Modification-Tolerant Signature Schemes using Combinatorial Group Testing: Theory, Algorithms, and Implementation

Dongxia Luo

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science Mathematics and Statistics*

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

© Dongxia Luo, Ottawa, Canada, 2024

*The M.Sc. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

Matrices that are d-cover-free are used in non-adaptive combinatorial group testing (CGT). They allow for locating up to d defectives within a set of n items by testing them in groups. In non-adaptive group testing, all tests are determined before any are conducted. A group test yields a negative result if and only if all items in the group are non-defective. In this thesis, we study d-cover-free families (CFFs) built from set systems and codes. Examples of these include Sperner sets, Steiner Triple Systems, and Reed-Solomon codes. Our primary focus is on decoding algorithms of CGT, which are used to accurately identify all defectives based on the test results. We introduce a general decoding algorithm that relies solely on the d-CFF property of the matrix, as well as specialized algorithms designed according to the specific construction of d-CFFs. Additionally, we conduct experiments to evaluate the performance of each decoding method and compare the general and specialized algorithms, particularly in terms of their efficiency as n and d grow.

This study also explores the application of non-adaptive combinatorial group testing using d-CFF to digital signatures. Digital signatures are mathematical schemes designed to ensure data integrity and authenticity of digital communications. They verify the authenticity of a signature and ensure that the signed document has not been tampered with. These schemes include classical methods like RSA-based and ECDSA, as well as post-quantum approaches such as FALCON and SPHINCS$^+$. However, they lack the ability to identify the location of changes if a document has been modified. By incorporating d-CFFs into these digital signatures, we can not only verify the authenticity of the signature, but also locate up to d modifications, leading to what is known as d-modification tolerant signature schemes (MTSS).

Our work advances the study of modification-tolerant signature schemes (MTSS) by practically implementing d-MTSS for various document types. Key achievements include developing efficient data structures and decoding algorithms for different constuctions of d-CFFs. We also conduct extensive testing of d-MTSS across multiple scenarios, and our results demonstrate its viability where document modifications are necessary. This research paves the way for future enhancements in data security and digital signature methods.

Dedication

To my father:

愿爸爸平安健康。

Acknowledgement

This thesis represents a journey made possible by the support of many. I would like to express my deepest gratitude to:

My supervisor, Dr. Lucia Moura, who has been a guiding light throughout these two years. This work would not have been possible without her unwavering support, thoughtful guidance, and encouragement. I am truly fortunate to have had her by my side.

My thesis committee, Dr. Daniel Panario and Dr. Monica Nevins, for their careful reading, valuable feedback, and insightful suggestions, all of which were essential in refining and shaping this work.

My professors, Dr. Anne Broadbent, Dr. Mateja Šajna, and Dr. Brett Stevens, whose courses sparked my passion for mathematics. Their teaching inspired me to dive deeper into the subject, fueling my desire to explore and contribute to this field.

My friends, Daniel, Masoomeh, and Prangya, who have made me feel incredibly fortunate throughout this academic journey. Their companionship and encouragement have brought strength and joy during challenging times.

My parents and sister, who, despite the long distance, have provided unconditional support and constantly wish for my well-being. Their love is an unbreakable bond and a constant source of strength, giving me the courage to pursue all my dreams.

Contents

List of Figures **viii**

List of Tables **x**

1 Introduction **1**

2 Combinatorial Group Testing and Cover-Free Families **4**

2.1 Combinatorial Group Testing 4

2.2 Cover-Free Families and Constructions 5

2.2.1 $d = 1$: Sperner Construction 7

2.2.2 $d = 2$: Steiner Triple System 7

2.2.3 Any $d \geq 2$: Reed-Solomon Codes 13

3 Data Structures and Fast Decoding Algorithms **16**

3.1 General Group Testing Decoding Algorithm 16

3.1.1 Matrix Representation Using Set Arrays to Encode Positions of 1s 17

3.1.2 Matrix Representation with Integer-Based Bit Arrays 20

3.2 Three Specific Group Testing Decoding Algorithms 21

3.2.1 Sperner Decoding Algorithm 22

3.2.2 STS Decoding Algorithm 23

3.2.3 Reed-Solomon Decoding Algorithm 26

3.3 Performance and Comparison 34

3.3.1 Revisiting the Theoretical Analysis of Decoding Algorithms . . . 34

3.3.2 General versus Sperner Decoding Algorithm 35

3.3.3 General versus STS Decoding Algorithm 36

3.3.4 General versus Reed-Solomon Decoding Algorithm 37

3.3.5 Performance of RSdecode 38

3.3.6 STS vs Reed-Solomon Decoding Algorithm when $d = 2$ 41

4 Digital Signatures 43

4.1 Conventional Digital Signature Schemes 43

4.1.1 Cryptographic Hash Functions 44

4.1.2 Classical Digital Signatures 46

4.1.3 Post-quantum Digital Signatures 48

4.2 Modification-Tolerant Signature Scheme 50

5 Implementation of MTSS 54

5.1 Message Division 54

5.1.1 Text Files 55

5.1.2 Image Files 55

5.2 Cryptography Library and Primitives 56

5.3 Determination of Number of Tests 57

5.4 MICO: a Revised d-MTSS Scheme 59

5.5 Illustration of Locating Modifications in an Image File 61

6 Experiments with the MICO Scheme 64

6.1 Experimental Approach 64

6.2 Performance of Underlying Choices 67

6.2.1 Performance for Different Choices of CDSS 67

6.2.2 Performance of Hash Algorithms 68

6.3 Analysis of MICO Signature Size 69

6.3.1 Analyzing MICO Signature Size from Direct Calculations 69

6.3.2 Analyzing MICO Signature Size Through Experiments 70

6.4 Efficiency Analysis of the MICO scheme 72

6.4.1 Evaluating the Impact of n and d on Signing Time 72

6.4.2 Impact of Other Parameter Changes on Signing Time 79

6.4.3 Evaluating the Impact of n and d on Verification Time 80

6.4.4 Impact of Other Parameter Changes on Verifying Time 86

7 Conclusion and Future Work 88

7.1 Conclusion 88

7.2 Future Work 89

Bibliography **94**

List of Figures

2.1 Latin square of order 5. 8
2.2 A symmetric idempotent Latin square of order 3. 10
2.3 A symmetric half-idempotent Latin square of order 2. 12

5.1 Tabaret Hall at the University of Ottawa, from [43]. 61
5.2 Image divided into blocks of 80×80 squares. 62
5.3 Image with modified blocks $\{1,41\}$. 62
5.4 Image with modified blocks $\{1,41\}$ changed to white. 63

6.1 SHA2/3-256: Comparison of MICO signature size for different CDSS. . . . 71
6.2 SHA2/3-512: Comparison of MICO signature size for different CDSS. . . . 72

List of Tables

2.1 An example of combinatorial group testing when $n = 12$, $t = 9$, and $d = 2$. 5
2.2 A (3, 9, 3) Reed-Solomon code over $\mathbb{F}_3$, where each row is a codeword. . . 15

3.1 Theoretical analysis of decoding algorithms. 34
3.2 Comparison among three decoding methods using Sperner when $d = 1$. . . 35
3.3 Comparison among three decoding methods using STS when $d = 2$. 36
3.4 Comparison among three decoding methods with different d. 37
3.5 Performance of three methods using parameters from Theorem 2.2.15. . . 38
3.6 Performance of RSDECODE using parameters from Theorem 2.2.15. 39
3.7 Performance of RSDECODE with different parameters. 40
3.8 STS vs RS in matrix format when $d = 2$. 41
3.9 STSDECODE vs RSDECODE when $d = 2$. 42

4.1 RSA versus ECDSA. 48
4.2 Variants of Dilithium. 50

5.1 Parameters of digital signature schemes used in our implementation. . . . 57

6.1 Files for testing correctness. 65
6.2 Text files for testing efficiency. 65
6.3 Images for testing efficiency. 66
6.4 Ten set-ups for testing correctness. 66
6.5 Running time of five CDSS for different message sizes. 67
6.6 Running time of hash algorithms with different message sizes. 68
6.7 Comparison of MICO signature size using direct calculation. 70
6.8 Comparison of MICO signature size when $t = 157$. 70
6.9 Comparison of MICO signature size when $t = 1{,}417$. 71

6.10 Set-ups for Sperner and $d = 1$. 73
6.11 Signing performance for different text files using Sperner set systems. . . . 73
6.12 Signing performance for different image files using Sperner set systems. . . 73
6.13 Set-ups for STS and $d = 2$. 74
6.14 Signing performance for different text files using STS. 74
6.15 Signing performance for different image files using STS. 74
6.16 Set-ups for Reed Solomon codes (RS codes). 75
6.17 Signing performance for Text_SM_40000 using RS codes. 75
6.18 Signing performance for Text_MED_200000 using RS codes. 76
6.19 Signing performance for Text_LAR_1000000 using RS codes. 76
6.20 Signing performance for Image_SM_200_200 using RS codes. 77
6.21 Signing performance for Image_MED_500_500 using RS codes. 77
6.22 Signing performance for Image_LAR_1000_1000 using RS codes. 77
6.23 Breakdown for signing time of Text_SM_40000 when $n = 4,000$. 78
6.24 Breakdown for signing time of Image_LAR_1000_1000 when $d = 8$. . . . 78
6.25 Signing performance for Text_SM_40000 when $d = 8$ and $n = 4000$. . . . 79
6.26 Signing performance for Image_LAR_1000_1000 when $d = 8$ and $n = 250000$. 80
6.27 Verification performance for different text files using Sperner set systems. 81
6.28 Verification performance for different image files using Sperner set systems. 81
6.29 Verification performance for different text files using STS. 82
6.30 Verification performance for different image files using STS. 82
6.31 Verification performance for Text_SM_40000 using RS codes. 83
6.32 Verification performance for Text_MED_200000 with RS codes. 83
6.33 Verification performance for Text_LAR_1000000 with RS codes. 83
6.34 Verification performance for Image_SM_200_200 using RS codes. 84
6.35 Verification performance for Image_MED_500_500 using RS codes. . . . 84
6.36 Verification performance for Image_LAR_1000_1000 using RS codes. . . 84
6.37 Breakdown of verification performance for Text_SM_40000 when $n = 4000$. 85
6.38 Breakdown of verification performance for Image_LAR_1000_1000 when $d = 8$. 85
6.39 Verification time for Text_SM_40000 when $d = 8$ and $n = 4,000$. 86
6.40 Verification time for Image_LAR_1000_1000 when $d = 8$ and $n = 250,000$. 87

Chapter 1

Introduction

In modern digital communications, digital signatures play an important role in guaranteeing the integrity and authenticity of messages and documents. These signature schemes allow a signer to generate a key pair containing a private key and a public key. Every time the signer signs a message or document with their private key, any party who has access to the corresponding public key can verify if the content received was signed by the owner of the private key and has not been modified during transit. The verification step outputs a boolean result: a result of 1 indicates that the signed message or document has not been changed and that it was signed by the owner of the public key, while a result of 0 indicates otherwise. In our thesis, this type of signature is called *conventional* digital signature. However, there are scenarios where some modifications to a signed document may be necessary or inevitable, such as collaborative editing or data redactions before transmission. This is where a *modification-tolerant* signature scheme (MTSS) comes into play.

The mechanisms behind MTSS were first introduced by Idalino et al. [21] in 2015, and Idalino et al.[20] refined a general framework for MTSS and provided a proof of its security in 2019. Unlike conventional digital signatures, MTSS not only ensures data authenticity but also tolerate a specified number of modifications in a document or message. Moreover, MTSS can identify the locations of these modifications and, in some cases, even correct them [20]. In this thesis, we focus on MTSS that specifically locates modifications and not the ones that correct them. Generally speaking, the scheme divides a message or document into n blocks, which are then tested in t different groups to determine which blocks have been modified. This process can be effectively achieved using combinatorial group testing (CGT).

The concept of CGT originated during World War II for the purpose of testing syphilis antigens among soldiers. Since then, it has become popular for screening viruses like COVID-19, HIV, and for cloning libraries for a DNA sequence. It also has many applications in cryptography [8, 9, 20]. CGT is a method designed to find at most d defective items among a set of n items. These items are combined into t groups, in which items are jointly tested, where t is significantly smaller than n, thus allowing savings on the number of tests.

The primary objectives of this thesis are first, to explore and implement a specific type of MTSS, known as d-MTSS, which focuses on locating up to d modifications in a message or document for an integer d fixed a priori. The second objective is to investigate the application

of combinatorial group testing in the context of d-MTSS.

The main contributions of this thesis are as follows. First, we provide a comprehensive explanation of the theoretical concepts underlying d-MTSS, including the cryptographic basics and the use of combinatorial group testing based on set systems called cover-free families. Second, we implement and compare different data structures and decoding methods for combinatorial group testing. Finally, we conduct a thorough assessment of the performance of d-MTSS by looking at factors like the size of the signature, the time it takes to generate keys, and the time needed for signing and verifying documents. Chapters 2 and 4 have background material and Chapters 3, 5, and 6 contain our main contributions.

In Chapter 2, we introduce the foundational concepts of combinatorial group testing (CGT) and cover-free families (CFFs). This chapter is crucial for understanding the mathematical basis of d-MTSS. It explores various methods for constructing d-CFFs, including Sperner sets for $d = 1$, Steiner Triple Systems for $d = 2$, and Reed-Solomon codes for any $d \geq 2$. By establishing this theoretical groundwork, we set the stage for the practical implementation of d-MTSS.

In Chapter 3, we explore the data structures and algorithms crucial for implementing combinatorial group testing (CGT) within d-MTSS. We present and compare different approaches to locate changes (or find defectives), including general methods and specialized algorithms designed for specific d-CFF constructions. This chapter demonstrates how mathematical concepts can be efficiently implemented in algorithms.

Chapter 4 provides essential background on conventional digital signatures and hash functions, which are the cryptographic building primitives of d-MTSS. Additionally, we revisit the definition of modification-tolerant signature schemes and step-by-step algorithms of d-MTSS which are proposed in [20]. This chapter sets the context for understanding how d-MTSS integrates with existing cryptographic primitives to achieve its goals.

Chapter 5 focuses on the practical implementation details of our revised d-MTSS, which we call the MICO scheme. We discuss parameter selection for cryptographic basics, methods for dividing documents into blocks based on their types, and strategies for computing the number of tests required in combinatorial group testing for different types of cover-free families. Furthermore, this chapter showcases how we have translated the theoretical scheme into practice.

In Chapter 6, we illustrate a comprehensive evaluation of the MICO scheme, our d-MTSS implementation. We analyze the performance of various components, including the underlying cryptographic primitives, and assess how different choices of parameters affect the signature size and overall efficiency of the system. This chapter provides crucial insights into the practical viability of d-MTSS and identifies areas for potential future improvements.

Finally, in Chapter 7, we briefly draw some conclusions and discuss future work.

Part of the contents of Chapter 3 was presented at the International Workshop on the Arithmetic of Finite Fields (WAIFI 2024) and has been accepted for the following publication [29]:

Dongxia Luo and Lucia Moura. Fast decoding of group testing results from

Reed-Solomon d-disjunct matrices. To appear in *International Workshop on the Arithmetic of Finite Fields*, volume 15176 of Lecture Notes in Computer Science. Springer International Publishing, 2024. 16 pages.

Chapter 2

Combinatorial Group Testing and Cover-Free Families

2.1 Combinatorial Group Testing

In combinatorial group testing, a set of n items is combined into t groups or pools that are tested together to identify defective items, assuming that there are at most d defectives. Each test checks a subset of n items and gives a negative result if and only if all items in this subset are non-defectives, while a positive result indicates the presence of at least one defective item. There are two types of combinatorial group testing: adaptive and non-adaptive. In adaptive testing, items are chosen based on the results of previous tests. In non-adaptive testing, tests are predetermined and conducted in parallel. In this thesis, we focus on non-adaptive combinatorial group testing, which can be represented by a $t \times n$ binary matrix. In this matrix, each column represents an item and each row represents a test (pool).

Let M be a $t \times n$ binary matrix. The *test result vector* $y \in \{0,1\}^t$ for M has $y_i = 1$ if a test corresponding to row M_i is positive, and $y_i = 0$ otherwise. A decoding algorithm determines the defective items based on y. To be able to locate up to d defectives, M must be $\bar{d}$-separable. This means that the boolean sum of any set of up to d columns must be distinct. A stronger requirement is for M to be the incidence matrix of a cover-free family (CFF), also called a $\bar{d}$-disjunct matrix. In a d-CFF, for any column index j and any other set of d column indices $j_1, \ldots, j_d$, there exists a row i such that $M_{i,j} = 1$, and $M_{i,j_1} = \cdots = M_{i,j_d} = 0$. In particular, for any non-defective item, there exists a test that contains that item but none of the up to d defective items. Our thesis focuses on decoding using d-CFFs, and the details are explained in Section 2.2.

In Table 2.1, we present an example of non-adaptive combinatorial group testing using a 2-CFF for COVID-19: 12 individuals are grouped together in 9 tests, with the assumption that there are at most 2 people who are COVID-positive ($n = 12$, $t = 9$, and $d = 2$). In this scenario, each column in the binary matrix represents one of the 12 people, and each row corresponds to a test. In each test, swabs from 4 individuals are combined into one specimen

tube, resulting in a total of 9 specimen tubes for testing. The *Result* column shows the result of the group test. In this example, tests 3, 4, 5, 6, and 7 yield positive results, indicating that these pools contain individuals who tested positive for COVID-19. Conversely, a negative test means that all individuals in that test are COVID-negative. Based on the negative tests, we can conclude that individuals 1, 3, 4, 5, 6, 7, 8, 9, 10, and 11 are healthy. Therefore, individuals 2 and 12 are identified to be COVID-positive. This conclusion is correct because the 2-CFF property guarantees that every positive individual will appear in a positive test. Indeed, after the identification of the negative individuals from negative tests, the only explanation for test t_4 and t_5 to be positive is that individual 2 is positive. Similar argument holds for test t_1 and t_9 with respect to individual 12.

individuals	1	2	3	4	5	6	7	8	9	10	11	12	Result	y
	✓	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	×		
t_1	1	0	0	1	0	0	1	0	0	1	0	0	negative	0
t_2	1	0	0	0	1	0	0	1	0	0	1	0	negative	0
t_3	1	0	0	0	0	1	0	0	1	0	0	1	positive	1
t_4	0	1	0	1	0	0	0	0	1	0	1	0	positive	1
t_5	0	1	0	0	1	0	1	0	0	0	0	1	positive	1
t_6	0	1	0	0	0	1	0	1	0	1	0	0	positive	1
t_7	0	0	1	1	0	0	0	1	0	0	0	1	positive	1
t_8	0	0	1	0	1	0	0	0	1	1	0	0	negative	0
t_9	0	0	1	0	0	1	1	0	0	0	1	0	negative	0

Table 2.1: An example of combinatorial group testing when $n = 12$, $t = 9$, and $d = 2$.

2.2 Cover-Free Families and Constructions

Cover-free families (CFFs) have been studied under different names: superimposed codes by Kautz and Singleton [23] in 1964, and d-disjunct matrix by Du and Hwang [8], among others. CFFs have found widespread applications across various areas, including information theory, combinatorial group testing, and cryptography [13, 23, 40, 42].

A set system is a pair $(X, \mathcal{B})$, where X is a set of points and $\mathcal{B}$ is a set of subsets (called blocks) of X. A precise definition of d-cover-free family is provided next.

Definition 2.2.1. [19] (**d-cover-free family**) Given $d < t \leq n$ positive integers, a d-cover-free family, denoted d-CFF(t, n), is a set system $(X, \mathcal{B})$ with $|X| = t$ and $|\mathcal{B}| = n$ such that for any block B_{i_0} and any other d blocks $B_{i_1}, ...B_{i_d} \in \mathcal{B}$, we have

$$|B_{i_0} \setminus \bigcup_{j=1}^{d} B_{i_j}| \geq 1. \tag{2.2.1}$$

A set system $(X, \mathcal{B})$ can also be represented as a $t \times n$ binary incidence matrix M. The rows in M correspond to the elements of X, the columns correspond to the blocks $B_1, B_2, \ldots, B_n \in \mathcal{B}$, and $M_{ij} = 1$ if $x_i \in B_j$ and $M_{ij} = 0$ otherwise.

For example, let $t = 9$, $n = 12$, $X = \{1,2,3,4,5,6,7,8,9\}$, and the set system $\mathcal{B}$ containing 12 subsets as follows: $B_1 = \{1,2,3\}, B_2 = \{4,5,6\}, B_3 = \{7,8,9\}, B_4 = \{1,4,7\}, B_5 = \{2,5,8\}, B_6 = \{3,6,9\}, B_7 = \{1,5,9\}, B_8 = \{2,6,7\}, B_9 = \{3,4,8\}, B_{10} = \{1,6,8\}, B_{11} = \{2,4,9\}, B_{12} = \{3,5,7\}$. In this example, no subset is contained in the union of any two others, so it is a 2-CFF$(9,12)$; the incidence matrix of this set system is shown in Table 2.1.

Alternatively, a d-CFF(t,n) can be equivalently defined as a $t \times n$ binary matrix M such that any set of $d+1$ columns contains a permutation sub-matrix of dimension $d+1$. A permutation matrix is a matrix obtained by a permutation of the rows of an identity matrix. The set system that has such a matrix as its incidence matrix satisfies precisely the requirements of Definition 2.2.1.

One of the main objectives in group testing is to minimize the number of tests, which is minimizing the number of rows in d-CFFs. Given d and n, let $t(d,n)$ be the minimum number of rows (tests) in a d-CFF with n columns. It has been shown that $\Omega(\frac{d^2 \log n}{\log d}) \leq t(d,n) \leq O(d^2 \log n)$. The lower bound was determined in [11] and the upper is achieved by the deterministic algorithm by Porat and Rotschild [34]. Closing the gap between lower and upper bound is an important open question.

Another important objective of group testing is that the decoding time $T(t,n)$, that is the determination of the defective items from the test result vector, be done efficiently. The next result shows that it is sufficient to remove all non-defective items identified from the negative test results of a set of items, as the remaining items will be the defective ones.

Proposition 2.2.2. Let M be a d-CFF(t,n). Let $y \in \{0,1\}^t$ be the test result vector for combinatorial group testing with at most d defective items. Then, the set of defectives $D = \{1,\ldots,n\} \setminus \bigcup\limits_{i:y_i=0} \{j : M_{i,j} = 1\}$.

Proof. Let $N = \bigcup\limits_{i:y_i=0} \{j : M_{i,j} = 1\}$. The fact that the items in N are non-defectives is obvious since they belong to negative tests. It remains to show that every non-defective item is in N. Suppose $j_1, \ldots, j_d$ are defectives, then for any non-defective item j, we can always find a row i such that $M_{i,j} = 1$, and $M_{i,j_1} = \cdots = M_{i,j_d} = 0$. This results in a negative test, so $y_i = 0$ and $j \in N$. ∎

If we conduct group testing using d-CFFs, the decoding algorithm computes a vector in $\{0,1\}^n$ that is the logical-OR of all rows M_i such that $y_i = 0$, to identify the healthy (negative) items. The complement of this vector then gives the defective (positive) items. In group testing with at most d positive items using a d-CFF(t,n), the decoding time $T(t,n)$ is in $O(tn)$. According to [12], a decoding algorithm for group testing is considered efficient if the decoding time is $O(tn)$ and is time-optimal if it is $O(t)$.

There exist various ways to construct CFFs, including direct construction from combinatorial designs and codes such as packing arrays, orthogonal arrays, and nonlinear codes [40, 42]. Additionally, recursive constructions can be used, building on smaller CFFs to build large CFFs [17, 26]. They are also built from other objects [26, 40], as well as the probabilistic method [5, 34]. In this thesis, we discuss how to construct CFFs using direct construction:

1-CFFs and 2-CFFs using set systems and general d-CFFs using Reed-Solomon codes. We describe these constructions in the following sections.

2.2.1 $d = 1$: Sperner Construction

A Sperner family is a set system $(X, \mathcal{B})$ such that none of the sets in $\mathcal{B}$ is a proper subset of another set in $\mathcal{B}$. Sperner's Theorem [38] states the following facts:

1. For every Sperner family $\mathcal{S}$ with $|X| = t$, $|S| \le \binom{t}{\lfloor t/2 \rfloor}$.
2. Equality holds if and only if $\mathcal{S}$ consists of all subsets of X that have size $\lfloor \frac{t}{2} \rfloor$ or $\mathcal{S}$ consists of all subsets of X that have size $\lceil \frac{t}{2} \rceil$.

A 1-CFF is a set system such that none of the sets in $\mathcal{B}$ is contained within another set, or in other words, it is a Sperner family. Furthermore, Sperner's Theorem implies that the minimum number of rows in a 1-CFF with n columns is $t(1, n) = \min\{t : \binom{t}{\lfloor t/2 \rfloor} \ge n\}$.

Note that for $t = t(1, n)$, $\binom{t-1}{\lfloor (t-1)/2 \rfloor} < n \le \binom{t}{\lfloor t/2 \rfloor}$. Applying Stirling's approximation to both binomial coefficients, we approximate the inequality as $\frac{2^{t-1}}{\sqrt{\pi(t-1)/2}} < n \le \frac{2^t}{\sqrt{\pi t/2}}$ for large t. Taking the natural logarithm in base 2 of this equation, we simplify to obtain $t - 1 - \log_2 \sqrt{\pi(t-1)/2} < \log_2 n \le t - \log_2 \sqrt{\pi t/2}$ as t increases. Consequently, we achieve $t(1, n) \approx \log_2 n$.

The matrix below illustrates an example of constructing a 1-CFF from the Sperner family when $n = 6$. In this instance, $X = \{1, 2, 3, 4\}$, and $\mathcal{B} = \{12, 13, 14, 23, 24, 34\}$:

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

2.2.2 $d = 2$: Steiner Triple System

When it comes to $d = 2$, a 2-CFF is a set system where no block can be entirely contained within the union of any other two blocks. Next, we show how to construct 2-CFFs using Steiner Triple Systems (STS). Constructing 2-CFFs using STS gives optimal $t(2, n)$ for $n = 7$ and $n = 9$.

Definition 2.2.3. (**Steiner Triple System**) Let $v > 3$ be an integer. A Steiner Triple System is a set system $(X, \mathcal{B})$, where X is a finite set of v points or symbols, and $\mathcal{B}$ is a set of 3-element subsets of X called triples (blocks), $|\mathcal{B}| = b$. Each pair of distinct elements of X occurs in exactly one of the blocks of $\mathcal{B}$.

In other words, this concept is equivalent to partitioning the edges of a complete graph on v vertices, K_v, into triangles, as each triple in the STS corresponds to a triangle in K_v,

and each pair of vertices is contained in exactly one triangle. Furthermore, the number of blocks, b, in a STS(v) is determined by a counting argument. Specifically, the total number of pairs among v elements is $\frac{v(v-1)}{2}$. Since each block (triple) contains exactly 3 pairs, the total number of blocks is $b = \frac{v(v-1)}{6}$. Proposition 2.2.4 states the necessary conditions for the existence of an STS(v).

Proposition 2.2.4. There exists an STS(v) only if $v \equiv 1, 3 \pmod 6$, $v \geq 7$.

Proof. [39] Every point in V has to occur $\frac{v-1}{3-1} = \frac{v-1}{2}$ times, so v has to be odd. Since the number of blocks $b = \frac{v(v-1)}{6}$ is a positive integer, we must have $v \equiv 0, 1, 3, 4 \pmod 6$. Since v is odd and $v > 3$, we conclude $v \equiv 1, 3 \pmod 6$, $v \geq 7$. ∎

Proposition 2.2.4 gives a necessary condition for existence of STS. To prove its sufficiency, there are two constructions: one for $v \equiv 1 \pmod 6$ and another for $v \equiv 3 \pmod 6$. In Section 2.2.2.2, we present the Bose Construction when $v \equiv 3 \pmod 6$ while in Section 2.2.2.3, we illustrate the Skolem Construction when $v \equiv 1 \pmod 6$. However, in order to understand the two constructions, we first talk about Latin squares and quasigroups.

2.2.2.1 Latin Squares and Quasigroups

We begin by defining Latin squares and quasigroups.

Definition 2.2.5. A Latin Square of order n with entries from n-set X is an $n \times n$ array L in which every cell contains an element of X such that every row and every column of L are permutations of X.

Figure 2.1 is an example of a Latin square of order 5:

0	3	1	4	2
3	1	4	2	0
1	4	2	0	3
4	2	0	3	1
2	0	3	1	4

Figure 2.1: Latin square of order 5.

Definition 2.2.6. Let X be a finite set of cardinality n, and let $\circ$ be a binary operation defined on X (i.e. $X \circ X \to X$). The pair $(X, \circ)$ is a quasigroup of order n if

1. For every $x, y \in X$, the equation $x \circ z = y$ has a unique solution for $z \in X$.
2. For every $x, y \in X$, the equation $z \circ x = y$ has a unique solution for $z \in X$.

Theorem 2.2.7 allows us to relate quasigroups to Latin squares.

Theorem 2.2.7. Suppose $\circ$ is a binary operation defined on a finite set X of cardinality n. Then $(X, \circ)$ is a quasigroup if and only if its operation table is a Latin square of order n.

Consequently, Latin squares and quasigroups provide two different ways of looking at the same thing. Now, we investigate some properties of them.

First, we give definitions of a commutative/symmetric and idempotent quasigroup/Latin square.

Definition 2.2.8. Suppose $(X, \circ)$ is a quasigroup, then:

- $(X, \circ)$ is an *idempotent* quasigroup if $x \circ x = x$ for all $x \in X$.
- $(X, \circ)$ is a *commutative* quasigroup if $x \circ y = y \circ x$ for all $x, y \in X$.

Definition 2.2.9. Suppose we have a Latin square of order n, then:

- It is *idempotent* if cell (i, i) contains i for $1 \leq i \leq n$.
- It is *symmetric* if cell (i, j) and cell (j, i) contain the same symbol, for all $1 \leq i,\ j \leq n$.

Figure 2.1 is a symmetric idempotent Latin square. Next, we introduce what is a half-idempotent quasigroup/Latin square.

Definition 2.2.10. A Latin square L of order $2n$ is *half-idempotent* if for $1 \leq i \leq n$ cells (i, i) and $(n + i, n + i)$ of L contain the symbol i.

Definition 2.2.11. Let $X = \{0, \ldots, n - 1\}$ for n even, a quasigroup $(X, \circ)$ is a *half-idempotent* quasigroup provided that:

$$x \circ x = \begin{cases} x & 0 \leq x < \frac{n}{2}, \\ x - \frac{n}{2} & \frac{n}{2} \leq x < n. \end{cases} \tag{2.2.2}$$

2.2.2.2 Bose Construction

The Bose Construction [27, 39] uses idempotent commutative quasigroups to construct a Steiner Triple System (STS) of all orders $v \equiv 3 \pmod 6$.

Let $v = 6n+3$ and let $(Q, \circ)$ be an idempotent commutative quasigroup of order $2n+1$, where $Q = \{0, 1, 2, 3, ..., 2n\}$. Let $X = Q \times \{0, 1, 2\}$, and define $\mathcal{B}$ to contain triples of two types:

Type 1: For $0 \leq x \leq 2n$, $\{(x, 0), (x, 1), (x, 2)\} \in \mathcal{B}$.

Type 2: For $0 \leq x < y \leq 2n$, $\{(x, 0),\ (y, 0),\ (x \circ y, 1)\}$, $\{(x, 1),\ (y, 1),\ (x \circ y, 2)\}$, $\{(x, 2),\ (y, 2),\ (x \circ y, 0)\} \in \mathcal{B}$.

The Bose Construction uses the following quasigroup operation for odd order $2n + 1$:

$$x \circ y = \left(\frac{(2n+1)+1}{2}\right)(x + y) \bmod 2n + 1 = (n + 1)(x + y) \bmod 2n + 1. \tag{2.2.3}$$

Below is an example of STS(9) using an idempotent commutative quasigroup of order 3, shown in Figure 2.2:

0	2	1
2	1	0
1	0	2

Figure 2.2: A symmetric idempotent Latin square of order 3.

After that, let $X = \{(0,1,2)\times(0,1,2)\}$ using Bose construction, $\mathcal{B}$ contains the following 12 triples:

- Type 1: $\{(0,0),(0,1),(0,2)\}$, $\{(1,0),(1,1),(1,2)\}$, $\{(2,0),(2,1),(2,2)\}$.
- Type 2:
  - When $x=0$, $y=1$:$\{(0,0),(1,0),(2,1)\}$, $\{(0,1),(1,1),(2,2)\}$, $\{(0,2),(1,2),(2,0)\}$.
  - When $x=0$, $y=2$:$\{(0,0),(2,0),(1,1)\}$, $\{(0,1),(2,1),(1,2)\}$, $\{(0,2),(2,1),(1,0)\}$.
  - When $x=1$, $y=2$:$\{(1,0),(2,0),(0,1)\}$, $\{(1,1),(2,1),(0,2)\}$, $\{(1,2),(2,2),(0,0)\}$.

By assigning the number 1 to $(0,0)$, the number 2 to $(0,1)$, and continuing lexicographically up to the number 12 for $(2,2)$, we are able to obtain the following set:

$$\mathcal{B} = \left\{ \begin{array}{ll} \{1,2,3\}, & \{4,5,6\}, \\ \{7,8,9\}, & \{1,4,7\}, \\ \{2,5,8\}, & \{3,6,9\}, \\ \{1,5,9\}, & \{2,6,7\}, \\ \{3,4,8\}, & \{1,6,8\}, \\ \{2,4,9\}, & \{3,5,7\} \end{array} \right\}.$$

When we convert this Steiner Triple System into a binary matrix, we obtain the following 9×12 matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Now, we are ready to establish the following lemma.

Lemma 1. The Bose construction gives an STS(v) for all $v \equiv 3 \pmod 6$, $v \geq 9$.

Proof. [27] The number of blocks is

$$(2n+1)+3\left(\frac{(2n+1)(2n)}{2}\right) = (2n+1)(3n+1) = \frac{v}{3}\left(\frac{v-1}{2}\right) = \frac{v(v-1)}{6}.$$

To show that every pair of points appears in at least one block, we consider an arbitrary pair of points (x,i) and (y,j). We analyze three cases:

1. **Case** $x = y$:
   Points (x,i) and (x,j) share the block $\{(x,0),(x,1),(x,2)\}$.
2. **Case** $i = j$:
   Points (x,i) and (y,i) share the block $\{(x,i),(y,i),(x \circ y,(i+1) \mod 3)\}$.
3. **Case** $x \neq y$ **and** $i \neq j$:
   Order pairs so that $j = (i+1) \mod 3$. Since we have a quasigroup, there exists a unique z such that $x \circ z = y$. As the quasigroup is idempotent and $x \neq y$, we have $x \neq z$. Therefore, points (x,i), (y,j), and (z,i) share the block $\{(x,i),(z,i),(x \circ z = z \circ x = y,(i+1) \mod 3 = j)\}$.

Finally, the counting of blocks guarantees that they appear exactly once. ∎

2.2.2.3 Skolem Construction

For all orders $v \equiv 1 \pmod 6$, $v \geq 7$, we use the Skolem Construction [27, 39] to build STS(v). Instead of using idempotent commutative quasigroups, the Skolem Construction uses half-idempotent commutative quasigroups.

Let $v = 6n+1$ and let $(Q, \circ)$ be a half-idempotent commutative quasigroup of order $2n$, where $Q = \{0,1,2,3,...,2n-1\}$. Let $S = \{\infty\} \cup (Q \times \{0,1,2\})$, and define $\mathcal{B}$ to contain triples of three types:

Type 1: For $0 \leq x \leq n-1$, $\{(x,0),(x,1),(x,2)\} \in \mathcal{B}$.

Type 2: For $0 \leq x \leq n-1$, $\{\infty,\ (n+x,0),\ (x,1)\}$, $\{\infty,\ (n+x,1),\ (x,2)\}$, $\{\infty,\ (n+x,2),\ (x,0)\} \in \mathcal{B}$.

Type 3: For $0 \leq x < y \leq 2n-1$, $\{(x,0),\ (y,0),\ (x \circ y,1)\}$, $\{(x,1),\ (y,1),\ (x \circ y,2)\}$, $\{(x,2),\ (y,2),\ (x \circ y,0)\} \in \mathcal{B}$.

The Skolem construction uses the following quasigroup operation for even order $2n$:

$$x \circ y = \pi((x+y)) \bmod 2n \tag{2.2.4}$$

where the permutation π is defined as

$$\pi(x) = \begin{cases} (x+2n-1)/2 & \text{if } x \text{ is odd,} \\ x/2 & \text{if } x \text{ is even.} \end{cases} \tag{2.2.5}$$

Figure 2.3 shows a half-idempotent commutative quasigroup of order 2 to construct an STS(7).

0	1
1	0

Figure 2.3: A symmetric half-idempotent Latin square of order 2.

Let $X = \infty \cup (\{0,1\} \times \{0,1,2\})$. $\mathcal{B}$ contains the following 7 triples:

- Type 1: $\{(0,0),(0,1),(0,2)\}$.
- Type 2: $\{\infty,(1,0),(0,1)\}$,$\{\infty,(1,1),(0,2)\}$,$\{\infty,(1,2),(0,0)\}$.
- Type 3: $\{(0,0),(1,0),(1,1)\}$, $\{(0,1),(1,1),(1,2)\}$,$\{(0,2),(1,2),(1,0)\}$.

If we assign 1 to $(0,0)$, and 2 to $(0,1)$, and continuing sequentially up to the number 7 for ∞, we can obtain the following set:

$$\mathcal{B} = \left\{ \begin{array}{c} \{1,2,3\}, \\ \{2,4,7\}, \\ \{3,5,7\}, \\ \{1,6,7\}, \\ \{1,4,5\}, \\ \{2,5,6\}, \\ \{3,4,6\} \end{array} \right\}.$$

When represented $\mathcal{B}$ as a binary matrix, we get the following 7×7 binary matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}.$$

Similar to Bose construction, we can establish the following lemma.

Lemma 2. Skolem Construction gives an STS(v) for all $v \equiv 1 \pmod 6$, $v \geq 7$.

Proof. [39] The number of blocks is

$$b = 4n + 3\left(\frac{2n(2n-1)}{2}\right) = 6n^2 + n = n(6n+1) = \frac{v(v-1)}{6}.$$

Now we need to verify that each pair of points P, Q occur at least once. The second coordinate is considered in $\mathbb{Z}_3$, so $i+1$ means $(i+1) \mod 3$.

Now consider the following three cases:

- $P = (x, i), Q = \infty$:
  If $x \leq n-1$, then the pair is $\{\infty, (n+x, i-1), (x, i)\}$.
  If $x \geq n$, then the pair is $\{\infty, (x, i), (x-n-1, i+1)\}$.

- $P = (x, i), Q = (x, j), i \neq j$:
  If $x \leq n-1$, then it is in Type 1.
  If $x \geq n$, assume without loss of generality that $j = i+1$. The equation $x \circ y = x$ has a unique solution y such that $y \neq x$. Then points (x, i), $(x, i+1)$ are in $\{(x, i), (y, i), (x \circ y = y \circ x = x, i+1)\}$, which is in Type 3.

- $P = (x, i), Q = (y, j)$ for $x < y$ and $x \neq y$. Consider the following cases: $i = j$, $j = i+1$, or $i = j+1$.
  - $(x, i), (y, i)$: it is a Type 3 block $\{(x, i), (y, i), (x \circ y, i+1)\}$.
  - $(x, i), (y, i+1)$: The equation $z \circ x = y$ has a unique solution z. Note that $z \neq x$ since $x < y$ and $x \circ x \leq x$ for all x (half-idempotent). Then (x, i) and $(y, i+1)$ are in $\{(x, i), (z, i), (x \circ z = z \circ x = y, i+1)\}$.
  - $(x, j+1), (y, j)$: The equation $z \circ y = x$ has a unique solution z. If $z = y$, then $z = x+n$, and the pair is in $\{\infty, (y = z = n+x, j), (x, j+1)\}$. If $z \neq y$, then the pair is in $\{(y, j), (z, j), (y \circ z = z \circ y = x, j+1)\}$.

Finally, the counting of blocks guarantees that they appear exactly once.

∎

2.2.2.4 Sufficient and Necessary Condition for the Existence of STS

Lemma 1 and Lemma 2 give sufficient condition for existence of STS: we can construct STS(v) using Bose construction for $v \equiv 3 \pmod 6$ and Skolem construction for $v \equiv 1 \pmod 6$. By combining these lemmas with Proposition 2.2.4, we can establish necessary and sufficient conditions for the existence of STS(v).

Theorem 2.2.12. There exists an STS(v) if and only if $v \equiv 1, 3 \pmod 6$, $v \geq 7$.

2.2.3 Any $d \geq 2$: Reed-Solomon Codes

While a Sperner family is a 1-CFFs and Steiner Triple System is a 2-CFF, constructing CFFs from Reed-Solomon codes allows to build CFFs for any integer $d \geq 2$. We start by giving the formal definition of a code.

Definition 2.2.13. (**code**) An (N, n, q) code $\mathcal{C}$ of length N over an alphabet Q of size q consists of n codewords $c = (c_1, ..., c_N)$, with $c_i \in Q$, $1 \leq i \leq N$. The code $\mathcal{C}$ has minimum distance D if any two codewords in $\mathcal{C}$ differ in at least D positions.

From [19], we know that we can obtain a set system $(X, \mathcal{B})$ from the codes by taking $X = \{1, ..., N\} \times Q$, and $\mathcal{B} = \{B_c = \{(1, c_1), ..., (N, c_N)\} : c = (c_1, ..., c_N) \in \mathcal{C}\}$. This will result in a uniform set system with $|X| = Nq$, $|\mathcal{B}| = n$, and $|B_c| = N$ for all $c \in \mathcal{C}$. Proposition 2.2.14 establishes the relationship between d-CFFs and codes.

Proposition 2.2.14. [42] If there exists an (N, n, q)-code $\mathcal{C}$ with minimum distance D, then there exists a d-CFF(Nq, n), for any $d \leq \lfloor \frac{N-1}{N-D} \rfloor$.

Below is an example of a $(3, 9, 3)$-code $\mathcal{C}$ with minimum distance $D = 2$, where columns of $\mathcal{C}$ represent the codewords.

$$\begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 2 & 3 & 1 & 3 & 1 & 2 \\ 1 & 2 & 3 & 3 & 1 & 2 & 2 & 3 & 1 \end{pmatrix}$$

In this example, the symbols 1, 2, and 3 in each codeword are mapped to the vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively. As a result, the $(3, 9, 3)$-code $\mathcal{C}$ is equivalent to a 2-CFF$(9, 9)$, where $d \leq \lfloor \frac{3-1}{3-2} \rfloor = 2$.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Let q be a prime power. A (N, q^k, q) *Reed-Solomon code* is a code with $n = q^k$ codewords, each of size $N \leq q$. Each column of a Reed-Solomon code represents a unique polynomial of degree at most $k - 1$ over the finite field $\mathbb{F}_q$. To obtain a codeword, evaluate each polynomial f at N distinct points $a_0, a_1, \ldots, a_{N-1}$ in $\mathbb{F}_q$. The resulting codeword is $(f(a_0), f(a_1), \ldots, f(a_{N-1}))$. We can construct Reed-Solomon codes as follows:

- Assign to each row a polynomial: $f(x) = a_0 + a_1x + ... + a_{k-1}x^{k-1}$ for each possible tuple $(a_0, a_1, \ldots, a_{k-1}) \in \mathbb{F}_q^k$.
- Allocate a distinct element $\alpha \in \mathbb{F}_q$ to each of the N columns.
- In the array position indexed by row $(a_0, a_1, ..., a_{k-1})$ and column α, put the value $f(\alpha) = a_0 + a_1\alpha + \cdots + a_{k-1}\alpha^{k-1}$. The rows of this matrix corresponds to codewords.
- The minimum distance is $D = N - k + 1$.

Table 2.2 gives an example of a Reed-Solomon code with parameters $q = 3$, $N = 3$ and $k = 2$. The minimum distance in this example is $D = 3 - 2 + 1 = 2$.

$f(x)$	$f(0)$	$f(1)$	$f(2)$
$0x + 0$	0	0	0
$0x + 1$	1	1	1
$0x + 2$	2	2	2
$1x + 0$	0	1	2
$1x + 1$	1	2	0
$1x + 2$	2	0	1
$2x + 0$	0	2	1
$2x + 1$	1	0	2
$2x + 2$	2	1	0

Table 2.2: A (3, 9, 3) Reed-Solomon code over $\mathbb{F}_3$, where each row is a codeword.

In the previous page, the values in $(3, 9, 3)$-code $\mathcal{C}$ range from 1 to N. However, in Reed-Solomon codes, the values are shifted to range from 0 to $N - 1$. As a result, the codeword $(1, 2, 3)$ in $\mathcal{C}$ becomes $(0, 1, 2)$ in Table 2.2.

The next known theorem shows that appropriate choices of parameters in Reed-Solomon codes yields d-CFFs(t,n) with $t \in O(d^2(\frac{\log n}{\log d})^2)$. This is not too far from the best known bounds $\Omega(\frac{d^2 \log n}{\log d}) \leq t(d, n) \leq O(d^2 \log n)$. A proof of Theorem 2.2.15 can be found in [3, Section 7.2.2].

Theorem 2.2.15. Let $d < n$ be positive numbers. There is a d-CFFs(t,n) from Reed-Solomon codes satisfying t in $O(d^2(\frac{\log n}{\log d})^2)$.

Proof. (sketch) Take $q \approx 2d\frac{\log n}{\log d}$ a prime number (by Bertrand's postulate there is always a prime between this number and its double); take $k = \lceil \log_q n \rceil$ and $N = d(k - 1) + 1$. We use Reed-Solomon codes and for large enough parameters this gives a d-CFF as

$$\left\lfloor \frac{q-1}{k-1} \right\rfloor \geq \left\lfloor \frac{2d\frac{\log n}{\log d} - 1}{\frac{\log n}{\log q} + 1 - 1} \right\rfloor = \left\lfloor 2d\frac{\log q}{\log d} - \frac{\log q}{\log n} \right\rfloor \geq d,$$

and $t = Nq \leq q^2 \approx (2d\frac{\log n}{\log q})^2$. ∎

In our work with d-CFFs based on Reed-Solomon codes, we restrict q to be a prime rather than an arbitrary prime power, so that $\mathbb{F}_q \cong \mathbb{Z}_q$. Since there is some flexibility in the choice of the prime q and the parameter k, and there are plenty of prime numbers available, we can still maintain the necessary relationship between $n \leq q^k$ and $n \approx q^k$, while achieving a good number of tests based on Theorem 2.2.15. Therefore, this restriction does not affect much our use of this construction, and enables the faster decoding presented in Section 3.2.3.

Chapter 3

Data Structures and Fast Decoding Algorithms for Group Testing

This chapter includes content from our paper published in the proceedings of the International Workshop on the Arithmetic of Finite Fields in 2024 [29].

In this chapter, we first present a general decoding algorithm to locate defective items using a d-CFF(t, n) binary matrix M from a vector of test results $y \in \{0,1\}^t$. Then we discuss two data structures for M. In addition, we present three specific fast decoding algorithms corresponding to three constructions: Sperner for $d = 1$, Steiner Triple System for $d = 2$, and Reed-Solomon codes for general $d \geq 2$ as discussed in Section 2.2. These specific algorithms allow us to use the structure of the set systems or codes to directly find defectives in group testing without having to build the whole binary matrix.

3.1 General Group Testing Decoding Algorithm

The advantage of the general group testing algorithm for decoding test results from M is that it works for all types of d-CFF constructions. As shown in Proposition 2.2.2, the d-CFF property guarantees that each non-defective item will appear in a negative test. Thus, from test results, we are able to calculate a vector in $\{0,1\}^n$ by applying logical-OR of all rows M_i such that $y_i = 0$ (negative tests) to find non-defectives. Finally, we simply complement the vector to find all the defective items, which is shown in Algorithm 1, namely MatrixDecode.

Converting sets or codes into a binary matrix for large dimensions can be resource-intensive and time-consuming, especially since each row typically contains at least half zeros. Storing the entire binary matrix may not be practical when we are more concerned with the presence of items in each test. Therefore, we have chosen two alternative data structures for storing the matrix more efficiently: matrix represented as set arrays to encode positions of 1s, called MatrixList, and matrix presented as integer-based bit arrays, called MatrixCompact.

Algorithm 1 MATRIXDECODE(d, M, y)

Input: the value of d, a d-CFF(t, n) M, a vector of test results $y \in \{0,1\}^t$
Output: a boolean value indicating correct decoding, the set of defective items I

```
1: Set U = ∅;                          ▷ a set to store the union of the non-defective items
2: for i = 1 to t do
3:     if (y_i == 0) then                                         ▷ if test i is negative
4:         U = U ∪ {j : M[i][j] = 1};               ▷ all items in test i are non-defective
5:     end if
6: end for
7: I = {1, . . . , n} \ U;        ▷ complementing set U gives the set of defective items
8: return ("|I| ≤ d", I);
```

Algorithm 1 can yield two possible outcomes depending on whether or not the actual number of defective item is smaller or equal to d. The first outcome is the algorithm outputting true, with the set I containing all the defective items (this is only guaranteed if $|I| \leq d$). The second outcome occurs when the algorithm outputs false; in this case, I includes all the defective items, but it may also contain some non-defective items because more than d defectives are present and the assumptions of Proposition 2.2.2 is not satisfied.

In the following two sections, we present the details of the two data structures to store matrix M and provide the pseudocodes for finding defectives using each of them according to Algorithm 1.

3.1.1 Matrix Representation Using Set Arrays to Encode Positions of 1s

The data structure MATRIXLIST represents a CFF Matrix M as an array of sets, where each set corresponds to a row and stores the column indices of entries that are 1 in that row. Algorithm 1 requires union of sets corresponding to rows (Step 4) and set complementation (Step 7). Therefore, using MATRIXLIST to implement Algorithm 1 requires an efficient data structure to store and do operations on sets, which we detail next.

In MATRIXLIST, each set corresponding to a row of M is implemented as a sorted array of integers in $\{1, \dots, n\}$. The union of two sets A and B can be done by merging the two sorted arrays in time proportional to $O(|A| + |B|)$. The complement of a subset of $\{1, 2, 3, ..., n\}$ can be computed in time $O(n)$. The pseudocodes for complementing a array with respect to $\{1, 2, 3, ..., n\}$ and unioning two sorted arrays are shown in Algorithms 2 and 3.

Algorithm 2 ComplementSet(L, n)

Input: a sorted array, L, containing the elements of a subset $S \subseteq \{1, 2, ..., n\}$
Output: a sorted array, L_c, containing the elements of $\{1, 2, ..., n\} \setminus S$

```
1: i = 1, j = 1, k = 1;
2: L_c.length = n − L.length;
```

```
3:  while j ≤ L.length do
4:      if i == L[j] then
5:          i = i + 1;
6:          j = j + 1;
7:      else
8:          L_c[k] = i;
9:          i = i + 1;
10:         k = k + 1;
11:     end if
12: end while
13: while i ≤ n do
14:     L_c[k] = i;
15:     i = i + 1;
16:     k = k + 1;
17: end while
18: Return L_c;
```

Algorithm 3 UnionSets(L_1, L_2)

Input: two sorted array: L_1 and L_2
Output: a merged sorted array, L_u

```
1:  i = 0, j = 0, k = 0;
2:  L_u.length = L_1.length + L_2.length
3:  while i < L_1.length and j < L_2.length do
4:      if L_1[i] == L_2[j] then
5:          L_u[k] = L_1[i];
6:          i = i + 1;
7:          j = j + 1;
8:          k = k + 1;
9:      else if L_1[i] < L_2[j] then
10:         L_u[k] = L_1[i];
11:         i = i + 1;
12:         k = k + 1;
13:     else
14:         L_u[k] = L_2[j];
15:         j = j + 1;
16:         k = k + 1;
17:     end if
18: end while
19: while i < L_1.length do          ▷ there are some elements remaining in L_1
20:     L_u[k] = L_1[i];
21:     i = i + 1;
22:     k = k + 1;
23: end while
```

```
24: while j < L_2.length do          ▷ there are some elements remaining in L_2
25:     L_u[k] = L_2[j];
26:     j = j + 1;
27:     k = k + 1;
28: end while
29: Adjust length of L_u to be k;
30: Return L_u;
```

Suppose each row of M has a constant number L of 1s; in this case we have at most t lists of size L to merge. If we implement Algorithm 1 in page 17 using this data structure, we obtain a running time of at most $2L + 3L + \cdots + tL + n = O(Lt^2 + n)$. However, if we combine the lists in a *bottom-up merge*, which first combines the pairs of lists of size L, then pairs of lists of size $2L$, then pairs of lists of size $4L$, etc., until all lists are merged, then the time complexity is $O(Lt \log t + n)$. Note that this analysis is a bit rough, since we are upper bounding the size of the union of two sets with the sum of the cardinalities of the sets. For both variations, in the worst case, with respect to L, we still have the upper bound $O(tn)$ time. For example, when each of the t tests has L linear in n (i.e., $L = \Theta(n)$), then $O(Lt^2 + n) = O(nt^2 + n)$ and $O(Lt \log t + n) = O(nt \log t + n)$. However, in both cases, this upper bound is not tight when $L = \Theta(n)$. Indeed, observe that each merge operation can cost at most $2n$, and we perform at most t merges in total. Therefore, regardless of the merging strategy, we arrive at a tighter worst-case bound of $O(tn)$. Nevertheless, in practice *bottom-up merge* is beneficial in many cases. We have experimented with both variations and opted for the one that uses the *bottom-up merge*. The decoding algorithm for MatrixList using *bottom-up merge* is shown in Algorithm 4. For a MatrixList M, the method M.getRow(i) returns the sorted array representing row i of the matrix.

Algorithm 4 findDefectives(d, M, y)

Input: the value of d, a d-(t, n) MatrixList M, a vector of test results $y \in \{0, 1\}^t$
Output: a boolean value indicating correct decoding, the set of defective items I

```
1: linked = a linked list used as a queue to process the merge of sorted arrays;
2: merged = an array of maximum size n;
3: I = an empty list;
4: for i = 0 to y.length−1 do
5:     if y[i] == 0 then                                   ▷ negative pool
6:         linked.add(M.getRow(i));    ▷ retrieve the sorted array representing row i of M
7:     end if
8: end for
9: while linked.length> 1 do
10:    L_1 = linked.removeFirst();
11:    L_2 = linked.removeFirst();
12:    merged = UnionSets(L_1, L_2);
13:    linked.addLast(merged, n);
14: end while
```

```
15: merged = linked.removeFirst();
16: I = ComplementSet(merged, n));
17: return ("|I| ≤ d", I);
```

3.1.2 Matrix Representation with Integer-Based Bit Arrays

In MatrixCompact data structure, each row of M corresponds to a set implemented as a bit array stored as an array of unsigned integers of B bits each. Each array has size n/B; we used $B = 64$. The union of two sets is done via n/B bitwise-or operations for each pair of integers. Complement is done with bitwise complement. Alternatively, instead of using the bitwise complement operation, we can directly identify the locations of 0s in the resulting bit array.

The running time for the algorithm using this data structure is $O(\frac{tn}{B})$, which is still $O(tn)$ for constant B. However, in practice, this linear reduction on running time makes a difference. Using bit arrays and bitwise operations significantly improves the algorithm's performance in real-world scenarios. This improvement is especially helpful when working with large matrices, where it greatly reduces the time needed for computations. The pseudocode for finding defectives using this data structure is shown next.

Algorithm 5 findDefectives(d, M, y)

Input: the value of d, a d-(t, n) MatrixCompact M, a vector of test results $y \in \{0, 1\}^t$
Output: a boolean value indicating correct decoding, the set of defective items I

```
1: B = the number of bits in an unsigned integer;
2: numCols = ⌈n/B⌉;
3: resultRow = an array with size of numCols;
4: I = an empty list;
5: for i = 0 to y.length−1 do
6:     if y[i] == 0 then                                    ▷ negative pool
7:         for j = 0 to numCols−1 do                        ▷ bitwise operation
8:             resultRow[j] = resultRow[j] ∨ M[i][j];       ▷ bitwise OR operation
9:         end for
10:    end if
11: end for
12: if n%B != 0 then                 ▷ the last integer does not fill the full base
13:     lastColValue = resultRow[numCol−1];
14:     bitsToPad = B − n%B;
15:     for i = 0 to bitsToPad−1 do
16:         lastColValue | = 1L << (B − 1 − i)           ▷ left padding 1s to the integer
17:     end for
18: end if
```

```
19: for i = 0 to numCols−1 do
20:     x = resultRow[i];
21:     element = i * B + 1;
22:     if x == 2^B − 1 then                      ▷ if the integer contains only 1s
23:         continue;
24:     end if
25:     for j = 0 to B − 1 do
26:         if (x&1) == 0 then                     ▷ locate 0s
27:             I.add(element);
28:         end if
29:         x >>= 1;
30:         element = element+1;
31:     end for
32: end for
33: return ("|I| ≤ d", I);
```

When the last integer in resultRow cannot fully occupy the base B, Step 12-18 in Algorithm 5 extend the integer to length B by padding the leftmost bitsToPad bits with 1s. Padding with 1s instead of 0s prevents interference when identifying bits of 0 later. Steps 19 to 31 iterate through each integer in resultRow to locate bits that are 0. For an integer x at index i in resultRow, the bits are denoted as $b_{B-1}b_{B-2}\dots b_1b_0$. The bits are checked starting from the least significant position: the bit b_0 corresponds to position $i \times B+1$, b_1 to $i \times B+2$, and so on, up to b_{B-1}, which corresponds to position $i \times B+B$. If any bit b_k is 0, its position $i \times B+(k+1)$ is added to set I. This process continues until all integers are processed, identifying the positions of all 0s, meaning that we have found all the defectives.

In this section, we presented two data structures and algorithms for decoding a general d-CFF matrix. Both decoding algorithms have $O(tn)$ worst-case running time. In the experiments used in Section 3.3, we observe that MatrixCompact algorithm is faster than MatrixList. However, the situation could be different when L, the number of 1's in each row, is small since the upper bound of $O(Lt\log t+n)$ for MatrixList can be tighter.

3.2 Three Specific Group Testing Decoding Algorithms

In the following sections, we present three specific fast decoding algorithms for the constructions discussed in Section 2.2: Sperner set systems for $d=1$, Steiner Triple Systems for $d=2$, and Reed-Solomon codes for any $d \geq 2$.

The general decoding algorithm using a binary matrix M in Section 3.1 locates defective items from the negative test results. In contrast, the specific decoding algorithms for $d \geq 2$ find defectives from the positive test results.

3.2.1 Sperner Decoding Algorithm

To compute $\lfloor t/2 \rfloor$-subsets from a set of size t for Sperner's construction, we use the lexicographic successor algorithm from Algorithm 2.6 in Kreher and Stinson's book on Combinatorial Algorithms [24, Chapter 2]. This enables us to generate a Sperner set system and store it in an array with dimensions $n \times \lfloor t/2 \rfloor$, for $n \leq \binom{t}{\lfloor t/2 \rfloor}$, where each row i stores the elements of $\{1, ..., t\}$ belonging to set i.

One of the properties of a 1-CFF using Sperner's theorem is that there is exactly one defective item if and only if $\lfloor t/2 \rfloor$ of the tests are positive. In this case, the decoding algorithm must return true and compute the corresponding defective, namely the item i corresponding to the set of results y. If there are more than $\lfloor t/2 \rfloor$ positive tests, the algorithm returns false and provides a list of defective items, which may also include some non-defective items. This list includes all $\lfloor t/2 \rfloor$-subsets that are contained in the set corresponding to the positive test results in y.

To achieve this, the algorithm first computes all the $\lfloor t/2 \rfloor$-subsets from the set $S = \{j \in \{1, 2, ..., t\} : y_j = 1\}$, where S contains the indices of the positive test results. Then, using Algorithm 2.7 from [24], the lexicographic rank of each subset of size $\lfloor t/2 \rfloor$ within the set S is computed. Therefore, locating defectives for group testing using Sperner set system eliminates the need to compute the actual set system, which consequently saves a lot of time when n is large.

Algorithm 6, referred to as SpernerDecode, is the corresponding decoding algorithm. If at most one item is defective, the running time for Algorithm 6 is $O(t)$. In the worst-case scenario where more than one item is defective, the running time is $O(tn)$.

Algorithm 6 SpernerDecode(y, t, n)

Input: test result y, the number of rows t, the number of columns n
Output: a boolean value indicating correct decoding, the set of defective items I

```
1:  result = list implemented as an array containing positive results;
2:  I = an empty list;
3:  p = an array with size ⌊t/2⌋;                  ▷ to store a subset of Sperner set system
4:  for i = 0 to y.length−1 do
5:      if y[i] == 1 then
6:          result.add(i);
7:      end if
8:  end for
9:  s = result.size();                                ▷ the size of result
10: if s == ⌊t/2⌋ then                                ▷ the case of exactly one defective
11:     for i = 0 to s − 1 do
12:         p[i] = result.get(i);   ▷ compute the subset p among ⌊t/2⌋-subsets of {1, 2, ..., t}
13:     end for
```

```
14:     ▷ calculate the lexicographic rank of p, which is Algorithm 2.7 from [24]
15:     r = rankSubset(p, t/2, t);
16:     I.add(r);
17:     return (true, I);
18: else                                    ▷ the case of more than one defective
19:     ▷ use Algorithm 2.6 from [24] to generate these subsets in lexicographic order
20:     subsets = a list of arrays containing all ⌊t/2⌋-subsets from the set S = {0, 1, ..., s−1};
21:     for every subset in subsets do
22:         ▷ map the subset indices to the corresponding elements in result
23:         for i = 0 to subset.length−1 do
24:             p[i] = result.get(subset[i]);
25:         end for
26:         ▷ calculate the lexicographic rank of p, which is Algorithm 2.7 from [24]
27:         r = rankSubset(p, t/2, t);
28:         if r ≤ n then
29:             I.add(r);
30:         else
31:             break;
32:         end if
33:     end for
34:     return (false, I);
35: end if
```

3.2.2 STS Decoding Algorithm

As mentioned in Sections 2.2.2.2 and 2.2.2.3, we use either the Bose construction for $v \equiv 3 \pmod 6$ or the Skolem construction for $v \equiv 1 \pmod 6$ to build Steiner Triple Systems based on the order v.

Let $S = \{j \in \{1, 2, ..., t\} : y_j = 1\}$, which contains the indices of positive test results from the vector y. By grouping three elements to a valid triple, we are able to detect the corresponding defective. If S is empty, this indicates that all test results are negative, meaning there are no defectives. In this case, the decoding algorithm will return true with an empty defective set I. If there is exactly one defective item, then $|S| = 3$, corresponding to a single block B_{i_1} in the STS. The decoding algorithm will return true with $I = \{i_i\}$. For two defective items, the size of S will be either 5 or 6, representing the union of two distinct blocks $S = B_{i_1} \cup B_{i_2}$ in the STS. The decoding method return true with $I = \{i_1, i_2\}$. If there are more than two defectives, $|S| > 6$ and S represents the union of more than 2 blocks of the STS. In this case, the decoding algorithm will return false with $I = \{i : B_i \subseteq S\}$, with the possibility that some of the items in I are non-defectives.

To locate defective items from Steiner Triple Systems directly, the main idea is that we need to quickly identity the third element of a block containing a pair (i, j), as well as the location of the pair in the STS. To achieve this, we design a more complex data structure to store the triples in a 2-dimensional array, as shown in Algorithm 7. In this structure,

the indices i and j correspond to a pair where $\{i, j\} \subseteq \{1, \ldots, v\}$, and the corresponding entry STS[i][j] is the third element, thereby forming a triple in the STS. Additionally, this algorithm stores the rank (or index) of the triple containing each pair of points. Unlike SPERNERDECODE, which does not require constructing the set system for decoding, the STS decoding algorithm must first compute all the triples and store the information about these triples in advance.

Algorithm 7 storeSTSpair(STS, v)

Input: STS (the STS set system with dimension b by 3), v (order of STS)
Output: STSrank (2D array storing the locations of each pair of triples), STSpair (2D array storing the third element for each pair of triples)

1: STSpair = 2D array with dimension of $v \times v$;
2: STSrank = 2D array with dimension of $v \times v$;
3: **for** $i = 0$ to |STS|−1 **do**
4: STSpair[STS[i][0]][STS[i][1]] = STS[i][2];
5: STSpair[STS[i][1]][STS[i][0]] = STS[i][2];
6: STSpair[STS[i][0]][STS[i][2]] = STS[i][1];
7: STSpair[STS[i][2]][STS[i][0]] = STS[i][1];
8: STSpair[STS[i][1]][STS[i][2]] = STS[i][0];
9: STSpair[STS[i][2]][STS[i][1]] = STS[i][0];
10: STSrank[STS[i][0]][STS[i][1]] = i;
11: STSrank[STS[i][1]][STS[i][0]] = i;
12: STSrank[STS[i][0]][STS[i][2]] = i;
13: STSrank[STS[i][2]][STS[i][0]] = i;
14: STSrank[STS[i][1]][STS[i][2]] = i;
15: STSrank[STS[i][2]][STS[i][1]] = i;
16: **end for**
17: **return** STSpair, STSrank;

Algorithm 8, referred to as STSDECODE, is the algorithm for detecting defectives for 2-CFF from STS. Given the result vector y, the algorithm groups the indices of positive tests into valid triples and ranks them to identify the corresponding defectives. The running time of Algorithm 8 is $O(v^2)$ for cases with one or two defectives. However, if there are more than two defectives, the running time increases to $O(v^3 + mv^2)$, where mv^2 represents the number of times Algorithm 8 goes through duplicate check.

Algorithm 8 STSDecode(y, STS, v)

Input: the test result y, STS (the STS set system), order of STS v
Output: a boolean value indicating correct decoding, the set of defective items I

1: result = list implemented as an array containing positive results;
2: I = an empty list;
3: **for** $i = 0$ to y.length−1 **do**
4: **if** $y[i] == 1$ **then**

```
5:          result.add (i);
6:      end if
7:  end for
8:  s =result.size();
9:  if s = 0 then                                        ▷ no defective items
10:     return True;
11: else if s < 3 or s == 4 then
12:     throw an error "Not a valid result."
13: else
14:     STSpair, STSrank = storeSTSpair(STS, v);    ▷ store the information of pairs
15:     if s <= 6 then                           ▷ one or two defective items
16:         triple = an empty list;
17:         element1 = result[0];                            ▷ the first triple
18:         triple = a list of size 3;
19:         for i = 1 to s − 1 do
20:             element2 = result[i];
21:             element3 = STSpair[element1][element2];
22:             if result.contains(element3) then
23:                 r = STSrank[element1][element2];         ▷ rank the first pair
24:                 I.add (r);
25:                 triple.add (element1, element2, element3);
26:                 result.remove (element1);
27:                 result.remove (element2);
28:                 result.remove (element3);
29:                 break;
30:             end if
31:         end for
32:         if |result| != 0 then                           ▷ the second triple
33:             ele1 = result[0];
34:             ele2 = result[1];
35:             ele3 = STSpair[ele1][ele2];
36:             if ( s == 3 && ele3 == result[2]) || ( s == 2 && triple.contains(ele3)) then
37:                 r = STSrank[element1][element2];        ▷ rank the second pair
38:                 I.add (r);
39:             else
40:                 return throw an error "Not a valid result.";
41:             end if
42:         end if
43:         return (true, I);
44:     else                                     ▷ more than two defective items
45:         triples = a list of integer lists           ▷ to store defective triples
46:         rCopy = a copy of result to keep track of unused elements in result;
```

```
47:         for i = 0 to s − 1 do                ▷ iterate over each possible pair in result
48:             for j = i + 1 to s − 1 do
49:                 first = result[i];
50:                 second = result[j];
51:                 third = STSpair[i][j];
52:                 if result.contains(third) then                  ▷ check if a valid triple
53:                     tripe = an empty list;
54:                     tripe.add(first);
55:                     tripe.add(second);
56:                     tripe.add(third);
57:                     sort(triple);
58:                     if !triples.contain(triple) then              ▷ avoid duplicates
59:                         triples.add(triple);
60:                         rCopy.remove(first);
61:                         rCopy.remove(second);
62:                         rCopy.remove(third);
63:                     end if
64:                 end if
65:             end for
66:         end for
67:         if rCopy.size !=0 then
68:             return throw an error "Not a valid result.";
69:         else
70:             for triple in triples do
71:                 int r = STSrank[triple.get(0)][triple.get(1)];      ▷ rank the pair
72:                 I.add(r);
73:             end for
74:             return (false, I);
75:         end if
76:     end if
77: end if
```

3.2.3 Reed-Solomon Decoding Algorithm

In this section, we illustrate a decoding algorithm for CFFs constructed from Reed-Solomon codes for the case of q a prime number. The construction is first mentioned in Section 2.2.3 and rephrased next.

Construction 1. Let $n > d \geq 2$ be integers. Let q be a prime, $N \leq q$ and $k = \lceil \log_q n \rceil$ such that $d(k-1) \leq N - 1 \leq q - 1$. Define a $q^2 \times q^k$ binary matrix R with rows indexed by $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ and columns indexed by all polynomials f of degree at most $k - 1$, such that $R_{(x,y),f} = 1$ if and only if $f(x) = y$. Take any $X \subseteq \mathbb{F}_q$, $|X| = N$, and only keep the rows (x, y) with $x \in X$. The constructed matrix R has $t = Nq \leq q^2$ rows and q^k columns.

To decode the test result vector y using a CFF constructed from a Reed-Solomon code is equivalent to solving the following problem, which may be of independent interest.

Problem 1. Let n, d and parameters N, q, k satisfying the conditions of Construction 1. Determine the unique set of $s \leq d$ polynomials $f_1, f_2, \ldots, f_s$ of degree $k-1$ over $\mathbb{F}_q$, given the sets of their evaluations on each point in $X \subseteq \mathbb{F}_q$, $|X| = N$, denoted as $S = (S_0, S_1, S_2, \ldots, S_{N-1})$, where $S_x = \{f_1(x), f_2(x), \ldots, f_s(x)\}$, for each $x \in X$.

Now, we demonstrate a recurrence relation to evaluate polynomials over prime fields, which we will use in our decoding algorithm. We begin with a lemma.

Lemma 3. Let k be an integer and use the convention that $0^0 = 1$. Then,

$$\sum_{a=0}^{k}(-1)^a\binom{k}{a}a^i = 0 \text{ for } 0 \leq i < k. \tag{3.2.1}$$

Proof. Consider the case when $i \leq k-1$. We know that $e^{az} = \sum_{j=0}^{\infty}\frac{a^j z^j}{j!}$. Thus, for each a^i, we have:

$$a^i = (i)![z^i]e^{az} \tag{3.2.2}$$

Now consider the equation:

$$\begin{aligned}
& \sum_{a=0}^{k}(-1)^a\binom{k}{a}a^i \\
= \; & \sum_{a=0}^{k}(-1)^a\binom{k}{a}(i!)[z^i]e^{az} \text{ , substituting Eq. 3.2.2} \\
= \; & (-1)^k(i)![z^i]\sum_{a=0}^{k}\binom{k}{a}(e^z)^a(-1)^{k-a} \text{ , rearranging and using } (-1)^{2k-a} = (-1)^a \\
= \; & (-1)^k(i)!\,[z^i](e^z-1)^k \text{ , using binomial expansion} \\
= \; & (-1)^k(i)!\,[z^i](z+\frac{z^2}{2!}+\frac{z^3}{3!}\ldots)^k = ((-1)^k(i)! \times 0) = 0.
\end{aligned}$$

∎

Let p be a prime, $k \geq 2$, and f be a polynomial of degree at most $k-1$ over $\mathbb{F}_p$. The next theorem shows how to recursively evaluate a polynomial f on $N \leq p$ consecutive points $\{0, \ldots, N-1\} \subseteq \mathbb{F}_p$ given their evaluations $f(0), \ldots, f(k-1)$. Indeed, the equations in Lemma 3 are valid in $\mathbb{Z}$, so they are also valid in $\mathbb{Z}_p$, whenever $k < p$.

Theorem 3.2.1. Let p be a prime number and let f be a polynomial over $\mathbb{F}_p$ of degree at most $k-1$. If $2 \leq k < p$, then

$$\sum_{a=0}^{k}(-1)^a\binom{k}{a}f(x-a) = 0, x \in \mathbb{F}_p. \tag{3.2.3}$$

Alternatively, we can express the recursive functions as:

$$f(x) = \sum_{a=1}^{k} (-1)^{a-1} \binom{k}{a} f(x-a). \tag{3.2.4}$$

Proof. Write $f(x) = \alpha_{k-1}x^{k-1} + \cdots + \alpha_1 x + \alpha_0$. As we expand the recursive formula, we have $\sum_{a=0}^{k}(-1)^a\binom{k}{a}f(x-a) = \binom{k}{0}f(x) - \binom{k}{1}f(x-1) + \binom{k}{2}f(x-2) - \cdots + (-1)^k\binom{k}{k}f(x-k)$. Moreover, we have the following set of equations:

$$\begin{aligned}
f(x) &= \alpha_{k-1}x^{k-1} + \alpha_{k-2}x^{k-2} + \alpha_{k-3}x^{k-3} + \cdots + \alpha_1 x + \alpha_0, \\
f(x-1) &= \alpha_{k-1}(x-1)^{k-1} + \alpha_{k-2}(x-1)^{k-2} + \cdots + \alpha_1(x-1) + \alpha_0, \\
&\vdots \\
f(x-k) &= \alpha_{k-1}(x-k)^{k-1} + \alpha_{k-2}(x-k)^{k-2} + \cdots + \alpha_1(x-k) + \alpha_0.
\end{aligned}$$

Substituting these equations into the left-hand side of Eq. 3.2.3, we obtain

$$\sum_{a=0}^{k}(-1)^a\binom{k}{a}f(x-a) = \sum_{j=0}^{k-1}\alpha_j\sum_{a=0}^{k}(-1)^a\binom{k}{a}(x-a)^j.$$

We complete the proof by showing $\sum_{a=0}^{k}(-1)^a\binom{k}{a}(x-a)^j = 0$, $0 \le j < k$:

$$\begin{aligned}
&\sum_{a=0}^{k}(-1)^a\binom{k}{a}(x-a)^j \\
=\; &\sum_{a=0}^{k}(-1)^a\binom{k}{a}\sum_{i=0}^{j}\binom{j}{i}(-a)^i(x^{j-i}) \text{ , using binomial expansion,} \\
=\; &\sum_{i=0}^{j}\binom{j}{i}(x^{j-i})(-1)^i\sum_{a=0}^{k}(-1)^a\binom{k}{a}a^i \text{ , rearranging,} \\
=\; &0, \text{ by Lemma 3.}
\end{aligned}$$

∎

Algorithm FindPolynomials(k, N, q, d, S) given in Algorithm 9 is the algorithm to find all the desired polynomials corresponding to defective elements. After setting up initialization of auxiliary variables in lines 1 to 17, it proceeds to the main loop in lines 18 to 68. The main loop will discover one polynomial f at a time as follows. It first selects $b \in \{0, \ldots, N-1\}$ such that S_b has at least one unused value, i.e. a value $y \in S_i$ that has not been the image of b for any polynomial previously found, and it will fix $f(b) = y$. Then, a set of k consecutive integers in $\{0, \ldots, N-1\}$ starting at value startPos and including b is determined; to simplify the explanation, suppose $b =$ startPos, and the values are $b, b+1, \ldots, b+k-1$. We need to iterate over all combinations of $(f(b+1), f(b+2), \ldots, f(b+k-1)) \in S_{b+1} \times S_{b+2} \times \cdots \times S_{b+k-1}$. Each of these combinations together with $f(b) = y$ determines a unique polynomial of degree at most $k-1$. This polynomial is tested to see if it is valid by

algorithm isValidPolynomial($k, q, N, S, codeword, startPos$) given in Algorithm 10, which uses Theorem 3.2.1 and the chosen k values of $f(b+i)$ to efficiently evaluate the polynomial in the rest of the values $x \in \{0, \ldots, N-1\} \setminus \{b, \ldots, b+k-1\}$. Whenever it is found that $f(x) \notin S_x$ for x in this range, the polynomial is deemed not valid, otherwise it is valid and the values of $f(i)$ are stored in codeword[i] and saved to the list of polynomials listPoly. We repeat this process until there are no unused elements remaining in any of the S_i, which means we have successfully found all the polynomials (corresponding to the defective items).

All the details of this algorithm are given in Algorithms 9 and 10, which refer to other auxiliary algorithms. Algorithms successor and sortList are provided in Algorithm 11 and 12. Algorithm successor(T, m, k) finds the successor of a mixed-radix k-tuple T in lexicographical order in $O(k)$. Algorithm sortList(A, Unused, $k, b, startPos$) rearranges the elements of a d-tuple A_i, $1 \leq i \leq k$ so that the indices of list Unused$_i$ that are true are placed in A_i before the indices that are false; this can be done in $O(kd)$. Note that a lot of bookkeeping is in place to keep track of the unused elements in each S_i, $0 \leq i \leq N-1$, and to accomplish all the tasks efficiently. Keeping track of unused elements in each S_i is useful not only to determine the value b at each iteration but also to employ a heuristic when trying different k-tuples of values for $(f(b+1), f(b+2), \ldots, f(b+k-1))$. Values of S_i can be used by more than one polynomial, i.e. two valid polynomials f_1, f_2 may have $f_1(i) = f_2(i)$; however, even though "repeats" are allowed, our heuristic that decides the order in which values are tried uses unused values of S_i first. This heuristic is accomplished by a rearrangement of unused indices moved to the front in line 43 by calling Algorithm sortList followed by the processing of tuples T in lexicographical order by invoking the algorithm successor(T, m, k) in line 52.

Algorithm 9 FindPolynomials (k, N, q, d, S)

Input: k (the degree of the polynomials is $\leq k-1$), N (codeword length), q (a prime number), d (upper bound on number of polynomials), $S = (S_0, S_1, S_2, \ldots, S_{N-1})$

Output: listPoly (a list of polynomials satisfying the requirements of Problem 1)

1: ▷ **Initialization of auxiliary variables:**
2: Unused = (Unused$_0, \ldots,$Unused$_{N-1}$) a list of boolean lists with the same sizes as in S with all values initialized to true;
3: Count = an array of $N+1$ integers; ▷ Stores the number of unused elements in S_i
4: total = 0;
5: **for** $i = 0$ to $N-1$ **do**
6: Count[i] = $|S_i|$; ▷ Initialize as the current number of unused
7: total + = $|S_i|$;
8: **end for**
9: Count[N] = total; ▷ Count[N] stores the total number of unused elements in S
10: binom1,binom2 = 2 arrays with k integers; ▷ Precomputed binomials
11: **for** $a = 1$ to k **do**
12: binom1[$a-1$] = $(-1)^{a-1}\binom{k}{a}$;
13: binom2[$a-1$] = $(-1)^{a-1}\binom{k}{a-1}$;
14: **end for**
15: listPoly = an empty list of codewords ▷ Store polynomials when they are found;
16: numPolyFound = 0; ▷ Keep track of number of polynomials found

---

```
17:                ▷ Next is the main loop that finds each defective polynomial
18: while Count[N] ≠ 0 do          ▷ There are unused elements, so not all polys found
19:     b = −1;              ▷ Index representing the first S_i that still has unused elements
20:     for i = 0 to N − 1 do                                                    ▷ Find b
21:         if Count[i] ≠ 0 then
22:             b = i; break;
23:         end if
24:     end for
25:          ▷ Find the starting position for k consecutive integers containing b in [0, N − 1]
26:     if b + k ≤ N then
27:         startPos = b;
28:     else
29:         startPos = N − k;
30:     end if
31:          ▷ m[i] stores the number of elements to consider from S_{startPos+i}, for 1 ≤ i ≤ k
32:     m = an array with size k;
33:     for i = 0 to k − 1 do
34:         if i+startPos == b then
35:             m[i] = 1;                                ▷ An unused element of S_b is fixed
36:         else if Count[i+startPos] == d−numPolyFound then
37:             m[i] = count[i+startPos];        ▷ Only unused in S_{startPos+i} are considered
38:         else
39:             m[i] =|S_{i+startPos}|;            ▷ All elements in S_{startPos+i} are considered
40:         end if
41:     end for
42:     A = a list of k arrays A_i of size m[i] containing indices of S;
43:     sortList(A,Unused,k,b,startPos);          ▷ Rearrange indexes with unused in S_i first
44:     T = [0, 0, . . . , 0], an array with size k;        ▷ Iterating over tuples T[i] ∈ [0, m[i]]
45:     codeword = an array with size N;           ▷ codeword corresponding to a polynomial
46:     found = false; hasSuccessorEnd = true;
47:     while (!found) && (hasSuccessor) do
48:         for i = 0 to k − 1 do                               ▷ Get the value from T and A
49:             codeword[i] = S_{i+startPos}.get(A_i[T[i]]);
50:         end for
51:         found = isValidPolynomial(k, q, N, S, binom1,binom2,codeword,startPos);
52:         hasSuccessor =successor(T, m, k);              ▷ T gets next tuple, false no next
53:     end while
54:     if !found then
55:         exit with an error;            ▷ It never happens for correct inputs; must find poly
56:     end if
57:     for i = 0 to N − 1 do              ▷ Poly used f(i)=codeword[i] in S_i, update Unused
58:         index = index of (codeword[i]) in S_i;
59:         if Unused_i[index] then                              ▷ Value used for the first time
60:             Unused_i[index] = false;                                      ▷ Updated Unused
61:             Count[i] = Count[i] − 1;                        ▷ Decrement unused count for S_i
```

```
62:            Count[N] = Count[N] − 1;        ▷ Decrement total count of unused elements
63:        end if
64:    end for
65:    numPolyFound++;
66:    listPoly.add(codeword);        ▷ Store codeword corresponding to polynomial found
67: end while
68: return listPoly;
```

Algorithm 10 isValidPolynomial(k,q,N,S, codeword, startPos)

Input: same as Algorithm 9; global arrays binom1, binom2 computed once in Algorithm 9

Output: a boolean result showing if a codeword of length N is a valid polynomial.

```
1: current = k+ startPos;
2: for i = current to N − 1 do
3:     f = 0;
4:     for j = 1 to k do
5:         previous = i − j;
6:         f+ = binom1[j − 1]× codeword[previous];
7:     end for
8:     if S_i.contains(f) then
9:         codeword[i] =f;        ▷ Update codeword
10:    else
11:        return false;
12:    end if
13: end for
14: current = startPos −1;
15: for i = current to 0 do
16:    f = 0;
17:    flag = i + k;
18:    for j = 0 to k − 1 do
19:        previous = i − j;
20:        f+ = binom2[j]× codeword[flag];
21:        flag−−;
22:    end for
23:    if S_i.contains(f) then
24:        codeword[i] =f;        ▷ Update codeword
25:    else
26:        return false;
27:    end if
28: end for
29: return true;
```

Algorithm 11 successor(T, m, k)

Input: T and m are arrays of size k, where $0 \leq T[i] < m[i]$.

Output: T is modified to contain its successor in lexicographical order, returning true, or returns false if input T is the last.

1: $i = k - 1$;
2: **while** $(i \leq 0)$ && $(T[i] == m[i] - 1)$ **do**
3: $T[i] = 0$;
4: $i = i - 1$;
5: **end while**
6: **if** $i < 0$ **then**
7: return false;
8: **else**
9: $T[i] + +$;
10: return true;
11: **end if**

Algorithm 12 sortList(A, Unused, k, b, startPos)

Input: A (a list of k arrays), Unused (a list of boolean lists), b (an index representing that the first S_i that still has unused elements), startPos (the starting position for k consecutive integers containing b)

Output: A is sorted to contain all unused elements from S at the beginning.

1: **for** $i = 0$ to $k - 1$ **do**
2: offset $=$ startPos $+ i$;
3: $s = |\text{Unused}_{\text{offset}}|$;
4: left $= 0$;
5: right $= s - 1$;
6: **if** offset $== b$ **then**
7: **for** $j = 0$ to $s - 1$ **do**
8: **if** $\text{Unused}_{\text{offset}}[j]$ **then**
9: $A_i[0] = j$;
10: break;
11: **end if**
12: **end for**
13: **else**
14: **for** $j = 0$ to $s - 1$ **do**
15: **if** $\text{Unused}_{\text{offset}}[j]$ **then**
16: $A_i[\text{left}] = j$;
17: left++;
18: **else if** right $\geq |A_i|$ **then**
19: right$--$;
20: **else**
21: $A_i[\text{right}] = j$;
22: right$--$;

```
23:         end if
24:       end for
25:     end if
26: end for
```

It is easy to verify the correctness of Algorithm 9, which gives the next proposition.

Proposition 3.2.2. Let n, d and parameters N, q, k satisfying the conditions of Construction 1, q a prime number. Let $S = (S_0, S_1, S_2, \ldots, S_{N-1})$, where $|S_i| \leq d$ and $S_i = \{f_1(x), f_2(x), \ldots, f_s(x)\}$ for $0 \leq i \leq N-1$. Then Algorithm FindPoynomials(k, N, q, d, S) given in Algorithm 9 correctly solves Problem 1.

Proposition 3.2.3. Algorithm 9 runs in time $O(Nkd^k)$.

Proof. The main loop in line 18 runs at most d times, and we claim that each iteration takes time $O(Nkd^{k-1})$. The crucial part to analyse is the inner loop in lines 47 to 53 that dominates the running time. This inner loop iterates over at most d^{k-1} tuples T in the first iteration of the main loop, at most $(d-1)^{k-1}$ tuples T in the second iteration of the main loop, until iterating over 1 tuple T in the dth iteration; at each iteration of the inner loop one call to isValidPolynomial is made. In total we have less than d^k calls to isValidPolynomial, which in turn runs in $O(Nk)$ (see Algorithm 10), completing the proof. ∎

Finally, we describe Algorithm RSDECODE which computes the defective items for group testing from the d-disjunct matrix constructed in Construction 1. From the list of failing tests it is easy to obtain the sets S_i, for all $1 \leq i \leq N$ in time $O(dN)$. Then, Algorithm 9 is run which returns the list of codewords (storing the evaluation vectors for the polynomials corresponding to defective items), in time $O(Nkd^k)$ (see Proposition 3.2.3). It remains to explain how to compute the defective items from the list of codewords, which stores the evaluation vectors for the polynomials corresponding to defective items. This is accomplished by solving a $k \times k$ system of linear equations to determine the coefficients of each polynomial from its k evaluations. Because the same system of equations is used for all polynomials, this task can be done by doing one matrix inversion and d multiplications of this matrix by a vector of length k, taking time $O(k^3 + dk^2)$. Then, each tuple of length k containing the coefficients of each polynomial is computed, and "unranked" as to obtain its index in $\{1, \ldots, n\}$; each unranking can be done in time $O(k)$, using time $O(dk)$ for all polynomials. In total, the defective items are computed in time $O(k^3 + dk^2)$. Therefore, RSDECODE time is dominated by the time of Algorithm 9, and runs in $O(Nkd^k)$. Note that the time complexity of computing the coefficients of the polynomials corresponding to defective items could be reduced to $O(dk \log k)$ by using Lagrange interpolation with Fast Fourier Transform. However, this would not affect the total running time which is dominated by $O(Nkd^k)$.

3.3 Performance and Comparison

As mentioned earlier, we have a general decoding algorithm for identifying defective items from a d-CFF and it has two data structures: MatrixList and MatrixCompact. Additionally, we have three fast direct decoding algorithms: SpernerDecode, STSDecode, and RSDecode.

With this in mind, our objective in this section is to compare the efficiency of each of the fast direct decoding algorithm for locating defective items in group testing with the general decoding algorithm. All the methods start by receiving a vector $y \in \{0,1\}^t$ with the outcome of the tests, where $y_i = 1$ if and only if test i is positive.

For the experiments, we run all the tests in Java with a laptop running MacOS with 8 GB of random-access memory, and the processor speed reaches a maximum of 3.49 GHz with a M2 chip. Unless specified otherwise, each test is iterated 50 times, the list of defective items is randomly generated for each test and the average data is recorded.

3.3.1 Revisiting the Theoretical Analysis of Decoding Algorithms

As discussed in Section 3.1, the running times for the two data structures used in the general decoding algorithm, MatrixCompact and MatrixList, are $O\left(\frac{tn}{B}\right) = \Theta(tn)$ and $\min(\Theta(Lt\log t + n), \Theta(tn))$, respectively. In Section 3.2, we mentioned that the running time for SpernerDecode is $O(t)$ when $d = 1$; for STSDecode it is $O(t^2)$ when $d = 2$; and for RSDecode it is $O(kNd^k)$ when $d \geq 2$.

For larger values of t, the number of rows t is approximately $\log n$ for Sperner codes. For STS codes, where $n = \frac{t(t-1)}{6}$, we have $t = \frac{1 \pm \sqrt{1+24n}}{2} < \frac{1+\sqrt{25n}}{2} = \frac{1+5\sqrt{n}}{2} \approx \frac{5}{2}\sqrt{n}$. For Reed-Solomon codes, $t = Nq \leq q^2$ and $n = q^k$, and in our theoretical analysis of Reed-Solomon codes, we assume $t = q^2$.

The number of 1's in each row, L, varies depending on the constructions: in the Sperner set systems, $L = \binom{t-1}{\lfloor \frac{t}{2} \rfloor - 1} \approx \frac{1}{2}\binom{t}{\lfloor \frac{t}{2} \rfloor} = \frac{n}{2}$; in STS, $L = \frac{t-1}{2} \approx \frac{5}{4}\sqrt{n}$; and in Reed-Solomon codes, $L = q^{k-1}$.

CFF Construction	t	d	running time		
			MatrixCompact	MatrixList	Specific
Sperner	$\log n$	1	$O(\frac{(n\log n)}{B})$ $= \Theta(n\log n)$	$\min = \Theta(tn)$ $= \Theta(n\log n)$	$O(\log n)$
STS	$\frac{5}{2}\sqrt{n}$	2	$\frac{O(n^{3/2})}{B} = \Theta(n^{3/2})$	$O\left(\left(\frac{5}{4}\sqrt{n}\right) t\log t + n\right)$ $= O\left(\frac{5}{4}\sqrt{n} \cdot \frac{5}{2}\sqrt{n}\log\left(\frac{5}{2}\sqrt{n}\right) + n\right)$ $= O(n\log n)$	$O(n)$
RS codes	q^2	≥ 2	$O\left(\frac{q^{k+2}}{B}\right) = \Theta\left(q^{k+2}\right)$	$O(q^{k-1} \cdot q^2 \log q^2 + q^k)$ $= O(q^{k+1}\log q^2)$ $= O(q^{k+1}\log q)$	$O\left(q \cdot k \cdot (\lfloor \frac{q-1}{k-1} \rfloor)^k\right)$ $= O\left(q \cdot k \cdot \frac{q^k}{k^k}\right)$ $= O\left(\frac{q^{k+1}}{k^{k-1}}\right)$

Table 3.1: Theoretical analysis of decoding algorithms.

When analyzing the decoding performance for different CFF constructions using both the general and specific algorithms, we substitute the values of t and L in terms of n for Sperner and STS, and we use k and q to replace L, t, and n for Reed-Solomon codes. This approach leads to the comparative analysis shown in Table 3.1.

3.3.2 General versus Sperner Decoding Algorithm

This subsection aims to evaluate the practical efficiency of MatrixCompact, MatrixList, and SpernerDecode for $d = 1$.

In Table 3.2, for an increasing number of items, n, ranging from tens to millions, we compute the corresponding number of tests, t, and run the three decoding algorithms with the same result vector y to compare their execution time. The fastest algorithm for each n is highlighted in bold.

n	t	MatrixCompact(ms)	MatrixList(ms)	SpernerDecode(ms)
10	5	0.13	0.07	**0.01**
252	10	0.34	0.43	**0.01**
924	12	0.48	0.89	**0.02**
3,432	14	0.75	1.57	**0.03**
12,870	16	1.36	3.95	**0.02**
48,620	18	3.55	11.39	**0.04**
184,756	20	10.17	42.70	**0.04**
352,716	21	26.02	118.93	**0.04**
705,432	22	47.50	437.78	**0.04**
1,352,078	23	80.77	1,305.84	**0.04**
2,704,156	24	293.23	4,910.11	**0.05**

Table 3.2: Comparison among three decoding methods using Sperner when $d = 1$.

From Table 3.2, we can conclude that when $d = 1$, the SpernerDecode method consistently outperforms the other two general decoding methods, MatrixCompact and MatrixList. Additionally, except for the case when $n = 10$, where MatrixList is faster than MatrixCompact, MatrixCompact generally has a shorter execution time compared to MatrixList. Furthermore, as n increases, the execution time for SpernerDecode remains relatively stable since t does not increase significantly. This stability is due to the SpernerDecode method depending on the size of the result vector y, which is $t \sim \log n$. However, this is not the case for MatrixCompact and MatrixList, whose execution times rise significantly with increasing n. The results in the table match with the theoretical analysis in Section 3.3.1. SpernerDecode is always faster than the general decoding algorithms, with a time complexity $O(\log n)$ while the general algorithms have time complexity $\Theta(n \log n)$. Additionally, MatrixList is the slowest in practice (except for very small $n \leq 10$). Even though the two general algorithms run in $\Theta(n \log n)$, we use $B = 64$ for MatrixCompact, resulting in time $\frac{1}{64} n \log n$, which has a small constant in front of the higher order term. This shows that the specific decoding algorithm generally works better than the general ones, and MatrixCompact is the better choice of general decoding.

3.3.3 General versus STS Decoding Algorithm

This subsection aims to evaluate the efficiency of MatrixCompact, MatrixList, and STSDecode for $d = 2$.

Similar to section 3.3.2, we also increase the number of items, n, from tens to millions in Table 3.3. The number of tests, t, increases accordingly, and we compare the time used for MatrixCompact, MatrixList, and STSDecode when $d = 2$. The fastest algorithm for different n is again highlighted in bold.

n	t	MatrixCompact(ms)	MatrixList(ms)	STSDecode (ms)
12	9	0.12	0.13	**0.06**
117	27	0.25	0.36	**0.13**
610	61	0.47	0.84	**0.34**
1,027	79	**0.48**	1.18	0.59
4,510	165	**0.74**	2.11	1.10
10,292	249	**0.85**	4.05	1.34
58,905	595	**2.90**	15.65	3.10
101,530	781	5.31	23.83	**5.21**
376,251	1,503	24.60	95.92	**19.33**
1,010,651	2,46	158.70	378.26	**64.68**
2,042,250	3,50	716.63	1,014.67	**166.92**

Table 3.3: Comparison among three decoding methods using STS when $d = 2$.

According to Table 3.3, the execution time for all three decoding methods increases as the value of n rises. Throughout all values of n in the table, MatrixList consistently appears to be the slowest method among the three methods. For n values ranging from tens to hundreds, STSDecode is the fastest decoding method. However, as n ranges from thousands to ten-thousand, MatrixCompact becomes the quickest. Beyond this range, STSDecode reclaims its position as the fastest method again. The tradeoff in decoding performance highlights that Big-O notation does not always capture the complete story. Even though STSDecode has an asymptotically advantage with linear complexity $O(n)$ while the general methods have complexities of $O(n^{3/2})$ and $O(n \log n)$, it is not always the fastest. This is because Big-O notation focuses only on the dominant term, ignoring constants, lower-order terms, and practical components like the value of B for MatrixCompact (B=64 in our case). These "hidden" factors can significantly impact actual running time for practical values, showing why both theoretical analysis and experimental evaluation are necessary to understand algorithm performance. Moreover, as shown in Section 3.3.1, MatrixList asymptotically performs better than MatrixCompact. However, this advantage is not reflected in our table because the values of n are not large enough to highlight the benefits of using MatrixList. As a result, one should consider using either STSDecode or MatrixCompact depending on the specific value of n. In conclusion, MatrixCompact could be used for $n \leq 100{,}000$, after which the advantage of using STSDecode becomes more evident.

3.3.4 General versus Reed-Solomon Decoding Algorithm

In this subsection, we run tests for comparing the efficiency between MatrixCompact, MatrixList, and RSdecode for various ranges of $d \geq 2$.

Table 3.4 presents four ranges for n: hundreds, thousands, tens of thousands, and hundreds of thousands, respectively. Within each range, we start with $d = 2$ and gradually increase it. We select combinations of d and n, and optimize the other parameters k, N, q to minimize t. More precisely, minimize $t = Nq$ subject to the constraints that q is prime, $k \geq 2$, $q^k \geq n$, $N = d(k-1) + 1 \leq q$, by inspection.

For each combination, we measure the time taken by each method in nanoseconds and the winning time is highlighted in bold.

In Table 3.4, each row is the average of 50 different random tests, except for the last row, where we only run 10 times, due to the slow running times of the naive algorithms.

d	n	k	N	q	t	MatrixCompact(ms)	MatrixList(ms)	RSdecode(ms)
2	125	3	5	5	25	**0.11**	0.42	0.21
3	121	2	4	11	44	**0.05**	0.35	0.25
5	121	2	6	11	66	**0.04**	0.42	0.47
2	2,401	4	7	7	49	**0.15**	1.46	0.25
3	1,331	3	7	11	77	**0.18**	1.47	0.36
5	1,331	3	11	11	121	**0.14**	1.72	0.81
15	1,369	2	16	37	592	**0.24**	2.02	1.01
20	1,369	2	21	37	777	**0.25**	2.18	1.59
2	14,641	4	7	11	77	**0.21**	4.37	0.28
3	14,641	4	10	11	110	**0.21**	6.27	0.46
15	29,791	3	31	31	961	**0.43**	28.30	2.24
20	68,921	3	41	41	1,681	**1.04**	81.47	2.50
35	10,201	2	36	101	3,636	**0.53**	20.51	3.53
2	161,051	5	9	11	99	0.47	41.15	**0.31**
3	371,293	5	13	13	169	2.58	139.76	**0.65**
15	103,823	3	31	47	1,457	**1.31**	105.00	1.86
25	148,877	3	51	53	2,703	88.37	319.97	**4.02**
45	912,673	3	91	97	8,827	212.02	8,334.31	**10.01**

Table 3.4: Comparison among three decoding methods with different d.

From Table 3.4, several key observations can be made. First, MatrixCompact always performs better when compared to MatrixList for the parameters in the table. Additionally, RSdecode is faster than MatrixList in all cases except when $d = 5$ and $n = 121$. Moreover, when n is between 100 and 100,000, MatrixCompact is always the fastest. However, beyond this range, RSdecode achieves the shortest execution time. Meanwhile, while the execution time of MatrixCompact and MatrixList increases as n increases, this trend is not observed in RSdecode. Even with larger values of n, RSdecode can perform better with a smaller value of d. This behavior can be explained due to the fact that the execution time for RSdecode depends on d^k, rather than $n = q^k$. Based on the results from Section 3.3.1, MatrixList is expected to perform faster than MatrixCompact; however, this is not observed in our experiments. This discrepancy could be due to

MatrixCompact having a lower constant factor in its runtime ($\frac{1}{B} = \frac{1}{64}$), which positively impacts its practical performance, specifically for the smaller range of values for n where it outperforms RSDecode. On the other hand, the performance of RSDecode aligns with the theoretical prediction, as it demonstrates the fastest decoding times for large values of k and q. In summary, during group testing, one should consider using MatrixCompact to locate defectives when n is relatively small. Conversely, as n grows larger, RSDecode becomes the best choice among the three methods.

Theorem 2.2.15 shows how we can obtain an asymptotically small t that is polynomial on d and $\log n$. Table 3.5 uses parameters derived from Theorem 2.2.15. We know that:

$$q \approx \frac{2 \times d \times \log n}{\log d} \approx \frac{2 \times d \times \log q^k}{\log d} \approx \frac{2 \times d \times k \times \log q}{\log d}. \tag{3.3.1}$$

Moving $\log q$ to the other side in Equation 3.3.1 results in the following:

$$\frac{q}{\log q} \approx \frac{2 \times d \times k}{\log d}. \tag{3.3.2}$$

Hence, in Table 3.5, our test approach is to compare the three algorithms with parameters chosen as follows: first, we set $k = 3$, and select values of d to be 2, 3, 5. We then calculate N as $N = d(k-1)+1$, and determine q using Equation 3.3.2. After that, we compute t and n from these three parameters and conduct tests. Again, the winning algorithm for each set of parameters is highlighted in bold.

d	k	N	q	n	t	MatrixCompact(ms)	MatrixList(ms)	RSDecode(ms)
2	3	5	79	493,039	395	1.75	116.84	**0.25**
5	3	11	83	571,787	913	89.72	412.07	**0.55**
8	3	17	107	1,225,043	1,819	182.86	6,996.29	**0.95**

Table 3.5: Performance of three methods using parameters from Theorem 2.2.15.

The results from Table 3.5 align with the observations from Table 3.4. As n is relatively large, RSDecode always takes the least amount of time to locate defective items, while MatrixList is always the slowest among the three. Moreover, as n grows, the execution time difference between RSDecode and the other two methods also increases. Furthermore, the value of t in this table remains small with large values of n, which successfully achieves our goal of maintaining t small.

3.3.5 Performance of RSDecode

Both Sperner and STS have a fixed parameter d, and the number of tests t can be directly calculated from the number of items n. In contrast, Reed-Solomon codes work for all $d \geq 2$, and involves more parameters. As a result, this section aims to analyze the influence of parameter choices on the efficiency of RSDecode. It consists of three parts describing different approaches to setting up parameters, followed by performance testing based on these parameters.

3.3.5.1 Parameters from Theorem 2.2.15

In this section, we adopt the same test approach as Table 3.5 by using Theorem 2.2.15. Nevertheless, unlike before, we are no longer restricting ourselves to n around 1,000,000, since we do not have to test the time-consuming naive algorithms. Moreover, on top of setting $k = 3$, we also introduce two more parts, by setting k to be 5 and 7, respectively. In order to evaluate the performance, we not only test the execution time, but also count the numbers of times the algorithm calls "isValidPolynomial" method (column labeled "#calls ivp"), as detailed in Section 3.2.3.

d	k	N	q	n	t	time (ms)	#calls ivp
2	3	5	79	493,039	395	0.25	2.0
5	3	11	83	571,787	913	0.55	6.18
8	3	17	107	1,225,043	1,819	0.95	16.28
15	3	31	173	5,177,717	5,363	2.16	75.98
25	3	51	263	18,191,447	13,413	3.84	378.76
40	3	81	389	58,863,869	31,509	8.35	1,999.92
2	5	9	149	73,439,775,749	1,341	0.49	2.0
5	5	21	157	95,388,992,557	3,297	0.95	13.62
8	5	33	211	418,227,202,051	6,963	2.30	91.64
15	5	61	331	3,973,195,810,651	20,191	4.80	2,677.46
25	5	101	479	25,216,079,618,399	48,379	11.41	34,411.2
40	5	161	719	192,151,797,699,599	115,759	59.41	604,302.26
2	7	13	223	27,424,204,663,190,048	2,899	0.47	2.0
5	7	31	239	44,543,599,279,432,080	7,409	1.54	68.82
8	7	49	311	281,399,112,371,155,264	15,239	3.82	3,851.14
15	7	91	479	5,785,602,523,725,084,672	43,589	14.58	87,974.48
25	7	151	719	9.9334985e19	108,569	518.31	9,676,274.02
40	7	241	1,061	1.513588e21	255,701	8,499.03	1.017418919e8

Table 3.6: Performance of RSDECODE using parameters from Theorem 2.2.15.

Table 3.6 illustrates the advantages of using RSDECODE. Unlike the general algorithms, which struggle with large datasets, RSDECODE can detect defectives with extremely large values of n while keeping a relatively small t. In the table, the number of calls to ivp tends to increase as d increases. In the worst-case scenario, we would expect to have a maximum of d^{k-1} calls to find the first polynomial, followed by $(d-1)^{k-1}$ calls for the second polynomial, and so on. However, the average number of calls to ivp never even reaches d^{k-1}, indicating that RSDECODE can likely perform much better than the worst-case scenario. Additionally, the largest execution time recorded in these tables is about 8.5 seconds when $n = 1.513588 \times 10^{21}$, which is quite reasonable. Therefore, RSDECODE proves to be capable of handling large datasets. On top of that, the difference in k across the table affects the number of calls. As k increases, so does the number of calls to ivp and the running time.

3.3.5.2 Parameters giving optimal t given n and d

As previously discussed in Section 3.3.4, the parameters k, q, and N presented in Table 3.4 are derived from fixing d and n and then determining which combination yields the smallest value of t. From Table 3.4, we notice that as n increases, the execution time for RSDECODE also increases. Nevertheless, this increase is significantly slower compared to the growth of n. In the table, n ranges from 100 to approximately 900,000, indicating a growth of 9000 times. In contrast, the execution time of RSDECODE only grows by a factor of 50.

d	k	N	q	n	t	time (ms)	#calls ivp
3	5	13	13	371,293	169	0.54	4.76
3	5	13	59	714,924,299	767	0.58	3.22
3	5	13	97	8,587,340,257	1,261	0.65	3.94
3	6	11	11	1,771,561	121	0.57	6.26
3	6	11	59	13,841,287,201	539	0.63	5.46
3	6	11	97	832,972,004,929	1,067	0.62	4.0
3	7	19	19	893,871,739	361	0.74	10.86
3	7	19	59	2,488,651,484,819	1,121	0.78	8.48
3	7	19	97	80,798,284,478,113	1,843	0.86	5.3
3	8	22	23	78,310,985,281	506	0.89	10.98
3	8	22	59	146,830,437,604,321	1,298	0.94	9.74
3	8	22	97	7,837,433,594,376,961	2,134	1.05	7.96
8	3	17	37	50,653	629	0.89	27.68
8	3	17	107	1,225,043	1,819	0.95	16.28
8	3	17	211	9,393,931	3,587	0.89	11.36
8	5	33	37	69,343,957	1,221	2.09	549.04
8	5	33	211	418,227,202,051	6,963	2.30	91.64
8	5	33	311	2,909,390,022,551	10,263	1.74	54.72
8	7	49	53	1,174,711,139,837	2,597	4.13	9,760.5
8	7	49	311	281,399,112,371,155,264	15,239	3.82	3,851.14
8	7	49	503	8,146,590,668,153,304,064	24,647	3.71	1,623.4
40	3	81	89	704,969	7,209	8.15	6,017.4
40	3	81	389	58,863,869	31,509	8.35	1,999.92
40	3	81	503	127,263,527	40,743	8.30	1,227.98
40	5	161	163	115,063,617,043	26,243	322.62	4,610,179.46
40	5	161	311	2,909,390,022,551	50,071	158.32	2,123,141.94
40	5	161	719	192,151,797,699,599	115,759	59.41	604,302.26
40	7	241	1061	1.513588e21	255,701	8,499.03	1.017418919e8
40	7	241	1297	6.1741878e21	312,577	6,214.73	9.78794511e7
40	7	241	2857	1.5537162e24	688,537	2,044.79	3.39832409e7

Table 3.7: Performance of RSDECODE with different parameters.

3.3.5.3 Analysing the effect of different parameters on the running time

Based on Table 3.7, it is evident that as d increases, the number of calls to ivp also grows. On the other hand, as q grows for the same set of d, k and N, we observe that the numbers of calls to ivp decreases, resulting in shorter execution time.

We are not quite sure what is the reason, but it appears that larger fields are making us get faster to the combinations of k-tuples in $S_p \times S_{p+1} \times \ldots \times S_{p+k-1}$ that leads to the right polynomials, causing less calls to ivp. This is an interesting observation with respect to Problem 1 in Section 3.2.3 that could be further investigated.

This section aims to analyze the performance of RSDECODE by choosing combinations of d and k. In Table 3.7, we choose d to be 3, 8 and 40, and we select k between 3 and 8, and determine N as $N = d(k-1)+1$. After that, we choose increasing values of q bigger than N to run our tests, making $n = q^k$ grow. Again, all the tests are measured in both execution time and number of calls.

3.3.6 STS vs Reed-Solomon Decoding Algorithm when $d = 2$

Since we have two different constructions for 2-CFF, namely Steiner Triple System and Reed-Solomon codes, this section first compares the general decoding algorithms for these two constructions: MATRIXLIST and MATRIXCOMPACT. It then compares the specific decoding algorithm: STSDECODE and RSDECODE. The comparison focuses on two major factors: the execution time and the number of tests t.

In Table 3.8 and Table 3.9, we start with $n = 100$. Each time we increase n by 4 times until n surpasses a million. For every n, we run the algorithms to obtain the number of tests, and the execution time.

n	STSLIST		STSCOMPACT		RSLIST		RSCOMPACT	
	t	time (ms)	t	time (ms)	t	time (ms)	t	time (ms)
100	25	0.37	25	0.12	25	0.23	25	0.06
400	51	0.77	51	0.32	49	0.87	49	0.07
1,200	87	1.45	87	0.39	49	1.05	49	0.11
4,800	171	2.56	171	0.91	77	2.25	77	0.30
19,200	343	6.40	343	1.57	91	5.90	91	1.18
76,800	681	19.87	681	4.14	99	23.71	99	3.79
307,200	1,359	78.46	1,359	24.37	117	80.08	117	5.42
1,228,800	2,719	481.89	2,719	283.45	121	554.90	121	10.21

Table 3.8: STS vs RS in matrix format when $d = 2$.

In Table 3.8, we display the performance of the four general decoding methods for 2-CFFs from STS and RS. From this table, as n increases, the execution times for STSLIST and RSLIST are very similar, even though t for STS is significantly larger than t for RS. According to Section 3.3.1, however, RSLIST tends to be slower than STSLIST as n grows.

This is because the running time of RSList is $O(q^{k+1}\log q) = O(q^k q\log q) = O(nq\log q) = O(n\frac{n^{1/k}}{k}\log n)$, whereas for STSList it is $O(n\log n)$. Given that $n \gg k$, the additional $\frac{n^{1/k}}{k}$ factor makes RSList less efficient than STSList. This trend is evident in the table as well, where RSList loses its advantage when n reaches 1,228,800. However, RSCompact is significantly faster than STSCompact. This is due to their time complexities: RSCompact has a complexity $O(q^{k+2}) = O(q^k q^2) = O(nq^2)$ while STSCompact has $O(n^{3/2})$. Since $q^2 = n^{2/k} \ll \sqrt{n}$ as n increases, RSCompact will decode much faster than STSCompact for large values of n. Additionally, the value of t increases much faster in both STSList and STSCompact compared to RSList and RSCompact. Since minimizing the number of tests, t, is a crucial factor in group testing, RSCompact should be considered for $d = 2$ when using the general decoding algorithm.

n	STSDecode		RSDecode	
	number of tests (t)	time (ms)	number of tests (t)	time (ms)
100	25	0.09	25	0.23
400	51	0.25	49	0.25
1,200	87	0.62	49	0.26
4,800	171	1.02	77	0.28
19,200	343	1.83	91	0.25
76,800	681	3.95	99	0.34
307,200	1,359	15.41	117	0.31
1,228,800	2,719	80.05	121	0.34

Table 3.9: STSDecode vs RSdecode when $d = 2$.

Table 3.9 displays the number of tests, t and the execution time of STSDecode and RSDecode. Similar to Table 3.8, the value of t increases much more significantly in STSDecode when compared to RSDecode. Moreover, when n is less than thousands, STSDecode is faster, but it becomes slower beyond this range. Furthermore, as n rises, the execution time for STSDecode also increases, whereas the execution time for RSDecode remains very stable. As mentioned earlier, the stability is due to the fact that the decoding time of RSDecode depends on d^k, not $n = q^k$. Since $d = 2$ is fixed, so the execution does not rise significantly. Moreover, from Section 3.3.1, the running time for STSDecode and RSDecode are $O(n)$ and $O(\frac{q^{k+1}}{k^{k-1}}) = O(q^k \frac{q}{k^{k-1}}) = O(n\frac{q}{k^{k-1}})$, respectively. Since $d = 2 \leq \lfloor\frac{q-1}{k-1}\rfloor$, q is approximately twice as large as k, the running time for RSDecode simplifies to $O(n\frac{2}{k^{k-2}})$. As k grows, this results in a substantial reduction in execution time relative to STSDecode. Consequently, for larger n, RSDecode performs more efficiently. In conclusion, RSDecode is the better option for specific decoding algorithm when $d = 2$.

Chapter 4

Digital Signatures

This chapter first presents conventional digital signature schemes, which include both classical and post-quantum schemes. The term *conventional* here refers to the standard, widely accepted cryptographic methods that are foundational to digital security practices. Classical schemes use well-known cryptographic techniques such as RSA and ECC (Elliptic Curve Cryptography), relying on mathematical problems that are currently computationally hard to solve using classical computers. On the other hand, post-quantum schemes are designed to be secure against quantum computers. Additionally, this chapter discusses modification-tolerant signature schemes (MTSS) and a specific d-MTSS that allows up to d modifications in a signed message, which is divided into n blocks. This is achieved by using non-adaptive combinatorial group testing mentioned in Chapter 2. Such signatures are particularly useful in scenarios where documents require updates or redactions while still maintaining the validity of the original signature.

The information presented in Section 4.1 is mainly derived from the book [22], while the content in Section 4.2 is based on the articles [20, 21].

4.1 Conventional Digital Signature Schemes

A conventional digital signature scheme (CDSS) is a cryptographic scheme used to verify the authenticity and integrity of digital messages. A CDSS always consists of three probabilistic polynomial-time algorithms $\Pi =$ (Key Generation, Sign, Verify), each serving a purpose in the signature process:

1. **CDSS-Key-Generation** (l): The key generation algorithm accepts a security parameter l and outputs a key pair (SK, PK). The private/secret key (SK) is kept secret by the signer, while the public key (PK) is shared with parties requiring signature verification. Both keys are of sufficient length, at least l bits.

2. **CDSS-Sign** (m, SK): The signing algorithm takes a message (m) and a private key (SK) as input and outputs a digital signature (σ).

3. **CDSS-Verify** (m, PK, σ): The verification algorithm takes input a message (m), a digital signature (σ), and a public key (PK), and outputs a boolean value indicating the validity of the signature. If the pair (m, σ) is valid, the output is 1; otherwise, it is 0.

Definition 4.1.1. Let Π be a CDSS as defined above, and let (SK, PK) be a pair of secret and public key. A pair of message and signature (m, σ) is valid if σ is a valid output of Π.Sign (m, SK).

An *existential forgery* occurs when an adversary is able to create at least one valid message/signature pair that has never been signed by the legitimate signer. An *adaptive chosen message attack* refers to a situation where the adversary has access to an oracle that signs messages of their choosing, and they adaptively select further messages based on the signatures received. Moreover, a function $f\colon \mathbb{N} \to \mathbb{R}$ is considered negligible if, for every polynomial p, there exists an N such that for all $n > N$, $f(n) < \frac{1}{p(n)}$. *Existential unforgeability* under an adaptive chosen message attack ensures that no probabilistic polynomial-time adversary can create such a forgery with non-negligible probability. The formal definition follows.

Definition 4.1.2. [20] A digital signature scheme is *existentially unforgeable* under an adaptive chosen message attack if there is no probabilistic polynomial-time adversary that can produce an existential forgery with non-negligible probability.

In many digital signature schemes, cryptographic hash functions play a crucial role in addressing limitations related to data size since they can compress any data into a fixed-length hash value. Most digital signature schemes are designed to sign relatively small amounts of data, typically ranging from tens to a few hundred bytes. To overcome this constraint, a hash function is integrated into the process to reduce the message to a smaller size (usually 32 or 64 bytes), which can then be signed in a single step.

As a result, in practice, an input message is always hashed using a hash function before signing. Even though the scheme in turn signs the hash value rather than the original message, this approach is just as secure as signing the message itself. Without the use of a hash function, signing long messages would require breaking them into small chunks and signing each piece individually, which could be time consuming.

In the following subsections, we first discuss cryptographic hash functions. Then, we talk about some well-known classical digital signatures as well as post-quantum digital signatures.

4.1.1 Cryptographic Hash Functions

A hash function, denoted as H, is a mathematical algorithm designed to take an input and compresses it into a fixed-length hash value, commonly referred as a *message digest*. One of the key attributes of a hash function is its ability to minimize collisions. In this context, a collision is defined as a pair of distinct elements, x and x', for which the hash function produces identical hash values, i.e., $H(x) = H(x')$.

In data structures, non-cryptographic hash functions are constructed with the goal of minimizing collisions. These hash functions are used to efficiently organize and access data in data structures like hash tables and dictionaries. However, when it comes to cryptography, the requirement for hash functions extends beyond collision minimization: they must be collision-resistant. Collision-resistance in cryptography means that it is computationally hard for any probabilistic polynomial-time algorithm to find a collision in H. In other words, it should be infeasible for an adversary to find two distinct inputs that produce the same hash value. With that being said, constructing collision-resistant hash functions is significantly more challenging compared to those used in data structures. Collision-resistant hash functions are widely used in both public-key and private-key cryptography. In the subsequent sections, we introduce some widely-used cryptographic hash algorithms.

4.1.1.1 SHA-2

SHA-2, introduced by the National Institute of Standards and Technology (NIST) in 2001, represents a significant advancement in cryptographic hash functions. It is built by a collision-resistant compression function from a block cipher with strong security attributes. Such a compression function is then further developed into a fully functional hash function. Initially, this approach involves applying the Davies–Meyer construction [35] to some underlying block cipher to obtain a fixed-length hash function, and then extending to a complete hash function through Merkle-Damgård transform [6, 31]. The length of the digest is equal to the block length of the block cipher.

The SHA-2 family includes six hash functions with four different digest sizes: SHA-2-224 and SHA-2-512/224 (224-bit digests), SHA-2-256 and SHA-2-512/256 (256-bit digests), SHA-2-384 (384-bit digest), and SHA-2-512 (512-bit digest). SHA-2 is currently considered secure and is one of the most popular hash functions used in cryptography. It has been involved in numerous security protocols, such as Transport Layer Security, Internet Protocol Security.

4.1.1.2 SHA-3

SHA-3 was the winner of the NIST hash function competition and was released by NIST in 2015. Unlike SHA-2, SHA-3 does not use the Merkle-Damgård transform. Instead, it uses an unkeyed permutation P with a large block size of 1600 bits called Keccak. Without the use of any compression function, it uses the so-called "sponge construction" to build a hash function from Keccak directly. If P is modeled as a random permutation, the resulting hash function can be proven to be collision-resistant.

Similar to SHA-2, SHA-3 has different versions like SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256. The first four give hash values of specific lengths: 224 bits, 256 bits, 384 bits, and 512 bits, respectively. The last two can produce hash values of any length. It is important to note that SHA-3 is not meant to replace SHA-2. Instead, it is a good alternative.

4.1.2 Classical Digital Signatures

Many classical digital signature schemes have been developed based on hardness assumptions on mathematical problems. We mainly discuss two of these assumptions: the RSA problem and the Discrete-Logarithm problem. Additionally, we also introduce two digital signature schemes made from these hard problems: RSA-based signature and elliptic curve digital signature algorithms (ECDSA).

4.1.2.1 RSA Problem

RSA (Rivest-Shamir-Adleman) is a popular public-key cryptography known for its use in encryption and key exchange. Additionally, it can also be used for creating digital signatures. RSA-based signature schemes is based on the RSA problem, which involves the properties of large prime numbers and modular exponentiation.

Mathematically, the RSA problem can be defined as follows:

1. Select two large prime numbers, p and q.
2. Compute their product, $N = p \cdot q$, which is the RSA modulus.
3. Choose a public exponent e, typically a small prime number.
4. Compute the private exponent d such that $e \cdot d \equiv 1 \pmod{((p-1)\cdot(q-1))}$.

Below is the scheme of plain RSA-based signatures:

1. **Key-Generation(l)**: The key generation algorithm accepts a security parameter l, which is the bit length of the modulus N, and outputs a key pair (SK, PK). The PK is (N, e) and the SK is (N, d).
2. **Sign**(m, SK): The signing algorithm takes a message m and a private key $SK = (N, d)$ as input and outputs a digital signature $\sigma = m^d (\text{mod } N)$.
3. **Verify**(m, PK, σ): The verification algorithm takes a message m, a digital signature σ, and a public key $PK = (N, e)$ as input, and outputs 1 if and only if $m = \sigma^e (\text{mod } N)$.

The security of the RSA encryption algorithm relies on the difficulty of solving this problem. In practice, it is computationally infeasible to factorize the RSA modulus N and find the private exponent d if the prime factors p and q are sufficiently large. This difficulty forms the basis for RSA's security, as breaking RSA encryption would require solving the RSA problem, which is currently considered a hard computational problem.

However, a plain RSA signature scheme is considered insecure and vulnerable to forgery, particularly under a no-message attack, using only the public key (N, e). For example, an adversary can select an arbitrary number, say r, and set the message to be $m \equiv r^e (\text{ mod } N)$. As a result, the forged signature r will be valid for the message m, even though the attacker never had the private key (N, d) or access to a legitimate signature.

To address the issue, it is common to use a cryptographic hash function to hash the input message before signing. However, even if we directly sign a hash value with RSA digital signature scheme, we are still open to some attacks, like the Desmedt-Odlyzko chosen text attack [7]. Therefore, in real-world scenarios, padding is applied to the hash value before computing an RSA signature. Padding schemes, such as *PKCS#1* or *RSA-PSS*, add randomness and structure, making it impractical for adversaries to exploit the signature process.

4.1.2.2 Discrete-Logarithm Problem

Another set of signature schemes is constructed using the discrete logarithm problem. The problem can be stated in general as follows: Let $\mathbb{G}$ be a cyclic group of order q with generator g. Given an element $h \in \mathbb{G}$, the discrete logarithm problem is to find $x \in \mathbb{Z}_q$ such that $h = g^x$. For large q, this problem is usually computationally hard.

A well-known signature scheme based on the discrete logarithm problem is the *Elliptic Curve Digital Signature Algorithm* (ECDSA). This scheme relies on the discrete logarithm problem specific to elliptic curves and is part of the *Digital Signature Standard* (DSS) defined by NIST. Let $\mathcal{G}$ be a polynomial-time algorithm that, given parameter l as input, outputs a cyclic group $\mathbb{G}$, its order q (where the bit length of q is l), and a generator g. The DSS consists of three main algorithms:

1. **Key-Generation(l)**: The key generation algorithm accepts a security parameter l, the bit length of q, and run $\mathcal{G}(1^l)$ to obtain $(\mathbb{G}, q, g)$. Choose $x \in \mathbb{Z}_q$ uniformly at random and set $y = g^x$, outputs a key pair (SK, PK). The public key PK is $(\mathbb{G}, q, g, y)$, and the private key SK is x. Additionally, two functions are specified: H: $\{0,1\}^* \to \mathbb{Z}_q$ and F: $\mathbb{G} \to \mathbb{Z}_q$. These functions are modeled as random oracles.

2. **Sign(m, SK)**: The signing algorithm takes a message m, a private key SK as input and outputs a digital signature $\sigma = (r, s)$, where:

   - k is a random integer chosen from $\mathbb{Z}_q^*$.
   - $r = F(g^k)$. If $r = 0$, choose a new k.
   - $s = k^{-1} \cdot (H(m) + xr) \bmod q$. If $s = 0$, choose a new k.

3. **Verify(m, PK, σ)**: The verification algorithm takes a message m, a signature $\sigma = (r, s)$ (with $r, s \neq 0 \mod q$), and a public key PK as input. It outputs 1 if and only if $r = F(g^{H(m)\cdot s^{-1}} y^{r \cdot s^{-1}})$.

In ECDSA, the problem involves choosing a random number x such that $y = x \cdot G$, where G is a known base point on the elliptic curve that generates a subgroup of large prime order p, and y is the public key point on the curve. Thus, G in ECDSA corresponds to the generator g in DSS, and $y = x \cdot G$ is analogous to $y = g^x$ in DSS, with x as the private key in both cases. Several widely used elliptic curves in cryptography include secp256k1, E-521, and NIST P-256.

The main advantage of ECDSA over RSA is that ECDSA keys are shorter for the same security level [41]. Table 4.1 shows the NIST recommended key sizes between RSA and ECDSA with the same security level. For example, a 256-bit ECDSA signature offers a security strength similar to a 3072-bit RSA signature. As a result, ECDSA is a better choice when it comes to limited resources.

level of security (bits)	RSA (bits)	ECDSA (bits)
112	2048	224
128	3072	256
192	7680	384

Table 4.1: RSA versus ECDSA.

4.1.3 Post-quantum Digital Signatures

All the digital signatures mentioned in Section 4.1.2 rely on the difficulty of either integer factorization or the discrete logarithm problem for their security. However, they are vulnerable to quantum computers, which can efficiently solve these problem using algorithms like Shor's Algorithm [37]. As a result, the need for post-quantum digital signatures has become critical.

In 2016, NIST launched the Post-Quantum Cryptography Standardization initiative, seeking algorithms for digital signatures, public key encryption, and key establishment mechanisms. By the end of 2017, NIST had received 69 submissions, including approximately 20 digital signature schemes, which were evaluated in the first round. These submissions included various types, including lattice-based, code-based, hash-based, multivariate system of equations, and others. In the second round, the field was narrowed to nine digital signature schemes. In the third round, NIST selected three finalists for digital signatures: CRYSTALS-Dilithium (lattice-based), FALCON (lattice-based), and Rainbow (multivariate-based), with SPHINCS$^+$ (hash-based), GeMSS (multivariate-based), and PICNIC (based on zero-knowledge proofs) as alternatives. However, in 2022, due to a cryptanalytic attack, NIST dropped Rainbow from the competition and focused on SPHINCS$^+$, FALCON, and CRYSTALS-Dilithium. On August 13, 2024, NIST officially announced three approved Federal Information Processing Standards (FIPS) for post-quantum cryptography, including FIPS 204 for CRYSTALS-Dilithium and FIPS 205 for SPHINCS$^+$. NIST is also developing a FIPS for FALCON and continues to evaluate other alternatives.

In this thesis, we mainly focus on the selected digital signature algorithms: SPHINCS$^+$, FALCON and CRYSTALS-Dilithium.

4.1.3.1 SPHINCS$^+$

We first introduce SPHINCS$^+$ (stateless hash-based signature scheme), which relies on the security properties of cryptographic hash functions rather than mathematical problems. Lamport [25] introduced the concept of a one-time hash-based signature scheme in 1979, which

was later extended to a few-time signature (FTS) scheme using a binary hash tree called Merkle tree [31]. The main drawback of Merkle's scheme is that it requires updating the secret key for each signing process.

SPHINCS$^+$ draws inspiration from these earlier schemes. The fundamental concept involves authenticating numerous FTS key pairs through a hypertree. Each time a new message is generated, a pseudo-random FTS key pair is selected for signing the message. The resulting signature consists of both the FTS signature and the FTS key pair information. Consequently, the authentication information is essentially a hypertree signature, which uses Merkle tree signatures [4]. Additionally, SPHINCS$^+$ is considered stateless as each signature is independent of any previous ones and we do not need to store information about previous signatures.

For SPHINCS$^+$, the key size and the signature size can vary depending on the chosen settings. For instance, SPHINCS$^+$-SHA2-128s has a public key size of 32 bytes, a private key of 64 bytes, and a signature size of 7856 bytes, while SPHINCS$^+$-SHA2-192f has a public key size of 48 bytes, a private key size of 96 bytes, and a signature size of 35664 bytes.

4.1.3.2 FALCON

Lattice-based cryptography has gained popularity in the post-quantum era. A lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^n$ generated by a set of linearly independent vectors $\boldsymbol{b}_1$, $\boldsymbol{b}_2$, ..., $\boldsymbol{b}_n \in \mathbb{R}^n$:

$$\mathcal{L}(\boldsymbol{b}_1, \boldsymbol{b}_2 ... \boldsymbol{b}_n) = \{\sum_{i=1}^{n} a_i \boldsymbol{b}_i : a_i \in \mathbb{Z}\}. \tag{4.1.1}$$

There are several long-standing hard problems when dealing with lattices, such as the shortest vector problem (SVP), the closest vector problem (CVP), the short integer solution problem (SIS), and the learning with errors problem (LWE).

FALCON, which stands for Fast Fourier Lattice-Based Compact Signatures over NTRU (N-th degree truncated polynomial ring), is built upon the SIS problem. FALCON uses the hash-and-sign technique within the GPV framework, developed by Gentry, Peikert, and Vaikuntanathan in 2007 [15], along with NTRU lattices [16] and a trapdoor sampler called fast Fourier sampling. The use of NTRU lattices allows FALCON to achieve a relatively small signature size compared to other lattice-based signatures with the same level of security. Additionally, the fast Fourier sampling technique enables efficient signature generation and verification.

Fouque et al. in [14] propose two versions of FALCON: FALCON-512 and FALCON-1024. FALCON-512 has a public key size of 897 bytes and a signature size of 666 bytes. The sizes approximately double in FALCON-1024, with a public key size of 1793 bytes and a signature size of 1280 bytes. The private key sizes for both versions are roughly three times the signature sizes. Furthermore, FALCON offers more than 64 bits of security. Roughly speaking, Falcon-512 is at the same security level as RSA-2048.

4.1.3.3 CRYSTALS-Dilithium

Similar to FALCON in Section 4.1.3.2, CRYSTALS (Cryptographic Suite for Algebraic Lattices)-Dilithium is also a lattice-based digital signature scheme, whose security is based on the hardness of finding short vectors in module lattices [10]. CRYSTALS-Dilithium uses Lyubashevsky's idea of Fiat-Shamir with Aborts for rejection sampling [30] and adopts the concept of uniform distribution from [2] to significantly reduce the public key size.

The strength of a CRYSTALS-Dilithium key is determined by the dimensions of its polynomial matrices. For instance, a key with parameters $(5,4)$ has a matrix size of 5×4. The public key in this configuration is around 1,500 bytes, and the signature size is about 2,700 bytes. Generally, the larger the matrix size, the stronger the key. Moreover, CRYSTALS-Dilithium uses SHAKE to expand the matrix and the masking vectors, as well as to sample the secret polynomials. Later implementations may use AES-256 for similar cryptographic operations.

Ducas et al. in [10] introduce three variants of CRYSTALS-Dilithium digital signatures, as shown in Table 4.2, each with different public key and signature sizes. They recommend using Dilithium3, which has been analyzed to achieve more than 128 bits of security against all known classical and quantum attacks.

Variant	Dilithium2	Dilithium3	Dilithium5
PK (bytes)	1312	1952	2592
signature (bytes)	2420	3293	4595

Table 4.2: Variants of Dilithium.

4.2 Modification-Tolerant Signature Scheme

Conventional digital signature schemes (CDSS) ensure that a document was created by the sender (authenticity) and has not been changed during transit (integrity). These schemes output a 0 if the signature is invalid, meaning either the signer is not authentic or the message has been changed after signing, but they lack the capability to identify where those changes have occurred. In [20, 21], Idalino et al. proposed a general framework called modification-tolerant digital signatures (MTSS). This framework enables the precise location of a specified number of allowed modifications within a document while maintaining the authenticity of the signature.

MTSS is indeed very useful in various scenarios. For instance, consider a person who needs to provide proof of a financial transaction or payment without revealing their full credit card number. Using MTSS, the person can selectively share specific information, such as the transaction amount and date, while redacting sensitive details like the credit card number. This allows the recipient to verify the integrity of the disclosed information without compromising the payer's privacy.

Another scenario where MTSS prove beneficial involves collaborative work. As a group of authors creates a document, they may want to make further edits while ensuring the

integrity of the original contributions. MTSS provide flexibility in collaboration while maintaining document integrity.

In an MTSS, a message m is divided into n blocks, denoted as $m = (m[1], \dots, m[n])$. For two messages m and m' divided into into n blocks, we define the modification set $I = \text{diff}(m, m') := \{i \in \{1, \dots, n\} : m[i] \neq m'[i]\}$. An *authorized modification structure*, denoted as S, is a collection of sets of blocks that the signer allows for change, where $\mathcal{S} \subseteq \mathcal{P}(\{1, \dots, n\})$ and $\mathcal{P}$ represents the power set of $\{1, \dots, n\}$. The goal is that if σ is a valid signature for m, then it remains valid for m' if and only if $I = \text{diff}(m, m') \in \mathcal{S}$.

An MTSS for authorized modification structure $\mathcal{S} \subseteq P(\{1, \dots, n\})$ on messages with n blocks has three probabilistic polynomial-time algorithms Π = (Key Generation, Sign, Verify), which are detailed as follows:

1. **MTSS-Key-Generation (l):** The key generation algorithm accepts a security parameter l and outputs a key pair (SK, PK) with certain length. The private key (SK) is kept secret by the signer, while the public key (PK) is made available to any party requiring signature verification.
2. **MTSS-Sign(m, SK):** The signing algorithm takes a message (m) divided into n blocks and a private key (SK) as input and outputs a digital signature (σ).
3. **MTSS-Verify(m, PK, σ):** The verification algorithm takes input a message (m), a digital signature (σ), and a public key (PK). It outputs (1, I) if (m, σ) is valid for modification set I and outputs 0 otherwise.

Definition 4.2.1. Let Π be an MTSS for authorized modification structure $\mathcal{S}$, and let (SK, PK) be a pair of secret and public key. A pair (m, σ) of message and signature is *valid if there exits a m' such that σ is a valid output of MTSS-Sign(m', SK) and diff(m, m')$\in \mathcal{S}$. In this case, we say (m, σ) is valid for modification set I (where I = diff(m, m')).*

The simplest authorized modification structure we can consider is one that simply provides an upper bound d on the number of modified blocks. An MTSS with such authorized modification structure is defined next.

Definition 4.2.2. A *d-modification tolerant signature scheme* (d-MTSS) is a MTSS with authorized modification structure $\mathcal{S} = \{T \subseteq \{1, ..., n\} : |T| \leq d\}$.

Our main focus is Scheme 1 in [20], which requires an underlying CDSS, a cryptographic hash function, and a d-CFF(t,n) matrix M, which is presented next.

- **MTSS-Key-Generation(l):**
  Input: a security parameter l.
  Output: a key pair (SK, PK).
  1. The key generation algorithm accepts l and generate the key pair using algorithm **CDSS-Key-Generation(l)**.

- **MTSS-Sign(m, SK):**
  Input: a message $m = (m[1], ..., m[n])$, and a private key SK.
  Output: a signature σ.

  1. Compute $h_j = h(m[j])$, for $1 \leq j \leq n$.
  2. For each $1 \leq i \leq t$, calculate c_i as the concatenation of all h_j such that $M_{i,j} = 1$. Set $T[i] = h(c_i)$.
  3. Use the hash function to compute $h^* = h(m)$. Set $T = (T[1], T[2], ...T[i], h^*)$.
  4. Use **CDSS-Sign(T, SK)** to generate a signature σ'. Output $\sigma = (\sigma', T)$.

- **MTSS-Verify(m', σ, PK):**
  Input: a message $m' = (m'[1], ..., m'[n])$, a public key PK, and $\sigma = (\sigma', T)$.
  Output: a boolean result and a modification set I. If the signature σ' is not valid for S, output $(0, -)$. Otherwise, if the size of I is smaller or equal to d, the output is $(1, I)$; otherwise, it is $(0, I)$.

  1. Ensure that **CDSS-Verify**$(T, \sigma', PK) = 1$. If not, stop and output $(0, -)$, indicating the signature is not valid.
  2. Check if $h^* = h(m')$. If that is the case, it means there is no modification, stop and output $(1, \emptyset)$; otherwise, continue.
  3. Repeat the processes as in step 1 and 2 of MTSS-Sign for $m' = (m'[1], ..., m'[n])$:
     (a) Calculate $h'_j = h(m'[j])$, for $1 \leq j \leq n$.
     (b) For each $1 \leq i \leq t$, calculate c'_i as the concatenation of all h'_j such that $M_{i,j} = 1$. Set $T'[i] = h(c'_i)$.
  4. Initialize an empty set V. For each $1 \leq i \leq t$ where $T[i] = T'[i]$, compute the set of indices of unmodified blocks, $V_i = \{j : \mathcal{M}_{ij} = 1\}$, and accumulate these values in the set of all indices of unmodified blocks, $V = \bigcup_{i=1}^{t} V_i$. Compute $I = \{1, \ldots, n\} \setminus V$. If $|I| \leq d$, output (1, I); otherwise, output (0, I).

Theorem 4.2.3 states the correctness of the scheme.

Theorem 4.2.3. [20] Consider a valid signature σ generated by MTSS-Sign(m, SK) for a message m and key pair (SK, PK), and let m' be a possibly modified message with $|\text{diff}(m, m')| \leq d$. Then, MTSS-Verify$(m, \sigma, PK) = (1, \text{diff}(m, m'))$.

The next theorem shows that when the conditions $|I| \leq d$ is not satisfied, MTSS-Verify outputs $(0, I)$. In this case, I includes all the defectives but may also contain some non-defectives.

Theorem 4.2.4. [20] Consider a valid signature σ generated by MTSS-Sign(m, SK) for a message m and key pair (SK, PK), and let m' be a possibly modified message with $|\text{diff}(m, m')| > d$. Then, MTSS-Verify$(m, \sigma, PK) = (0, I)$, and for any $i \in \{1, \ldots, n\} \backslash I$, block $m[i]$ is guaranteed to be unmodified.

The security of d-MTSS depends solely on the underlying CDSS and the cryptographic hash function. The following theorem establishes the security of d-MTSS.

Theorem 4.2.5. [20] Let X be a d-MTSS described above based on an existentially unforgeable CDSS and on a collision resistant hash function h. Then X is existentially unforgeable.

Chapter 5

Implementation of Modification-Tolerant Signature Scheme

An introduction to the modification-tolerant signature scheme (MTSS) proposed by Idalino et al. [20, 21] is given in Section 4.2. This chapter talks about our own implementation of the d-MTSS, which is the focus of this thesis. The main idea of the d-MTSS is to first divide a message into n blocks and then concatenate the blocks according to a d-CFF. By signing the hashes of these concatenations, the scheme allows for the identification of up to d modified blocks during the verification process.

While Idalino at al.[20] provide the definition of the general d-MTSS, a concrete implementation of the scheme requires specification of various features and primitives, which are only described in a general way. One crucial feature is how to divide the message into blocks in such a way, that after modifications are introduced, the block structure remains intact. Other issues to be addressed in an implementation are: choosing a conventional digital signature scheme to employ, choosing a d-CFF to use, and possibly refining the MTSS scheme to improve its efficiency. Section 5.1 explains how to divide messages for two types of files. Section 5.2 provides information on the hash functions and digital signature schemes used in the implementation, along with their parameters. In Section 5.3, we comment about how to determine the number of tests t based on n and d for different constructions. Finally, Section 5.4 provides a detailed description of a revised d-MTSS scheme, which we call MICO scheme.

5.1 Message Division

The authors in [21] pointed out that an important issue to consider in MTSS is how to divide messages. They proposed two possible approaches: use special delimiters (e.g., tags in an XML document or reserved characters in text), or use the own data organization (e.g. the records of a database). Moreover, we can also divide messages into fixed-size blocks. However, using a single method for all message types can cause problems. For instance, in text files, if a fixed-size approach is used, any insertions in one block can shift the blocking

of the remaining contents. To avoid such issues, a block structure should be preserved after modifications are made, and it is essential to divide messages based on their specific block structure. In our implementation, we support two file types, text files and images, and define their block structures.

For text files, the choice of block structure depends on the format and type of file. For plain text, it is convenient to use lines or paragraphs as blocks. In cases like CSV files, rows naturally serve as blocks. In our case, since we are working with plain text and the scheme requires a defined block structure, we divide the file into blocks that consist of a fixed number of lines, where lines are separated by a newline character. This allows modifications within lines, but the insertion or deletion of lines is not permitted.

For images, formats like bitmap (BMP) and Portable Gray Map (PGM) store pixels in grids. A common approach to structuring blocks is to divide the image into fixed-size blocks. This method ensures that modifications to one part of the image do not affect the rest of the data. Similar to text files, modifications within the pixels of a block are allowed, but adding or removing extra pixels is not permitted.

The following sections provide detailed explanations on how to divide these two types of files into blocks.

5.1.1 Text Files

In our implementation for text files, a line is separated from the next line by a newline character, and the block size b represents a fixed number of lines, as previously mentioned.

Given a text file with k total lines, we can divide them based on a predefined block size b. Alternatively, if we do not specify the block size but instead specify the number of blocks n we want, we can calculate the block size b by dividing the total number of lines by the number of blocks, using the formula $b = \frac{k}{n}$. If the calculation yields a decimal value, it is rounded: values of 0.5 or greater round up, while smaller values round down. Once b is determined, we can divide the lines into blocks of size b. However, if $k \pmod{b} \neq 0$, any remaining lines that do not fit into the full blocks will form an additional, smaller block at the end.

There are some edge cases to consider. First, if the specified block size exceeds the total number of lines, i.e., $b > k$, then n will be set to 1, meaning the entire file will be treated as a single block. Similarly, if the specified number of blocks n exceeds the total number of lines, i.e., $n > k$, then b will be set to 1, ensuring that each block contains exactly one line.

5.1.2 Image Files

For image files, we use the fixed-size blocks technique to divide them into small squares. The block size, denoted as b, refers to the side length of each square, so we divide the image into squares with dimensions $b \times b$.

Like text files, if we prefer not to specify the block size, we can instead specify the number of blocks, denoted as n. In such cases, the block size b of an image with dimensions $w \times h$ is determined by the equation $b = \sqrt{\frac{w \times h}{n}}$. Similar to text files, b will be rounded to the nearest integer if it is a decimal value. If the image dimensions are $w \times h$ and the desired or calculated block size is b, the actual number of blocks n is given by: $n = \lceil \frac{w}{b} \rceil \times \lceil \frac{h}{b} \rceil$. This means that any remaining pixels that cannot form a complete square are grouped into rectangular blocks.

It is also important to consider special cases. If the number of blocks n is larger than the image dimension, $w \times h$, the block size b is set to 1. Conversely, if the block size b exceeds both the width w and the height h of the image, the number of blocks n is set to 1, meaning the entire image is treated as a single block.

In the case of image files, we work with the "plain" PGM format since it is particularly well-suited for image processing. In this format, each pixel of the image is represented as a decimal number within a 2-dimensional array structure. Furthermore, the plain PGM format is stored as plain text, making it human-readable and accessible for manual inspection or debugging.

The first four lines of each "plain" PGM image contain the following information: the first line has a "magic number" to identify the file type. For "plain" PGM images, this is the two characters "P2". The second line is either a whitespace or contains the name of the software used to create the image. The third line shows the image's width w and height h, in pixels, as two decimal numbers. The fourth line indicates the maximum gray value of the image in decimal, which is a value from 0 to 255. Following these four lines, the rest of the file contains $w \times h$ pixels arranged as a 2D array, where each number represents the gray value of a pixel.

In our implementation, we exclude the initial four lines when performing block division. During this process, we divide the 2D array of $w \times h$ pixels into smaller blocks. If the block size $b = 1$, each block corresponds directly to a single pixel. However, for larger block sizes, each block will encompass a group of pixels, effectively partitioning the image into squares of pixels.

5.2 Cryptography Library and Primitives

The d-MTSS requires two cryptographic primitives, namely a digital signature scheme and and a cryptographic hash functions. During the cryptography implementation, we utilize an external open-source library called Bouncy Castle Java [33], specifically for ready-to-use hash algorithms and digital signature schemes. The version of the signer JAR file we use is *bcprov-jdk18on-177.jar*. The primary packages we use from Bouncy Castle are *org.bouncycastle.crypto* and *org.bouncycastle.pqc.crypto*, in conjunction with the Java Cryptography Architecture (JCA) standard library.

Recall that in Scheme 1 discussed in Chapter 4, hash functions are utilized to hash each individual block and block concatenations. For hash algorithms, we have chosen SHA2-256,

SHA2-512, SHA3-256, and SHA3-512. In terms of the digital signature schemes, we employ two classical schemes (RSA and ECDSA), and three post-quantum schemes (SPHINCS$^+$, FALCON, and CRYSTALS-Dilithium). Table 5.1 shows the sizes of public keys, private keys, and signatures, along with other scheme-specific parameters.

	PK size (bytes)	signature size (bytes)	Other
RSA-2048	256	256	Certainty: 80 Padding: PKCS#1 v2.1 Public exponent: 65537
ECDSA	64	70-72	Curve: secp256k1
SPHINCS$^+$	32	7,856	Parameter: SHA2-128s
FALCON-512	897	650-660	-
Dilithium 3	1,952	3,309	-

Table 5.1: Parameters of digital signature schemes used in our implementation.

All the sizes listed in Table 5.1 are based on the Bouncy Castle library and may vary slightly with different cryptography libraries. In RSA key generation, as with other cryptography libraries, Bouncy Castle uses "probable prime" numbers. These are numbers that have passed primality tests, such as the Miller-Rabin test [36], indicating they are highly likely to be prime. For RSA, two large prime numbers, p and q, are required, as shown in Chapter 4. When setting up RSA, we must choose a certainty level, which dictates how confident we are in these numbers being prime. We have selected a certainty level of 80, as going higher would slow down key generation significantly. This setting means there is a $1/2^{80}$ probability that a selected number might not be prime. Furthermore, 65537 is the most popular choice of public exponent for RSA signatures. The secp256k1 curve for ECDSA is commonly used in various applications, including Bitcoin. Additionally, SPHINCS$^+$ offers multiple parameter sets for different security and performance needs. The parameter SHA2-128s, which uses the SHA-2 hash algorithm, is designed to achieve a 128-bit security level.

5.3 Choice of d-CFF Construction and Determination of Number of Tests

For our implementation, the number of items (blocks) n and the maximum defective number (modified blocks) d are predefined, so we need to decide which d-CFF constructions will be used and determine how many tests t need to be conducted based on n and d. Therefore, the focus of this section is on discussing how to determine t for the different CFF constructions discussed in Chapter 2: Sperner sets, Steiner Triple System, and Reed-Solomon codes.

The Sperner construction is used when $d = 1$, and provides the minimum t for a given n. To find t, we simply need to compute the smallest t such that $\binom{t}{\lfloor t/2 \rfloor} \geq n$.

The STS construction is used when $d = 2$. The determination of t, i.e., the order of the STS is more complicated. The problem can be stated as follows: Given n, find the smallest

order t such that there exists an STS(t) with b blocks where $b \geq n$. In other words, given n, determine t =min$\{v \equiv 1, 3 \pmod 6$ and $\frac{v(v-1)}{6} \geq n\}$.

To solve this, we start by finding v using the equation: $v = \frac{1+\sqrt{1+24n}}{2}$. Next, we adjust v to satisfy the condition $v \equiv 1$ or $3 \pmod 6$. The adjustments based on the remainder $r \equiv v \pmod 6$ as follows:

1. If r is either 0 or 2, increment v by 1.
2. If r is 1 or 3, no adjustment is needed.
3. If r is 4 or 5, adjust v as $v = v + 7 - (v \bmod 6)$.

It is important to note that we use STS construction only when $n \geq 10$. If $n < 10$, we use an identity matrix to construct CFF instead. This is because for $n \leq 9$, $t(2, n) = n$ (see [26]), and an STS could result in the same or even more rows compared to using an identity matrix.

Regarding Reed-Solomon codes, determining t involves identifying the appropriate parameters k, N, q where k, N, $q \in \mathbb{Z}^+$ that satisfy certain requirements based on n and d. Instead of using an arbitrary prime power, we use q a prime number to facilitate implementation, since in this case field operations can be implemented using modular arithmetic. In addition, this allows us to use the fast specialized decoding algorithm described in Section 3.2.3. By using Construction 1, given n and d, the requirements for parameters k, N, q to yield a d-CFF(Nq,n) must satisfy:

1. q is a prime;
2. $N \leq q$;
3. $q^k \geq n$;
4. $d \leq \lfloor \frac{N-1}{k-1} \rfloor$ with $k \geq 2$.

The problem to solve is: Given n and d, choose $k \geq 2$, q prime, and $N \leq q$, such that the Reed-Solomon code in Construction 1 minimizes $N \cdot q$. In other words, given n and d, determine $t = \min\{N \cdot q : q \text{ is prime}, k \geq 2, n \leq q^k, N = d(k-1) + 1 \leq q\}$.

When determining the parameters, we first consider the relationship $n \leq q^k$ and start by computing k rather than q. If we start with q, we would need to check all prime numbers from 2 up to the smallest prime greater than $\lceil \sqrt{n} \rceil$. This approach involves a larger number of iterations as n increases. On the other hand, starting with k involves examining possible values of k from 2 to $\lceil \log_2 n \rceil$, assuming $q = 2$ as the smallest base. Since $\lceil \log_2 n \rceil$ is in $\mathcal{O}(\lceil \sqrt{n} \rceil)$, this method results in fewer iterations as n grows larger.

Once k is fixed, the next step is to calculate either N or q. Starting with q can lead to scenarios where $N > q$, making the solution invalid. Therefore, we start with $N = d(k-1)+1$. From there, q is determined as the smallest prime that is greater than or equal to the maximum of $n^{1/k}$ and N, ensuring that the conditions $q^k \geq n$ and $N \leq q$ are both satisfied.

Below is the algorithm for finding the optimal k, N, q:

1. Initialize the variable *product* with $+\infty$.
2. For each k from 2 to $\lceil \log_2 n \rceil$, compute $N = d(k-1)+1$.
3. Calculate q as the smallest prime that is greater than or equal to the maximum of $n^{(1/k)}$ and N, and compute $N \cdot q$.
4. If the value of $N \cdot q$ is less than *product*, update $k_{best} = k$, $q_{best} = q$, $N_{best} = N$ and $product = N \cdot q$.

If the resulting number of rows $N \cdot q$ exceeds n, we use an identity matrix instead.

5.4 MICO: a Revised d-MTSS Scheme

In Section 4.2, we present a d-MTSS scheme from [20], which requires an underlying conventional digital signature scheme, a public hash function h, and a d-CFF(t,n) matrix M. Our revised scheme, renamed as MICO, addresses three key aspects of the original scheme, which are discussed next. Here, "M" stands for Modification, "I" for Integrity, "C" for Cryptography, and "O" for Optimization, as our goal is to minimize the number of tests needed for combinatorial group testing.

First, we eliminate the step of hashing individual blocks before concatenation, significantly reducing processing time, especially for large numbers of blocks. For instance, hashing 1,000,000 blocks originally took about 10 minutes in our tests. We note that the elimination of hashing individual blocks does not compromise the security proof of Theorem 4.2.5 found in [20, Theorem 6]. In this proof, an existential forgery of MTSS would imply an existential forgery of the underlying CDSS or a collision pair for h. The relevant argument to be modified in the original proof is the part about a supposed existence of a collision $h(x) = h(x')$, where $x = (h(m[j_1])||\dots||h(m[j_s]))$ and $x' = (h(m'[j_1])||\dots||h(m'[j_s]))$ and $m[k] \neq m'[k]$ for some $k \in \{j_1, \dots, j_s\}$. Two cases are analyzed in [20], namely the case $h(m[k]) = h(m'[k])$ or the case ($h(m[k]) \neq h(m'[k])$ and $h(x) = h(x')$). In either case, a collision pair would be found for h, $(m[k], m[k'])$ or (x, x'), respectively, leading to a contradiction. In our revised scheme, we would instead suppose we have $h(y) = h(y')$, where $y = (m[j_1]||\dots||m[j_s])$ and $y' = (m'[j_1]||\dots||m'[j_s])$, with $m[k] \neq m'[k]$, and the collision pair would be (y, y'), also leading to a contradiction. The reader can refer to the proof of [20, Theorem 6] to verify that the security proof with this modification still holds.

Second, while the original scheme only proposed a general decoding algorithm using negative tests to identify defectives, our revised version can employ fast, specific algorithms based on positive tests, as detailed in Section 3.2.

Third, in the MICO scheme, the signature σ comprises more components. Similar to the original scheme, it includes a tuple of hashes of block concatenations (not individually hashed), denoted as $h[j]$ for $1 \leq j \leq n$, the hash of the entire message h^*, and a signature σ' of this tuple and h^* using the underlying CDSS. Additionally, the signature contains header information (parameter specs) specifying the name of the underlying CDSS, the

cryptographic hash algorithm, the file type, the CFF construction method type, the CFF matrix data structure type, the block size b, the number of blocks n, the value of d, and the number of rows t. This comprehensive header allows the verification algorithm to reconstruct the matrix M without storing it separately in the signature, enhancing the scheme's efficiency and self-containment.

The step-by-step description of the scheme is given next:

- **MICO-Key-Generation**(l, specs):
  Input: a security parameter l, and specs, which includes the user-specified choice for the underlying CDSS.
  Output: a key pair (SK, PK)

  1. The key generation algorithm accepts l and generate a key pair using **CDSS-Key-Generation(**l**)** for the specific underlying CDSS.

- **MICO-Sign**(m, SK, specs):
  Input: a message m, a private key SK, and algorithm specifications (specs). In specs, the user specifies the underlying CDSS, the hash algorithm h, the value of d, the CFF construction method type, the CFF matrix data structure type, the file type, and either the block size b or the number of blocks n, along with the corresponding value.
  Output: a signature σ = (σ', S)

  1. **Pre Step 1:** Divide message into n blocks $m = (m[1], ..., m[n])$ based on the file type, specified block size or the number of blocks.
  2. **Pre Step 2:** Construct the d-CFF M based on the value of d, the chosen CFF construction method type and the CFF data structure type, which includes computing the number of rows t of M.
  3. **Hash Entire Message:** Use the hash algorithm h to compute $h^* = h(m)$.
  4. **Concatenating According to d-CFF:** For each $1 \leq i \leq t$, calculate c_i as the concatenation of blocks such that $M_{i,j} = 1$, for $1 \leq j \leq n$.
  5. **Hash the t Block Concatenations:** For each $1 \leq i \leq t$, hash each concatenation c_i to obtain $T[i] = h(c_i)$.
  6. **Signing Preparation:** Set S = (specs, t, $T[i]$ for $1 \leq i \leq t$, h^*).
  7. **Sign with SK:** Use **CDSS-Sign(**S, SK**)** of the underlying CDSS to generate a signature σ' of S. Output σ = (σ', S).

- **MICO-Verify**(m', σ, PK, dalg):
  Input: a message m', a public key PK, a signature $\sigma = (\sigma', S)$, and dalg. Parameter dalg indicates the choice of the decoding algorithm, which can be general (MatrixCompact or MatrixList) or specific to the CFF construction method type.
  Output: a boolean result and a modification set I. If the signature σ' is not valid for S, output $(0, -)$. Otherwise, if the size of I is smaller or equal to d, the output is $(1, I)$; otherwise, it is $(0, I)$.

1. **Pre Step 1:** Divide the message into n blocks $m' = (m'[1], ..., m'[n])$ based on the file type, and block size or number of blocks contained in S.
2. **Pre Step 2:** Construct the d-CFF M using the value of d, the specified CFF construction method type, and the CFF data structure type given in S.
3. **Verify Signature:** Ensure that **CDSS-Verify**(S, σ', PK)= 1 using the underlying CDSS in S. If not, stop and output $(0, -)$, indicating the signature is not valid.
4. **Check Modifications:** Retrieve h^* from S, and check if $h^* = h(m')$. If true, it implies that m' has not been modified. Stop and output $(1, \emptyset)$; otherwise, continue.
5. Repeat the processes as in Step 4 and 5 of MICO-Sign for $m' = (m'[1], ..., m'[n])$:
   (a) **Concatenating According to d-CFF:** Calculate c_i' for $1 \leq i \leq t$ using m' and M.
   (b) **Hash the t Block Concatenations:** Using the hash function in S, hash each concatenation to obtain $T'[i] = h(c_i')$.
6. **Locate Modifications:** Retrieve $T[i]$ for $1 \leq i \leq t$ from S. Compare $T[i]$ and $T'[i]$ for each $1 \leq i \leq t$ to create a result vector $y \in \{0, 1\}^t$, where 0 indicates unmodified ($T[i] = T'[i]$) and 1 indicates modified. Using the decoding algorithm dalg, compute the set of defective items I from y. If $|I| \leq d$, output (1, I); otherwise, output (0, I).

5.5 Illustration of Locating Modifications in an Image File

In this section, we present an illustration of detecting modifications in an image using the MICO scheme. Figure 5.1 is the original image in the plain PGM format, which has a size of 470×640 (height $\times$ width).

Figure 5.1: Tabaret Hall at the University of Ottawa, from [43].

We fix the block size, which is the side dimension of a square, to 80. Figure 5.2 shows the image being divided into 48 blocks.

Figure 5.2: Image divided into blocks of 80×80 squares.

Figure 5.3 shows the image with modifications in blocks 1 and 41.

Figure 5.3: Image with modified blocks {1,41}.

After running the MICO scheme, we identified that the modified blocks are block 1 and block 41. These blocks are whited out to indicate the modifications, as shown in Figure 5.4.

Figure 5.4: Image with modified blocks {1,41} changed to white.

Chapter 6

Experiments with the MICO Scheme

This chapter aims to evaluate the practical performance of the MICO scheme, which is the modification-tolerant signature scheme (MTSS) proposed in Section 5.4. That is, how efficiently the scheme locates modifications and ensures data integrity.

First, in Section 6.1, we briefly discuss our experimental approach and the tests conducted to ensure the correctness of the scheme before performing any efficiency experiments. Next, in Section 6.2, we evaluate the performance of the underlying primitives of the MICO scheme, including conventional digital signature schemes and hash algorithms. Finally, in Section 6.3 and 6.4, we examine the overall efficiency of the MICO scheme based on two criteria: signature size and running time.

6.1 Experimental Approach

Similarly to Section 3.3, we run all the tests in Java on a MacOS laptop with 8 GB of RAM and an M2 chip with a maximum processor speed of 3.49 GHz. Each test is run 50 times, except when otherwise specific, and we record the average results.

As mentioned in Chapter 5, our schemes allow for two types of files: text files and PGM images. Tables 6.1, 6.2, and 6.3 present the details of the text files and images used in our tests.

All the files in Table 6.1 are used to test the correctness of our scheme. Text_1 to Text_5 and Image_1 to Image_5 are generated by Java programs. For Text_2 to Text_4, each line contains 60 random uppercase English letters. In Text_1 and Text_5, each line contains a number corresponding to the line number (e.g., line i has the number i). Each pixel in Image_1, Image_2, and Image_3 has a decimal value of 0, making them completely black. In Image_4 and Image_5, the pixels have random colors ranging from 0 to 255.

Text_6 is the LaTeX file from Chapter 3 of this thesis, while Text_7 and Text_8 are the scripts of two plays: A Doll's House by Henrik Ibsen and Romeo and Juliet by Shakespeare. Text_9 and Text_10 are two novels: En Route by Joris-Karl Huysmans and Little Women by Louisa May Alcott. Additionally, Image_6 through Image_10 are five

pictures downloaded from Google Images and converted to "plain" PGM format using GIMP (GNU Image Manipulation Program).

File Name	File Type	File Size (bytes)	Source
Text_1	Text	200	Java
Text_2	Text	5,000	Java
Text_3	Text	100,000	Java
Text_4	Text	500,000	Java
Text_5	Text	1,000,000	Java
Text_6	Text	1,034	Chapter 3 of this thesis
Text_7	Text	5550	A Doll's House
Text_8	Text	5,647	Romeo and Juliet
Text_9	Text	14,192	En Route
Text_10	Text	22,847	Little Women
Image_1	Image	450 × 379	Java
Image_2	Image	800× 1,000	Java
Image_3	Image	1,400 × 1,288	Java
Image_4	Image	2,637 × 2,092	Java
Image_5	Image	6,000 × 3,000	Java
Image_6	Image	288 × 298	Online
Image_7	Image	470 × 640	Online
Image_8	Image	870 × 450	Online
Image_9	Image	1,488 × 2,240	Online
Image_10	Image	5,088 × 3,253	Online

Table 6.1: Files for testing correctness.

The files in Tables 6.2 and 6.3 are used for efficiency testing in Section 6.4. They are generated by Java programs in a similar way to some files in Table 6.1. This means the text files contain 60 random English characters per line, and the image files are all black, regardless of size. This setup not only makes it easy for others to replicate our tests to verify our findings, but also provides a foundation for understanding how the MICO scheme performs with various file sizes and types. This baseline can guide us in choosing appropriate set-ups for testing and comparing the performance of our scheme with more complex file inputs, such as text files with varying line lengths or images with a broader color range.

File Name	number of lines	number of characters each line
Text_SM_40000	40,000	60
Text_MED_200000	200,000	60
Text_LAR_1000000	1,000,000	60

Table 6.2: Text files for testing efficiency.

File Name	dimension (width × height)	each pixel (decimal number)
Image_SM_200_200	200 × 200	0
Image_MED_500_500	500 × 500	0
Image_LAR_1000_1000	1,000 × 1,000	0

Table 6.3: Images for testing efficiency.

We have different values of d for various tests. Therefore, for each specific test, we have exactly d modified blocks. For easier verification of our scheme's results while still spreading out the locations of modified blocks, we distribute the modifications equally. For example, if $d = 2$, the first block and the last block will be modified. If $d = 3$, the first, middle, and last blocks will be modified. For instance, if $n = 2000$ and $d = 3$, we modify the first block, the 1000th block, and the last block.

In terms of modifying our files, for text files, we simply rewrite a line in the modified block to contain only the number 255. For image modifications, we change the decimal number of a pixel in the modified block to 255, which means changing the pixel from black to white.

Moreover, for correctness of experiments, we have 10 different combinations of setups, where each setup includes 8 parameters: the underlying CDSS, the hash algorithm, the value of d, the CFF construction method type, the CFF matrix data structure type, the choice between fixing block size or the number of blocks (0 for fixing the block size and 1 for fixing the number of blocks), the corresponding fixed number, and the group testing method for verification (general decoding method or specific method based on the construction method as discussed in Chapter 3). We have included different possible options for each parameter, as shown in Table 6.4. Moreover, we have considered different values of n, ranging from tens to hundred-thousands.

	CDSS	Hash	d	CFFMethod	CFFDataStructure	choice	number	GTMethod
set-up 1	ECDSA	SHA2256	1	Sperner	MatrixList	0	1,000	general
set-up 2	RSA	SHA2512	2	STS	MatrixList	1	2,942	general
set-up 3	SPHINCS$^+$	SHA3256	10	RS	MatrixCompact	0	9	specific
set-up 4	FALCON	SHA3512	20	RS	MatrixCompact	1	33,334	specific
set-up 5	Dilithium	SHA2256	30	RS	MatrixCompact	0	1	specific
set-up 6	ECDSA	SHA2512	1	Sperner	MatrixCompact	1	20	specific
set-up 7	RSA	SHA3256	2	STS	MatrixCompact	0	25	specific
set-up 8	SPHINCS$^+$	SHA3512	10	RS	MatrixList	1	6,643	general
set-up 9	FALCON	SHA2256	20	RS	MatrixList	0	5	general
set-up 10	Dilithium	SHA2512	30	RS	MatrixList	1	89,178	general

Table 6.4: Ten set-ups for testing correctness.

For testing the correctness of our scheme, we modify our files from Table 6.1 to have the corresponding d modified blocks as specified in the column labeled d in Table 6.4 and test them with different set-ups. In all cases, the scheme successfully verified the signatures and accurately detected modifications. This confirms that the scheme's implementation is likely

correct and reliable under the tested conditions. As a result, we then proceed with running the efficiency tests in the following sections.

6.2 Performance of Underlying Choices

In this section, we examine the performance of two underlying cryptographic primitives used in the MICO scheme, conventional digital signature schemes (CDSS) and hash algorithms, which will directly influence the effectiveness of the scheme. These evaluations aim to identify the underlying choices suited for different preferences.

6.2.1 Performance for Different Choices of CDSS

A comparative analysis of five different digital signatures is conducted in this section. We compare the signature length of each digital signature scheme and measure the execution times for key generation, signing, and verification. To provide a comprehensive evaluation, we run three sets of experiments using messages of varying sizes: 100 bytes, 10,000 bytes, and 1,000,000 bytes, which is shown in Table 6.5. This allows us to assess how each CDSS performs across different data scales. The digital signature schemes are given in increasing order of signature sizes.

	Key Gen (ms)	Sign (ms)	Verify (ms)	Signature Size (bytes)
	Message Size: 100 bytes			
ECDSA	2.87	0.51	1.15	70
RSA	204.03	2.02	0.10	256
FALCON	17.84	3.10	0.29	656
Dilithium	1.71	1.16	0.55	3,309
SPHINCS+	152.53	1,263.52	1.38	7,856
	Message Size: 10,000 bytes			
ECDSA	2.64	0.54	1.21	72
RSA	167.78	2.19	0.17	256
FALCON	17.37	3.24	0.37	652
Dilithium	1.70	1.32	0.63	3,309
SPHINCS+	145.76	1,224.60	1.45	7,856
	Message Size: 1,000,000 bytes			
ECDSA	2.81	6.56	7.26	71
RSA	187.36	8.52	6.21	256
FALCON	18.16	9.66	6.80	656
Dilithium	1.76	8.52	6.79	3,309
SPHINCS+	144.14	1,245.10	7.75	7,856

Table 6.5: Running time of five CDSS for different message sizes.

As message sizes increase, key generation times remain relatively stable, while signing and verification take longer for all the five digital signatures. Comparing classical digital signature schemes, ECDSA demonstrates quicker key generation and signing processes with shorter signature sizes compared to RSA. However, RSA wins in signature verification time.

Among post-quantum digital signatures, FALCON produces the smallest signature sizes. Dilithium consistently is the fastest in key generation and signing. For verification, FALCON initially performs better with message sizes of 100 and 10,000 bytes, but its verification time approaches that of Dilithium at 1,000,000 bytes.

Overall, SPHINCS$^+$ has the slowest performance in both signing and verification. Therefore, ECDSA is recommended for scenarios requiring rapid message signing and the shortest signature size, where quantum resistance is not required. Conversely, if quantum safety is needed, Dilithium or FALCON should be considered, depending on whether a faster running time or a smaller signature size is prioritized.

6.2.2 Performance of Hash Algorithms

The goal of this section is to determine the selection criteria for hash functions based on the message size and digest size.

In our implementation, there are four hash algorithms for digesting: SHA2-256, SHA2-512, SHA3-256, and SHA3-512. SHA2-256 and SHA3-256 offer 256 bits of pre-image resistance and 128 bits of collision resistance while SHA2-512 and SHA3-512 provide 512 bits of pre-image resistance and 256 bits of collision resistance. Table 6.6 presents the hashing times for each algorithm with different message sizes.

Message Size (bytes)	Digest Size: 32 bytes		Digest Size: 64 bytes	
	SHA2-256 (ms)	SHA3-256 (ms)	SHA2-512 (ms)	SHA3-512 (ms)
10	0.65	0.62	0.13	0.51
1,000	0.72	0.69	0.24	0.64
100,000	1.54	1.22	1.19	1.46
1,000,000	7.76	4.68	4.89	7.70
10,000,000	62.41	39.80	44.26	69.02
100,000,000	616.04	392.92	394.18	685.57

Table 6.6: Running time of hash algorithms with different message sizes.

According to Table 6.6, hashing times for each algorithm increases as the message size increases. Additionally, among the hash algorithms with a digest size of 32 bytes, namely SHA2-256 and SHA3-256, SHA3-256 hashes faster than SHA2-256. The difference becomes more obvious when the message size gets larger. For algorithms with a digest size of 64 bytes, SHA2-512 is quicker than SHA3-512. Therefore, the choice of a hash function should be based on individual priorities. For instance, those prioritizing a shorter digest size and quicker performance for hashing messages should choose SHA3-256. On the other hand, for those seeking a higher security level and faster processing, SHA2-512 would be preferable.

6.3 Analysis of MICO Signature Size

The signature size is one of the important factors in assessing the efficiency of a digital signature. In our implementation, there are two factors affecting its signature size: the underlying digital signatures and the hash functions. Therefore, in the following sections, we analyze the signature size of the MICO scheme with different choices of CDSS and hash algorithms. This analysis helps provide insights into the expected signature size for various combinations and guide the selection of choices based on different scenarios in practice.

As previously discussed in Section 5.4, the MICO signature σ includes header information. The header information is around 40 to 50 bytes. Excluding the header information, this section analyzes the size of the MICO signature, denoted as σ^*, which comprises the tuple of hashes $T[i]$ for $1 \leq i \leq t$, the hash of the message h^*, and the underlying CDSS signature, σ'.

We begin by comparing the MICO signature size with different CDSS and hash algorithms using direct calculation, shown in Section 6.3.1. Subsequently, we conduct experiments to compare the signature size, as detailed in Section 6.3.2. We also make a comparison between the sizes derived from theoretical calculations and experimental results.

6.3.1 Analyzing MICO Signature Size from Direct Calculations

We are able to calculate the length of σ^* as a function of t. Since σ^* includes a tuple of t hashes and h^*, the total number of hashes is $t + 1$.

In our implementation, there are two hash lengths: either 32 bytes (SHA2-256 or SHA3-256) or 64 bytes (SHA2-512 or SHA3-512). There are five digital signature schemes: RSA, ECDSA, SPHINCS$^+$, FALCON, and Dilithium. Hence, a total of 10 different functions of σ^* as a function of t are derived in Table 6.7. Moreover, we include two plots showing the signature size for each hash length with five different CDSS.

The signature sizes for RSA, SPHINCS$^+$, and Dilithium are fixed at 256 bytes, 7856 bytes, and 3309 bytes, respectively. On the other hand, the signature size for ECDSA fluctuates between 70 to 72 bytes, and for FALCON fluctuates from 650 to 660 bytes in the implementation. For consistency, we choose the median values: 71 bytes for ECDSA and 655 bytes for FALCON.

From Table 6.7, it is evident that when t is relatively small and for the same hash length, SPHINCS$^+$ yields the largest signature size while ECDSA has the shortest among all schemes, primarily due to the effect of the constant term in the function of σ^*. In terms of quantum-safe, FALCON has the smallest signature size. Therefore, one should consider use either ECDSA or FALCON based on their needs for a shorter MICO signature size when t is small.

However, as t becomes larger, the difference in signature sizes among different CDSS with the same hash algorithm become negligible because the slope term becomes dominant. Consequently, for large t, the choice of CDSS does not significantly affect the signature size. On top of that, as t becomes relatively large, the signature size in SHA2/3-512 is nearly twice

Hash Algorithm	CDSS	MICO Signature Size (bytes)
SHA2/3-256	ECDSA	$\sigma^*(t) = 32(t+1) + 71$
SHA2/3-512	ECDSA	$\sigma^*(t) = 64(t+1) + 71$
SHA2/3-256	RSA	$\sigma^*(t) = 32(t+1) + 256$
SHA2/3-512	RSA	$\sigma^*(t) = 64(t+1) + 256$
SHA2/3-256	FALCON	$\sigma^*(t) = 32(t+1) + 655$
SHA2/3-512	FALCON	$\sigma^*(t) = 64(t+1) + 655$
SHA2/3-256	Dilithium	$\sigma^*(t) = 32(t+1) + 3309$
SHA2/3-512	Dilithium	$\sigma^*(t) = 64(t+1) + 3309$
SHA2/3-256	SPHINCS$^+$	$\sigma^*(t) = 32(t+1) + 7856$
SHA2/3-512	SPHINCS$^+$	$\sigma^*(t) = 64(t+1) + 7856$

Table 6.7: Comparison of MICO signature size using direct calculation.

as large as in SHA2/3-256. As a result, for a shorter signature size, SHA2-256 or SHA3-256 is preferable in these cases, although this comes with a trade-off in security level.

6.3.2 Analyzing MICO Signature Size Through Experiments

An alternative way to compare signature sizes among different combinations is to write the MICO signature into a file and measure the file size. By signing the same message using different hash algorithms and conventional digital signatures, we can determine the respective signature sizes. We have chosen to construct CFF using STS, as its t increases the most rapidly as n rises. We then conduct tests for $t = 157$ and $t = 1417$ using a text file with 1,000,000 lines. The results and calculated values from equations in Table 6.7 are shown in the following tables.

Hash Algorithm	Signature Scheme	Signature Size (bytes)	Calculated Size (bytes)
SHA2/3-256	ECDSA	5,126	5,127
SHA2/3-256	RSA	5,312	5,312
SHA2/3-256	FALCON	5,709	5,711
SHA2/3-256	Dilithium	8,365	8,365
SHA2/3-256	SPHINCS$^+$	12,912	12,912
SHA2/3-512	ECDSA	10,184	10,183
SHA2/3-512	RSA	10,368	10,368
SHA2/3-512	FALCON	10,769	10,767
SHA2/3-512	Dilithium	13,421	13,421
SHA2/3-512	SPHINCS$^+$	17,968	17,968

Table 6.8: Comparison of MICO signature size when $t = 157$.

Hash Algorithm	Signature Scheme	Signature Size (bytes)	Calculated Size (bytes)
SHA2/3-256	ECDSA	45,447	45,447
SHA2/3-256	RSA	45,632	45,632
SHA2/3-256	FALCON	46,034	46,031
SHA2/3-256	Dilithium	48,685	48,685
SHA2/3-256	SPHINCS$^+$	53,232	53,232
SHA2/3-512	ECDSA	90,824	90,823
SHA2/3-512	RSA	91,008	91,008
SHA2/3-512	FALCON	91,406	91,407
SHA2/3-512	Dilithium	94,061	94,061
SHA2/3-512	SPHINCS$^+$	98,608	98,608

Table 6.9: Comparison of MICO signature size when $t = 1{,}417$.

The signature sizes for RSA, SPHINCS$^+$, and Dilithium in both tables match the calculated values for both hash lengths exactly. As previously mentioned, ECDSA's signature size ranges between 70 and 72 bytes, while FALCON's size varies between 650 and 660 bytes in practice. Therefore, since the median value was used to calculate the signature sizes in both tables, the results for ECDSA and FALCON are very close but not exactly the same as the calculated values.

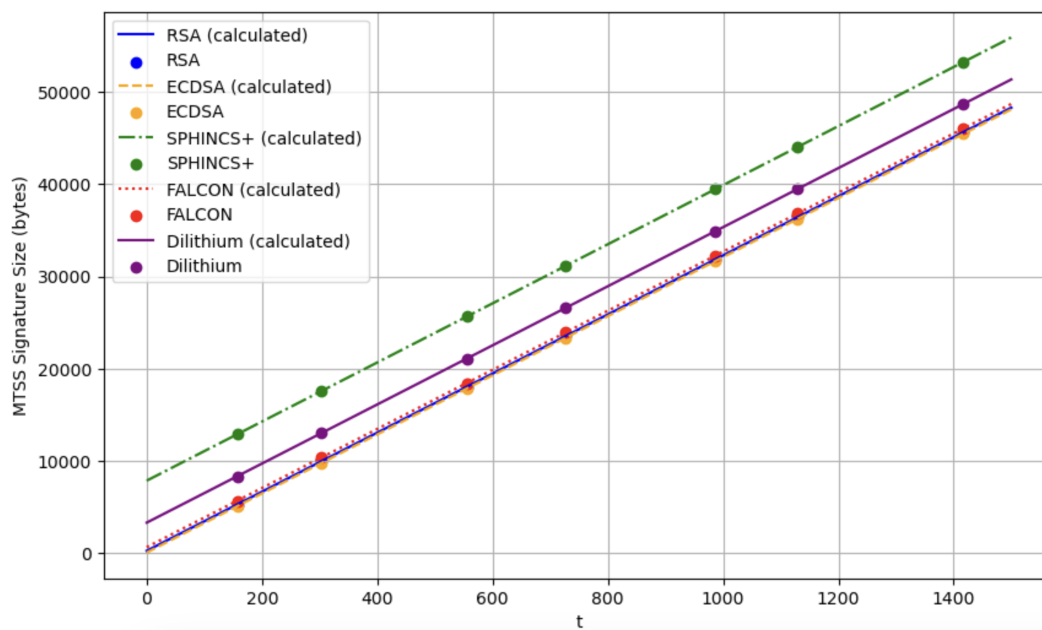

Figure 6.1: SHA2/3-256: Comparison of MICO signature size for different CDSS.

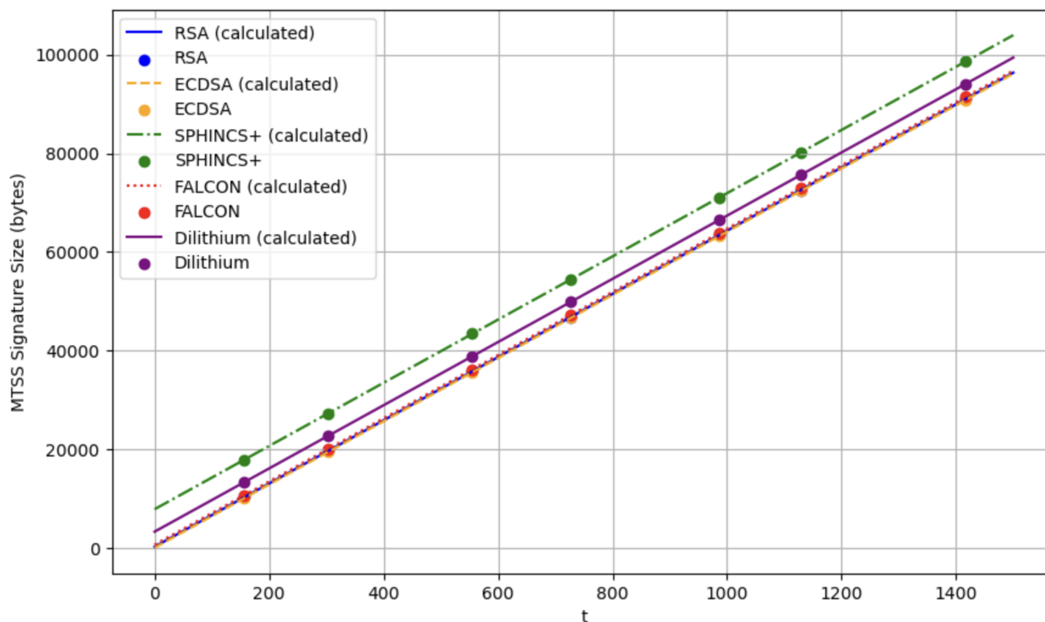

Figure 6.2: SHA2/3-512: Comparison of MICO signature size for different CDSS.

Figure 6.1 and Figure 6.2 display the MICO signature sizes for five digital schemes based on SHA2/3-256 and SHA2/3-512, respectively. The straight lines represent the signature sizes calculated using the equations from Table 6.7, while the dots indicate the signature sizes obtained from file experiments at specific t-values: 157, 303, 555, 727, 987, 1129, and 1417. The comparison between $t = 157$ and $t = 1417$ is already shown in Tables 6.8 and 6.9. The close alignment between the lines and dots indicates that the calculated sizes match the experimental results well, with only slight differences seen in schemes like ECDSA and FALCON, as previously noted.

6.4 Efficiency Analysis of the MICO scheme

In evaluating the performance of the MICO scheme, it is crucial to consider not only the signature size of the scheme, but also the time required to generate keys, sign, and verify. Therefore, in this section, our goal is to provide a comprehensive understanding of the scheme's performance across various scenarios in terms of running time and signature size, thereby validating its feasibility and effectiveness. We run tests with three different file sizes—small, medium, and large—for two types of files: text files and images, as detailed in Section 6.1. Each test in this section is run 10 times.

6.4.1 Evaluating the Impact of n and d on Signing Time

In this subsection, we examine how the number of blocks n and the value of d affect the signing time and the signature size of the MICO scheme, while using the CDSS and hash

function that produce the shortest signature size.

From previous experiments, we have determined that ECDSA with SHA2-256 or SHA3-256 should be used for the shortest signature size. Additionally, we have also observed that SHA3-256 hashes quicker compared to SHA2-256 when comparing the hashing time among hash functions. Therefore, in our experiments, we have decided to make use of ECDSA and SHA3-256 in the MICO scheme.

First, we run tests for $d = 1$ using Sperner set systems and Table 6.10 shows the set-ups for our tests.

CDSS	ECDSA
Hash algorithm	SHA3-256
CFF Method	Sperner
CFF data structure	MATRIXCOMPACT

Table 6.10: Set-ups for Sperner and $d = 1$.

File Name	n	t	Signature Size (bytes)	Signing Time (ms)
Text_SM_40000	200	10	424	78.43
	4,000	15	584	105.95
	10,000	16	614	108.28
Text_MED_200000	200	10	424	351.56
	4,000	15	562	462.98
	40,000	18	679	587.93
Text_LAR_1000000	200	10	424	1695.52
	4,000	15	583	2,451.21
	200,000	21	776	3,321.73

Table 6.11: Signing performance for different text files using Sperner set systems.

File Name	n	t	Signature Size (bytes)	Signing Time (ms)
Image_SM_200_200	100	9	392	4.87
	2,500	14	550	6.43
	10,000	16	615	10.55
Image_MED_500_500	100	9	392	41.19
	2,500	14	552	42.58
	62,500	19	710	62.36
Image_LAR_1000_1000	100	9	392	90.62
	2,500	14	552	116.41
	250,000	21	775	182.91

Table 6.12: Signing performance for different image files using Sperner set systems.

According to Tables 6.11 and 6.12, we can observe several interesting trends in the signing performance across different file sizes and block numbers. As file sizes increase, signing times also rise, even when the number of blocks remains constant. This suggests that file size significantly affects signing time. Additionally, for each file size, increasing the number of blocks n leads to larger signature sizes and longer signing times, with the number of rows t also growing correspondingly.

Next, we test the signing performance of the MICO scheme while using Steiner Triple Systems for $d = 2$ and we use the set-ups shown in Table 6.13.

CDSS	ECDSA
Hash algorithm	SHA3-256
CFF Method	STS
CFF data structure	MatrixCompact

Table 6.13: Set-ups for STS and $d = 2$.

File Name	n	t	Signature Size (bytes)	Signing Time (ms)
Text_SM_40000	200	37	1,288	58.88
	4,000	157	5,128	66.57
	10,000	247	8,007	69.99
Text_MED_200000	200	37	1,288	253.83
	4,000	157	5,126	257.01
	40,000	493	15,879	302.53
Text_LA_1000000	200	37	1,288	1,196.32
	4,000	157	5,127	1,266.79
	200,000	1,099	35,270	1,770.62

Table 6.14: Signing performance for different text files using STS.

File Name	n	t	Signature Size (bytes)	Signing Time (ms)
Image_SM_200_200	100	25	903	5.10
	2,500	123	4,038	14.91
	10,000	247	8,007	25.86
Image_MED_500_500	100	25	904	28.99
	2,500	123	4,039	34.56
	62,500	613	19,719	124.26
Image_LAR_1000_1000	100	25	903	78.65
	2,500	123	4,040	79.10
	250,000	1,227	39,366	672.83

Table 6.15: Signing performance for different image files using STS.

For both text files and images, as n increases, t also increases, which means the signing time gets longer. This can be seen in Tables 6.14 and 6.15 for $d = 2$. For example, for an image file with dimensions 500×500, when n increases from 2,500 to 62,500, t goes up from 123 to 613, and the signing time increases by 4 times. Additionally, when n and t are the same, the signing time also has a positive relation with the file size. For example, for $n = 4000$ and $t = 157$, the signing time increases by 6 times as the file size goes from 200,000 lines to 1,000,000 lines.

Now we run tests for constructing d-CFFs from Reed-Solomon codes for $d \geq 2$. For each file size, we run tests on three different values of n with three calculated values of d: $\sqrt[6]{n}$, $\ln n$, and $4 \log n$. These values are rounded. Table 6.16 is the overview of the setups for tests.

CDSS	ECDSA
Hash algorithm	SHA3-256
CFF Method	Reed-Solomon codes
CFF data structure	MatrixCompact

Table 6.16: Set-ups for Reed Solomon codes (RS codes).

Tables 6.17, 6.18, and 6.19 display the signing time for three text files: Text_SM_40000, Text_MED_200000, and Text_LAR_1000000, with different n and d.

Number of Blocks (n)	d	t	k	N	q	Signature Size (bytes)	Signing Time (ms)
Small: 200 blocks	2	35	3	5	7	1,222	77.30
	5	102	2	6	17	3,367	85.99
	9	170	2	10	17	5,544	142.99
Medium: 4,000 blocks	4	153	3	9	17	5,000	122.46
	8	289	3	17	17	9,352	215.87
	14	841	3	29	29	27,015	337.10
Large: 10,000 blocks	5	253	3	11	23	8,198	146.88
	9	437	3	19	23	14,088	229.75
	16	1,221	3	33	37	39,176	415.27

Table 6.17: Signing performance for Text_SM_40000 using RS codes.

Number of Blocks (n)	d	t	k	N	q	Signature Size (bytes)	Signing Time (ms)
Small: 200 blocks	2	35	3	5	7	1,224	360.62
	5	102	2	6	17	3,367	409.68
	9	170	2	10	17	5,544	598.53
Medium: 4,000 blocks	4	153	3	9	17	4,998	569.20
	8	289	3	17	17	9,350	1,023.29
	14	841	3	29	29	27,015	1,557.51
Large: 40,000 blocks	6	361	4	19	19	11,656	1,126.29
	11	851	3	23	37	27,334	1,466.98
	18	1,369	3	37	37	43,911	2,132.41

Table 6.18: Signing performance for Text_MED_200000 using RS codes.

Number of Blocks (n)	d	t	k	N	q	Signature Size (bytes)	Signing Time (ms)
Small: 200 blocks	2	35	3	5	7	1,224	1,746.39
	5	102	2	6	17	3,367	2,034.46
	9	170	2	10	17	5,544	3,037.02
Medium: 4,000 blocks	4	153	3	9	17	4,999	2,886.67
	8	289	3	17	17	9,352	5,031.65
	14	841	3	29	29	27,014	9,758.71
Large: 200,000 blocks	8	725	4	25	29	23,303	8,892.22
	12	1,369	4	37	37	43,910	13,525.03
	21	2,537	3	43	59	81,287	16,385.75

Table 6.19: Signing performance for Text_LAR_1000000 using RS codes.

From Tables 6.17 to 6.19, we observe that when n and d remain constant, the running time increases dramatically across the three different text file sizes. For example, when $n = 200$ and $d = 2$, the signing time increases from 0.08 seconds to 0.36 seconds as the file size increases from 40,000 lines to 200,000 lines. The signing time further increases to 1.7 seconds if the text file is increased to 1,000,000 lines. Moreover, when either n or d increases, both the signing time and signature size increase significantly. For instance, in Table 6.19, when $n = 200,000$ and d is increased from 8 to 21, the signing time doubles.

Next, we conduct tests with three different image files: a small image called Image_SM_200_200, a medium image called Image_MED_500_500, and a large image called Image_LAR_1000_1000, as shown in Table 6.20, 6.21, and 6.22.

Number of Blocks (n)	d	t	k	N	q	Signature Size (bytes)	Signing Time (ms)
Small: 100 blocks	2	25	3	5	5	903	5.27
	5	66	2	6	11	2,216	10.13
	8	99	2	9	11	3,271	16.91
Medium: 2,500 blocks	4	153	3	9	17	4,999	19.12
	8	289	3	17	17	9,352	22.78
	14	795	2	15	53	25,544	27.99
Large: 10,000 blocks	5	253	3	11	23	8198	24.49
	9	437	3	19	23	14,087	32.66
	16	1,221	3	33	37	39,174	71.37

Table 6.20: Signing performance for Image_SM_200_200 using RS codes.

Number of Blocks (n)	d	t	k	N	q	Signature Size (bytes)	Signing Time (ms)
Small: 100 blocks	2	25	3	5	5	903	31.98
	5	66	2	6	11	2,216	33.79
	8	99	2	9	11	3,272	37.37
Medium: 2,500 blocks	4	153	3	9	17	5,000	42.41
	8	289	3	17	17	9,351	48.65
	14	795	2	15	53	25,544	51.70
Large: 62,500 blocks	6	361	4	19	19	11,655	140.88
	11	943	3	23	41	30,278	217.61
	19	1,599	3	39	41	51,272	347.23

Table 6.21: Signing performance for Image_MED_500_500 using RS codes.

Number of Blocks (n)	d	t	k	N	q	Signature Size (bytes)	Signing Time (ms)
Small: 100 blocks	2	25	3	5	5	902	84.90
	5	66	2	6	11	2,215	87.81
	8	99	2	9	11	3,270	97.14
Medium: 2,500 blocks	4	153	3	9	17	4,998	117.97
	8	289	3	17	17	9,352	138.83
	14	795	2	15	53	25,543	167.33
Large: 250,000 blocks	8	725	4	25	29	23,304	818.17
	12	1,369	3	37	37	43,911	1,874.97
	22	3,015	3	45	67	96,583	2,590.94

Table 6.22: Signing performance for Image_LAR_1000_1000 using RS codes.

Based on Tables 6.20 to 6.22, similar to text files, we observe that as d or n increases, so does the signing time and signature size for the same image. Moreover, among different images, the signing time rises noticeably for the same d and n, while the signature size remains the same.

Furthermore, we analyze the time required for different steps of the scheme in the signing process as described in Section 5.4. This analysis helps us understand which step dominates the signing time and how increasing d or n affects the time for each step.

In Table 6.23, we measure the signing time for each step with increasing values of d when $n = 4000$ for Text_SM_40000. The total signing time also appear in rows 5-7 in Table 6.17.

	Steps	$d = 4$ $t = 153$ (ms)	$d = 8$ $t = 289$ (ms)	$d = 14$ $t = 841$ (ms)
Pre-step	divide message into blocks	11.25	11.02	11.12
	construct d-CFF	0.49	0.49	2.60
Key generation	generate the key pair	2.70	2.77	2.86
Sign	hash entire message	9.40	10.12	9.82
	concatenating according to d-CFF	8.39	9.92	16.01
	hash the t block concatenations	84.82	172.25	280.56
	signing preparations	4.65	7.96	12.13
	sign with SK	0.76	1.34	1.20
Total		122.47	215.87	337.10

Table 6.23: Breakdown for signing time of Text_SM_40000 when $n = 4,000$.

Table 6.24 measures the signing time for each step with increasing values of n when $d = 8$ for Image_LAR_1000_1000. The total signing time also appear in rows 4,6, and 8 in Table 6.22.

	Steps	$n = 100$ $t = 99$ (ms)	$n = 2,500$ $t = 289$ (ms)	$n = 250,000$ $t = 725$ (ms)
Pre-step	divide message into blocks	46.63	47.02	50.34
	construct d-CFF	0.15	0.46	78.73
Key generation	generate the key pair	2.57	2.67	2.69
Sign	hash entire message	8.27	7.99	8.37
	concatenating according to d-CFF	0.94	4.05	563.53
	hash the t block concatenations	35.56	70.99	99.95
	signing preparations	2.23	4.82	12.82
	sign with SK	0.79	0.84	1.73
Total		97.14	138.83	818.17

Table 6.24: Breakdown for signing time of Image_LAR_1000_1000 when $d = 8$.

From Table 6.23 and Table 6.24, it is clear that increases in either d or n do not affect the time required for message division, key generation, or hashing the entire message.

Additionally, the time taken to generate the key pair and sign with SK during the signing process depends only on the underlying CDSS. In Table 6.23, for a relatively small $n = 4000$, the time for hashing the concatenated blocks dominates. However, in Table 6.24, when $n = 100$, dividing the message into blocks takes most of the time, while with $n = 250000$, the concatenation according to d-CFF becomes the most time-consuming process. Furthermore, the running time for hashing the entire message is much longer than the average hashing time used for concatenated blocks due to the differences in message sizes. This observation aligns with the performance of hash algorithms discussed in Section 6.2.2. Therefore, we can conclude that the value of t and the message size are two key factors influencing the signing time. To further reduce the signing time, a faster but potentially weaker cryptographic hash function could be used. It would be interesting to study the effect of a weaker hash function on the overall efficiency and security of the MICO scheme.

6.4.2 Impact of Other Parameter Changes on Signing Time

Now that we understand how n and d affect the efficiency of the MICO scheme and identify the most time-consuming steps in the signing process for different cases, we can analyze other parameters. This section examines how changes in the underlying CDSS, hash functions, and the CFF data structure influence signing time and signature size for a specific file with certain values of d and n.

The data in the first row of both tables are highlighted in bold and are from previous experiments, derived from Table 6.17 and Table 6.22. Moreover, the remaining rows each have one parameter different from the first row.

CDSS	Hash	CFF data structure	Signing Time (ms)	Signature Size (bytes)
ECDSA	**SHA3-256**	**MatrixCompact**	**215.87**	**9,352**
ECDSA	SHA2-256	MatrixCompact	308.20	9,351
ECDSA	SHA3-256	MatrixList	221.24	9,352
RSA	SHA3-256	MatrixCompact	318.26	9,536
FALCON	SHA3-256	MatrixCompact	219.09	9,939
Dilithium	SHA3-256	MatrixCompact	191.82	12,589
SPHINCS$^+$	SHA3-256	MatrixCompact	1,572.15	17,136
ECDSA	SHA2-512	MatrixCompact	226.90	18,631
ECDSA	SHA3-512	MatrixCompact	337.01	18,632

Table 6.25: Signing performance for Text_SM_40000 when $d = 8$ and $n = 4000$.

Table 6.25 shows that for a small text file with 40,000 lines, with $d = 8$ and $n = 4000$, changing CFF data structures does not significantly affect the signing time. However, when we change SHA3-256 to SHA2-256, the signing time increases noticeably. Moreover, if we replace SHA3-256 with either SHA2-512 or SHA3-512, the signature size approximately doubles. The signing time for SHA2-512 stays roughly the same, while the signing time for SHA3-512 increases dramatically. Additionally, replacing ECDSA with other underlying CDSS, except for Dilithium and FALCON, significantly increases the signing time. For

instance, using SPHINCS$^+$ increases the signing time more than seven times due to its longer signing process. Furthermore, using other signature schemes also increase the signature size.

Therefore, for a relatively small text file with a small number of blocks n, if we want a smaller signature size with a quicker signing speed, the setup in the first row, which uses ECDSA, SHA3-256, and MatrixCompact, is the best choice. If considering quantum-safe options, replace ECDSA with Dilithium for a quicker signing time, or replace ECDSA with FALCON for a shorter signature size. Finally, if we want to use a hash function with a higher security level, choose SHA2-512 rather than SHA3-512.

CDSS	Hash	CFF data structure	Signing Time (ms)	Signature Size (bytes)
ECDSA	**SHA3-256**	**MatrixCompact**	**818.17**	**23,304**
ECDSA	SHA2-256	MatrixCompact	871.59	23,303
ECDSA	SHA3-256	MatrixList	825.77	23,302
RSA	SHA3-256	MatrixCompact	914.93	23,488
FALCON	SHA3-256	MatrixCompact	868.43	23,886
Dilithium	SHA3-256	MatrixCompact	836.47	26,541
SPHINCS$^+$	SHA3-256	MatrixCompact	2,201.11	31,088
ECDSA	SHA2-512	MatrixCompact	813.40	46,534
ECDSA	SHA3-512	MatrixCompact	884.48	46,535

Table 6.26: Signing performance for Image_LAR_1000_1000 when $d = 8$ and $n = 250000$.

Moreover, for a image with dimensions 1000×1000 while $n = 250000$ and $d = 8$, as shown in Table 6.26, changing the CFF data structure also does not significantly affect the signing time. Replacing SHA3-256 with SHA2-512 results in a quicker signing time, although the signature size nearly doubles. Additionally, when only one variable is changed compared to row 1, replacing ECDSA with SPHINCS$^+$ results in a significantly greater signing time than any other changes. This is due to its slow signing time.

Based on these observations, we can conclude that SPHINCS$^+$ should be avoided as the underlying digital signature for similar values of n and d when aiming for quantum resistance. Instead, FALCON can be selected for a shorter signature size or Dilithium for faster signing time, depending on the specific needs. For underlying hash functions, SHA2-512 can be chosen for a higher security level, while SHA3-256 can be selected for a shorter digest size.

6.4.3 Evaluating the Impact of n and d on Verification Time

Section 6.4.1 analyzes the impact of n and d on the signing time while using the CDSS and hash function that produce the shortest signature size. In this section, we measures their impact on the verification time. All the tests in this section involve modifications to ensure the MICO scheme goes through the entire verification process, and the number of modifications is exactly the value of d.

We use the same setups and files from Section 6.4.1 to test the verification performance of the MICO scheme. We begin by testing the efficiency for $d = 1$ using Sperner set systems,

then for $d = 2$ using Steiner Triple Systems, and finally for various $d \geq 2$ using Reed-Solomon codes. Additionally, we measure the time required for different decoding methods, including the general and specific decoding methods detailed in Chapter 3.

Tables 6.27 and 6.28 present the verification times for two decoding methods applied to two different types of files with different sizes and values of n for $d = 1$ using Sperner set systems.

File Name	n	t	MatrixCompact (ms)	SpernerDecode (ms)
Text_SM_40000	200	10	79.201	79.13
	4,000	15	91.51	90.84
	10,000	16	115.97	114.34
Text_ME_200000	200	10	352.57	352.53
	4,000	15	458.82	458.05
	40,000	18	581.89	578.80
Text_LA_1000000	200	10	1,695.85	1,695.77
	4,000	15	2,263.03	2,262.60
	200,000	21	3,071.34	3,059.16

Table 6.27: Verification performance for different text files using Sperner set systems.

File Name	n	t	MatrixCompact (ms)	SpernerDecode (ms)
Image_SM_200_200	100	9	5.27	5.27
	2,500	14	6.56	5.91
	10,000	16	9.16	7.86
Image_ME_500_500	100	9	23.90	23.84
	2,500	14	24.42	23.90
	62,500	19	60.87	57.78
Image_LA_1000_1000	100	9	79.87	79.82
	2,500	14	89.68	89.10
	250,000	21	196.97	184.78

Table 6.28: Verification performance for different image files using Sperner set systems.

From Table 6.27, we see that the verification time goes up as n or file size increases. For example, for a large text of 1,000,000 lines, increasing n from 200 to 200,000 results in an increase of approximately 1.3 seconds. The same trend is shown in Table 6.28. Additionally, the specific decoding method, SpernerDecode, is faster than the general decoding method, MatrixCompact. As n increases, the time difference between the two methods becomes larger.

Using STS for $d = 2$, Tables 6.29 and 6.30 show the verification times for two decoding methods applied to different types of files with varying sizes and values of n.

File Name	n	t	MatrixCompact (ms)	STSDecode (ms)
Text_SM_40000	200	37	53.55	53.51
	4,000	157	55.33	55.71
	10,000	247	57.28	57.75
Text_ME_200000	200	37	253.94	253.79
	4,000	157	255.51	256.03
	40,000	493	318.56	318.95
Text_LA_1000000	200	37	1,213.26	1,213.16
	4,000	157	1,272.63	1,273.02
	200,000	1,099	1,787.59	1,782.72

Table 6.29: Verification performance for different text files using STS.

File Name	n	t	MatrixCompact (ms)	STSDecode (ms)
Image_SM_200_200	100	25	8.80	8.60
	2,500	123	12.34	12.55
	10,000	247	16.54	16.75
Image_ME_500_500	100	25	20.12	20.00
	2,500	123	20.75	20.91
	62,500	613	114.04	114.33
Image_LA_1000_1000	100	25	74.20	74.02
	2,500	123	74.61	74.90
	250,000	1,227	642.82	639.10

Table 6.30: Verification performance for different image files using STS.

In Table 6.29, the verifying time increases as n or file size increases. For $n = 200$ and 200,000, STSDecode is faster than MatrixCompact. However, MatrixCompact is faster for $n =$ 4,000, 10,000, and 40,000. This pattern is also observed in Table 6.30. Therefore, we can conclude that for relatively small and large n, STSDecode is quicker than MatrixCompact. However, when n falls in the middle range, MatrixCompact is faster. Moreover, both the file size and the value of n affect the verifying time. For example, for $n = 2500$, if n increases to 10,000 for a small image, the verifying time increases from 0.12 seconds to 0.16 seconds for both decoding methods. If we change the file size from small to medium for $n = 2500$, the verifying time increases from 0.12 seconds to 0.21 seconds.

Tables 6.31 to 6.33 present the verification times for three different sizes of text files with various d values using Reed-Solomon codes.

Number of Blocks (n)	d	t	k	N	q	MatrixCompact (ms)	RSDecode (ms)
Small: 200 blocks	2	35	3	5	7	70.62	70.67
	5	102	2	6	17	89.50	89.61
	9	170	2	10	17	119.38	119.45
Medium: 4,000 blocks	4	153	3	9	17	116.87	117.10
	8	289	3	17	17	202.83	203.05
	14	841	3	29	29	321.63	321.82
Large: 10,000 blocks	5	253	3	11	23	141.90	142.02
	9	437	3	19	23	225.90	225.97
	16	1,221	3	33	37	391.71	391.87

Table 6.31: Verification performance for Text_SM_40000 using RS codes.

Number of Blocks (n)	d	t	k	N	q	MatrixCompact (ms)	RSDecode (ms)
Small: 200 blocks	2	35	3	5	7	369.42	369.48
	5	102	2	6	17	394.94	395.13
	9	170	2	10	17	597.16	597.25
Medium: 4,000 blocks	4	153	3	9	17	545.01	545.30
	8	289	3	17	17	1024.35	1024.51
	14	841	3	29	29	1673.11	1673.17
Large: 40,000 blocks	6	361	4	19	19	1157.10	1157.20
	11	851	3	23	37	1,371.17	1,371.12
	18	1,369	3	37	37	2,115.90	2,115.53

Table 6.32: Verification performance for Text_MED_200000 with RS codes.

Number of Blocks (n)	d	t	k	N	q	MatrixCompact (ms)	RSDecode (ms)
Small: 200 blocks	2	35	3	5	7	1692.276	1692.332
	5	102	2	6	17	1960.50	1960.69
	9	170	2	10	17	2991.38	2991.52
Medium: 4,000 blocks	4	153	3	9	17	2799.20	2799.56
	8	289	3	17	17	4,863.62	4,863.83
	14	841	3	29	29	8,322.18	8,323.23
Large: 200,000 blocks	8	725	4	25	29	8,425.15	8,424.22
	12	1,369	4	37	37	14,095.29	14,089.16
	21	2,537	3	43	59	15,997.14	15,961.70

Table 6.33: Verification performance for Text_LAR_1000000 with RS codes.

According to Tables 6.31 to 6.33, as the message size, d, or n increases, the decoding time also increases noticeably. For instance, if we increase a file size from 200,000 to 1,000,000 lines when $d = 8$ and $n = 4000$, the decoding time rises more than four times. Additionally, if we increase n from 4,000 to 200,000 for a large text file, the signing time doubles for

$d = 8$. Moreover, for $n = 200$ and $n = 4000$, MatrixCompact performs slightly better, whereas RSDecode becomes more efficient when n reaches 200,000. This trend has also been discussed in Chapter 3.

Similarly, Tables 6.34 to 6.36 show the verification times for three different image sizes using Reed-Solomon codes.

Number of Blocks (n)	d	t	k	N	q	MatrixCompact (ms)	RSDecode (ms)
Small: 100 blocks	2	25	3	5	5	4.87	4.93
	5	66	2	6	11	5.32	5.35
	8	99	2	9	11	10.59	10.71
Medium: 2,500 blocks	4	153	3	9	17	8.74	8.94
	8	289	3	17	17	12.53	12.64
	14	795	2	15	53	13.08	13.32
Large: 10,000 blocks	5	253	3	11	23	12.58	12.34
	9	437	3	19	23	22.79	22.49
	16	1,221	3	33	37	50.94	50.49

Table 6.34: Verification performance for Image_SM_200_200 using RS codes.

Number of Blocks (n)	d	t	k	N	q	MatrixCompact (ms)	RSDecode (ms)
Small: 100 blocks	2	25	3	5	5	27.32	27.43
	5	66	2	6	11	28.97	29.01
	8	99	2	9	11	31.26	31.37
Medium: 2,500 blocks	4	153	3	9	17	34.62	34.80
	8	289	3	17	17	40.43	40.54
	14	795	2	15	53	43.26	43.39
Large: 62,500 blocks	6	361	4	19	19	133.55	133.35
	11	943	3	23	41	199.84	199.69
	19	1,599	3	39	41	345.24	344.79

Table 6.35: Verification performance for Image_MED_500_500 using RS codes.

Number of Blocks (n)	d	t	k	N	q	MatrixCompact (ms)	RSDecode (ms)
Small: 100 blocks	2	25	3	5	5	81.45	81.54
	5	66	2	6	11	87.67	87.72
	8	99	2	9	11	97.27	97.30
Medium: 2,500 blocks	4	153	3	9	17	101.60	101.72
	8	289	3	17	17	129.13	129.21
	14	795	2	15	53	140.51	140.65
Large: 250,000 blocks	8	725	4	25	29	798.70	798.33
	12	1,369	3	37	37	1,318.90	1,317.68
	22	3,015	3	45	67	2,409.19	2,405.17

Table 6.36: Verification performance for Image_LAR_1000_1000 using RS codes.

Tables 6.34 to 6.36 demonstrate that changes in the decoding method do not significantly affect the verification time while other variables are kept the same. When n is small, the general decoding method, MatrixCompact, always performs slightly better. As n grows larger, the specific method, RSDecode, shows a slight advantage. On the other hand, as d or n increases using the same image file, the verification time increases dramatically. Additionally, the file size also has a significant impact on verification time while d and n remain the same. For example, for $n = 2500$ and $d = 8$, the verification time increases from 0.04 seconds to 0.13 seconds when the image size changes from medium to large for both decoding methods.

Similar to the experiments conducted for signing time, we also measure the time for each verification step, as shown in Table 6.37 for a text file with 40,000 lines when $n = 4000$ and in Table 6.38 for an image file with dimensions 1000×1000 when $d = 8$.

	Steps	$d = 4$ $t = 153$ (ms)	$d = 8$ $t = 289$ (ms)	$d = 14$ $t = 841$ (ms)
Pre-step	divide message into blocks	11.76	11.66	10.62
	construct d-CFF	0.340	0.41	2.57
Verify	verify signature	0.98	0.89	1.37
	check modifications	9.62	9.83	10.13
	concatenating according to d-CFF	8.28	10.60	16.17
	hash the t block concatenations	85.72	169.28	280.55
	locate modifications: MatrixCompact	0.11	0.16	0.22
	locate modifications: RSDecode	0.34	0.38	0.41
Total	MatrixCompact	116.87	202.83	321.63
	RSDecode	117.10	203.05	321.82

Table 6.37: Breakdown of verification performance for Text_SM_40000 when $n = 4000$.

	Steps	$n = 100$ $t = 99$ (ms)	$n = 2,500$ $t = 289$ (ms)	$n = 250,000$ $t = 725$ (ms)
Pre-step	divide message into blocks	51.07	46.51	44.68
	construct d-CFF	0.22	0.45	77.44
Verify	verify signature	1.08	1.10	1.49
	check modifications	8.18	8.21	8.82
	concatenating according to d-CFF	0.90	3.90	565.49
	hash the t block concatenations	35.77	68.40	99.43
	locate modifications: MatrixCompact	0.05	0.56	1.35
	locate modifications: RSDecode	0.08	0.64	0.98
Total	MatrixCompact	97.27	129.13	798.70
	RSDecode	97.30	129.21	798.33

Table 6.38: Breakdown of verification performance for Image_LAR_1000_1000 when $d = 8$.

The step in which the MICO scheme verifies the signature in Tables 6.37 and 6.38 depends only on the underlying CDSS. According to both tables, among all the verification steps, hashing the concatenated blocks takes the most time in the majority of cases, except for n =100 and $250,000$ in Table 6.38. In these exceptions, dividing the message into blocks is the most time-consuming step for the former, while the concatenation according to d-CFF takes the most time for the latter. This observation is what we have seen in the signing process. Since the time for hashing concatenation depends on the value of t, both the message size and t are crucial factors influencing verification time.

6.4.4 Impact of Other Parameter Changes on Verifying Time

In this section, we analyze how changing various parameters, including the underlying CDSS, hash functions, and the CFF data structure, affects the verification time for a specific file with certain values of d and n, as we have done for signing time. The data in the first and second rows of both tables are derived from previous tests in Table 6.37 and 6.38 and are highlighted in bold. Each of the remaining rows differs by one parameter compared to row 1.

CDSS	Hash	Decoding method	Verifying time (ms)
ECDSA	**SHA3-256**	**MatrixCompact**	**202.83**
ECDSA	**SHA3-256**	**RSdecode**	**203.05**
ECDSA	SHA2-256	MatrixCompact	303.46
ECDSA	SHA3-256	MatrixList	207.56
RSA	SHA3-256	MatrixCompact	200.45
FALCON	SHA3-256	MatrixCompact	194.77
Dilithium	SHA3-256	MatrixCompact	204.30
SPHINCS$^+$	SHA3-256	MatrixCompact	205.36
ECDSA	SHA2-512	MatrixCompact	192.41
ECDSA	SHA3-512	MatrixCompact	327.91

Table 6.39: Verification time for Text_SM_40000 when $d = 8$ and $n = 4,000$.

The verifying time for a small text file of 40,000 lines when $d = 8$ and $n = 4000$ does not vary much if we change digital signatures or the decoding method, as shown in Table 6.39. However, if we replace SHA3-256 with either SHA3-512 or SHA2-256, the verification time increases significantly. With that being said, the signer should also keep in mind avoiding the use of SHA2-256 and SHA3-512 when choosing parameters, as verifiers must follow the same setups to verify the signature.

CDSS	Hash	Decoding method	Verifying time (ms)
ECDSA	**SHA3-256**	**MatrixCompact**	**798.70**
ECDSA	**SHA3-256**	**RSDecode**	**798.33**
ECDSA	SHA2-256	MatrixCompact	842.41
ECDSA	SHA3-256	MatrixList	891.60
RSA	SHA3-256	MatrixCompact	787.62
FALCON	SHA3-256	MatrixCompact	806.80
Dilithium	SHA3-256	MatrixCompact	813.25
SPHINCS$^+$	SHA3-256	MatrixCompact	799.98
ECDSA	SHA2-512	MatrixCompact	799.77
ECDSA	SHA3-512	MatrixCompact	904.83

Table 6.40: Verification time for Image_LAR_1000_1000 when $d = 8$ and $n = 250,000$.

According to Table 6.40, in most cases, when we change only one parameter compared to row 1, the time remains almost the same, except when we replace SHA3-256 with SHA2-256 or SHA3-512, and when we replace the CFF data structure from compact to list representation. As a result, the underlying CDSS does not substantially affect the verification time with an image file of 1000×1000 when $d = 8$ and $n = 250000$. On the other hand, it is the CFF data structure and underlying hash functions that affect the verification time. Therefore, when a signer signs an image with similar d and n, they should be careful when choosing the CFF data structure and hash functions, as these choices will impact the verification time as well.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis aimed to explore the theoretical foundations and practical implementations of d-modification tolerant signature schemes (d-MTSS) using non-adaptive combinatorial group testing (CGT) and established cryptographic principles. Our main goals were to develop efficient algorithms and data structures for CGT using d-CFFs and to assess the performance of d-MTSS across different scenarios. Reflecting on our work, we can draw several key conclusions that show our contributions and suggest paths for future research in this domain.

One of our main accomplishments was developing efficient data structures and algorithms for non-adaptive combinatorial group testing, which we have discussed in Chapter 3 of the thesis. We introduced the MatrixList and MatrixCompact data structures, and specialized decoding algorithms for Sperner sets, Steiner Triple System and Reed-Solomon codes. These innovations have greatly improved our ability to locate modifications (or defects), particularly with the specific decoding algorithms for Sperner sets and Reed-Solomon codes. They allow us to locate defectives from a large number of items without actually building sets or codes, which significantly reduce computation time. This achievement addresses our objective of making d-MTSS more practical for applications. The main results in this chapter was published in [29].

Another significant aspect of our research was integrating d-MTSS with existing cryptographic primitives. By implementing d-MTSS with both classical and post-quantum digital signatures, we have demonstrated that the MICO scheme can adapt to both current and future cryptographic standards. As mentioned in Chapter 4 and 5, the security of d-MTSS depends entirely on these cryptographic building primitives, making this integration crucial for its long-term effectiveness and security, especially as we move into the era of quantum-safe cryptography.

Finally, our comprehensive experiments in Chapter 6 have given us important insights into how the MICO scheme, our implementation of d-MTSS, performs. We have identified the most time-consuming steps and observed how different settings affect the scheme's efficiency in various scenarios. These results are key for refining d-MTSS and will certainly influence

future research in this field. The differences we saw in performance depending on the settings show that d-MTSS can be adjusted for specific needs, balancing between security, signature size and efficiency.

7.2 Future Work

Looking ahead, there are several paths for future research. First, optimizing the time-consuming processes we identified could make d-MTSS even more efficient. For example, as noted in Chapter 6, hashing concatenated blocks is particularly slow for large value of n. Exploring weaker hash functions could speed this up, but one would need to carefully study how that affects security and efficiency of d-MTSS. Additionally, other methods for constructing d-CFFs have been proposed [5, 17, 26, 34, 40] that might give fewer tests t for the same number of blocks n and number of allowed modifications d. These could potentially reduce the time needed for the block concatenations according to d-CFF in the scheme, further enhancing its efficiency.

Another direction for future research is the application of combinatorial group testing using cover-free families on hypergraphs [18, 32]. While traditional group testing problems typically consider all d-subsets of an n-set as the potential sets of defective items, generalizing to hypergraphs could significantly enhance the efficiency of d-MTSS implementation, particularly for handling file types such as images. In our current approach for images, we divide the image into n blocks and detect up to d modifications among these individual blocks. This method, however, can become computationally expensive for large n, as it essentially considers all possible combinations of d blocks out of n. Images, however, have an structure where modifications are likely to affect adjacent blocks. Therefore, instead of aiming at locating any d modified blocks, we can concentrate on locating modifications on groups of adjacent blocks. This approach allows us to reduce the number of tests required or locate a larger number of modifications more efficiently.

As illustrated by Idalino et al. [20], d-MTSS has the potential to not only detect but also correct modifications. This capability offers a significant improvement over existing schemes. By keeping the size s of each block sufficiently small, it becomes possible to perform a brute-force search to correct the modified block, ensuring the hash of the concatenation of a specific group of blocks matches the original hash. This process involves 2^s steps for each modified block, assuming a collision-resistant hash function, and while computationally demanding, it remains practical for small-scale modifications. Moreover, the efficiency of the scheme will be further improved if one could find a way to predict the original values. Since a d-MTSS with correction capabilities is very useful for mitigating damage in documents, it is worthwhile to study and implement such scheme. An initial implementation in Python has been explored in an undergraduate project [1], providing a foundation for further development. Our Java implementation of d-MTSS developed for this thesis could be extended to allow for correction of modifications.

Finally, an open-source Java implementation of the MICO scheme using the Bouncy Castle library is available for free at [28]. This code shows that our ideas work in practice

and gives other researchers a starting point for their own work. We built it so that different parts can be easily changed or replaced. This means users can add new d-CFF constructions, try different data structures and decoding methods, or use other digital signature schemes without having to rewrite everything. We hope this will help researchers test new ideas and develop new uses for modification-tolerant signatures in their research.

Bibliography

[1] Paola de Oliveira Abel. Esquema de assinatura digital tolerante a modificações. Undergraduate project, Universidade Federal de Santa Catarina, Florianópolis, 2023. Honour's Bachelor of Computer Science.

[2] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology–CT-RSA 2014: The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, pages 28–47. Springer International Publishing, 2014.

[3] Afonso S. Bandeira. Ten lectures and forty-two open problems in the mathematics of data science. `https://ocw.mit.edu/courses/18-s096-topics-in-mathematics-of-data-science-fall-2015/resources/mit18_s096f15_tenlec/`, December 2015. MIT OpenCourseWare.

[4] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ signature framework. Cryptology ePrint Archive, Paper 2019/1086, 2019.

[5] Nader H. Bshouty. Linear time constructions of some d-restriction problems. In Vangelis Th. Paschos and Peter Widmayer, editors, *International Conference on Algorithms and Complexity*, pages 74–88. Springer International Publishing, 2015.

[6] Ivan B. Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 416–427, New York, NY, 1990. Springer New York.

[7] Yvo Desmedt and Andrew M. Odlyzko. A chosen text attack on the RSA cryptosystem and some discrete logarithm schemes. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 516–522, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[8] Ding Z. Du and Frank K. Hwang. *Combinatorial Group Testing and Its Applications.* WORLD SCIENTIFIC, 2nd edition, 1999.

[9] Ding Z. Du and Frank K. Hwang. *Pooling Designs and Nonadaptive Group Testing.* WORLD SCIENTIFIC, 2006.

[10] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[11] Arkadii G. D'yachkov and Vladimir Vasil'evich Rykov. Bounds on the length of disjunctive codes. *Problemy Peredachi Informatsii*, 18(3):7–13, 1982.

[12] David Eppstein, Michael T. Goodrich, and Daniel S. Hirschberg. Improved combinatorial group testing algorithms for real-world problem sizes. *SIAM Journal on Computing*, 36(5):1360–1375, 2007.

[13] Paul Erdős, Peter Frankl, and Zoltán Füredi. Families of finite sets in which no set is covered by the union of r others. *Israel J. Math*, 51(1-2):79–89, 1985.

[14] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, Zhenfei Zhang, et al. FALCON: Fast-Fourier lattice-based compact signatures over NTRU. *Submission to the NIST's post-quantum cryptography standardization process*, 36(5):1–75, 2018.

[15] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. Cryptology ePrint Archive, Paper 2007/432, 2007.

[16] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[17] Thaís B. Idalino and Lucia Moura. Nested cover-free families for unbounded fault-tolerant aggregate signatures. *Theoretical Computer Science*, 854:116–130, 2021.

[18] Thaís B. Idalino and Lucia Moura. Structure-aware combinatorial group testing: A new method for pandemic screening. In Cristina Bazgan and Henning Fernau, editors, *Combinatorial Algorithms, Lecture Notes in Computer Science*, pages 143–156. Springer International Publishing, 2022.

[19] Thaís B. Idalino and Lucia Moura. A survey of cover-free families: Constructions, applications, and generalizations. In C.J. Colbourn and J.H. Dinitz, editors, *New Advances in Designs, Codes and Cryptography, NADCC 2022, Fields Institute Communications*, volume 86, pages 195–239. Springer, 2024.

[20] Thaís B. Idalino, Lucia Moura, and Carlisle Adams. Modification tolerant signature schemes: location and correction. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019, Lecture Notes in Computer Science*, pages 23–44. Springer International Publishing, 2019.

[21] Thaís B. Idalino, Lucia Moura, Ricardo F. Custódio, and Daniel Panario. Locating modifications in signed data for partial data integrity. *Information Processing Letters*, 115(10):731–737, 2015.

[22] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC Press, 3rd edition, 2023.

[23] William Kautz and Roy Singleton. Nonrandom binary superimposed codes. *IEEE Transactions on Information Theory*, 10(4):363–377, 1964.

[24] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: generation, enumeration, and search*. CRC Press, 1999.

[25] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, Microsoft, October 1979. This paper was published by IEEE in the Proceedings of HICSS-43 in January, 2010.

[26] P. C. Li, G. H. John. van Rees, and Ruizhong. Wei. Constructions of 2-cover-free families and related separating hash families. *Journal of Combinatorial Designs*, 14(6):423–440, 2006.

[27] Charles C. Lindner and Christopher A. Rodger. *Design theory*. Chapman and Hall/CRC, 2017.

[28] Dongxia Luo. Java implementation of d-MTSS. `https://github.com/micoluo/Modification-Digital-Signature-Scheme-Using-Combinatorial-Group-Testing`, October 2024.

[29] Dongxia Luo and Lucia Moura. Fast decoding of group testing results from Reed-Solomon d-disjunct matrices. In Svetla Nikova and Daniel Panario, editors, *International Workshop on the Arithmetic of Finite Fields*, volume 15176 of *Lecture Notes in Computer Science*. Springer International Publishing, 16 pages (to appear), 2024.

[30] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[31] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[32] Pavlos Nikolopoulos, Sundara R. Srinivasavaradhan, Tao Guo, Christina Fragouli, and Suhas Diggavi. Group testing for overlapping communities. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–7, 2021.

[33] Legion of the Bouncy Castle Inc. Bouncy Castle open-source cryptographic apis. `https://www.bouncycastle.org`, 2024. Accessed: 2024-08-27.

[34] Ely Porat and Amir Rothschild. Explicit nonadaptive combinatorial group testing schemes. *IEEE Transactions on Information Theory*, 57(12):7982–7989, 2011.

[35] Bart Preneel. Davies–Meyer hash function. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, Boston, MA, 2005.

[36] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[37] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[38] Emanuel Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27:544–548, 1928.

[39] Douglas R. Stinson. *Combinatorial Designs: Constructions and Analysis*. Springer, 1 edition, 2004.

[40] Douglas R. Stinson and Ruizhong Wei. Generalized cover-free families. *Discrete Mathematics*, 279(1):463–477, 2004.

[41] Dhanashree Toradmalle, Rohan Singh, Het Shastri, Nikita Naik, and Vishal Panchidi. Prominence of ECDSA over RSA digital signature algorithm. In *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC) I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC), 2018 2nd International Conference on*, pages 253–257. IEEE, 2018.

[42] Ruizhong Wei. On cover-free families. Technical report, Lakehead University, 2006. arxiv: `https://arxiv.org/abs/2303.17524`.

[43] Wikimedia Commons. UOttawa Tabaret Hall, 2008. [Online; accessed July 24, 2024].