2}$ steps and no comparators, as the shuffle is merely a permutation. The second step is the sorting of each double column that requires $2k$ steps and $\frac{k}{2}\frac{2k(2k-1)}{2}$ comparators. Finally, to perform the $2k$ OETS steps, $2k\frac{k^2}{2}$ comparators are required.

The LS3-sort function is recursive which, at the end of the recursion, calls the `LS3Merge` function; it therefore requires $\sum_{i=0}^{\log_2 n-1} \frac{9}{2}\frac{n}{2^i}$ steps and $\sum_{i=0}^{\log_2 n-1} 4^i \frac{\left(\frac{n}{2^i}\right)^2\left(4\frac{n}{2^i}-1\right)}{2}$ comparators.

The depth required to sort an $n \times n$ mesh, and thus the number of steps required, is:

$$T(n^2) = 9n - 9 \tag{1.8}$$

The number of comparators required to implement this sorting network are:

$$C(n^2) = 4n^3 - \frac{1}{2}n^2 \log_2 n - 4n^2 \tag{1.9}$$

**Input:** $M \in \mathbb{Z}^{k \times k}$: the matrix, whose $\frac{k}{2} \times \frac{k}{2}$-submatrices are sorted
**Output:** $M' \in \mathbb{Z}^{k \times k}$: the sorted matrix
**Data:** $v \leftarrow \mathbb{Z}^{2k}$: the double column
$\mathbf{t} \in \mathbb{Z}^{k^2}$: the flattened matrix
$idx \in \mathbb{Z}$: the index used in the matrix-to-vector transformation

```
1  M' ← M
2  for i ← 0 to k − 1 do
3  |  m'_(i,:) ← shuffle(m'_(i,:))
   // sort each double column in snake-like direction
4  for j ← 0 to k/2 − 1 do
5  |  v ← 0
6  |  for i ← 0 to k − 1 do
7  |  |  v_i ← m'_(i,2j)
8  |  |  v_(2k−1−i) ← m'_(i,2j+1)
9  |  v ← sort(v)
10 |  for i ← 0 to k − 1 do
11 |  |  m'_(i,2j) ← v_i
12 |  |  m'_(i,2j+1) ← v_(2k−1−i)
   // flatten the matrix
13 t ← 0
14 idx ← 0
15 for i ← 0 to k − 1 do
16 |  if i mod 2 = 0 then
17 |  |  for j ← 0 to k − 1 do
18 |  |  |  t_idx ← m'_(i,j);
19 |  |  |  idx ← idx + 1;
20 |  else
21 |  |  for j ← k − 1 to 0 do
22 |  |  |  t_idx ← m'_(i,j);
23 |  |  |  idx ← idx + 1;
   // apply 2k OETS-steps to the snake
24 for s ← 0 to k − 1 do
25 |  t ← oetsOddStep(t)
26 |  t ← oetsEvenStep(t)
   // map back the flattend matrix into matrix
27 idx ← 0
28 for i ← 0 to k − 1 do
29 |  if i mod 2 = 0 then
30 |  |  for j ← 0 to k − 1 do
31 |  |  |  m'_(i,j) ← t_idx idx ← idx + 1
32 |  else
33 |  |  for j ← k − 1 to 0 do
34 |  |  |  m'_(i,j) ← t_idx idx ← idx + 1
35 return M'
```

**Algorithm 1.10:** LS3-merge algorithm

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the sorted matrix

$M' \leftarrow M$
**if** $n > 1$ **then**
  $M'_{(0,\ldots,\frac{n}{2}-1,0,\ldots,\frac{n}{2}-1)} \leftarrow \texttt{ls3Sort}(M'_{(0,\ldots,\frac{n}{2}-1,0,\ldots,\frac{n}{2}-1)})$
  $M'_{(0,\ldots,\frac{n}{2}-1,\frac{n}{2},\ldots,n-1)} \leftarrow \texttt{ls3Sort}(M'_{(0,\ldots,\frac{n}{2}-1,\frac{n}{2},\ldots,n-1)})$
  $M'_{(\frac{n}{2},\ldots,n-1,0,\ldots,\frac{n}{2}-1)} \leftarrow \texttt{ls3Sort}(M'_{(\frac{n}{2},\ldots,n-1,0,\ldots,\frac{n}{2}-1)})$
  $M'_{(\frac{n}{2},\ldots,n-1,\frac{n}{2},\ldots,n-1)} \leftarrow \texttt{ls3Sort}(M'_{(\frac{n}{2},\ldots,n-1,\frac{n}{2},\ldots,n-1)})$
  $M' \leftarrow \texttt{LS3Merge}(M')$
**return** $M'$

**Algorithm 1.11:** LS3-sort algorithm

### 1.3.6. 3n-Sort of Schnorr and Shamir

The 3n-sort of Schnorr and Shamir [43] is a two-dimensional sorting network that achieves an optimal time complexity of $T(n^2) = 3n + \mathcal{O}(n)$. This network has optimal time complexity as it is the closest to the theoretical lower bound Equation 1.1 demonstrated by Kunde [25].

The $n \times n$ mesh is decomposed into vertical slices of size $n \times n^{\frac{3}{4}}$, horizontal slices of size $n^{\frac{3}{4}} \times n$ and blocks of size $n^{\frac{3}{4}} \times n^{\frac{3}{4}}$. The algorithm proceeds as follows: the blocks are sorted, the $n^{\frac{3}{4}}$-way unshuffle is applied along the rows, the blocks are sorted again, the columns are sorted, the vertical slices are sorted, the rows are sorted in alternating directions (even-numbered rows in ascending order and odd-numbered rows in descending order), and finally, $n^{\frac{3}{4}}$ OETS [23] passes are applied to the mesh. The full procedure is detailed in Algorithm 1.12.

**Definition 1.3.3** ($k$-Way Unshuffle). *The $k$-way unshuffle is a permutation defined as the distribution of a sequence of $n$ elements amongst $k$ distinct sets, and can be represented as:*

$$u : i \mapsto (i \bmod k) \left\lceil \frac{n}{k} \right\rceil + \left\lfloor \frac{i}{n} \right\rfloor \tag{1.10}$$

*The $k$-way unshuffle requires that $k$ divides $n$ and is the exact mathematical inverse of the $k$-way perfect shuffle.*

The number of steps and the number of comparators required to implement this sorting network will now be examined in detail. OETS is used to sort the rows and columns, whilst 4-way mergesort [41] is used to sort the blocks.

**Input:** $M \in \mathbb{Z}^{n \times n}$: the unsorted matrix
**Output:** $M' \in \mathbb{Z}^{n \times n}$: the sorted matrix
**Data:** $\mathbf{t} \in \mathbb{Z}^{k^2}$: the flattened matrix
$idx \in \mathbb{Z}$: the index used in the matrix-to-vector transformation

$M' \leftarrow M$
// sort the blocks using 4-way mergesort
$b \leftarrow n^{\frac{3}{4}}$
**for** $i \leftarrow 0$ **to** $n^{\frac{1}{4}} - 1$ **do**
  **for** $j \leftarrow 0$ **to** $n^{\frac{1}{4}} - 1$ **do**
    $r_{start} \leftarrow i\ b$
    $r_{end} \leftarrow (i+1)b - 1$
    $c_{start} \leftarrow j\ b$
    $c_{end} \leftarrow (j+1)b - 1$
    $M'_{(r_{start},\ldots,r_{end},c_{start},\ldots,c_{end})} \leftarrow$ `4wayMergesort`$(M'_{(r_{start},\ldots,r_{end},c_{start},\ldots,c_{end})})$
**for** $i \leftarrow 0$ **to** $n - 1$ **do**
  $\mathbf{m}'_{(i,:)} \leftarrow$ `kWayUnshuffle`$(\mathbf{m}'_{(i,:)}, n^{\frac{1}{4}})$
// sort the blocks using 4-way mergesort
$b \leftarrow n^{\frac{3}{4}}$
**for** $i \leftarrow 0$ **to** $n^{\frac{1}{4}} - 1$ **do**
  **for** $j \leftarrow 0$ **to** $n^{\frac{1}{4}} - 1$ **do**
    $r_{start} \leftarrow i\ b$
    $r_{end} \leftarrow (i+1)b - 1$
    $c_{start} \leftarrow j\ b$
    $c_{end} \leftarrow (j+1)b - 1$
    $M'_{(r_{start},\ldots,r_{end},c_{start},\ldots,c_{end})} \leftarrow$ `4wayMergesort`$(M'_{(r_{start},\ldots,r_{end},c_{start},\ldots,c_{end})})$
**for** $j \leftarrow 0$ **to** $n - 1$ **do**
  $\mathbf{m}'^{\intercal}_{(:,j)} \leftarrow$ `sort`$(\mathbf{m}'^{\intercal}_{(:,j)})$
**for** $f \leftarrow 0$ **to** $\sqrt{n} - 1$ **do**
  $s \leftarrow f \cdot \sqrt{n}$
  $e \leftarrow s + \sqrt{n} - 1$
  $M'_{(:,s,\ldots,e)} \leftarrow$ `sortVerticalSlice`$(M'_{(:,s,\ldots,e)})$
**for** $i \leftarrow 0$ **to** $n - 1$ **do**
  **if** $i \bmod 2 = 0$ **then**
    $\mathbf{m}'_{(i,:)} \leftarrow$ `sort`$(\mathbf{m}'_{(i,:)}$, *ASC*$)$ // sort the row in ascending order
  **else**
    $\mathbf{m}'_{(i,:)} \leftarrow$ `sort`$(\mathbf{m}'_{(i,:)}$, *DESC*$)$ // sort the row in descending order
// flatten the matrix
$\mathbf{t} \leftarrow \mathbf{0}$
$idx \leftarrow 0$
**for** $i \leftarrow 0$ **to** $k - 1$ **do**
  **if** $i \bmod 2 = 0$ **then**
    **for** $j \leftarrow 0$ **to** $k - 1$ **do**
      $t_{idx} \leftarrow m'_{(i,j)}$;
      $idx \leftarrow idx + 1$;
  **else**
    **for** $j \leftarrow k - 1$ **to** $0$ **do**
      $t_{idx} \leftarrow m'_{(i,j)}$;
      $idx \leftarrow idx + 1$;
// apply $n^{\frac{3}{4}}$ OETS-steps to the snake
**for** $f \leftarrow 0$ **to** $\frac{n^{\frac{3}{4}}}{2} - 1$ **do**
  $\mathbf{t} \leftarrow$ `oetsOddStep`$(\mathbf{t})$
  $\mathbf{t} \leftarrow$ `oetsEvenStep`$(\mathbf{t})$
// map back the flattend matrix into matrix
$idx \leftarrow 0$
**for** $i \leftarrow 0$ **to** $k - 1$ **do**
  **if** $i \bmod 2 = 0$ **then**
    **for** $j \leftarrow 0$ **to** $k - 1$ **do**
      $m'_{(i,j)} \leftarrow t_{idx}\ idx \leftarrow idx + 1$
  **else**
    **for** $j \leftarrow k - 1$ **to** $0$ **do**
      $m'_{(i,j)} \leftarrow t_{idx}\ idx \leftarrow idx + 1$
**return** $M'$

**Algorithm 1.12:** 3n-sort of Schnorr and Shamir algorithm

Sorting the blocks requires $7n^{\frac{3}{4}} - 6$ steps and

$$n^{\frac{1}{2}} \left( 4 \left( n^{\frac{3}{4}} \right)^3 - 2 \left( n^{\frac{3}{4}} \right)^2 \log_2 n^{\frac{3}{4}} - \frac{7}{2} \left( n^{\frac{3}{4}} \right)^2 \right) \tag{1.11}$$

comparators; performing the $n^{\frac{1}{4}}$-way unshuffle along the rows requires $n$ steps and no comparators, as it is simply a permutation. To sort the columns, $n$ steps and $n\frac{n(n-1)}{2}$

comparators are required; to sort the vertical slices, $7n^{\frac{3}{4}} - 6$ steps and

$$n^{\frac{1}{2}}\left(4\left(n^{\frac{3}{4}}\right)^3 - 2\left(n^{\frac{3}{4}}\right)^2 \log_2 n^{\frac{3}{4}} - \frac{7}{2}\left(n^{\frac{3}{4}}\right)^2\right) \tag{1.12}$$

comparators are required, because they contain a region of only $n^{\frac{1}{4}}$ unsorted rows (for example, by sorting the blocks and subsequently the vertically overlapping blocks of $n^{\frac{1}{4}}$ rows). Then, to sort the rows using OETS, $n$ steps and $n\frac{n(n-1)}{2}$ comparators are required. Finally, to perform the $n^{\frac{3}{4}}$ OETS steps, $n^{\frac{3}{4}}\frac{n^2}{2}$ comparators are required.

The depth required to sort an $n \times n$ mesh, and thus the number of steps required, is:

$$T\left(n^2\right) = 3n + 22n^{\frac{3}{4}} - 18 \tag{1.13}$$

The number of comparators required to implement this sorting network are:

$$C\left(n^2\right) = n^3 + \frac{25}{2}n^{\frac{11}{4}} - \frac{9}{2}n^2 \log_2 n - 13n^2 \tag{1.14}$$

## 1.4. The Parallel Brute-Force Model

In [8], Bernstein identifies and criticizes a widespread methodological error in the cryptographic literature: the security of a cryptosystem is typically assessed by *considering only attacks using a big sequential machine*, ignoring the possibility that an adversary might have access to a huge amount of cheap small computing resources. Consequently, an attack algorithm by exploiting these resources, could prove more effective than its serial counterpart.

Although Bernstein formulates his criticism in the specific context of exhaustive AES [36] key search, the resulting methodological principle is general: a suitably designed parallel machine can be superior to a serial machine of comparable cost in terms of both execution speed and cost-performance ratio.

### 1.4.1. The Problem Analyzed

The context chosen by Bernstein in [8] is the exhaustive search for keys to the AES cipher [36], formulated as follows. Let $m \in \mathbf{M}$ be a fixed and public plaintext, and let $n$ ciphertexts be given (in [8] $n = 2^{10}$), obtained using the same plaintext with different keys $k_1, \ldots, k_n \in \mathbf{K}$. The objective of the problem is to find the keys based on knowledge of the plaintext $m \in \mathbf{M}$ and the ciphertexts $c_1, \ldots, c_n$.

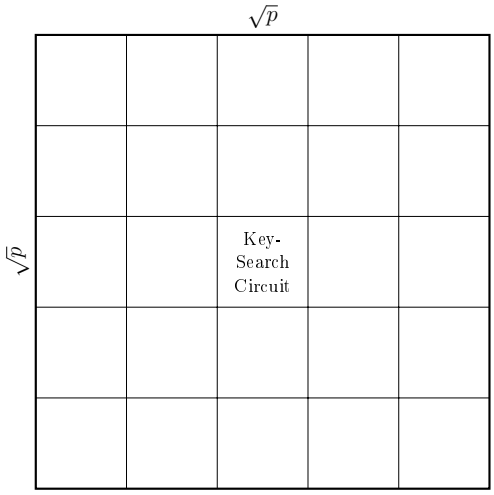


Figure 1.1: Key-Search Machine

The key point of the problem is that the $n$ keys have been used to encrypt the same plaintext $m \in \mathbf{M}$. This structure of the problem allows the attacker to construct computation chains, which, by propagating through repeated application, can yield multiple target keys within the same computation. The attacker does not know the keys, but knows the plaintext $m \in \mathbf{M}$ and the $n$ ciphertexts $c_1, \ldots, c_n$, through which the keys can be deduced.

### 1.4.2. The Parallel Machine

Bernstein designs a parallel machine to reduce the inefficiency of serial attacks against this problem (Subsection 1.4.1). The machine consists of $p$ *key-search circuits* (in [8] $p = 2^{32}$) arranged in a two-dimensional mesh of size $\sqrt{p} \times \sqrt{p}$, where each circuit is connected only to its four immediate neighbors (north, south, west and east) in the mesh. Each circuit operates autonomously and is equipped with its own local memory in which to store $2^4$ results.

This architecture reflects the characteristics of realistic dedicated hardware: an Application-Specific Integrated Circuit (ASIC) where long-distance communication is costly, and therefore local connectivity is the only viable form of communication on a large scale. The total cost of the machine is proportional to the number of circuits $p$ and the cost of each Boolean operation performed within it.

The general structure of the machine is shown in Figure 1.1, whilst the detailed internal structure of each circuit, with particular attention to the buffer used to store partial results, is shown in Figure 1.2.

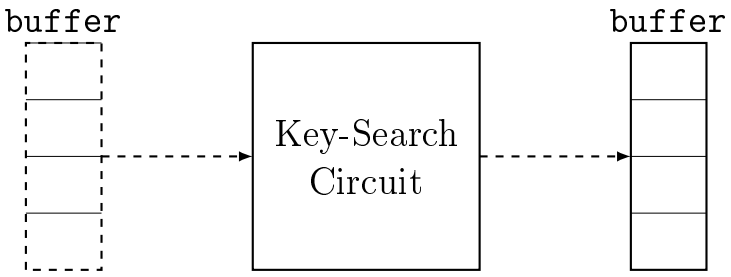


Figure 1.2: Key-Search Circuit with the Buffer

The attack implemented on the parallel machine consists of three main phases: the local computation phase, the global sorting phase and the collision search phase.

### 1.4.3. Local Computation Phase

During the local computation phase, the mesh's circuits operate in parallel to explore the key space. The aim of this phase is to produce, for each input provided, the final value of the corresponding computation chain.

**Key-Search Circuit** The key-search circuit is the basic component of the machine, which implements the chain function $z(\mathcal{B}, q, s)$ defined recursively as:

$$z(\mathcal{B}, 0, s) = s \tag{1.15}$$

$$z(\mathcal{B}, q+1, s) = z(\mathcal{B}, q, \texttt{aes}(m, s \oplus (\mathcal{B} \parallel n))) \tag{1.16}$$

where $\mathcal{B}$ is a 96 bit random string fixed a priori and identical for all circuits, $n$ represents the length the chain must have, $s$ is the current state of the chain, and and the `aes`$(\cdot, \cdot)$ function takes as parameters the message to be encrypted and the key to be used for encryption. The chain function $z(\cdot)$ produces a chain of length $n$ by consecutively applying the AES cipher [36], starting from the initial value $s$.

**The Inputs Provided to the Parallel Machine** At the start of the attack, the attacker selects a seed $\mathcal{B} \in \mathbb{F}_2^{96}$ and distributes $2^{36}$ inputs amongst the $2^{32}$ circuits (each receiving $2^4$ inputs to fill its local memory). The inputs are divided into two categories.

In type 1, there are $2^{36} - 2^{33}$ random inputs of the form $(\mathcal{B}, 2^{23}, r_i)$, where $r_i$ represents a different 128 bit random string for each input. Each $r_i$ is the starting point of a chain of $2^{23}$ steps, and the $2^{23}$ intermediate values represent the key attempts explored from that input.

Type 2 consists of $2^{33}$ target inputs: for each of the $2^{10}$ ciphertexts, $c_1, \ldots, c_n$, $2^{23}$ distinct inputs $(\mathcal{B}, 0, c_i), \ldots, (\mathcal{B}, 2^{23}-1, c_i)$ are generated, for a total of $2^{10} \cdot 2^{23} = 2^{33}$ target inputs.

**Parallel Execution** Once all the input data to be computed has been obtained, all $2^{32}$ circuits execute the chain function in parallel on their respective $2^4$ inputs, producing the final values in a time equivalent to approximately $2^{23}\, 2^4 = 2^{27}$ AES evaluations.

The reason why the type 2 inputs are set to all possible values of $n$, from 0 to $2^{23} - 1$, is as follows. If the type 1 chain starting from $r_i$ passes through a state $s$ at step $n$ such that $s \oplus (\mathcal{B} \parallel n) = k_j$, then we have that:

$$z\big(\mathcal{B}, 2^{23}, r_i\big) = z(\mathcal{B}, n, c_j) \tag{1.17}$$

that is, the final values of the two chains collide. By inserting all possible $n$ starting points for the target chains, it is guaranteed that, whatever the step at which the random chain encounters $k_j$, there will exist a target input whose final value coincides with that of the random chain, making the collision detectable after sorting.

### 1.4.4. Global Sorting Phase

At the end of the local computation phase, the $2^{32}$ circuits produced a total of $2^{36}$ results. These values must be sorted globally so that collisions can be searched for. The sorting is performed directly on the two-dimensional $\sqrt{p} \times \sqrt{p}$ mesh using the LS3-sort algorithm [26], which requires $2^{21}$ CAS steps to sort $2^{36}$ elements, a negligible cost compared to the $2^{27}$ steps of the computation phase.

### 1.4.5. Collision Search Phase

The collision search is not a separate phase, but takes place during the global sorting phase. Every CAS operation between two adjacent elements in the mesh does not merely compare the values to determine the order, but also checks whether the two elements are identical. If, during a CAS, two identical final values are detected, one from a random chain generated from $r_i$ and one from a target chain generated from $c_j$, then the sorting is halted and the chain of $r_i$ is recalculated step by step, until an intermediate value $s$ is found such that:

$$\texttt{encryption}(s \oplus (\mathcal{B} \parallel n)) = c_j \tag{1.18}$$

At that point, the found key $k_j = s \oplus (\mathcal{B} \parallel n)$ is returned, and the sort continues.

### 1.4.6. Success Probability

The probability that the parallel machine will find a specific target key, for example $k_1$, in a single run can be estimated as follows. The parallel machine studied previously, with $p = 2^{32}$ parallel machines, each with a memory capable of holding $2^4$ elements, generates $2^{36}$ random chains of length $2^{23}$, for a total of $2^{59}$ intermediate results. If at least one of these results matches the target key, $k_1$, the machine will find the key. The probability of this event is:

$$\Pr[\text{find } k_1] = \frac{2^{59}}{|\mathbf{K}|} = \tag{1.19}$$

$$= \frac{2^{59}}{2^{128}} = \tag{1.20}$$

$$= 2^{-69} \tag{1.21}$$

To increase the probability of success, the machine can be run multiple times, for example $r = 2^5$ times, with different choices of the parameter $\mathcal{B}$; in this way, the probability increases to:

$$\Pr[\text{find } k_1 \text{ in } r \text{ runs}] = \Pr[\text{find } k_1] \cdot r = \tag{1.22}$$

$$= 2^{-69} \cdot 2^5 = \tag{1.23}$$

$$= 2^{-64} \tag{1.24}$$

In the problem analyses by Bernstein, $n = 2^{10}$ keys must be searched for; the probability of finding at least one is:

$$\Pr[\text{find a key}] = \Pr[\text{find } k_1] \cdot n = \tag{1.25}$$

$$= 2^{-69} \cdot 2^{10} = \tag{1.26}$$

$$= 2^{-59} \tag{1.27}$$

### 1.4.7. Comparison of Parallel and Serial Machines

Bernstein compares the parallel machine described above with a serial version that utilizes a time-memory trade-off, implemented, e.g., using Oechslin's rainbow tables [35].

The serial machine computes the same $2^{36}$ chain of length $2^{23}$, now sequentially, using approximately $2^{59}$ AES [36] evaluations, which is about $2^{32}$ times slower than the parallel machine, which completes the same operations in just $2^{27}$ evaluations. However, the serial machine is not $2^{32}$ times cheaper: it does not have $2^{32}$ key-search circuits, but only one, yet it requires the same amount of memory to store the $2^{36}$ results, and its overall cost is

comparable to that of the parallel machine.

The structural drawback of the serial version is an excess of memory relative to computational capacity, as a large amount of memory remains idle whilst waiting for a single processor, which utilizes it sequentially. This imbalance is not a marginal issue, but is intrinsic to the serial model; indeed, increasing the size of the machine to improve performance proportionally worsens the imbalance, as memory grows linearly with the size of the machine whilst computational capacity remains that of a single circuit.

In contrast, in a parallel machine, the computational load is distributed evenly across all $p$ circuits, each of which works autonomously on its own portion of local memory. There is therefore no structural bottleneck: increasing the number of circuits improves performance linearly without degrading the ratio between computational capacity and available memory.

From these analyses, Bernstein concludes that any cryptographic analysis must not be limited to comparing a new attack with the best known serial attack, but must also consider parallel variants.

## 1.5. Code-based Cryptosystem

Code-based cryptography is one of the most extensively studied families within the field of PQC. The scientific interest in these cryptosystems stems from recent developments in the field of quantum computing, which threaten the security of the most widely used public-key cryptosystems. Current standard cryptosystems, in fact, base their security on the computational difficulty of solving complex mathematical problems such as the factorization of large integers (RSA [39]) or the calculation of discrete logarithms over finite fields (Diffie-Hellman [15]) or elliptic curves (ECC [33]). Although these problems are computationally impossible to solve for a classical computer (i.e. there are no algorithms capable of solving them in polynomial time), with the advent of quantum computers, it is possible to implement algorithms that can solve these problems in polynomial time; one example is the algorithm proposed by Peter Shor in 1994 [44], rendering current cryptosystems obsolete.

A further critical issue is the *harvest now, decrypt later* paradigm: an attacker can intercept and store communications encrypted using classical algorithms, with the intention of decrypting them later once suitable quantum hardware becomes available. This prospect necessitates the immediate adoption of quantum-resistant solutions to ensure the long-term confidentiality of sensitive data and critical infrastructure.

In this scenario, code-based cryptography stands out for the robustness of its theoretical foundations. The idea of using error-correcting codes not for their traditional purpose (protecting signals from noise in telecommunications), but as the basis for a public-key cryptosystem, is attributed to Robert McEliece [32]. The principle underlying code-based cryptosystems lies in the intrinsic difficulty of problems related to linear code theory, in particular the GDP and SDP. Unlike number theory problems, which have been used in cryptography to date, there are no known standard or quantum algorithms capable of drastically reducing the complexity of decoding a generic linear code; this ensures that these systems remain highly secure even against adversaries with quantum computational capabilities.

From an operational perspective, code-based cryptosystems use error-correcting codes with an efficient algebraic structure, known only to the legitimate receiver, which is appropriately concealed within the public-key so as to appear indistinguishable from a random code. In summary, at a high level, the encryption and decryption process is as follows. During the encryption phase, the sender encodes the message by intentionally introducing a controlled error vector. To decrypt the received message, the receiver must use their knowledge of the code's secret structure to correct the error and recover the original message.

The main advantage of code-based cryptosystems lies in their robustness, which stems from decades of cryptographic research and analysis on the subject. Historically, however, their widespread standardized adoption has been limited by the large size of public-keys, which are significantly larger than those used by the RSA and ECC cryptosystems currently in use.

The recent standardization process promoted by NIST in the field of PQC has renewed academic and industrial interest in this family of algorithms, with several code-based cryptosystems now among the leading candidates for PQC.

### 1.5.1. Coding Theory

Coding theory is a branch of theoretical computer science and applied mathematics that focuses on studying methods designed to ensure the integrity and reliability of transmissions over channels affected by noise. This theory originated in the context of digital communications; it provides the theoretical framework necessary for designing codes capable of identifying and correcting errors that occur during the transmission of a message over a channel.

The fundamental principle upon which the entire theory is based consists of introducing

controlled redundancy into the original message to be transmitted. Through systematic encoding, the message is transformed into a codeword belonging to a structured set. This encoding operation endows the message with specific algebraic and metric properties, which allow the receiver to decode the message, thereby detect the presence of errors, and, within certain predefined limits, unambiguously reconstruct the original message.

Let $\mathbf{M}$ be the message space and $\mathbf{C}$ the codeword space. The encoding function maps each message $\mathbf{m} \in \mathbf{M}$ to a codeword $\mathbf{c} \in \mathbf{C}$. The codeword is transmitted over a noisy channel; the codeword $\mathbf{c}$ is perturbed by a vector $\mathbf{e}$, so the receiver will receive the codeword $\mathbf{r} = \mathbf{c} + \mathbf{e}$. At this point, the receiver must decode the received message by recovering the original message $\mathbf{m}$ from the received codeword $\mathbf{r}$. The effectiveness of the decoding operation depends on the error-correcting capacity of the code used, defined by its minimum distance and its structural properties.

In the context of this discussion, the focus is on **binary linear codes**, a family of codes that possesses algebraic properties making them suitable for efficient implementations. These codes also form the fundamental core of *code-based cryptography*.

**Definition 1.5.1** (Linear Code). *A linear code $\mathcal{C}[n, k]$ of length $n$ and dimension $k$ is a linear subspace $\mathbf{C}$ with dimension $k$ of the vector space $\mathbb{F}_q^n$, where $\mathbb{F}_q$ is the finite field with $q$ elements.*

The elements of $\mathcal{C}[n, k]$ are called *codeword*, $\mathbf{c} \in \mathbb{F}_q^n$. The parameter $n$ is the code length, the number of transmitted symbols, and $k$ is the code dimension, the number of pure information symbols. The transmission rate is defined as $r := \frac{k}{n}$ and measures the code's efficiency.

If the finite field of definition is the two-element Galois field, denoted by $\mathbb{F}_2$, the code is called a **binary linear code**. In this context, the elements of the code are vectors whose components belong to the alphabet $\{0, 1\}$.

**Definition 1.5.2** (Generator Matrix). *A generator matrix $G \in \mathbb{F}_q^{k \times n}$ is a matrix whose rows form a basis for a linear code.*

The codewords are all the linear combinations of the rows of the generator matrix; in other words, the linear code is the space of the rows of its generator matrix.

The encoding operation maps the message space $\mathbb{F}_q^k$ onto the codeword space $\mathbb{F}_q^n$. This

process is formalized by a linear injective map:

$$c_{[n,k]} : \mathbb{F}_q^k \to \mathbb{F}_q^n \tag{1.28}$$

$$\mathbf{m} \mapsto \mathbf{c} \tag{1.29}$$

For the mapping between messages and their corresponding code words to be injective, the encoding function $c_{[n,k]}$ must be defined by a generator matrix $G \in \mathbb{F}_q^{k\times n}$. This matrix must have full rank, $\text{rank}\,(G) = k$, which ensures that its rows are linearly independent and form a basis for the linear subspace $\mathcal{C}[n,k]$. In this way, the $k$-dimensional message space is faithfully mapped onto a $k$-dimensional subspace within the $n$-dimensional ambient space, preserving the integrity of the information during the encoding process.

It is important to note that the generator matrix $G$ for a given linear code $\mathcal{C}[n,k]$ is not unique: any set of $k$ linearly independent vectors belonging to $\mathcal{C}[n,k]$ can serve as a basis and, therefore, constitute a valid generator matrix. However, regardless of the choice of basis, all generator matrices of the same $\mathcal{C}[n,k]$ code necessarily share the same rank $k$, which is equal to the dimension of the subspace.

To generate a codeword $\mathbf{c} \in \mathbb{F}_q^n$ from a message $\mathbf{m} \in \mathbb{F}_q^k$, we proceed via the vector-matrix product:

$$\mathbf{c} = \mathbf{m}G \tag{1.30}$$

**Definition 1.5.3** (Dual Code). *Let a linear code $\mathcal{C}[n,k]$, the dual code is defined as:*

$$\mathcal{C}^{\perp}[n, n-k] \coloneqq \left\{\mathbf{x} \in \mathbb{F}_q^n \mid \mathbf{x}\cdot\mathbf{c} = 0, \forall \mathbf{c} \in \mathcal{C}[n,k]\right\} \tag{1.31}$$

The operation $\mathbf{x}\cdot\mathbf{c}$ is the scalar product defined as $\mathbf{x}\cdot\mathbf{c} \coloneqq \sum_{i=0}^{n-1} x_i c_i$.

**Definition 1.5.4** (Parity-Check Matrix). *A parity-check matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ of a linear code $\mathcal{C}[n,k]$ is a generator matrix of the dual code $\mathcal{C}^{\perp}[n, n-k]$.*

From the definition of the parity-check matrix, it is possible derive that a vector $\mathbf{c} \in \mathbb{F}_q^n$ is a codeword if and only if $H\mathbf{c}^{\intercal} = \mathbf{0}$.

The relationship between the generator matrix $G \in \mathbb{F}_q^{k\times n}$ and the parity-check matrix $H \in \mathbb{F}_q^{n-k\times n}$ is defined by the orthogonality condition between the code and its dual. Since each row of $G$ is, by definition, a codeword of $\mathcal{C}[n,k]$, it must belong to the kernel

space of $H$. This property is formalized by the identity:

$$GH^\intercal = 0, \quad 0 \in \mathbb{F}_q^{k\times(n-k)} \tag{1.32}$$

This relationship allows for an immediate construction of one matrix from the other when the code is represented in systematic form. If the generator matrix is expressed as $G = \begin{bmatrix} I_k & A \end{bmatrix} \in \mathbb{F}_q^{k\times n}$ the parity-check matrix can be easily represented as $H = \begin{bmatrix} -A^\intercal & I_{n-k} \end{bmatrix} \in \mathbb{F}_q^{(n-k)\times n}$.

**Definition 1.5.5** (Syndrome). *Let $H \in \mathbb{F}_q^{(n-k)\times n}$ the parity-check matrix of the linear code $\mathcal{C}[n,k]$, a syndrome $\mathbf{s} \in \mathbb{F}_q^{n-k}$ of a vector $\mathbf{x} \in \mathbb{F}_q^n$ is:*

$$\mathbf{s}^\intercal \coloneqq H\mathbf{x}^\intercal \tag{1.33}$$

The vector $\mathbf{x} \in \mathbb{F}_q^n$ is a codeword if and only if the syndrome $\mathbf{s} = \mathbf{0}$, $\mathbf{0} \in \mathbb{F}_q^{n-k}$.

**Definition 1.5.6** (Hamming Distance). *The Hamming distance between two equal-length vector is the number of positions at which the corresponding symbols are different. Let $\mathbf{x}, \mathbf{y} \in \mathbb{F}_q^m$ the Hamming distance is:*

$$\mathrm{d_h}(\mathbf{x}, \mathbf{y}) \coloneqq |\{i \mid x_i \neq y_i\}| \tag{1.34}$$

**Definition 1.5.7** (Hamming Weigth). *The Hamming weight of a vector is the number of elements that are different from the zero. Let $\mathbf{x} \in \mathbb{F}_q^m$ the Hamming weight is:*

$$\mathrm{HW}(\mathbf{x}) \coloneqq |\{i \mid x_i \neq 0\}| \tag{1.35}$$

The Hamming weight of a vector $\mathbf{x} \in \mathbb{F}_q^m$ is equivalent to the Hamming distance from the zero vector of the same length $\mathrm{HW}(\mathbf{x}) = \mathrm{d_h}(\mathbf{x}, \mathbf{0})$, $\mathbf{0} \in \mathbb{F}_q^m$

**Definition 1.5.8** (Minimum Distance of a Linear Code). *The minimum distance of a linear code $\mathcal{C}[n,k]$ is:*

$$d(\mathcal{C}[n,k]) \coloneqq \min\{\mathrm{d_h}(\mathbf{c_1}, \mathbf{c_2}) \mid \mathbf{c_1}, \mathbf{c_2} \in \mathcal{C}[n,k] \wedge \mathbf{c_1} \neq \mathbf{c_2}\} \tag{1.36}$$

Henceforth, a linear code $\mathcal{C}$ of length $n$, dimension $k$ and minimum distance $d$ will be denoted as $\mathcal{C}[n,k,d]$.

For linear codes, due to the property of closure under addition, the minimum distance is equivalent to the minimum Hamming weight of any non-zero codeword:

$$d(\mathcal{C}[n,k]) = \min_{\mathbf{c}\in\mathcal{C}[n,k],\mathbf{c}\neq\mathbf{0}} \mathrm{HW}(\mathbf{c}) \tag{1.37}$$

The minimum distance of a linear code is the parameter that determines the number of errors the code can correct; the greater the minimum distance, the greater the number of errors the code can correct. Specifically, a linear code can detect up to $d(\mathcal{C}[n,k])-1$ errors and correct up to $t = \left\lfloor \frac{d(\mathcal{C}[n,k])-1}{2} \right\rfloor$ errors. Moreover, the number of detectable errors are $q^n - q^k$.

A central issue in coding theory is the trade-off between the coding rate $r = \frac{k}{n}$ and the error-correction capacity $d = d(\mathcal{C}[n,k]) - 1$. This relationship is subject to several theoretical limits, the most immediate of which is the Singleton bound.

**Definition 1.5.9** (Singleton Bound). *Let $\mathcal{C}[n,k,d]$, then the maximum number of codewords is $q^k$ and the Singleton bound implies:*

$$q^k \leq q^{n-d+1} \tag{1.38}$$

$$k \leq n - d + 1 \tag{1.39}$$

$$d \leq n - k + 1 \tag{1.40}$$

The Singleton bound is an optimistic upper bound. It defines a region of unfeasibility and is not a guarantee of existence; therefore, it does not guarantee that a code satisfying the Singleton bound is actually implementable. To determine the conditions under which the existence of a linear code is guaranteed, we must use the Gilbert-Varshamov bound, which provides a sufficient condition for existence by identifying the parameters for which at least one linear code is known to exist.

**Definition 1.5.10** (Gilbert-Varshamov Bound). *The Gilbert-Varshamov bound provides a sufficient condition for the existence of a linear code. A linear code $\mathcal{C}[n,k,d]$ is guaranteed to exist if:*

$$\sum_{i=0}^{d-2} \binom{n-1}{i} (q-1)^i < q^{n-k} \tag{1.41}$$

In summary, the Singleton bound is a necessary condition for existence: if the code's parameters do not satisfy it, the code is mathematically impossible. The Gilbert-Varshamov bound, on the other hand, provides a sufficient condition: if the parameters satisfy this

bound, the existence of at least a linear code is guaranteed. Thus, the search space for linear codes lies between what definitely exists (the Gilbert-Varshamov bound) and what is theoretically unattainable (the Singleton bound).

**Theorem 1.1** (Error Correction for Linear Code). *Let $\mathcal{C}[n,k,d]$ be a binary linear code, the parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$, the syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$ and the error $\mathbf{e} \in \mathbb{F}_2^{\mathbf{n}}$, such that $\mathbf{s}^\intercal = H\mathbf{e}^\intercal$.*

*There exist one and only one error $\mathbf{e}$ such that*

$$\mathrm{HW}(\mathbf{e}) = w \leq \left\lfloor \frac{d-1}{2} \right\rfloor \tag{1.42}$$

The Theorem 1.1 [29] guarantees that the error found matches the one used to encrypt the message, when we solve a SDP having parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ of a binary linear code $\mathcal{C}[n,k,d]$, $d > 2w$.

**Theorem 1.2** (Syndrome Invariance under Linear Transformations Error Correction for Linear Code). *Let $\mathcal{C}[n,k,d]$ be a binary linear code, the parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$, the syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$ and the error $\mathbf{e} \in \mathbb{F}_2^{\mathbf{n}}$, such that $\mathbf{s}^\intercal = H\mathbf{e}^\intercal$.*

*For any non-singular matrix $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$ and any permutation matrix $P \in \mathbb{F}_2^{n\times n}$ the two following SDP are equivalent to solve:*

$$\mathbf{s}^\intercal = H\mathbf{e}^\intercal \iff \widehat{\mathbf{s}}^\intercal = \widehat{H}\widehat{\mathbf{e}}^\intercal \tag{1.43}$$

*where $\widehat{\mathbf{s}}^\intercal = U\mathbf{s}^\intercal$, $\widehat{H} = UHP$ and $\widehat{\mathbf{e}}^\intercal = P^\intercal\mathbf{e}^\intercal$.*

*Proof.* Consider the permuted syndrome equation $\widehat{\mathbf{s}}^\intercal = \widehat{H}\widehat{\mathbf{e}}^\intercal$. By substituting the definitions of $\widehat{\mathbf{s}}$, $\widehat{H}$ and $\widehat{\mathbf{e}}$, we obtain:

$$\widehat{H}\widehat{\mathbf{e}}^\intercal = (UHP)\left(P^\intercal\mathbf{e}^\intercal\right) = \tag{1.44}$$

$$= UH\left(PP^\intercal\right)\mathbf{e}^\intercal \tag{1.45}$$

Since $P$ is a permutation matrix, $PP^\intercal = I_n$. Thus:

$$\widehat{H}\widehat{\mathbf{e}}^\intercal = UHI_n\mathbf{e}^\intercal = \tag{1.46}$$
$$= UH\mathbf{e}^\intercal = \tag{1.47}$$
$$= U\mathbf{s}^\intercal = \tag{1.48}$$
$$= \widehat{\mathbf{s}}^\intercal \tag{1.49}$$

□

The Theorem 1.2 [29] provide the mathematical foundation for the ISD family of algorithms. Specifically, Theorem 1.2 allows us to transform the parity-check matrix $H$ into a more convenient form without altering the problem. The parity-check matrix is transformed into a systematic matrix; to do this, we must identify a subset of columns of the matrix that are linearly independent. This leads to the definition of an Information Set (IS).

**Definition 1.5.11** (Information Set). *Let $\mathcal{C}[n, k, d]$ be a binary linear code.*

*A set of indices $\mathbf{I} \subset \{0, \ldots, n-1\}$ is called Information Set (IS) if $|\mathbf{I}| = k$ and the column of the generator matrix $G \in \mathbb{F}_2^{k\times n}$ indexed by $\mathbf{I}$ are linearly independent.*

In the context of the SDP, where the parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ is used, the IS is defined equivalently via its complement. Let $\mathbf{J} = \{0, \ldots, n-1\} \setminus \mathbf{I}$ be the *Redundancy Set (RS)*, with $|\mathbf{J}| = n-k$. The set $\mathbf{I}$ is the IS if and only if the submatrix formed by the columns of the parity-check matrix indexed by the $\mathbf{J}$, $H_{(:,\mathbf{J})}$, has full rank:

$$\operatorname{rank}\left(H_{(:,\mathbf{J})}\right) = n-k \tag{1.50}$$

### 1.5.2. McEliece Cryptosystem

Despite its theoretical robustness, historically, this cryptosystem has not been widely adopted due to the large size of the keys. To guarantee a minimum security level of 80 bit, the public-key must be approximately 60 kB, whilst to achieve the 256 bit security required in modern contexts, the size rises to approximately 958 kB [10]. Despite this limitation, its invulnerability to known quantum algorithms makes it one of the most reliable and extensively studied candidates in the PQC landscape today.

The security of the cryptosystem is based on the GDP, which has been shown to be NP-hard [6].

**Definition 1.5.12** (General Decoding Problem). *Let a generation matrix $G \in \mathbb{F}_2^{k \times n}$ of a linear code $\mathcal{C}[n,k]$, a Hamming weight $w$ of the error vector $\mathbf{e} \in \mathbb{F}_2^n$, and a vector $\mathbf{y} \in \mathbb{F}_2^n$.*

*The objective of the General Decoding Problem (GDP) is to find a message vector $\mathbf{m} \in \mathbb{F}_2^k$ and an error vector $\mathbf{e} \in \mathbb{F}_2^n$ such that*

$$\begin{cases} \mathbf{y} = \mathbf{m}G + \mathbf{e} \\ \mathrm{HW}(\mathbf{e}) \leq w \end{cases} \tag{1.51}$$

The security of the scheme is based on the fact that solving the GDP for a general linear code is computationally infeasible. The most effective class of algorithms for attacking this problem is known as ISD, which will be discussed in detail in the following sections.

In the original construction, McEliece proposed the use of *binary Goppa codes*; this choice is due to their ease of generation and the availability of highly efficient decoding algorithms, such as Patterson's algorithm [37]. These codes allow the legitimate recipient to decode messages efficiently, whilst their structure is hidden within the public-key to prevent an attacker from exploiting it.

As a public-key cryptosystem, each user must first generate a key pair: a public-key accessible to anyone, and a private-key, which the owner keeps secret. The key pair is generated using the key generation primitive described in Algorithm 1.13. The public-key consists of a public generator matrix $\widehat{G}$ and the number $w$ of errors that the code is capable of correcting; the private-key, on the other hand, consists of a random invertible dense scrambling matrix $S$, a random permutation matrix $P$ and the code's generator matrix $G$.

To send a message $\mathbf{m}$ to Bob, Alice must follow the encoding and error-insertion steps described in Algorithm 1.14. By following the steps of the encryption algorithm, the message $\mathbf{m}$ is transformed into the ciphertext $\mathbf{c}$, which represents a codeword of the linear code $\mathcal{C}[n,k]$. To decrypt a message $\mathbf{c}$ received by Alice, Bob must use his private-key to remove the error and recover the original message, as illustrated in Algorithm 1.15.

The main advantage of this cryptosystem is the computational efficiency of encryption and decryption, which can be reduced to simple matrix and vector multiplications and the application of decoding algorithms. However, in addition to the aforementioned key size, the cryptosystem suffers from significant data expansion: the resulting ciphertext is considerably longer than the original message, an intrinsic characteristic of error-correcting systems.

**Input:** $n \in \mathbb{N}$: the code length
$k < n$: the code dimension
$w < n$: the number of errors the code can correct
**Output:** $(k_{private}, k_{public})$: the key pair
**Data:** $G \in \mathbb{F}_2^{k \times n}$: the generator matrix for the Goppa linear code
$S \in \mathbb{F}_2^{k \times k}$: the random invertible dense scrambling matrix
$P \in \mathbb{F}_2^{n \times n}$: the random permutation matrix
$\widehat{G} \in \mathbb{F}_2^{k \times n}$: the public matrix

1 generate a random Goppa linear code $\mathcal{C}[n, k]$ that can correct $w$ errors
2 derive the generator matrix $G \in \mathbb{F}_2^{k \times n}$ for $\mathcal{C}[n, k]$
3 generate a random invertible dense scrambling matrix $S \in \mathbb{F}_2^{k \times k}$
4 generate a random permutation matrix $P \in \mathbb{F}_2^{n \times n}$
5 $\widehat{G} \leftarrow SGP$
6 $k_{private} \leftarrow (S, P, G)$
7 $k_{public} \leftarrow \left(\widehat{G}, w\right)$
8 **return** $(k_{private}, k_{public})$

**Algorithm 1.13:** McEliece key generation algorithm

**Input:** $k^B_{public}$: the public-key of the receiver
$\mathbf{m} \in \mathbb{F}_2^k$: the message to encrypt
**Output:** $\mathbf{y} \in \mathbb{F}_2^n$: the message encrypted
**Data:** $\mathbf{e} \in \mathbb{F}_2^n$: the error vector

1 $\mathbf{e} \overset{\$}{\in} \mathbb{F}_2^n$, s.t. $\mathrm{HW}(\mathbf{e}) = w$
2 $\mathbf{y} \leftarrow \mathbf{m}SGP + \mathbf{e}$
3 **return** $\mathbf{y}$

**Algorithm 1.14:** McEliece encryption algorithm

Recently, this cryptosystem has been the subject of various optimizations and variants, due to the NIST standardization process, where the Classic McEliece variant [14] was a finalist in the fourth round of the process but was not selected for standardization. This variant retains the original structure of the cryptosystem, introducing optimizations in the parameters and implementation to meet the security and interoperability requirements of modern digital infrastructures.

### 1.5.3. Niederreiter cryptosystem

The Niederreiter cryptosystem, proposed in 1986 by Harald Niederreiter [34], is a variant of the McEliece cryptosystem [32]. Although both cryptosystems are based on code theory, the Niederreiter cryptosystem is distinguished by its use of the *parity-check matrix* $H \in \mathbb{F}_2^{(n-k) \times n}$ in place of the generator matrix $G \in \mathbb{F}_2^{kn}$. This approach, whilst equivalent

**Input:** $k^B_{private}$: the private-key of the receiver
$\mathbf{y} \in \mathbb{F}_2^n$: the message to decrypt
**Output:** $\mathbf{m} \in \mathbb{F}_2^k$: the message decrypted
**Data:** $\mathbf{y}' \in \mathbb{F}_2^n$: the unpermuted ciphertext
$\mathbf{m}' \in \mathbb{F}_2^k$: the scrambled message

1 $\mathbf{y}' \leftarrow \mathbf{y}P^{-1} = \mathbf{m}SG + \mathbf{e}P^{-1}$
2 $\mathbf{m}' \leftarrow \texttt{decode}(G, \mathbf{y}')$
3 $\mathbf{m} \leftarrow \mathbf{m}'S^{-1}$
4 **return** $\mathbf{m}$

**Algorithm 1.15:** McEliece decryption algorithm

in terms of computational security, offers advantages in terms of reducing the size of the public-key and improving algorithmic efficiency.

The security of the cryptosystem is based on SDP, the formulation of which is computationally equivalent to the GDP discussed in relation to McEliece. This problem, like GDP, is known to be NP-hard [6], and forms the basis of the cryptosystem's security.

**Definition 1.5.13** (Syndrome Decoding Problem). *Let a parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ of al linear code $\mathcal{C}[n,k]$, a Hamming weight $w$ of the error vector $\mathbf{e} \in \mathbb{F}_2^n$, and a syndrome vector $\mathbf{s} \in \mathbb{F}_2^{n-k}$.*

*The objective of the Syndrome Decoding Problem (SDP) is to find an error vector $\mathbf{e} \in \mathbb{F}_2^n$ such that*

$$\begin{cases} \mathbf{s}^\intercal = H\mathbf{e}^\intercal \\ \mathrm{HW}(\mathbf{e}) \leq w \end{cases} \tag{1.52}$$

In the original version, Niederreiter proposed using *generalized Reed-Solomon codes*; however, as demonstrated by Vladimir Sidelnikov and Sergei Shestakov [45], such codes introduce structural vulnerabilities that allow the system to be compromised. Consequently, the original family of codes has been replaced by *binary Goppa codes*, bringing the robustness of the scheme into line with that of McEliece.

Niederreiter's cryptosystem is also a public-key cryptosystem; each user must therefore generate a key pair: one public and one private. The generation of the key pair follows a logic similar to that of McEliece, but using the parity-check matrix instead of the generator matrix. The key generation procedure is described in Algorithm 1.16.

The encryption process involves representing the plaintext message $\mathbf{m}$ as an error vector $\mathbf{e}$. To transmit the message $\mathbf{m}$ to Bob, Alice must first convert it into an error vector $\mathbf{e}$. This is done using a *constant-weight encoding algorithm*, which establishes a one-to-one

**Input:** $n \in \mathbb{N}$: the code length
$k < n$: the code dimension
$w < n$: the number of errors the code can correct
**Output:** $(k_{private}, k_{public})$: the key pair
**Data:** $H \in \mathbb{F}_2^{(n-k)\times n}$: the parity-check matrix for the Goppa linear code
$S \in \mathbb{F}_2^{(n-k)\times(n-k)}$: the random invertible dense scrambling matrix
$P \in \mathbb{F}_2^{n\times n}$: the random permutation matrix
$\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the public matrix

1 generate a random Goppa linear code $\mathcal{C}[n,k]$ that can correct $w$ errors
2 derive the parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ for $\mathcal{C}[n,k]$
3 generate a random invertible dense scrambling matrix $S \in \mathbb{F}_2^{(n-k)\times(n-k)}$
4 generate a random permutation matrix $P \in \mathbb{F}_2^{n\times n}$
5 $\widehat{H} \leftarrow SHP$
6 $k_{private} \leftarrow (S, P, H)$
7 $k_{public} \leftarrow \left(\widehat{H}, w\right)$
8 **return** $(k_{private}, k_{public})$

**Algorithm 1.16:** Niederreiter key generation algorithm

**Input:** $k^B_{public}$: the public-key of the receiver
$\mathbf{m} \in \mathbb{F}_2^k$: the message to encrypt
**Output:** $\mathbf{s} \in \mathbb{F}_2^{n-k}$: the message encrypted
**Data:** $\mathbf{e} \in \mathbb{F}_2^n$: the error vector

1 $\mathbf{e} \leftarrow$ `constantWeigthEncoding(m)`
2 $\mathbf{s}^\intercal \leftarrow \widehat{H}\mathbf{e}^\intercal$
3 **return** $\mathbf{s}$

**Algorithm 1.17:** Niederreiter encryption algorithm

correspondence between the plaintext message and the permissible error patterns. The complete encryption process is described by Algorithm 1.17. Upon receiving the syndrome $\mathbf{s}$, Bob must derive the error vector $\mathbf{e}$ by solving the SDP using his own private-key and subsequently apply a constant-weight decoding algorithm to obtain the plaintext message sent by Alice. The decryption process is set out in Algorithm 1.18.

Compared to McEliece's cryptosystem, Niederreiter's cryptosystem offers a number of advantages. In particular, the ciphertext is more compact, as it consists of a syndrome of size $n-k$ rather than a codeword of size $n$. Furthermore, this structure lends itself naturally to the construction of digital signature schemes, in which the signature can be interpreted as a solution to the decoding problem of an assigned syndrome. As with the GDP, the difficulty of the SDP is fundamental to the security of the cryptosystem. State-of-the-art attacks against this problem rely on ISD techniques. Although the problem is

**Input:** $k^B_{private}$: the private-key of the receiver
$\mathbf{s} \in \mathbb{F}_2^{n-k}$: the message to decrypt
**Output:** $\mathbf{m} \in \mathbb{F}_2^k$: the message decrypted
**Data:** $\mathbf{s}' \in \mathbb{F}_2^{n-k}$: the unscrambled syndrome
$\mathbf{e}' \in \mathbb{F}_2^n$: the permuted error vector
$\mathbf{e} \in \mathbb{F}_2^n$: the error vector

1 $\mathbf{s}'^\intercal \leftarrow S^{-1}\mathbf{s}^\intercal$
2 $\mathbf{e}' \leftarrow$ `syndromeDecode(`$H$, $\mathbf{s}'$`)`
3 $\mathbf{e}^\intercal \leftarrow P^{-1}\mathbf{e}'^\intercal$
4 $\mathbf{m} \leftarrow$ `constantWeigthDecoding(e)`
5 **return** $\mathbf{m}$

**Algorithm 1.18:** Niederreiter decryption algorithm

formulated using the parity-check matrix, these algorithms maintain a similar exponential complexity, ensuring that Niederreiter's variant remains as robust as McEliece's original when appropriate parameters are selected.

## 1.6. CryptAttackTester

Quantitative analysis of the cost of attack algorithms against cryptographic systems is fundamental to modern cryptography: it guides the choice of security parameters, drives research towards more efficient attacks, and determines which cryptosystems are standardized. However, there are several examples in the literature of how these analyses are prone to errors, which sometimes take years to detect, with concrete repercussions on the security of cryptosystems ([42], [22], [13], [18], [16]).

The underlying problem with these incidents is that state of the art analyses of cryptographic attacks are, in some cases, not formal proofs. These analyses are based on heuristic estimates, conjectures and experimental simulations. Often, simulations are carried out informally, without a precisely defined computational model or cost metric, which makes it difficult to identify errors.

To address these challenges, in 2023, Bernstein and Chou introduced CAT [9], a software framework designed for the highly-assurance quantification of the effectiveness of cryptographic attacks.

The underlying idea behind CAT is to replace the informal process of analyzing cryptographic attacks with a *fully formalized and automatically verifiable chain*. Thus, each analysis process must formally define: the computational model and cost metric to be used, the problem to be attacked, the attack algorithm to be used within the previously

chosen computational model, the formula for predicting the cost and probability of success for the attack algorithm, and, finally, simulate the attack algorithm and automatically compare the observed cost and probability with the predicted values.

Although this approach does not yield mathematically formal proofs, a goal that is unattainable for heuristic-based analyses, it eliminates ambiguity in the definitions and immediately highlights the discrepancies between the predictions and the actual behavior of the attack algorithm. It is important to emphasis that the comparison between the prediction and the simulation is sufficient to detect errors arising from inconsistencies between the various components of the analysis, which would have been immediately apparent had all components been explicitly linked and automatically verifiable.

In CAT, each component of the analysis (computational model, problem, attack algorithm, cost formula, probability formula) is defined independently and linked to the others via explicit interfaces. Thanks to this approach, it is possible to review and scrutinies each component separately without having to go through the entire analysis from the start.

### 1.6.1. Computational Model and Cost Metric

CAT uses a *boolean-circuit model* as its computational model, in which attack algorithms are represented as logic circuits that transform an input bit sequence into an output bit sequence by sequentially applying elementary logical operations.

More precisely, the circuit used by CAT is an *a-bit-to-b-bit circuit*, with the set of permissible gates comprising all Boolean functions with at most 2 bit inputs.

**Definition 1.6.1** ($a$-bit-to-$b$-bit Circuit)**.** *The a-bit-to-b-bit circuit is a circuit that converts an a-bit input into a b-bit output. Internally, the circuit consists of a sequence of gates (at least as many as b) that iteratively transform the input bits into the output bits. The gates used in this model are all those that require zero (the constants), one or two bits as input.*

The cost metric used by CAT is the total number of one-bit or two-bit input gates used in the $a$-bit-to-$b$-bit circuit. Therefore, the Boolean logic functions that contribute to the cost are: NOT, AND, OR, XOR, XNOR, NAND, ANDN ($a \wedge \neg b$), ORN ($\neg a \vee b$) and NOR. The constants 0 and 1, and the copy of a previously computed bit, have a cost of zero. The cost of interconnections between gates and propagation delays are not considered in this cost model. The 2 bit gates have all the same costs.

The choice of the Boolean circuit model over the more common Random-Access Memory

(RAM) model is motivated by the fact that CAT aims to quantify the cost of an attack in cases where the adversary has access to fully combinatorial dedicated hardware, i.e. a circuit designed specifically to carry out an attack against a specific problem in the most efficient way possible. This scenario is the most relevant when considering an adversary with substantial resources who is not constrained to using General-Purpose Processors (GPPs) but can build ASICs optimized for the specific problem.

RAM models have a structural flaw: they assign a unit cost to random access to an array of arbitrary size. This assumption does not hold true in reality, as in a real circuit, accessing an array of size $n$ requires traversing a multiplexer tree, with a cost that increases as the size increases. As a result, RAM models are systematically further removed from the actual cost-performance ratio than the Boolean circuit model, contrary to the common perception that they are more realistic due to their similarity to Central Processing Unit (CPU) instructions.

### 1.6.2. Architecture and Interface

CAT is implemented in C++. Internally, each attack is implemented as a *meta-circuit*: a function which, given the problem parameters and the attack parameters, constructs the corresponding circuit in the computational model described above. The `bit` class, defined in `bit.h`, replaces the standard Boolean types. Every logical operation performed on `bit`-type objects not only produces the Boolean result, but also automatically tracks the cost of the logical operation during circuit execution; this makes the counting of operations an integral part of circuit execution rather than a separate, error-prone step.

The CAT external interface is organized into three categories of functions that perform specific tasks. All three categories of functions are parameterized by the problem to be analyses, along with its parameters, and by the attack under consideration, along with its parameters.

The first category of functions is that of *predictors*, which provide an analytical estimate of the cost in terms of logic gates or of the circuit's probability of success, without performing any hardware synthesis. Instead, they are based on a detailed analytical model that operates exclusively on the parameters the circuit would have if it were synthesized, and on cryptographic parameters.

Specifically, the cost predictor calculates the exact number of logic gates required to implement the circuit. This is achieved by taking into account, when programming the cost predictor, the various gates required for each subcomponent, and multiplying them by the exact number of iterations of the algorithm, avoiding crude asymptotic approximations

in favour of a concrete assessment of resources. The success probability predictor models the algorithm's operational behavior as a stochastic process; this makes it possible to instantly assess the cryptographic strength of certain parameters without the enormous computational overhead associated with physical execution.

The second category of functions are the *simulators*, which perform the circuit simulation and return the actual cost, the actual probability of obtaining the correct solution (estimated over a configurable number of runs), the execution time (estimated over a configurable number of runs), or the output calculated by the circuit. If the simulated cost differs from the predicted cost, or the observed probability deviates by more than 10 % from the predicted probability, the simulator automatically issues an alert.

Finally, the last category consists of the *explorers*, which generate sequences of problem and attack parameters on which to run the tests, and perform the search for optimal attack parameters given the problem parameters. The goal is to assist the design of code-based cryptosystems.

### 1.6.3. The Uniform Matrix Problem

The problem that CAT [9] uses as the target for ISD attacks is called `uniformmatrix`. This formulation represents an instance of the SDP.

The problem is defined by a triple of parameters, namely: the length $n$, the dimension $k$ of the binary linear code, and the Hamming error weight $w$. CAT provides default value ranges for these parameters targeted for the Classical McEliece [14] using a Niederreiter structure [34], but it is possible to specify particular instances of the problem by setting the desired values.

An instance of the problem, given the parameters, is characterized by: a matrix $V \in \mathbb{F}_2^{(n-k)\times k}$ generated uniformly at random, representing the dense part of the parity-check matrix $H$, and a syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$, generated from an error vector $\mathbf{e} \in \mathbb{F}_2^n$, which is also generated randomly.

In this context, the *dense part of the parity-check matrix* refers to the non-identity submatrix obtained after performing Gaussian elimination to bring the matrix into its standard form. Formally, given a parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ of a linear code, it can be mathematically transformed via a process of scrambling and permutation into a systematic representation, expressed as $\widehat{H} = \begin{bmatrix} V & I_{n-k} \end{bmatrix}$.

Whilst $I_{n-k}$ represents the sparse identity matrix of dimension $n-k$, the submatrix $V \in \mathbb{F}_2^{(n-k)\times k}$ constitutes the *dense part*. It is defined as "dense" because, in the case of a linear

random or pseudo-random code, its entries are uniformly distributed over $\mathbb{F}_2$, meaning that each bit has a probability of $\frac{1}{2}$ of being non-zero. This dense submatrix concentrates the computational complexity during the ISD routines, as it defines the linear combinations of the syndrome equations that the adversary must solve.

In the implementation of CAT, only the dense submatrix $V$ is used as the key representation, rather than the entire parity-check matrix $H$. This choice is justified both by mathematical redundancy and computational efficiency. Indeed, once the matrix has been reduced to its systematic form, the identity block $I_{n-k}$ is deterministic and carries no cryptographic entropy. Since its structure is static and implicitly known to the algorithm, storing it is redundant. Storing the full matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ would entail a huge memory overhead. By isolating only the dense component $V \in \mathbb{F}_2^{(n-k)\times k}$, CAT drastically reduces the memory required.

The objective of the problem is to solve the SDP, i.e. to find the uniformly generated error vector, $\mathbf{e}$, from the known information, $(n, k, w)$, $V$ and $\mathbf{s}$.

The `uniformmatrix` problem can also be interpreted as an instance of the One-Wayness under Chosen Plaintext Attack (OW-CPA) problem for a public-key cryptosystem, in which the public-key is the matrix $V$, the encryption of the vector $\mathbf{e}$ produces the vector $\mathbf{s}$, and the decryption function always fails.

This formulation of the problem is simpler than the actual McEliece cryptosystem [32], where the parity-check matrix has an additional algebraic structure derived from Goppa codes. However, for parameters of cryptographic interest, all known methods for distinguishing between the two formulations are much slower than solving the SDP; this makes `uniformmatrix` an equivalent target for the purposes of security analysis.

The choice to work with uniformly random matrices allows CAT to scale the problem parameters, $(n, k, w)$, to small values whilst maintaining the same structure of the problem, thereby making it possible to simulate and experimentally verify predictions on reduced instances before extrapolating the results to real cryptographic parameters.

# 2 | The Information Set Decoding Algorithms Implemented in CryptAttackTester

Code-based cryptosystems derive their security from the computational difficulty of solving the GDP [6] or the SDP; as the two problems are equivalent, we will focus on analyzing the SDP without loss of generality.

The most significant family of algorithms for attacking these two problems to date is *ISD*. The aim of ISD algorithms, in solving the SDP, is to find an error $\mathbf{e} \in \mathbb{F}_2^n$ given a parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$ and a syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$ more efficiently than by performing an exhaustive search.

ISD algorithms were originally introduced by Prange in 1962 [38]; these algorithms aim to solve the decoding problem regardless of any specific algebraic structure of the code, which has made them particularly relevant in the security analysis of code-based cryptosystems. Over the years, Prange's algorithm has undergone numerous optimizations that have improved its performance both asymptotically and in practice.

In this chapter, we examine in detail the formalization of the ISD algorithms implemented by the CAT framework [9]. The ISD algorithms formalized in CAT are: Prange [38], Lee-Brickell [27], Leon [28], Ball-Collision [12], Stern [46], MMT [31] and BJMM [5]. These ISD algorithms, are combined with the random walk technique introduced by Omura [21] and subsequently generalized by Bernstein-Lange-Peters [11].

## 2.1. High-Level Information Set Decoding Algorithm

The basic idea behind every ISD algorithm is to randomly select an IS, $\mathbf{I}$, and assume that the error vector $\mathbf{e}$ follows a specific distribution associated with that set. The error vector $\mathbf{e}$ is ideally split into two disjoint subvectors, $\mathbf{e}_\mathbf{s} \in \mathbb{F}_2^k$ and $\mathbf{e}_{\mathbf{s}^*} \in \mathbb{F}_2^{n-k}$, corresponding to the positions indexed by the IS and the RS respectively.

Once the IS has been selected, the algorithm applies a linear transformation designed to simplify the system of equations. Using a permutation matrix $P \in \mathbb{F}_2^{n\times n}$, the columns of the parity-check matrix $H$ are reordered so that those corresponding to the IS occupy the first $k$ positions, and consequently the columns of the RS occupy the remaining $n-k$ positions. Subsequently, via Gaussian elimination, represented by the non-singular matrix $U \in \mathbb{F}_2^{(n-k)\times(n-k)}$, the parity-check matrix $H$ is brought into reduced row-echelon form: $\widehat{H} = \begin{bmatrix} V & I_{n-k} \end{bmatrix}$. The transformations applied to the matrix are also applied to the syndrome, $\widehat{\mathbf{s}}^\intercal = U\mathbf{s}^\intercal$, and the error vector, $\widehat{\mathbf{e}}^\intercal = P^\intercal \mathbf{e}^\intercal$. In this new representation, the permuted error vector, $\widehat{\mathbf{e}}$, is partitioned into two contiguous blocks: the first $k$ elements represent the error in the IS, $\widehat{\mathbf{e}_\mathbf{s}}$, whilst the remaining $n-k$ represent the error in the redundancy, $\widehat{\mathbf{e}_{\mathbf{s}^*}}$.

By applying Theorem 1.2, we can transform our original problem $\mathbf{s}^\intercal = H\mathbf{e}^\intercal$ into its equivalent form $\widehat{\mathbf{s}}^\intercal = \widehat{H}\widehat{\mathbf{e}}^\intercal$. Given the nature of the transformation, the equivalent parity-check matrix is in systematic form $\widehat{H} = \begin{bmatrix} V & I_{n-k} \end{bmatrix}$, so we can rewrite the system as follows:

$$\widehat{\mathbf{s}}^\intercal = \widehat{H}\widehat{\mathbf{e}}^\intercal \tag{2.1}$$

$$\widehat{\mathbf{s}}^\intercal = \begin{bmatrix} V & I_{n-k} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{e}_\mathbf{s}}^\intercal \\ \widehat{\mathbf{e}_{\mathbf{s}^*}}^\intercal \end{bmatrix} \tag{2.2}$$

$$\widehat{\mathbf{s}}^\intercal = V\widehat{\mathbf{e}_\mathbf{s}}^\intercal + I_{n-k}\widehat{\mathbf{e}_{\mathbf{s}^*}}^\intercal \tag{2.3}$$

$$\widehat{\mathbf{s}}^\intercal = V\widehat{\mathbf{e}_\mathbf{s}}^\intercal + \widehat{\mathbf{e}_{\mathbf{s}^*}}^\intercal \tag{2.4}$$

The algorithm assumes that $\widehat{\mathbf{e}_\mathbf{s}}$ has a fixed Hamming weight of $p$. This assumption allows the remaining part of the error $\widehat{\mathbf{e}_{\mathbf{s}^*}}$ to be uniquely determined using the Theorem 1.1. From the Equation 2.4 equation, we find that:

$$\widehat{\mathbf{e}_{\mathbf{s}^*}}^\intercal = \widehat{\mathbf{s}}^\intercal - V\widehat{\mathbf{e}_\mathbf{s}}^\intercal \tag{2.5}$$

The algorithm succeeds if and only if the Hamming weight of the calculated vector $\widehat{\mathbf{e}_{\mathbf{s}^*}}$ is $w-p$. If the algorithm finds an error vector that satisfies these conditions, the original vector is reconstructed, $\mathbf{e} = \widehat{\mathbf{e}}P^{-1}$.

The efficiency of ISD algorithms depends on the probability of selecting a favorable IS; therefore, as these are probabilistic algorithms, the procedure described above is repeated several times, selecting a new IS until a solution is found.

Table 2.1 shows all the parameters used by CAT, accompanied by a brief description and an indication of the connection variants in which they are used.

To provide a unified taxonomic overview of how individual ISD algorithms are represented within CAT, Table 2.3 details the parametric configurations that characterize each historical variant. The design philosophy behind the framework treats the entire spectrum of ISD attacks not as isolated procedures, but as specific algorithmic tuning profiles of a generalized search structure.

## 2.2. Attack Overview

Each ISD attack implemented in CAT [9] takes as input the attack parameters, a matrix $V \in \mathbb{F}_2^{(n-k)\times k}$ and a vector $\mathbf{s} \in \mathbb{F}_2^{n-k}$, and returns a vector $\mathbf{e} \in \mathbb{F}_2^n$. The matrix $V$ is used to construct the parity-check matrix $H = \begin{bmatrix} V^\intercal & I_{n-k} \end{bmatrix}$, whilst the vector $\mathbf{s}$ represents the syndrome and the vector $\mathbf{e}$ is an attempt to recover the error vector that generated it.

In CAT, all ISD attacks have a binary parameter, `FW`. If active (`FW` = 1), the $H$ matrix is extended by an additional row consisting entirely of ones, and the Hamming weight of the error modulo 2 is added to the syndrome. This step exploits knowledge of the error weight to impose an additional constraint on the system, reducing the dimension $k$ by one and consequently the search space [19].

$$H = \begin{bmatrix} V & I_{n-k} \\ \mathbf{1} & \mathbf{1} \end{bmatrix} \tag{2.6}$$

$$\mathbf{s} = \begin{bmatrix} \mathbf{s} & w \bmod 2 \end{bmatrix} \tag{2.7}$$

At a high level, each attack consists of a sequence of `I` iterations followed by a post processing phase. Each iteration comprises two sequential phases: the column permutation phase and the search phase. It is important to note that both the column permutation and post processing phases are independent of the variant of the ISD algorithm used, whereas the search phase is specific to the variant employed. In Algorithm 2.1, we can see the high-level structure of each ISD attack implemented in CAT.

During the search phase, we seek a permuted error vector $\widehat{\mathbf{e}}$ that satisfies the transformed system of equations. This phase is specific to each variant of the ISD approach, and we will examine each variant in detail later. CAT implements three families of attacks based on the number of collision-search levels: `isd0` (no collision levels), `isd1` (one collision level) and `isd2` (two collision levels). During the search phase, a parameter common to all variants, `Z`, is used; this reduces the number of columns in the matrix $\widehat{H}$ considered during the search phase from $k$+`L` to $k$+`L`−`Z`. The use of this parameter introduces a new trade-off between computational cost and probability of success per iteration, since as `Z`

**Input:** $\mathbf{s} \in \mathbb{F}_2^{n-k}$: the syndrome
$V \in \mathbb{F}_2^{(n-k)\times k}$: the dense part of the parity-check matrix
$w$: the Hamming weight of the error
**Output:** $\mathbf{e} \in \mathbb{F}_2^n$: the target error of the syndrome problem
**Data:** $\mathtt{I} \in \mathbb{Z}$: the number of iterations
$H \in \mathbb{F}_2^{(n-k)\times n}$: the parity-check matrix
$\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the permuted parity-check matrix
$\widehat{\mathbf{s}} \in \mathbb{F}_2^{n-k}$: the permuted syndrome
$\widehat{\mathbf{e}} \in \mathbb{F}_2^n$: the permuted error

1 $H \leftarrow \begin{bmatrix} V^\intercal & I_{n-k} \end{bmatrix}$
2 **if** $\mathtt{FW} = 1$ **then**
3 $\quad H \leftarrow \begin{bmatrix} H \\ \mathbf{1} \end{bmatrix}$
4 $\quad \mathbf{s} \leftarrow \begin{bmatrix} \mathbf{s} & w \bmod 2 \end{bmatrix}$
5 **for** $i \leftarrow 0$ **to** $\mathtt{I} - 1$ **do**
6 $\quad (\widehat{H}, \widehat{\mathbf{s}}, P) \leftarrow \mathtt{columnPermutationPhase}(H,\ \widehat{H},\ \mathbf{s},\ \widehat{\mathbf{s}},\ i)$
7 $\quad \widehat{\mathbf{e}} \leftarrow \mathtt{searchPhase}(\widehat{H},\ \widehat{\mathbf{s}})$
8 $\mathbf{e} \leftarrow \mathtt{postProcessingPhase}(\widehat{\mathbf{e}},\ P)$
9 **return** $\mathbf{e}$

**Algorithm 2.1:** High-level ISD algorithm in CAT

increases, both the computational cost, as work is carried out on a smaller submatrix, and the probability of success per iteration, as the search space is reduced, decrease.

## 2.3. Column Permutation Phase

The column permutation phase transforms the matrix $H$ into partial systematic form by applying column permutations and partial Gaussian reduction. In this phase, the syndrome $\mathbf{s}$ is permuted by applying the same permutation operations as those applied to the parity-check matrix $H$.

The systematic reduction operation represents a computational bottleneck in ISD algorithms due to a combination of algorithmic complexity and hardware constraints. Mathematically, converting the randomly permuted parity-check matrix into its systematic form $\begin{bmatrix} V & I_{n-k} \end{bmatrix}$ requires the execution of Gaussian elimination, which entails a deterministic computational complexity of $\mathcal{O}(n^3)$ operations. Since this reduction must be repeated at every global reset cycle whenever the current IS proves unsuccessful, the cumulative overhead becomes high. From a hardware architecture perspective, Gaussian elimination is inherently non-local, requiring continuous global synchronization and massive XOR operations across entire rows. Furthermore, the probability of encountering singular sub-

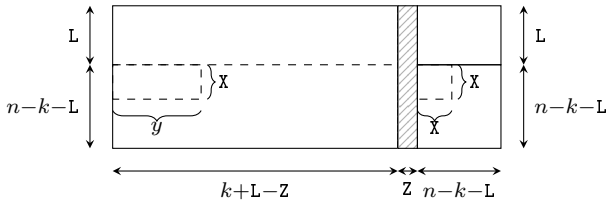


Figure 2.1: The attack parameters $\mathtt{X}$ and $y = \mathtt{X} + \mathtt{YX}$

matrices forces the system to abort and restart the reduction, wasting clock cycles and further limiting the overall effective speed of the attack.

To address this issue, CAT [9] employs the *random walk* technique, originally introduced by Omura [21] and subsequently generalized by Bernstein-Lange-Peters [11].

The underlying idea is to abandon the classical approach, which involves sampling a completely new and independent IS at every single iteration, opting instead for an incremental and localized modification to the parity-check matrix available in the current iteration. Rather than discarding the entire identity block and reconstructing it from scratch via a full Gaussian elimination, the random walk allows for a limited number of column swaps between the current IS and a small set of candidate columns outside it.

Formally, at each incremental step a local swap is performed: a number $\mathtt{X}$ of columns currently belonging to the IS is removed and replaced by an equal number $\mathtt{X}$ of columns selected from a designated set of candidate columns of size $y = \mathtt{X}+\mathtt{YX}$. Since the structural modification is highly localized, the computational cost of updating the matrix does not depend on the overall size of the code, but exclusively on the parameters $\mathtt{X}$ and $y$, as shown in Figure 2.1. Since $\mathtt{X}, y \ll (n-k), k$, the algorithmic complexity of selecting a new IS decreases drastically compared to a full Gaussian reduction.

To organism this dynamic process efficiently, without losing the probabilistic benefits of global randomization, the *iterations are grouped into structured chains* of fixed length, the size of which is determined by the $\mathtt{RE}$ parameter. At the beginning of each chain, a hard reset is triggered, in which a new initial state (IS) is generated via a Gaussian reduction. This complex operation establishes a new and mathematically valid baseline state, albeit at a high computational cost. For the remaining iterations within the chain, the algorithm explores the combinatorial space by performing only local swaps. By applying only localized row operations to calculate the new pivots of the swapped columns, a valid representation is maintained across the subsequent steps, completely bypassing a further Gaussian reduction until the end of the chain.

However, for random walks to function efficiently, the structure of the matrix $\widehat{H}$ must

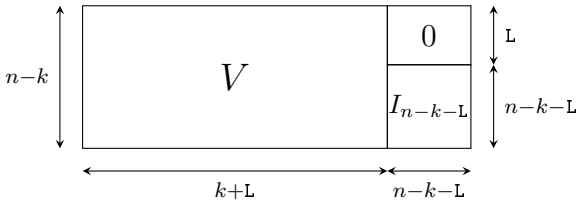


Figure 2.2: Partial systematic form of the parity-check matrix

support local updates without requiring a full Gaussian reduction to restore the systematic form.

The complete systematic form, expressed as $\begin{bmatrix} V & I_{n-k} \end{bmatrix}$, does not satisfy this operational requirement. In this form, every single row of the redundancy block is rigidly tied to a specific and unique pivot column within the identity matrix $I_{n-k}$. If a local swap replaces a column of the identity block $I_{n-k}$ with a column of the dense matrix $V$, the identity block is immediately compromised. The newly introduced column has non-zero elements spread across multiple rows, invalidating the previous diagonalization of the pivots. To eliminate these spurious elements and restore the identity block required for calculating the syndrome equations, the algorithm would be forced to perform a complete cascade of row reductions across the entire matrix. This would completely negate the computational savings promised by the random walk, reducing the performance of the local update to the cubic complexity of a standard global reset. The complete random walk procedure used is shown in Algorithm 2.2.

To resolve this issue, CAT avoids the full systematic representation and instead adopts the highly flexible *partial systematic form*, which is mathematically parameterized by an internal window controlled by the `L` parameter. In this form, the matrix $\widehat{H}$ takes on the structure illustrated in Figure 2.2.

The implementation of the `columnPermutationPhase` function is shown in Algorithm 2.3.

## 2.4. Search Phase

The search phase is the computational core in which the search for error vectors with Hamming weight $w$. Once the geometric structure of the code has been defined and the problem parameters configured, the search phase aims to identify a linear combination of columns of the parity-check matrix $\widehat{H}$ capable of reconstructing the syndrome $\widehat{\mathbf{s}}$.

In CAT [9], there are three different algorithms ISD based on the number of collision search levels used, representing all the variants currently present in the state-of-art.

**Input:** $\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the permuted parity-check matrix
$\widehat{\mathbf{s}} \in \mathbb{F}_2^{n-k}$: the permuted syndrome
$P \in \mathbb{F}_2^{n\times n}$: the permutation matrix
$i \in \mathbb{Z}$: the current iteration
**Output:** $\widehat{H'} \in \mathbb{F}_2^{(n-k)\times n}$: the new permuted parity-check matrix
$\widehat{\mathbf{s}'} \in \mathbb{F}_2^{n-k}$: the new permuted syndrome
$P \in \mathbb{F}_2^{n\times n}$: the permutation matrix
**Data:** $y \in \mathbb{Z}$: the dimension of the set of candidate columns to swap
$P_1 \in \mathbb{F}_2^{(k+\mathtt{L})\times(k+\mathtt{L})}$: the permutation matrix applied to the first block of columns
$P_2 \in \mathbb{F}_2^{(n-(k+\mathtt{L}))\times(n-(k+\mathtt{L}))}$: the permutation matrix applied to the second block

$y \leftarrow \mathtt{X} + \mathtt{YX}$
$\pi_1 \xleftarrow{\$} \mathbf{S}_{\mathbf{k}+\mathtt{L}}$
$P_1 \leftarrow \mathtt{permutationMatrix}(\pi_1)$
$\pi_2 \xleftarrow{\$} \mathbf{S}_{\mathbf{n}-(\mathbf{k}+\mathtt{L})}$
$P_2 \leftarrow \mathtt{permutationMatrix}(\pi_2)$
$P \leftarrow P \begin{bmatrix} P_1 & 0 \\ 0 & I_{n-(k+\mathtt{L})} \end{bmatrix}$
$P \leftarrow P \begin{bmatrix} I_{k+\mathtt{L}} & 0 \\ 0 & P_2 \end{bmatrix}$
$H' \leftarrow HP_1$
**for** $j \leftarrow 0$ **to** $k + \mathtt{L} - 1$ **do**
  $\mathbf{h'}^{\intercal}_{(:,j)} \leftarrow \begin{bmatrix} I_{\mathtt{L}} & 0 \\ 0 & P_2 \end{bmatrix} \mathbf{h'}^{\intercal}_{(:,j)}$
$\widehat{\mathbf{s}'}^{\intercal} \leftarrow \begin{bmatrix} I_{\mathtt{L}} & 0 \\ 0 & P_2 \end{bmatrix} \widehat{\mathbf{s}}^{\intercal}$
// Gauss Jordan elimination on X rows
$\left(\widehat{H'}_{(\{\mathtt{L},\ldots,\mathtt{L}+\mathtt{X}-1\},\{0,\ldots,k+\mathtt{L}+\mathtt{X}-1\})},\ \widehat{\mathbf{s}'}_{\{\mathtt{L},\ldots,\mathtt{L}+\mathtt{X}-1\}}\right) \leftarrow$
$\mathtt{gaussJordanElimination}(\widehat{H'}_{(\{\mathtt{L},\ldots,\mathtt{L}+\mathtt{X}-1\},\{0,\ldots,k+\mathtt{L}+\mathtt{X}-1\})},\ \widehat{\mathbf{s}'}_{\{\mathtt{L},\ldots,\mathtt{L}+\mathtt{X}-1\}})$
$\left(\widehat{H'},\ \widehat{\mathbf{s}'}\right) \leftarrow \mathtt{fixSystematicForm}(\widehat{H'},\ \widehat{\mathbf{s}'})$
**return** $(\widehat{H'}, \widehat{\mathbf{s}'}, P)$

**Algorithm 2.2:** Random walk algorithm

### 2.4.1. Search Phase with Zero Collision Levels

`isd0` is the first variant of the ISD algorithms implemented in CAT [9], and does not use any collision level during the search phase. The attack follows the structure shown in Section 2.2; the difference resides in the implementation of the search phase. The ISD algorithms that fall under this variant are Prange [38] ($\mathtt{P} = 0$, $\mathtt{L} = 0$, $\mathtt{Z} = 0$), Ball-Collision [12] ($\mathtt{P} = 0$, $\mathtt{L} > 0$) Lee-Brickell [27] ($\mathtt{P} > 0$, $\mathtt{L} = 0$, $\mathtt{Z} = 0$) and Leon [28] ($\mathtt{P} > 0$, $\mathtt{L} > 0$,

**Input:** $\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the permuted parity-check matrix
$\widehat{\mathbf{s}} \in \mathbb{F}_2^{n-k}$: the permuted syndrome
$i \in \mathbb{Z}$: the current iteration
**Output:** $\widehat{H'} \in \mathbb{F}_2^{(n-k)\times n}$: the new permuted parity-check matrix
$\widehat{\mathbf{s}'} \in \mathbb{F}_2^{n-k}$: the new permuted syndrome
$P \in \mathbb{F}_2^{n\times n}$: the permutation matrix

1 **if** $i \bmod \mathtt{RE} \neq 0$ **then**
2 $\quad (\widehat{H'}, \widehat{\mathbf{s}'}, P) \leftarrow \mathtt{randomWalk}(\widehat{H}, \mathbf{s}, P, i)$
3 **else**
4 $\quad (\widehat{H'}, \widehat{\mathbf{s}'}, P) \leftarrow \mathtt{gaussJordanElimination}(\widehat{H}, \mathbf{s})$
5 **return** $(\widehat{H'}, \widehat{\mathbf{s}'}, P)$

**Algorithm 2.3:** Column permutation phase

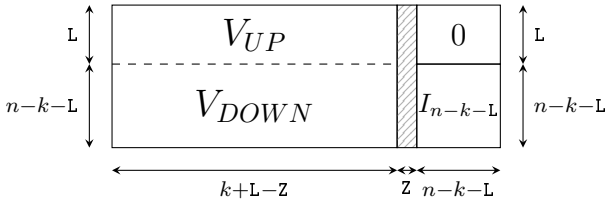


Figure 2.3: Partial systematic form of the parity-check matrix for `isd0`

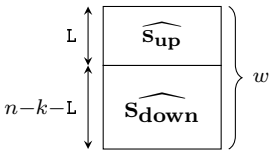


Figure 2.4: The syndrome division used in `isd0`

$\mathtt{Z} = 0$).

The parity-check matrix in partial systematic form $\widehat{H}$, obtained during the column permutation phase, on which collisions will be sought, is shown in Figure 2.3, whilst the permuted syndrome is shown in Figure 2.4.

The search phase of `isd0` begins by enumerating all subsets of `P` columns among the first $k+\mathtt{L}-\mathtt{Z}$ columns of matrix $\widehat{H}$, yielding the vector indexed by the IS, and the respective partial sums are calculated by considering only the upper part of the parity-check matrix, as shown in Equation 2.8, which represents the systematic syndrome equation (Equation 2.4) with the conventions adopted in this attack. The specific behavior of the attack algorithm depends on the values of the parameters `P` and `L`.

$$\begin{cases} \widehat{\mathbf{s_{up}}}^{\intercal} = V_{UP}\,\widehat{\mathbf{e}}_{\mathbf{s}}^{\intercal} + \mathbf{0}^{\intercal} \\ \widehat{\mathbf{s_{down}}}^{\intercal} = V_{DOWN}\,\widehat{\mathbf{e}}_{\mathbf{s}}^{\intercal} + \widehat{\mathbf{e}'_{\mathbf{s}^*}}^{\intercal} \end{cases} \tag{2.8}$$

The `isd0` implementation introduces, for the variant relating to the Leon algorithm, the use of a queue of size `QU` to manage the candidates that pass the preliminary filter. Candidates that pass this filter are accumulated and then processed in batches every $\texttt{QF} \cdot \texttt{QU}$ checks. This mechanism introduces a trade-off between computational cost and probability of success per iteration, as reducing the queue size decreases the cost of managing candidates but increases the probability of losing valid candidates due to potential overflow.

The implementation of the `searchPhase` function for the `isd0` variant is shown in Algorithm 2.4.

$\texttt{P} = 0$ It holds that the lower part of the syndrome has a Hamming weight exactly equal to $w$. The vector can therefore be represented as

$$\widehat{\mathbf{e}} = \begin{bmatrix} \mathbf{0} & \widehat{\mathbf{e}_{\mathbf{s}^*}} \end{bmatrix} \tag{2.9}$$

where the two blocks have dimensions $k{+}\texttt{L}$ and $n{-}k{-}\texttt{L}$ respectively. In this case, the systematic syndrome equation reduces to the form shown in Equation 2.11.

$$\widehat{\mathbf{s}}^{\intercal} = V\mathbf{0}^{\intercal} + \widehat{\mathbf{e}_{\mathbf{s}^*}}^{\intercal} \tag{2.10}$$

$$\widehat{\mathbf{s}}^{\intercal} = \widehat{\mathbf{e}_{\mathbf{s}^*}}^{\intercal} \tag{2.11}$$

Furthermore, if $\texttt{L} = 0$, the `isd0` algorithm implements Prange's [38] variant of the ISD algorithm. In this variant, it is assumed that the permuted vector $\widehat{\mathbf{e}}^{\intercal}$ has the first $k{+}\texttt{L}{-}\texttt{Z}$ positions set to zero and therefore the entire Hamming weight of the error, $w$, is concentrated in the remaining $n{-}k{-}\texttt{L}$ positions.

If, on the other hand, $\texttt{L} > 0$, the Ball-Collision [12] variant is implemented: it is verified that the upper part of the syndrome is the zero vector and that the lower part has weight $w$.

**Input:** $\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the permuted parity-check matrix
$\widehat{\mathbf{s}} \in \mathbb{F}_2^{n-k}$: the permuted syndrome
**Output:** $\widehat{\mathbf{e}} \in \mathbb{F}_2^n$: the permuted error
**Data:** *list*: the list containing all possible vectors $\widehat{\mathbf{e}_\mathbf{s}}$ and their partial sums
*queue*: the queue that manage the candidates that pass the preliminary filter
*timer* $\in \mathbb{Z}$: the counter used to trigger the periodic queue flush

$V_{UP} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L}-1\},\,\{0,\ldots,k+\mathtt{L}-\mathtt{Z}-1\})}$
$V_{DOWN} \leftarrow \widehat{H}_{(\{\mathtt{L},\ldots,n-k-1\},\,\{0,\ldots,k+\mathtt{L}-\mathtt{Z}-1\})}$
$\widehat{\mathbf{s}_\mathbf{up}} \leftarrow \widehat{\mathbf{s}}_{\{0,\ldots,\mathtt{L}-1\}}$
$\mathbf{s}_{\widehat{\mathbf{down}}} \leftarrow \widehat{\mathbf{s}}_{\{\mathtt{L},\ldots,n-k-1\}}$
$list \leftarrow \{\}$
**if** $\mathtt{L} = 0$ **then**
  $list \leftarrow$ `exhaustiveEnumeration(`$V_{DOWN}$, $\widehat{\mathbf{s}_\mathbf{down}}$, `P) // each list element contains` $(\widehat{\mathbf{e}_\mathbf{s}},\ \widehat{\mathbf{s}_\mathbf{down}}^\intercal - V_{DOWN}\,\widehat{\mathbf{e}_\mathbf{s}}^\intercal)$
**else**
  $list \leftarrow$ `exhaustiveEnumeration(`$V_{UP}$, $\widehat{\mathbf{s}_\mathbf{up}}$, `P) // each list element contains` $(\widehat{\mathbf{e}_\mathbf{s}},\ \widehat{\mathbf{s}_\mathbf{up}}^\intercal - V_{UP}\,\widehat{\mathbf{e}_\mathbf{s}}^\intercal)$
$queue \leftarrow \{\}$ `//` $|queue| = \mathtt{QU}$
$timer \leftarrow 0$
**foreach** $(\widehat{\mathbf{e}_\mathbf{s}},\ \mathbf{x})$ **in** *list* **do**
  **if** $\mathtt{P} = 0$ **then**
    **if** $\mathrm{HW}(\widehat{\mathbf{s}_\mathbf{down}}) = w - \mathtt{P}$ **then**
      **if** $\mathtt{L} > 0$ **then**
        **if** $\mathrm{HW}(\widehat{\mathbf{s}_\mathbf{up}}) = 0$ **then**
          $\widehat{\mathbf{e}} \leftarrow \begin{bmatrix}\mathbf{0} & \mathbf{x}\end{bmatrix}$ `//` $\mathbf{0} \in \mathbb{F}_2^{k+\mathtt{L}}$ `and` $\mathbf{x} \in \mathbb{F}_2^{n-k-\mathtt{L}}$
      **else**
        $\widehat{\mathbf{e}} \leftarrow \begin{bmatrix}\mathbf{0} & \mathbf{x}\end{bmatrix}$ `//` $\mathbf{0} \in \mathbb{F}_2^{k}$ `and` $\mathbf{x} \in \mathbb{F}_2^{n-k}$
  **if** $\mathtt{P} > 0 \wedge \mathtt{L} = 0$ **then**
    **if** $\mathrm{HW}(\mathbf{x}) = w - \mathtt{P}$ **then**
      $\widehat{\mathbf{e}} \leftarrow \begin{bmatrix}\widehat{\mathbf{e}_\mathbf{s}} & \mathbf{0} & \mathbf{x}\end{bmatrix}$ `//` $\widehat{\mathbf{e}_\mathbf{s}} \in \mathbb{F}_2^{k-\mathtt{Z}}$, $\mathbf{0} \in \mathbb{F}_2^{\mathtt{Z}}$ `and` $\mathbf{x} \in \mathbb{F}_2^{n-k}$
  **if** $\mathtt{P} > 0 \wedge \mathtt{L} > 0$ **then**
    **if** $\mathbf{x} = 0$ **then**
      $queue \leftarrow queue \cup \{\widehat{\mathbf{e}_\mathbf{s}}\}$
    **if** $timer \bmod \mathtt{QU} \cdot \mathtt{QF}$ **then**
      **foreach** $\widehat{\mathbf{e}_\mathbf{s}}$ **in** *queue* **do**
        $\widehat{\mathbf{e}'_{\mathbf{s}^*}} \leftarrow \mathbf{s}_{\widehat{\mathbf{down}}}^\intercal - V_{DOWN}\,\widehat{\mathbf{e}_\mathbf{s}}^\intercal$
        **if** $\mathrm{HW}\left(\widehat{\mathbf{e}'_{\mathbf{s}^*}}\right) = w - \mathtt{P}$ **then**
          $\widehat{\mathbf{e}} \leftarrow \left[\widehat{\mathbf{e}_\mathbf{s}} \quad \mathbf{0} \quad \widehat{\mathbf{e}'_{\mathbf{s}^*}}\right]$ `//` $\widehat{\mathbf{e}_\mathbf{s}} \in \mathbb{F}_2^{k+\mathtt{L}-\mathtt{Z}}$, $\mathbf{0} \in \mathbb{F}_2^{\mathtt{Z}}$ `and` $\widehat{\mathbf{e}'_{\mathbf{s}^*}} \in \mathbb{F}_2^{n-k-\mathtt{L}}$
      $queue \leftarrow \{\}$
    $timer \leftarrow timer + 1$
**return** $\widehat{\mathbf{e}}$

**Algorithm 2.4:** Search phase for `isd0`

`P` $> 0$, `L` $= 0$ The ISD variant implemented by `isd0` with these parameters is the Lee-Brickell [27] variant. The permuted error vector is split into three blocks

$$\widehat{\mathbf{e}} = \left[\widehat{\mathbf{e}_{\mathbf{s}}} \quad \mathbf{0} \quad \widehat{\mathbf{e}_{\mathbf{s}^*}}\right] \tag{2.12}$$

of dimension $k$+`L`−`Z`, `Z` and $n-k$−`L` respectively. All possible vectors $\widehat{\mathbf{e}_{\mathbf{s}}}$ of dimension $k$+`L`−`Z` and weight `P` indexed by the IS are enumerated, and for each of them the corresponding partial sums, which represent the vector $\widehat{\mathbf{e}_{\mathbf{s}^*}}$, are calculated using Equation 2.4. Once all the elements have been obtained, for each one it is checked that the partial sum has weight $w$−`P`; if so, a valid permuted error candidate has been found.

`P` $> 0$, `L` $> 0$ The variant implemented with these parameters is Leon's [28]. The parameter `L` represents the size of the window in the redundancy section where the error components are assumed to be zero, so that the entire $w$−`P` contribution of the error outside the IS lies in the remaining $n-k$−`L` positions. The vector is therefore divided into three blocks

$$\widehat{\mathbf{e}} = \left[\widehat{\mathbf{e}_{\mathbf{s}}} \quad \mathbf{0} \quad \widehat{\mathbf{e}'_{\mathbf{s}^*}}\right] \tag{2.13}$$

of dimension $k$+`L`−`Z`, `Z` and $n-k$−`L` respectively, and the corresponding systematic syndrome equation is shown in Equation 2.8.

The first step consists of listing all vectors $\widehat{\mathbf{e}_{\mathbf{s}}}$ of weight `P` and dimension $k$+`L`−`Z` and calculating their partial sum with respect to the upper part of $\widehat{H}$, as indicated by the first equation of the system in Equation 2.14. Once all combinations have been obtained, the vectors whose upper partial sum is equal to zero are placed in a queue of dimension `QU`. Every `QU` $\cdot$ `QF` steps, the queue is checked and emptied: for each element in this queue, it is verified whether the lower equation in the system shown in Equation 2.14 is satisfied; if so, the candidate is accepted as a valid permuted error.

$$\begin{cases} \widehat{\mathbf{s}_{\mathbf{up}}}^{\intercal} = V_{UP}\,\widehat{\mathbf{e}_{\mathbf{s}}}^{\intercal} + \mathbf{0}^{\intercal} \\ \widehat{\mathbf{s}_{\mathbf{down}}}^{\intercal} = V_{DOWN}\,\widehat{\mathbf{e}_{\mathbf{s}}}^{\intercal} + \widehat{\mathbf{e}'_{\mathbf{s}^*}}^{\intercal} \end{cases} \tag{2.14}$$

**Expected Number of Iterations** It is possible to determine the expected number of iterations required for the algorithm to find the error based on the probability of success of a single iteration. Since each iteration is independent of the previous ones, due to the random permutation of the columns carried out at the start of each iteration, the number of iterations required follows a geometric distribution, and its expected value is simply $\frac{1}{p_s}$, where $p_s$ is the probability of success of a single iteration.

An upper bound for $p_s$ is obtained by considering only the probability that, following the permutation, the $\mathtt{P}$ errors sought within the IS all fall within the $k{+}\mathtt{L}{-}\mathtt{Z}$ valid positions, and the remaining $w{-}\mathtt{P}$ errors are found in the $n{-}k{-}\mathtt{L}$ positions outside the IS. Formally:

$$p_s = \Pr[\text{success of a single iteration}] \leq \frac{\binom{k+\mathtt{L}-\mathtt{Z}}{\mathtt{P}}\binom{n-k-\mathtt{L}}{w-\mathtt{P}}}{\binom{n}{w}} \tag{2.15}$$

The numerator counts the number of favorable configurations: $\binom{k+\mathtt{L}-\mathtt{Z}}{\mathtt{P}}$ is the number of ways in which the $\mathtt{P}$ errors can be distributed across the valid positions of the IS, whilst $\binom{n-k-\mathtt{L}}{w-\mathtt{P}}$ is the number of ways in which the remaining $w{-}\mathtt{P}$ errors are distributed outside the IS. The denominator $\binom{n}{w}$ represents the total number of possible configurations for an error of weight $w$ across $n$ positions.

The expected number of iterations is at least:

$$\mathbb{E}(\text{iterations}) \geq \frac{\binom{n}{w}}{\binom{k+\mathtt{L}-\mathtt{Z}}{\mathtt{P}}\binom{n-k-\mathtt{L}}{w-\mathtt{P}}} \tag{2.16}$$

The formula represents an upper bound on $p_s$, and consequently a lower bound on the expected number of iterations, because it takes into account only the geometric condition on the error distribution, whilst ignoring all the additional constraints that, in practice, further reduce the probability of success. Among these additional constraints that are not taken into account are: the queue mechanism, which may discard valid candidates in the event of an overflow; the check on the upper part of the syndrome, which requires that $\widehat{\mathbf{s_{up}}}$ be the zero vector.

### 2.4.2. Search Phase with One Collision Level

`isd1` is the second variant of the ISD algorithms implemented in CAT [9], and uses a collision level during the search phase. The attack follows the structure shown in Section 2.2. This attack implements the attack proposed by Dumer [17] for $\mathtt{Z} = 0$ and by Stern [46] for $\mathtt{Z} = \mathtt{L}$.

Compared to `isd0`, this variant introduces an enumeration technique inspired by the *meet-in-the-middle paradigm*, which reduces the computational cost of the candidate enumeration phase. The key innovation introduced by Stern concerns the way in which the combinations of weight $\mathtt{P}$ for the error term $\widehat{\mathbf{e}_\mathbf{s}}$ are enumerated. Both Lee-Brickell [27] and Leon [28] iterated over all the combinations $\binom{k+\mathtt{L}-\mathtt{Z}}{\mathtt{P}}$ to find the error that satisfied the required constraints. Stern, on the other hand, divides the IS into two halves, each of size $\mathtt{PI} = \frac{\mathtt{P}}{2}$. Thus, instead of enumerating all $\binom{k+\mathtt{L}-\mathtt{Z}}{\mathtt{P}}$ combinations, Stern enumerates only

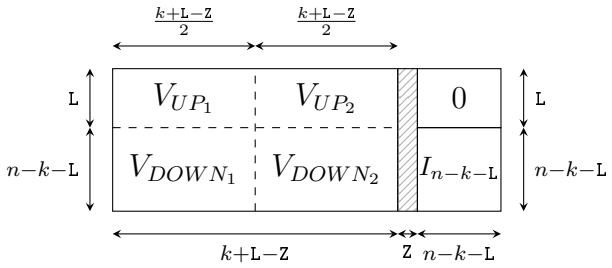


Figure 2.5: Partial systematic form of the parity-check matrix for `isd1`

$\binom{(k+L-Z)/2}{PI}$ combinations for each half. The permuted vector therefore takes the following four-block structure:

$$\widehat{\mathbf{e}} = \begin{bmatrix} \widehat{\mathbf{e}_{\mathbf{s_1}}} & \widehat{\mathbf{e}_{\mathbf{s_2}}} & \mathbf{0} & \widehat{\mathbf{e}'_{\mathbf{s}^*}} \end{bmatrix} \tag{2.17}$$

where the first two blocks cover the two halves of the IS with weight PI each, $\widehat{\mathbf{e}_{\mathbf{s_1}}}, \widehat{\mathbf{e}_{\mathbf{s_2}}} \in \mathbb{F}_2^{\frac{k+L-Z}{2}}$, the third block is constrained to be the zero vector, $\mathbf{0} \in \mathbb{F}_2^Z$, as dictated by the windowing condition introduced by Leon, and the last is the portion of redundancy excluded from the window with weight $w-P$, $\widehat{\mathbf{e}'_{\mathbf{s}^*}} \in \mathbb{F}_2^{n-k-L}$.

The parity-check matrix in partial systematic form $\widehat{H}$, obtained during the column permutation phase, on which collisions will be sought, is shown in Figure 2.5.

In mathematical terms, using the division between upper and lower parts introduced by Leon, the system can be derived from the syndrome equation in systematic form obtained by Leon (Equation 2.14), yielding:

$$\begin{cases} \widehat{\mathbf{s_{up}}}^\intercal = V_{UP_1}\, \widehat{\mathbf{e}_{\mathbf{s_1}}}^\intercal + V_{UP_2}\, \widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal + \mathbf{0}^\intercal \\ \widehat{\mathbf{s_{down}}}^\intercal = V_{DOWN}\, \widehat{\mathbf{e}_{\mathbf{s}}}^\intercal + \widehat{\mathbf{e}'_{\mathbf{s}^*}}^\intercal \end{cases} \tag{2.18}$$

The filtering method applied by Leon consisted of listing all $\binom{k+L-Z}{P}$ candidate $\widehat{\mathbf{e}_{\mathbf{s}}}$ values and checking, for each one, the first equation of the system shown in Equation 2.18. Stern, on the other hand, rather than listing all the candidates $\widehat{\mathbf{e}_{\mathbf{s}}}$ as a single vector, divides them into two halves, $\widehat{\mathbf{e}_{\mathbf{s_1}}}$ and $\widehat{\mathbf{e}_{\mathbf{s_2}}}$, constructing two separate lists. The first list contains all $\binom{(k+L-Z)/2}{PI}$ candidates of $\widehat{\mathbf{e}_{\mathbf{s_1}}}$ with the corresponding value of the partial sum $\widehat{\mathbf{s_{up}}}^\intercal - V_{UP_1}\, \widehat{\mathbf{e}_{\mathbf{s_1}}}^\intercal$, whilst the second contains all the $\binom{(k+L-Z)/2}{PI}$ candidates of $\widehat{\mathbf{e}_{\mathbf{s_2}}}$ with the corresponding partial sum $V_{UP_2}\, \widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal$. Once the two lists have been created, collisions are sought between the values of the partial sums of the lists:

$$V_{UP_1}\, \widehat{\mathbf{e}_{\mathbf{s_1}}}^\intercal = \widehat{\mathbf{s_{up}}}^\intercal - V_{UP_2}\, \widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal \tag{2.19}$$

which, rewritten, is equivalent to the first equation of the system in Equation 2.18.

In a standard RAM model, this meet-in-the-middle approach is typically optimized to minimize memory consumption: generally, only the first list containing the $\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}$ combination of $V_{UP_1}\,\widehat{\mathbf{e_{s_1}}}^{\intercal}$ is computed and stored inside an efficient lookup structure. Subsequently, the elements of the second list are generated on-the-fly; for each newly computed term $\widehat{\mathbf{s_{up}}}^{\intercal} - V_{UP_2}\,\widehat{\mathbf{e_{s_2}}}^{\intercal}$, a query is carried out in the saved lookup structure to intercept any matching entry, bypassing the need to store both lists simultaneously.

This optimization, however, relies on three primitive operations present in a RAM model, which are absent in the Boolean circuit model: the ability to *write* a value to an addressable memory location, the ability to *read it back* at a later time step, and the *time step* itself. A Boolean circuit is a direct sequence of gates with no feedback and no internal state; it computes a fixed function of its inputs in a single combinatorial pass, without any mechanism for storing intermediate results between one generated element and the next. Consequently, the entire concept of *building a list incrementally and querying it on the fly* has no combinatorial counterpart.

In this model, the only physical carrier of information is the cable. Generating a value means instantiating the combinatorial logic that calculates it and allowing the resulting signal to propagate; storing a list of $m$ elements is equivalent to physically routing $m$ independent signal buses in parallel, each wide enough to carry one element. Both lists must therefore be generated concurrently and kept active as active signals simultaneously, as no sequential phase is available.

To detect collisions among these concurrently live signals, the wires carrying the elements of both lists are fed into a single global sorting architecture, specifically, a *one-dimensional odd-even mergesort network* [24]. Once the sorting network has performed all its CAS operations, predetermined at synthesis time, any valid collision satisfying Equation 2.19 will appear in adjacent positions within the globally sorted array. A localized layer of comparative Boolean gates connected to neighboring wires is then sufficient to isolate the matching pairs. However, since multiple elements from both lists may share the same partial-sum value, giving rise to a contiguous block of equal entries in the sorted output, a single nearest-neighbor comparison is insufficient to capture all valid pairs at the boundary of such a block. Each element must therefore be compared against its `WI` closest neighbors on each side, where `WI` is chosen to be at least as large as the maximum expected multiplicity of any collision value. A localized layer of comparative Boolean gates, each connected to a window of `WI` neighboring wires, is then sufficient to isolate all matching pairs.

**Input:** $\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the permuted parity-check matrix
$\widehat{\mathbf{s}} \in \mathbb{F}_2^{n-k}$: the permuted syndrome
**Output:** $\widehat{\mathbf{e}} \in \mathbb{F}_2^n$: the permuted error
**Data:** *list*: the list containing all possible vectors $\widehat{\mathbf{e}_\mathbf{s}}$ and their partial sums
*queue*: the queue that manage the candidates that pass the preliminary filter
$timer \in \mathbb{Z}$: the counter used to trigger the periodic queue flush
$b_1, b_2 \in \{0,1\}$: flags indicating which of the two lists the element belongs to

$V_{UP_1} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L}-1\},\,\{0,\ldots,\frac{k+\mathtt{L}-\mathtt{Z}}{2}-1\})}$
$V_{UP_2} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L}-1\},\,\{\frac{k+\mathtt{L}-\mathtt{Z}}{2},\ldots,k+\mathtt{L}-\mathtt{Z}-1\})}$
$V_{DOWN} \leftarrow \widehat{H}_{(\{\mathtt{L},\ldots,n-k-1\},\,\{0,\ldots,k+\mathtt{L}-\mathtt{Z}-1\})}$
$\widehat{\mathbf{s_{up}}} \leftarrow \widehat{\mathbf{s}}_{\{0,\ldots,\mathtt{L}-1\}}$
$\widehat{\mathbf{s_{down}}} \leftarrow \widehat{\mathbf{s}}_{\{\mathtt{L},\ldots,n-k-1\}}$
$list \leftarrow \{\}$
$list \leftarrow$ `exhaustiveEnumeration(`$V_{UP_1}$, $\mathbf{0}$, `PI) // each list element contains` $(\widehat{\mathbf{e}_{\mathbf{s_1}}}, \mathbf{0}^\intercal + V_{UP_1}\,\widehat{\mathbf{e}_{\mathbf{s_1}}}, 0)$
$list \leftarrow list \cup$ `exhaustiveEnumeration(`$V_{UP_2}$, $\widehat{\mathbf{s_{up}}}$, `PI) // each list element contains` $(\widehat{\mathbf{e}_{\mathbf{s_2}}}, \widehat{\mathbf{s_{up}}}^\intercal - V_{UP_2}\,\widehat{\mathbf{e}_{\mathbf{s_2}}}, 1)$
$list \leftarrow$ `sort(`$list$`)`
$queue \leftarrow \{\}$ `//` $|queue| =$ `QU`
$timer \leftarrow 0$
**for** $i \leftarrow 0$ **to** `size(`$list$`)` **do**
  **for** $j \leftarrow 1$ **to** `WI` **do**
    $(\widehat{\mathbf{e}_{\mathbf{s_1}}}, \mathbf{x_1}, b_1) \leftarrow list[i]$
    $(\widehat{\mathbf{e}_{\mathbf{s_2}}}, \mathbf{x_2}, b_2) \leftarrow list[i+j]$
    **if** $b_1 \neq b_2$ **then**
      **if** $\mathbf{x_1} = \mathbf{x_2}$ **then**
        $\widehat{\mathbf{e}_\mathbf{s}} \leftarrow \left[\widehat{\mathbf{e}_{\mathbf{s_1}}} \quad \widehat{\mathbf{e}_{\mathbf{s_2}}}\right]$
        $queue \leftarrow queue \cup \{\widehat{\mathbf{e}_\mathbf{s}}\}$
    **if** $timer$ mod `QU` $\cdot$ `QF` **then**
      **foreach** $\widehat{\mathbf{e}_\mathbf{s}}$ **in** $queue$ **do**
        $\widehat{\mathbf{e}'_{\mathbf{s}^*}} \leftarrow \widehat{\mathbf{s_{down}}} - V_{DOWN}\,\widehat{\mathbf{e}_\mathbf{s}}^\intercal$
        **if** HW $\left(\widehat{\mathbf{e}'_{\mathbf{s}^*}}\right) = w-$`P` **then**
          $\widehat{\mathbf{e}} \leftarrow \left[\widehat{\mathbf{e}_\mathbf{s}} \quad \mathbf{0} \quad \widehat{\mathbf{e}'_{\mathbf{s}^*}}\right]$ `//` $\widehat{\mathbf{e}_\mathbf{s}} \in \mathbb{F}_2^{k+\mathtt{L}-\mathtt{Z}}$, $\mathbf{0} \in \mathbb{F}_2^{\mathtt{Z}}$ `and` $\widehat{\mathbf{e}'_{\mathbf{s}^*}} \in \mathbb{F}_2^{n-k-\mathtt{L}}$
      $queue \leftarrow \{\}$
    $timer \leftarrow timer + 1$
**return** $\widehat{\mathbf{e}}$

**Algorithm 2.5:** Search phase for `isd1`

When a collision is found, the vector $\widehat{\mathbf{e}_\mathbf{s}}$ is created by concatenating the two halves $\widehat{\mathbf{e}_{\mathbf{s_2}}}$ and $\widehat{\mathbf{e}_{\mathbf{s_2}}}$. Once the vector $\widehat{\mathbf{e}_\mathbf{s}}$ has been obtained, $\widehat{\mathbf{e}'_{\mathbf{s}^*}}$ can be calculated using the second equation of the system in Equation 2.18, and it is verified that it has a weight of exactly $w-$`P`, as illustrated in Algorithm 2.5.

**Expected Number of Iterations** It is possible to calculate the expected number of iterations required for the algorithm to find the error based on the probability of success of a single iteration. As with `isd0`, each iteration is independent of the previous ones; thanks to the random permutation of the columns carried out at the start of each iteration, the number of iterations required follows a geometric distribution, and its expected value is simply $\frac{1}{p_s}$, where $p_s$ is the probability of success of a single iteration.

In `isd1`, the valid IS of size $k+\mathtt{L}-\mathtt{Z}$ is divided into two halves. An iteration is successful when `PI` errors fall in the left half and `PI` in the right half, and the remaining $w-2\,\mathtt{PI}$ errors in the remaining $n-k-\mathtt{L}$ positions outside the IS. The upper bound for $p_s$ is therefore:

$$p_s = \Pr[\text{success of a single iteration}] \leq \frac{\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}\binom{n-k-\mathtt{L}}{w-2\,\mathtt{PI}}}{\binom{n}{w}} \tag{2.20}$$

The numerator counts the favorable configurations: $\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ and $\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ are, respectively, the number of ways in which `PI` errors are distributed across the left and right halves of the IS, whilst $binomn-k-\mathtt{L}w-2\,\mathtt{PI}$ represents the number of configurations in which the remaining $w-2\,\mathtt{PI}$ errors are distributed across positions outside the IS. The denominator $\binom{n}{w}$ represents the total number of possible configurations for an error of weight $w$ across $n$ positions.

The expected number of iterations is at least:

$$\mathbb{E}(\text{iterations}) \geq \frac{\binom{n}{w}}{\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}\binom{n-k-\mathtt{L}}{w-2\,\mathtt{PI}}} \tag{2.21}$$

The formula represents an upper bound on $p_s$, and consequently a lower bound on the expected number of iterations, because it takes into account only the geometric condition on the distribution of errors, whilst ignoring all the additional constraints that, in practice, further reduce the probability of success. Among these additional constraints that are not taken into account are: the window mechanism, which limits the number of pairs compared during the collision phase to only those elements within `WI` positions in the globally sorted list, thereby discarding potentially valid collisions; and queue management, which may discard valid candidates in the event of an overflow.

### 2.4.3. Search Phase with Two Collision Levels

`isd2` is the latest and most advanced variant of the ISD algorithms implemented in CAT [9]; it differs from previous variants in that it uses two levels of collision search during the

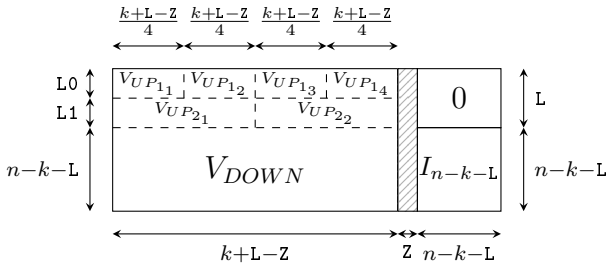


Figure 2.6: Partial systematic form of the parity-check matrix for `isd2`

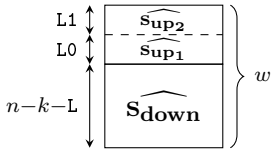


Figure 2.7: The syndrome division used in `isd2`

search phase. The algorithms included in this variant are MMT [31] (`CP` $= 0$, `PI` $= 2\,$`PIJ`) and BJMM [5] (`CP` $= 0$, `PI` $< 2\,$`PIJ`). The BJMM algorithm represents an evolution of the MMT algorithm. While maintaining the same hierarchical structure introduced by MMT, BJMM introduces the concept of *multiple representations*. Thanks to this concept, BJMM can also be interpreted as a direct generalization of MMT, of which it is the special case with `D` $= 1$.

As in the previous variants, the first step consists of selecting a valid IS and reducing the parity-check matrix to systematic form. Compared to `isd0` and `isd1`, a finer subdivision of the parameter `L` into two components, `L0` and `L1`, with `L0` $+$ `L1` $=$ `L`, which control the first and second levels of collision search, respectively, as illustrated in Figure 2.6 and Figure 2.7.

The systematic component of the error vector is divided into four blocks of equal size:

$$\widehat{\mathbf{e}} = \left[\widehat{\mathbf{e}_{\mathbf{s}_{1_1}}} \quad \widehat{\mathbf{e}_{\mathbf{s}_{1_2}}} \quad \widehat{\mathbf{e}_{\mathbf{s}_{2_1}}} \quad \widehat{\mathbf{e}_{\mathbf{s}_{2_2}}} \quad \mathbf{0} \quad \widehat{\mathbf{e}'_{\mathbf{s}^*}}\right] \tag{2.22}$$

where the first four blocks, $\widehat{\mathbf{e}_{\mathbf{s}_{1_1}}}, \widehat{\mathbf{e}_{\mathbf{s}_{1_2}}}, \widehat{\mathbf{e}_{\mathbf{s}_{2_1}}}, \widehat{\mathbf{e}_{\mathbf{s}_{2_2}}} \in \mathbb{F}_2^{\frac{k+L-z}{4}}$, are weighted `PI` in pairs and represent the candidates for the four subproblem of the first hierarchical level, the penultimate block is constrained to be the zero vector, $\mathbf{0} \in \mathbb{F}_2^z$, and the last block, $\widehat{\mathbf{e}'_{\mathbf{s}^*}} \in \mathbb{F}_2^{n-k-L}$ represents the redundancy portion excluded from the window.

In mathematical terms, the division of the matrix and the syndrome shown in Figures

Figure 2.6 and Figure 2.7 translates into the following system of equations:

$$\begin{cases} \widehat{\mathbf{s_{up_1}}}^{\mathsf{T}} = V_{UP_{1_1}}\,\widehat{\mathbf{e_{s_{1_1}}}}^{\mathsf{T}} + V_{UP_{1_2}}\,\widehat{\mathbf{e_{s_{1_2}}}}^{\mathsf{T}} + V_{UP_{2_1}}\,\widehat{\mathbf{e_{s_{2_1}}}}^{\mathsf{T}} + V_{UP_{2_2}}\,\widehat{\mathbf{e_{s_{2_2}}}}^{\mathsf{T}} + \mathbf{0} \\ \widehat{\mathbf{s_{up_2}}}^{\mathsf{T}} = V_{UP_1}\,\widehat{\mathbf{e_{s_1}}}^{\mathsf{T}} + V_{UP_2}\,\widehat{\mathbf{e_{s_2}}}^{\mathsf{T}} + \mathbf{0} \\ \widehat{\mathbf{s_{down}}}^{\mathsf{T}} = V_{DOWN}\,\widehat{\mathbf{e_s}}^{\mathsf{T}} + \widehat{\mathbf{e'_{s^*}}}^{\mathsf{T}} \end{cases} \tag{2.23}$$

The first equation involves all four blocks and is satisfied by the first level of collision; the second concerns the aggregate vectors $\widehat{\mathbf{e_{s_1}}}$ and $\widehat{\mathbf{e_{s_2}}}$ produced by the first level; finally, the third determines the residual vector $\widehat{\mathbf{e'_{s^*}}}$.

The key mechanism of `isd2` lies in the way Hamming weights are distributed across the two hierarchical levels. In MMT, this decomposition is rigid: the vectors in the four lists of the first level all have weight $\mathtt{PIJ} = \frac{\mathtt{P}}{4}$, so that two colliding vectors necessarily produce a vector of weight $\mathtt{PI} = \frac{\mathtt{P}}{2}$ at the second level, and two colliding vectors of weight $\mathtt{PI}$ produce the solution vector of weight $\mathtt{P}$. This rigidity constrains the number of pairs with which each solution can be represented, limiting the probability of success per iteration.

BJMM relaxes this constraint by introducing the concept of *multiple representations*. The idea is to allow first-level vectors to have a slightly higher weight: $\mathtt{PIJ} = \frac{\mathtt{P}}{4} + \Delta$, where $\Delta > 0$. Two vectors of weight $\mathtt{PIJ}$ may still collide to produce a vector of weight $\mathtt{PI}$, but only on condition that the excess $\Delta$ bits occupy the same positions in both vectors, cancelling each other out in the sum in $\mathbb{F}_2$. The number of distinct pairs satisfying this condition for a given solution of weight $\mathtt{PI}$ is called the *number of representations*:

$$R = \binom{\mathtt{PI}}{\mathtt{P}/4}\binom{k/2 - \mathtt{PI}}{\Delta} \tag{2.24}$$

The first factor represents the number of ways in which the $\mathtt{PI}$ positions can be distributed in groups of $\frac{\mathtt{P}}{4}$ between the two vectors; the second represents the number of ways in which the $\Delta$ excess bits can be placed among the remaining $\frac{k}{2} - \mathtt{PI}$ positions so that they cancel each other out. As $\Delta$ increases, the number of representations $R$ grows, increasing the probability that at least one valid representation will be enumerated per iteration.

In CAT, this concept is reflected in the parameter `D`, which controls how many distinct configurations of the lists are explored before moving on to the next iteration. Rather than regenerating the lists entirely, from the second exploration onwards a single bit flip is applied to the partial sums of the second list, yielding a new configuration of the subwindow of width `L0` at incremental cost. Thus, `D` is the operational implementation of $\Delta$ from the theory of the BJMM algorithm: in theory, $\Delta$ structurally determines the number of representations that exist for each solution; in the implementation of CAT, `D`

determines how many of these representations are actively explored. In the case $\texttt{D} = 1$, the behavior reduces to that of MMT.

The search phase for algorithms that utilize two levels of collision search is divided into two levels. In the first level, four base lists $\theta_{1_1}$, $\theta_{1_2}$, $\theta_{2_1}$, $\theta_{2_2}$ are generated, each containing all $\binom{\frac{k+\texttt{L}-\texttt{z}}{4}}{\texttt{PIJ}}$ candidate vectors of weight PIJ for the respective block. Subsequently, collisions are sought between $\theta_{1_1}$ and $\theta_{1_2}$ that satisfy the first equation of the system Equation 2.23 restricted to the first subwindow of amplitude L0, producing the intermediate list $\theta_1$ whose elements have weight PI; Similarly, collisions between $\theta_{2_1}$ and $\theta_{2_2}$ produce the list $\theta_2$. In the second level, collisions between $\theta_1$ and $\theta_2$ that satisfy the second equation of the system, restricted to the second subwindow of width L1, are sought. Each valid collision produces the complete vector $\widehat{\mathbf{e}_\mathbf{s}}$ with weight P. Finally, $\widehat{\mathbf{e}'_{\mathbf{s}^*}}$ is calculated using the third equation of the system and its weight is verified, which must be exactly $w - \texttt{P}$, as illustrated in Algorithm 2.6.

In the CAT implementation, the search phase differs slightly from the theoretical description due to an algebraic optimization that reduces the number of base lists from four to three. Since the four submarines $V_{UP_{1_1}}, V_{UP_{1_2}}, V_{UP_{2_1}}, V_{UP_{2_2}}$ all have the same dimension and the candidates all have the same weight PIJ, the lists $\theta_{1_1}$ and $\theta_{2_1}$ are identical in terms of their set of candidates, and similarly $\theta_{1_2}$ and $\theta_{2_2}$. It therefore makes no sense to generate them twice: only two lists are produced by exhaustive enumeration, L0_sum and L1_sum[0], corresponding respectively to $V_{UP_1}\,\widehat{\mathbf{e}_{\mathbf{s_1}}}^\intercal$ and $V_{UP_2}\,\widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal$. A third list, L1_sum[1], is obtained at no cost by performing an element-by-element XOR of L1_sum[0] with $\widehat{\mathbf{s}_{\mathbf{up}}}^\intercal$, thus containing $\widehat{\mathbf{s}_{\mathbf{up}}}^\intercal + V_{UP_2}\,\widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal$, without any additional enumeration.

The first level is therefore carried out in two distinct steps, indexed by the parameter $\texttt{t} \in \{0, 1\}$. For $\texttt{t} = 0$, the combined list contains the elements of L0_sum and L1_sum[0], and collisions are sought, considering only the first L0 positions, which satisfy:

$$V_{UP_1}\widehat{\mathbf{e}_{\mathbf{s_1}}}^\intercal = V_{UP_2}\widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal \tag{2.25}$$

For $\texttt{t} = 1$, the combined list uses L0_sum and L1_sum[1] instead, and collisions are sought by considering only the first L0 positions that satisfy:

$$V_{UP_1}\widehat{\mathbf{e}_{\mathbf{s_1}}}^\intercal = \widehat{\mathbf{s}_{\mathbf{up}}}^\intercal + V_{UP_2}\widehat{\mathbf{e}_{\mathbf{s_2}}}^\intercal \tag{2.26}$$

Together, these two steps cover exactly the same condition as the first level of the theoretical description, with a saving in the generation of the base lists since L1_sum[1] does not require any additional enumeration.

**Input:** $\widehat{H} \in \mathbb{F}_2^{(n-k)\times n}$: the permuted parity-check matrix
$\widehat{\mathbf{s}} \in \mathbb{F}_2^{n-k}$: the permuted syndrome
**Output:** $\widehat{\mathbf{e}} \in \mathbb{F}_2^n$: the permuted error
**Data:** $list_1$, $list_2$, $list$: the lists containing all possible vectors and their partial sums
$queue$: the queue that manage the candidates that pass the preliminary filter
$timer \in \mathbb{Z}$: the counter used to trigger the periodic queue flush
$b_1, b_2 \in \{0,1\}$: flags indicating which of the two lists the element belongs to

1 $\mathtt{L} \leftarrow \mathtt{L0} + \mathtt{L1}$
2 $V_{UP_{1_1}} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L0}-1\},\{0,\ldots,(k+\mathtt{L}-\mathtt{z})/4-1\})}$ $V_{UP_{1_2}} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L0}-1\},\{(k+\mathtt{L}-\mathtt{z})/4,\ldots,(k+\mathtt{L}-\mathtt{z})/2-1\})}$
$V_{UP_{2_1}} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L0}-1\},\{(k+\mathtt{L}-\mathtt{z})/2,\ldots,3(k+\mathtt{L}-\mathtt{z})/4-1\})}$ $V_{UP_{2_2}} \leftarrow \widehat{H}_{(\{0,\ldots,\mathtt{L0}-1\},\{3(k+\mathtt{L}-\mathtt{z})/4,\ldots,k+\mathtt{L}-\mathtt{z}-1\})}$
3 $V_{UP_1} \leftarrow \widehat{H}_{(\{\mathtt{L0},\ldots,\mathtt{L1}-1\},\{0,\ldots,(k+\mathtt{L}-\mathtt{z})/2-1\})}$ $V_{UP_2} \leftarrow \widehat{H}_{(\{\mathtt{L0},\ldots,\mathtt{L1}-1\},\{(k+\mathtt{L}-\mathtt{z})/2,\ldots,k+\mathtt{L}-\mathtt{z}-1\})}$
4 $V_{DOWN} \leftarrow \widehat{H}_{(\{\mathtt{L1},\ldots,n-k-1\},\{0,\ldots,k+\mathtt{L}-\mathtt{z}\})}$
5 $\widehat{\mathbf{s}_{\mathbf{up_1}}} \leftarrow \widehat{\mathbf{s}}_{\{0,\ldots,\mathtt{L0}-1\}}$ $\widehat{\mathbf{s}_{\mathbf{up_2}}} \leftarrow \widehat{\mathbf{s}}_{\{\mathtt{L0},\ldots,\mathtt{L1}-1\}}$
6 $\widehat{\mathbf{s}_{\mathbf{down}}} \leftarrow \widehat{\mathbf{s}}_{\{\mathtt{L1},\ldots,n-k-1\}}$
7 $list_1 \leftarrow \{\}$ $list_2 \leftarrow \{\}$ $list \leftarrow \{\}$
8 $list_1 \leftarrow$ exhaustiveEnumeration$(V_{UP_{1_1}}, \mathbf{0}, \mathtt{PIJ})$ // each list element contains $(\widehat{\mathbf{e}_{\mathbf{s1_1}}}, \mathbf{0}^\intercal + V_{UP_{1_1}}\,\widehat{\mathbf{e}_{\mathbf{s1_1}}}, 0)$
9 $list_1 \leftarrow$ exhaustiveEnumeration$(V_{UP_{1_2}}, \mathbf{0}, \mathtt{PIJ})$ // each list element contains $(\widehat{\mathbf{e}_{\mathbf{s1_2}}}, \mathbf{0}^\intercal + V_{UP_{1_2}}\,\widehat{\mathbf{e}_{\mathbf{s1_2}}}, 1)$
10 $list_2 \leftarrow$ exhaustiveEnumeration$(V_{UP_{2_1}}, \mathbf{0}, \mathtt{PIJ})$ // each list element contains $(\widehat{\mathbf{e}_{\mathbf{s2_1}}}, \mathbf{0}^\intercal + V_{UP_{2_1}}\,\widehat{\mathbf{e}_{\mathbf{s2_1}}}, 0)$
11 $list_2 \leftarrow$ exhaustiveEnumeration$(V_{UP_{2_2}}, \widehat{\mathbf{s}_{\mathbf{up_1}}}, \mathtt{PIJ})$ // each list element contains $(\widehat{\mathbf{e}_{\mathbf{s2_2}}}, \widehat{\mathbf{s}_{\mathbf{up_1}}}^\intercal + V_{UP_{2_2}}\,\widehat{\mathbf{e}_{\mathbf{s2_2}}}, 1)$
12 $list_1 \leftarrow$ sort$(list_1)$
13 $list_2 \leftarrow$ sort$(list_2)$
14 $queue \leftarrow \{\}$ // $|queue| = \mathtt{QU0}$
15 $timer \leftarrow 0$
16 **for** $i \leftarrow 0$ **to** size$(list_1)$ **do**
17 **for** $j \leftarrow 1$ **to** $\mathtt{WI1}$ **do**
18 $(\widehat{\mathbf{e}_{\mathbf{s1_1}}}, \mathbf{x_1}, b_1) \leftarrow list_1[i]$
19 $(\widehat{\mathbf{e}_{\mathbf{s1_2}}}, \mathbf{x_2}, b_2) \leftarrow list_1[i+j]$
20 **if** $b_1 \neq b_2$ **then**
21 **if** $\mathbf{x_1} = \mathbf{x_2}$ **then**
22 $\widehat{\mathbf{e}_{\mathbf{s1}}} \leftarrow \left[\widehat{\mathbf{e}_{\mathbf{s1_1}}} \quad \widehat{\mathbf{e}_{\mathbf{s1_2}}}\right]$
23 $\mathbf{x} \leftarrow V_{UP_1}\,\widehat{\mathbf{e}_{\mathbf{s1}}}^\intercal$
24 $queue \leftarrow queue \cup \left(\widehat{\mathbf{e}_{\mathbf{s1}}},\ \mathbf{x}\right)$
25 **if** $timer \bmod \mathtt{QU0} \cdot \mathtt{QU0}$ **then**
26 **foreach** $\left(\widehat{\mathbf{e}_{\mathbf{s1}}},\ \mathbf{x}\right)$ **in** $queue$ **do**
27 $list \leftarrow list \cup (\widehat{\mathbf{e}_{\mathbf{s1}}}, \mathbf{x}, 0)$
28 $queue \leftarrow \{\}$
29 $timer \leftarrow timer + 1$
30 $queue \leftarrow \{\}$ // $|queue| = \mathtt{QU0}$
31 $timer \leftarrow 0$
32 **for** $i \leftarrow 0$ **to** size$(list_2)$ **do**
33 **for** $j \leftarrow 1$ **to** $\mathtt{WI1}$ **do**
34 $(\widehat{\mathbf{e}_{\mathbf{s2_1}}}, \mathbf{x_1}, b_1) \leftarrow list_2[i]$
35 $(\widehat{\mathbf{e}_{\mathbf{s2_2}}}, \mathbf{x_2}, b_2) \leftarrow list_2[i+j]$
36 **if** $b_1 \neq b_2$ **then**
37 **if** $\mathbf{x_1} = \mathbf{x_2}$ **then**
38 $\widehat{\mathbf{e}_{\mathbf{s2}}} \leftarrow \left[\widehat{\mathbf{e}_{\mathbf{s2_1}}} \quad \widehat{\mathbf{e}_{\mathbf{s2_2}}}\right]$
39 $\mathbf{x} \leftarrow \widehat{\mathbf{s}_{\mathbf{up_2}}}^\intercal - V_{\widehat{UP_2}}\,\widehat{\mathbf{e}_{\mathbf{s2}}}^\intercal$
40 $queue \leftarrow queue \cup \left(\widehat{\mathbf{e}_{\mathbf{s2}}},\ \mathbf{x}\right)$
41 **if** $timer \bmod \mathtt{QU0} \cdot \mathtt{QU0}$ **then**
42 **foreach** $\left(\widehat{\mathbf{e}_{\mathbf{s2}}},\ \mathbf{x}\right)$ **in** $queue$ **do**
43 $list \leftarrow list \cup (\widehat{\mathbf{e}_{\mathbf{s2}}}, \mathbf{x}, 1)$
44 $queue \leftarrow \{\}$
45 $timer \leftarrow timer + 1$
46 $list \leftarrow$ sort$(list)$
47 $queue \leftarrow \{\}$ // $|queue| = \mathtt{QU1}$
48 $timer \leftarrow 0$
49 **for** $i \leftarrow 0$ **to** size$(list)$ **do**
50 **for** $j \leftarrow 0$ **to** $\mathtt{WI1}$ **do**
51 $(\widehat{\mathbf{e}_{\mathbf{s1}}}, \mathbf{x_1}, b_1) \leftarrow list[i]$
52 $(\widehat{\mathbf{e}_{\mathbf{s2}}}, \mathbf{x_2}, b_2) \leftarrow list[j]$
53 **if** $b_1 \neq b_2$ **then**
54 **if** $\mathbf{x_1} = \mathbf{x_2}$ **then**
55 $\widehat{\mathbf{e}_{\mathbf{s}}} \leftarrow \left[\widehat{\mathbf{e}_{\mathbf{s1}}} \quad \widehat{\mathbf{e}_{\mathbf{s2}}}\right]$
56 $queue \leftarrow queue \cup \{\widehat{\mathbf{e}_{\mathbf{s}}}\}$
57 **if** $timer \bmod \mathtt{QU1} \cdot \mathtt{QF1}$ **then**
58 **foreach** $\widehat{\mathbf{e}_{\mathbf{s}}}$ **in** $queue$ **do**
59 $\widehat{\mathbf{e}'_{\mathbf{s}*}} \leftarrow \widehat{\mathbf{s}_{\mathbf{down}}} - V_{DOWN}\,\widehat{\mathbf{e}_{\mathbf{s}}}^\intercal$
60 **if** $\mathrm{HW}(\widehat{\mathbf{e}_{\mathbf{s}}}) + \mathrm{HW}\left(\widehat{\mathbf{e}'_{\mathbf{s}*}}\right) = w$ **then**
61 $\widehat{\mathbf{e}} \leftarrow \left[\widehat{\mathbf{e}_{\mathbf{s}}} \quad \mathbf{0} \quad \widehat{\mathbf{e}'_{\mathbf{s}*}}\right]$ // $\widehat{\mathbf{e}_{\mathbf{s}}} \in \mathbb{F}_2^{k+\mathtt{L}-\mathtt{z}}$, $\mathbf{0} \in \mathbb{F}_2^{\mathtt{z}}$ and $\widehat{\mathbf{e}'_{\mathbf{s}*}} \in \mathbb{F}_2^{n-k-\mathtt{L}}$
62 $queue \leftarrow \{\}$
63 $timer \leftarrow timer + 1$
64 **return** $\widehat{\mathbf{e}}$

**Algorithm 2.6:** Search phase for `isd2`

The parameter `D` controls the number of times the entire cycle over the two steps `t` is repeated. In the first iteration ($d = 0$), the lists `L0_sum`, `L1_sum[0]` and `L1_sum[1]` are generated as described; in subsequent iterations ($d > 0$), rather than regenerating everything, a single bit flip is applied via Gray coding to the elements of `L1_sum[0]` and `L1_sum[1]` simultaneously, exploring a new configuration of the `L0` subwindow at incremental cost. `D` is therefore the operational implementation of the $\Delta$ in the BJMM theory: it determines how many distinct representations of the same solution are actively explored before moving on to the next iteration of the external algorithm.

In both steps of the first level, the two lists under consideration are concatenated and sorted, as with `isd1`, using the one-dimensional odd-even mergesort sorting network [24] with respect to the first `L0` bits. Valid collisions are identified by comparing neighboring elements in the sorted global list using a window of width `WI0`; for each collision, the remaining partial-sum over the remaining `L1` bits is propagated as an element of the second-level intermediate list.

In the second level, the intermediate results of the `2D` steps of the first level are collected in the list `L_root`, which is sorted via a sorting network with respect to the total partial sum of `L` bits. Valid collisions are identified using a window of width `WI1`, with the option to verify the weight `PI` of the intermediate vectors via the flag `CP`. Each valid collision at the second level produces the complete vector $\widehat{\mathbf{e}_{\mathbf{s}}}$, from which $\widehat{\mathbf{e}'_{\mathbf{s}^*}}$ is calculated using the third equation of the system Equation 2.23, with its weight verified in the manner controlled by the `CS` flag. If `CS` is active, a check is performed to ensure that the weight of the entire error vector is $w$; otherwise, a check is performed to ensure that the weight of $\widehat{\mathbf{e}'_{\mathbf{s}^*}}$ is $w-$`P`.

**Expected Number of Iterations** It is possible to calculate the expected number of iterations required for the algorithm to find the error based on the probability of success of a single iteration. As with the previous variants, each iteration is independent of the previous ones; thanks to the random permutation of the columns performed at the beginning of each iteration, the number of iterations required follows a geometric distribution, and its expected value is simply $\frac{1}{p_s}$, where $p_s$ is the probability of success of a single iteration.

As seen earlier, `isd2` introduces two levels of collision, each with its own window. The IS of size $k+$`L`$-$`Z` is divided into two halves. Unlike `isd1`, in `isd2` each error of weight `PI` is in turn divided into two halves. The first level finds pairs that collide on the window of size `L0`, while the second level finds pairs that collide on the remaining `L1` bits. An iteration is successful when `PI` errors fall in the left half and the same number in the right

half, and the remaining errors are found in positions outside the IS. The upper bound for $p_s$ is formally identical to that of `isd1`:

$$p_s = \Pr[\text{success of a single iteration}] \leq \frac{\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}\binom{n-k-\mathtt{L}}{w-2\,\mathtt{PI}}}{\binom{n}{w}} \tag{2.27}$$

as well as the expected number of iterations:

$$\mathbb{E}(\text{iterations}) \geq \frac{\binom{n}{w}}{\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}\binom{n-k-\mathtt{L}}{w-2\,\mathtt{PI}}} \tag{2.28}$$

Although the upper bound formula coincides with that of `isd1`, the internal structure of `isd2` is more complex; in fact, each of the `PI` errors in each half is computed as the XOR of two weight vectors `PIJ`. This means that the true probability $p_s$ suffers additional losses compared to `isd1`, due to three factors that the formula neglects: multiple representations (only `D` possible difference values $\Delta$ out of $2^{\mathtt{LO}}$ are considered in the first merge, reducing the number of representations searched); the two windows `WI0` and `WI1`, one for each merge level, which discard valid pairs outside the window; and the two queues, which may lose candidates due to overflow at both levels.

It follows that `isd2` generally requires a greater number of iterations than `isd1`. However, this disadvantage is offset by the fact that the cost of a single iteration is significantly lower thanks to the two levels of collision search employed. While `isd1` enumerates two lists of size $\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}$, `isd2` enumerates four lists of size $\binom{(k+\mathtt{L}-\mathtt{z})/4}{\mathtt{PIJ}}$, which are much smaller since both $\mathtt{PIJ} < \mathtt{PI}$ and the portion of IS considered is halved. The total cost of the attack, given by the product of the number of iterations and the cost per iteration, is therefore generally lower than that of `isd1`.

## 2.5. Post Processing Phase

Finally, at the end of all `I` iterations, during the post processing phase, the permuted vector, $\widehat{\mathbf{e}}$, found during the search phase is transformed into the error vector $\mathbf{e}$ that satisfies the original system of equations.

This phase consists of two sequential steps. The first step involves assembling the vector $\widehat{\mathbf{e}}$ from the results produced during the search phase: the error component $\widehat{\mathbf{e}}_{s^*}$, indexed by the RS, and the error component $\widehat{\mathbf{e}}_s$, indexed by the IS, are placed into their corresponding positions within $\widehat{\mathbf{e}}$.

The second step applies the inverse permutation to map $\widehat{\mathbf{e}}$ back to the original coordinate

system, yielding the final error vector:

$$\mathbf{e} = \widehat{\mathbf{e}} P^{-1}. \tag{2.29}$$

The resulting vector $\mathbf{e}$ satisfies the original syndrome equation $H\,\mathbf{e}^{\intercal} = \mathbf{s}^{\intercal}$ and constitutes the solution to the SDP instance under attack.

Table 2.1: Attack parameters used by CAT

| Parameter | Description | Value | Attack |
|---|---|---|---|
| I | Number of iterations | $\mathtt{I} > 0$ | isd0, isd1, isd2 |
| RE | Chain length | $\mathtt{RE} \leq \mathtt{I}$ | isd0, isd1, isd2 |
| X | Number of columns swapped per random walk step | $1 \leq \mathtt{X} \leq n-k-\mathtt{L}$ | isd0, isd1, isd2 |
| YX | Number of additional candidate columns for random walk | $0 \leq \mathtt{YX} \leq k-\mathtt{L}-\mathtt{X}$ | isd0, isd1, isd2 |
| Z | Number of columns excluded from the search phase | $0 \leq \mathtt{Z} \leq k+\mathtt{L}-\mathtt{P}$ | isd0, isd1, isd2 |
| FW | If active, encodes the known error weight as an additional equation in the system, reducing the dimension $k$ by one | $\mathtt{FW} \in \{0,1\}$ | isd0, isd1, isd2 |
| L | Number of additional columns in the dense part | $0 \leq \mathtt{L} \leq n-k-w+\mathtt{P}$ | isd0, isd1 |
| QU | Queue size for candidate storage in the search phase | $\mathtt{QU} > 0$ | isd0, isd1 |
| QF | Multiplier for the queue processing frequency | $\mathtt{QF} \geq \frac{1}{\mathtt{QU}}$ | isd0, isd1 |
| PI | Hamming weight of each half of the error vector in the dense part | $0 \leq \mathtt{PI} \leq \frac{w}{2}$ | isd1, isd2 |
| P | Hamming weight of the error vector in the dense part | $0 \leq \mathtt{P} \leq w$ | isd0 |
| WI | Window size for collision search | $\mathtt{WI} > 0$ | isd1 |
| PIJ | Hamming weight of each quarter of the error vector in the dense part | $1 \leq \mathtt{PIJ} \leq \frac{w}{4}$, $\mathtt{PIJ} \geq \frac{\mathtt{PI}}{2}$ | isd2 |
| L0 | Number of rows of the upper block | $\mathtt{L0} > 0$, $\mathtt{L0}+\mathtt{L1} \leq n-k-w$ | isd2 |
| L1 | Number of rows of the middle block | $\mathtt{L1} > 0$, $\mathtt{L0}+\mathtt{L1} \leq n-k-w$ | isd2 |
| CP | If active, checks that the partial weights of the left and right halves of the error are both equal to $p$ | $\mathtt{CP} \in \{0,1\}$ | isd2 |
| CS | If active, checks that the total weight of the found error vector equals $w$ | $\mathtt{CS} \in \{0,1\}$ | isd2 |
| D | Controls the randomization depth of the collision search tree | $1 \leq \mathtt{D} \leq 2^{\mathtt{L0}}$ | isd2 |
| QU0 | Queue size for candidate storage at the first collision level | $\mathtt{QU0} > 0$ | isd2 |
| QF0 | Multiplier for the queue processing frequency at the first collision level | $\mathtt{QF0} > 0$ | isd2 |
| WI0 | Window size for collision search at the first level | $\mathtt{WI0} > 0$ | isd2 |
| QU1 | Queue size for candidate storage at the second collision level | $\mathtt{QU1} > 0$ | isd2 |
| QF1 | Multiplier for the queue processing frequency at the second collision level | $\mathtt{QF1} > 0$ | isd2 |
| WI1 | Window size for collision search at the first level | $\mathtt{WI1} > 0$ | isd2 |

Table 2.3: Algorithmic configuration of ISD variants within the CAT framework based on structural parameters

| **ISD variant** | `P` | `L` | `Z` | `PI` | `PIJ` | `CP` | `D` |
|---|---|---|---|---|---|---|---|
| Prange | 0 | 0 | 0 | — | — | — | — |
| Ball-Collision | 0 | $> 0$ | $\geq 0$ | — | — | — | — |
| Lee-Brickell | $> 0$ | 0 | 0 | — | — | — | — |
| Leon | $> 0$ | $> 0$ | 0 | — | — | — | — |
| Dumer | — | $\geq 0$ | 0 | $> 0$ | — | — | — |
| Stern | — | $\geq 0$ | `L` | $> 0$ | — | — | — |
| MMT | — | $\geq 0$ | $\geq 0$ | `2PIJ` | $> 0$ | 0 | $= 1$ |
| BJMM | — | $\geq 0$ | $\geq 0$ | $<$ `2PIJ` | $> 0$ | 0 | $> 1$ |

# 3 | CryptAttackTester Modification with Parallel Computation

This chapter presents the main contribution of the thesis: the parallelization of the ISD algorithms within the CAT framework [9]. We begin by analyzing the opportunities for parallelization within the ISD algorithms and the parallelization strategy adopted, following the approach proposed by Bernstein in [8]. Finally, the design and implementation of the parallel attack `isd1_parallel`, the parallel version of the `isd1` algorithm developed within CAT, will be described, detailing the stages of the algorithm and their implementation in the parallel context.

## 3.1. Parallelism in Information Set Decoding algorithms

Bernstein's paper [8] offers a systematic critique of a widespread methodological error in cryptanalysis: using only serial computational models as the sole reference for assessing the complexity and cost of an attack, and consequently for determining key sizes, whilst ignoring attacks that exploit parallelism. Although Bernstein formulates his criticism in the context of exhaustive AES [36] key search, the methodological principle he sets out is general: a suitably designed parallel machine can be superior to a serial one both in terms of speed and in terms of price-performance ratio.

This methodological error does not apply solely to symmetric cryptography and exhaustive key search, but recurs in a similar manner in the context of ISD algorithms. The latter are traditionally analyses and compared in terms of serial complexity, whereas in CAT [9] this is measured in terms of bit-operations. However, by considering only the serial case, the parallel version of ISD algorithms is not taken into account, which could offer better performance in terms of the time required to find a valid solution. It follows that

comparisons between serial ISD variants, and more generally the security estimates of code-based cryptosystem whose analysis relies on the optimization of such algorithms, may suffer from the same distortion that Bernstein highlighted in [8].

In this section, we will analyze how to apply parallelization to ISD attacks implemented in CAT, examining which stages lend themselves best to effective parallelization, how to apply it, and what implications this has for the assessment of the algorithms' complexity.

### 3.1.1. Parallelization in CryptAttackTester

As seen in the Chapter 2, every ISD attack implemented in CAT [9] follows three main phases: the column permutation phase, the search phase and the post processing phase. An analysis of the dependencies between the operations that make up each phase makes it possible to identify which of them lend themselves to effective parallelization.

For the parallelization of ISD attacks, the model proposed by Bernstein in [8] will be followed: rather than performing the entire computation on a single computing machine operating sequentially, the work will be distributed across `PF` separate machines operating concurrently. The concurrent machines are arranged on a two-dimensional mesh of size $\sqrt{\texttt{PF}} \times \sqrt{\texttt{PF}}$, where each machine is connected to four neighboring machines (to the north, south, west and east) within the mesh.

In particular, as will be explained in detail in the following sections, the collision search phase will be accelerated by combining the parallel Bernstein mesh model from [8] with a two-dimensional sorting network. Instead of relying on a one-dimensional sorting network as in the serial version, the LS3 sorting network [26] is employed: designed natively for two-dimensional grids of concurrent machines, it allows the data within the mesh to be sorted in parallel efficiently.

Each concurrent machine performs operations on an independent portion of the data; the data structures involved will be replicated for each machine to avoid read and write conflicts, and the results produced by each computing machine will be aggregated before proceeding to the subsequent sequential operations. The degree of parallelism, i.e. the number of concurrent machines, is determined by the parameter `PF`, introduced as an additional parameter to those listed in Table 2.1.

In the following, the analysis will focus on how parallelization can be applied to the `isd1` attack; the `isd2` attack will not be examined separately, as it extends the same underlying concept of `isd1` by applying two nested levels of collision search, and its parallelization follows directly from the same principles.

The column permutation phase, whether performed via partial Gaussian reduction at the start of a chain or via a random walk within a chain, cannot be parallelized in either case, due to the cascading dependencies inherent to both methods. Similarly, the post processing phase operates on a single resulting vector through sequentially dependent steps whose granularity is too basic to benefit from parallelization. The search phase, by contrast, offers the greatest opportunities for parallelization, as detailed below.

**Search phase** The search phase offers the greatest opportunities for parallelization. It consists of three distinct sub-phases: subset enumeration, sorting and collision search.

The first sub-phase is subset enumeration. In this step, two distinct lists are constructed containing all combinations of vectors of dimension $\frac{(k+\mathtt{L}-\mathtt{Z})}{2}$ and weight `PI`, along with their corresponding partial sums. In the serial version, the two lists are constructed sequentially, one after the other; as they are independent, this step is *embarrassingly parallel*. In the parallel version, the two enumerations are performed simultaneously; each is distributed across $\frac{\mathtt{PF}}{2}$ concurrent machines, each of which calculates a portion of the total enumeration, so that all machines work simultaneously on portions of similar size. Once all machines have completed the computation of their respective portions of the enumeration, the results are aggregated into a single list which will be sorted in the next sub-phase.

The next sub-phase involves sorting the combinations and their corresponding partial sums. In the serial version, the sorting algorithm used is odd-even mergesort. This choice is dictated by the fact that odd-even mergesort is a sorting network; a standard sorting algorithm would require conditionally skipping instructions based on the values in the list, whereas sorting networks perform the sort by always carrying out all operations regardless of the data, making them compatible with the Boolean circuit model adopted by CAT. In the parallel version, this step is parallelized by replacing the one-dimensional sorting network, odd-even mergesort, with a two-dimensional sorting network, the *LS3 sorting network* [26], designed to intrinsically exploit the parallelism of the mesh of concurrent machines. Further details on sorting networks and the choice of LS3 will be provided in the following sections.

Finally, the last sub-phase is the collision search among the elements of the sorted list. In the serial version, each element is compared with the `WI` subsequent elements. In the parallel version, however, the elements are divided into contiguous batches and assigned to the `PF` concurrent machines, each of which performs the same process on its own portion. The management of boundaries between adjacent portions is achieved via global indices; when a comparison involves an element belonging to the adjacent machine, this

is retrieved by calculating the machine index and the corresponding local index. At the end of this step, all results produced by each concurrent machine are aggregated into a single structure before proceeding further.

### 3.1.2. Circuit Depth as the Primary Objective

The cost metric used in CAT [9] is the total number of logic gates used in the circuit, in order to capture the computational complexity of an attack. This metric, however, does not directly quantify the benefits that a parallel architecture can offer.

In a serial execution, the total number of gates equals the execution time, since each gate must be evaluated one after the other. However, when the computation is distributed across multiple concurrent machines, multiple gates can be evaluated simultaneously; therefore, what limits the execution time is no longer the total number of gates, but rather the length of the critical path, that is, the minimum number of gates that must be executed strictly in sequence: the depth of the circuit.

The *depth*, of a circuit is defined as *the length of the critical path*, that is, the minimum number of gates that no parallelization can eliminate, since each gate on the path depends on the output of the preceding ones. All operations not on this critical path can be executed concurrently and therefore do not directly contribute to the execution time of a parallel implementation. This distinction makes circuit depth the fundamental metric for evaluating parallel attack algorithms. Two circuits with the same total number of gates may behave differently as the parallelization factor varies: one might have a smaller depth, allowing much of the computation to proceed concurrently; while the other might concentrate its work along a longer critical path, thus leaving little room for improvement.

In the context of this work, the central question is not merely to determine how many gates are required to implement the parallel version of the attack, but specifically whether the depth is reduced compared to the serial counterpart. A reduction in depth directly corresponds to a reduction in the actual time required to find a valid error vector, representing a concrete advantage that an attacker with parallel resources can exploit. Furthermore, even if parallelization were to introduce an overhead, thereby increasing the number of gates required to implement the attack, this additional cost would be justified only if offset by a reduction in depth.

## 3.2. Parallel Version of isd1

This section provides a detailed description of the parallelization scheme proposed in [8] as applied to `isd1`, explaining the differences from the serial version and the implementation details of `isd1_parallel`.

`isd1_parallel` is an ISD attack implemented for integration into the CAT [9] framework; it therefore follows the cost model described in Subsection 1.6.1. This attack implements the parallel version of the `isd1_parallel` attack, based on the concept and analysis described in Section 3.2. Like `isd1`, the `isd1_parallel` attack also uses a single collision level but, unlike the latter, it can distribute the computation of certain phases across concurrent `PF` machines, organized in a $\sqrt{\texttt{PF}} \times \sqrt{\texttt{PF}}$ mesh where each machine is connected to its four neighbors (north, south, west and east).

To implement this parallel version, a new attack parameter, `PF`, has been introduced in addition to those listed in the Table 2.1. As `isd1_parallel` is a modification of the `isd1`, it uses the same parameters as the latter, with the addition of the `PF` parameter, which is specific to parallelization. The `PF` parameter represents *the number of concurrent machines* over which the computation of certain stages of the algorithm can be distributed, and determines the mesh size. The `PF` parameter is not a free parameter; it must satisfy the following constraints: it must be a perfect square ($\texttt{PF} = x^2$ for $x \in \mathbb{Z}$) so that the mesh can be structured as a square; it must be even so that the two enumerations can be distributed symmetrically; it must divide the total number of candidates generated during the two subset enumerations so that each machine is assigned an identical portion of candidates to generate.

`isd1_parallel` supports both parallel and serial execution, without distributing the execution across the `PF` machines. The execution mode can be selected at two distinct stages: at compile-time by adding the `PARALLEL=1` compilation flag to obtain an executable that runs the parallel version by default, or by not adding any flags to obtain a serial executable. Alternatively, at run-time, by setting the environment variable `PARALLEL=1` to force parallel execution, or `SERIAL=1` to force the serial version.

Since the two versions, serial and parallel, use the same code and the same parameters, this design choice makes it easier to verify the correctness of the parallel version's implementation by comparing it directly with the serial version.

As described previously, the search phase can in turn be divided into three distinct sub-phases: subset enumeration, sorting and collision search; these sub-phases will be analyses in detail, describing the modifications made compared to isd1 and the detailed application

of parallelism.

### 3.2.1. Subset Enumeration

In `isd1`, the two lists of candidates containing the combination of the weight vector `PI` and the dimension $\frac{(k+\mathtt{L}-\mathtt{Z})}{2}$, along with the corresponding partial sum, corresponding respectively to the first and second halves of the error indexed by the IS, are constructed sequentially, one after the other. Each list contains $\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ elements, each represented by three distinct components, stored in three separate lists: `L_set`, which contains the `PI` positions selected from the $\frac{(k+\mathtt{L}-\mathtt{Z})}{2}$ available positions that generate the corresponding error component; `L_sum`, which contains the associated partial sum; and `L_01`, which contains a bit indicating which of the two subsets the element belongs to. The generation of candidates is performed via the `subset` function, which sequentially calculates all combinations and their corresponding partial sums, thereby populating the lists.

In `isd1_parallel`, however, the two lists of candidates are calculated simultaneously using the `subsetIterativeParallel` function. The `PF` available machines are split evenly: $\frac{\mathtt{PF}}{2}$ machines are assigned to the first list of candidates and the remaining $\frac{\mathtt{PF}}{2}$ to the second. Within each enumeration, the $\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ elements are partitioned into $\frac{\mathtt{PF}}{2}$ contiguous ranges of equal size, so that machine $i$ is responsible for generating only the elements whose global index falls in the interval $[start_i,\ end_i)$, where $start_i = i \cdot \frac{2}{\mathtt{PF}}\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ and $end_i = start_i + b$, with $b = \frac{2}{\mathtt{PF}}\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$. Each machine stores its portion in three dedicated local buffers: `L_setPerThread`, which contains the error vector formed by the `PI` selected positions; `L_sumPerThread`, which contains the corresponding partial sum; and `L_01PerThread`, which holds a bit indicating whether the element belongs to the first or second list of candidates. Each of the `PF` machines uses its own copy of these buffers to avoid write conflicts.

Since the two enumerations are independent, they are started concurrently: $\frac{\mathtt{PF}}{2}$ machines handle the first subset and the remaining $\frac{\mathtt{PF}}{2}$ the second. Within each enumeration, the $\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ elements to be calculated are distributed equally among the assigned machines, so that each machine calculates exactly $b = \frac{2}{\mathtt{PF}}\binom{(k+\mathtt{L}-\mathtt{Z})/2}{\mathtt{PI}}$ elements.

### 3.2.2. Sorting

Once the subset enumeration is complete, after the two subsets have been generated, they are then sorted.

In `isd1`, the lists `L_set`, `L_sum` and `L_01` are sorted using the `sorting` function, which

Table 3.1: Complexity of parallelization strategies, where $s$ is the side length of the mesh and $b$ is the size of each machine's local buffer

| Version | $C(s^2b)$ | $T(s^2b)$ |
|---|---|---|
| Local-then-Global Sorting | $(4b\log_2 b + 4)\, s^3 + \mathcal{O}(s^2)$ | $\frac{\log_2 b(\log_2 b+1)}{2} + 9s\log_2 b + \mathcal{O}(1)$ |
| Only Global Sorting | $\left(2b\,(1+\log_2 b)^2 + 6b - 2b\log_2 b - 4\right) s^3 + \mathcal{O}(s^2)$ | $9s\frac{\log_2 2b(\log_2 2b+1)}{2} + \mathcal{O}(1)$ |

internally implements the one-dimensional odd-even mergesort [24] sorting network, which performs all comparisons unconditionally, regardless of the data, without ever skipping an operation based on the value of the elements. This choice is compatible with the Boolean circuit model used in CAT [9] (Subsection 1.6.1). The original `sorting` function simultaneously sorts the three lists, `L_set`, `L_sum` and `L_01`, using the `L_sum` list as the sort key. Whenever two elements of the `L_sum` list are compared and swapped, the same swap is also applied to the corresponding positions in `L_set` and `L_01`, so that the three lists remain aligned at the end of the sort.

In `isd1_parallel`, once the subset enumeration is complete, there are local `PF` buffers distributed across the machines in the mesh; this necessitates a different approach to that used for `isd1`. An initial solution would be to aggregate all the buffers into a single global list and sort it using the previous `sorting` function; however, this strategy does not make use of the available parallelism. The solution adopted is instead to use a two-dimensional sorting network, specifically designed to exploit the mesh structure of the concurrent machines and operate in parallel.

The two-dimensional sorting network chosen is *LS3* [26], which is also used in [8] and operates on an $\sqrt{\texttt{PF}} \times \sqrt{\texttt{PF}}$ mesh in which each machine communicates exclusively with its four immediate neighbors. When using LS3, there are two distinct strategies available, which differ in terms of whether or not the local buffers are sorted prior to the global sort on the mesh. The analysis of the cost in terms of logic gates $C(s^2b)$ and sorting time $T(s^2b)$ for the two strategies is shown in Table 3.1. To make the comparison between the various complexities clearer and quicker, the following notation is adopted in Table 3.1: $s = \sqrt{\texttt{PF}}$ represents the side length of the mesh, $b = \frac{2}{\texttt{PF}}\binom{(k+\texttt{L}-\texttt{Z})/2}{\texttt{PI}}$ represents the size of each machine's local buffer, and finally $e = s^2b = 2\binom{(k+\texttt{L}-\texttt{Z})/2}{\texttt{PI}}$ represents the total number of elements to be sorted.

When comparing the two parallel strategies (Table 3.1), the strategy of sorting locally and then globally is preferable both in terms of the number of logic gates used and in terms of the time required for sorting. As regards cost, the dominant term for the local-then-global sort is $(4b\log_2 b + 4)\, s^3$, whereas for the global-only sort it is $\left(2b\,(1+\log_2 b)^2 + 6b - 2b\log_2 b - 4\right) s^3$; therefore, the first strategy is preferable as it requires fewer logic gates. As for time complexity, however, the dominant term for the first strategy is $9s\log_2 b$ whilst for the second it is $9s\frac{(\log_2 2b)^2}{2}$; in this case too, the first sorting strategy is more efficient.

When comparing the two strategies, the most efficient, in terms of both the number of logic gates used and the time required for sorting, is to *sort each buffer locally first and then sort globally.* At the end of the subset enumeration, the `subsetIterativeParallel` function must sort each buffer locally using the one-dimensional odd-even mergesort sorting network, making each buffer of the parallel machines' `PF` available in a locally sorted state. Subsequently, instead of the `sorting` function used in `isd1`, the `ls3Sort` function is used, which implements the LS3 algorithm with the *odd-even merge network* instead of simple element comparisons, so as to merge two buffers that are already locally sorted in an ordered manner at each step of the sorting algorithm.

### 3.2.3. Collision Search

Once the global sorting of all elements has been completed, we move on to the final sub-phase: searching for collisions between the elements of the sorted list.

In `isd1`, each element of `L_sum` is compared with the next `WI` elements in the list to check for a collision. Two elements collide if they have the same partial sum and belong to two different subsets, i.e. if the corresponding elements in `L_sum` match and the elements in `L_01` differ. The elements of `L_set` corresponding to the pair under examination are saved in the `todo_set` list, whilst a bit is saved in the `todo_check` list indicating whether the pair corresponds to a collision or not. Once the collision search for all elements of `L_sum` has been completed, the `todo_set` and `todo_check` lists are processed to verify which pairs that passed the first filter actually generate the sought-after permuted error vector, by calculating the residual error component and verifying its Hamming weight.

In `isd1_parallel`, the globally sorted list is divided among the local `PF` buffers of the parallel machines, each containing $\frac{2}{\mathtt{PF}}\binom{(k+\mathtt{L}-\mathtt{z})/2}{\mathtt{PI}}$ elements. Each machine performs the same sliding-window comparison procedure used by `isd1` on its own local buffer via the `searchCollisions` function. The handling of elements at the boundary between adjacent buffers is carried out via arithmetic on the global indices. Each parallel machine has a direct connection to the adjacent machine; in this way, the last elements of each local

buffer, which have no subsequent elements within it, can be compared directly with the elements of the adjacent machine's buffer, by deterministically calculating their local index within the machine. Upon completion, the results produced by each machine are aggregated into the global lists `todo_set` and `todo_check` before proceeding in the same manner as in `isd1`.

# 4 | Experimental Evaluation

This chapter presents the results of the experimental campaign conducted to evaluate the performance of the implemented parallel version, `isd1_parallel`, compared to its original serial version, `isd1`, within the CAT [9] framework.

The purpose of the experiments conducted is to determine whether, and to what extent, parallelization across a network of `PF` concurrent machines yields a tangible benefit in terms of circuit depth, i.e., the attack's execution time. In particular, we aim to verify whether the additional cost introduced by parallelization, due primarily to the parallel sorting phase, is effectively offset by the gain derived from the concurrent execution of the attack compared to the original serial version.

The experiments were conducted on the parameters of several cryptographic schemes nominated for the fourth round of the NIST standardization process for PQC [2]. These schemes are classified into three security levels: L1, L3, and L5, defined by analogy with the strength of AES algorithms [36]: Level L1 corresponds to security equivalent to AES-128, i.e., an attack cost of at least $2^{143}$ gates; Level L3 corresponds to AES-192, with a minimum cost of $2^{207}$ gates; Level L5 corresponds to AES-256, with a minimum cost of $2^{272}$ gates. These values are expressed in terms of gate count; however, in the presence of parallelism, the relevant metric is not the total gate count, but rather the circuit depth: it is this that determines the actual execution time of an attack carried out by an adversary with access to parallel resources. Consequently, the objective of the experimental campaign is to evaluate the minimum depth achievable by `isd1_parallel` for each set of parameters and compare it with that of `isd1`, in order to determine whether the parameters standardized by NIST maintain an adequate margin of security even against a parallel adversary.

## 4.1. Methodology

This section describes the methodology adopted during the experimental campaign, outlining the tools implemented, the design choices made, and the structure of the collected

data. The purpose of the experiments is to evaluate whether parallelizing the `isd1` attack, in its `isd1_parallel` variant, offers a tangible advantage in terms of depth compared to its original serial version, given the same attack parameters. In particular, we aim to determine how depth evolves as the degree of parallelism varies, i.e., as the attack parameter `PF` varies, which corresponds to the number of concurrent machines on which the computations of the attack's parallelizable phases are distributed.

To this end, new binaries were implemented within the CAT framework [9] that allow for enumerating valid attack parameter configurations, measuring the expected circuit depth, and verifying its correctness through actual execution of the attack circuit. The orchestration of the entire experimental campaign was managed via a Python script that calls the necessary binaries and collects the results in a `.csv` file. To ensure the reproducibility of the results, the collected dataset has been made publicly available on 10.5281/zenodo.20729869. The analysis of the data obtained in this way was conducted using a Python notebook that calculates the metrics useful for the analysis.

The following subsections describe in detail the components and the design choices that guided them.

### 4.1.1. Implementation of Supporting Binaries

To conduct the experimental campaign, three new binaries were implemented within the CAT [9] framework: `attackparamsdepthprob`, `predicteddepth`, and `circuitdepth`.

**Enumeration of Attack Configurations** `attackparamsdepthprob` lists all valid configurations of the attack parameters and, for each one, reports the expected circuit depth, the analytically estimated probability of success, and their logarithmic ratio, `lgratio` $= \log_2 depth - \log_2 prob$, which represents the primary metric used to compare and select the optimal parameter configurations.

The need to implement this binary stems from a structural limitation of `searchparams` relative to the campaign's objectives. The different configurations of `isd1_parallel` differ from one another exclusively in the value of `PF`: they share the same probability of success, since the attack structure and the parameters determining that probability do not change, but they exhibit an increasing gate count as `PF` increases, due to the overhead introduced by the sorting network. Consequently, `searchparams` would systematically select the configuration with the lowest `PF` value, that is, the one with the minimum total cost, discarding all configurations with higher parallelism. This behavior is correct from the perspective of gate count optimization, but it is incompatible with the campaign's

objective, which is instead to verify whether the effective depth decreases as `PF` increases despite the additional gate cost.

For this reason, it was necessary to implement `attackparamsdepthprob`, which returns the complete set of valid configurations without making any selection, allowing the entire depth trajectory to be analyzed as `PF` varies and enabling the identification, for each set of problem parameters, of the configuration that minimizes the critical path.

**Predicted Depth Calculation** `predicteddepth` analytically calculates the predicted depth of the circuit for a given set of input parameters, without simulating the circuit.

For serial attacks (such as `isd1`), the binary returns the total gate count of the circuit, which, in the absence of parallelism, coincides with the depth. For parallel attacks (in our case, `isd1_parallel`), it returns the gate count of the critical path, calculated by propagating, phase by phase, the length of the longest path in the circuit graph. This approach allows one to evaluate the depth of configurations with large problem parameters, for which actual circuit simulation would take an inordinate amount of time, in constant time and with negligible computational cost.

**Measurement of Depth** `circuitdepth` verifies the correctness of the predictions produced by `predicteddepth` by simulating the circuit on concrete instances of the problem and measuring their actual depth.

Since circuit simulation has a computational cost proportional to the problem size, this binary was used exclusively on reduced parameter sets, for which simulation times remain acceptable. The verification campaign systematically compared the depth predicted by `predicteddepth` with that measured by `circuitdepth` on these reduced instances, confirming the correctness of the analytical implementation. Thanks to this validation, the results presented in the following sections are entirely based on the predictions of `predicteddepth`, which can therefore be considered reliable even for large cryptographic parameters.

### 4.1.2. Metrics Used

After collecting the data from the experimental campaign, the following metrics were defined to analyze the results obtained. Each metric captures a distinct aspect of the behavior of the parallel attack compared to the serial attack, allowing us to evaluate the benefits of parallelization in terms of both execution time and the overall cost of the hardware resources used.

**Depth Speedup** The *depth speedup* is the ratio of the depth of the serial version `isd1` to that of the parallel version `isd1_parallel`, given the same problem parameters and approach, and is defined as follows:

$$speedup = \frac{depth_{ser}}{depth_{par}} \tag{4.1}$$

This metric quantifies the reduction in the depth achieved through parallelization. A value of $speedup < 1$ indicates that the parallel version has a shorter critical path than the serial version, meaning that parallelization yields a tangible benefit in terms of execution time. A value of $speedup = 1$ indicates that the depth has not changed, while a value of $speedup > 1$ would indicate that parallelization has worsened the critical path, which is not expected but could occur for very low values of `PF` where the coordination overhead exceeds the gain obtained.

The ideal speedup would be linear in `PF`, i.e., $speedup \approx$ `PF`; in practice, due to the non-parallelizable phases of the attack—the column permutation phase and the post-processing phase, the speedup is sub-linear and converges to a finite maximum value as `PF` increases, in accordance with Amdahl's law [3].

**Area-Time Product** The area-time product combines the total number of logic gates in the circuit, which represents the hardware area required to implement the attack, with the circuit's depth, which represents the execution time, to provide a measure of the total cost of the resources used.

$$at = gate \cdot depth \tag{4.2}$$

This metric is calculated for both the serial version and the corresponding parallel version, as well as their ratio:

$$at_{ratio} = \frac{at_{ser}}{at_{par}} \tag{4.3}$$

indicates whether parallelization is worthwhile even when taking into account the additional hardware required. A value of $at_{ratio} > 1$ indicates that the depth gain achieved through parallelization offsets the additional gate cost, making the parallel attack more efficient than its serial counterpart in terms of cost-performance. A value of $at_{ratio} = 1$ indicates that the two approaches are equivalent in terms of total cost. A value of $at_{ratio} < 1$, on the other hand, indicates that the gate overhead introduced by parallelization grows faster than the depth savings, making parallelization counterproductive in terms of total resources used.

### 4.1.3. Problem Parameters Used

The problem parameters $(n, k, w)$ used in the experimental campaign are drawn from the candidate schemes in the fourth round of the NIST standardization process for PQC [2]. These schemes are classified into three security levels, *L1, L3, and L5*, defined by analogy with the strength of AES symmetric algorithms [36]: level L1 corresponds to security equivalent to AES-128, with an attack cost of at least $2^{143}$ gates; Level L3 corresponds to AES-192, with a minimum cost of $2^{207}$ gates; Level L5 corresponds to AES-256, with a minimum cost of $2^{272}$ gates.

The parameter sets used belong to three distinct code-based cryptosystems: Classic McEliece [14], BIKE [4], and HQC [1]. Classic McEliece is a public-key encryption scheme based on binary Goppa codes, which was a finalist in the fourth round but was not selected for standardization. BIKE and HQC, on the other hand, are schemes based on quasi-cyclic codes and quasi-cyclic codes with a random structure, respectively, both of which were candidates in the fourth round as alternatives with lower key sizes compared to Classic McEliece. Although the three cryptosystems differ in the algebraic structure of the codes used, all three are based on the computational difficulty of the SDP [6], which makes them susceptible to attacks from the ISD family and therefore analyzable within the CAT framework [9].

Table 4.1 lists the fourteen sets of parameters used, along with their corresponding security levels and the corresponding cryptosystems.

### 4.1.4. Attack Parameter Selection and Result Collection

The entire experimental campaign is orchestrated by the Python script `experimental.py`, which coordinates the invocation of the CAT [9] binaries, the selection of optimal attack parameters, and the collection of results into a `.csv` file. The script accepts as input one or more `.csv` files containing the problem parameter tuples $(n, k, w)$ to be analyzed.

For each set of problem parameters, the data selection and collection procedure consists of the following three steps.

**Enumeration of Configurations** `attackparamsdepthprob` is called with the number of iterations set to `I` $= 1$. Choosing `I` $= 1$ means that the circuit performs a single iteration without extracting a new IS; this choice is motivated by the fact that, for the purpose of measuring depth, the ratio between the execution times of the serial and parallel versions remains unchanged as `I` varies: performing more iterations increases the

Table 4.1: Sets of problem parameters used in the experimental campaign, along with the corresponding NIST security level and reference cryptosystem

| $n$ | $k$ | $w$ | Security level | Cryptosystem |
|---|---|---|---|---|
| 3488 | 2720 | 64 | L1 | Classic McEliece |
| 4608 | 3360 | 96 | L3 | Classic McEliece |
| 6688 | 5024 | 128 | L5 | Classic McEliece |
| 6960 | 5413 | 119 | L5 | Classic McEliece |
| 8192 | 6528 | 128 | L5 | Classic McEliece |
| 24 646 | 12 323 | 134 | L1 | BIKE |
| 24 646 | 12 323 | 142 | L1 | BIKE |
| 49 318 | 24 659 | 199 | L3 | BIKE |
| 49 318 | 24 659 | 206 | L3 | BIKE |
| 81 946 | 40 973 | 264 | L5 | BIKE |
| 81 946 | 40 973 | 274 | L5 | BIKE |
| 35 338 | 17 669 | 132 | L1 | HQC |
| 71 702 | 35 851 | 200 | L3 | HQC |
| 115 274 | 57 637 | 262 | L5 | HQC |

total execution time of each circuit and the success probability, but does not alter the ratio of improvement or deterioration between the two versions. The binary returns all valid configurations of the input parameters for `isd1_parallel`, each accompanied by the expected depth, probability of success, and `lgratio`.

**Selecting the Optimal Configuration for Each Value of `PF`** The configurations returned by `attackparamsdepthprob` are grouped by value of `PF`. For each group, the configuration that minimizes `lgratio` is selected. The `PF` value is then removed from the selected attack parameters, in order to obtain a common base configuration for both the serial and parallel versions and allow for a direct comparison with all other parameters

Table 4.3: Structure of the `.csv` file containing the results of the experimental campaign

| Column name | Description |
|---|---|
| `problem` | The name of the problem under consideration |
| `problem_params` | The parameters of the problem $(n, k, w)$ under consideration |
| `attack` | The name of the attack and the method of execution |
| `attack_params` | The parameters of the attack |
| `gate_count` | The number of gates required to implement the circuit |
| `depth` | The depth of the circuit |
| `probability` | The probability of success when using the circuit in question |

held constant.

**Collecting Results for all Configurations** For each selected optimal configuration, `predictedcostdepthprob` is invoked sequentially for `isd1` and for `isd1_parallel`, as all values of `PF` compatible with that configuration are varied. For each invocation, the gate count, depth, and success probability are recorded in the `.csv` file, along with the problem identifier and the attack configuration.

**Structure of the `.csv` file containing the results of the experimental campaign** The data collected during the experimental campaign is saved in a `.csv` file structured to uniquely map each instance of the cryptographic problem to the computational complexity metrics of the corresponding attack. Specifically, for each record, the problem parameters, the configuration parameters of the applied attack, and the related cost metrics, expressed in terms of gate count, circuit depth, and the probability of success for a single iteration, are saved, as shown in Table 4.3.

**Values of `PF`** The values of `PF` considered include all even perfect squares in the interval $\left[4, 2\binom{(k+\mathrm{L}-\mathrm{z})/2}{PI}\right]$, where the upper bound corresponds to the total number of candidates generated in the two enumerations and represents the maximum useful degree of par-

allelization: beyond this value, each machine would have fewer than one candidate to process, rendering the additional parallelism computationally meaningless.

## 4.2. Results

This section presents the results of the experimental campaign conducted on the parameter sets listed in Table 4.1. For each set, the collected data are analyzed using the metrics defined in Subsection 4.1.2, in order to assess whether and to what extent the parallelization introduced by `isd1_parallel` yields a tangible benefit compared to the serial version `isd1`.

It is important to note that the results presented in this section are based entirely on the analytical predictions of `predicteddepth`, the correctness of which has been verified via `circuitdepth` on reduced instances of the problem's parameters. This choice is motivated by the fact that the actual execution of the circuit for large cryptographic parameters would require prohibitively long simulation times, making a full-scale experimental campaign impractical without resorting to analytical predictions.

The analysis is divided into four parts. In the first part, we examine the speedup as the parallelization factor varies, to identify the value of `PF` beyond which the speedup saturates, indicating that the limit imposed by the non-parallelizable stages of the input has been reached. In the second part, the behavior of the area-time product is analyzed as `PF` varies, to determine whether the gain in depth obtained from parallelization compensates for the additional gate cost introduced by the parallel sorting network, and to identify the optimal parallelization factor in terms of cost-performance. In the third part, we report the minimum achievable depth for each NIST security level, expressed as a power of two and translated into physical execution time, in order to assess whether the standardized parameters maintain an adequate security margin even against an adversary equipped with parallel resources. In the final part, we will conduct a theoretical analysis of how the depth behaves as the parameter `L` varies.

### 4.2.1. Depth Speedup as `PF` Varies

Figures 4.1–4.14, shown below, illustrate the trend of the speedup as a function of `PF`, for all sets of problem parameters tested (Table 4.1). *Depth speedup allows us to understand how cost-effective* it is to increase the parallelization factor before saturation is reached, that is, the point at which further increases in parallelization yield no additional improvements.

Each graph shows curves corresponding to the attack configurations selected as optimal for that set of problem parameters. The attack configurations differ in the values of the parameters `PI`, `L`, and `Z`, which determine the size of the candidate lists generated during the subset enumeration phase and thus directly influence the degree of parallelism that can be exploited.

For all tested parameters and all connection configurations, the increase in speed grows as `PF` increases, confirming that `isd1_parallel` consistently reduces the critical path of the circuit compared to `isd1` in all configurations considered. The curves share a common qualitative behavior: an initial phase of rapid growth, in which an increase in `PF` produces a significant reduction in depth, followed by a gradual saturation beyond which further parallelization yields diminishing returns. This behavior reflects the structure of the algorithm, since the non-parallelizable phases, the column permutation phase and the post processing phase, constitute a lower bound on the depth independent of `PF`, in accordance with Amdahl's law [3].

Both the asymptotic speedup value and the value of `PF` at which saturation is reached depend on the configuration of the attack parameters. Configurations with high values of `L` achieve higher asymptotic speedups, since larger candidate lists allow the enumeration work to be distributed across a greater number of machines before the non-parallelizable steps become the bottleneck. Configurations with higher values of `PI` and `Z` further expand the search space, shifting the saturation point to the right. Conversely, configurations with more compact candidate lists saturate at lower values of `PF` and achieve lower asymptotic speedups, indicating that the potential for parallelization is intrinsically limited by the structure of the optimal attack parameters for that set of problem parameters.

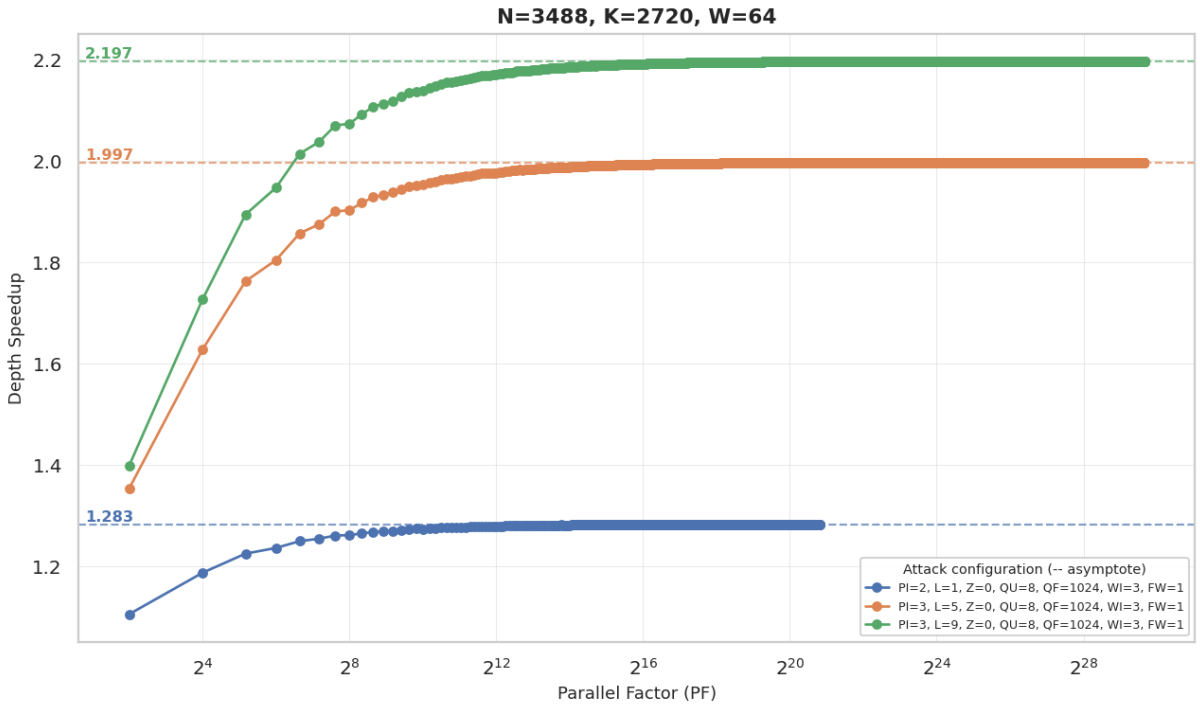


Figure 4.1: Depth speedup as a function of PF, for $(n = 3488, k = 2720, w = 64)$

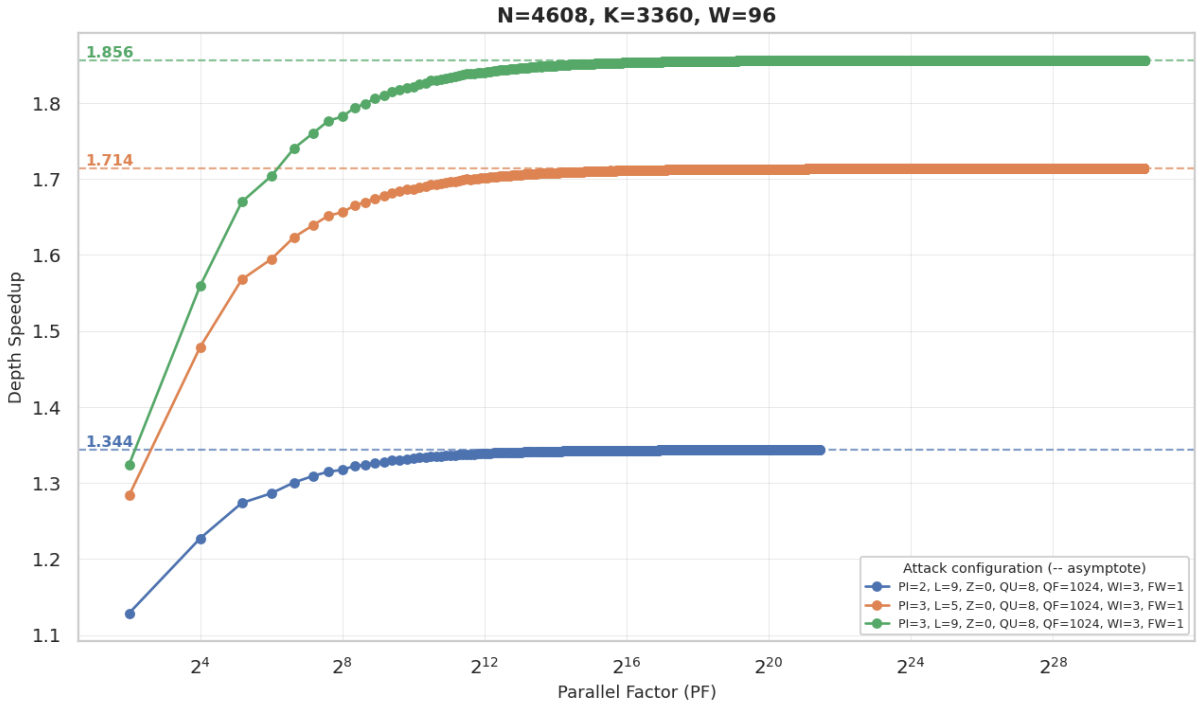


Figure 4.2: Depth speedup as a function of PF, for $(n = 4608, k = 3360, w = 96)$

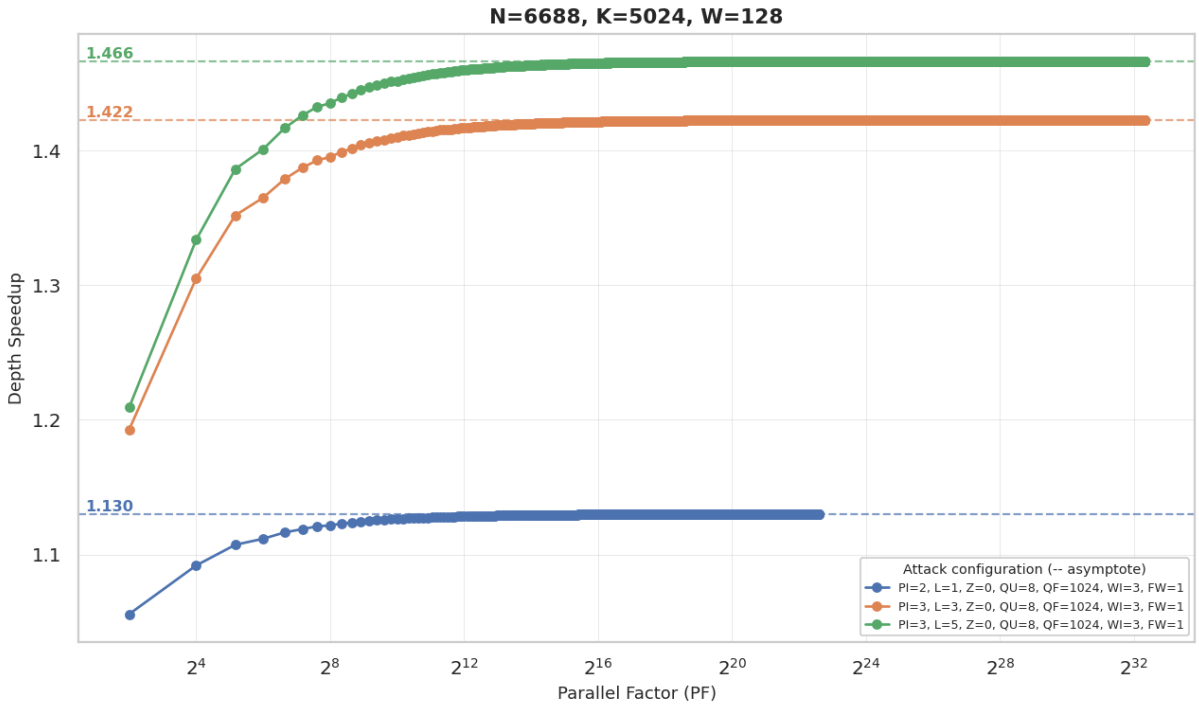


Figure 4.3: Depth speedup as a function of PF, for $(n = 6688, k = 5024, w = 128)$

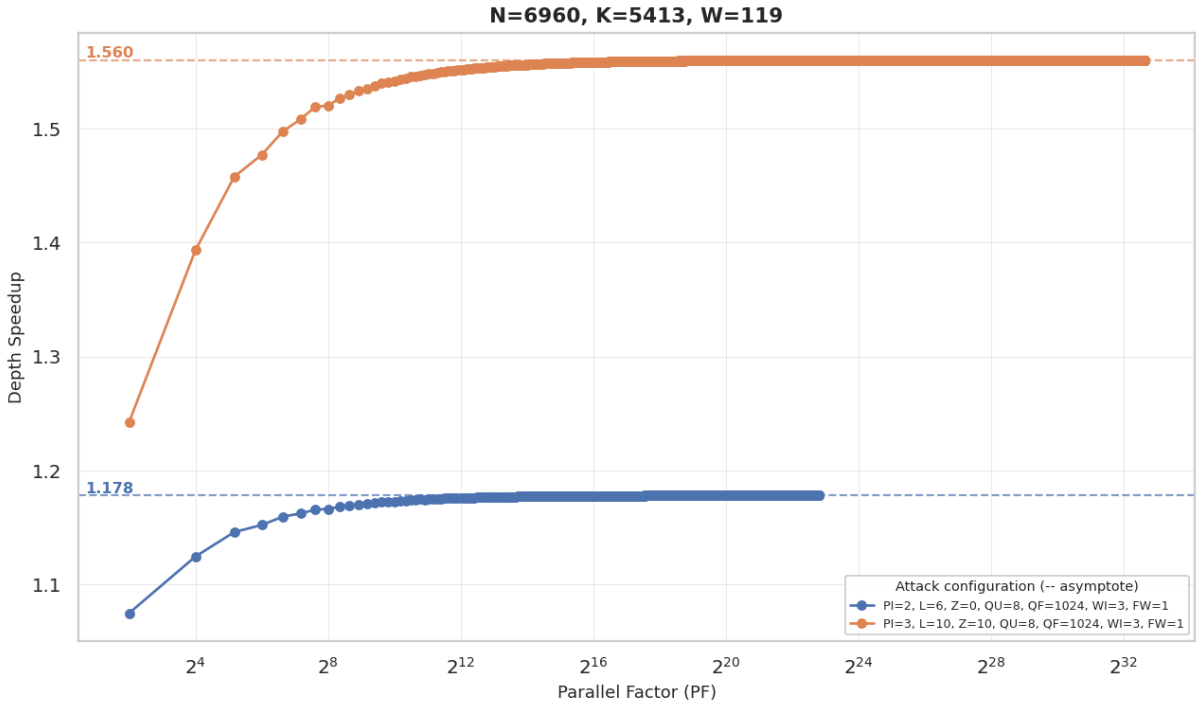


Figure 4.4: Depth speedup as a function of PF, for $(n = 6960, k = 5413, w = 119)$

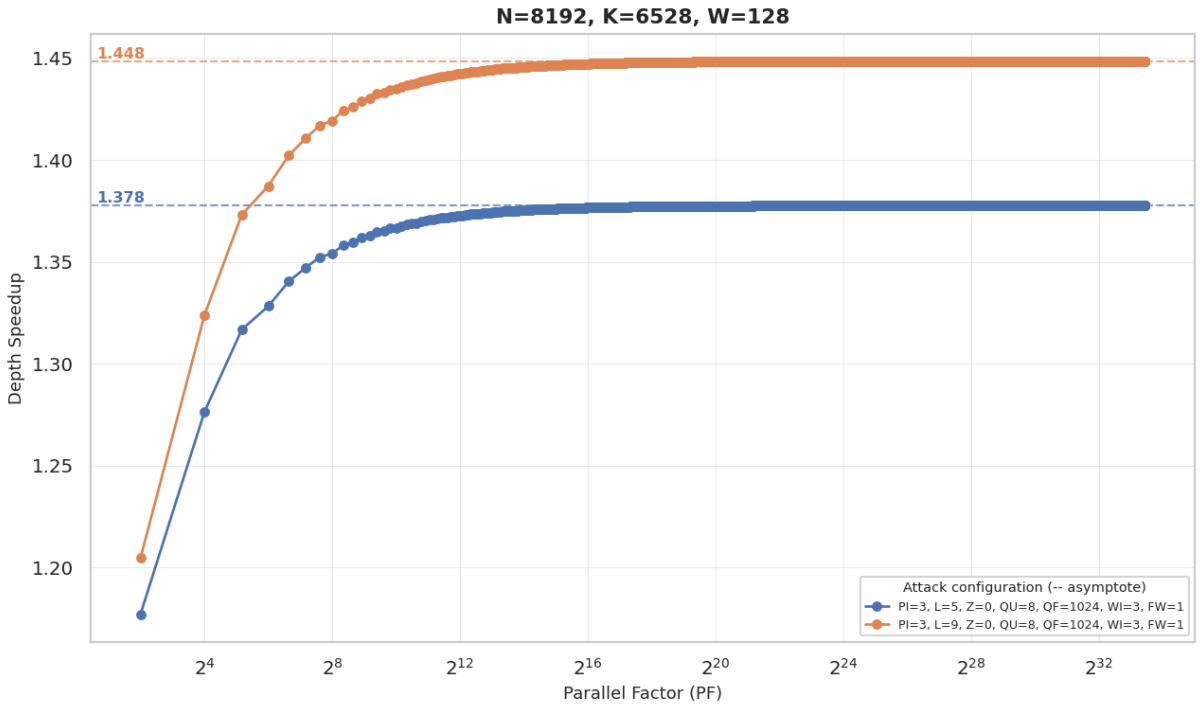


Figure 4.5: Depth speedup as a function of PF, for $(n = 8192, k = 6582, w = 128)$

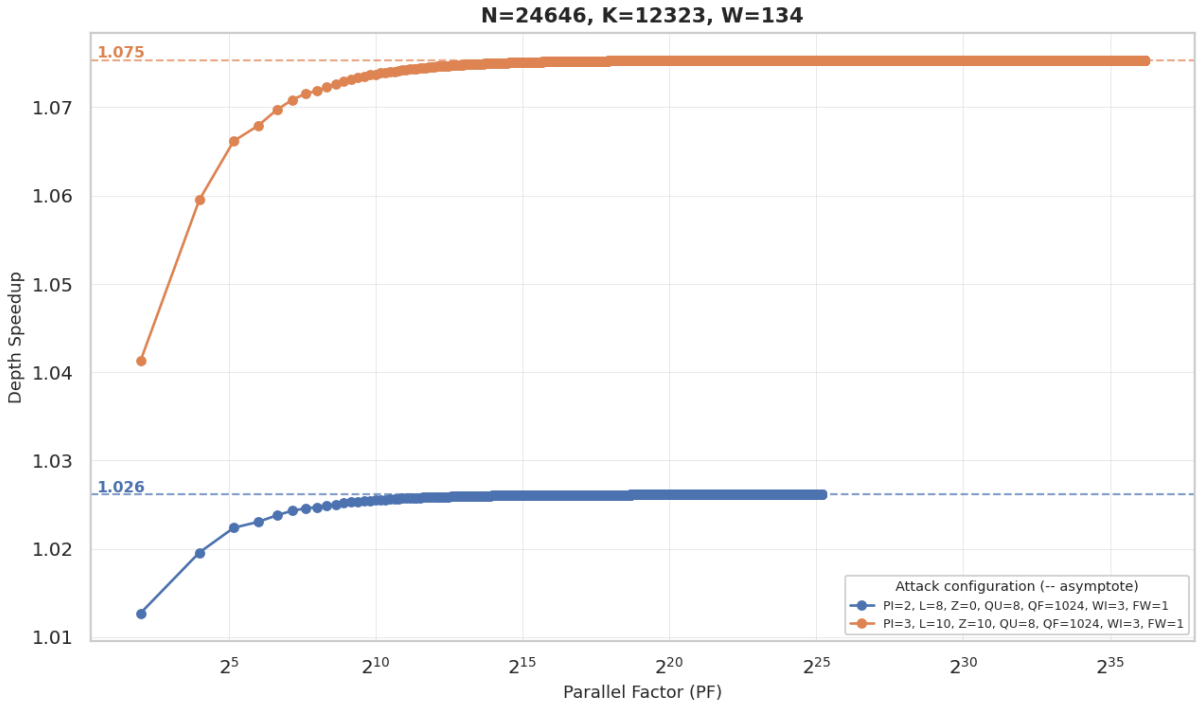


Figure 4.6: Depth speedup as a function of PF, for $(n = 24\,646, k = 12\,323, w = 134)$

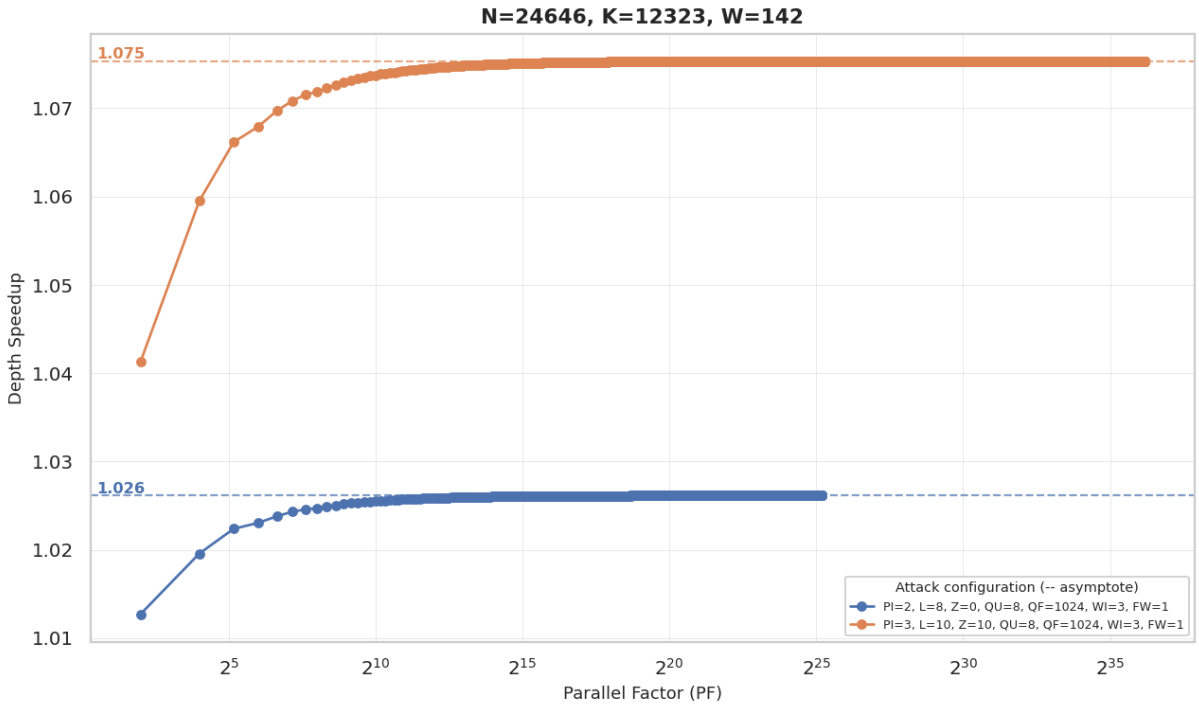


Figure 4.7: Depth speedup as a function of PF, for $(n = 24\,646, k = 12\,323, w = 142)$

Figures 4.8–4.14 do not show all PF values associated with the identified optimal configurations. In fact, above a certain PF threshold, the system reaches a state of saturation beyond which further increases in the parameter do not produce significant changes in the speedup. Therefore, the analysis was limited to a maximum PF value of $2^{33}$, a limit that is more than sufficient to describe the system's behavior under saturation conditions.

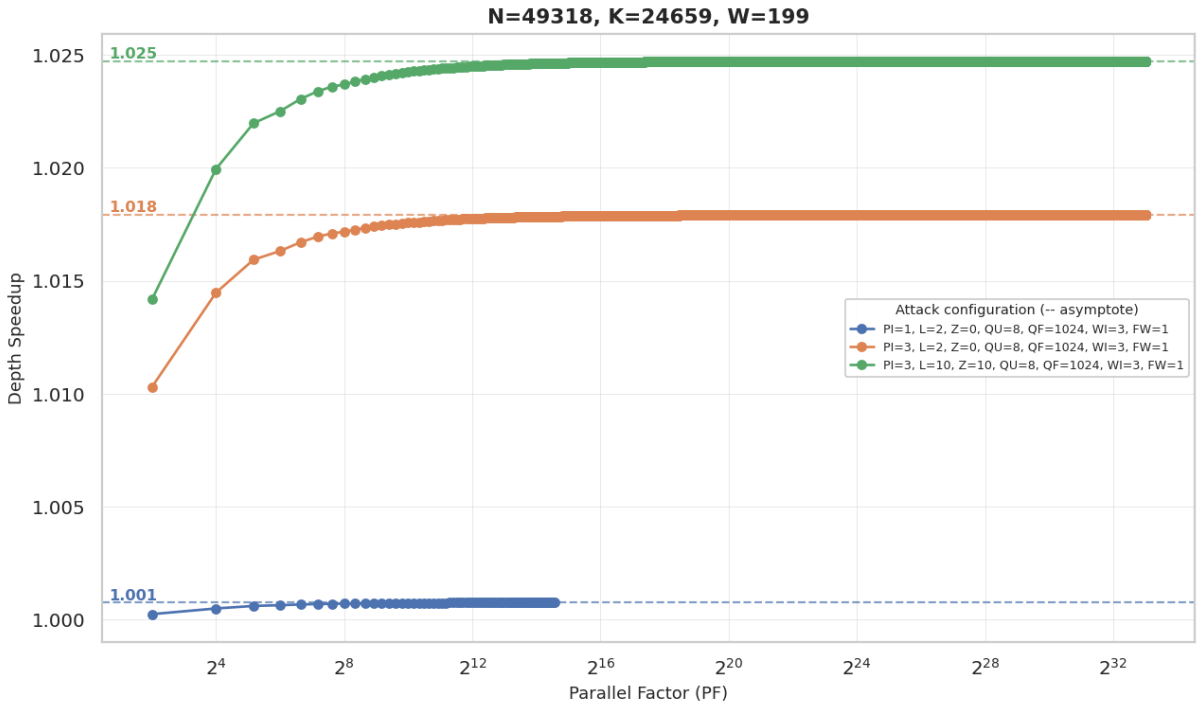


Figure 4.8: Depth speedup as a function of `PF`, for $(n = 49\,318, k = 24\,659, w = 199)$

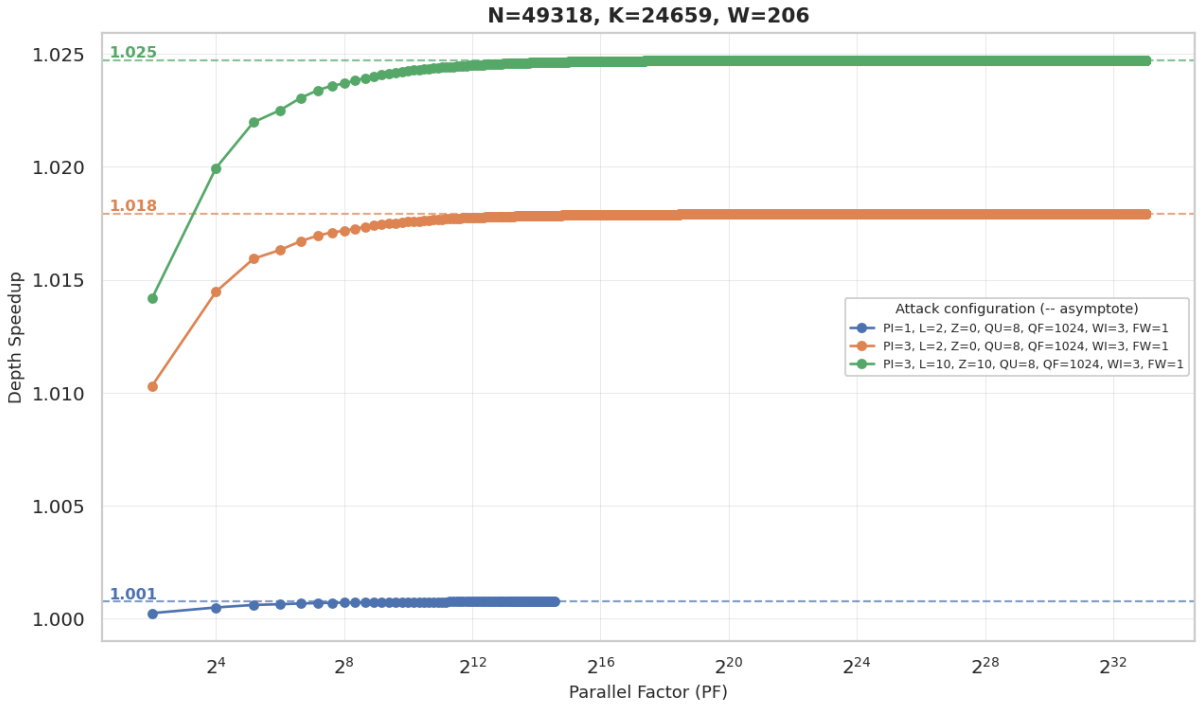


Figure 4.9: Depth speedup as a function of `PF`, for $(n = 49\,318, k = 24\,659, w = 206)$

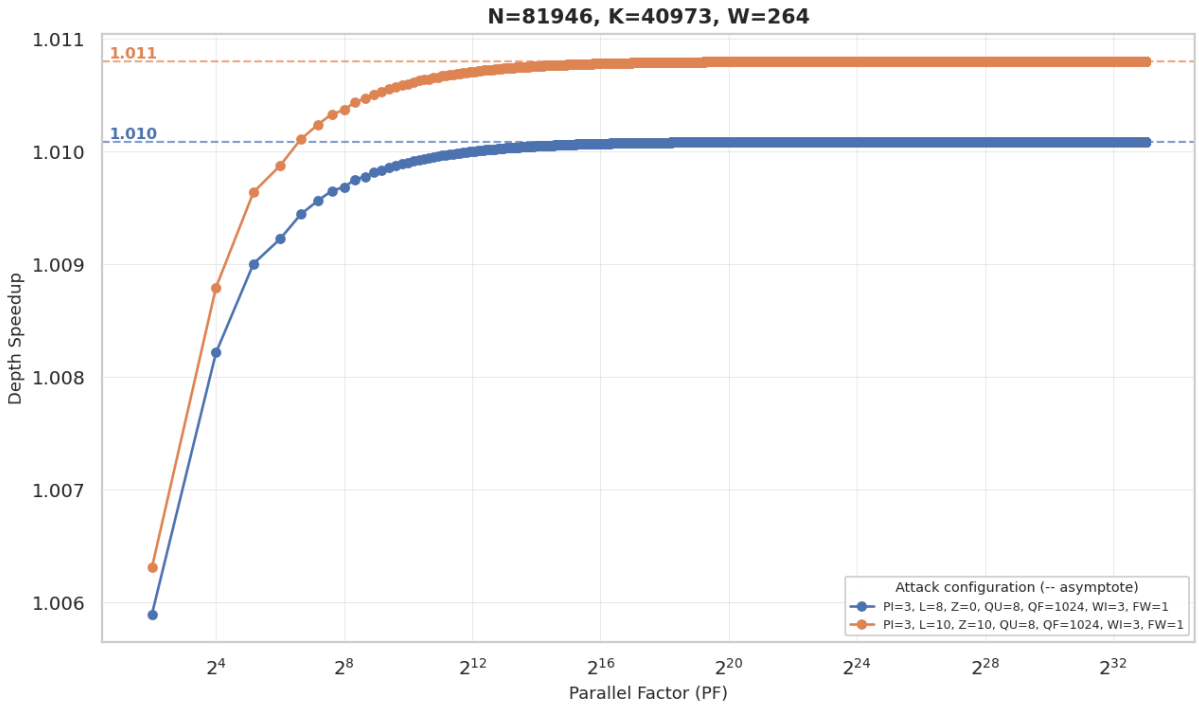


Figure 4.10: Depth speedup as a function of PF, for $(n = 81\,946, k = 40\,973, w = 264)$

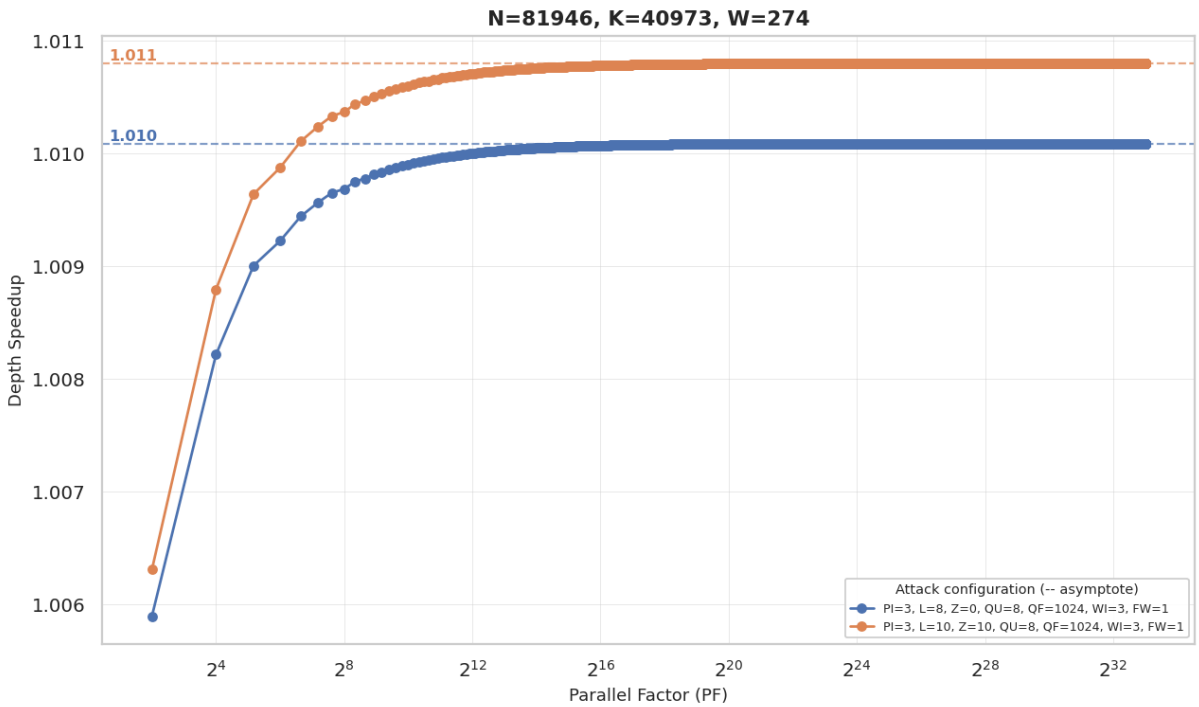


Figure 4.11: Depth speedup as a function of PF, for $(n = 81\,946, k = 40\,973, w = 274)$

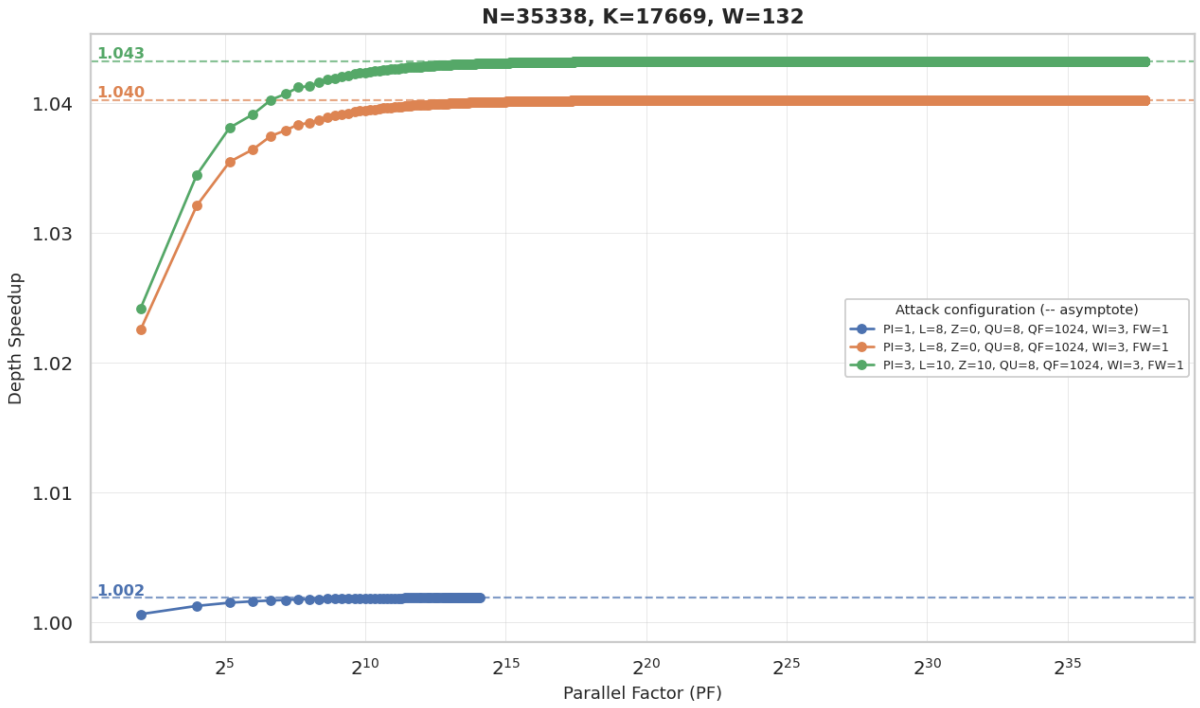


Figure 4.12: Depth speedup as a function of PF, for $(n = 35\,338, k = 17\,669, w = 132)$

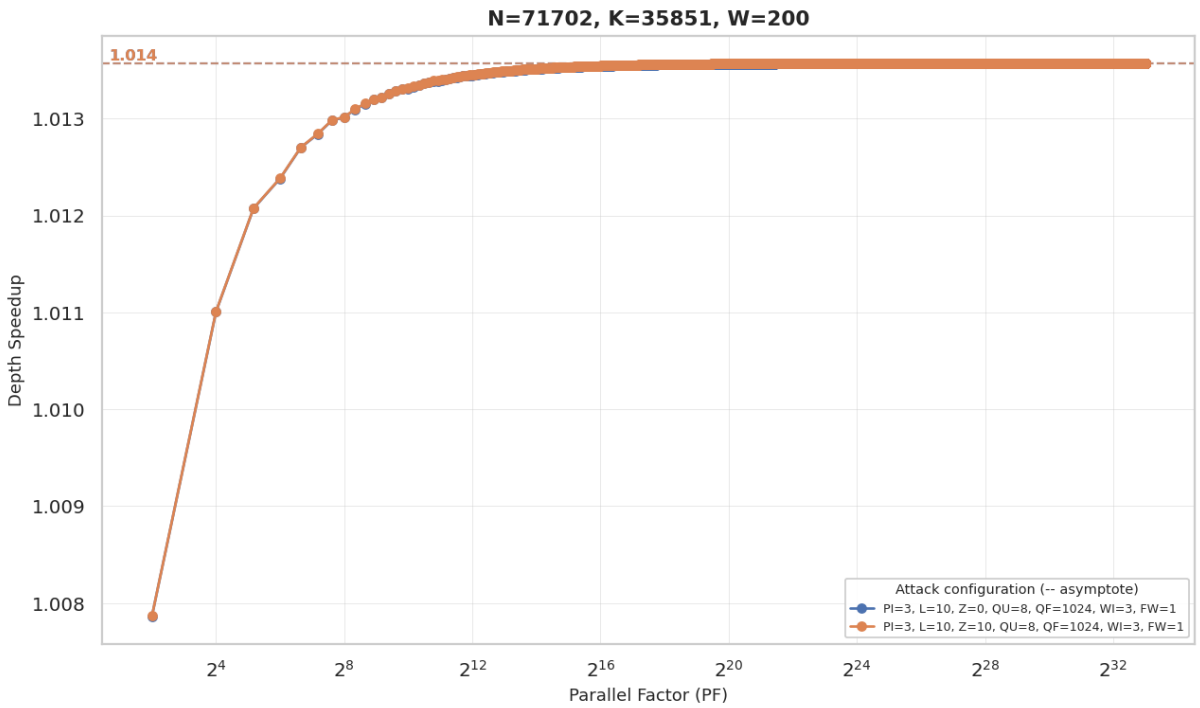


Figure 4.13: Depth speedup as a function of PF, for $(n = 71\,702, k = 35\,851, w = 200)$

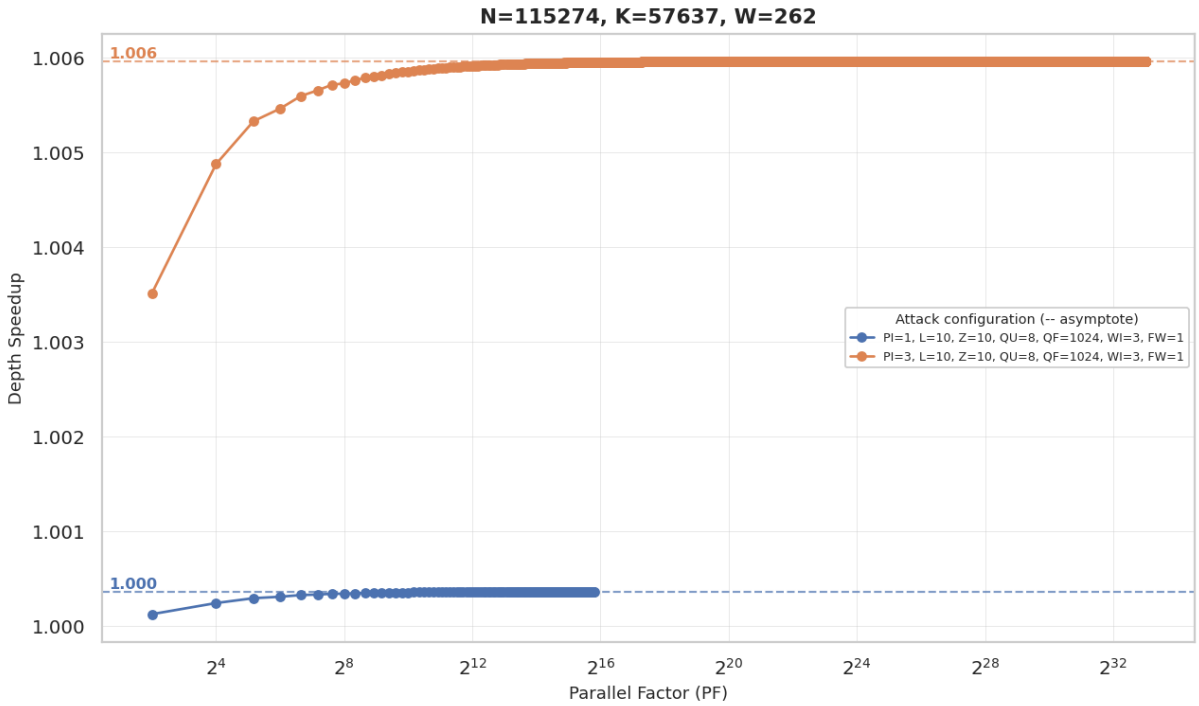


Figure 4.14: Depth speedup as a function of PF, for $(n = 115\,274, k = 57\,637, w = 262)$

As can be seen from the graphs, the maximum speedup, equal to approximately 2.2, is observed for the minimal configuration (Figure 4.1). As the problem size increases, the speedup gradually decreases, asymptotically tending toward unity.

This phenomenon is mathematically determined by the different asymptotic complexities that characterize the phases of the ISD algorithm. The transformation of the parity-check matrix into systematic form is an inherently sequential operation and is performed identically in both serial and parallel architectures. Since the computational complexity of this phase scales polynomially with respect to the size of the matrix, $\mathcal{O}\big((n-k)^3\big)$, as $n$ and $k$ increase, the execution time of this phase becomes dominant compared to the time required for collision detection. Consequently, the benefit derived from parallelization is mathematically offset by the computational overhead of the non-parallelized portion of the circuit.

### 4.2.2. Area-Time Product as PF Varies

Figures 4.15–4.28, shown below, illustrate the behavior of the area-time product as a function of PF, for all sets of problem parameters tested (Table 4.1). While depth speedup analysis focuses exclusively on reducing execution time as the number of parallel processors (PF) increases, the *area-time product introduces an evaluation of overall economic and*

*hardware efficiency*. Area corresponds to the total number of gates, while time corresponds to the circuit's critical path. Area-time product analysis is used to determine whether the time savings justify the silicon investment required by the parallelization infrastructure.

Each graph shows, for every execution configuration, the solid curve corresponding to the parallel version `isd1_parallel` and the dashed line corresponding to the value $at_{ser}$ of the serial version `isd1`, which serves as a reference threshold for evaluating the benefits of parallelization: configurations below the dashed line indicate that the gain in depth achieved by parallelization offsets the additional gate cost introduced by parallelization, making `isd1_parallel` more efficient than `isd1` even in terms of overall hardware resources; configurations above the line, on the other hand, indicate that the gate overhead exceeds the benefit gained by reducing the depth, making parallelization counterproductive from the perspective of overall cost.

The results show that the behavior of the area-time product depends heavily on the configuration of the binding parameters. For configurations with low values of `L` and `PI`, the curves start at or slightly above the serial threshold for every value of `PF` analyzed and increase progressively. In these cases, the gate overhead introduced by the sorting network is already greater than the depth gain starting from the minimum value of `PF`, and there is no parallelization factor for which the parallel version is advantageous in terms of area-time product. This behavior is attributable to the fact that configurations with compact candidate lists offer a reduced enumeration space, so the fixed cost of the sort network is not offset by a sufficient depth gain.

Conversely, for configurations with higher values of `L` and `PI`, the curves start below the serial threshold for small values of `PF`, indicating that in this regime the gain in depth is sufficient to offset the gate overhead. However, as `PF` increases, the area-time product grows and quickly exceeds the serial threshold: the cost of the sort network scales faster than the savings in depth, making additional parallelization counterproductive. Consequently, there exists an optimal value of `PF`, specific to each configuration, that minimizes this product and represents the point of greatest cost-performance efficiency for that configuration.

In summary, the analysis of the area-time product shows that parallelization is economically advantageous, in terms of resource efficiency, only when dealing with large candidate lists and at low levels of parallelism. Once the optimal value of `PF` is exceeded, the total hardware cost of the parallel approach exceeds that of the serial counterpart for any configuration analyzed. This indicates that, beyond this threshold, the benefit of parallelism is reflected solely in a reduction in execution time, but at the cost of an increase in the

computational resources used.

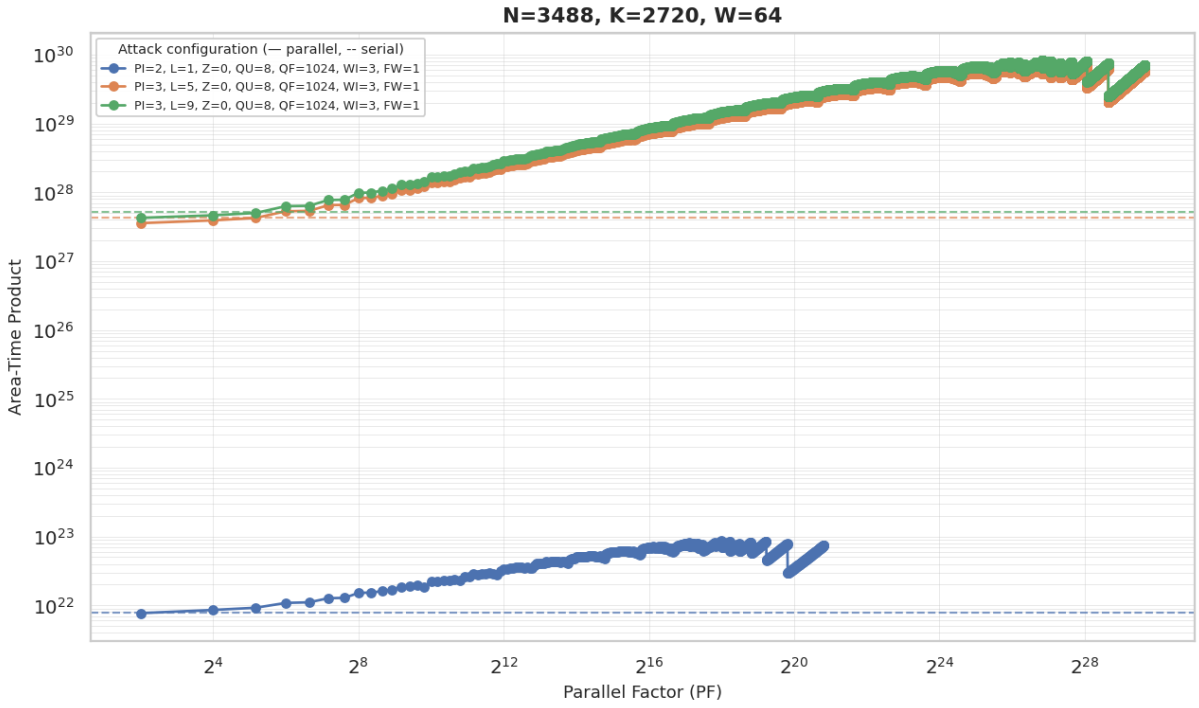


Figure 4.15: Area-time product as a function of PF, for $(n = 3488, k = 2720, w = 64)$

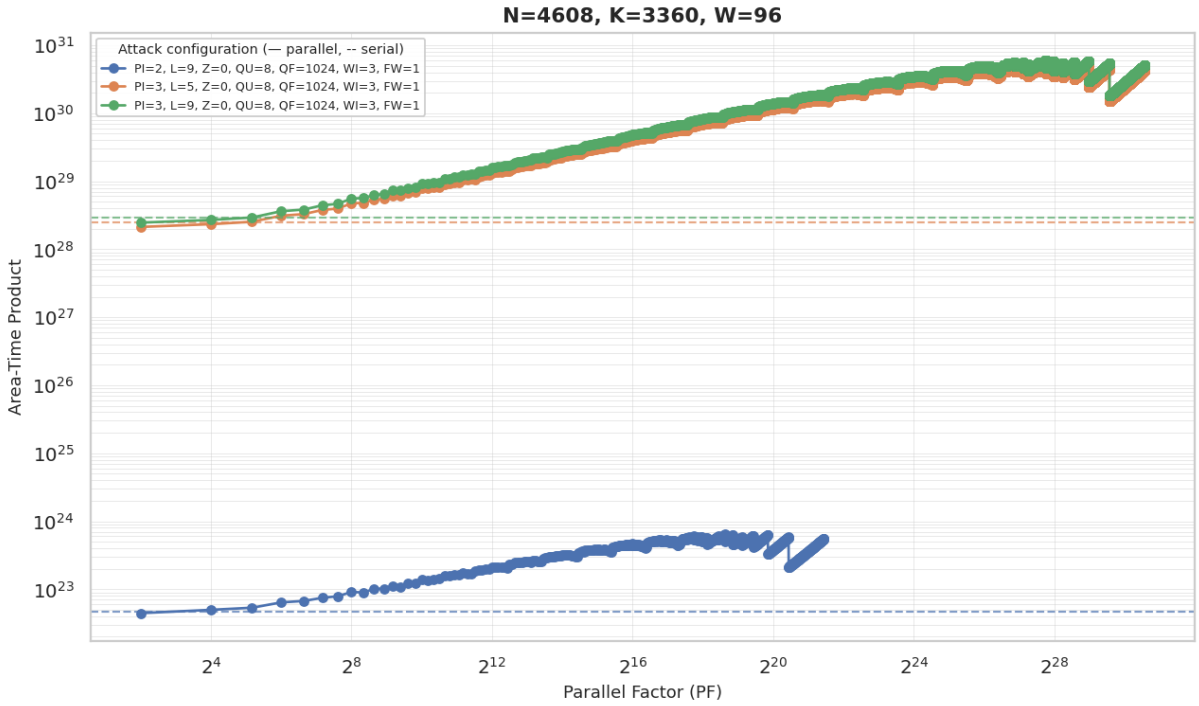


Figure 4.16: Area-time product as a function of PF, for $(n = 4608, k = 3360, w = 96)$

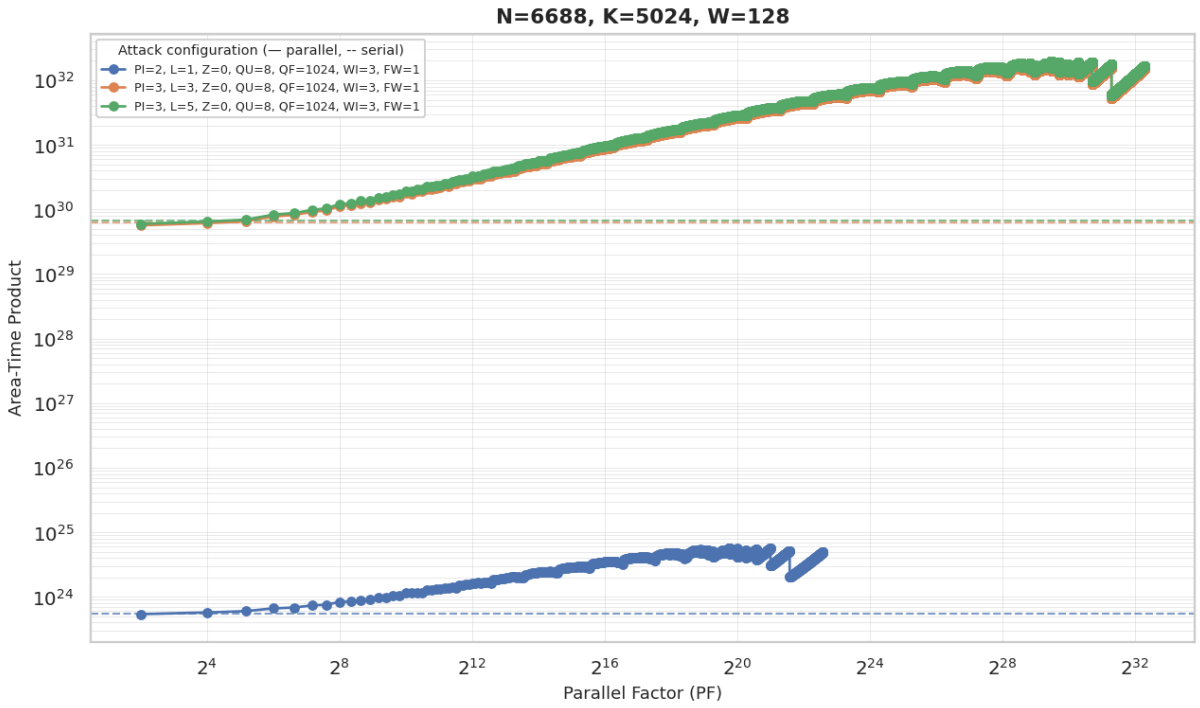


Figure 4.17: Area-time product as a function of PF, for $(n = 6688, k = 5024, w = 128)$

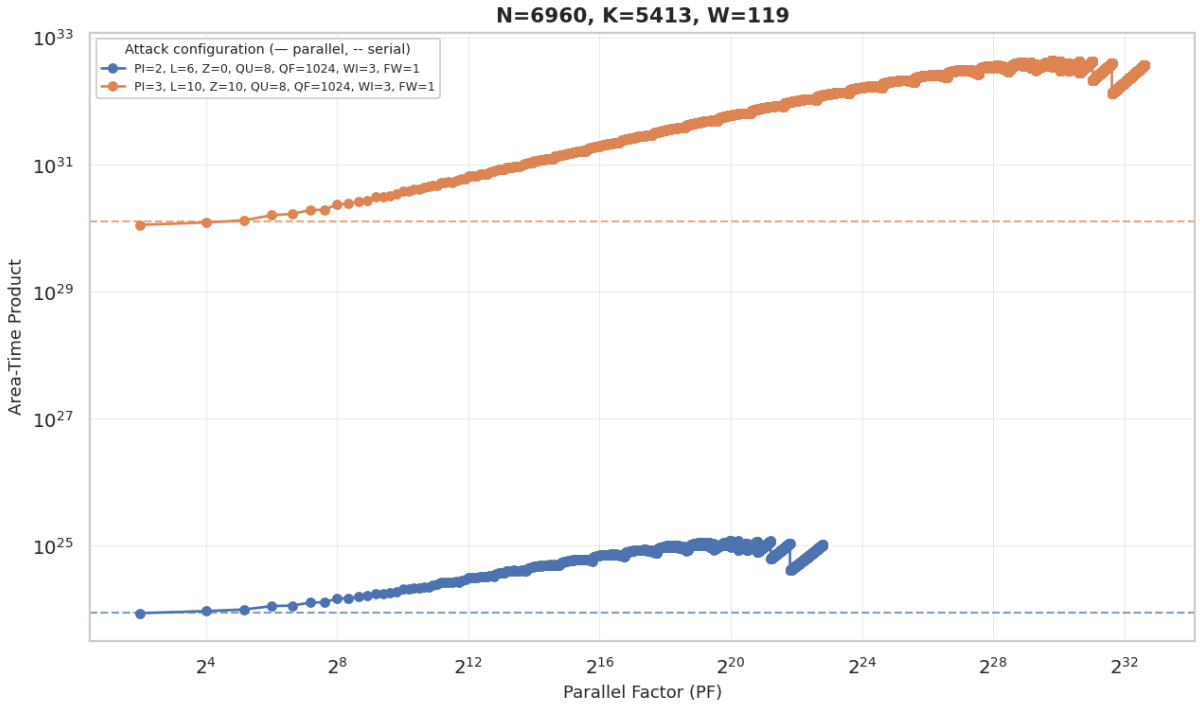


Figure 4.18: Area-time product as a function of PF, for $(n = 6960, k = 5413, w = 119)$

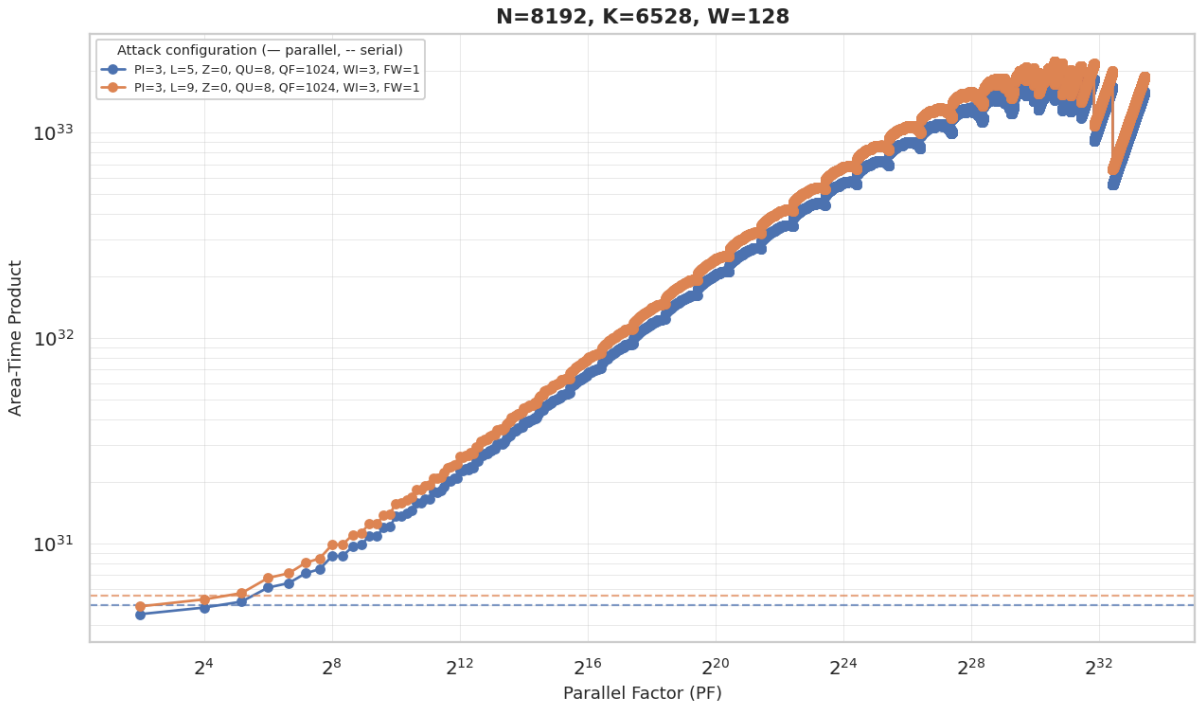


Figure 4.19: Area-time product as a function of PF, for $(n = 8192, k = 6528, w = 128)$

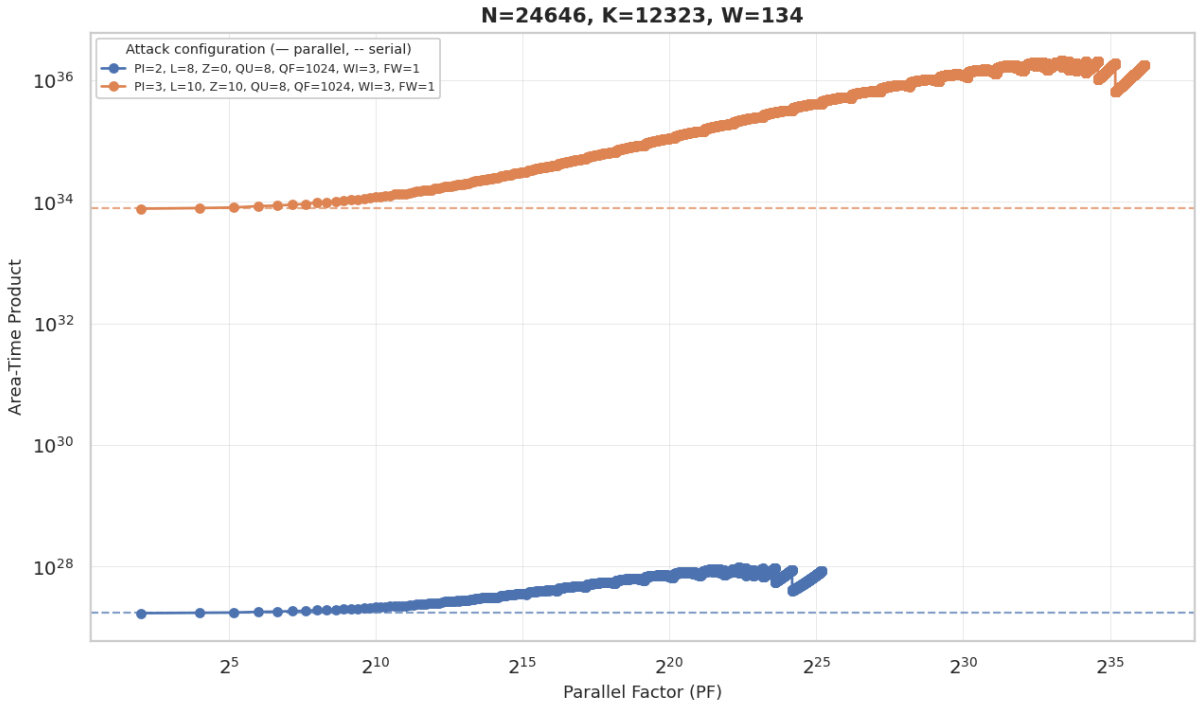


Figure 4.20: Area-time product as a function of PF, for $(n = 24\,646, k = 12\,323, w = 134)$

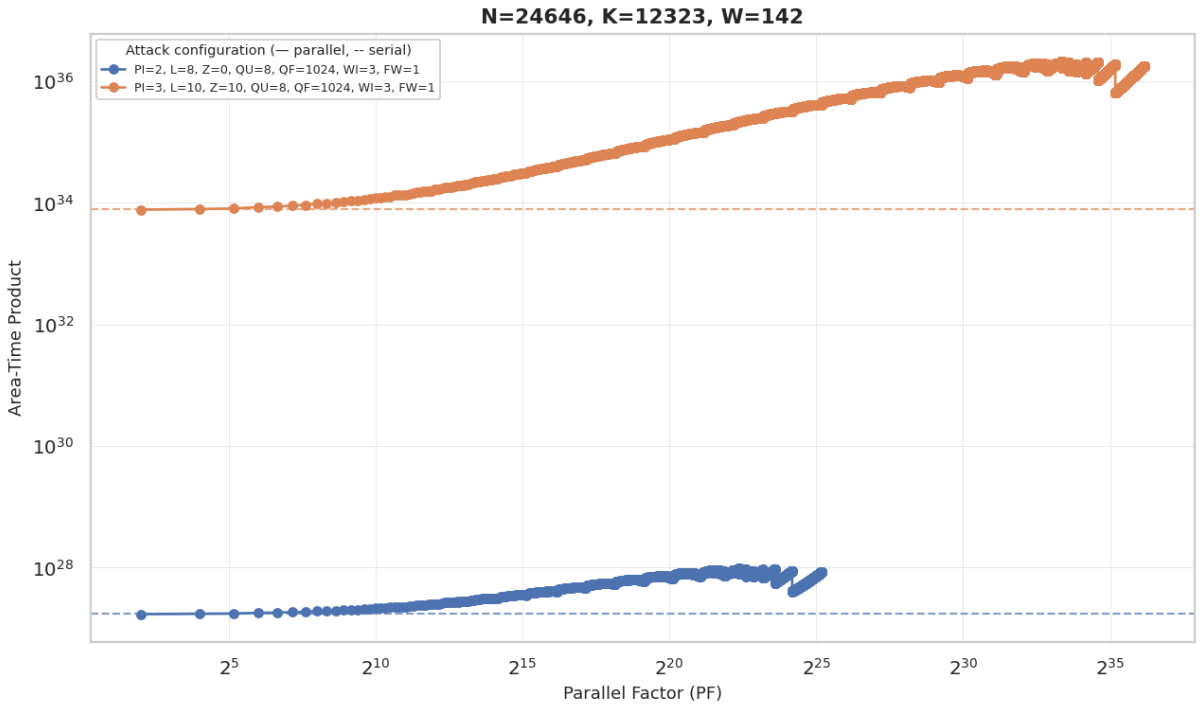


Figure 4.21: Area-time product as a function of PF, for $(n = 24\,646, k = 12\,323, w = 142)$

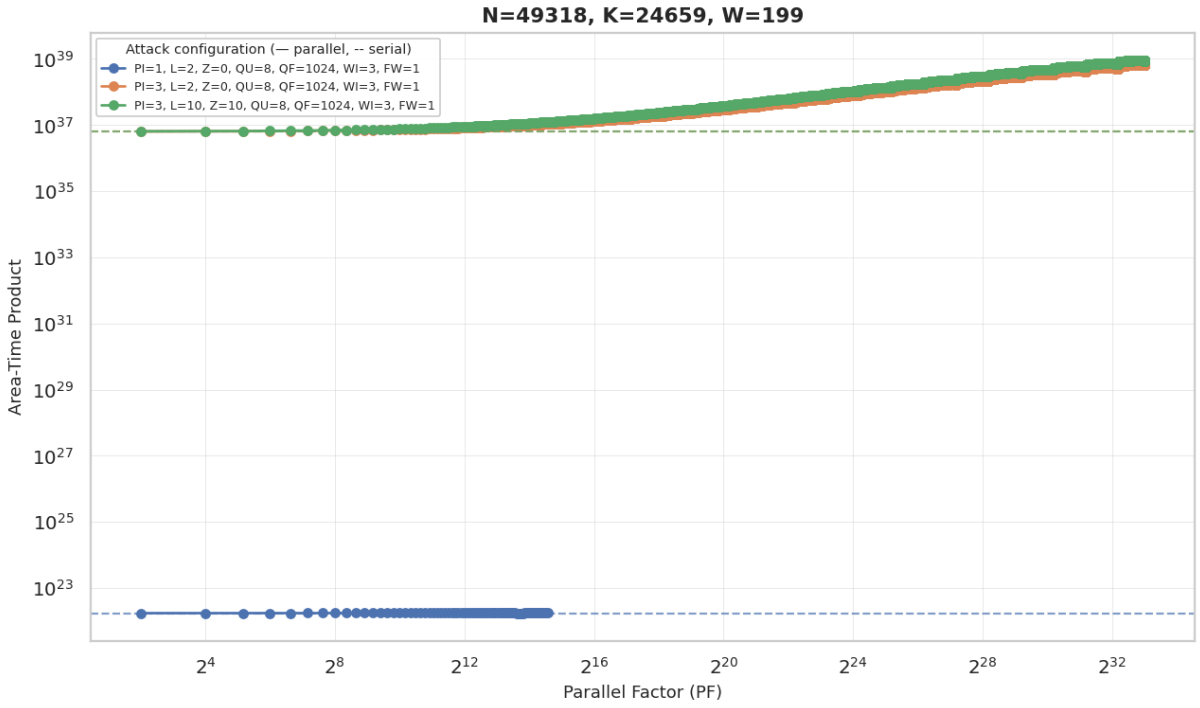


Figure 4.22: Area-time product as a function of PF, for $(n = 49\,318, k = 24\,659, w = 199)$

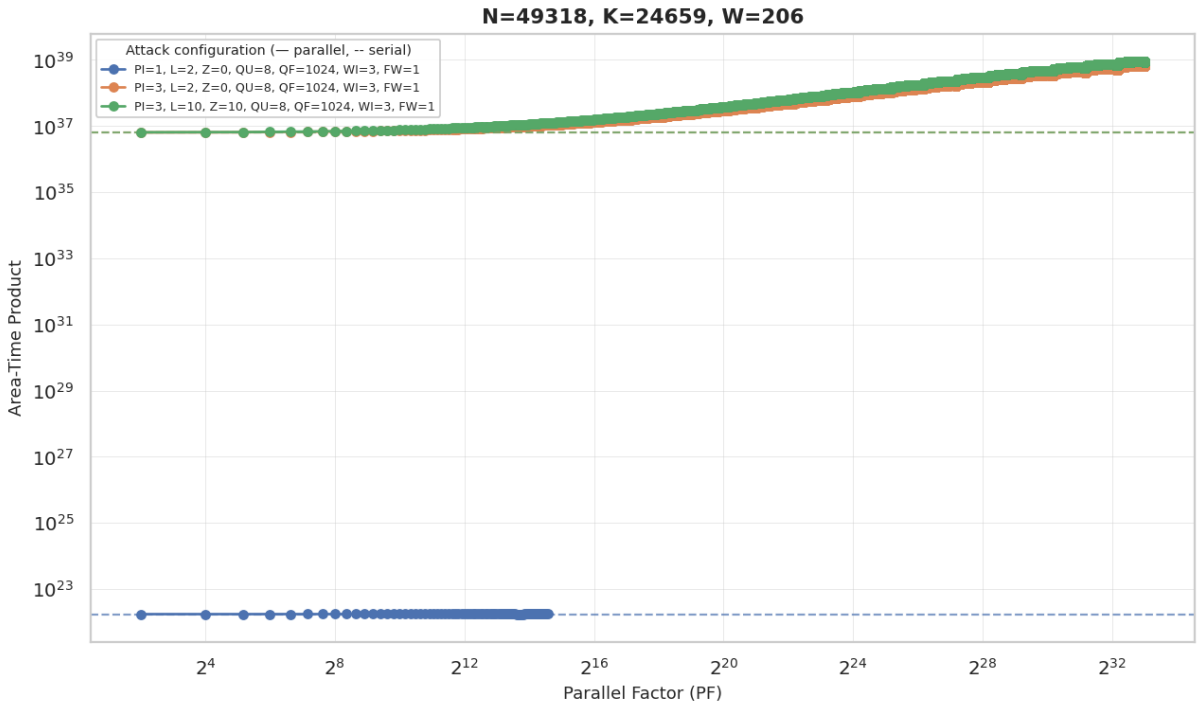


Figure 4.23: Area-time product as a function of PF, for $(n = 49\,318, k = 24\,659, w = 206)$

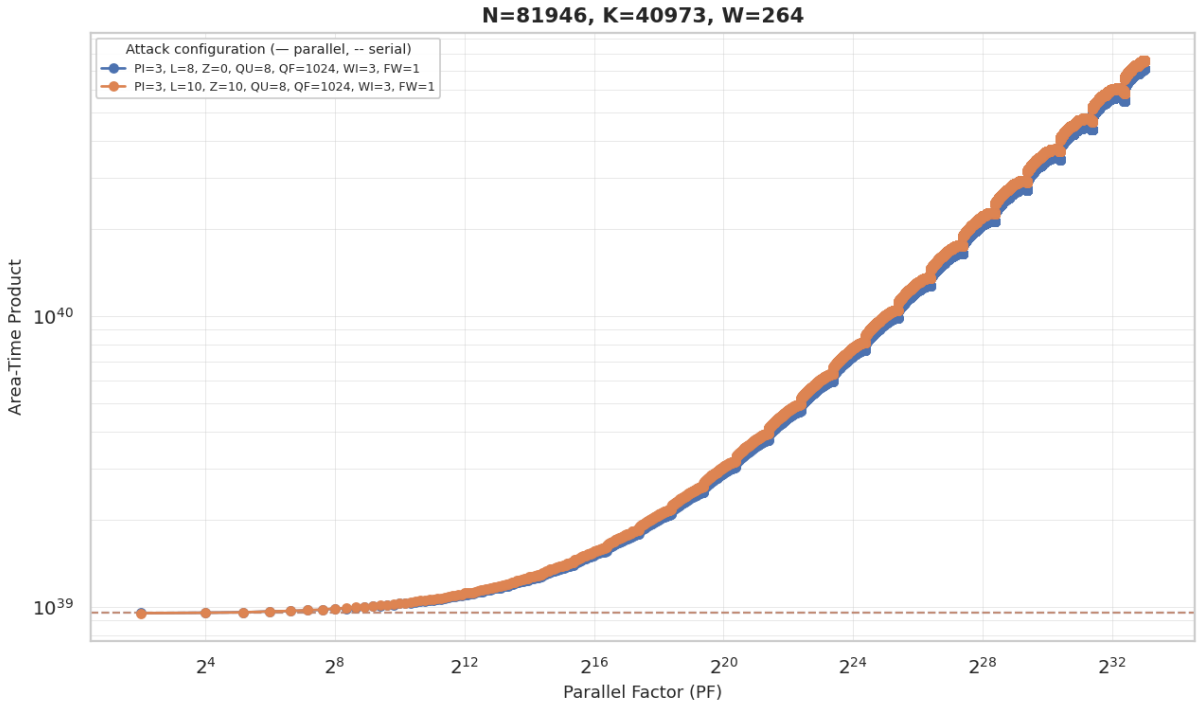


Figure 4.24: Area-time product as a function of PF, for $(n = 81\,946, k = 40\,973, w = 264)$

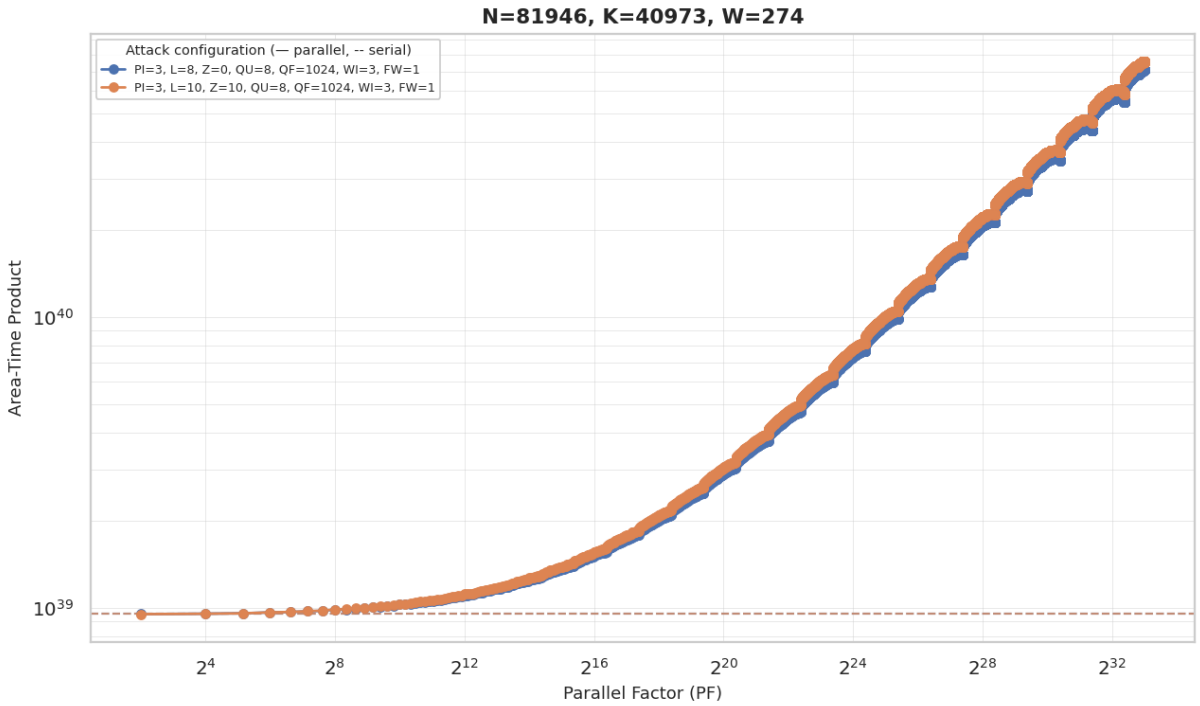


Figure 4.25: Area-time product as a function of PF, for $(n = 81\,946, k = 40\,973, w = 274)$

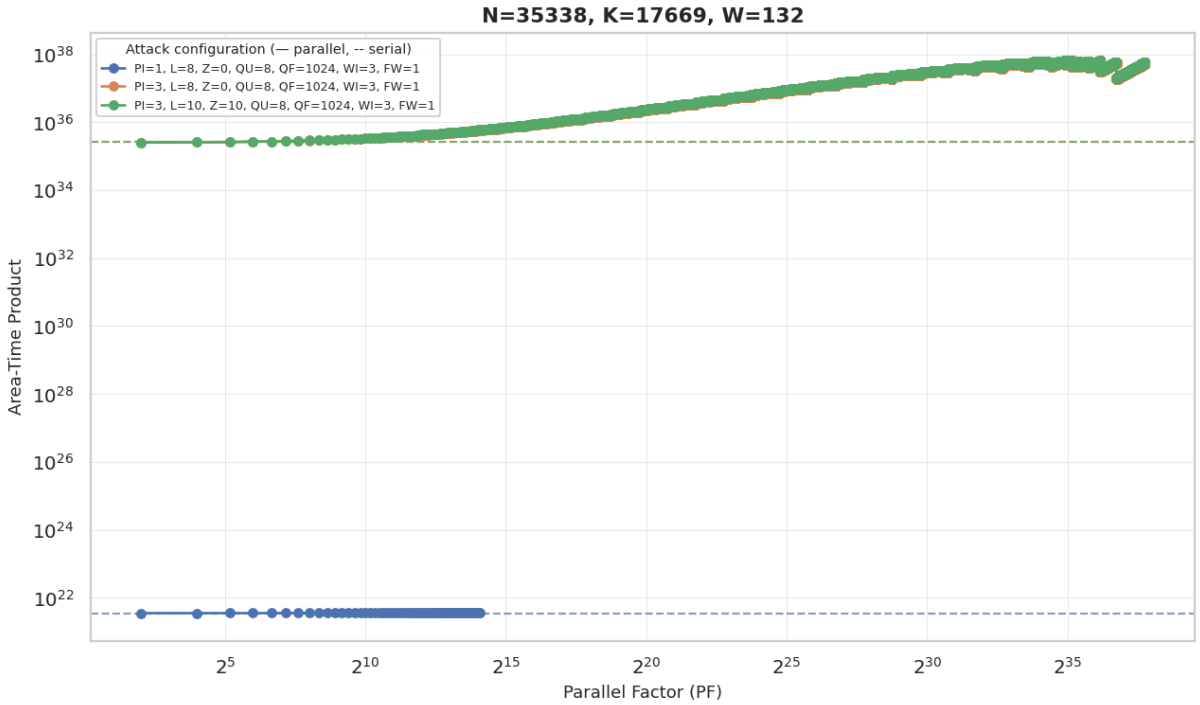


Figure 4.26: Area-time product as a function of PF, for $(n = 35\,338, k = 17\,669, w = 132)$

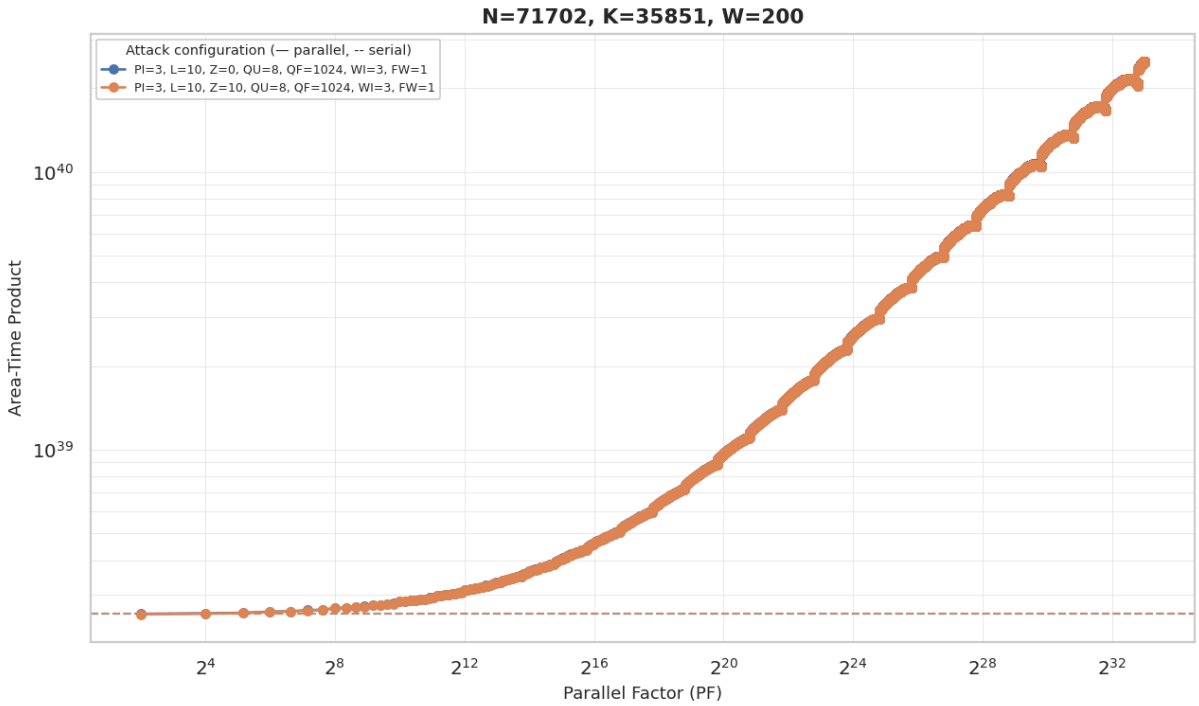


Figure 4.27: Area-time product as a function of PF, for $(n = 71\,702, k = 35\,851, w = 200)$

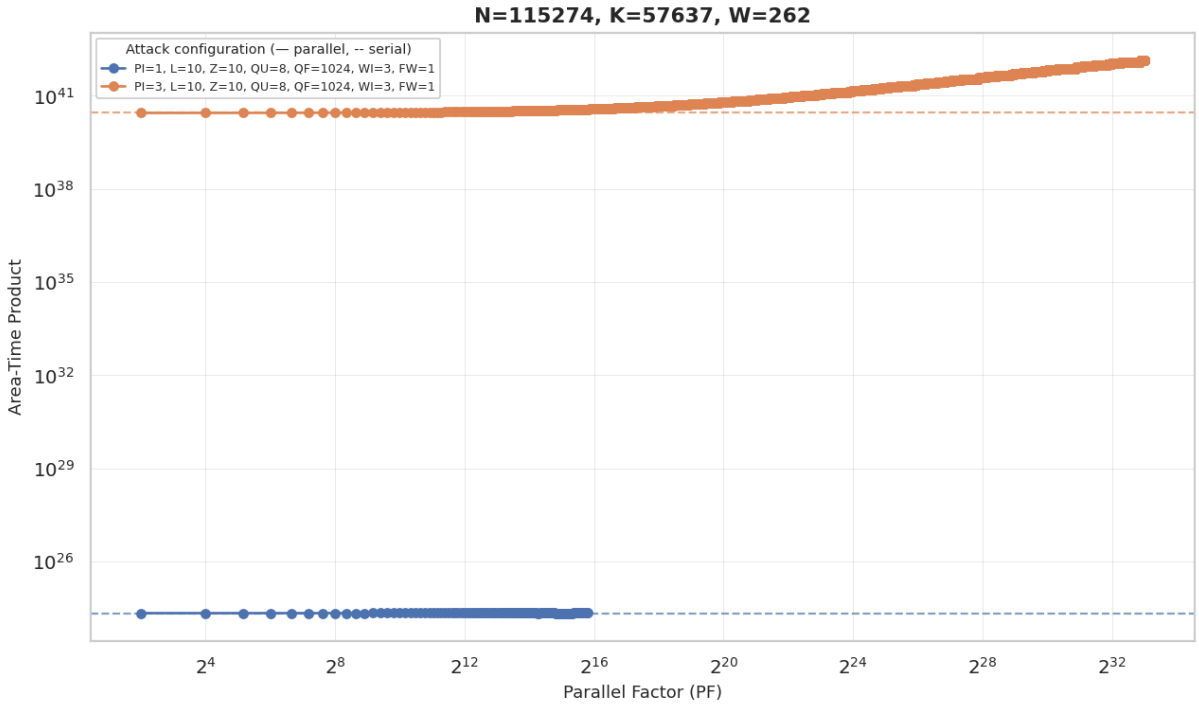


Figure 4.28: Area-time product as a function of PF, for $(n = 115\,274, k = 57\,637, w = 262)$

Table 4.5: Security margin for the sets of problem parameters that belong to security level L1

| Problem parameters | Attack configuration | Depth | Probability | Security margin |
|---|---|---|---|---|
| $n$=3488, $k$=2720, $w$=64 | $\texttt{PI}=3, \texttt{L}=9, \texttt{Z}=0, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=422\,055\,936$ | $2^{44.9036}$ | $2^{-144.2566}$ | $2^{189.1603}$ |
| $n$=24 646, $k$=12 323, $w$=134 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=38\,958\,074\,884$ | $2^{56.2031}$ | $2^{-151.2846}$ | $2^{207.4877}$ |
| $n$=24 646, $k$=12 323, $w$=142 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=38\,958\,074\,884$ | $2^{56.2031}$ | $2^{-158.8411}$ | $2^{215.0442}$ |
| $n$=35 338, $k$=17 669, $w$=132 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=114\,861\,343\,744$ | $2^{58.7547}$ | $2^{-152.3526}$ | $2^{211.1073}$ |

### 4.2.3. Security Margin for each Sets of Problem Parameters

The problem parameters used for the experimental evaluation (Table 4.1) have well-defined theoretical security margins, determined by their respective reference security levels. In this section, the analysis focuses on comparing the theoretical security of each set of parameters with the actual values observed empirically, using the optimal configuration identified for each set as a reference. In the following sections, the individual security levels will be examined in detail, assessing the actual impact of the parallel architecture relative to the theoretical thresholds.

**Security Level L1** The first set of parameters analyzed corresponds to security level L1, which establishes the minimum threshold of computational robustness required by modern standards for PQC, associated with the complexity of an exhaustive attack against the AES-128 cipher [36] of $2^{143}$.

**Security Level L3** The second security level is L3, which is associated with the complexity of an exhaustive attack against the AES-192 [36], with a minimum cost of $2^{207}$ gates.

Table 4.7: Security margin for the sets of problem parameters that belong to security level L3

| Problem parameters | Attack configuration | Depth | Probability | Security margin |
|---|---|---|---|---|
| $n$=4608, $k$=3360, $w$=96 | $\texttt{PI}=3, \texttt{L}=9, \texttt{Z}=0, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=794\,563\,344$ | $2^{46.3878}$ | $2^{-186.9351}$ | $2^{233.3229}$ |
| $n$=49 318, $k$=24 659, $w$=199 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=8\,555\,510\,016$ | $2^{61.1128}$ | $2^{-218.8558}$ | $2^{279.9686}$ |
| $n$=49 318, $k$=24 659, $w$=206 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=8\,555\,510\,016$ | $2^{61.1128}$ | $2^{-225.5956}$ | $2^{286.7084}$ |
| $n$=71 702, $k$=35 851, $w$=200 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=7\,440\,097\,536$ | $2^{63.7793}$ | $2^{-222.8639}$ | $2^{286.6432}$ |

**Security Level L5** The last security level is L5, which is associated with the complexity of an exhaustive attack against the AES-256 [36], with a minimum cost of $2^{272}$ gates.

Table 4.9: Security margin for the sets of problem parameters that belong to security level L5

| Problem parameters | Attack configuration | Depth | Probability | Security margin |
|---|---|---|---|---|
| $n$=6688, $k$=5024, $w$=128 | $\texttt{PI}=3, \texttt{L}=5, \texttt{Z}=0, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=2\,645\,044\,900$ | $2^{48.9911}$ | $2^{-271.9112}$ | $2^{320.9023}$ |
| $n$=6960, $k$=5413, $w$=119 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=3\,298\,894\,096$ | $2^{49.3756}$ | $2^{-263.5601}$ | $2^{312.9358}$ |
| $n$=8192, $k$=6528, $w$=128 | $\texttt{PI}=3, \texttt{L}=9, \texttt{Z}=0, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=5\,811\,622\,756$ | $2^{50.5314}$ | $2^{-302.1358}$ | $2^{352.6673}$ |
| $n$=81 946, $k$=40 973, $w$=264 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=8\,579\,205\,376$ | $2^{64.7301}$ | $2^{-285.8055}$ | $2^{350.5356}$ |
| $n$=81 946, $k$=40 973, $w$=274 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=8\,579\,205\,376$ | $2^{64.7301}$ | $2^{-295.5292}$ | $2^{360.2593}$ |
| $n$=115 274, $k$=57 637, $w$=262 | $\texttt{PI}=3, \texttt{L}=10, \texttt{Z}=10, \texttt{QU}=8, \texttt{QF}=1024, \texttt{WI}=3, \texttt{FW}=1, \texttt{PF}=7\,774\,654\,276$ | $2^{67.1637}$ | $2^{-286.6337}$ | $2^{353.7974}$ |

**Execution Time for all sets of Problem Parameters** To provide a concrete assessment of the safety margins discussed above, the abstract metrics of circuit depth and

success probability can be translated into an estimate of the actual physical execution time required to carry out the attacks. Assuming a massively parallel hardware implementation, the expected total time complexity is determined by the ratio of the critical path depth of a single iteration to its success probability, multiplied by the propagation delay of a single gate, which depends on the technology used.

This analysis considers two distinct operational scenarios, representing different levels of hardware optimization: a conservative gate delay of $2^{-9}$ seconds and a more optimistic, faster scenario with a gate delay of $2^{-12}$ seconds. Given the astronomical scale of the resulting time intervals, execution times are normalized and expressed in millennia.

As illustrated in Figure 4.29, even under the most optimistic acceleration scenario guaranteed by the maximum value of `PF`, the time required to successfully break any of the parameter sets remains orders of magnitude greater than the age of the universe. This graphical representation confirms that, although the ISD parallel architecture achieves significant speedups in theory, the absolute security of the underlying post-quantum cryptographic systems remains completely intact in practice.

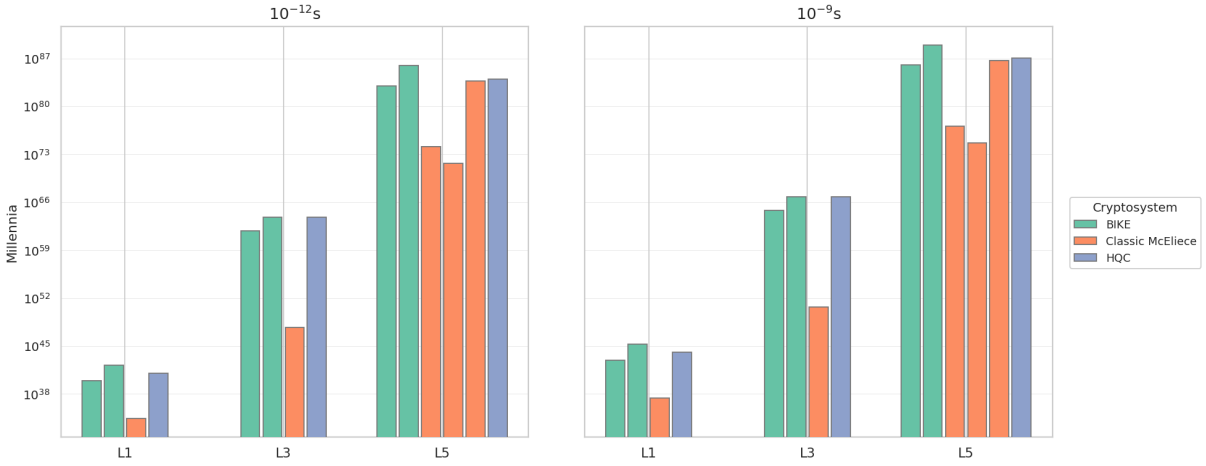


Figure 4.29: Representation of execution times in millennia for each security level, assuming a gate delay of $2^{-12}$s and $2^{-9}$s. Individual instances for each cryptosystem are plotted separately.

### 4.2.4. Theoretical Depth as `L` Varies

This subsection analyzes how the parameter `L` influences the overall theoretical depth of the circuit for a single iteration of the parallel attack. As will be shown, `L` plays a central role in balancing two competing factors: the size of the candidate sub-lists and the filtering efficiency of the subsequent verification stages.

**Analytical Model of Circuit Depth** To formally quantify this trade-off, the circuit depth is approximated analytically as a function of the code parameters $n$, $k$, $w$, the permutation weight $\mathtt{PI}$, and the value of $\mathtt{L}$. The total computational cost is decomposed into three macro-blocks, each capturing a distinct phase of the computation.

The first block accounts for the cost of reducing the parity-check matrix to partial systematic form. Its depth contribution is approximated as:

$$d_{systematic\ form} \approx 6\mathtt{L}^2\,(n-k) + 3n\left\lceil \log_2 \frac{k+\mathtt{L}}{2} \right\rceil \tag{4.4}$$

The second block models the parallel sorting network. In high-parallelization regimes, this phase dominates the total depth, making the list generation phase comparatively negligible. Its contribution is approximated as:

$$d_{sorting} \approx 14\left\lceil \log_2 \frac{k+\mathtt{L}}{2} \right\rceil^3 \left(\mathtt{L}+1+\mathtt{PI}\left\lceil \log_2 \frac{k+\mathtt{L}}{2} \right\rceil\right) \tag{4.5}$$

The third block covers the collision search and queue management phases:

$$d_{collision} \approx 6\mathtt{PI}\left\lceil \log_2 \frac{k+\mathtt{L}}{2} \right\rceil + 3\,(n-k-\mathtt{L}) + 3n\left\lceil \log_2 \frac{k+\mathtt{L}}{2} \right\rceil \tag{4.6}$$

Combining the three contributions, the total theoretical depth per iteration is:

$$depth \approx d_{systematic\ form} + \binom{(k+\mathtt{L})/2}{\mathtt{PI}}\,(d_{sorting} + d_{collision}) \tag{4.7}$$

The structure of Equation 4.7 reflects the natural separation between a fixed setup cost, captured by $d_{systematic\ form}$, and a combinatorial term that grows with the list size and multiplies the per-element operational cost of sorting and verification.

**Optimal Value of $\mathtt{L}$** To characterize the behavior of *depth* as a function of $\mathtt{L}$, it is useful to identify the theoretical optimum $\mathtt{L}_{\mathrm{opt}}$. This is defined as the value of $\mathtt{L}$ for which the expected number of collisions found during the collision search phase equals approximately 1:

$$\frac{2\binom{\frac{(k+\mathtt{L}+z)}{2}}{\mathtt{PI}}}{2^{\mathtt{L}}} \approx 1 \tag{4.8}$$

Since $\mathtt{Z} \ll k$, the parameter $\mathtt{Z}$ can be treated as negligible in this context. Solving Equation 4.8 for $\mathtt{L}$ yields the closed-form approximation:

$$\mathtt{L}_{optimal} \approx 1 + \mathtt{PI} \log_2 k - \mathtt{PI} - \log_2 \mathtt{PI}! \tag{4.9}$$

**Depth Profiles and Sensitivity Analysis** To evaluate how sensitive the circuit depth is to deviations from $\mathtt{L}_{\text{opt}}$, the depth profiles are analyzed by sampling values of $\mathtt{L}$ within a range of $\pm 20\%$ around $\mathtt{L}_{optimal}$, for $\mathtt{PI} \in \{1, 2, 3\}$. The resulting curves are shown in Figures 4.30–4.43.

As the graphs show, as $\mathtt{L}$ increases, the combinatorial size of the list increases, thereby also increasing the overall depth of the circuit. Conversely, choosing a value of $\mathtt{L}$ that is excessively lower than the optimal threshold, while reducing the depth of a single iteration, drastically reduces the overall probability of success of the attack, necessitating a greater number of external iterations required for a successful attack.

The analysis further reveals the effect of varying $\mathtt{PI}$ on the depth profiles. Increasing $\mathtt{PI}$ produces two concurrent effects: it raises the overall depth due to larger list sizes, and it shifts the optimal window for $\mathtt{L}$ toward higher values, as a larger search space must be accommodated. This interplay confirms that $\mathtt{L}_{optimal}$ is not an absolute constant, but rather a quantity that depends jointly on the parallelism level $\mathtt{PI}$ and the geometric parameters $n$, $k$, $w$ of the linear code under attack.

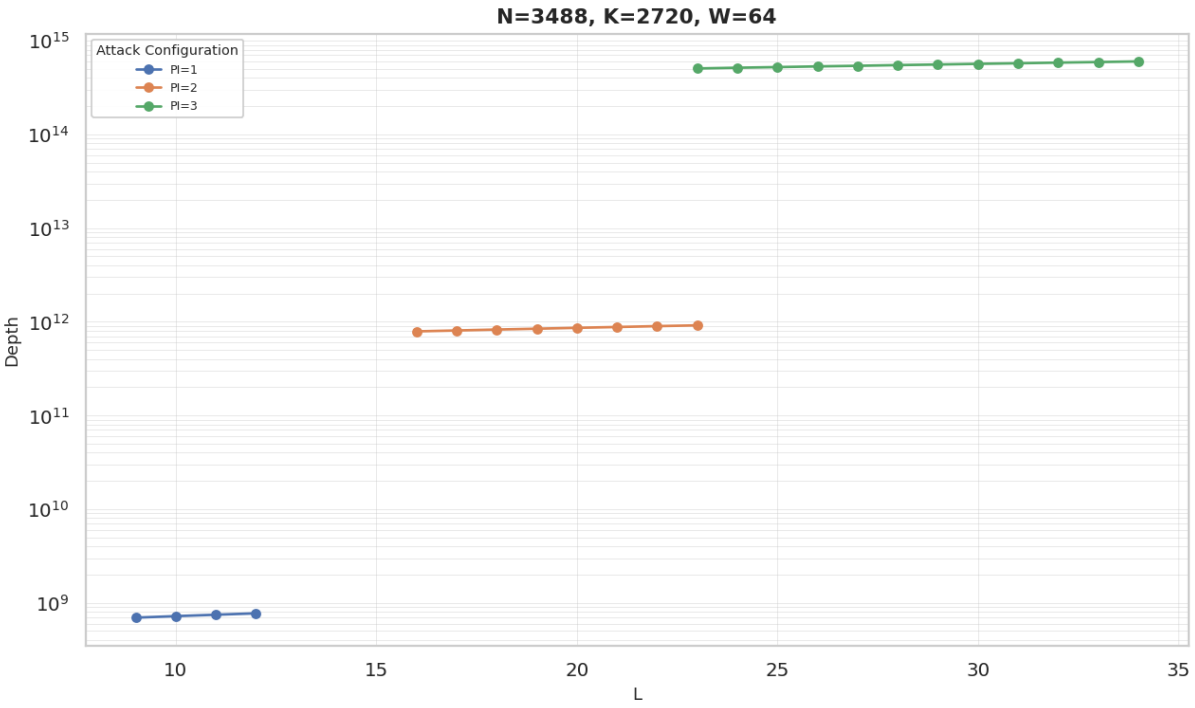


Figure 4.30: Theoretical depth as function of L, for $(n = 3488, k = 2720, w = 64)$

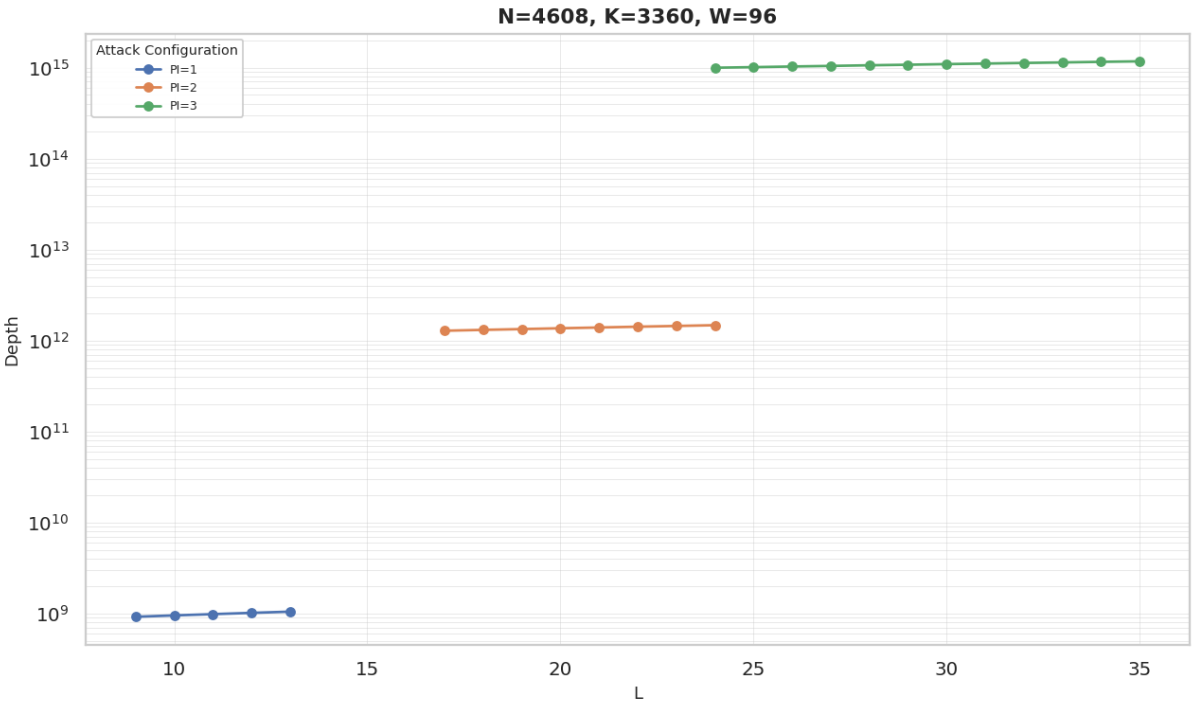


Figure 4.31: Theoretical depth as function of L, for $(n = 4608, k = 3360, w = 96)$

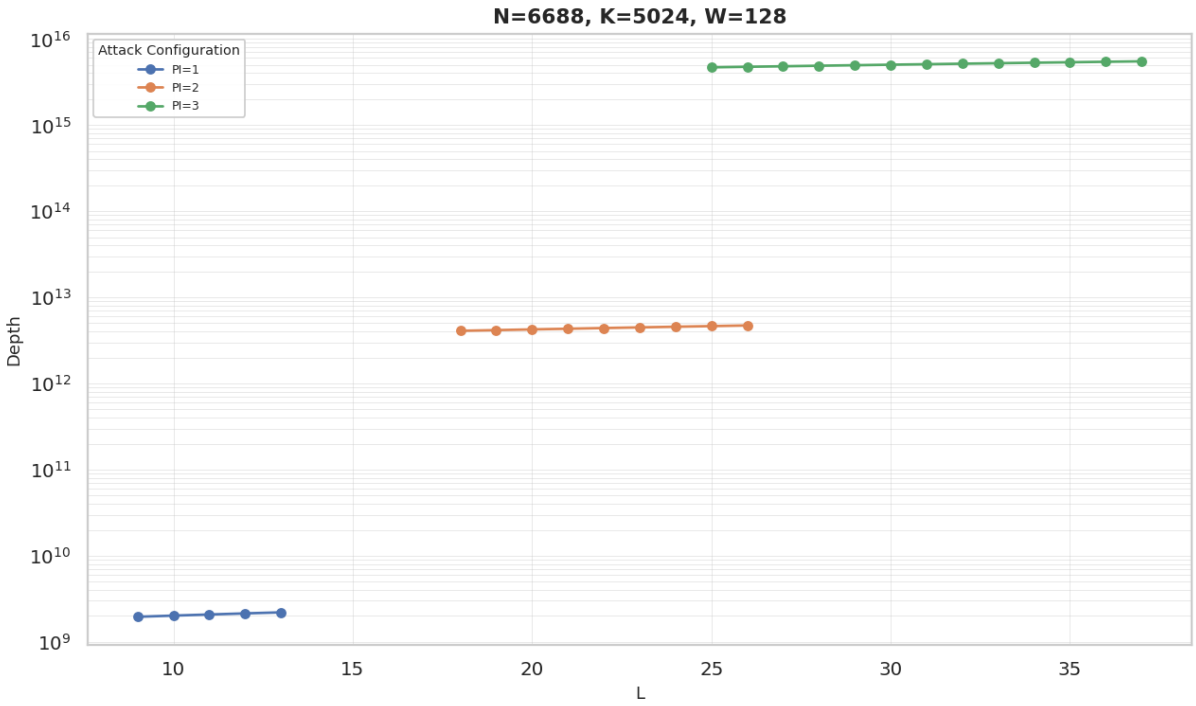


Figure 4.32: Theoretical depth as function of L, for $(n = 6688, k = 5024, w = 128)$

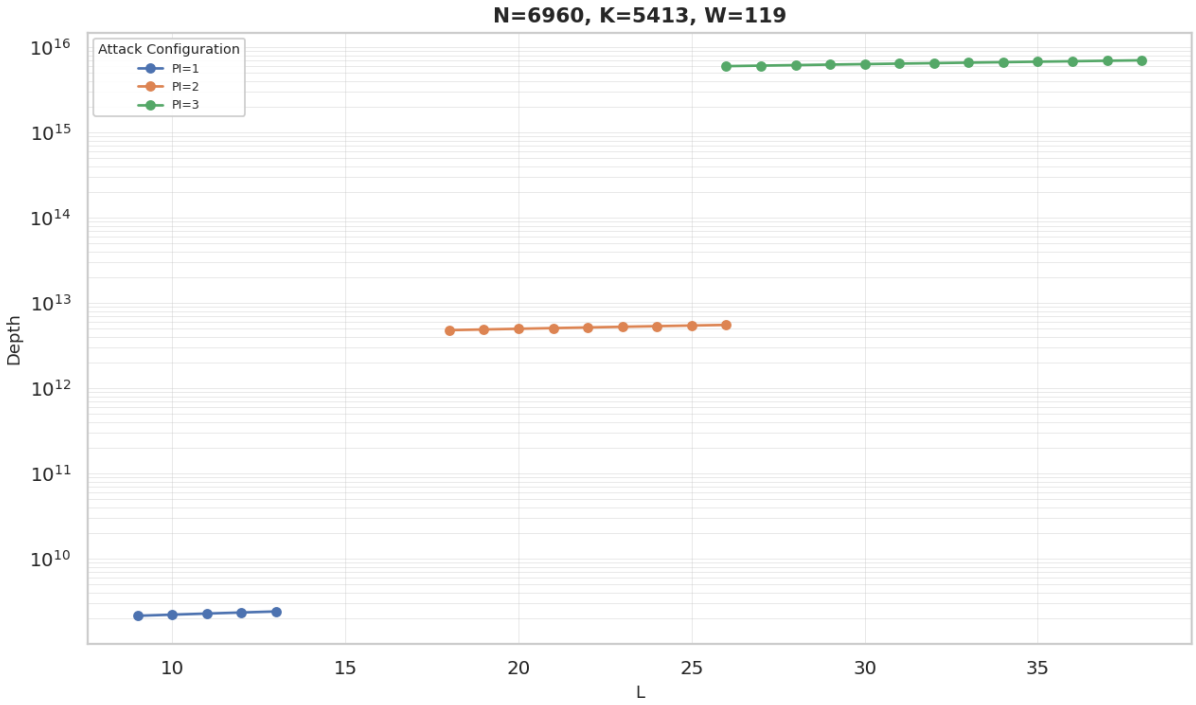


Figure 4.33: Theoretical depth as function of L, for $(n = 6960, k = 5413, w = 119)$

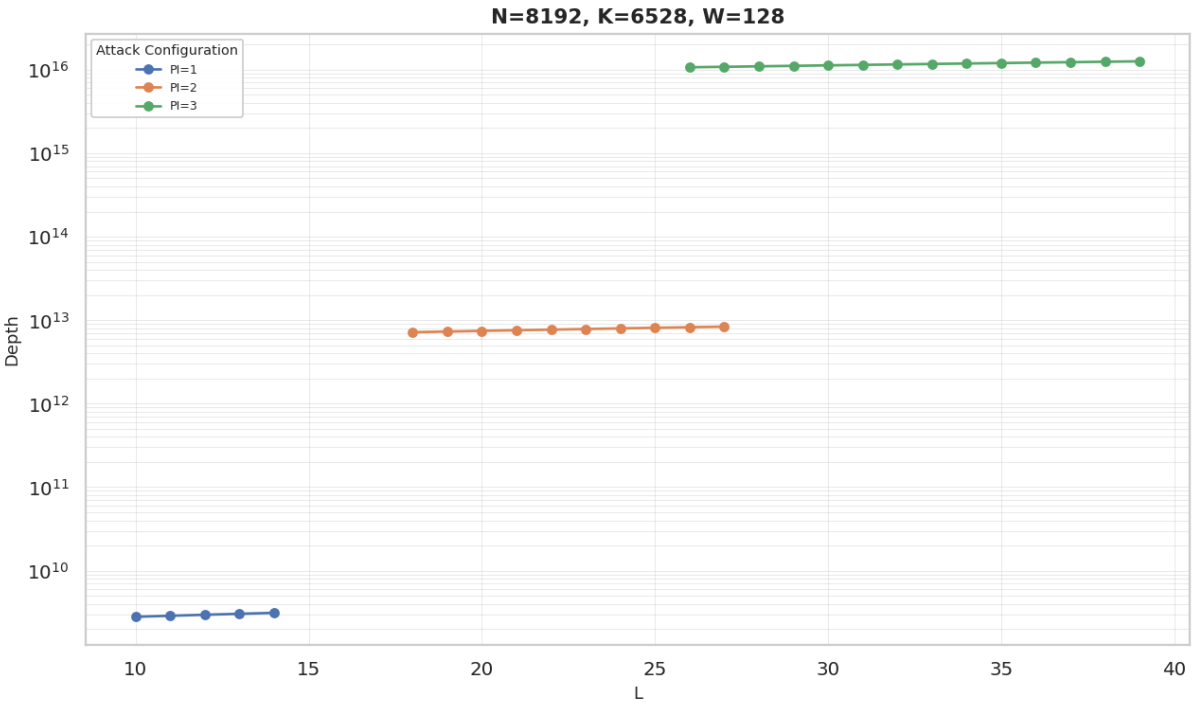


Figure 4.34: Theoretical depth as function of L, for $(n = 8192, k = 6528, w = 128)$

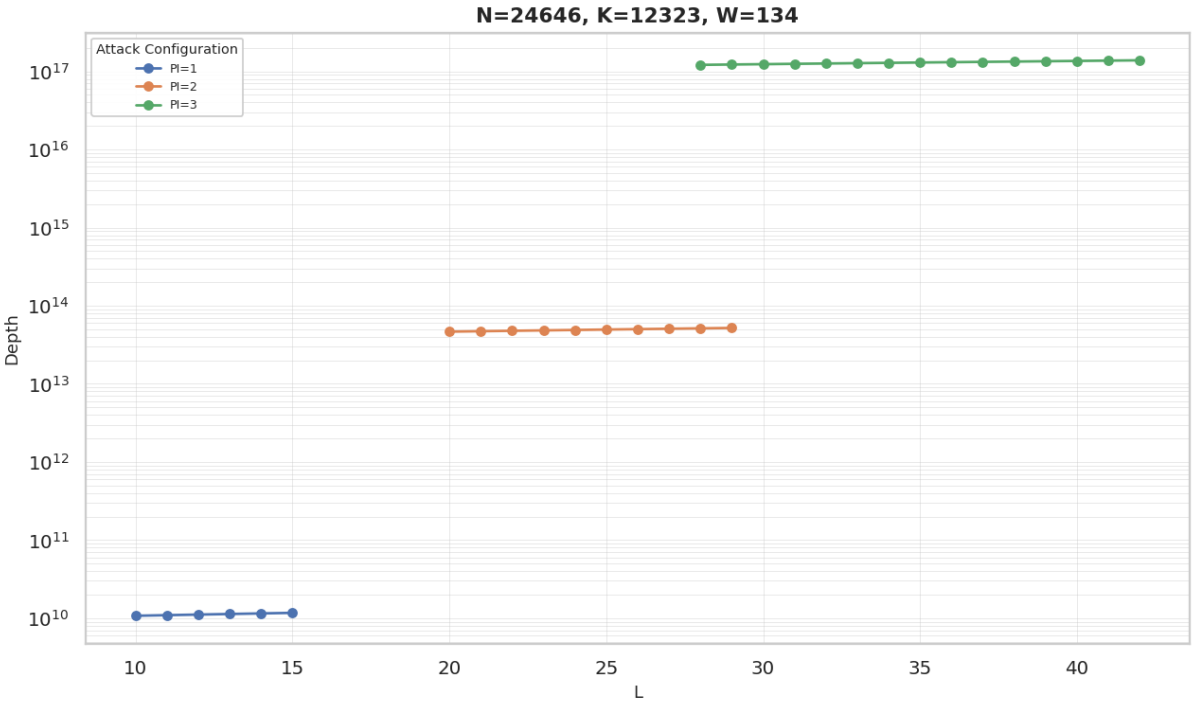


Figure 4.35: Theoretical depth as function of L, for $(n = 24\,646, k = 12\,323, w = 134)$

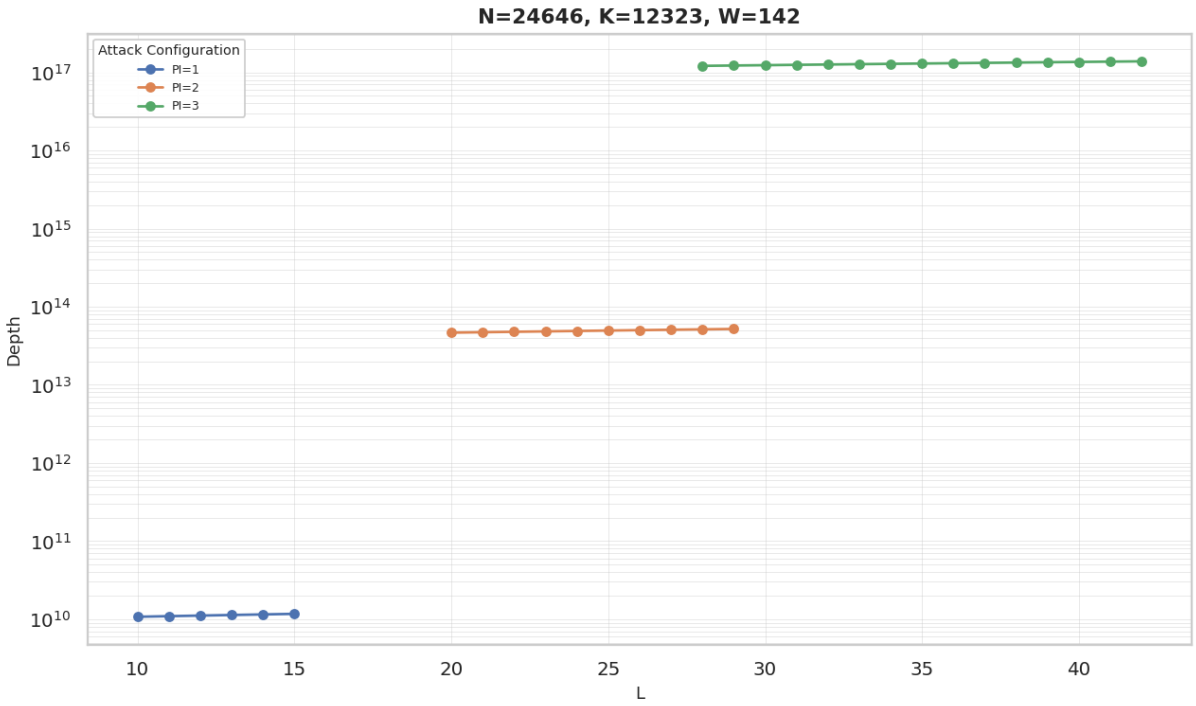


Figure 4.36: Theoretical depth as function of L, for $(n = 24\,646, k = 12\,323, w = 142)$

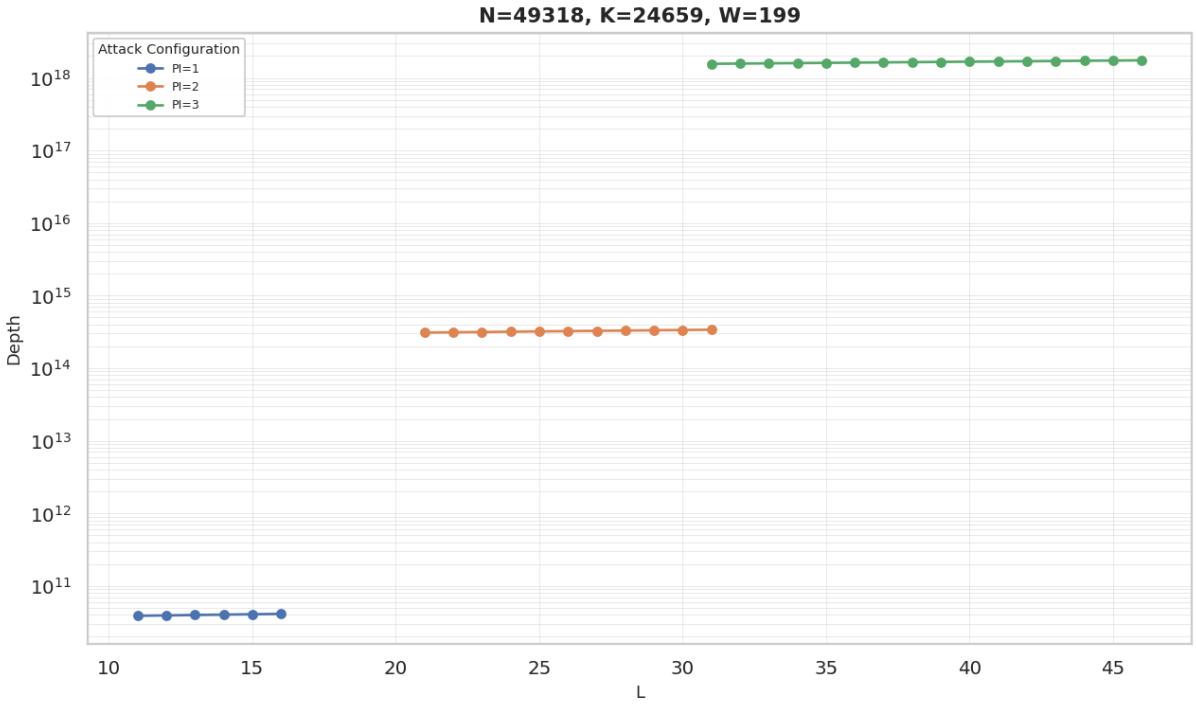


Figure 4.37: Theoretical depth as function of L, for $(n = 49\,318, k = 24\,659, w = 199)$

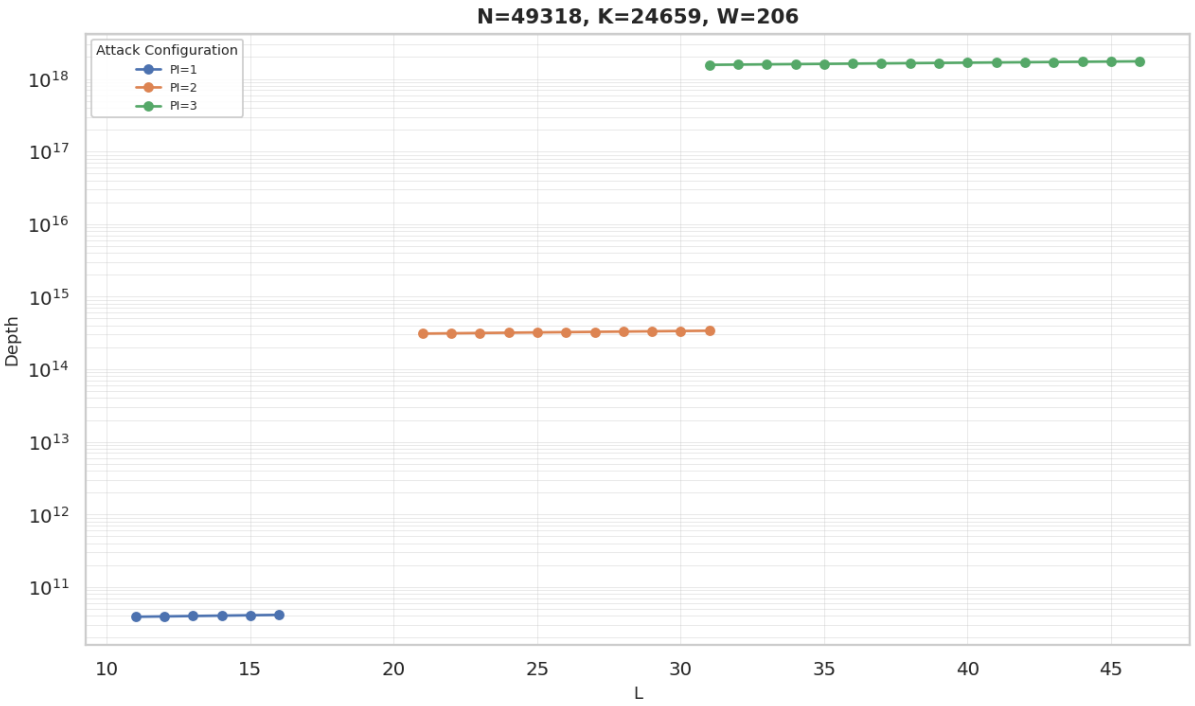


Figure 4.38: Theoretical depth as function of L, for $(n = 49\,318, k = 24\,659, w = 206)$

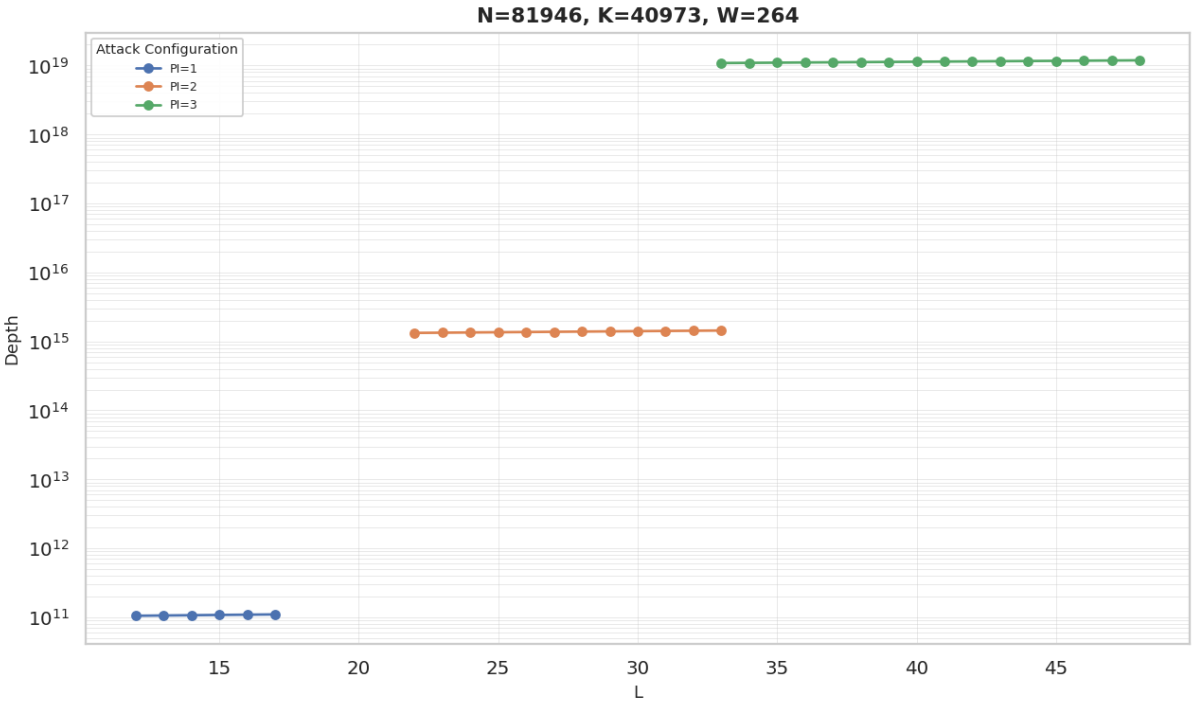


Figure 4.39: Theoretical depth as function of L, for $(n = 81\,946, k = 40\,973, w = 264)$

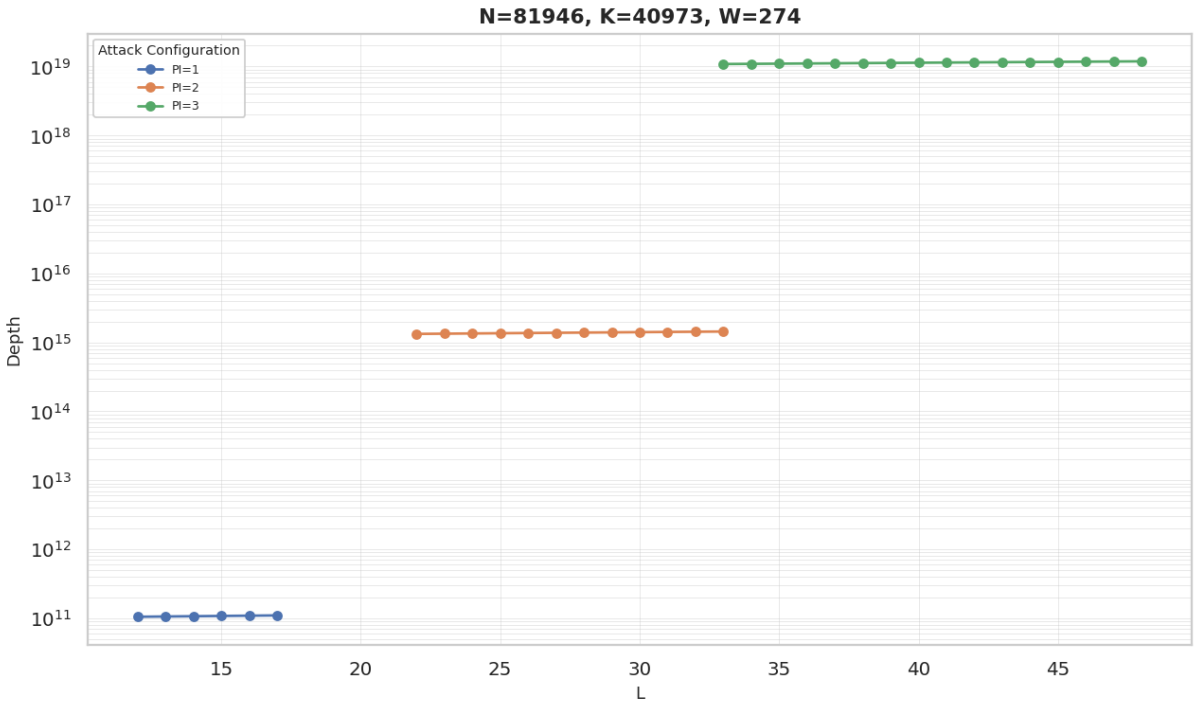


Figure 4.40: Theoretical depth as function of L, for $(n = 81\,946, k = 40\,973, w = 274)$

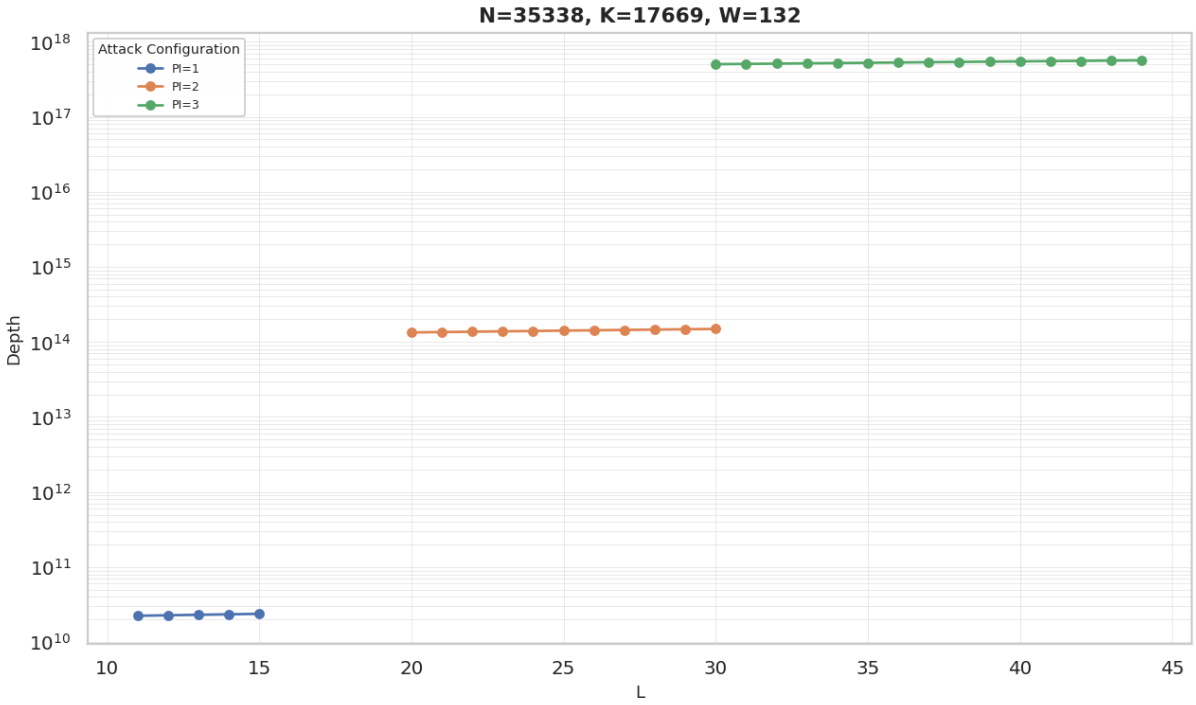


Figure 4.41: Theoretical depth as function of L, for $(n = 35\,338, k = 17\,669, w = 132)$

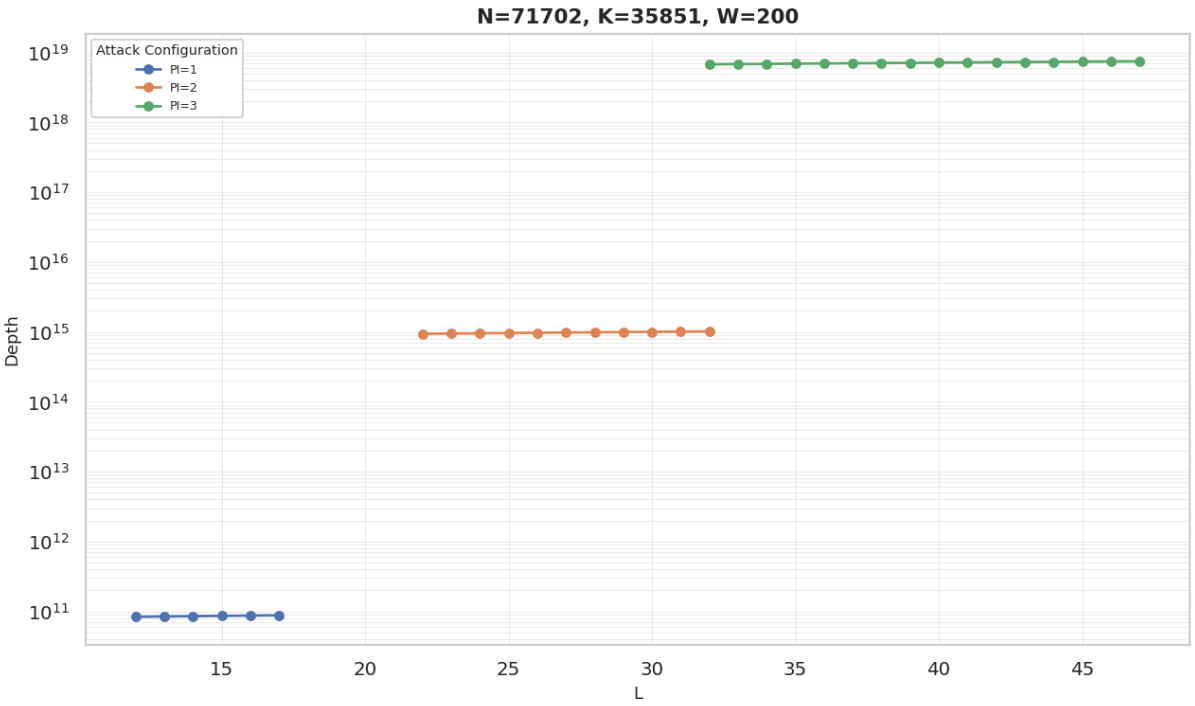


Figure 4.42: Theoretical depth as function of L, for $(n = 71\,702, k = 35\,851, w = 200)$

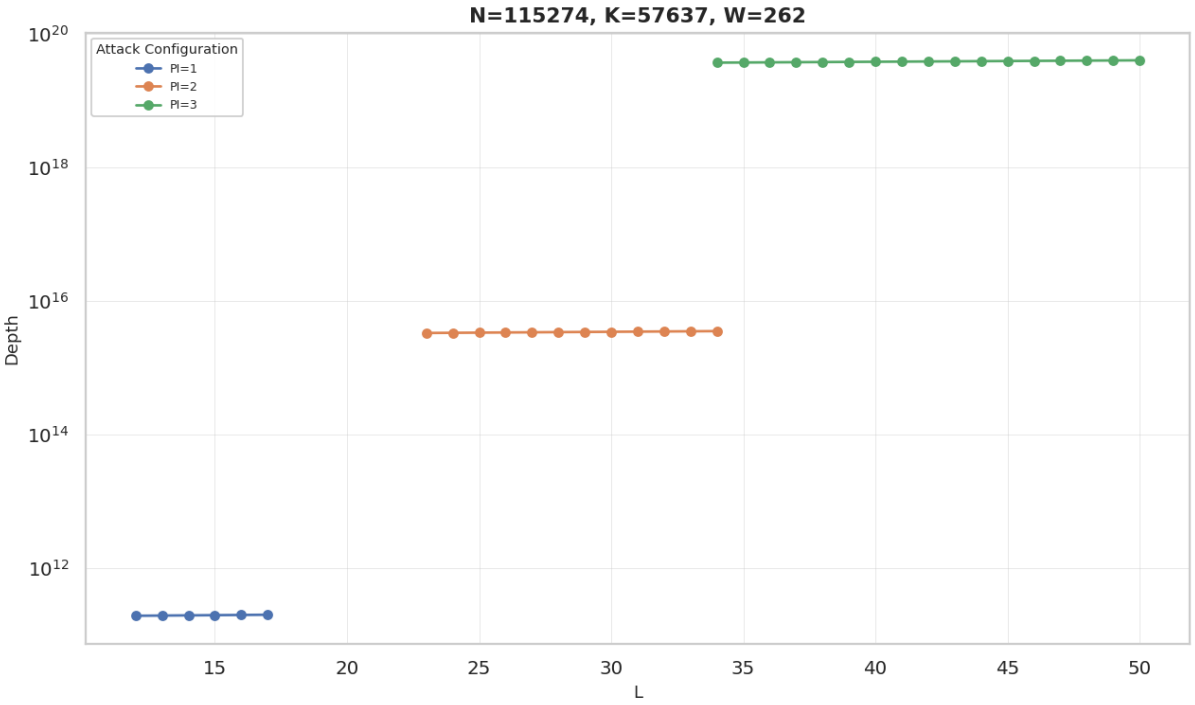


Figure 4.43: Theoretical depth as function of L, for $(n = 115\,274, k = 57\,637, w = 262)$

# 5 | Conclusion

This thesis addressed the problem of the parallel design of attack algorithms belonging to the ISD family, with the aim of evaluating whether, and to what extent, the introduction of hardware parallelism can reduce the execution times of attacks on code-based cryptography, a fundamental pillar of PQC.

The main contribution of this research was a systematic study of the impact of parallelism on existing ISD algorithms, identifying the most computationally expensive stages and evaluating the possibility of distributing the load across parallel architectures composed of multiple concurrent machines following the idea of [8]. The work focused on the analysis and extension of the CAT [9] framework, which was extended to integrate parallel paradigms, with particular attention to the design and implementation of the parallel version of the `isd1` algorithm, named `isd1_parallel`. To complete the integration of the work into the existing framework, the necessary support binaries were developed for generating parameters for the `isd1_parallel` run, orchestrating concurrent execution, and subsequently collecting experimental data.

The experimental campaign involved sets of candidate parameters (Table 4.1) for the fourth round of the NIST standardization process [2] and was conducted by comparing the original `isd1` algorithm with its parallel counterpart, `isd1_parallel`. The results showed that increasing parallelism leads to a *reduction in circuit depth*: as the parallelization factor, `PF`, increases, the critical path of the circuit is reduced, and thus the actual execution time of the attack is shortened compared to the serial counterpart. However, the speedup does not grow indefinitely with the addition of new machines; after an initial growth phase, the trajectory reaches a saturation plateau. This trend closely mirrors Amdahl's law [3], in which the inherently non-parallelizable phases of the attack (the column permutation phase and the post-processing phase) act as asymptotic bottlenecks.

Analysis of the area-time product confirmed the *typical trade-off in parallel systems*: time acceleration comes at the cost of an overall number of logic gates that grows more than linearly, due to the overhead introduced by global ordering networks. Therefore, reducing the attack's execution time is achievable only at the expense of increased hardware

complexity required to implement the circuit. In conclusion, the study demonstrates that, although a parallel adversary can drastically reduce the execution time of an ISD attack, the parameters proposed by the NIST maintain a robust computational security margin that complies with the requirements, since the complexity of the attack remains exponential even in the parallel circuit model.

From a future development perspective, a natural extension of this work involves applying the parallelization methodology and concurrent ordering constructs to the `isd2` variant implemented in CAT, which models two-level collision algorithms such as MMT [31] and BJMM [5]. A further line of research concerns the transfer of the parallelism paradigms investigated to a different computational model. The transition from the Boolean circuit model, adopted in this thesis, to the parallel RAM model, would allow one to determine whether the benefits and structural limitations observed are intrinsic to the ISD algorithms themselves, or whether they are partially linked to the topological and implementation constraints specific to the circuit model.

# Bibliography

[1] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Inria C. Bourges. Hamming Quasi-Cyclic (HQC). Official submission to the NIST Post-Quantum Cryptography Standardization Process, 2017. Available at `https://pqc-hqc.org/`.

[2] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization process. NIST Internal Report (NISTIR) 8413, National Institute of Standards and Technology, Gaithersburg, MD, jul 2022.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[4] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. BIKE: Bit flipping key encapsulation. Official submission to the NIST Post-Quantum Cryptography Standardization Process, 2017. Available at `https://bikesuite.org`.

[5] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding random binary linear codes in 2 n/20: How 1 + 1 = 0 improves information set decoding. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, Lecture Notes in Computer Science, pages 520–536. Springer, 2012.

[6] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. On the in-

herent intractability of certain coding problems (corresp.). *IEEE Trans. Inf. Theory*, 24(3):384–386, 1978.

[7] Daniel Bernstein. Chacha, a variant of salsa20, 01 2008.

[8] Daniel J. Bernstein. Understanding brute force, 2005.

[9] Daniel J. Bernstein and Tung Chou. Cryptattacktester: high-assurance attack analysis. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part VI*, Lecture Notes in Computer Science, pages 141–182. Springer, 2024.

[10] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, Lecture Notes in Computer Science, pages 31–46. Springer, 2008.

[11] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, Lecture Notes in Computer Science, pages 31–46. Springer, 2008.

[12] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Smaller decoding exponents: Ball-collision decoding. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, Lecture Notes in Computer Science, pages 743–760. Springer, 2011.

[13] André Chailloux, María Naya-Plasencia, and André Schrottenloher. An efficient quantum collision search algorithm and implications on symmetric cryptography. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 211–240, Heidelberg, Germany, 2017. Springer.

[14] Classic McEliece Team. Classic McEliece: Conservative code-based cryptography: Cryptosystem specification. Technical report, NIST Post-Quantum Cryptography Standardization, October 2022. Round 4 Submission.

[15] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[16] Léo Ducas, Maxime Plançon, and Benjamin Wesolowski. On the shortness of vectors to be found by the ideal-SVP quantum algorithm. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 10625 of *Lecture Notes in Computer Science*, pages 322–351, Heidelberg, Germany, 2019. Springer.

[17] Ilya Dumer. Two decoding algorithms for linear codes. *Problems of Information Transmission*, 25, 03 1989.

[18] Andre Esser and Alexander May. Better sample – random subset sum in $2^{0.255n}$ and its impact on decoding random linear codes, 2019. Withdrawn.

[19] Andre Esser, Alexander May, and Floyd Zweydinger. Mceliece needs a break - solving mceliece-1284 and quasi-cyclic-2918 with modern ISD. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part III*, Lecture Notes in Computer Science, pages 433–457. Springer, 2022.

[20] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[21] Jr. George C. Clark and J. Bibb Cain. Error-correction coding for digital communications, 1982. 2nd printing.

[22] Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 235–256, Heidelberg, Germany, 2010. Springer.

[23] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[24] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[25] Manfred Kunde. Lower bounds for sorting on mesh-connected architectures. *Acta Informatica*, 24(2):121–130, 1987.

[26] Hans-Werner Lang, Manfred Schimmler, Hartmut Schmeck, and Heiko Schröder.

Systolic sorting on a mesh-connected network. *IEEE Trans. Computers*, 34(7):652–658, 1985.

[27] Pil Joong Lee and Ernest F. Brickell. An observation on the security of mceliece's public-key cryptosystem. In Christoph G. Günther, editor, *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings*, Lecture Notes in Computer Science, pages 275–280. Springer, 1988.

[28] Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Trans. Inf. Theory*, 34(5):1354–1359, 1988.

[29] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*, volume 16 of *North-Holland Mathematical Library*. North-Holland, 1977.

[30] John M. Marberg and Eli Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3:561–572, 1988.

[31] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, Lecture Notes in Computer Science, pages 107–124. Springer, 2011.

[32] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. Technical Report 44, Jet Propulsion Laboratory, 1978.

[33] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology - CRYPTO '85, Santa Barbara, California, USA, August 18-22, 1985, Proceedings*, Lecture Notes in Computer Science, pages 417–426. Springer, 1985.

[34] Harald Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):157–166, 1986.

[35] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, Lecture Notes in Computer Science, pages 617–630. Springer, 2003.

[36] National Institute of Standards, Technology (NIST), Morris J. Dworkin, Meltem Son-

mez Turan, and Nicky Mouha. Advanced encryption standard (aes), 2023-05-09 04:05:00 2023.

[37] Nicholas J. Patterson. The algebraic decoding of goppa codes. *IEEE Trans. Inf. Theory*, 21(2):203–207, 1975.

[38] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Trans. Inf. Theory*, 8(5):5–9, 1962.

[39] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983.

[40] Isaac D. Scherson and Sandeep Sen. Parallel sorting in two-dimensional VLSI models of computation. *IEEE Trans. Computers*, 38(2):238–249, 1989.

[41] Manfred Schimmler. Fast sorting on the instruction systolic array. Bericht 8709, Institut für Informatik, Christian-Albrechts-Universität Kiel, 1987.

[42] Claus P. Schnorr and Hendrik W. Lenstra, Jr. A Monte Carlo factoring algorithm with linear storage. *Mathematics of Computation*, 43(167):289–311, 1984.

[43] Claus-Peter Schnorr and Adi Shamir. An optimal sorting algorithm for mesh connected computers. In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 255–263. ACM, 1986.

[44] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, November 20-22, 1994*, pages 124–134. IEEE Computer Society, 1994.

[45] V. M. Sidelnikov and S. O. Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Mathematics and Applications*, 2(4):439–444, 1992.

[46] Jacques Stern. A method for finding codewords of small weight. In Gérard D. Cohen and Jacques Wolfmann, editors, *Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings*, Lecture Notes in Computer Science, pages 106–113. Springer, 1988.

# List of Figures

# List of Tables

# List of Algorithms

# List of Acronyms

**AES** Advanced Encryption Standard. 7, 22, 24–26, 71, 81, 85, 106, 107

**ASIC** Application-Specific Integrated Circuit. 23, 41

**BJMM** Becker-Joux-May-Meurer. 2, 45, 61, 62, 65, 69, 120

**CAS** Compare-and-Swap. 9, 10, 25, 58

**CAT** CryptAttackTester. 2, 5, 39–43, 45–51, 56, 60, 62, 63, 69, 71–75, 77, 81, 82, 85, 119, 120, 129, 131

**CPU** Central Processing Unit. 41

**ECC** Elliptic Curve Cryptography. 1, 8, 27, 28

**GDP** General Decoding Problem. 1, 28, 34, 35, 37, 38, 45

**GPP** General-Purpose Processor. 41

**IS** Information Set. 34, 45, 46, 48, 49, 52, 55–57, 60, 61, 65, 66, 76, 85

**ISD** Information Set Decoding. 2, 5, 34, 35, 38, 42, 43, 45–48, 50, 51, 53, 55, 56, 60, 69, 71, 72, 75, 85, 97, 109, 119, 120, 129, 131

**MMT** May-Meurer-Thomae. 2, 45, 61–63, 69, 120

**NIST** National Institute of Standards and Technology. 1, 2, 28, 36, 81, 85, 86, 88, 119, 120, 129

**OETS** Odd-Even Transposition Sort. 10–12, 15, 18–22

**OW-CPA** One-Wayness under Chosen Plaintext Attack. 43

**PQC** Post-Quantum Cryptography. 1, 27, 28, 34, 81, 85, 106, 119

**RAM** Random-Access Memory. 40, 41, 58, 120

**RS** Redundancy Set. 34, 45, 46, 66

**RSA** Rivest, Shamir, Adleman. 1, 8, 27, 28

**SDP** Syndrome Decoding Problem. 1, 2, 28, 33, 34, 37, 38, 42, 43, 45, 67, 85

# List of Symbols

| Symbol | Description |
|---|---|
| $\mathbf{K}$ | The set of keys |
| $\mathbf{M}$ | The set of plaintext |
| $\mathbf{C}$ | The set of ciphertext |
| $\mathbf{I}$ | An Information Set |
| $\mathbf{J}$ | A Redundancy Set |
| $k_{private}$ | The private-key of a public-key cryptosystem |
| $k_{public}$ | The public-key of a public-key cryptosystem |
| $\mu_i$ | The mean at the $i$-th iteration according to Welford's algorithm |
| $m_{2_i}$ | The sum of squared deviations at the $i$-th iteration according to Welford's algorithm |
| $n$ | The length of a linear code (the number of transmitted symbols) |
| $k$ | The dimension of a linear code (the number of pure information symbols) |
| $w$ | The Hamming weight of a target error vector |
| $\mathcal{B}$ | A random seed used in the parallel machine designed by Bernstein in [8] |
| $\mathbf{c}$ | A codeword |
| $\mathbf{e}$ | An error vector |
| $\mathbf{s}$ | A syndrome |
| $\mathbf{0}$ | A zero vector |
| $\widehat{\mathbf{s}}$ | The permuted syndrome |
| $\widehat{\mathbf{s}_{\mathbf{up}}}$ | The first $\mathtt{L}$ positions of the permuted syndrome $\widehat{\mathbf{s}}$ |
| $\widehat{\mathbf{s}_{\mathbf{down}}}$ | The last $n-k-\mathtt{L}$ positions of the permuted syndrome $\widehat{\mathbf{s}}$ |
| $\widehat{\mathbf{e}}$ | The permuted error |
| $\widehat{\mathbf{e}_{\mathbf{s}}}$ | The part of the permuted error indexed by Information Set |
| $\widehat{\mathbf{e}_{\mathbf{s}^*}}$ | The part of the permuted error indexed by Redundancy Set |

| Symbol | Description |
|---|---|
| $\mathrm{d_h}(\mathbf{x}, \mathbf{y})$ | The Hamming distance between the vector $\mathbf{x}$ and $\mathbf{y}$ (the number of positions at which the corresponding symbols are different) |
| $\mathrm{HW}(\mathbf{x})$ | The Hamming weight of the vector $\mathbf{x}$ (the number of elements different from zero) |
| $d(\mathcal{C}[n,k])$ | The minimum distance of a linear code $\mathcal{C}[n,k]$ |
| $\mathcal{C}[n,k]$ | The linear code of length $n$ and dimension $k$ |
| $\mathcal{C}^{\perp}[n,n-k]$ | The dual code of a linear code $\mathcal{C}[n,k]$ |
| $I_m$ | An identity matrix of size $m$ |
| $G$ | A generator matrix |
| $\widehat{G}$ | A permuted generator matrix ($\widehat{G} = SGP$) |
| $H$ | A parity-check matrix |
| $\widehat{H}$ | The permuted parity-check matrix ($\widehat{H} = SHP$) |
| $V$ | The dense part of a parity-check matrix in systematic form |
| $V_{UP}$ | The first $\mathtt{L}$ rows of the dense matrix $V$ |
| $V_{DOWN}$ | The last $n-k-\mathtt{L}$ rows of the dense matrix $V$ |
| $S$ | A random invertible dense scrambling matrix |
| $P$ | A random permutation matrix |
| $C(n)$ | The number of comparators required to sort a sequence of $n$ elements |
| $T(n)$ | The number of steps required to sort a sequence of $n$ elements, the depth of the sorting network |

# Acknowledgements

I would like to express my sincere gratitude to my advisor, Professor Alessandro Barenghi, for proposing this stimulating research project and for his continuous support throughout the process, offering expert guidance combined with great professionalism. I would also like to extend my heartfelt thanks to Professor Gerardo Pelosi, co-advisor of this thesis, for his constant availability, invaluable support, and the crucial insights he shared throughout this work. Finally, special thanks go to Dr. Simone Perriello for his essential practical assistance, patience, and reliability over the past few months, as well as for the fruitful discussions that helped address the challenges encountered during this research.