Master Thesis
Computing Science

Radboud University

Optimizing the MEDS Implementation for ARMv8

Author:
Lars Jeurissen
s1022856
lars.jeurissen@ru.nl

First supervisor/assessor:
prof. dr. Peter Schwabe
peter@cryptojedi.org

Second supervisor:
dr. Simona Samardjiska
simonas@cs.ru.nl

August 23, 2024

Abstract

As the risk of quantum computers breaking current cryptographic schemes grows, the need for post-quantum cryptography becomes more pressing. The recent NIST competition on post-quantum signature schemes has led to the creation of MEDS, a signature scheme based on the Matrix Code Equivalence (MCE) problem. In this work, we optimize the existing MEDS implementation for the ARMv8 CPU architecture using parameter-specific optimizations and NEON SIMD instructions.

We explore two approaches: a low-level approach that optimizes the individual operations of the scheme and a high-level approach that parallelizes over the main commitment loop. The low-level approach gives the best results, with a speedup of 3.2× for key generation, 3.9× for signing, and 4.1× for verification, in NIST category 3, on the ARM Cortex-A72.

Additionally, we suggest an optimization to the hashing structure used in MEDS which, when combined with the low-level optimizations, increases the speedup for signing and verification to 4.1× and 4.4×, respectively, in NIST category 3, on the ARM Cortex-A72.

We also provide a brief analysis of the performance on the Apple M2, which shows that our optimizations are even more effective on this architecture.

Contents

Chapter 1

Introduction

1.1 Context

As the research on quantum computers progresses, we are getting increasingly closer to the point where quantum computers will be able to utilize algorithms such as Shor's algorithm [69] and Grover's algorithm [44] to break various cryptographic schemes, causing the absolute collapse of the present public key algorithms that are considered secure [53]. As the majority of the digital world relies on the security of these cryptographic schemes, this will have devastating consequences for the security of not only the internet, but also financial transactions, secure communication, and many other critical sectors.

The solution to this problem lies in the development of cryptographic schemes that are secure against quantum computers. Such algorithms have been around for a long time, but this area of research has experienced a boost in attention ever since the National Institute of Standards and Technology (NIST) started the post-quantum cryptography (PQC) standardization process in 2017 [60]. The goal of this process is to standardize cryptographic schemes that are secure against quantum computers.

In 2022, NIST announced the set of selected PQC algorithms, which included three digital signature schemes: CRYSTALS-Dilithium [36], Falcon [39], and SPHINCS$^+$ [21]. As two of these schemes are based on structured lattices, NIST announced a second competition in the PQC standardization process, which aims to find additional general-purpose signature schemes that are not based on structured lattices. One of the candidates in this competition is Matrix Equivalence Digital Signature (MEDS) [29]. MEDS is a code-based digital signature scheme based on the notion of Matrix Code Equivalence. In this thesis, we aim to optimize the performance of the MEDS implementation for the ARMv8 architecture.

1.2 Motivation

The need for digital signature schemes that are secure against quantum computers is increasing. Of course, the security of these schemes is the most important aspect, but the speed at which an implementation can create or verify a signature is also important. The speed of a digital signature scheme is essential for many applications, such as TLS/SSL certificate verification, electronic payments, and blockchain transactions.

Although the MEDS scheme is actively being optimized in terms of key and signature sizes, the performance of the scheme is still lacking. The reported signature verification times are in the order of hundreds of milliseconds and sometimes even seconds, depending on the security level and the chosen parameters. Traditional digital signature schemes such as RSA [64] and ECDSA [45] can verify thousands of signatures per second on both modern and even older hardware [34, 49]. The new post-quantum digital signature schemes must eventually achieve similar performance levels.

The two most widely used CPU architectures in the world are x86 (used in Intel and AMD processors) and ARM. ARM is used in a wide variety of devices, for example,

- mobile devices and tablets such as the Apple iPhone and iPad, Samsung Galaxy, and Google Pixel;
- embedded and Internet of Things (IoT) devices such as smart light bulbs, smart thermostats, and smart doorbells;
- Apple M1/M2/M3 chips used in Apple MacBooks.

Optimizing the MEDS implementation for the ARMv8 architecture will improve the MEDS performance for these devices, as well as provide valuable insights into the performance of MEDS on ARMv8.

1.3 Scope

The goal of this thesis is to optimize the performance of the MEDS implementation for the ARMv8 architecture to the extent that further optimizations are either infeasible or provide only minimal performance improvements. To achieve this goal, we investigate the following research questions:

RQ I. How is CPU time distributed across the code of the MEDS implementation?

RQ II. What optimizations can be made to improve the performance of the MEDS implementation for the ARMv8 architecture?

a) Which of these optimizations can be used in general for any architecture?

Since the submission of the original MEDS implementation to the NIST PQC competition in 2023 [28], the authors have proposed several optimizations to the scheme to reduce the size of the public key and the signatures [30]. The authors have provided a new reference implementation for MEDS that implements these optimizations. In this thesis, we focus on optimizing this new implementation for the ARMv8 architecture. There are two main reasons for this focus:

- Research has been done on the optimization of the original MEDS implementation for the x86 architecture (using AVX512 instructions) in [1, 2]. Although the ARMv8 architecture is different and an optimization of the original implementation for ARMv8 would be interesting, we believe that optimizing the new implementation provides better insights into the performance of a state-of-the-art MEDS implementation.
- The new MEDS variant has quite a few differences compared to the original, but the core structure and algorithms remain the same. This suggests that the optimizations that we make for the new implementation can also be applied to the original implementation, should the need arise.

In this thesis, when talking about the MEDS implementation, we refer to the new reference implementation that was provided by the authors in [30], unless stated otherwise.

1.4 Related work

Digital signature schemes and their optimization have been the subject of many research papers over the last few decades. Traditional widely-used digital signature schemes such as RSA [64], ECDSA [45], and EdDSA [20] have received optimizations for various versions of ARM CPUs over the years [74, 75, 81]. The introduction of NEON [6] (ARM's SIMD instruction set) in ARMv7 has led to a new wave of optimizations for traditional digital signature schemes based on NEON instructions, starting with the work of Bernstein and Schwabe in 2011 [22]. This work has been continued for both symmetric and asymmetric cryptographic algorithms over the years [11, 42, 67, 68, 78].

The introduction of the NIST post-quantum cryptography standardization process in 2016 has led to the development of many new public key encryption

(PKE), key encapsulation mechanism (KEM), and digital signature schemes that are believed to be secure against quantum computers. Research into the NEON optimization of PKE/KEM schemes soon followed [57, 58, 61, 70, 73]. The optimization of post-quantum digital signature schemes has mostly focused on finalists of the competition: CRYSTALS-Dilithium [36], Falcon [39], SPHINCS$^+$ [21], and Rainbow [35] have received NEON-based optimizations for ARMv8 CPUs [17, 18, 46, 47, 50, 59].

More relevant to this thesis is the work on the implementation and optimization of other (code-based) schemes that are similar to MEDS. Many schemes in the ongoing NIST PQC competition, including ALTEQ [26], LESS [12], PERK [3], and RYDE [5], have been optimized for x86 CPUs using AVX2/AVX512 [63] instructions. However, only a few schemes, such as MiRitH [4] and UOV [24], have been optimized for ARMv8.

Of particular interest is the work on the optimization of the original MEDS implementation for the x86 architecture using AVX512 instructions [1, 2]. Both this optimization and ALTEQ [26] provide the high-level optimization idea of intertwining multiple mathematical objects to allow for more parallelism in the signature generation and verification algorithms. Besides this, LESS [12] explores the low-level optimization of the computation of the RREF of a matrix, which is also used in MEDS.

Besides the use of NEON instructions, the use of already optimized algorithms can greatly improve the performance of a cryptographic scheme. Common cryptographic building blocks such as the KECCAK permutation [23] (which is used in MEDS) and Montgomery multiplication [54] have already been optimized for various ARM CPUs [18, 27, 66] and are used in many digital signature schemes.

1.5 Contributions

This thesis makes the following contributions:

- We provide a detailed profiling of the MEDS implementation for the ARMv8 architecture (specifically, the ARM Cortex-A72), which gives insights into the performance bottlenecks of the implementation. These profiling results, although specific to the ARMv8 architecture, can be used as a guideline for optimizing the MEDS implementation for other architectures.
- We provide two mutually exclusive optimizations for the MEDS implementation: a low-level optimization that focuses on the optimization of underlying functions such as matrix multiplication and the matrix systemizer, and a high-level optimization that focuses on the paral-

lelization of the commitment loop in the signature and verification algorithms.

- We provide novel assembly implementations of essential functions such as modular reduction and matrix multiplication that are optimized for the ARMv8 architecture. These implementations can be used in other cryptographic schemes that require similar functions.
- We propose a change to the MEDS scheme that allows for larger parallelizability in the hashing process. This change provides a speedup of both signature generation and verification.
- We present the benchmarking results of the optimized MEDS implementation on the ARM Cortex-A72 and the Apple M2 and compare these results to the reference implementation.
- We compare MEDS to similar digital signature schemes in terms of performance and show that the optimized MEDS implementation is competitive with these schemes.

1.6 Outline

In Chapter 2, we provide the necessary background information on the functioning of MEDS and the specific details of the ARMv8 architecture. In Chapter 3, we discuss the profiling techniques that we use to obtain an understanding of the performance of MEDS and present the profiling results of the MEDS implementation. Following that, in Chapter 4, we discuss the optimization techniques that we use to optimize the MEDS implementation. In Chapter 5, we present and discuss the benchmarking and profiling results of our optimizations. We conclude the thesis in Chapter 6, where we reflect on the results that we obtained. Finally, we discuss the remaining future work in Chapter 7.

All code used in this thesis is available on GitHub for reference under the GPL-3.0 license.[1]

[1] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis

Chapter 2

Preliminaries

In this chapter, we provide the relevant background concepts on the functioning of the MEDS scheme, the ARMv8 architecture, the optimization of cryptographic schemes and modular reduction. Additionally, we give a small overview of the notations that we use in this thesis.

2.1 Notations

In the MEDS scheme and this thesis, we use the following notations:

- $\mathbb{F}_q$: The finite field of size q.
- $\mathbb{F}_q^{m \times n}$: The set of matrices of size $m \times n$ (m rows and n columns) over $\mathbb{F}_q$, meaning that each element of the matrix is an element of $\mathbb{F}_q$.
- $\mathrm{GL}_k(q)$: The set of all invertible $k \times k$ matrices over $\mathbb{F}_q$.
- $\mathbf{A} \in \mathbb{F}_q^{m \times n}$: A matrix $\mathbf{A}$ of size $m \times n$ over $\mathbb{F}_q$.
- $\mathbf{I}_n$: The identity matrix of size $n \times n$.
- $\mathbf{A}[i][j]$: The element in the i-th row and j-th column of matrix $\mathbf{A}$.
- $\mathbf{A}^T$: The transpose of matrix $\mathbf{A}$.
- $\mathbf{A}^{-1}$: The inverse of matrix $\mathbf{A}$.
- $\mathbf{A} \times \mathbf{B}$ or simply $\mathbf{AB}$: The matrix product of $\mathbf{A}$ and $\mathbf{B}$.
- $\mathbf{A} + \mathbf{B}$: The element-wise matrix sum of $\mathbf{A}$ and $\mathbf{B}$.
- $\mathbf{A} \otimes \mathbf{B}$: The Kronecker product of $\mathbf{A}$ and $\mathbf{B}$.
- $\pi_{\mathbf{A},\mathbf{B}}(\mathbf{G})$: Simplified notation for the operation $\mathbf{G}(\mathbf{A}^T \otimes \mathbf{B})$.
- $(\mathbf{A} \mid \mathbf{B})$: The matrix formed by concatenating matrices $\mathbf{A}$ and $\mathbf{B}$.

2.2 Matrix Equivalence Digital Signature (MEDS)

Matrix Equivalence Digital Signature (MEDS) [29] is a code-based digital signature scheme and the candidate in the NIST PQC competition that we aim to optimize in this thesis. A series of optimizations for the key and signature sizes of MEDS have already been proposed by the authors of the scheme [30], together with a new reference implementation that implements these optimizations. In this paper, we focus our efforts on optimizing this new reference implementation.

2.2.1 Signature schemes

A digital signature scheme is a cryptographic scheme with the purpose of verifying the authenticity of a message. The scheme allows a party to sign a piece of data such as a message or a document, after which any party can verify the signature and thereby the authenticity of the data.

A digital signature scheme consists of three algorithms:

- **Key generation**: This algorithm generates 2 keys, a private key and a public key. The private key is used to sign the data, and the public key is used to verify the signature.
- **Signature generation**: Given a message and the private (and sometimes public) key, this algorithm generates a signature for the message.
- **Signature verification**: Given a message, a signature, and the public key, this algorithm verifies the signature over the message.

The formal definition of a digital signature scheme is given in Definition 2.2.1.

Definition 2.2.1 (Signature Scheme, following [40])**.**
A digital signature scheme is a tuple of algorithms $(\text{Keygen}, \text{Sign}, \text{Verify})$ where:

- $\text{Keygen}(1^n)$ generates a key pair (pk, sk), where pk is the public key and sk is the private key. The parameter n represents the security parameter which determines the security level of the scheme.
- $\text{Sign}(sk, m)$ generates a tag T which represents the signature over the message m using the private key sk.
- $\text{Verify}(pk, m, T)$ outputs 1 if the tag T is a valid signature over the message m using the public key pk, and 0 otherwise.

Such that, given $(pk, sk) \leftarrow \text{Keygen}(1^n)$:

$$\text{Verify}(pk, m, \text{Sign}(sk, m)) = 1$$

for all m (with a negligible probability of error).

2.2.2 Codes and Matrix Code Equivalence (MCE)

Most PQC schemes are based on mathematical concepts such as linear codes, isogenies, lattices and multivariate equations. These concepts have associated decisional and computational problems that are believed to be hard to solve for both classical and quantum computers. MEDS is based on the notion of Matrix Code Equivalence (MCE), which is closely related to the notion of Code Equivalence that is used in LESS [25], a similar scheme in the NIST PQC Signature competition. In this section, we provide background information on codes and the MCE problem. A more detailed explanation of matrix codes can be found in [41]. A study on the hardness of the MCE problem, as well as related problems, can be found in [62].

Codes

Each matrix has a so-called rank, of which the definition is given in Definition 2.2.2.

Definition 2.2.2 (Matrix Rank)**.**
The rank of a matrix $\mathbf{A} \in \mathbb{F}_q^{m \times n}$, denoted as $\text{rank}(\mathbf{A})$, is the maximum number of linearly independent rows (or columns) of $\mathbf{A}$.

The matrix rank can be used to create a distance metric between matrices. This metric is called the rank metric, and its definition is provided in Definition 2.2.3.

Definition 2.2.3 (Rank Metric, following [41])**.**
The rank metric between two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{F}_q^{m \times n}$ is defined as:

$$d(\mathbf{A}, \mathbf{B}) = \text{rank}(\mathbf{A} - \mathbf{B})$$

where $\mathbf{A} - \mathbf{B}$ is the element-wise difference of the matrices.

Using the rank metric, we can define a (matrix) rank metric code: the underlying mathematical object of MEDS. The definition of a rank metric code is given in Definition 2.2.4.

Definition 2.2.4 ((Matrix) Rank Metric Code, following [41])**.** A rank metric code (also called matrix code) $\mathcal{C}$ is a subspace of $\mathbb{F}_q^{m\times n}$, meaning it is a set of matrices of size $m \times n$ over $\mathbb{F}_q$ that are closed under addition and scalar multiplication.

A 'codeword' of a rank metric code is a matrix that is an element of the code. Codewords in a rank metric code can have different ranks.

A mapping, also called an isometry, can be applied to a code $\mathcal{C}$ to create a new code $\mathcal{D}$. An isometry is denoted by ϕ and is defined in Definition 2.2.5.

Definition 2.2.5 (Isometry, following [41])**.** An isometry ϕ is a bijective transformation on $\mathbb{F}_q^{m\times n}$, defined by a pair of matrices $(\mathbf{A}, \mathbf{B})$ where $\mathbf{A} \in \mathrm{GL}_m(q)$ and $\mathbf{B} \in \mathrm{GL}_n(q)$, having the following form:

$$\mathbf{M} \mapsto \mathbf{AMB}$$

ϕ can be applied to a code $\mathcal{C}$ to obtain a new code $\mathcal{D}$: $\phi(\mathcal{C}) = \mathcal{D}$. For every codeword $\mathbf{C}$ in $\mathcal{C}$, the matrix $\mathbf{D} = \mathbf{ACB}$ is in $\mathcal{D}$. All codewords in $\mathcal{D}$ can be obtained by applying ϕ to the codewords in $\mathcal{C}$.

Matrix Code Equivalence (MCE)

MEDS, just like many other (post-quantum) signature schemes, is based on an equivalence problem, which asks if there exists a mapping (isometry) between two mathematical objects that preserves certain properties. When given two objects, it is usually hard to find such a mapping, but it is easy to verify if a given mapping is correct. This is the underlying principle of the security of many cryptographic schemes.

In the case of MEDS, these mathematical objects are rank metric codes. Two rank metric codes $\mathcal{C}$ and $\mathcal{D}$ are called equivalent if there exists an isometry ϕ with $\phi(\mathcal{C}) = \mathcal{D}$. The isometry should preserve the rank of the matrices in the code. This means that after applying ϕ, the rank distance between any two matrices in $\mathcal{C}$ should be the same as the rank distance between the corresponding matrices in $\mathcal{D}$.

The computational form of the MCE problem is shown in Definition 2.2.6.

Definition 2.2.6 (Matrix Code Equivalence Problem, following [28])**.** Given two rank metric codes $\mathcal{C}, \mathcal{D} \in \mathbb{F}_q^{m\times n}$, find an isometry ϕ on $\mathbb{F}_q^{m\times n}$ such that $\mathcal{C} = \phi(\mathcal{D})$ and ϕ preserves the rank metric.

2.2.3 Sigma protocol and Fiat-Shamir transform

Sigma protocol

The MCE problem is used in MEDS to construct a 3-pass Sigma protocol [32]. A Sigma protocol (Σ-protocol) is a variant of a zero-knowledge proof, which is a protocol between a prover and a verifier where the prover convinces the verifier that it knows a piece of information without revealing the information itself. A slightly simplified definition of a Sigma protocol is given in Definition 2.2.7.

Definition 2.2.7 (Sigma Protocol, following [32])**.**
A Sigma protocol for a relation R over a pair (x, w) where x is a statement (an instance of some computational problem) and w is a witness (a solution to that instance) is a 3-pass protocol between a prover P and a verifier V consisting of the following steps:

- **Commitment**: The prover P generates a commitment a based on generated values of x and w and sends it to the verifier V.
- **Challenge**: The verifier V sends a random challenge e to the prover P.
- **Response**: The prover P sends a reply z based on the challenge e, the commitment a, and the information x and w. The verifier V can verify z based on a and e.

In the above definition, a prover P can cheat the verifier V with some probability by guessing the challenge e before sending the initial commitment.

In the case of MEDS, the prover convinces the verifier that it knows a certain isometry ϕ that satisfies an instance of the MCE problem, without revealing the isometry itself. In the definition of the Sigma protocol, x is an instance of the MCE problem (two rank metric codes $\mathcal{C}$ and $\mathcal{D}$), and w is an isometry ϕ (such that $\phi(\mathcal{C}) = \mathcal{D}$). The Sigma protocol for the optimized version of MEDS is provided in [30, Section 4.2].

Fiat-Shamir transform

To convert a Sigma protocol into a digital signature scheme, the Fiat-Shamir transform [38] is used. The initial Sigma protocol is interactive, meaning that the prover and verifier exchange messages, which is not suitable for a digital signature. The Fiat-Shamir transform converts the Sigma protocol such that the prover can show knowledge of the isometry while only sending a single message to the verifier, making it non-interactive. This is achieved by creating the challenge based on a collision-resistant hash of the message

to be signed and the commitment. The bits in the resulting digest are used as the challenge in the Sigma protocol.

When working with the Fiat-Shamir transform, various techniques and optimizations can be used to increase the security or lower the size of the public key and the signature. In the list below, we provide an overview of some of the techniques that are considered in the MEDS scheme. In addition to this list, Section 2.2.5 discusses more complex optimizations that are not listed here.

- **Multiple challenges**:
  Following the structure of a Sigma protocol used in MEDS, where a challenge is either 0 or 1, an attacker can impersonate an honest prover with $\frac{1}{2}$ probability. This can be prevented by extracting not one, but t challenges from the bits in the digest that was created by the hash of the message and the commitment. This reduces the probability of impersonation to $\frac{1}{2^t}$.
- **Multiple public keys**:
  As mentioned before, an attacker can impersonate an honest prover with $\frac{1}{2}$ probability. To reduce this probability, the scheme can use multiple public keys, each of which is used to compute a different isometry. This increases the challenge space from 2 to $s+1$, reducing the probability of impersonation to $\frac{1}{s+1}$ (s is the number of public keys used in the scheme).
- **Fixed-weight challenge strings**:
  A challenge is a number in the range $[0, s]$. If a challenge is 0, the response consists of matrices that are generated uniformly at random. In this case, it is sufficient to set the response to the seed that was used to generate the matrices, greatly reducing the size of the signature. By fixing a certain number w of challenges to 0, the average size of a response can be reduced. This technique has a slightly negative impact on the security of the scheme, but this can be compensated by increasing the number of challenges.
- **Seed tree**:
  If a scheme requires sending multiple seeds for the generation of matrices (or other objects), a seed tree can be used to reduce the size of the public key and the signature. This is a structure that allows the prover to transmit a smaller amount of bits than the size of the seeds, at the cost of an increased computational complexity.

By selecting t, s and w carefully and combining them with other parameters of the scheme, the security of the scheme can be increased to the desired level. Multiple combinations of parameters are used in MEDS to achieve various security levels [28]. The selection of these parameters has a big influence on

Table 2.1: Recommended MEDS parameter sets. **pk** and **sig** represent the size in bytes of the public key and the signature, respectively.

Parameter Set	q	n	m	k	s	t	w	**pk**	**sig**
MEDS-21595	4093	26	25	25	2	144	48	21595	5200
MEDS-55520	4093	35	34	34	2	208	75	55520	10906
MEDS-122000	4093	45	44	44	2	272	103	122000	19068

the size of the public key and the signature, as well as the computational performance of the scheme.

2.2.4 Parameter sets

The security of MEDS depends on the choice of a set of parameters. The parameters that are used in the MEDS scheme are the following:

- q: The size of the finite field $\mathbb{F}_q$ over which all computations are done.
- n: The width and height of the private matrices $A_i \in \mathbb{F}_q^{n \times n}$ that are used to generate the key pair.
- m: The width and height of the private matrices $B_i \in \mathbb{F}_q^{m \times m}$ that are used to generate the key pair.
- k: The width and height of the private matrices $T_i \in \mathbb{F}_q^{k \times k}$ that are used to generate the key pair.
- s: The number of different public keys that are used in the scheme.
- t: The number of challenges that are used in the Fiat-Shamir transform.
- w: The number of challenges in the Fiat-Shamir transform that are fixed to be 0.

The team behind MEDS has proposed three parameter sets for the new optimized version of the scheme [30]. These parameter sets are optimized for the three different security levels that are required in the NIST PQC competition. The parameter sets are shown in Table 2.1. The security level for each parameter set is shown in Table 2.2.

We can see that all parameter sets use the same finite field size $q = 4093$. The dimensions of the matrices that are used increase with each security level, as well as the number of challenges t (and the number of fixed challenges w). The number of public keys s is always set to 2, meaning the scheme does not use the multiple public keys technique. Note that this differs from the original MEDS scheme [28], which used multiple public keys.

Table 2.2: MEDS security levels. **FS** denotes the claimed security of a MEDS parameter set in bits, following the currently best-known attack of [55].

Parameter Set	**NIST Category**	**FS**
MEDS-21595	Level 1	128.406
MEDS-55520	Level 3	192.058
MEDS-122000	Level 5	256.005

2.2.5 MEDS algorithms

In this section, we give an algorithmic overview of the three algorithms of MEDS: key generation, signature generation, and signature verification. The complete and detailed pseudocode of these algorithms is shown in Appendix A. Additionally, we provide some information on the matrix systemizer and isometry mapping derivation, two common functions used in the MEDS algorithms.

Key generation

A simplified overview version of the MEDS key generation algorithm is shown in Algorithm 2.1. The full and detailed key generation algorithm for MEDS is shown in Algorithm A.1 (Appendix A).

Algorithm 2.1 MEDS key generation (overview)

Input: -
Output: public key **pk**, private key **sk**
Generate random matrix $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn}$
for $i \in \{1, \ldots, s-1\}$ **do**
    Generate random invertible matrix $\mathbf{T}_i \in \mathbb{F}_q^{k \times k}$
    Compute $\mathbf{G}'_0 \in \mathbb{F}_q^{k \times mn} = \mathbf{T}_i \times \mathbf{G}_0$
    Compute isometry $(\mathbf{A}_i \in \mathbb{F}_q^{m \times m}, \mathbf{B}_i \in \mathbb{F}_q^{n \times n})$ from codewords in $\mathbf{G}'_0$
    Compute $\mathbf{G}_i \in \mathbb{F}_q^{k \times mn} = \pi_{\mathbf{A}_i, \mathbf{B}_i}(\mathbf{G}_0)$
    Convert $\mathbf{G}_i$ to systematic form
return pk $= \mathbf{G}_0, \mathbf{G}_i$, sk $= \mathbf{G}_0, \mathbf{A}_i, \mathbf{B}_i, \mathbf{T}_i$ (for $i \in \{1, \ldots, s-1\}$)

Signature generation

A simplified overview version of the MEDS signature algorithm is shown in Algorithm 2.2. The full and detailed signature generation algorithm for MEDS is shown in Algorithm A.2 (Appendix A).

Algorithm 2.2 MEDS signature generation (overview)

Input: private key $\mathbf{sk}$, message m
Output: signature σ (contains the tag)
Parse $\mathbf{G}_0$ and $\mathbf{T}_i$ from $\mathbf{sk}$ for $i \in \{1, \ldots, s-1\}$
for $i \in \{0, \ldots, t-1\}$ **do**
    Generate random matrix $\tilde{\mathbf{M}}_i \in \mathbb{F}_q^{2\times k}$
    Compute $\mathbf{C} \in \mathbb{F}_q^{2\times mn} = \tilde{\mathbf{M}}_i \times \mathbf{G}_0$
    Compute isometry $(\mathbf{A} \in \mathbb{F}_q^{m\times m}, \mathbf{B} \in \mathbb{F}_q^{n\times n})$ from codewords in $\mathbf{C}$
    Compute $\tilde{\mathbf{G}}_i \in \mathbb{F}_q^{2\times mn} = \pi_{\mathbf{A},\mathbf{B}}(\mathbf{G}_0)$
    Convert $\tilde{\mathbf{G}}_i$ to systematic form
Hash m and $\tilde{\mathbf{G}}_i$ for $i \in \{0, \ldots, t-1\}$ to obtain d
Parse a set of hashes $h_0, \ldots, h_{t-1}$ from d
for $i \in \{0, \ldots, t-1\}$ **do**
    if $h_i > 0$ **then**
        Compute $\kappa_i \in \mathbb{F}_q^{2\times k} = \tilde{\mathbf{M}}_i \times \mathbf{T}_{h_i}^{-1}$
return Signature $\sigma = \kappa_0, \ldots, \kappa_{t-1}, h_0, \ldots, h_{t-1}, m$

Signature verification

A simplified overview version of the MEDS signature verification algorithm is shown in Algorithm 2.3. The full and detailed signature verification algorithm for MEDS is shown in Algorithm A.3 (Appendix A).

Matrix systemizer

The matrix systemizer is responsible for converting a matrix $A \in \mathbb{F}_{4093}^{m\times n}$ over the finite field $\mathbb{F}_{4093}$ into a systematic form. In MEDS, the systematic form $A' \in \mathbb{F}_{4093}^{m\times n}$ of a matrix $A \in \mathbb{F}_{4093}^{m\times n}$ can take two forms:

- **REF'**:
  A' is in row-echelon form (REF) [77, Section 3.2], with the additional property that the leading coefficient of row i is in column i.
- **RREF'**:
  A' is in reduced row-echelon form (RREF), with the additional property that the first $m \times m$ submatrix of A' is the identity matrix.

The algorithm used in MEDS to systemize a matrix is a Gaussian elimination algorithm with some extra properties. The algorithm is implemented such that it runs in constant time, meaning that the execution time is the same for all matrices of equal size. Additionally, the algorithm contains three optional features that can be used (based on the input arguments):

- $r_{\max}$: Set the number of rows of the input matrix that need to be systemized. If not specified, the entire matrix is systemized.

Algorithm 2.3 MEDS signature verification (overview)

Input: public key $\mathbf{pk}$, signature σ
Output: 1 if the signature is valid, 0 otherwise
Parse $\mathbf{G}_0$ and $\mathbf{G}_i$ from $\mathbf{pk}$ for $i \in \{1, \ldots, s-1\}$
Parse κ_i, h_i, d and m from σ for $i \in \{0, \ldots, t-1\}$
for $i \in \{0, \ldots, t-1\}$ **do**
    if $h_i > 0$ **then**
        Compute $\mathbf{G}'_0 = \kappa_i \times \mathbf{G}_{h_i}$
        Cmpt. isometry $(\mathbf{A} \in \mathbb{F}_q^{m \times m}, \mathbf{B} \in \mathbb{F}_q^{n \times n})$ from codewords in $\mathbf{G}'_0$
    else
        Re-generate matrix $\tilde{\mathbf{M}}_i \in \mathbb{F}_q^{2 \times k}$ as in Algorithm 2.2
        Compute $\mathbf{C} \in \mathbb{F}_q^{2 \times mn} = \tilde{\mathbf{M}}_i \times \mathbf{G}_0$
        Compute isometry $(\mathbf{A} \in \mathbb{F}_q^{m \times m}, \mathbf{B} \in \mathbb{F}_q^{n \times n})$ from codewords in $\mathbf{C}$
    Compute $\hat{\mathbf{G}}_i \in \mathbb{F}_q^{2 \times mn} = \pi_{\mathbf{A},\mathbf{B}}(\mathbf{G}_0)$
    Convert $\hat{\mathbf{G}}_i$ to systematic form
Hash m and $\hat{\mathbf{G}}_i$ for $i \in \{0, \ldots, t-1\}$ to obtain d'
return 1 if $d = d'$, 0 otherwise

- `do_swap`: Allow the algorithm to swap columns of the matrix to ensure that the leading coefficient of a row is not 0.
- `do_backsub`: Whether or not to perform back substitution after the matrix has been systemized into REF'/RREF' form.

The complete algorithm is depicted in Algorithm A.7 (Appendix A.3).

Deriving isometry mappings

For more information on the isometry derivation process, we refer the reader to [30, Section 4.2], where the authors provide a detailed explanation of the technique.

In key generation, signing, and verification, as a result of an optimization technique that is applied to reduce the signature size [30], MEDS requires the derivation of an isometry mapping $\phi = (\mathbf{A} \in \mathbb{F}_q^{m \times m}, \mathbf{B} \in \mathbb{F}_q^{n \times n})$. This isometry mapping maps two full-rank, linearly independendent codewords $\tilde{\mathbf{C}}_1, \tilde{\mathbf{C}}_2 \in \mathbb{F}_q^{1 \times mn}$ that are stored in a matrix $\mathbf{C} \in \mathbb{F}_q^{2 \times mn}$ to two matrices $\mathbf{D}_0, \mathbf{D}_1 \in \mathbb{F}_q^{m \times n}$. The deriving of this isometry is done by constructing a linear system of $2mn$ equations and $m^2 + n^2$ variables formed by:

$$\mathbf{A} \times \mathbf{C}_0 = \mathbf{D}_0 \times \mathbf{B}^{-1}$$
$$\mathbf{A} \times \mathbf{C}_1 = \mathbf{D}_1 \times \mathbf{B}^{-1}$$

The resulting system of linear equations can be solved using a simple Gaussian elimination algorithm. However, because of the large size of the linear system,

this would result in a complexity of $\mathcal{O}(2mn \cdot (m^2+n^2)^2) = \mathcal{O}(n^6)$ (as $n = m+1$ for the parameter sets that we consider) which is very inefficient.

Luckily, there is a better alternative. As the values of $\mathbf{D}_0$ and $\mathbf{D}_1$ can be public information or random matrices, we have the freedom to choose them. By setting $\mathbf{D}_0 = (\mathbf{I}_m \mid 0) \in \mathbb{F}_q^{m \times n}$ and $\mathbf{D}_1 = (0 \mid \mathbf{I}_n) \in \mathbb{F}_q^{m \times n}$, we obtain a much more sparse and structured linear system which can be solved with an algorithm that has a complexity of $\mathcal{O}(n^3)$.

This is precisely what the `solve_opt` function in the MEDS reference implementation is responsible for. Its input is a matrix $\mathbf{C} \in \mathbb{F}_q^{2 \times mn}$ that represents the two codewords. It then constructs the sparse linear system of equations mentioned above and solves it in $\mathcal{O}(n^3)$ time. The output of the function are the matrices $\mathbf{A}$ and $\mathbf{B}$ that represent the isometry mapping. The code for this function is rather long and is therefore omitted, but can be found in the code repository of this thesis.[1]

2.3 ARMv8, Cortex-A72, and NEON

In this thesis, we focus on optimizing the MEDS implementation for the ARMv8 CPU architecture [52]. ARMv8 supports a wide range of instruction sets and extensions, of which the following are relevant to this thesis:

- **A64**: The default 64-bit instruction set for ARMv8.
- **NEON** (Advanced SIMD): The NEON instruction set is an extension to the ARMv8 architecture that is mandatory in all ARMv8 implementations. It allows for SIMD operations on 128-bit registers (see Section 2.3.1).
- **Crypto**: The cryptographic extension to the ARMv8 architecture. This is a non-mandatory extension that supports hardware support for various cryptographic algorithms.

2.3.1 Vectorization and SIMD

Single Instruction, Multiple Data (SIMD) is a type of instruction that operates on multiple pieces of data (vectors) in parallel. Using this technique, it is possible to execute a single operation (such as an addition or multiplication) on a vector of multiple numbers in a timeframe that is similar to the conventional operation on a single number. This can greatly improve the performance of algorithms that lend themselves to vectorization.

[1] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/ref/src/util.c

Table 2.3: Possible ARMv8 NEON lane sizes and their assembly instruction suffixes.

Lane Count	Lane Width	ASM Suffix
2	64 bits	`.2d`
1	64 bits	`.1d`
4	32 bits	`.4s`
2	32 bits	`.2s`
8	16 bits	`.8h`
4	16 bits	`.4h`
16	8 bits	`.16b`
8	8 bits	`.8b`

In ARM, the SIMD instruction set is called NEON or Advanced SIMD (both terms refer to the same instruction set). NEON is a 128-bit SIMD architecture extension that is required in all standard ARMv8 implementations [6]. The NEON unit contains 32 128-bit registers, each of which can be used to store a vector of values. Each NEON register can be split into several different lanes. The lane sizes that are supported in NEON, together with their assembly (ASM) code suffixes, are shown in Table 2.3.

In assembly code, we can refer to a NEON register by using the register prefix `v0-v31` followed by a suffix that indicates the lane size. For example, `v0.4s` refers to a 128-bit NEON register that contains 4 lanes of 32 bits each and can therefore store 4 32-bit values.

The NEON instruction set contains a wide range of instructions that can be used to perform various operations on these vectors. Some illustrative examples are included in Algorithm 2.4. A graphical depiction of the instructions in the code snippet above is shown in Figure 2.1.

In the remainder of this thesis, we use such NEON instructions in two forms:

- **Assembly**: Directly writing assembly code that uses NEON instructions, such as the `add v2.4s, v0.4s, v1.4s` instruction that adds two 4-lane vectors of 32-bit values.
- **C intrinsics**: Using C intrinsics, which are functions that map directly to NEON instructions. ARM provides a set of intrinsics that can be used when including the `arm_neon.h` header file [9].[2] An example is the `vadd_u16(uint16x4_t a, uint16x4_t b)` intrinsic, which adds two 4-lane vectors of 16-bit unsigned integers.

[2] https://developer.arm.com/architectures/instruction-sets/intrinsics/#f:@navigationhierarchiessimdisa=[Neon]&f:@navigationhierarchiesarchitectures=[A64]

Algorithm 2.4 NEON instruction example (assembly)

```
// Load 4 32-bit unsigned integers stored at the memory address
    in x0 and x1 into v0.4s and v1.4s
ld1 {v0.4s}, [x0]
ld1 {v1.4s}, [x1]
// Add the values in v0.4s and v1.4s into v2.4s
add v2.4s, v0.4s, v1.4s
// Multiply v2.4s with itself into v3.2d and v4.2d
umull v3.2d, v2.2s, v2.2s   // Lower half
umull2 v4.2d, v2.4s, v2.4s  // Upper half
// Combine the upper halves of v3.2d and v4.2d into v5.4s
uzp2 v5.4s, v3.4s, v4.4s
// Store the result in v5.4s to the memory address in x2
st1 {v5.4s}, [x2]
```

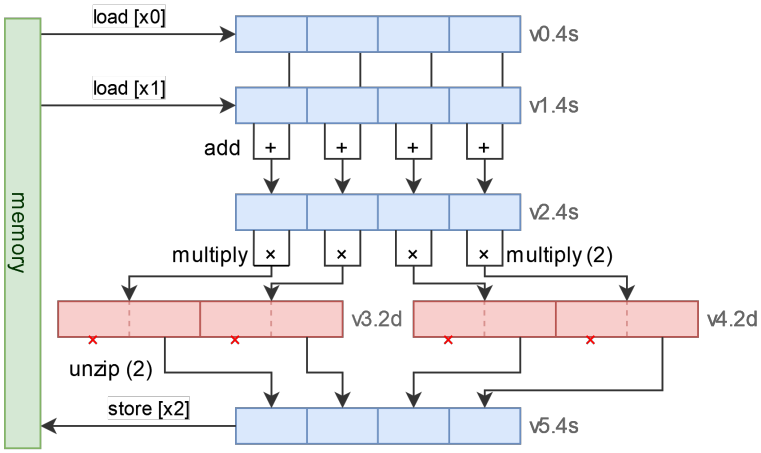

Figure 2.1: Visual representation of the NEON registers and instructions in Algorithm 2.4.

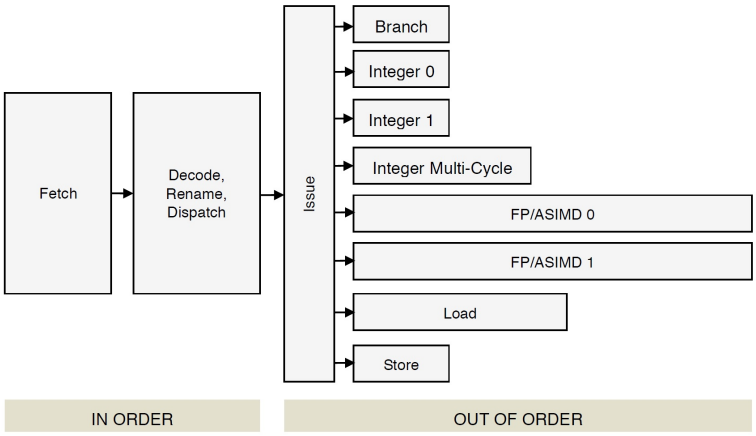

Figure 2.2: High-level overview of the ARM Cortex-A72 instruction processing pipeline [7].

2.3.2 Cortex-A72 Micro-Architecture

For profiling (and benchmarking) our code, we use a Raspberry Pi 4 Model B, which has a 64-bit quad-core ARM Cortex-A72 CPU that uses the ARMv8-A architecture (which is a specific profile of ARMv8). The term 'Cortex-A72 Micro-Architecture' refers to the specific design of the ARM Cortex-A72 CPU, which differs from other CPUs that implement the ARMv8-A architecture.

We work with the A64 instruction set, combined with the NEON extension. Unfortunately, the Cortex-A72 does not support the Crypto extension, which means that we cannot use hardware acceleration for cryptographic operations such as SHA-3 [37].

In this thesis, we focus on optimizing the MEDS implementation for the Cortex-A72 CPU. However, we also provide benchmarks for the Apple M2 CPU, which runs on the ARMv8.6-A architecture and supports the Crypto extension.

Core pipeline

The core pipeline of the Cortex-A72 consists of 15 stages which are split into a front-end and a back-end. A high-level overview of the instruction processing pipeline [7] is shown in Figure 2.2.

The pipeline consists of two parts:

1. **Front-end** (In Order): This part of the pipeline is responsible for fetching and decoding incoming instructions into micro-ops: smaller instructions that are easier to execute. These micro-ops are then

fed into the back end. The front end works on instructions in order, meaning that it processes them in the order that they are received.

2. **Back-end** (Out of Order): This part of the pipeline is responsible for executing the micro-ops that are generated by the front end. It consists of multiple pipelines that serve different purposes, such as branching, integer arithmetic, NEON operations, and memory operations. The back-end pipelines can execute micro-ops out of order, meaning they can execute operations in a different order than they were received.

The front-end can work on 3 instructions at the same time and it can dispatch up to 5 micro-ops per cycle to the back-end [65]. The back-end pipelines can execute one or two micro-ops per cycle, depending on the specific pipeline. The main goal of the front end (and this structure in general) is to make sure that the back-end pipelines are always filled with micro-ops to minimize stalls (where a pipeline has to wait for an earlier operation to finish).

Assembly instructions and latency

In this thesis, we use A64 assembly instructions to optimize the MEDS implementation. Each used instruction has three relevant properties:

- **Execution latency**: The number of CPU cycles after starting an instruction that it takes for the results of that instruction to be available for usage in the next instruction.
- **Execution throughput**: The maximum amount of times that a particular instruction can be executed per cycle.
- **Utilized pipeline**: The back-end pipeline(s) that this instruction uses.

As the Cortex-A72 uses an out-of-order execution model, it is impossible to provide instruction timing information that will predict how long a certain instruction will take to execute in a certain context [7]. Nevertheless, the execution latency and throughput of instructions provide valuable information about the performance of that instruction. Therefore, we list the execution latency and throughput of the NEON instructions that we use in the assembly code that was written for this thesis in Table 2.4. We also use many intrinsic functions in the C code, which are functions that map directly to assembly instructions. However, as the compiler is still able to reorder and optimize these instructions, it is not directly beneficial to know the latencies and throughputs of these instructions.

Table 2.4: ARM Cortex-A72 latency and throughput for the NEON assembly instructions used in this thesis [7].

Instruction	Description	Latency	Throughput
`ld1`	Load from memory	5	1
`st1`	Store to memory	5	1
`umull`	Unsigned multiply long	4	1
`umull2`	Unsigned multiply long (high half)	4	1
`umlal`	Uns. mult. long & accumulate	4	1
`mls`	Multiply subtract	5	1/2
`add`	Add	3	2
`sub`	Subtract	3	2
`and`	Bitwise AND	3	2
`ushr`	Unsigned shift right	3	1
`dup`	Store in all lanes	8	1
`ins`	Store in one lane	3	2
`xtn`	Extract and narrow	3	2
`uzp1`	Unzip (keep low)	3	2
`uzp2`	Unzip (keep high)	3	2
`cmeq`	Compare equal	3	2
`cmhs`	Compare higher or same	3	2

Instruction scheduling

At first glance, it might seem that the order in which instructions of a certain instruction sequence are written in the assembly/intrinsic code influences the performance of the code, as the latencies and throughputs of the various instructions can influence the execution time of the code. This is the case for earlier ARM architectures, where hand-tuning of the instruction order could lead to performance improvements. However, the technical reference manual of the Cortex-A72 states that the out-of-order pipeline of the CPU can schedule and execute the instructions of a certain instruction sequence in an optimal fashion without any instruction reordering required [8]. This means that the CPU can 'hide' the latencies of instructions (by executing other micro-ops in the meantime). Although it is debatable whether this is always the case and whether there are no instruction sequences that the CPU is not able to schedule optimally, it is a good indication that we can focus on writing clean and readable code without having to worry about the order of the instructions.

It is important to note that this does not mean that there are no benefits to knowing the latencies and throughputs of instructions. Choosing a different sequence of instructions to compute the same result can still have an impact

on the performance of the code because some instructions might be better suited for that specific computation.

2.4 Optimizing cryptographic schemes

When programming or optimizing cryptographic schemes or primitives, it is essential that the security of the scheme is not compromised because of vulnerabilities that are introduced in the code. One of the most important aspects in this context is the notion of constant-time execution [48]. A cryptographic function is said to be executed in constant-time if there exist no side-channel attacks that can extract secret information (such as a private key) by looking at the execution time of the function or any sub-sequence of the instructions in the function. In this section, we go over the main pitfalls that can lead to non-constant-time execution.

2.4.1 Branching and conditional execution

One of the most common reasons for non-constant-time execution occurs when the code contains branches that depend on secret data, allowing an attacker to perform a timing attack. A timing attack is a side-channel attack where an attacker measures the time that it takes to execute a certain piece of code and uses this measurement to derive information about the secret data that is used in the code. Consider the example in Algorithm 2.5.

Algorithm 2.5 Branching on secret data (unsafe)

```
1: if secret_data = 0 then
2:     doExpensiveOperation()          ▷ This operation is slow
3: else
4:     doCheapOperation()              ▷ This operation is fast
```

The execution time of this code snippet depends on the value of 'secret_data'. If 'secret_data' is 0, the execution time will be longer than when 'secret_data' is not 0, allowing an attacker to derive information about the value of 'secret_data' by measuring the execution time of the code.

Another example of branching on secret data occurs when the code uses a loop in which the execution time depends on secret data. Consider the example in Algorithm 2.6.

Algorithm 2.6 Looping on secret data (unsafe)

```
1: for i = 0 to secret_data do
2:     doSomething()
```

The execution time of this code snippet depends on the value of 'secret_data'.

Compiler optimizations and branch prediction

The previous examples are relatively obvious, but there are many more subtle ways in which branching on secret data can occur, especially when working with a smart compiler that tries to optimize the code. Consider the following example in Algorithm 2.7.

Algorithm 2.7 Conditional swap (unsafe)

```
1: function conditionalSwap(x, y, secret_data)
2:     if secret_data = 1 then
3:         x, y ← y, x
4:     else
5:         x, y ← x, y
6:     return x, y
```

Although it might seem that this function is safe, nearly all compilers will optimize this code such that the assignment in the 'else' branch is removed, making a timing attack possible.

Even if you can guarantee that the compiler does not optimize this code, the 'branch prediction' feature in modern CPUs might still make a timing attack possible. Branch prediction is a technique that tries to predict the outcome of a branch instruction before it is executed, allowing a CPU to start executing the instructions within the predicted branch. If the prediction is correct, the code will execute faster, but if the prediction is incorrect, the CPU will need to discard the results and start executing the correct branch, causing a delay. If an attacker knows how the CPU will predict the branch, they can use this information to perform a timing attack.

Mitigating timing attacks

The most important technique to mitigate timing attacks on conditional execution is to never branch on secret data at all, not even if it seems safe. Most common operations can be rewritten in a way that removes conditional execution. For example, the conditional swap operation in Algorithm 2.7 can be rewritten as shown in Algorithm 2.8.

Algorithm 2.8 Conditional swap (safe)

```
1: function conditionalSwap(x, y, secret_data)
2:     d ← (x ⊕ y) · secret_data          ▷ '⊕' is the XOR operation
3:     x ← x ⊕ d
4:     y ← y ⊕ d
```

In this code snippet, if 'secret_data' is 1, $x = x \oplus (x \oplus y) = y$. Otherwise, $x = x \oplus 0 = x$. This way, the code does not branch (on secret data) and is

therefore safe from timing attacks, assuming the compiler does not optimize the code in such a way that the branch is reintroduced.

2.4.2 Data-dependent memory access

Another common cause for non-constant-time execution is caused by accessing memory at an index that depends on secret data. Consider the example in Algorithm 2.9.

Algorithm 2.9 Data-dependent memory access (unsafe)

1: $x \leftarrow$ public_array[secret_data]

This code is susceptible to a 'cache-timing attack'. A cache-timing attack is a side-channel attack where an attacker measures the time that it takes to access a certain memory location [19, 48]. If the data at that location is stored in the CPU cache, the access time will be faster than if the data is not (yet) in the cache. By measuring the time that it takes to access a certain memory location, an attacker can infer information about the value of 'secret_data'.

This is only the most basic form of a cache-timing attack. More advanced attacks can be performed if the attacker has more control over the system, such as the ability to manipulate the cache to ensure that the data is (or is not) in the cache. Although there exist techniques to mitigate these attacks (such as always accessing every index of the array), it is best to avoid accessing memory at an index that depends on secret data altogether.

2.4.3 Non-constant-time operations

A final common cause for non-constant-time execution is the use of CPU instructions that do not execute in constant time. Common examples of such instructions are (integer) division, math functions such as sin/cos, and various other instructions, depending on the CPU architecture. Instructions like these can take a variable amount of time to execute, depending on the input data. It is important to avoid using such instructions with secret data.

2.4.4 Preventing timing attacks with Valgrind and Timecop

It has been shown that even if you write code that should theoretically compile to a constant-time binary, the compiler might still optimize the code in such a way that it is vulnerable to various kinds of timing attacks [16, 71]. Therefore, it is good practice to make sure the resulting binary is indeed constant-time.

Vulnerabilities to timing attacks described in Section 2.4.1 and Section 2.4.2 can be detected using Valgrind [56] using a technique that was described

by Langley [51]. Valgrind is a programming tool that is used to detect memory leaks, buffer overflows, and other memory-related problems. Among many other things, Valgrind can detect branching on uninitialized data and memory accessing at an uninitialized index. Valgrind is unable to detect timing attack vulnerabilities caused by non-constant-time operations such as mentioned in Section 2.4.3. Using the following routine, we can use Valgrind to detect timing attack vulnerabilities:

1. **Uninitialized secret data**: Make sure our code does not initialize any secret data. Normal operations (such as arithmetic operations) on uninitialized data are ignored by Valgrind.
2. **Run Valgrind**: Run Valgrind on our code. Any resulting complaints about uninitialized data are possible locations at which a vulnerability to a timing attack might exist.

Timecop [76] provides a `poison.h` header file that can be used to poison secret data, which essentially makes it uninitialized. This can be used in combination with Valgrind to detect timing attack vulnerabilities.

2.5 Modular reduction

As MEDS is a cryptographic scheme that operates on elements in a finite field, modular reduction is a basic building block that is used in many parts of the scheme. Various algorithms can be used to perform modular reduction. In this section, we go over the most common algorithms.

- **Naive reduction**
  The naive reduction algorithm is slow and not suitable for our purposes, but we use it as a baseline to compare the other algorithms to. This algorithm works by applying an integer division instruction to the input, followed by a multiplication with the modulus and a subtraction. A big disadvantage of this approach is that division is not a constant time operation, which can lead to timing attacks (see Section 2.4.3). Furthermore, the division operation is usually relatively slow compared to other operations.
- **Montgomery reduction**
  Montgomery reduction [54] is a modular reduction algorithm that is based on the Montgomery multiplication algorithm. It does not use a division instruction, which makes it suitable for constant time operations. Instead, it works by subtracting a multiple of the modulus from the input such that the input is (almost) smaller than the modulus. The algorithm requires the input and outputs to be converted to and from a Montgomery representation.

- **Barrett reduction**
  Barrett reduction [15] is a modular reduction algorithm that is similar to Montgomery reduction in the sense that it also subtracts a multiple of the modulus from the input. However, it does not require the input to be converted to a different representation. Additionally, the cost of the reduction is slightly lower for small inputs compared to Montgomery reduction.

Montgomery reduction is generally very efficient when a large chain of multiplications is required and the overhead of converting to and from Montgomery representation is negligible compared to the number of multiplications/reductions, whereas Barrett reduction is typically a bit faster for small inputs.

2.5.1 Barrett reduction

Barrett reduction works by approximating the modular reduction. Given an unsigned integer a and a modulus n,

$$a \bmod n = a - \left\lfloor \frac{a}{n} \right\rfloor \cdot n.$$

From this, the Barrett reduction formula for unsigned integers can be derived (Definition 2.5.1).

Definition 2.5.1 (Barrett reduction for unsigned integers)**.**
Let a be an unsigned integer and n be a modulus. Let R be a constant such that $R = 2^k > n$ for some k. Then:

$$a \bmod n = a - \left\lfloor \frac{a \cdot \left\lceil \frac{R}{n} \right\rfloor}{R} \right\rfloor \cdot n.$$

where $\left\lceil \frac{R}{n} \right\rfloor$ represents $\frac{R}{n}$ rounded up or down, depending on the choice of the rounding operation.

As R is a power of 2, the division by R can be implemented as a cheap right-shift operation. Additionally, the value of $m = \left\lceil \frac{R}{n} \right\rfloor$ can be precomputed because the modulus n is a constant. This leaves us with two things to choose: the value of k and the rounding operation.

Choice of k

Usually, the value of k chosen for Barrett reduction is as small as possible such that $2^k > n$. Combined with setting the rounding function of the precomputed value to the 'floor' function, this results in a reduction that reduces to a value between 0 and $2n - 1$, requiring an additional conditional

subtraction at the end of the reduction. Choosing a larger value of k has one major disadvantage: the precomputed value $m = \left\lceil \frac{R}{n} \right\rceil$ becomes larger, which means that the multiplication $a \cdot m$ will result in a larger value (possibly leading to a register overflow). However, choosing a larger value of k can also be beneficial: the quotient m will be closer to the actual value of $\frac{R}{n}$, which means that the reduction will be more accurate. If we know that the input a will never be larger than a particular value, we can choose k such that the conditional subtraction at the end of the reduction is not required. For this to work, we need to use the ceiling function for the rounding operation.

Chapter 3

Profiling

In this chapter, we discuss and execute the profiling of the optimized MEDS implementation from [30], with the goal of identifying the code sections that take up the most time. We use this information to optimize the code in the next chapter.

3.1 Profiling techniques

To obtain a better understanding of the performance of (specific functions of) MEDS, we need to profile the implementation. Profiling is the process of measuring the space or time complexity of a program or a specific function. The goal of profiling is to identify the bottlenecks in the speed or memory usage of a program: these are the parts of the program that take the most time or memory. Usually, we hope that a small part of the program is responsible for a large part of the time or memory usage and that this part can be optimized to improve the overall performance.

Typically, profiling is done by running the program with a profiler. There exist a wide variety of profilers for C, such as GProf [43], Valgrind [56], and Linux-Perf [33]. In our case, the most accurate way to measure the performance is to measure the number of cycles that are used by the program or a specific function, which can be done with Linux-Perf.

3.1.1 Cycle counting

Cycle counting is a technique that is used to measure the number of CPU cycles that it takes to execute a certain program or function. This is usually done by accessing the performance monitoring unit (PMU) of the CPU, which is a CPU component that measures the performance of the processor. Typically, these PMUs contain a register that can be read to obtain the

number of cycles executed since a certain point in time. On Linux, we can access this data using the Linux-Perf tool [33], which can be used to measure the number of instructions executed, the number of cache misses, etc.

Advantages

The advantages of profiling the code using a cycle counter are:

- **Accurate**: Measuring the cycle counter results in the most accurate measurement of time. For comparison, using a technique that measures the current time in (nano)seconds is less accurate, because the CPU is capable of executing multiple cycles in a single nanosecond.
- **Low overhead**: Measuring the cycle counter has a very low additional performance overhead to the program, as it usually consists of reading a single register.
- **Precise**: By annotating the code with our own cycle counter, we can measure the performance of specific functions or even specific lines of code.

Disadvantages

The disadvantages of profiling the code using a cycle counter are:

- **Interference**: The program to be measured shares the CPU (core) with other programs, which can interfere with measurements. If another program is switched in by the operating system, the cycle counter will also count the cycles that were used by this program. Although this usually does not have a big impact on profiling results, this can be problematic for obtaining accurate benchmarks.
- **Architecture**: The way in which the cycle counter is accessed is different for each architecture. However, since we use the Linux-Perf tool, this is abstracted away for us.

Problem mitigation

There are a few problems with cycle counting that we need to mitigate. First of all, there are some features on modern CPUs that will cause cycle counting to produce inaccurate results. These features include frequency scaling (based on the current workload, a CPU can change its clock frequency to save power) and hyperthreading (multiple threads share the same CPU core at the same time by interleaving instructions). It is essential to disable these features to obtain accurate results.

Table 3.1: MEDS-55520 key generation profiling results for the ARM Cortex-A72.

Function	**# MCycles** (±)	**% of Total** (±)	**# Calls**
`pmod_mat_mul`	15.98	69.78	70
`pmod_mat_syst`	2.08	9.07	6
`rnd_sys_mat`	2.07	9.05	1
`solve_opt`	1.42	6.20	1
`bs_fill`	1.06	4.63	1
Cumulative	22.61	98.73	
Remaining	0.29	1.27	

Another problem is program interference (see above). Unfortunately, this is a problem that is hard to prevent completely, but we can work around it. By running the program multiple times and taking the median of the results, we can reduce the impact of interference on the results. This is a common technique in benchmarking and is used in many other cryptographic papers [17, 36].

3.2 MEDS profiling results

3.2.1 Measurement setup

We added cycle count measurements to all MEDS functions that we anticipate will require a significant amount of time. We decided to profile the code for the MEDS-55520 parameter set, which is in the middle of the three parameter sets in terms of security level. The results are representative of the other parameter sets as well, only the scale is different. We executed measurements for the three algorithms of a digital signature scheme: key generation, signing, and verification, on the ARM Cortex-A72. The results are shown in Tables 3.1 (key generation), 3.2 (signing), and 3.3 (verification). For each algorithm, we list the functions or code sections that take up more than 1% of the total number of cycles. We provide the number of megacycles (MCycles) that were used in that function, the percentage of the total number of cycles that were used by that function, and the number of times that function was called. Note that the number of cycles cannot be divided by the number of calls to obtain the average number of cycles per call, because the number of cycles that are used by a function can depend on the input.

3.2.2 Result analysis

Because of the way that MEDS works, the results of the signing and verification operations are very similar. For key generation, the results in the table

Table 3.2: MEDS-55520 signing profiling results for the ARM Cortex-A72.

Function	**# MCycles** (±)	**% of Total** (±)	**# Calls**
`pmod_mat_mul`	2518.85	69.17	14635
`pmod_mat_syst`	391.36	10.75	1040
`solve_opt`	293.44	8.06	208
`bs_fill`	212.74	5.84	1
`shake256_absorb`	203.06	5.58	212
Cumulative	3619.46	99.39	
Remaining	22.24	0.61	

Table 3.3: MEDS-55520 verification profiling results for the ARM Cortex-A72.

Function	**# MCycles** (±)	**% of Total** (±)	**# Calls**
`pmod_mat_mul`	2537.20	69.24	14560
`pmod_mat_syst`	393.22	10.73	1040
`solve_opt`	294.20	8.03	208
`bs_fill`	212.88	5.81	1
`shake256_absorb`	203.31	5.55	267
Cumulative	3640.80	99.35	
Remaining	23.80	0.65	

account for 98.76% of the total number of cycles. For signing and verification, these numbers are 99.42% and 99.39%, respectively. In all three algorithms, the remainder of the cycles is spent on a large set of functions that take up a small amount of time. Given that there are only a few functions that take up a significant amount of time, we can conclude that the performance of MEDS is mostly determined by these functions.

Matrix multiplication

For all three operations, the `pmod_mat_mul` function takes up the most time, almost 70%. This function is used to multiply two matrices $A \in \mathbb{F}_q^{m \times n}$ and $B \in \mathbb{F}_q^{n \times o}$ over a finite field $\mathbb{F}_q$. The function is implemented in MEDS using a naive algorithm that computes the dot product of each row of A with each column of B, followed by a reduction modulo q. As will be shown in Section 4.2.1, the time complexity of this algorithm is $\mathcal{O}(mno)$ ($= \mathcal{O}(n^3)$ for square matrices).

Matrix systemizer

The `pmod_mat_syst` function (`pmod_mat_syst_ct_partial_swap_backsub` in the code, but shortened for readability) is responsible for about 11% of the total number of cycles for signing and verification. This function is used to systemize a matrix $A \in \mathbb{F}_q^{m \times n}$ over a finite field $\mathbb{F}_q$ (see Section 2.2.5). This is done using a Gaussian elimination algorithm that has a complexity of $\mathcal{O}(m^2 n)$ (this is calculated in Section 4.2.2).

Isometry derivation

The `solve_opt` function is responsible for about 8% of the total number of cycles for signing and verification. This function derives an isometry mapping $\phi = (\mathbf{A} \in \mathbb{F}_q^{m \times m}, \mathbf{B} \in \mathbb{F}_q^{n \times n})$ by constructing and solving a sparse system of linear equations, see Section 2.2.5. The algorithm has a complexity of $\mathcal{O}(n^3)$.

Bitstream filling

The `bs_fill` section is responsible for about 6% of the total number of cycles for signing and verification. In this section of the code, multiple calls are made to the `bs_init`, `bs_write`, and `bs_finalize`. We decided to group them because they are all part of the same operation: filling a bitstream with elements of the finite field $\mathbb{F}_q$. In the MEDS parameter sets that we consider, the finite field is $\mathbb{F}_{4093}$, which means that the bitstream is filled with 12-bit elements. This is done to reduce the number of bytes required to store a list of field elements.

SHAKE256

A small percentage of the total number of cycles in each of the three operations is used by either `shake256_squeeze` or `shake256_absorb`. SHAKE256 is an extendable output function (XOF). It is part of the SHA-3 family [37] and is based on the KECCAK sponge construction [23]. MEDS uses SHAKE256 to generate random field elements and to hash the challenge strings that are used in the Fiat-Shamir transform (see Section 2.2.3).

Random systemized matrix generation

The `rnd_sys_mat` function is responsible for about 9% of the total number of cycles for key generation. This function is used to generate a random systemized matrix over a finite field $\mathbb{F}_q$. Nearly all of its cycles are spent on generating random field elements using the `shake256_squeeze` function (see above).

Chapter 4

Methodology

We have established two approaches to optimize the MEDS implementation. These approaches are not specific to ARMv8, but their implementation will be tailored to the ARMv8 architecture. Note that the two approaches are mutually exclusive, meaning they cannot be combined. We implement both approaches and evaluate their performance to determine which approach gives the highest speedup.

- **Low-level optimization**:
  This approach focuses on optimizing the MEDS implementation at a low level. This means that we look at individual functions (such as matrix multiplication and the matrix systemizer) that take up a significant amount of time and optimize them. The input and output of these functions are thus not changed, but the way in which the result is computed is optimized using techniques such as vectorization.
- **High-level optimization**:
  This approach focuses on optimizing the MEDS implementation at a high level. The Fiat-Shamir transform used in MEDS (see Section 2.2.3) uses t challenges, where t is a parameter set according to the target security level. The values of t for each parameter set are displayed in Table 2.1 (Section 2.2.4). As can be seen from the algorithmic overviews in Section 2.2.5, the computation of each challenge (and commitment for that challenge) is done in the same way. This means that we can compute multiple commitments in parallel, which can greatly increase the performance of the scheme. As opposed to low-level optimization, this approach changes the input and output of the underlying functions to take in and return the inputs and outputs of multiple commitments at once.

In this chapter, we start by discussing how we implement modular reduction, an operation that is the same for both approaches, in Section 4.1. We then discuss the implementation of the two approaches in Sections 4.2 and 4.3. After this, we discuss bitstream filling and the usage of an alternative hash structure (two optimizations that can be applied to both the low-level and high-level approaches) in Section 4.4 and Section 4.5, respectively. Finally, we discuss a few non-constant-time optimizations that can be applied to the verification algorithm in Section 4.6.

4.1 Modular reduction

Throughout the entirety of the MEDS implementation, modular reduction is used extensively to reduce the size of the elements modulo q, where q is the modulus of the finite field $\mathbb{F}_q$. The finite field that is used in MEDS for the parameter sets that we consider is $\mathbb{F}_{4093}$, which means that all elements are reduced modulo 4093. In the reference implementation of MEDS, all modulo operations are done using the `%` operator in C. As we use NEON assembly instructions and C intrinsics to optimize the MEDS implementation, we cannot rely on this operator, as it is not defined for NEON registers and there is no single-instruction alternative in assembly.

Because of this, we need to implement our own modular reduction function that can be used with NEON registers. This function must be as fast as possible, as it is used in many places in the MEDS implementation. The various modular reduction algorithms are considered in Section 2.5. Our NEON assembly implementation of both the Barrett and Montgomery reduction algorithms is shown in Algorithm 4.1. As can be seen from the implementation, both algorithms use the same number of instructions. We therefore choose to use Barrett reduction, as it does not require the input and output to be converted to a different representation. We further elaborate on the Barrett reduction algorithm in the remainder of this section. As we will not use the Montgomery reduction algorithm, we only add it for the sake of comparison and do not further explain how it works.

4.1.1 Choosing k for Barrett reduction

As mentioned in Section 2.5.1, the choice of k for Barrett reduction is important. If possible, we should pick a k such that the reduction will not require a final conditional subtraction. Although it is possible to continue working with small values that are congruent to the original value modulo q and thereby avoid the conditional subtraction, this will eventually still require a few extra instructions to make sure the final value is reduced modulo q. If we can find a k such that the reduction will not require a conditional

Algorithm 4.1 NEON Barrett and Montgomery reduction for MEDS

Input: $a_i \in [0, 2^{29.5})$ for $0 \leq i < 4$ (in `v0.4s`)
`MEDS_p` $= 4093$ (in `v2.4s`)
Output: a_i mod 4093 for $0 \leq i < 4$ (in `v0.4s`)

Barrett
$m =$ 0x80180481 (in `v1.4s`)

```
umull  v3.2d, v0.2s, v1.2s
umull2 v4.2d, v0.4s, v1.4s
uzp2   v3.4s, v3.4s, v4.4s
ushr   v3.4s, v3.4s, 11
mls    v0.4s, v3.4s, v2.4s
xtn    v0.4h, v0.4s
```

Montgomery
$N' = 2731$ (in `v1.4s`)
$R_{\text{mask}} =$ `0xFFF` (in `v3.4s`)

```
mul  v1.4s, v0.4s, v1.4s
and  v1.16b, v1.16b, v3.16b
xtn  v1.4h, v1.4s
umlal v0.4s, v1.4h, v2.4h
ushr v0.4s, v0.4s, #12
xtn  v0.4h, v0.4s
```

subtraction whilst also not requiring additional instructions, we should pick this value.

For the MEDS parameter sets that we consider, we know that all field elements fit into 12 bits. This means that any multiplication of two field elements will result in a value that fits into 24 bits. The largest possible value that any value can become (before reduction) results from the matrix multiplication algorithm, see Section 4.2.1. In this algorithm, the temporary value can (for parameter set MEDS-122000) become as large as $\log_2(k \cdot (q-1) \cdot (q-1)) = \log_2(44 \cdot 4092 \cdot 4092) \approx 29.5$ bits (see Section 4.2.1). This means that we should find a k such that for all $0 \leq a < 2^{30}$, the reduction will not require a conditional subtraction.

As 2^{30} is a relatively small number, we use a brute force approach to find the smallest k that satisfies this condition. This value turns out to be $k = 43$, which gives $m = \lceil \frac{2^{43}}{4093} \rceil =$ 0x80180481. This number fits nicely into 32 bits, which means that we can store it in 32-bit NEON lanes. Because of this, using $k = 43$ does not require any additional instructions compared to using a smaller value of k.

4.1.2 Implementation

The reference code for our Barrett reduction algorithm is shown in Algorithm 4.2. This algorithm does not operate on vectorized data and serves as a baseline for implementing the NEON version.

The NEON version of the Barrett reduction algorithm is shown in Algorithm 4.3 (C code) and Algorithm 4.4 (assembly code). The C code uses NEON intrinsics to perform the reduction on 128-bit NEON registers. The assembly code is a direct translation of the C code to ARMv8 assembly. Both algorithms assume that the input consists of four 32-bit unsigned integers

Algorithm 4.2 MEDS Barrett reduction

1: **function** REDUCE(a)
2: $m \leftarrow$ 0x80180481
3: $v \leftarrow a \cdot m$
4: $v \leftarrow v \gg 43$
5: **return** $a - v \cdot 4093$

that are stored in four lanes of a 128-bit NEON register. The output can either be stored in four 32-bit lanes or four 16-bit lanes, depending on the requirements of the calling function.

Algorithm 4.3 MEDS NEON Barrett reduction (C)

```c
uint16x4_t reduce(uint32x4_t a)
{
  // Compute a*m for the lower and higher registers
  uint64x2_t low = vmull_u32(vget_low_u32(a), vget_low_u32(
    MAGIC_VEC));
  uint64x2_t high = vmull_high_u32(a, MAGIC_VEC);
  // Combine low and high parts. Gets rid of least significant
    32 bits of each element, effectively executing a right
    shift by 32
  uint32x4_t zip = vuzp2q_u32((uint32x4_t)low, (uint32x4_t)high
    );
  // Right shift by 11 (remaining part)
  uint32x4_t val = vshrq_n_u32(zip, 11);
  // Multiply by MEDS_p and subtract from a
  uint32x4_t result = vmlsq_u32(a, val, MEDS_p_VEC_32x4);
  // (Optional) shrink to uint16x4_t
  return vmovn_u32(result);
}
```

From the assembly code, we can see that the reduction is done using five instructions (not counting the optional shrink).

4.1.3 Freeze operation

A useful operation when working with numbers outside of the finite field is the 'freeze' operation. This operation takes a number $a \in [0, 2n-1]$ and returns $a \bmod n$, useful after the addition of two field elements. The advantage over the reduction functions mentioned before is that this operation can be executed with fewer instructions. Additionally, the operation can be executed over eight lanes of a 128-bit NEON register (as opposed to four lanes for the reduction operation). The freeze operation is useful after (for example) two field elements are added together and need to be reduced, in which case we can do a cheaper reduction operation. The freeze operation for MEDS is shown in Algorithm 4.5.

Algorithm 4.4 MEDS NEON Barrett reduction (assembly)

Input: $a_i \in [0, 2^{29.5})$ for $0 \leq i < 4$ (in `v0.4s`)
$m = 0$x80180481 (in `v1.4s`)
`MEDS_p` $= 4093$ (in `v2.4s`)
Output: $a_i \bmod$ `MEDS_p` for $0 \leq i < 4$ (in `v0.4s`)

```
// Compute a*m for the lower and higher registers
umull v3.2d, v0.2s, v1.2s
umull2 v4.2d, v0.4s, v1.4s
// Combine low and high parts. Gets rid of least significant 32
    bits of each element, effectively executing a right shift
   by 32
uzp2 v3.4s, v3.4s, v4.4s
// Right shift by 11 (remaining part)
ushr v3.4s, v3.4s, 11
// Multiply by MEDS_p and subtract from a
mls v0.4s, v3.4s, v2.4s
// (Optional) shrink to 16-bit lane
xtn v0.4h, v0.4s
```

Algorithm 4.5 MEDS Freeze Operation

function FREEZE(a)
    if $a \geq 4093$ **then**
        $a = a - 4093$
    return a

Algorithm 4.6 NEON freeze operation (C)

```
uint16x8_t freeze(uint16x8_t a)
{
    // Create mask of 1^* for all lanes where a >= MEDS_p
    uint16x8_t mask = vcgeq_u16(a, MEDS_p_VEC_16x8);
    // Create vector of MEDS_p for all lanes with a >= MEDS_p
    uint16x8_t val = vandq_u16(mask, MEDS_p_VEC_16x8);
    // Subtract MEDS_p from all lanes where a >= MEDS_p
    return vsubq_u16(a, val);
}
```

Algorithm 4.7 NEON freeze operation (assembly)

Input: $a_i \in [0, 2 \cdot 4093 - 1]$ for $0 \leq i < 4$ (in `v0.4s`)
`MEDS_p` $= 4093$ (in `v1.4s`)
Output: $a_i \bmod$ `MEDS_p` for $0 \leq i < 4$ (in `v0.4s`)

```
cmhs v2.8h, v0.8h, v1.8h
and v2.16b, v2.16b, v1.16b
sub v0.8h, v0.8h, v2.8h
```

As explained in Section 2.4.1, we cannot use branching in constant-time code. However, using NEON intrinsics (or assembly instructions), we can implement the freeze operation in constant time. The NEON version of the freeze operation is shown in Algorithm 4.6 (C code) and Algorithm 4.7 (assembly code).

4.2 Low-level optimization

The low-level optimization approach focuses on optimizing the individual functions that take up a significant amount of time in the MEDS implementation. As can be seen from the profiling results displayed in Section 3.2, over 99% of the total number of cycles used in signing and verification is used on just five functions. In this section, we discuss the optimization of three of these functions: matrix multiplication (Section 4.2.1), matrix systemizer (Section 4.2.2), and isometry derivation (Section 4.2.3). The optimization of the other two functions is not specific to the low-level optimization approach and will be discussed in Section 4.4 (bitstream filling) and Section 4.5 (SHAKE256).

4.2.1 Matrix multiplication

Matrix multiplication is by far the most time-consuming function in MEDS, taking up almost 70% of the total number of cycles in key generation, signing, and verification. The function is responsible for the simple task of multiplying two matrices $A \in \mathbb{F}_{4093}^{m \times n}$ and $B \in \mathbb{F}_{4093}^{n \times o}$ over the finite field $\mathbb{F}_{4093}$.

Complexity analysis

Matrix multiplication is implemented using a naive algorithm (see Algorithm A.4 in Appendix A.3) that works by calculating the dot product of each row of A with each column of B, followed by a reduction modulo 4093. Three nested loops loop over the m rows of A, the o columns of B, and the n columns of A and rows of B. The resulting time complexity of this algorithm is therefore $\mathcal{O}(mno)$, and the amount of work that needs to be done can be expressed as mno. Algorithms have been developed that can multiply two matrices in a lower time complexity, such as the Strassen algorithm [72] and (generalizations of) the Coppersmith-Winograd algorithm [31]. However, these algorithms have such a high constant factor that they are not suitable for our purposes. Instead, we focus on optimizing the naive algorithm.

Optimization

We optimize matrix multiplication by applying vectorization to the naive algorithm. We compute the dot product of multiple rows of A with multiple columns of B at the same time.

Our first observation lies in the fact that all intermediate values that are computed while calculating the dot product fit into 32 bits.

- Given $A \in \mathbb{F}_q^{m \times n} \times B \in \mathbb{F}_q^{n \times o} = C \in \mathbb{F}_q^{m \times o}$, the maximum value of n for all matrix multiplications in all MEDS parameter sets is 44 (MEDS-122000).
- The values stored in A and B are field elements that fit into 12 bits. Their value is 4092 at most.
- In the computation of a single element of C, exactly n multiplications are performed, which are all added together.
- This means that n values are added together, which results in a value that fits in $\log_2(n \cdot 4092 \cdot 4092) = \log_2(44 \cdot 4092 \cdot 4092) \approx 29.5$ bits.

As a result, we do not need to reduce the intermediate values modulo 4093 during the computation of the dot product if we accumulate the multiplications of the 16-bit values into 32-bit registers. Instead, we can reduce the resulting value after the dot product is computed. From this observation, we can derive that we can compute the dot product of four rows of A with four columns of B at the same time, as we can fit four 32-bit values in four lanes of a 128-bit NEON register.

However, we can do slightly better than that. The NEON unit is capable of performing four (widening) multiplications of 16-bit unsigned integers at the same time, but a NEON register can hold eight 16-bit unsigned integers. This means that we can compute the dot product of eight rows of A with eight columns of B using two multiplication instructions. This does not change the total number of multiplications that need to be performed, but it does halve the number of memory loads and stores that need to be executed.

Unfortunately, we cannot directly apply an easy 8-way vectorization to (one of) the loops in the naive algorithm. The NEON unit can load 128 bits of data from memory at a time, but it is not able to do this with an arbitrary distance between the elements. The matrices in MEDS are stored in row-major order (each row is stored after the previous row), meaning we can load eight elements of a row using a single NEON instruction, but we cannot load eight elements of a column using a single NEON instruction. An easy solution would be to transpose matrix B before performing the multiplication, but this would require additional memory and time to perform the transpose.

Instead, we use an approach where 16 dot products are computed at the same time, in the form of an 8×8 submatrix of C. We load an 8×8 submatrix of A and an 8×8 submatrix of B into NEON registers and compute the dot product of these submatrices, adding the result to the submatrix of C. We repeat this process until all dot products are computed and stored in C. The resulting algorithm is shown in Algorithm 4.8. A visualization of the functionality of this algorithm is shown in Figure 4.1. This algorithm works for matrices of which the dimensions are multiples of eight.

Algorithm 4.8 Vectorized matrix multiplication for matrices that are multiples of eight in size

```
1:  function MATRIX_MUL(A ∈ F_q^{m×n}, B ∈ F_q^{n×o})
2:      C ← zero matrix of size m × o
3:      for c ← 0 to m in steps of 8 do
4:          for r ← 0 to o in steps of 8 do
5:              C_0, ..., C_7 ← empty NEON register
6:              for k ← 0 to n in steps of 8 do
7:                  A_0, ..., A_7 ← load 8 × 8 submatrix at A[r][k]
8:                  B_0, ..., B_7 ← load 8 × 8 submatrix at B[k][c]
9:                  for i ← 0 to 8 do
10:                     C_i ← C_i + B_0 × A_i[0]
11:                     C_i ← C_i + B_1 × A_i[1]
12:                     C_i ← C_i + B_2 × A_i[2]
13:                     C_i ← C_i + B_3 × A_i[3]
14:                     C_i ← C_i + B_4 × A_i[4]
15:                     C_i ← C_i + B_5 × A_i[5]
16:                     C_i ← C_i + B_6 × A_i[6]
17:                     C_i ← C_i + B_7 × A_i[7]
18:             for i ← 0 to 8 do
19:                 C_i ← reduce(C_i)
20:             store 8 × 8 submatrix C_0, ..., C_7 at C[r][c]
21:     return C
```

Handling non-multiples of eight

The algorithm in Algorithm 4.8 only works for matrices of which the dimensions are multiples of eight. Unfortunately, MEDS uses matrices of which the dimensions are usually not multiples of eight. To solve this problem, there are various approaches that we can take. An easy solution is to pad the matrices such that the dimensions become multiples of eight, but this would require additional memory and, if performed dynamically, additional computation time.

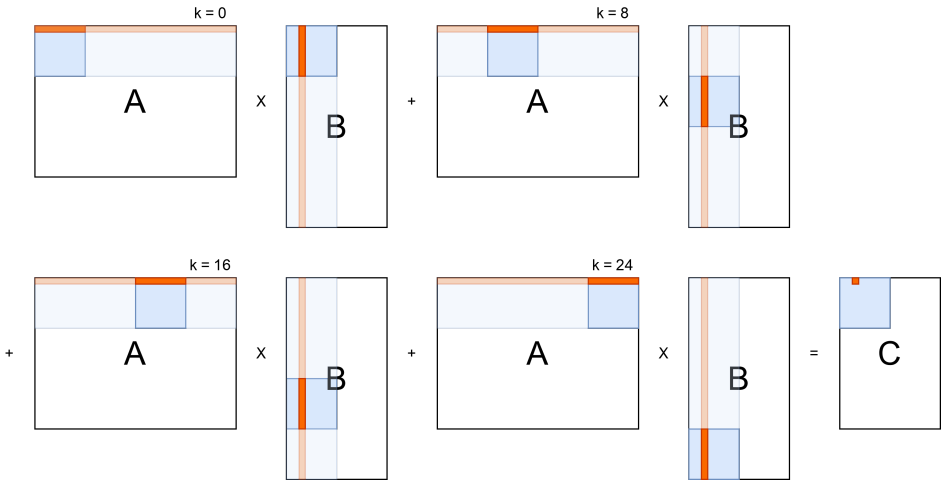

Figure 4.1: Visualization of the vectorized matrix multiplication algorithm. With one run of the k-loop, we compute one 8×8 submatrix of C (marked in blue). The orange cells depict the calculation of a single element of C.

Our solution to this problem is threefold, as three dimensions can be non-multiples of eight: m, o, and n. We handle each of these dimensions separately.

- **Dimension** m:
  When m is not a multiple of eight, we can use the normal approach until we reach the last multiple of eight. After this, when loading the submatrix of A, we load an $(m \bmod 8) \times 8$ submatrix. Additionally, we only compute the dot products and store the results for the $m \bmod 8$ rows of C that are required.

- **Dimension** o:
  When o is not a multiple of eight, we can use the normal approach until we reach the last multiple of eight. After this, when loading the submatrix of B, we load an $8 \times (o \bmod 8)$ submatrix. The remaining lanes in the NEON registers containing B_i are set to 0, making sure no incorrect values are added to C. When storing the results, we store only the first $o \bmod 8$ columns of C.

- **Dimension** n:
  When n is not a multiple of eight, we can use the normal approach until we reach the last multiple of eight. After this, we load only an $8 \times (n \bmod 8)$ submatrix of A and an $(n \bmod 8) \times 8$ submatrix of B. When computing C_i, we skip the last $8 - (n \bmod 8)$ multiplications and additions.

It is also possible for multiple of these dimensions to be non-multiples of eight at the same time. In this case, we can combine the approaches described above. In a few cases where the remaining number of rows or columns is less than eight but at least four, we can use 4-way vectorized instructions for a very small extra speedup.

Implementation

The actual implementation of Algorithm 4.8 combined with the handling of non-multiples can be done in multiple ways. As the resulting algorithm becomes quite complex, we figured that the C compiler might not be able to optimize the code as well as we can. Therefore, we have decided to implement a Python script[1] that generates optimized ARMv8 assembly code. The script generates a specialized assembly function for each set of input dimensions that is used in MEDS.

Minimum cycle bound

To understand the performance of our optimized matrix multiplication algorithm, we need to find a minimum bound on the number of cycles that the algorithm will take to execute. To do this, we establish a minimum bound on the number of instructions that the algorithm uses.

A very crude minimum cycle bound on the naive algorithm (see Algorithm A.4) can be established by counting the number of arithmetic operations required:

- $m \cdot o \cdot n$ multiplications and additions;
- $m \cdot o$ modulo operations.

The 'multiply-accumulate' operation is a single instruction on ARMv8, and modular reduction can be done in five instructions (see Section 2.5.1). This means that we can establish a minimum bound of $m \cdot o \cdot n + m \cdot o \cdot 5$ on the number of arithmetic instructions required for the naive algorithm. Any algorithm that follows this structure will require at least this number of instructions to execute, divided by the parallelization factor of that algorithm. In the case of the vectorized algorithm, this factor is eight.

[1] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/opt-low-level/src/asm/matmul/generate_matmul_asm.py

A more accurate minimum cycle bound can be established by looking at the number of instructions required for the vectorized algorithm (see Algorithm 4.8). This algorithm obtains a speedup because of two factors:

- The NEON unit can perform operations in parallel.
- As we compute the result for an 8×8 submatrix of C, we can use the same input value eight times, reducing the number of loads required.

The vectorized algorithm loads and later re-loads the same values multiple times. It is hard to account for this in a minimum cycle bound, as various implementations of the algorithm might require more or less loads. Therefore, we establish a minimum cycle bound based on the idea that we have infinite registers and need to load and store all required values only once. The number of instructions required for the vectorized algorithm can then be calculated as follows:

- $\frac{1}{8}m \cdot \frac{1}{8}n \cdot 8$ loads of A;
- $\frac{1}{8}o \cdot \frac{1}{8}n \cdot 8$ loads of B;
- $\frac{1}{8}m \cdot \frac{1}{8}o \cdot \frac{1}{8}n \cdot 8 \cdot 8 \cdot 2$ multiply-accumulate operations;
- $\frac{1}{8}m \cdot \frac{1}{8}o \cdot 8 \cdot 2$ reductions;
- $\frac{1}{8}m \cdot \frac{1}{8}o \cdot 8$ stores of C.

Given that reduction takes five cycles and the other operations take one cycle, we establish the following minimum cycle bound:

$$\frac{1}{8}mn + \frac{1}{8}on + \frac{1}{4}mon + \frac{1}{4}mo \cdot 5 + \frac{1}{8}mo = \frac{1}{4}mno + \frac{1}{8}mn + \frac{11}{8}mo + \frac{1}{8}no$$

Theoretical speedup

We compare the minimum cycle bound of the naive algorithm to the minimum cycle bound of the vectorized algorithm. We calculate the difference factor between the two bounds, which is the factor by which the vectorized algorithm should theoretically be faster than the naive algorithm. Using a similar approach as in the previous section, we establish the bound for the naive algorithm:

$$mo + no + mn + 2mno + mo \cdot 5$$

The theoretical speedup between the two algorithms is then given by:

$$\frac{2mno + mn + 6mo + no}{\frac{1}{4}mno + \frac{1}{8}mn + \frac{11}{8}mo + \frac{1}{8}no}$$

The theoretical and actual speedup factors are compared in Section 5.1.

4.2.2 Matrix systemizer

The matrix systemizer function is the second most time-consuming function in MEDS, taking up about 9% of the total number of cycles for key generation and 11% of the total number of cycles for signing and verification. The function is responsible for the task of systemizing a matrix $A \in \mathbb{F}_{4093}^{m \times n}$ over the finite field $\mathbb{F}_{4093}$ into a systemized matrix of REF' or RREF' form (see Section 2.2.5).

Complexity analysis

We analyze the complexity of the matrix systemizer algorithm used in the reference implementation (see Algorithm A.7). We first consider the basic algorithm where $r_{\max} = m$ and `do_swap` and `do_backsub` are set to false. We can then compute the complexity of the algorithm as follows:

$$
\begin{aligned}
&\sum_{r=0}^{m-1} \left(\sum_{r_2=r+1}^{m-1} \sum_{c=r}^{n-1} \mathcal{O}(1) + \mathcal{O}(1) + \sum_{c=r}^{n-1} \mathcal{O}(1) + \sum_{r_2=r+1}^{m-1} \sum_{c=r}^{n-1} \mathcal{O}(1) \right) \\
&= 2 \cdot \left(\sum_{r=0}^{m-1} \sum_{r_2=r+1}^{m-1} \sum_{c=r}^{n-1} \mathcal{O}(1) \right) + \left(\sum_{r=0}^{m-1} \sum_{c=r}^{n-1} \mathcal{O}(1) \right) + \left(\sum_{r=0}^{m-1} \mathcal{O}(1) \right) \\
&= 2 \cdot \left(\sum_{r=0}^{m-1} \mathcal{O}((m-r-1)(n-r)) \right) + \left(\sum_{r=0}^{m-1} \mathcal{O}(n-r) \right) + \mathcal{O}(m) \\
&= 2 \cdot \left(\sum_{r=0}^{m-1} \mathcal{O}(mn - mr - nr + r^2 - n + r) \right) + \left(\sum_{r=0}^{m-1} \mathcal{O}(n-r) \right) + \mathcal{O}(m) \\
&= \mathcal{O}\left(mn \sum_{r=0}^{m-1} 1 - m \sum_{r=0}^{m-1} r - n \sum_{r=0}^{m-1} r + \sum_{r=0}^{m-1} r^2 - n \sum_{r=0}^{m-1} 1 + \sum_{r=0}^{m-1} r + n \sum_{r=0}^{m-1} 1 - \sum_{r=0}^{m-1} r + m \right) \\
&= \mathcal{O}\left(mn \cdot m - m \cdot \frac{(m-1)m}{2} - n \cdot \frac{(m-1)m}{2} + \frac{(m-1)m(2m-1)}{6} + m \right) \\
&= \mathcal{O}\left(m^2 n - \frac{m^3 - m^2}{2} - \frac{m^2 n - mn}{2} + \frac{2m^3 - 3m^2 + m}{6} + m \right) \\
&= \mathcal{O}\left(m^2 n - \frac{m^3}{2} + \frac{m^2}{2} - \frac{m^2 n}{2} + \frac{mn}{2} + \frac{2m^3}{6} - \frac{3m^2}{6} + \frac{m}{6} + m \right) \\
&= \mathcal{O}(m^2 n)
\end{aligned}
$$

We can use arithmetic series to replace sums with closed-form expressions. In the final step, as $n \geq m$ and therefore $m^2 n \geq m^3$, we find that the complexity of the basic algorithm is $\mathcal{O}(m^2 n)$.

We consider the changes that the three optional features of the algorithm bring to the complexity:

- Using $r_{\max}$ will change the dimensions of the outer loop of the algorithm to $r_{\max}$ instead of m. This will change the complexity of the algorithm

to $\mathcal{O}(r_{\max} mn)$. As $r_{\max}$ is only ever m or $m-1$ in MEDS, this has no drastic effect on the complexity of the algorithm.

- When `do_swap` is set to true, an additional loop is added to the algorithm which adds the following complexity:

$$\left(\sum_{r=0}^{m}\sum_{r_2=r+1}^{m}\mathcal{O}(1)\right)+\left(\sum_{r=0}^{m}\sum_{i=0}^{r}\mathcal{O}(1)\right)$$

  Following a similar approach as above, we find that this results in a complexity of $\mathcal{O}(m^2)$, which does not change the complexity of the algorithm.

- When `do_backsub` is set to true, we perform another triple-nested loop to perform back substitution. This loop has the following complexity:

$$\left(\sum_{r=0}^{m}\sum_{r_2=0}^{r}\mathcal{O}(1)\right)+\left(\sum_{r=0}^{m}\sum_{r_2=0}^{r}\sum_{c=m}^{n}\mathcal{O}(1)\right)$$

  Again, following a similar approach as above, we find that this results in a complexity of $\mathcal{O}(m^2 + m^2 n - m^2) = \mathcal{O}(m^2 n)$, which does not change the complexity of the algorithm.

From this, we can conclude that the complexity of the matrix systemizer algorithm is $\mathcal{O}(m^2 n)$.

Optimization

We optimize the matrix systemizer algorithm by applying vectorization to the loops in the algorithm. Contrary to the matrix multiplication algorithm, we cannot easily parallelize the main r-loop in the algorithm, as the starting values of subloops depend on the (changing) value of r. This structure makes parallelization over the r-loop impossible. Instead, we optimize each inner loop separately by applying a parallelization technique tailored to the specific inner loop.

For all nested loops, we apply parallelization to the innermost loop. In the systemizer algorithm, this loop is always the easiest to parallelize, as it accesses matrix elements that are stored sequentially in memory. Each input matrix element is stored in a 12-bit field element, which fits into a 16-bit register. This means that we can compute 8 elements at the same time using 128-bit NEON registers.

Unfortunately, we are unable to use a full 8-way vectorization for all loops, as the algorithm uses multiplication operations. When multiplying two 16-bit values, the result is a 32-bit value, of which we cannot store eight in a 128-bit NEON register. We can work around this issue by using the `umull`

instruction to multiply the lower four 16-bit values of two registers and the `umull2` instruction to multiply the upper four 16-bit values of two registers. After reducing both results, we can use the `uzp1` operation to combine the results back into a single 128-bit register.

Handling non-multiples of eight

The inner loops that we are optimizing usually do not loop over a dimension that is a multiple of eight. To handle this, we use the following approach:

1. While at least eight elements are left to process, use the 8-way vectorization approach;
2. After this, if there are at least four elements left to process, use one iteration of a 4-way vectorized approach;
3. After this, use a non-vectorized approach for any remaining elements (maximum 3).

Implementation

As with the matrix multiplication algorithm, the resulting systemizer algorithm becomes quite complex. Although it can be implemented with NEON intrinsics, we opted for the faster approach of generating optimized ARMv8 assembly code using a Python script.[2] The script generates a specialized assembly function for each set of input dimensions that is used in MEDS.

Minimum cycle bound

Finding a minimum cycle bound for the optimized systemizer algorithm is very similar to determining its complexity. As we can follow the same structure as in Section 4.2.2, we omit some of the steps taken and refer the reader to that section for a more detailed explanation. As with matrix multiplication, we assume infinite registers and that all values are loaded and stored only once.

For the basic algorithm, the number of instructions required is as follows:

- $r_{\max} \cdot n \cdot \frac{1}{8}$ loads of A;
- $\sum_{r=0}^{r\_\max-1} \sum_{r_2=r+1}^{m-1} \sum_{c=r}^{n-1} \frac{1}{8}$ add, bitwise AND, and freeze instructions (first inner loop);
- $\sum_{r=0}^{r\_\max-1} 1$ finite field inversions;

[2] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/opt-low-level/src/asm/systemizer/generate_systemizer_asm.py

- $\sum_{r=0}^{r\_max-1} \sum_{c=r}^{n-1} (\frac{1}{8} \cdot 2)$ multiply and reduce instructions (normalize loop);
- $\sum_{r=0}^{r\_max-1} \sum_{c=r}^{n-1} \frac{1}{8}$ combine instructions (normalize loop);
- $\sum_{r=0}^{r\_max-1} \sum_{r_2=r+1}^{m-1} \sum_{c=r}^{n-1} (\frac{1}{8} \cdot 2)$ multiply and reduce instructions (last inner loop);
- $\sum_{r=0}^{r\_max-1} \sum_{r_2=r+1}^{m-1} \sum_{c=r}^{n-1} \frac{1}{8}$ combine, add, subtract and freeze instructions (last inner loop);
- $r_{\max} \cdot n \cdot \frac{1}{8}$ stores of A.

Given that reduction takes five cycles, freezing takes three cycles, field inversion takes 115 cycles, and all other instructions take one cycle, combining these results using the same techniques as for the complexity analysis gives us the following minimum cycle bound for the basic algorithm:

$$\begin{aligned}
&\frac{1}{4} \cdot r_{\max} \cdot n + 115 \cdot r_{\max} + \frac{13}{8} \left(n \cdot r_{\max} - \frac{(r_{\max} - 1) r_{\max}}{2} \right) \\
&+ \frac{23}{8} \left(mn \cdot r_{\max} - m \cdot \frac{(r_{\max} - 1) r_{\max}}{2} - n \cdot \frac{(r_{\max} - 1) r_{\max}}{2} \right. \\
&\left. + \frac{(r_{\max} - 1) r_{\max} (2 r_{\max} - 1)}{6} - n \cdot r_{\max} + \frac{(r_{\max} - 1) r_{\max}}{2} \right)
\end{aligned}$$

When using the algorithm with `do_swap` set to true, the following instructions are added to the minimum cycle bound:

- $\sum_{r=0}^{r\_max-1} \sum_{r_2=r}^{m-1} \frac{1}{8}$ bitwise OR instructions (swap loop 1).

Following the same approach as above, using `do_swap` adds the following value to the minimum cycle bound:

$$\frac{1}{8} \left(m \cdot r_{\max} - \frac{(r_{\max} - 1) r_{\max}}{2} \right)$$

When using the algorithm with `do_backsub` set to true, the following instructions are added to the minimum cycle bound:

- $\sum_{r=0}^{r\_max-1} \sum_{r_2=0}^{r-1} (\frac{1}{8} \cdot 2)$ multiply and reduce instructions (backsub middle loop);
- $\sum_{r=0}^{r\_max-1} \sum_{r_2=0}^{r-1} \frac{1}{8}$ combine, add, subtract, and freeze instructions (backsub middle loop);
- $\sum_{r=0}^{r\_max-1} \sum_{r_2=0}^{r-1} \sum_{c=r_{\max}}^{n-1} (\frac{1}{8} \cdot 2)$ multiply and reduce instructions (backsub inner loop);
- $\sum_{r=0}^{r\_max-1} \sum_{r_2=0}^{r-1} \sum_{c=r_{\max}}^{n-1} \frac{1}{8}$ combine, add, subtract, and freeze instructions (backsub inner loop).

Following the same approach as above, using `do_backsub` adds the following value to the minimum cycle bound:

$$\frac{18}{8}\left(\frac{(r_{\max}-1)r_{\max}}{2}\right)+\frac{18}{8}(n-r_{\max})\left(\frac{(r_{\max}-1)r_{\max}}{2}\right)$$

Theoretical speedup

Similarly to the matrix multiplication algorithm, we establish a theoretical speedup factor. Using a similar approach as in the previous section, we establish the bound for the non-optimized systemizer algorithm with `do_swap` and `do_backsub` set to false:

$$\begin{aligned}&2\cdot r_{\max}\cdot n+115\cdot r_{\max}+6\cdot\left(n\cdot r_{\max}-\frac{(r_{\max}-1)r_{\max}}{2}\right)\\&+11\cdot\left(mn\cdot r_{\max}-m\cdot\frac{(r_{\max}-1)r_{\max}}{2}-n\cdot\frac{(r_{\max}-1)r_{\max}}{2}\right.\\&\left.+\frac{(r_{\max}-1)r_{\max}(2r_{\max}-1)}{6}-n\cdot r_{\max}+\frac{(r_{\max}-1)r_{\max}}{2}\right)\end{aligned}$$

When using the algorithm with `do_swap` set to true, the bound is increased by:

$$1\cdot\left(m\cdot r_{\max}-\frac{(r_{\max}-1)r_{\max}}{2}\right)$$

When using the algorithm with `do_backsub` set to true, the bound is increased by:

$$11\cdot\left(\frac{(r_{\max}-1)r_{\max}}{2}\right)+11\cdot(n-r_{\max})\left(\frac{(r_{\max}-1)r_{\max}}{2}\right)$$

As these bounds are very complex, we do not provide a closed-form expression for the theoretical speedup factor. This factor is obtained by dividing the bound for the non-optimized algorithm by the bound for the optimized algorithm (possibly with the additional instructions for `do_swap` and `do_backsub`). The theoretical and actual speedup factors are compared in Section 5.1.

4.2.3 Isometry derivation

In all three operations of MEDS, the `solve_opt` function is used to derive an isometry mapping $\phi = (\mathbf{A} \in \mathbb{F}_q^{m\times m}, \mathbf{B} \in \mathbb{F}_q^{n\times n})$ by constructing and solving a sparse system of linear equations, see Section 2.2.5. The system that needs to be solved is constructed in a very specific way, which allows for a more efficient method of solving it (as opposed to using a more general algorithm like Gaussian elimination). Even though this more efficient method

is used, the function still takes up a significant amount of time in the MEDS implementation.

Unfortunately, the function is also extremely large, with the reference code containing over 300 lines of code, making it very tedious to optimize the function as a whole. Instead, we profiled the function and found that the majority of the time of this function (±70 %) is spent on two relatively small triple-nested loops, while the remainder of the time (±30 %) is spent on all other parts of the function combined. Therefore, we focus on optimizing these two loops. Both loops have the same structure, so we only discuss the optimization of the first loop. This loop is shown in Algorithm 4.9.

Algorithm 4.9 Isometry derivation: time-consuming loop 1

```
for (int b = MEDS_m - 3; b >= 0; b--)
  for (int c = MEDS_m - 1; c >= 0; c--)
    for (int r = 0; r < MEDS_m; r++)
    {
      uint64_t tmp1 = pmod_mat_entry(N, MEDS_n-1,MEDS_m, r, c);
      uint64_t tmp2 = sol[(MEDS_m+1)*MEDS_n+b*MEDS_m+MEDS_m+c];
      uint64_t prod = (tmp1 * tmp2) % MEDS_p;
      uint64_t val = sol[(MEDS_m+1) * MEDS_n + b * MEDS_m + r];
      val = ((MEDS_p + val) - prod) % MEDS_p;
      sol[(MEDS_m + 1) * MEDS_n + b * MEDS_m + r] = val;
    }
```

Complexity analysis

As mentioned in Section 2.2.5, the complexity of the entire `solve_opt` function is $\mathcal{O}(n^3)$. Analyzing the complexity of the two time-consuming loops specifically is trivial as they are both triple-nested loops that start and end at specific values. As can be derived from Algorithm 4.9, the first loop has a complexity of:

$$\mathcal{O}((m-3)(m-1)m) = \mathcal{O}(3m - 4m^2 + m^3) = \mathcal{O}(m^3)$$

Using the same approach, we find that the second time-consuming loop has a complexity of $\mathcal{O}(m^2 n)$. Note that as $n = m + 1$ for the parameter sets that we consider, both loops are also $\mathcal{O}(n^3)$, which is the same as the complexity of the entire function.

Optimization

Fortunately, the loop structure of the time-consuming loops in the algorithm is very suitable for vectorization, meaning the C compiler might already have optimized the loops to a degree that we cannot improve upon. Nevertheless, we try to optimize the loops by applying vectorization to the innermost loop, which always loops from 0 to `MEDS_m` (`MEDS_n` for the second loop).

As the field elements are stored in 16 bits, we can compute eight elements at the same time using 128-bit NEON registers. Therefore, we compute the results for eight values of r at the same time. We start by loading `tmp1`, which gives us a small problem, as these eight values are not stored sequentially in memory. Therefore, we load these values using normal load instructions and store them in a 128-bit NEON register. The remaining loads and stores can be done using vectorized instructions as those values are stored next to each other in memory.

As with the matrix systemizer, we are unable to use a full 8-way vectorization as the algorithm uses multiplication instructions, which result in 32-bit values. We apply the approach used in the matrix systemizer to work around this issue, see Section 4.2.2.

Handling non-multiples of eight

Similar to the matrix systemizer function, the number of elements that need to be processed is usually not a multiple of eight. We use the same approach as with the matrix systemizer function to handle this issue (see Section 4.2.2), meaning we decrease the parallelization factor to four and one when there are not enough elements left to process.

Implementation

As the `solve_opt` function is very large, we have decided not to implement the full function in ARMv8 assembly. This presented us with two options: either we use NEON intrinsics to parallelize the two time-consuming loops, or we write (a script that generates) ARMv8 assembly code for the two time-consuming loops. As the loops themselves are not very complex, we have decided to use NEON intrinsics to parallelize them, under the assumption that the assembly variant would not be much faster. Exploring the assembly-based optimization of these loops and the full function is left as future work, see Section 7.1. The optimized C code with NEON intrinsics is quite elaborate, so we do not show it here. Instead, we refer to the code in the code repository.[3]

Minimum cycle bound

We establish a minimum cycle bound for the part of the `solve_opt` function that we optimized: the two time-consuming loops. As they share the same structure, we show a general approach for establishing a minimum cycle bound for these loops, which we then apply to both loops to obtain a minimum cycle bound for the combined loops.

[3] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/opt-low-level/src/util.c

We call x the number of iterations of the outermost loop, y the number of iterations of the middle loop, and z the number of iterations of the innermost loop. As with matrix multiplication, we assume infinite registers and that all values are loaded and stored only once. The minimum number of instructions for the triple-nested loop can be calculated as follows:

- $x \cdot y$ loads of `tmp2` (not dependent on z);
- $y \cdot \frac{1}{8}z \cdot 8$ loads to load the 8 values of `tmp1` (not dependent on x);
- $x \cdot y \cdot \frac{1}{8}z \cdot 2$ multiplications and reductions to compute `prod_low/high`;
- $x \cdot y \cdot \frac{1}{8}z$ combinations of `prod_low` and `prod_high` to compute `prod`;
- $x \cdot \frac{1}{8}z$ loads to load `val` (not dependent on y);
- $x \cdot y \cdot \frac{1}{8}z$ additions, subtractions, and final reductions to compute `val`;
- $x \cdot \frac{1}{8}z$ stores of `val` (not dependent on y).

Given that reductions take five cycles, freezes take three cycles, and all other operations take one cycle, we establish a minimum bound of:

$$xy + yz + \frac{1}{8}xyz \cdot (2 \cdot (1+5) + 1 + (1+1+3)) + \frac{1}{4}xz = xy + yz + \frac{1}{4}xz + \frac{18}{8}xyz$$

We have $x = m-3, y = m-1, z = m$ for loop 1 and $x = m-2, y = m-1, z = n$ for loop 2, where m and n are the values of the parameter set that is used (as shown in Table 2.1). This gives us a total of

$$\begin{aligned}&(m-3)(m-1) + (m-1)m + \frac{1}{4}(m-3)m + \frac{18}{8}(m-3)(m-1)m \\ &= \frac{9}{4}m^3 - \frac{27}{4}m^2 + m + 3\end{aligned}$$

cycles for loop 1 and

$$\begin{aligned}&(m-2)(m-1) + (m-1)n + \frac{1}{4}(m-2)n + \frac{18}{8}(m-2)(m-1)n \\ &= \frac{9}{4}m^2n + m^2 - \frac{11}{2}mn - 3m + 3n + 2\end{aligned}$$

cycles for loop 2, giving a combined minimum cycle bound of:

$$\frac{9}{4}m^3 + \frac{9}{4}m^2n - \frac{23}{4}m^2 - \frac{11}{2}mn - 2m + 3n + 5$$

Theoretical speedup

As with the matrix multiplication and systemizer algorithms, we establish a theoretical speedup factor for the optimized isometry derivation algorithm.

Using a similar approach as in the previous section, we establish the bound for the two time-consuming loops in the non-optimized isometry derivation algorithm:

$$13m^3 + 13m^2n - 47m^2 - 36mn + 25m + 21n + 5$$

The theoretical speedup between the two algorithms is then given by:

$$\frac{13m^3 + 13m^2n - 47m^2 - 36mn + 25m + 21n + 5}{\frac{9}{4}m^3 + \frac{9}{4}m^2n - \frac{23}{4}m^2 - \frac{11}{2}mn - 2m + 3n + 5}$$

The theoretical and actual speedup factors are compared in Section 5.1.

4.3 High-level optimization

The high-level optimization approach focuses on optimizing the MEDS implementation by parallelizing over the challenge space. MEDS uses a large number (t) of challenges and commitments in both signing and verification, which are all computed independently in a for-loop. Additionally, the computation for each commitment is the same for signing and very similar for verification. This means that we can parallelize the computation of the commitments for each challenge.

Before we can parallelize the computation of the commitments, we need to determine the number of commitments that we compute in parallel. The commitment computation is executed over $\mathbb{F}_{4093}$, which means that the field elements can be stored in 12 bits (which fits in a 16-bit register). This means that we can store up to eight field elements in a single 128-bit NEON register. From this, there are two possible parallelization factors we can consider.

1. **Four-way parallelization**:
   Nearly all operations on field elements allow the values to stay in 16-bit registers. An exception is multiplication: the result of a multiplication of two field elements is a 24-bit value, which fits in a 32-bit register. This means that we can compute four multiplications at the same time using 128-bit NEON registers. Because of this limitation, we can choose to compute four commitments at a time.

2. **Eight-way parallelization**:
   We can work around the multiplication limitation by using instructions such as `umull` and `umull2` to multiply the lower and upper four values (respectively) of two 128-bit NEON registers, after which we can reduce the results and combine them back into a single 128-bit register. Using this technique, we can compute eight commitments at the same time.

Initially, we implemented the 4-way parallelization approach as it is easier to implement. However, we found that the 8-way parallelization approach is

more efficient, as the extra parallelization far outweighs the small overheads introduced by the more complex implementation. Therefore, we focus on the 8-way parallelization approach in this section.

In this section, we discuss the high-level optimization of the MEDS implementation. We first discuss the alterations that need to be made to the underlying datatypes and supplemental algorithms to allow for high-level parallelization in Section 4.3.1. Next, we discuss the actual parallelization of the commitment computations in Section 4.3.2. Finally, we discuss the limitations of the high-level optimization approach in Section 4.3.3.

4.3.1 Parallelization of datatypes and supplemental algorithms

In the reference implementation of MEDS, the main datatypes used are defined as follows:

```
#define GFq_t uint16_t
#define pmod_mat_t GFq_t
```

These two datatypes represent a field element and a matrix element, respectively. Matrices are stored as one-dimensional arrays in row-major order and are passed to functions using pointers to a `pmod_mat_t`.

To parallelize the algorithms that use these datatypes, we adjust the datatypes such that they represent multiple field/matrix elements at the same time. In our implementation, we add a `vec_16x8.h` file[4] containing these new datatypes, which are defined as follows:

```
#define GFq_vec_t uint16x8_t
#define pmod_mat_vec_t GFq_vec_t
```

These two datatypes represent eight field elements and eight matrix elements, respectively. By passing these datatypes to a modified version of each function, we can compute the function output for eight commitments at the same time.

[4] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/opt-high-level/include/vec_16x8.h

Additionally, we define a wide range of functions that operate on these vectorized datatypes in the `vec_16x8.h` file. Examples of such definitions include:

```
#define ADD_VEC(a, b) vaddq_u16(a, b)
#define MUL_VEC_LOW(a, b) vmull_u16(vget_low_u16(a),
                                    vget_low_u16(b))
#define MUL_VEC_HIGH(a, b) vmull_high_u16(a, b)
#define OR_VEC(a, b) vorrq_u16(a, b)
```

The main advantage of this approach is that the entire high-level-optimized MEDS implementation can easily be converted to use a different CPU architecture (that supports vectorization) such as AVX512 [63] simply by changing the definitions in the `vec_16x8.h` file.

Supplemental algorithms

To parallelize the commitment computations, we need to modify the supplemental algorithms that are used in the MEDS implementation. In the reference implementation, these algorithms perform operations for matrices or values that belong to a single commitment. We need to modify these algorithms such that they can perform operations for multiple commitments at the same time. The following algorithms need to be modified:

1. **Matrix multiplication**: `pmod_mat_mul`
   Responsible for multiplying two matrices over the finite field $\mathbb{F}_{4093}$.
2. **Matrix systemizer**: `pmod_mat_syst_ct_partial_swap_backsub`
   Responsible for converting a matrix into a systematic form (Algorithm A.7, Appendix A.3).
3. **Field inversion**: `GF_inv`
   Responsible for inverting an element over the finite field $\mathbb{F}_{4093}$.
4. **Matrix inversion**: `pmod_mat_inv`
   Responsible for inverting a matrix over the finite field $\mathbb{F}_{4093}$.
5. **Isometry derivation**: `solve_opt`
   Responsible for deriving an isometry mapping ϕ by constructing and solving a sparse system of linear equations over the finite field $\mathbb{F}_{4093}$.
6. **Applying the π function**: `pi`
   Responsible for applying the π function (Algorithm A.5, Appendix A.3) to a pair of matrices.
7. **Applying the SF function**: `SF`
   Responsible for applying the SF function (Algorithm A.6, Appendix A.3) to a matrix.

Each of these algorithms will be modified to work with the vectorized datatypes defined in the `vec_16x8.h` file. These algorithms will then be plugged into the MEDS implementation, which will be modified to compute multiple commitments at the same time.

To give a brief overview of the modifications that need to be made to these algorithms, we provide an example for the matrix multiplication algorithm. The reference matrix multiplication algorithm is shown in Algorithm A.4 (Appendix A.3). The C code for this algorithm is shown in Algorithm 4.10. The modified matrix multiplication algorithm is shown in Algorithm 4.11. Both algorithms are slightly simplified for the sake of readability.

Algorithm 4.10 Matrix multiplication (non-vectorized)

```
1  void pmod_mat_mul(pmod_mat_t *C, int C_r, int C_c,
2                    pmod_mat_t *A, int A_r, int A_c,
3                    pmod_mat_t *B, int B_r, int B_c)
4  {
5    for (int c = 0; c < C_c; c++)
6      for (int r = 0; r < C_r; r++)
7      {
8        uint64_t val = 0;
9        for (int i = 0; i < A_c; i++)
10         val = val +
11               (uint64_t)pmod_mat_entry(A, A_r, A_c, r, i) *
12               (uint64_t)pmod_mat_entry(B, B_r, B_c, i, c);
13       C[r * C_c + c] = val % MEDS_p;
14     }
15 }
```

As can be seen from the code, the vectorized version is essentially the same as the non-vectorized version, but with certain operations altered so they work with the vectorized datatypes and functions defined in the `vec_16x8.h` file and thus perform 8 computations in parallel. The same approach is used for all other supplemental algorithms that require parallelization, their modified implementations can be found in the code repository.[5]

[5] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/tree/main/opt-high-level/src

Algorithm 4.11 Matrix multiplication (vectorized for eight commitments)

```
void pmod_mat_mul_vec(pmod_mat_vec_t *C, int C_r, int C_c,
                      pmod_mat_vec_t *A, int A_r, int A_c,
                      pmod_mat_vec_t *B, int B_r, int B_c)
{
  for (int c = 0; c < C_c; c++)
    for (int r = 0; r < C_r; r++)
    {
      pmod_mat_vec_w_t val_low = ZERO_VEC_W;
      pmod_mat_vec_w_t val_high = ZERO_VEC_W;
      for (int i = 0; i < A_c; i++)
      {
        val_low = MUL_ACC_VEC_LOW(val_low,
                          pmod_mat_entry(A, A_r, A_c, r, i),
                          pmod_mat_entry(B, B_r, B_c, i, c));
        val_high = MUL_ACC_VEC_HIGH(val_high,
                          pmod_mat_entry(A, A_r, A_c, r, i),
                          pmod_mat_entry(B, B_r, B_c, i, c));
      }
      pmod_mat_vec_t val_low_red = REDUCE_VEC_32BIT(val_low);
      pmod_mat_vec_t val_high_red = REDUCE_VEC_32BIT(val_high);
      C[r * C_c + c] = COMBINE_VEC(val_low_red, val_high_red);
    }
}
```

4.3.2 Parallelization of commitment computations

Using the modified datatypes and supplemental algorithms, we can parallelize the computation of the commitments in MEDS. As can be seen in the algorithms for both signing (Algorithm 2.2) and verification (Algorithm 2.3), the computation of the commitments is done in a for-loop that loops from 0 to t. As the generation of a commitment can fail (for example, when a randomly generated matrix is not invertible), the generation of a particular commitment is done in a while loop that runs until the commitment is successfully generated. This is why the signing and verification code exhibits the following structure:

```
// Initial definitions and loading
for (int i = 0; i < t; i++) {
  while (1) {
    // Compute commitment
    if (commitment is computed) break;
  }
}
// Final operations
```

This structure introduces a problem if we want to parallelize the computation of the commitments. We cannot simply parallelize the for-loop to use $\frac{t}{8}$

iterations, as the while loop might run for a different number of iterations for each commitment. Additionally, although the computation of two different commitments is independent, the computation of the same commitment (after a failure) is dependent on the previous computation of that commitment, as the seed for the random number generator is updated after each failure. This means that we also cannot parallelize the while-loop.

Optimization

We can overcome the aforementioned problems by using a different approach to parallelize the commitment computations. Instead of finishing the computation for a commitment before moving on to the next, we start by running the first computation attempt for each commitment. If a computation fails, we store the necessary information to retry the computation later. After the first computation attempt for all commitments is done, we run the second computation attempt for all commitments that failed the first time. This process is repeated until all commitments are successfully computed. This approach is inspired by [1]. It is suitable for parallelization and has the following structure:

```
// Initial definitions and loading
while (there are uncomputed/failed commitments) {
  // 1. Load data for next 8 uncomputed/failed commitments
  // 2. Compute these commitments in parallel
  // 3(a). Store results for successful commitments
  // 3(b). Store information for failed commitments
}
// Final operations
```

Of course, this introduces some overhead, as steps 1 and 3 in the structure above cannot be parallelized and take slightly more time than their equivalent in the reference implementation. However, the parallelization of step 2 will result in a significant speedup, as the computation of the commitments itself is by far the most time-consuming part of the MEDS implementation.

Verification

For the signature verification algorithm, the same approach can be used with a few minor adjustments. The verification algorithm is shown in Algorithm 2.3. As can be seen, the verification of a commitment can be done in two ways. If h_i is not zero, we compute the isometry mapping based on the commitment that is contained in the signature. If h_i is zero, we re-compute the isometry mapping in the same way as in the signing algorithm.

A key observation is that in both cases, the same matrix multiplication and isometry derivation algorithms are used, with the same dimensions and

parameters. The only difference is the source of the matrices. This means that we can use the same parallelization approach described earlier, with a small change in the way the matrices are loaded. This is done separately for each matrix, which will bring a small overhead, but will still result in a significant speedup.

4.3.3 Limitations

Theoretically, we should expect that this approach will result in a speedup factor of eight. However, there are a few important reasons that will prevent us from reaching this speedup factor:

1. **Overhead**:
   As mentioned earlier, the parallelization approach introduces some overhead. This overhead is caused by the fact that we cannot parallelize the loading of the data and the storing of the results.

2. **Memory bandwidth**:
   The parallelization approach will result in a much higher memory bandwidth usage, as the inputs and outputs of each supplemental algorithm are now 8 times larger. This implies that the inputs and outputs to supplemental algorithms will no longer fit in the L1 cache of the CPU, which will result in a larger amount of cache misses, causing a severe slowdown.

4.4 Bitstream filling

After the commitments are computed, the resulting matrices are hashed, after which the hash output is converted into an array of t challenges. The input to the hash function is an array of 8-bit unsigned integers. For the MEDS parameter sets that we consider, the matrix elements fit in 12 bits and are stored as 16-bit unsigned integers. This means that the 16-bit values need to be converted to 8-bit values before they can be hashed. Although this can be done by simply changing the pointer type (for C implementations), MEDS uses the more complex approach of only using the 12 least significant bits (which contain the field element) of each 16-bit value. These 12-bit values are concatenated into a bitstream which is then hashed.

The main advantage of this approach is that the number of 8-bit values that need to be hashed is reduced by a factor of $\frac{16}{12} = \frac{4}{3}$, lowering the number of calls to the hash function. However, it also comes with the disadvantage that the process of filling the bitstream is more complex and time-consuming than the alternative of changing the pointer type.

We discuss the optimization of the existing approach for filling the bitstream. The alternative approach of changing the pointer type is left as future work, see Section 7.1.

The reference implementation[6] for filling the bitstream is generalized to work with up to 32 bits per value and therefore has to keep track of and update the current bit position in the bitstream. This means that we can save some time by using a specialized approach for filling bitstreams with 12-bit values.

Our optimization relies on the observation that two 12-bit values can be converted to three 8-bit values (which are required by the hash function) using a few simple bitwise operations. A conceptual pseudocode implementation of this approach is shown in Algorithm 4.12. The actual implementation[7] is a bit more complex as the structure of the input data is different and it needs to handle the case where the number of values is not a multiple of two. The high-level variant[8] is also slightly different as it needs to parallelize this operation.

Algorithm 4.12 Bitstream filling for 12-bit values

1: **Input:** v: array of n 16-bit unsigned integers
2: **Output:** bs: array of $\frac{3}{2}n$ 8-bit unsigned integers
3: $bs \leftarrow$ empty array of size $\frac{3}{2}n$
4: $i \leftarrow 0$
5: **for** $j \leftarrow 0$ to n in steps of 2 **do**
6: $\quad v_1 \leftarrow v[j]$
7: $\quad v_2 \leftarrow v[j+1]$
8: $\quad bs[i] \leftarrow v_1$ & `0xFF`
9: $\quad bs[i+1] \leftarrow (v_1 \gg 8) \mid ((v_2$ & `0xF`$) \ll 4)$
10: $\quad bs[i+2] \leftarrow (v_2 \gg 4)$ & `0xFF`
11: $\quad i \leftarrow i+3$

[6] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/ref/src/bitstream.c
[7] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/opt-low-level/src/meds.c
[8] https://github.com/MeItsLars/MEDS-ARMv8-optimization-thesis/blob/main/opt-high-level/src/vec_16x8.c

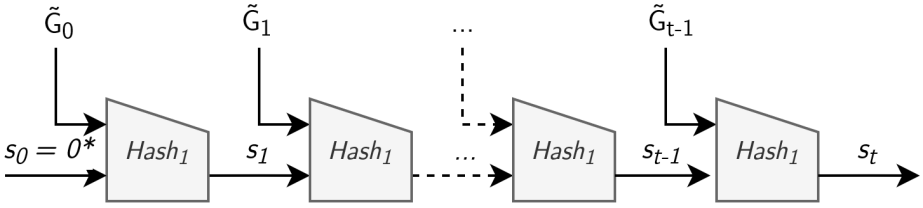

Figure 4.2: MEDS commitment hashing structure. $\tilde{\mathbf{G}}_i$ represents the commitment matrix for commitment i, $Hash_1$ represents a call to SHAKE256, and s_i represents the intermediate KECCAK state after hashing commitment i.

4.5 Hash structure

The result of the computation of each commitment is a large matrix. To convert these commitments into a set of challenges (as is required by the Fiat-Shamir structure, see Section 2.2.3), the commitments are hashed (using the SHAKE256 XOF) into a single KECCAK state. This state is then converted into an array of t challenges.

The commitment matrices are hashed sequentially, meaning that to hash commitment i, we are dependent on the hash output of commitment $i-1$. The structure of this hashing process is depicted in Figure 4.2. In this figure, $\tilde{\mathbf{G}}_i$ represents the commitment matrix for commitment i, converted such that it is stored in an array of $k(mn-k)$ unsigned 8-bit integers (see Section 2.2.4 for values of m, n, and k). $Hash_1$ represents a call to SHAKE256, which in turn calls the Keccak-f[1600] permutation until the entire input array is absorbed. s_i represents the intermediate KECCAK state after hashing commitment i, where s_0 is 0. The resulting state s_t is used to generate the array of challenges.

The hashing process for a single commitment is already quite computationally expensive. The number of bits to be hashed per commitment is $8 \cdot \lceil \frac{k(mn-k)\cdot b}{8} \rceil$, where b is the number of bits required to store a field element. For MEDS-55520 (see Section 2.2.4), this results in a value of 471648. As SHAKE256 uses a rate of 1088 bits, this means that the hashing process for a single commitment requires $\lceil \frac{471648}{1088} \rceil = 434$ calls to the Keccak-f[1600] permutation (for MEDS-55520).

Additionally, the hashing process is not parallelizable because of the sequential structure of the process. The profiling results (see Section 3.2) show that the hashing process takes up a little over 5% of the total execution time of signing and verification, which makes it worthwhile to explore possible optimizations.

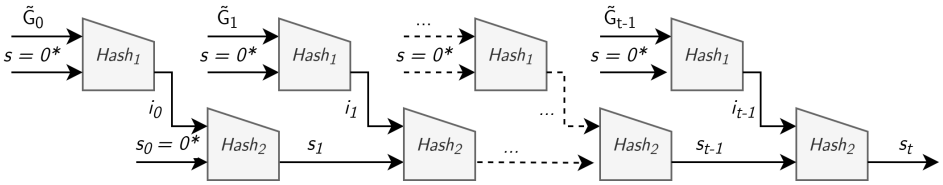

Figure 4.3: Optimized MEDS commitment hashing structure. $\tilde{\mathbf{G}}_i$ represents the commitment matrix for commitment i, $Hash_1$ and $Hash_2$ represent calls to SHAKE256, and i_i and s_t represent intermediate and final KECCAK states, respectively.

4.5.1 Hash structure optimization

There is little to no room for improvement in the actual SHAKE256 XOF, as it (and the underlying KECCAK permutation) is already widely studied and optimized. However, we can optimize the structure of the hashing process in such a way that we can compute the hashing result for multiple commitments in parallel. To do this, we need to alter the structure of the hashing process. This altered structure is depicted in Figure 4.3. In this structure, the hashing structure is split into two stages:

1. **Stage 1**: $Hash_1$
   In the first stage, the commitment matrices are hashed into a set of intermediate states $i_0, i_1, \ldots, i_{t-1}$ using the same $Hash_1$ function as in the original structure. This stage can be parallelized, as the hashing of each commitment no longer depends on other commitments.

2. **Stage 2**: $Hash_2$
   In the second stage, we combine the intermediate states $i_0, i_1, \ldots, i_{t-1}$ into a single state s_t that we can use to generate the array of challenges. We do this by repeatedly applying the $Hash_2$ function (which also calls SHAKE256) to the intermediate states and the running state s. This stage is once again sequential, but it only requires t calls to the Keccak-f[1600] permutation.

The resulting structure introduces a small amount of overhead, as stage 2 requires t additional calls to the Keccak-f[1600] permutation. However, as stage 1 can now be parallelized, the overall execution time of the hashing process is expected to decrease.

Unfortunately, the final state s_t that the optimized structure generates is different from the final state that the original structure generates. This means that the resulting challenges will also be different between the two structures, meaning that the optimized and reference implementations are not compatible with each other. Therefore, we leave this optimized hash structure as a suggested change to the MEDS scheme.

4.5.2 Implementation

The implementation of the optimized hash structure into the MEDS implementation is trivial and requires only one important detail: a parallelized version of SHAKE256, optimized for the ARMv8 architecture. The extended KECCAK code package [80] does not contain vectorized implementations of the Keccak-f[1600] permutation for ARMv8 (only for ARMv7). Fortunately, recent research by Becker and Kannwischer [18] has resulted in a large set of optimized and vectorized implementations of the Keccak-f[1600] permutation for ARMv8 and various extensions of the ARMv8 architecture.

We have benchmarked these implementations on the Cortex-A72 and found that the `keccak_f1600_x4_hybrid_asm_v3p`[9] optimized permutation (which can process 4 states in parallel) requires the lowest amount of cycles per state permutation on the ARM Cortex-A72. As the number of challenges t is a multiple of 4 for all MEDS parameter sets that we consider, this means that we can easily plug this optimized permutation into the new hash structure. On the Apple M2, we found that the `keccakx2_bas`[10] optimized permutation (which can process 2 states in parallel) requires the lowest amount of cycles per state permutation. This variant was developed by Westerbaan [79] and is also used in SPHINCS$^+$ [10]. The main advantage of this permutation is the fact that it uses the cryptographic extension of the ARMv8 architecture (available on the Apple M2 but not on the ARM Cortex-A72), which contains instructions that accelerate the parallel performance of the Keccak-f[1600] permutation.

4.6 Non-constant-time implementations

To prevent timing and cache-based side-channel attacks (see Section 2.4), the MEDS implementation must be constant-time. However, this is only the case for the key generation and signing phases of MEDS. The verification phase is not required to be constant-time, as it operates only on public data. This means that we can use non-constant time code for the verification phase. In this section, we list the functions that can be optimized using non-constant time implementations. The implementation of each optimization is slightly different for the low-level and high-level approaches, but the idea is the same.

4.6.1 Finite field inversion

MEDS requires the inversion of field elements over the finite field $\mathbb{F}_{4093}$. The constant-time algorithm used in the reference implementation utilizes an

[9] https://gitlab.com/arm-research/security/pqax/-/blob/master/asm/manual/keccak_f1600/keccak_f1600_x4_hybrid_asm_v3p.s

[10] https://gitlab.com/arm-research/security/pqax/-/blob/master/tests/keccak_neon/manual/third_party/keccakx2_bas.s

optimal addition chain based on the inversion approach of Fermat's Little Theorem, which uses 115 instructions. However, as the number of possible field elements is limited to 4093, we can precompute the inverse of each field element and store it in a lookup table. This allows us to invert a field element simply by executing an array lookup, which takes only a few instructions to execute.

4.6.2 Matrix systemizer

In the matrix systemizer (shown in Algorithm A.7), the algorithm spends some time making sure the leading coefficient of each row is nonzero (it almost always is). As the implementation is constant time, this entire process is executed even if the leading coefficient is already nonzero. In a non-constant time implementation, we add a check to see if the leading coefficient is zero, and if it is not, skip the entire process of making it nonzero.

Chapter 5

Results

In this chapter, we present the results of the optimizations that we have performed on the MEDS implementation. We compiled the code using gcc (Debian 12.2.0-14) 12.2.0 with the `-O3` optimization flag and executed it on the ARM Cortex-A72 (ARMv8-A) clocked at 1.5 GHz with frequency scaling disabled. The overall performance of the scheme, discussed in Section 5.3, is also analyzed on the Apple M2 (ARMv8.6-A), on which we compiled the code using gcc (Apple clang-1500.3.9.4) 15.0.0 with the `-O3` optimization flag and executed it on one of the performance cores clocked at 3.49 GHz. In all tables and figures, the numbers shown represent the number of cycles, kilocycles (KCycles, 1 KCycle = 1 thousand cycles), or megacycles (MCycles, 1 MCycle = 1 million cycles) that the respective algorithm or function took to execute.

We first look at the performance results of the three algorithms that were optimized specifically for the low-level approach in Section 5.1. After that, we look at the post-optimization profiling results of the functions that originally took up the most time in the MEDS implementation in Section 5.2. Following this, we look at the overall performance results of all three MEDS parameter sets for both the reference implementation, low-level optimized implementation, and high-level optimized implementation in Section 5.3. Finally, we compare the performance of the optimized MEDS implementation to other state-of-the-art signature schemes in Section 5.4.

5.1 Low-level optimizations

In our low-level optimization approach (see Section 4.2), we optimized matrix multiplication, the matrix systemizer, and isometry derivation. For each of these functions, we established a lower bound on the number of cycles that the (optimized part of the) function should take to execute. The calculated

Table 5.1: Results of low-level algorithm optimizations for parameter set MEDS-55520. 'Cycles' represents the number of cycles that the reference (Ref.) or optimized (Opt.) function (part) took to execute on the Cortex-A72. 'Bound' represents the lower bound (as calculated in the relevant section in Chapter 4) on the number of cycles that the function (part) should take to execute. 'Ratio' represents the ratio between the number of cycles that the optimized function (part) took to execute and the lower bound. For the Matrix Systemizer, the arguments (as explained in Section 4.2.2) are represented by * (apply back substitution), ** (apply systemizer to a limited number of rows), and *** (swap columns to ensure non-zero leading coefficients).

Function	**Input size**		**Cycles**	**Cycles**	**Bound**	**Ratio**
	A	B	(Ref.)	(Opt.)		
Matrix Multiplication	$2 \times k$	$k \times k$	6450	1348	824	1.63
Matrix Multiplication	$2 \times mn$	$mn \times k$	226217	39234	28568	1.37
Matrix Multiplication	$k \times mn$	$mn \times k$	3757155	489947	404744	1.21
Matrix Multiplication	$m \times n$	$n \times m$	109556	15947	12044	1.32
Matrix Multiplication	$m \times n$	$n \times n$	111334	16794	12351	1.36
Matrix Multiplication	32×32	32×32	89725	10750	9856	1.09
Matrix Systemizer	$k \times k$		169141	95299	42800	2.23
Matrix Syst.	$k \times 2k$		292309	175155	99805	1.75
Matrix Syst. (bsub*)	$n \times 2n$		453498	292422	156774	1.87
Matrix Syst. (bsub*)	$m \times 2m$		407491	261331	143984	1.82
Matrix Syst. (bsub*)	$k \times 2k$		407583	261028	143984	1.81
Matrix Syst. (n-1**) (bsub*)	$n \times 2m$		422529	269895	149018	1.81
Matrix Syst. (swap***) (bsub*)	$m - 1 \times m$		190097	123796	43413	2.85
Isometry Derivation (part)	$2 \times mn$		1142883	350909	166319	2.11

lower bounds apply to the algorithms that we used for the optimizations. Using a different algorithm might result in a different lower bound.

To determine how close we got to our calculated bound, we benchmarked the optimized functions for parameter set MEDS-55520. Together with the cycle count of the original function and the calculated minimum bound for each function (part), this gives us an understanding of how close we got to the optimal performance of the function (with the specific algorithm that we used). The results for these optimizations are shown in Table 5.1. Each benchmark result was obtained by running that function 128 times and taking the median of the timing results.

Because of our assumption that we have infinite registers (and therefore only need to load or store each value once) in the calculation of the minimum cycle bound, none of the functions reach the lower bound. This is especially true for the matrix systemizer and isometry derivation functions, both of

these use a lot of intermediate loads and stores and are therefore relatively far from the lower bound.

The results show that we were able to optimize the matrix multiplication function for the 32×32 case to within 9% of the lower bound. This case is not used in MEDS, but we added it to show that the optimization works almost perfectly for matrices of which the dimensions are a multiple of 8. The other cases show that the optimization works fairly well for matrices with more than 8 rows and columns. These cases are unable to reach the lower bound because of the overhead of having to deal with dimensions that are not a multiple of 8, but the results are still very good. The worst case arises when one of the input matrices has only 2 rows. This makes sense, as we are unable to use full 8-way parallelization and therefore lose performance. The optimization of matrix multiplication for these cases might be improved by using a different approach to parallelization, which we leave as future work (see Section 7.1).

When looking at the matrix systemizer, we see that we get reasonably close to the lower bound for most cases. The difference is mostly caused by the assumption that we have infinite registers, which is not the case in reality. The exceptions are the cases where the matrix size is relatively small, which is to be expected. For these matrix sizes, the additional overhead of having to deal with dimensions that are not a multiple of 8 is relatively large, compared to the parallelization speedup.

Finally, we consider the isometry derivation function. The results show that we were able to optimize this function to within $2.11\times$ of the lower bound. This is a major improvement over the reference implementation, but it is still quite far from the lower bound. This can be attributed to the infinite register assumption and the fact that we only optimized the two most time-consuming parts of the function. Furthermore, we used NEON intrinsics for the optimization, whereas an assembly implementation might have resulted in a larger speedup. We leave this as future work (see Section 7.1).

5.1.1 Theoretical and actual speedup factors

Besides determining the proximity of the optimized functions to the calculated lower bound, we also calculate the theoretical and actual speedup factors for each optimization to get an understanding of how well we optimized the functions compared to the theoretical expectation. The results are shown in Table 5.2.

Table 5.2: Theoretical and actual speedup factors of the low-level optimizations for parameter set MEDS-55520. Factors are calculated as the ratio between the number of cycles that the reference function took to execute and the number of cycles that the optimized function took to execute. The cycle counts are taken from Table 5.1 and are omitted here for brevity. Additionally, the definitions of *, **, and *** are also taken from Table 5.1.

Function	**Input size**		**Factor**	**Factor**
	A	B	Theoretical	Actual
Matrix Multiplication	$2 \times k$	$k \times k$	7.6	4.8
Matrix Multiplication	$2 \times mn$	$mn \times k$	7.6	5.8
Matrix Multiplication	$k \times mn$	$mn \times k$	7.5	7.7
Matrix Multiplication	$m \times n$	$n \times m$	7.5	6.9
Matrix Multiplication	$m \times n$	$n \times n$	7.5	6.6
Matrix Multiplication	32×32	32×32	7.5	8.3
Matrix Systemizer	$k \times k$		3.6	1.8
Matrix Syst.	$k \times 2k$		3.7	1.7
Matrix Syst. (bsub*)	$n \times 2n$		4.1	1.6
Matrix Syst. (bsub*)	$m \times 2m$		4.1	1.6
Matrix Syst. (bsub*)	$k \times 2k$		4.1	1.6
Matrix Syst. (n-1**) (bsub*)	$n \times 2m$		4.1	1.6
Matrix Syst. (swap***) (bsub*)	$m - 1 \times m$		3.7	1.5
Isometry Derivation (part)	$2 \times mn$		5.7	3.3

The results show that especially for the matrix multiplication function, the actual speedup factors are close to the theoretical speedup factors, indicating that we were able to optimize these functions very well. Again, the exceptions are the cases where the matrix size is relatively small, in which case we are unable to use full 8-way parallelization, causing the actual speedup factor to be lower than the theoretical speedup factor. For the matrix systemizer and isometry derivation functions, the actual speedup factors are lower than the theoretical speedup factors. This is because the C compiler was already able to optimize these functions to a certain extent, which means that the speedup factor that we could achieve was limited.

5.2 Profiling implementation variants

The results of profiling the reference implementation were shown earlier in Section 3.2. In this section, we profile the optimized implementations (both low-level and high-level) for parameter set MEDS-55520 to obtain an understanding of how much impact each optimization had. The relative results for the other two parameter sets are very similar and are therefore omitted. As the main goal of profiling does not revolve around getting accurate measurements but rather around understanding the relative performance

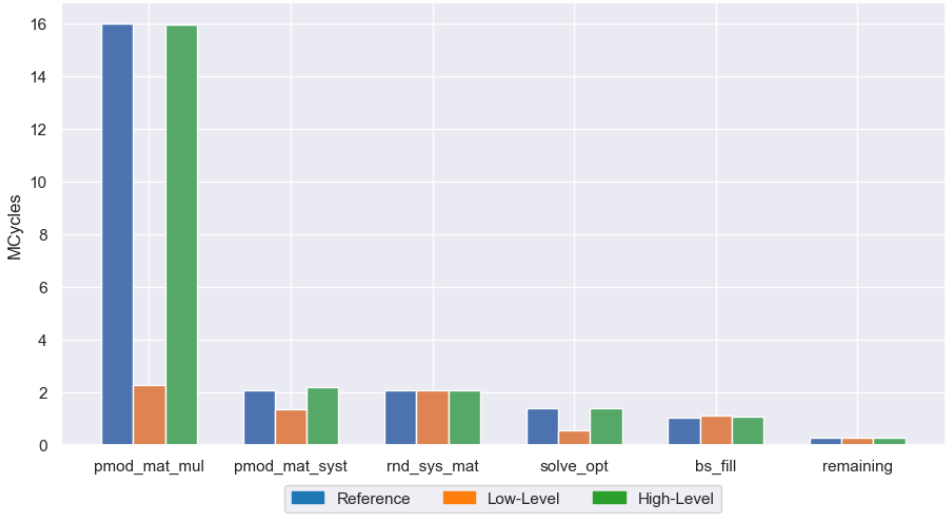

Figure 5.1: MEDS-55520 profiling results on the Cortex-A72 for functions used in key generation.

of (functions within) the different implementations, we only ran one profiling iteration for each of the three algorithms. The profiling results, depicted in bar charts, are shown in Figure 5.1 (key generation), Figure 5.2 (signing), and Figure 5.3 (verification).

The figures show major improvements for all three algorithms in both the low-level and high-level optimized implementations, with a few notable exceptions:

- The key generation algorithm used in the high-level optimized implementation is the same as the one in the reference implementation, because the parameter sets that we consider do not allow for large speedup factors for the high-level optimizations of this algorithm. We leave this as future work (see Section 7.1).
- In key generation, the `rnd_sys_mat` (responsible for generating random systemized matrices) and `bs_fill` (responsible for writing to a bitstream) functions have not been optimized, and therefore show no improvement. Generating random systemized matrices cannot be optimized, as it relies on the generation of random field elements using the SHAKE256 XOF (which is already optimized). Writing to a bitstream can be optimized using the approach described in Section 4.4, which we leave as future work.

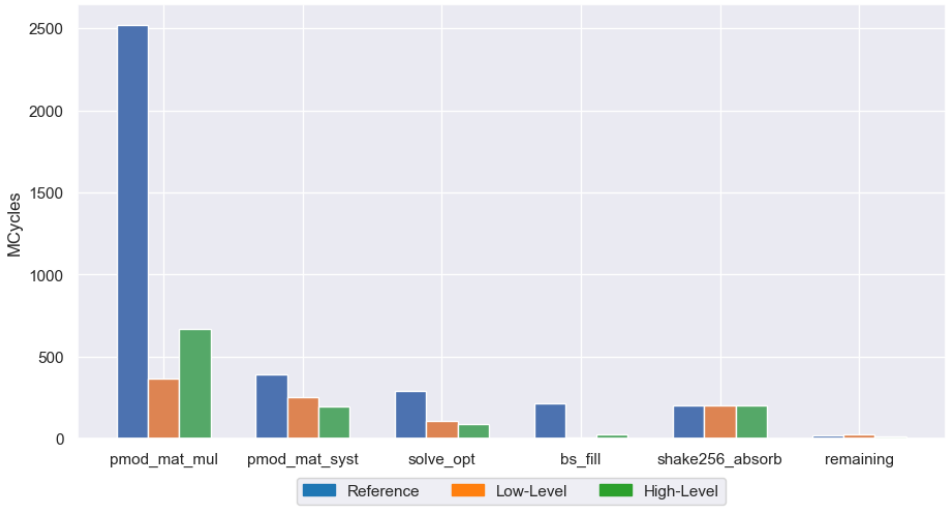

Figure 5.2: MEDS-55520 profiling results on the Cortex-A72 for functions used in signing.

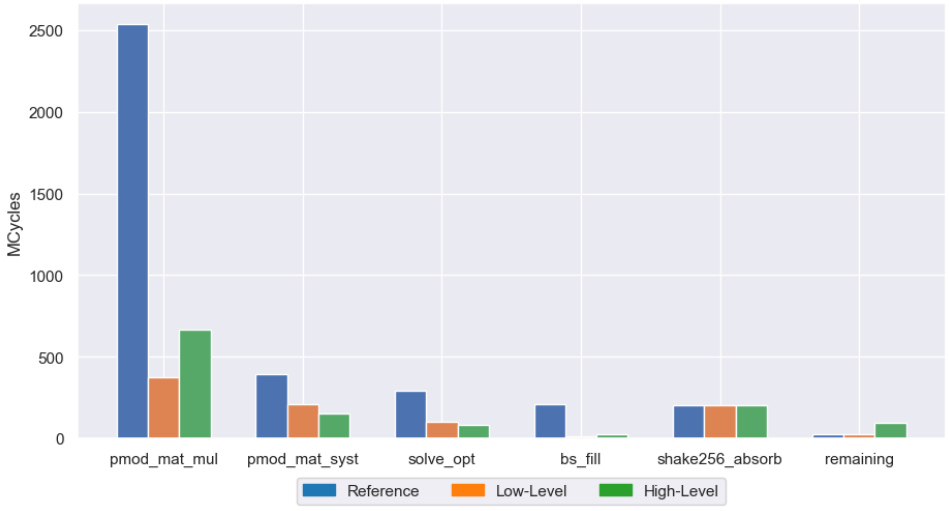

Figure 5.3: MEDS-55520 profiling results on the Cortex-A72 for functions used in verification.

- The `shake256_absorb` function used in signing and verification does not show any improvement. This is because the SHAKE256 XOF is already heavily optimized. The only thing that can be optimized is the structure of the hashing process, which we execute and explain in Section 4.5.

Besides these exceptions, the results show that the optimizations have been very effective. Particularly matrix multiplication shows a huge speedup, but the other functions also show major improvements. We note the following observations:

- The low-level optimized matrix multiplication function is faster than its high-level optimized counterpart. We believe that this is because the high-level optimized function needs to access 8 times as many values in memory as the low-level optimized function. This results in a larger amount of cache misses, which slows down the function.
- The high-level optimized matrix systemizer function is faster than its low-level optimized counterpart. We believe that this is because the low-level optimized function is not able to parallelize every aspect of the systemizer process (such as the inversion of field elements), whereas the high-level optimized function can do this. This results in a larger speedup for the high-level optimized function.
- The high-level optimized isometry derivation function is slightly faster than its low-level optimized counterpart. We believe that this is because the high-level optimized function parallelizes the entire function, whereas the low-level optimized function only parallelizes the two most time-consuming loops. The low-level optimized function might therefore benefit from further optimization, which we leave as future work (see Section 7.1).

5.3 Overall performance

We have benchmarked the performance of the MEDS for all combinations of the four relevant variables to consider:

- **Processor**: We benchmarked the performance on both the ARM Cortex-A72 and the Apple M2.
- **MEDS parameter set**: We analyzed the performance for each of the three considered parameter sets: MEDS-21595, MEDS-55520, and MEDS-122000 (see Section 2.2.4).
- **Algorithm**: We tested each of the three signature algorithms: key generation, signing, and verification.

- **Implementation variant**: We tested all implementations: reference, low-level optimized, high-level optimized, low-level optimized with alternative hash structure, and high-level optimized with alternative hash structure.

MEDS was benchmarked for every possible combination of these variables, resulting in a total of 45 benchmarks per CPU. The result for each benchmark was obtained in the following way:

1. 16 'warmup' runs were executed to ensure that the cache was filled with the necessary data and the branch predictor was optimized;
2. 128 measurement runs were executed to obtain 128 measurements of the execution time of the algorithm;
3. The final benchmark result was obtained by taking the median of the 128 measurements.

The exact results of the benchmarks on the Cortex-A72 are shown in Table B.1 (MEDS-21595), Table B.2 (MEDS-55520), and Table B.3 (MEDS-122000) in Appendix B. For the Apple M2, the results are shown in Table B.4 (MEDS-21595), Table B.5 (MEDS-55520), and Table B.6 (MEDS-122000) in Appendix B.

As the relative performance of the implementations is more important than the exact numbers and as we focused our optimization efforts on the Cortex-A72, we depict the Cortex-A72 results in the form of bar charts in Figure 5.4 (MEDS-21595), Figure 5.5 (MEDS-55520), and Figure 5.6 (MEDS-122000).

The results show that both the low-level and high-level optimizations have resulted in a major speedup of the MEDS implementation (except for high-level key generation) on both CPUs. The low-level optimized implementation is faster than the high-level optimized implementation for all algorithms and parameter sets. This is because the high-level optimized variant needs to transfer larger amounts of data to and from the various functions it uses, causing a larger amount of cache misses, which slows down the implementation.

Additionally, the results show that the alternative hash structure presented in Section 4.5 results in a small speedup for the signing and verification algorithms. It does not have a noticeable impact when applied to the reference implementation.

The speedup factors are larger on the Apple M2 than on the ARM Cortex-A72, especially for the variants with the optimized hash structure. This is because the Apple M2 can utilize the cryptographic extension of the ARMv8 architecture, which contains instructions that accelerate the parallel performance of the Keccak-f[1600] permutation.

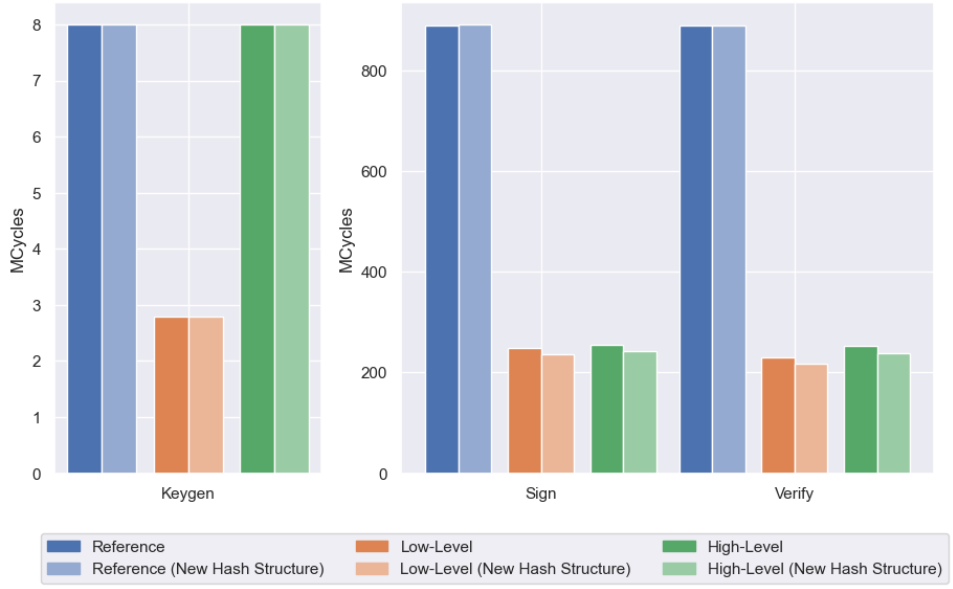

Figure 5.4: Overall performance of MEDS-21595 variants on the Cortex-A72. Low-Level refers to the low-level optimized implementation and High-Level to the high-level optimized implementation. The 'New Hash Structure' variants refer to the optimized hash structure described in Section 4.5.

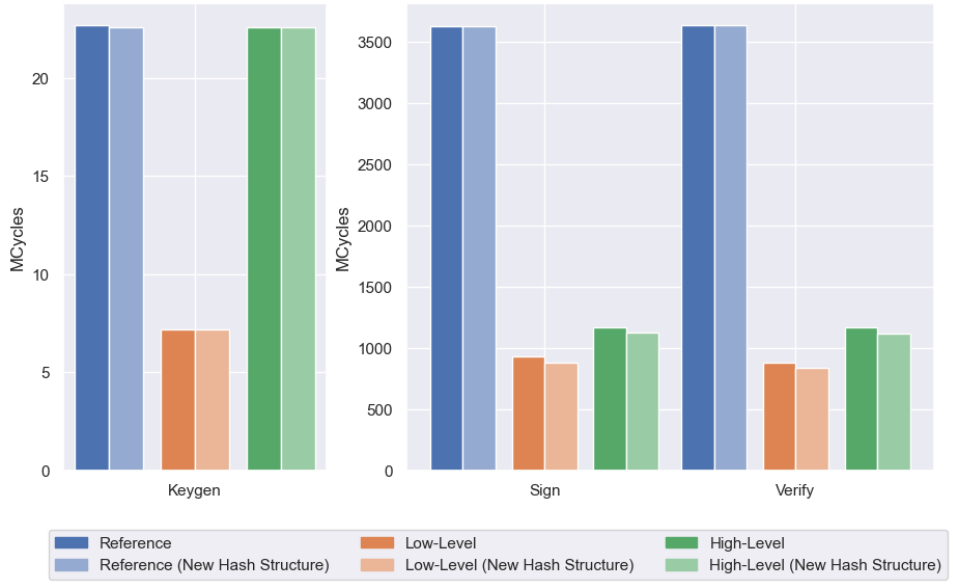

Figure 5.5: Overall performance of MEDS-55520 variants on the Cortex-A72. Low-Level refers to the low-level optimized implementation and High-Level to the high-level optimized implementation. The 'New Hash Structure' variants refer to the optimized hash structure described in Section 4.5.

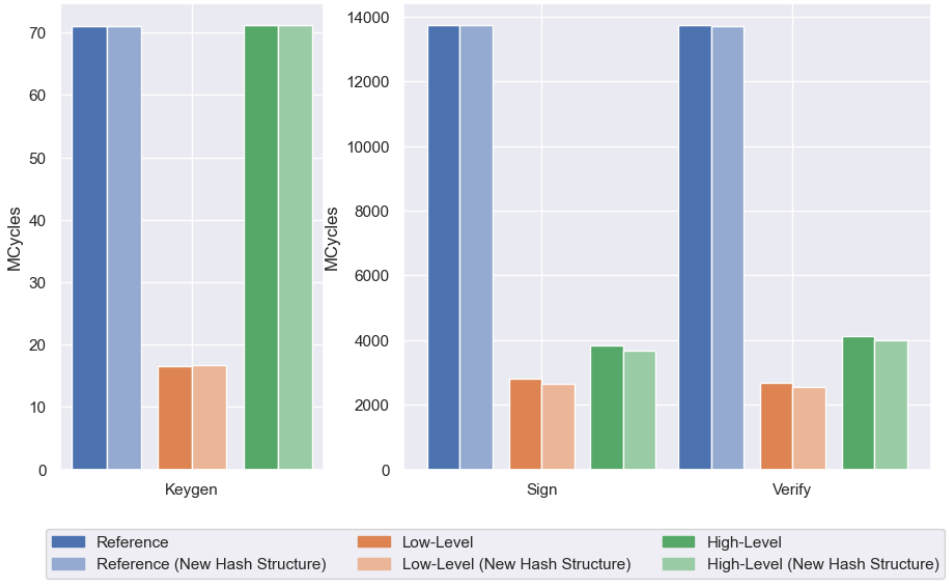

Figure 5.6: Overall performance of MEDS-122000 variants on the Cortex-A72. Low-Level refers to the low-level optimized implementation and High-Level to the high-level optimized implementation. The 'New Hash Structure' variants refer to the optimized hash structure described in Section 4.5.

5.4 Comparison to similar schemes

We compare the performance of MEDS on the Cortex-A72 to the performance of other digital signature schemes. Note that we compare to the self-reported performance of the schemes over various CPUs, which means the results are not directly comparable. Nevertheless, the comparison indicates the rough relative performance of the schemes. Besides this, note that the comparison is only based on the performance of the schemes and does not take key and signature sizes into account.

For reference, we include a comparison to Dilithium [36], Falcon [39], and SPHINCS$^+$ [10], the three selected signature schemes in the initial NIST post-quantum standardization process. More importantly, though, we compare to a set of related (mostly code-based) post-quantum digital signature schemes that are, like MEDS, in the current NIST post-quantum digital signature standardization process. The results are shown in Table 5.3.

When comparing the performance of MEDS to other similar post-quantum digital signature schemes, the results are mixed. ALTEQ, CROSS, PERK, and RYDE outperform MEDS for all three algorithms. UOV also outperforms MEDS by a large factor, except for key generation. MEDS performs relatively similarly to LESS, although for larger security levels, LESS is significantly

Table 5.3: Comparison of MEDS performance to self-reported times of other similar or relevant signature schemes in the NIST standardization process. The numbers represent the number of MCycles that the key generation, signing, and verification algorithms take on the mentioned CPUs: (1) AMD Ryzen 5 Pro 3500U, (2) Intel Core i7-12700, (3) Intel Comet Lake, (4) ARM Cortex-A72, (5) Intel i5-1135G7, (6) Intel Xeon E-25588G, (7) Intel Xeon E3-1220, (8) Intel Core i9-13900K. For SPHINCS$^+$ measurements, the SHA2-simple variant is used.

NIST Level	**Scheme**	**CPU**	**Keygen**	**Sign**	**Verify**
I	Dilithium2 (NEON) [17]	(4)	0.27	0.65	0.27
	Falcon512 (NEON) [59]	(4)	-	1.04	0.06
	SPHINCS$^+$-128s (AVX2) [10]	(7)	84.97	644.74	0.86
	Balanced-ALTEQ [26]	(6)	0.36	2.74	2.11
	CROSS-R-SDP-b [13]	(1)	0.04	2.38	1.44
	LESS-1b (AVX2) [12]	(2)	0.90	263.60	271.40
	MEDS-21595 (reference)	(4)	7.97	890.71	889.77
	MEDS-21595 (ours)	(4)	2.82	248.22	229.79
	PERK-I-short3 (AVX2) [3]	(8)	0.08	38.0	27.0
	RYDE-128S (AVX2) [5]	(8)	0.03	23.40	20.10
	uov-Ip-classic (NEON) [24]	(4)	11.17	0.25	0.14
	Wave822 [14]	(5)	13946.20	1156.18	206.10
III	Dilithium3 (NEON) [17]	(4)	0.52	1.09	0.45
	SPHINCS$^+$-192s (AVX2) [10]	(7)	125.31	1246.38	1.44
	Balanced-ALTEQ [26]	(6)	2.23	28.46	26.20
	CROSS-R-SDP-b [13]	(1)	0.08	4.97	2.89
	LESS-3b (AVX2) [12]	(2)	2.80	2446.90	2521.40
	MEDS-55520 (reference)	(4)	22.66	3623.80	3628.16
	MEDS-55520 (ours)	(4)	7.18	927.91	878.27
	PERK-III-short3 (AVX2) [3]	(8)	0.18	80.0	64.0
	RYDE-192S (AVX2) [5]	(8)	0.05	49.60	44.80
	uov-III-classic (NEON) [24]	(4)	66.87	1.54	0.57
	Wave1249 [14]	(5)	46285.89	3534.75	467.36
V	Dilithium5 (NEON) [17]	(4)	0.78	1.44	0.77
	Falcon1024 (NEON) [59]	(4)	-	2.14	0.13
	SPHINCS$^+$-256s (AVX2) [10]	(7)	80.94	1025.72	1.99
	CROSS-R-SDP-b [13]	(1)	0.14	8.26	5.00
	LESS-5b (AVX2) [12]	(2)	6.40	10212.60	10458.80
	MEDS-122000 (reference)	(4)	71.02	13748.90	13731.38
	MEDS-122000 (ours)	(4)	16.65	2794.53	2693.83
	PERK-V-short3 (AVX2) [3]	(8)	0.31	182.0	142.0
	RYDE-256S (AVX2) [5]	(8)	0.07	105.50	95.90
	uov-V-classic (NEON) [24]	(4)	313.81	3.32	1.32
	Wave1644 [14]	(5)	106260.00	7851.48	806.86

slower at generating and verifying signatures. The only scheme that MEDS outperforms for all three algorithms is Wave. Although these results do not seem too promising, it is important to note that, except for Wave and UOV, MEDS provides smaller signature sizes than all of these schemes. This makes MEDS a viable competitor in the digital signature scheme landscape.

When comparing the performance of MEDS to standardized post-quantum digital signature schemes, we see that MEDS is significantly slower than Falcon and Dilithium but can compete with SPHINCS$^+$. Although MEDS and a lot of other schemes are slower than Falcon and Dilithium, they can still be viable alternatives. Firstly, there can be specialized use cases where performance is not the most important factor, and where the potentially smaller key or signature sizes of alternative schemes are more important. Secondly, standardizing schemes that rely on different mathematical problems provides more diversity in security assumptions. If the security of structured lattices (on which both Falcon and Dilithium rely) is ever broken, it is important to have alternative schemes that rely on different mathematical problems.

5.5 Discussion

Throughout our research, we have found answers to our research questions.

RQ I. From our profiling efforts in Chapter 3, we have found that the matrix multiplication, the matrix systemizer, and isometry derivation functions are the most time-consuming functions in the MEDS implementation. Together, they account for over 87 % of the total execution time of the MEDS implementation on the Arm Cortex-A72. After optimizing these functions, we have found that these functions still take up a significant amount of the total execution time. Additionally, we found that the KECCAK permutation now takes up a significant amount of time in all three algorithms.

RQ II. We have found that the low-level optimization approach results in a large speedup for all parameter sets on both the ARM Cortex-A72 and the Apple M2. The high-level optimization approach is less effective but still results in a significant speedup. On architectures with larger cache sizes and/or larger SIMD registers, the high-level optimization approach might be more effective. As for the low-level approach, we optimized the matrix multiplication, matrix systemizer, and isometry derivation functions using NEON instructions. Additionally, we obtained a speedup by optimizing the bitstream filling process and suggested a new hash structure that allows for parallelization of the hashing process, providing another small speedup on the Cortex A72 and a larger speedup on the Apple M2 which supports the cryptographic extension of the ARMv8 architecture.

a) The bitstream filling and hash restructure speedups can be applied to the reference implementation as they do not require any ARMv8-specific instructions. The remaining optimizations (which make up the bulk of the speedup) are ARMv8-specific and can therefore not directly be applied to the reference implementation. However, most other architectures have their own SIMD instruction set, which can be used in the same way as NEON instructions and should achieve similar speedups.

Chapter 6

Conclusions

From the results in Chapter 5, it is clear that our optimizations have resulted in a major speedup of the MEDS implementation. The low-level optimization approach (as described in Section 4.2) has provided the highest speedup. On the ARM Cortex-A72, MEDS-21595 is sped up by a factor of 2.9, 3.6, and 3.9 for key generation, signing, and verification, respectively; MEDS-55520 by factors 3.2, 3.9, and 4.1; and MEDS-122000 by factors 4.3, 4.9, and 5.1. On the Apple M2, the speedup factors are slightly larger, with MEDS-21595 being sped up by factors 3.0, 3.7, and 4.1; MEDS-55520 by factors 3.8, 4.4, and 4.7; and MEDS-122000 by factors 4.4, 5.1, and 5.4. The high-level optimization approach (as described in Section 4.3) has provided a smaller speedup due to the limitations in the cache sizes of the CPUs.

We present an alternative hash structure in Section 4.5 that further improves the performance of signing and verification, at a negligible cost to the non-parallelized reference implementation. The performance is improved by another ± 5 % on the ARM Cortex-A72 and another ± 35 % on the Apple M2, which supports the cryptographic extension of the ARMv8 architecture and can therefore utilize instructions that accelerate the parallel performance of the Keccak-f[1600] permutation. Since MEDS produces different challenges and signatures when using this alternative hash structure, we leave it as a suggested change to the MEDS scheme.

We have shown that the MEDS implementation is very suitable for optimization using SIMD instructions. The achieved speedup on ARMv8 is big, but we expect that future research can achieve even larger speedups on other architectures that support SIMD instructions, such as AVX512 [63], on which 4 times as many field elements can be processed in parallel compared to ARMv8.

Compared to standardized post-quantum signature schemes, MEDS is heavily outperformed by Falcon and Dilithium. However, if the security of structured lattices (on which both Falcon and Dilithium rely) is ever broken, MEDS can serve as a viable alternative, as it is based on a different mathematical problem. When compared to other similar (code-based) post-quantum signature schemes, MEDS provides a smaller signature size than nearly all of them but is outperformed by most of them in terms of performance. The choice between these schemes should be based on a wide range of factors, among which are the security requirements of the application, the performance requirements of the key generation, signing, and verification algorithms, the private and public key sizes, and the signature size.

Chapter 7

Future Work

Although we have optimized the MEDS implementation significantly, we have mentioned several limitations and possible improvements. In this chapter, we discuss these ideas in more detail. Additionally, there are some other unexplored but interesting topics that we discuss.

7.1 Further optimization possibilities

Bitstream optimization

As explained in Section 4.4, the commitment matrices are hashed by first converting them into a bitstream. An alternative approach is to change the pointer type of the matrix elements from 16-bit to 8-bit unsigned integers (which would practically take no time to execute) and then hash the resulting array. The main advantage of this approach is that we no longer need to execute the relatively costly bitstream filling process. However, it also comes at the cost of having $\frac{4}{3}$ times as many values to hash. It would be interesting to see if this approach results in a speedup of the general hashing process.

Optimizing isometry derivation

Our optimizations to the isometry derivation function `solve_opt` as described in Section 4.2.3 have been very effective, resulting in a speedup factor of 2.7 for MEDS-55520 (see also the results in Section 5.2). However, our optimizations were solely based on the vectorization of the two most time-consuming loops in the `solve_opt` function using NEON intrinsics. We expect that an additional speedup can be achieved by optimizing the remaining loops in the function. Additionally, we believe that a pure-assembly approach will result in an even larger speedup.

Additional multiplication techniques for specific matrix dimensions

We have heavily optimized the matrix multiplication function, achieving an average matrix multiplication speedup factor of 6.9 for MEDS-55520 (see also the results in Section 5.2). Naturally, the optimizations are slightly less effective for matrices that use dimensions that are not a multiple of 8. For most matrix sizes, the results were still close to the lower bound. However, we noticed that matrices with dimensions $2 \times x$ were not optimized as well as we would have liked. The creation of an additional algorithm that is specifically tailored to such matrix sizes could pull the performance of these matrix multiplications closer to the lower bound.

High-level key generation optimization

We have not applied high-level optimizations to the key generation function. The main loop of the key generation generates all s matrices that represent the public key. For the parameter sets that we consider, s is always 2, which heavily limits the possibilities for parallelization. The speedup factor would be limited to 2, which we already exceeded with our low-level optimizations. However, it might still be interesting to explore the possibilities for high-level optimizations for key generation, especially considering that the need for parameter sets with larger values of s might arise in the future.

7.2 Additional research topics

Memory and power usage analysis

We have focused on optimizing the performance of the MEDS implementation, but we have not analyzed the memory and power usage of any of the implementations. It would be interesting to see what the impact of our optimizations on memory and power usage is. Additionally, the possibilities for further optimizations in these areas could be explored, which is especially relevant for embedded systems and IoT devices.

Optimizing for other CPU architectures

We have optimized the MEDS implementation for the ARMv8 architecture, but many other CPU architectures support SIMD instructions. One interesting architecture to explore is the AVX512 architecture, which is supported by Intel and AMD CPUs. AVX512 can process 4 times as many field elements in parallel compared to ARMv8, which could result in a significant speedup of the MEDS implementation.

Bibliography

[1] IIS Summer 2023. meds-simd-highlevel. https://github.com/IIS-summer-2023/meds-simd-highlevel, 2023. Accessed: 08-07-2024. 5, 6, 59

[2] IIS Summer 2023. meds-simd-lowlevel. https://github.com/IIS-summer-2023/meds-simd-lowlevel, 2023. Accessed: 08-07-2024. 5, 6

[3] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, and Jean-Pierre Tillich. PERK. https://pqc-perk.org/assets/downloads/PERK_2023_10_16.pdf, October 2023. Accessed: 2024-08-21. 6, 76

[4] Gora Adj, Stefano Barbero, Emanuele Bellini, Andre Esser, Luis Rivera-Zamarripa, Carlo Sanna, Javier Verbel, and Floyd Zweydinger. MiRitH. https://pqc-mirith.org/assets/downloads/mirith_specifications_v1.0.0.pdf, May 2023. Accessed: 2024-08-23. 6

[5] Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vinçotte. RYDE. https://pqc-ryde.org/assets/downloads/ryde_spec.pdf, June 2023. Accessed: 2024-08-21. 6, 76

[6] ARM. ARM Cortex-A Series Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/latest/AArch64-Floating-point-and-NEON. Accessed: 26-03-2024. 5, 19

[7] Arm Limited. Cortex-A72 Software Optimization Guide. https://developer.arm.com/documentation/uan0016/latest/, March 10 2015. Accessed: 12-07-2024. 21, 22, 23

[8] ARM Limited. ARM® Cortex®-A72 MPCore Processor: Technical Reference Manual. https://developer.arm.com/documentation/100095/0003/?lang=en, 2016. Accessed: 08-09-2024. 23

[9] Arm Limited. Arm Neon Intrinsics Reference. https://arm-software.github.io/acle/neon_intrinsics/advsimd.html, August 4 2023. Accessed: 12-07-2024. 19

[10] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS$^+$. https://sphincs.org/data/sphincs+-r3.1-specification.pdf, 2022. Accessed: 2024-08-21. 64, 75, 76

[11] Reza Azarderakhsh, Zhe Liu, Hwajeong Seo, and Howon Kim. NEON PQCryto: Fast and Parallel Ring-LWE Encryption on ARM NEON Architecture. Cryptology ePrint Archive, Paper 2015/1081, 2015. https://eprint.iacr.org/2015/1081. 5

[12] Marco Baldi, Alessandro Barenghi, Luke Beckwith, Jean-François Biasse, Andre Esser, Kris Gaj, Kamyar Mohajerani, Gerardo Pelosi, Edoardo Persichetti, Markku-Juhani O. Saarinen, Paolo Santini, and Robert Wallace. LESS: Linear Equivalence Signature Scheme. https://www.less-project.com/LESS-2024-02-19.pdf, February 2024. Accessed: 2024-08-09. 6, 76

[13] Marco Baldi, Alessandro Barenghi, Sebastian Bitzer, Patrick Karl, Felice Manganiello, Alessio Pavoni, Gerardo Pelosi, Paolo Santini, Jonas Schupp, Freeman Slaughter, Antonia Wachter-Zeh, and Violetta Weger. CROSS: Codes and Restricted Objects Signature Scheme. https://www.cross-crypto.com/CROSS_Specification_v1.2.pdf, February 2024. Accessed: 2024-08-09. 76

[14] Gustavo Banegas, Pierre Karpman, Kévin Carrier, Johanna Loyer, André Chailloux, Ruben Niederhagen, Alain Couvreur, Nicolas Sendrier, Thomas Debris-Alazard, Benjamin Smith, Philippe Gaborit, and Jean-Pierre Tillich. WAVE. https://wave-sign.org/wave_documentation.pdf, June 2023. Accessed: 2024-08-09. 76

[15] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. 28

[16] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019. 26

[17] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. Cryptology ePrint Archive, Paper 2021/986, 2021. https://eprint.iacr.org/2021/986. 6, 32, 76

[18] Hanno Becker and Matthias J. Kannwischer. Hybrid scalar/vector implementations of Keccak and SPHINCS$^+$ on AArch64. Cryptology ePrint Archive, Paper 2022/1243, 2022. https://eprint.iacr.org/2022/1243. 6, 64

[19] Daniel J Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005. 26

[20] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. 5

[21] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical Stateless Hash-Based Signatures. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 368–397, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. 3, 6

[22] Daniel J. Bernstein and Peter Schwabe. NEON Crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, pages 320–339, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 5

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 6, 34

[24] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. UOV: Unbalanced Oil and Vinegar. https://drive.google.com/file/d/1NdMHuCyyFG6xgQGrpssM99kyiNwA9JG-/view, May 2023. Accessed: 2024-08-21. 6, 76

[25] Jean-Francois Biasse, Giacomo Micheli, Edoardo Persichetti, and Paolo Santini. LESS is More: Code-Based Signatures without Syndromes. Cryptology ePrint Archive, Paper 2020/594, 2020. https://eprint.iacr.org/2020/594. 10

[26] Markus Bläser, Dung Hoang Duong, Anand Kumar Narayanan, Thomas Plantard, Youming Qiao, Arnaud Sipasseuth, and Gang Tang. The

ALTEQ Signature Scheme: Algorithm Specifications and Supporting Documentation. https://pqcalteq.github.io/ALTEQ_spec_2024.03.05.pdf, 2024. Accessed: 2024-08-21. 6, 76

[27] Joppe W. Bos, Peter L. Montgomery, Daniel Shumow, and Gregory M. Zaverucha. Montgomery Multiplication Using Vector Instructions. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, pages 471–489, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. 6

[28] Tung Chou, Ruben Niederhagen, Edoardo Persichetti, Lars Ran, Tovohery Hajatiana Randrianarisoa, Krijn Reijnders, Simona Samardjiska, and Monika Trimoska. Matrix Equivalence Digital Signature. https://www.meds-pqc.org/spec/MEDS-2023-07-26.pdf, 2023. 5, 11, 13, 14

[29] Tung Chou, Ruben Niederhagen, Edoardo Persichetti, Tovohery Hajatiana Randrianarisoa, Krijn Reijnders, Simona Samardjiska, and Monika Trimoska. Take Your MEDS: Digital Signatures from Matrix Code Equivalence. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *Progress in Cryptology - AFRICACRYPT 2023*, pages 28–52, Cham, 2023. Springer Nature Switzerland. 3, 9

[30] Tung Chou, Ruben Niederhagen, Lars Ran, and Simona Samardjiska. Reducing signature size of matrix-code-based signature schemes. Cryptology ePrint Archive, Paper 2024/495, 2024. https://eprint.iacr.org/2024/495. 5, 9, 12, 14, 17, 30

[31] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987. 40

[32] Ivan Damgård. On Σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 84, 2002. 12

[33] De Melo, Arnaldo Carvalho. The new linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010. 30, 31

[34] Erik De Win, Serge Mister, Bart Preneel, and Michael Wiener. On the performance of signature schemes based on elliptic curves. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 252–266, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. 4

[35] Jintai Ding and Dieter Schmidt. Rainbow, a New Multivariable Polynomial Signature Scheme. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 164–175, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 6

[36] Leo Ducas, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehle. CRYSTALS – Dilithium: Digital Signatures from Module Lattices. Cryptology ePrint Archive, Paper 2017/633, 2017. https://eprint.iacr.org/2017/633. 3, 6, 32, 75

[37] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. 21, 34

[38] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986. 12

[39] 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. 3, 6, 75

[40] Shafi Goldwasser and Mihir Bellare. Lecture Notes on Cryptography. Chapter 10: Digital signatures, pages 168–169, 2008. 9

[41] Elisa Gorla. Rank-metric codes. In *Concise Encyclopedia of Coding Theory*, pages 227–250. Chapman and Hall/CRC, 2021. 10, 11

[42] Conrado P. L. Gouvêa and Julio López. Implementing GCM on ARMv8. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, pages 167–180, Cham, 2015. Springer International Publishing. 5

[43] Graham, Susan L and Kessler, Peter B and McKusick, Marshall K. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982. 30

[44] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996. 3

[45] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International journal of information security*, 1:36–63, 2001. 4, 5

[46] Youngbeom Kim, Jingyo Song, and Seog Chung Seo. Accelerating Falcon on ARMv8. *IEEE Access*, 10:44446–44460, 2022. 6

[47] Youngbeom Kim, Jingyo Song, Taek-Young Youn, and Seog Chung Seo. CRYSTALS-Dilithium on ARMv8. *Security and Communication Networks*, 2022(1):5226390, 2022. 6

[48] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. 24, 26

[49] Martin Koppl, Dmytro Siroshtan, Milos Orgon, Stefan Pocarovsky, Antonin Bohacik, Karel Kuchar, and Eva Holasova. Performance Comparison of ECDH and ECDSA. In *2021 2nd International Conference on Electronics, Communications and Information Technology (CECIT)*, pages 825–829. IEEE, 2021. 4

[50] Hyeokdong Kwon, Hyunjun Kim, Minjoo Sim, Wai-Kong Lee, and Hwajeong Seo. Look-up the Rainbow: Table-based Implementation of Rainbow Signature on 64-bit ARMv8 Processors. *ACM Trans. Embed. Comput. Syst.*, 22(5), sep 2023. 6

[51] Adam Langley. ImperialViolet: Checking that functions are constant time with Valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html, April 2010. Accessed: 08-07-2024. 27

[52] Arm Limited. Armv8-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0553/latest/, 2015. Accessed: 04-07-2024. 18

[53] Vasileios Mavroeidis, Kamer Vishi, Mateusz D Zych, and Audun Jøsang. The impact of quantum computing on present cryptography. *arXiv preprint arXiv:1804.00200*, 2018. 3

[54] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985. 6, 27

[55] Anand Kumar Narayanan, Youming Qiao, and Gang Tang. Algorithms for matrix code and alternating trilinear form equivalences via new isomorphism invariants. Cryptology ePrint Archive, Paper 2024/368, 2024. https://eprint.iacr.org/2024/368. 15

[56] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007. 26, 30

[57] Duc Tri Nguyen and Kris Gaj. Fast NEON-Based Multiplication for Lattice-Based NIST Post-quantum Cryptography Finalists. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography*, pages 234–254, Cham, 2021. Springer International Publishing. 6

[58] Duc Tri Nguyen and Kris Gaj. Optimized software implementations of CRYSTALS-Kyber, NTRU, and Saber using NEON-based special

instructions of ARMv8. In *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*, 2021. 6

[59] Duc Tri Nguyen and Kris Gaj. Fast Falcon Signature Generation and Verification Using ARMv8 NEON Instructions. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *Progress in Cryptology - AFRICACRYPT 2023*, pages 417–441, Cham, 2023. Springer Nature Switzerland. 6, 76

[60] NIST. Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization, 2017. Accessed: 08-07-2024. 3

[61] Jheyne N Ortiz, Félix Carvalho Rodrigues, Décio Gazzoni Filho, Caio Teixeira, Julio López, and Ricardo Dahab. Evaluation of CRYSTALS-Kyber and Saber on the ARMv8 architecture. In *Anais do XXII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 372–377. SBC, 2022. 6

[62] Krijn Reijnders, Simona Samardjiska, and Monika Trimoska. Hardness estimates of the code equivalence problem in the rank metric. *Designs, Codes and Cryptography*, pages 1–30, 2024. 10

[63] James R. Reinders. Intel® AVX-512 Instructions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html, June 2017. Accessed: 08-07-2024. 6, 56, 79

[64] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. 4, 5

[65] Sand, software and sound. ARM Cortex-A72 Fetch and Branch Processing. https://sandsoftwaresound.net/arm-cortex-a72-fetch-and-branch-processing/, December 2020. Accessed: 02-07-2024. 22

[66] Hwajeong Seo, Zhe Liu, Johann Großschädl, Jongseok Choi, and Howon Kim. Montgomery Modular Multiplication on ARM-NEON Revisited. In Jooyoung Lee and Jongsung Kim, editors, *Information Security and Cryptology - ICISC 2014*, pages 328–342, Cham, 2015. Springer International Publishing. 6

[67] Hwajeong Seo, Zhe Liu, Johann Großschädl, and Howon Kim. Efficient arithmetic on arm-neon and its application for high-speed rsa implementation. *Security and Communication Networks*, 9(18):5401–5411, 2016. 5

[68] Hwajeong Seo, Zhe Liu, Taehwan Park, Hyunjin Kim, Yeoncheol Lee, Jongseok Choi, and Howon Kim. Parallel Implementations of LEA. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology – ICISC 2013*, pages 256–274, Cham, 2014. Springer International Publishing. 5

[69] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994. 3

[70] Minjoo Sim, Siwoo Eum, Hyeokdong Kwon, Hyunjun Kim, and Hwajeong Seo. Optimized Implementation of Encapsulation and Decapsulation of Classic McEliece on ARMv8. Cryptology ePrint Archive, Paper 2022/1706, 2022. https://eprint.iacr.org/2022/1706. 6

[71] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15. IEEE, 2018. 26

[72] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969. 40

[73] Silvan Streit and Fabrizio De Santis. Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple. *IEEE Transactions on Computers*, 67(11):1651–1662, 2018. 6

[74] Ahmed Talal, Mohamed A. Sobh, and Ayman M. Bahaa Eldin. An efficient implementation of RSA for low cost microprocessors. In *2009 4th International Design and Test Workshop (IDT)*, pages 1–4, 2009. 5

[75] Haluk Kent Tanik. ECDSA Optimizations on an ARM Processor for a NIST Curve Over GF(p). 2001. Master's Thesis, Oregon State University. 5

[76] Timecop. Timecop. https://www.post-apocalyptic-crypto.org/timecop/. Accessed: 03-07-2024. 27

[77] J. H. van Lint. *Introduction to Coding Theory*, volume 86 of *Graduate Texts in Mathematics*. Springer, third edition, 1999. 16

[78] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, pages 181–198, Cham, 2015. Springer International Publishing. 5

[79] Bas Westerbaan. ARMv8.4-A implementation for Keccak-f1600. https://github.com/bwesterb/armed-keccak, 2023. Accessed: 2024-08-22. 64

[80] XKCP contributors. XKCP: Extended Keccak Code Package. https://github.com/XKCP/XKCP, 2024. Accessed: 2024-06-14. 64

[81] Sheng-Bo Xu and Lejla Batina. Efficient Implementation of Elliptic Curve Cryptosystems on an ARM7 with Hardware Accelerator. In George I. Davida and Yair Frankel, editors, *Information Security*, pages 266–279, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 5

Appendix A

MEDS Algorithms

A.1 Notations and functions

A.1.1 Notations

In the algorithms in this appendix, we use the following notations in addition to the notations mentioned in Section 2.1:

- ℓ_x: The size of the variable x in bytes.
- $\mathcal{B}^x$: The set of all byte strings of length x.
- σ: A seed used to generate random values.
- $b[i, j]$: The $j - i$ byte long substring of byte string b starting at index i.
- $\mathbf{A}[; i, j]$: The submatrix of $\mathbf{A}$ that starts at column i (inclusive) and ends at column j (exclusive), containing all rows.
- $(x \mid y)$: The concatenation of byte strings x and y.

A.1.2 Functions

In the algorithms in this appendix, we use the following functions (in order of appearance):

- Randombytes(x): Generates a random byte string of length x.
- ExpandSysMat(σ): Generate a random systematic matrix from seed σ.
- XOF(σ, x, y): Generates two random byte strings of length x and y from seed σ.
- ExpandInvMat(σ, k): Generates a random invertible matrix of size $k \times k$ from seed σ.

- Solve($\mathbf{G}$): Computes an isometry mapping $\phi = (\mathbf{A}, \mathbf{B})$ from the (first) two codewords represented by the rows in $\mathbf{G}$.
- SF($\mathbf{G}$): Converts matrix $\mathbf{G}$ to systematic form.
- Compress(G)($\mathbf{A}$): Compresses matrix $\mathbf{A}$ into a byte string.
- Decompress(G)(x): Decompresses byte string x into a matrix.
- SeedTree$_t(\rho, \alpha)$: Constructs a seed tree of height $\log_2(t)$ from root seed ρ and salt α and return the first t leaf nodes.
- ToBytes(x, y): Converts x to a byte string of length y.
- ExpandRndMat(σ): Generates a random matrix from seed σ.
- H(x): Hashes byte string x.
- ParseHash$_{s,t,w}(d)$: Parses hash d into t challenges that are smaller than s each, where w challenges are 0.
- SeedTreeToPath$_t(h_0, \ldots, h_{t-1}, \rho, \alpha)$: Reconstructs a seed tree with the same structure as SeedTree$_t(\rho, \alpha)$. Based on the values of $h_0, \ldots, h_{t-1}$, returns a seed-tree path with which the leaf node seeds can be reconstructed.
- ParseSig(m_s): Parses the signed message m_s into its components.
- PathToSeedTree$_t(h_0, \ldots, h_{t-1}, p, \alpha)$: Reconstructs a seed tree with the same structure as SeedTree$_t(\rho, \alpha)$. Based on the values of $h_0, \ldots, h_{t-1}$ and the path p, returns the seeds of the leaf nodes.

A.2 Main algorithms

Algorithm A.1 MEDS.KeyGen()

Input: -

Output: public key $\mathbf{pk} \in \mathcal{B}^{\ell_{\mathbf{pk}}}$, secret key $\mathbf{sk} \in \mathcal{B}^{\ell_{\mathbf{sk}}}$

1: $\delta \in \mathcal{B}^{\ell_{\text{sec\_seed}}} \leftarrow \text{Randombytes}(\ell_{\text{sec\_seed}})$
2: $\sigma_{\mathbf{G}_0} \in \mathcal{B}^{\ell_{\text{pub\_seed}}}, \sigma \in \mathcal{B}^{\ell_{\text{sec\_seed}}} \leftarrow \text{XOF}(\delta, \ell_{\text{pub\_seed}}, \ell_{\text{sec\_seed}})$
3: $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn} \leftarrow \text{ExpandSysMat}(\sigma_{\mathbf{G}_0})$
4: **for all** $i \in \{1, \ldots, s-1\}$ **do**
5: $\quad \sigma_{\mathbf{T}_i}, \sigma \in \mathcal{B}^{\ell_{\text{sec\_seed}}} \leftarrow \text{XOF}(\sigma, \ell_{\text{sec\_seed}}, \ell_{\text{sec\_seed}})$
6: $\quad \mathbf{T}_i \in \text{GL}_k(q) \leftarrow \text{ExpandInvMat}(\sigma_{\mathbf{T}_i}, k)$
7: $\quad \mathbf{G}'_0 \in \mathbb{F}_q^{k \times mn} \leftarrow \mathbf{T}_i \mathbf{G}_0$
8: $\quad \check{\mathbf{A}}_i \in \mathbb{F}_q^{m \times m} \cup \{\perp\}, \check{\mathbf{B}}_i \in \mathbb{F}_q^{n \times n} \cup \{\perp\} \leftarrow \text{Solve}(\mathbf{G}'_0)$
9: $\quad$ **if** $(\check{\mathbf{A}}_i = \perp$ **and** $\check{\mathbf{B}}_i = \perp)$ **or** $\check{\mathbf{A}}_i \notin \text{GL}_m(q)$ **or** $\check{\mathbf{B}}_i \notin \text{GL}_n(q)$ **then**
10: $\quad\quad$ **goto** line 5
11: $\quad \mathbf{A}_i, \mathbf{A}_i^{-1} \in \text{GL}_m(q) \leftarrow \check{\mathbf{A}}_i, \check{\mathbf{A}}_i^{-1}$
12: $\quad \mathbf{B}_i, \mathbf{B}_i^{-1} \in \text{GL}_n(q) \leftarrow \check{\mathbf{B}}_i, \check{\mathbf{B}}_i^{-1}$
13: $\quad \mathbf{G}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \pi_{\mathbf{A}_i, \mathbf{B}_i}(\mathbf{G}_0)$
14: $\quad \mathbf{T}_i^{-1} \in \mathbb{F}_q^{k \times k} \leftarrow \mathbf{G}_i[; 0, k-1]$
15: $\quad \mathbf{G}_i \in \mathbb{F}_q^{k \times mn} \cup \{\perp\} \leftarrow \text{SF}(\mathbf{G}_i)$
16: $\quad$ **if** $\mathbf{G}_i = \perp$ **then**
17: $\quad\quad$ **goto** line 5
18: $\text{pk} \in \mathcal{B}^{\ell_{\mathbf{pk}}} \leftarrow (\sigma_{\mathbf{G}_0} \mid \text{CompressG}(\mathbf{G}_1) \mid \ldots \mid \text{CompressG}(\mathbf{G}_{s-1}))$
19: $\text{sk} \in \mathcal{B}^{\ell_{\mathbf{sk}}} \leftarrow (\delta \mid \sigma_{\mathbf{G}_0} \mid \text{Compress}(\mathbf{A}_1^{-1}) \mid \ldots \mid \text{Compress}(\mathbf{A}_{s-1}^{-1})$
20: $\qquad\qquad \mid \text{Compress}(\mathbf{B}_1^{-1}) \mid \ldots \mid \text{Compress}(\mathbf{B}_{s-1}^{-1})$
21: $\qquad\qquad \mid \text{Compress}(\mathbf{T}_1^{-1}) \mid \ldots \mid \text{Compress}(\mathbf{T}_{s-1}^{-1}))$
22: **return** pk, sk

Algorithm A.2 MEDS.Sign()

Input: secret key $\mathbf{sk} \in \mathcal{B}^{\ell_{\mathbf{sk}}}$, message $m \in \mathcal{B}^{\ell_m}$
Output: signed message $m_s \in \mathcal{B}^{\ell_{\text{sig}}+\ell_m}$

1: $f_{\text{sk}} \leftarrow \ell_{\text{sec\_seed}}$
2: $\sigma_{\mathbf{G}_0} \leftarrow \text{pk}[f_{\text{sk}}, f_{\text{sk}} + \ell_{\text{pub\_seed}} - 1]$
3: $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn} \leftarrow \text{ExpandSysMat}(\sigma_{\mathbf{G}_0})$
4: $f_{\text{sk}} \leftarrow f_{\text{sk}} + \ell_{\text{pub\_seed}} + (s-1) \cdot (\ell_{\mathbb{F}_q^{m \times m}} + \ell_{\mathbb{F}_q^{n \times n}})$
5: **for all** $i \in \{1, \ldots, s-1\}$ **do**
6: $\quad \mathbf{T}_i^{-1} \in \mathbb{F}_q^{k \times k} \leftarrow \text{Decompress}(\text{sk}[f_{\text{sk}}, f_{\text{sk}} + \ell_{\mathbb{F}_q^{k \times k}}])$
7: $\quad f_{\text{sk}} \leftarrow f_{\text{sk}} + \ell_{\mathbb{F}_q^{k \times k}}$
8: $\delta \in \mathcal{B}^{\ell_{\text{sec\_seed}}} \leftarrow \text{Randombytes}(\ell_{\text{sec\_seed}})$
9: $\rho \in \mathcal{B}^{\ell_{\text{tree\_seed}}}, \alpha \in \mathcal{B}^{\ell_{\text{salt}}} \leftarrow \text{XOF}(\delta, \ell_{\text{tree\_seed}}, \ell_{\text{salt}})$
10: $\sigma_0, \ldots, \sigma_{t-1} \in \mathcal{B}^{\ell_{\text{tree\_seed}}} \leftarrow \text{SeedTree}_t(\rho, \alpha)$
11: **for all** $i \in \{0, \ldots, t-1\}$ **do**
12: $\quad \sigma'_i \in \mathcal{B}^{\ell_{\text{salt}}+\ell_{\text{tree\_seed}}+4} \leftarrow (\alpha \mid \sigma_i \mid \text{ToBytes}(2^{1+\lceil \log_2(t) \rceil + i}, 4))$
13: $\quad \sigma_{\tilde{\mathbf{M}}_i} \in \mathcal{B}^{\ell_{\text{pub\_seed}}}, \sigma_i \in \mathcal{B}^{\ell_{\text{tree\_seed}}} \leftarrow \text{XOF}(\sigma'_i, \ell_{\text{pub\_seed}}, \ell_{\text{tree\_seed}})$
14: $\quad \tilde{\mathbf{M}}_i \in \mathbb{F}_q^{2 \times k} \leftarrow \text{ExpandRndMat}(\sigma_{\tilde{\mathbf{M}}_i})$
15: $\quad \mathbf{C} \in \mathbb{F}_q^{2 \times mn} \leftarrow \tilde{\mathbf{M}}_i \mathbf{G}_0$
16: $\quad \widetilde{\mathbf{A}}_i \in \mathbb{F}_q^{m \times m} \cup \{\perp\}, \widetilde{\mathbf{B}}_i \in \mathbb{F}_q^{n \times n} \cup \{\perp\} \leftarrow \text{Solve}(\mathbf{C})$
17: $\quad$ **if** $(\widetilde{\mathbf{A}}_i = \perp$ **and** $\widetilde{\mathbf{B}}_i = \perp)$ **or** $\widetilde{\mathbf{A}}_i \notin \text{GL}_m(q)$ **or** $\widetilde{\mathbf{B}}_i \notin \text{GL}_n(q)$ **then**
18: $\quad\quad$ **goto** line 12
19: $\quad \tilde{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \pi_{\widetilde{\mathbf{A}}_i, \widetilde{\mathbf{B}}_i}(\mathbf{G}_0)$
20: $\quad \tilde{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \cup \{\perp\} \leftarrow \text{SF}(\tilde{\mathbf{G}}_i)$
21: $\quad$ **if** $\tilde{\mathbf{G}}_i = \perp$ **then**
22: $\quad\quad$ **goto** line 12
23: $d \in \mathcal{B}^{\ell_{\text{digest}}} \leftarrow \text{H}(\text{Compress}(\tilde{\mathbf{G}}_0[; k, mn-1]) \mid \ldots$
24: $\qquad\qquad \mid \text{Compress}(\tilde{\mathbf{G}}_{t-1}[; k, mn-1]) \mid m)$
25: $h_0, \ldots, h_{t-1} \in \{0, \ldots, s-1\} \leftarrow \text{ParseHash}_{s,t,w}(d)$
26: **for all** $i \in \{0, \ldots, t-1\}$ **do**
27: $\quad$ **if** $h_i > 0$ **then**
28: $\quad\quad \kappa_i \in \mathbb{F}_q^{2 \times k} \leftarrow \tilde{\mathbf{M}}_i T_{h_i}^{-1}$
29: $p \in \mathcal{B}^{\ell_{\text{path}}} \leftarrow \text{SeedTreeToPath}_t(h_0, \ldots, h_{t-1}, \rho, \alpha)$
30: **return** $m_s \in \mathcal{B}^{w \cdot \ell_{\mathbb{F}_q^{2 \times k}} + \ell_{\text{path}} + \ell_{\text{digest}} + \ell_{\text{salt}} + \ell_{\text{m}} = \ell_{\text{sig}} + \ell_{\text{m}}}$
31: $\qquad = (\kappa_0 \mid \ldots \mid \kappa_{t-1} \mid p \mid d \mid \alpha \mid m)$

Algorithm A.3 MEDS.Verify()

Input: public key $\mathbf{pk} \in \mathcal{B}^{\ell_{\mathbf{pk}}}$, signed message $m_s \in \mathcal{B}^{\ell_{\text{sig}}+\ell_m}$
Output: message $m \in \mathcal{B}^{\ell_m}$ or $\perp$

1: $\sigma_{\mathbf{G}_0} \leftarrow \text{pk}[0, \ell_{\text{pub\_seed}} - 1]$
2: $\mathbf{G}_0 \in \mathbb{F}_q^{k \times mn} \leftarrow \text{ExpandSysMat}(\sigma_{\mathbf{G}_0})$
3: $f_{\text{pk}} \leftarrow \ell_{\text{pub\_seed}}$
4: **for all** $i \in \{1, \ldots, s-1\}$ **do**
5: $\quad \mathbf{G}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \text{DecompressG}(\text{pk}[f_{\text{pk}}, f_{\text{pk}} + \ell_{\mathbb{F}_q^{k \times mn}}])$
6: $\quad f_{\text{pk}} \leftarrow f_{\text{pk}} + \ell_{G_i}$
7: $p \in \mathcal{B}^{\ell_{\text{path}}}, \alpha \in \mathcal{B}^{\ell_{\text{salt}}}, d \in \mathcal{B}^{\ell_{\text{digest}}}, m \in \mathcal{B}^{\ell_m} \leftarrow \text{ParseSig}(m_s)$
8: $\sigma_0, \ldots, \sigma_{t-1} \in \mathcal{B}^{\ell_{\text{tree\_seed}}} \leftarrow \text{PathToSeedTree}_t(h_0, \ldots, h_{t-1}, p, \alpha)$
9: **for all** $i \in \{0, \ldots, t-1\}$ **do**
10: $\quad$ **if** $h_i > 0$ **then**
11: $\quad\quad \kappa_i \in \mathbb{F}_q^{2 \times k} \leftarrow m_s[i \cdot \ell_{\mathbb{F}_q^{2 \times k}}, (i+1) \cdot \ell_{\mathbb{F}_q^{2 \times k}} - 1]$
12: $\quad\quad \mathbf{G}'_0 \in \mathbb{F}_q^{2 \times mn} \leftarrow \kappa_i \mathbf{G}_{h_i}$
13: $\quad\quad \hat{\mathbf{A}}_i \in \mathbb{F}_q^{m \times m} \cup \{\perp\}, \hat{\mathbf{B}}_i \in \mathbb{F}_q^{n \times n} \cup \{\perp\} \leftarrow \text{Solve}(\mathbf{G}'_0)$
14: $\quad\quad$ **if** $(\hat{\mathbf{A}}_i = \perp$ **and** $\hat{\mathbf{B}}_i = \perp)$**or** $\hat{\mathbf{A}}_i \notin \text{GL}_m(q)$ **or** $\hat{\mathbf{B}}_i \notin \text{GL}_n(q)$ **then**
15: $\quad\quad\quad$ **return** $\perp$
16: $\quad\quad \hat{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \pi_{\hat{\mathbf{A}}_i, \hat{\mathbf{B}}_i}(\mathbf{G}_{h_i})$
17: $\quad\quad \hat{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \cup \{\perp\} \leftarrow \text{SF}(\hat{\mathbf{G}}_i)$
18: $\quad\quad$ **if** $\hat{\mathbf{G}}_i = \perp$ **then**
19: $\quad\quad\quad$ **return** $\perp$
20: $\quad$ **else**
21: $\quad\quad \sigma'_i \in \mathcal{B}^{\ell_{\text{salt}}+\ell_{\text{tree\_seed}}+4} \leftarrow (\alpha \mid \sigma_i \mid \text{ToBytes}(2^{1+\lceil \log_2(t) \rceil + i}, 4))$
22: $\quad\quad \sigma_{\hat{M}_i} \in \mathcal{B}^{\ell_{\text{pub\_seed}}}, \sigma_i \in \mathcal{B}^{\ell_{\text{tree\_seed}}} \leftarrow \text{XOF}(\sigma'_i, \ell_{\text{pub\_seed}}, \ell_{\text{tree\_seed}})$
23: $\quad\quad \hat{\mathbf{M}}_i \in \mathbb{F}_q^{2 \times k} \leftarrow \text{ExpandRndMat}(\sigma_{\hat{\mathbf{M}}_i})$
24: $\quad\quad \hat{\mathbf{C}}_i \in \mathbb{F}_q^{2 \times mn} \leftarrow \hat{\mathbf{M}}_i \mathbf{G}_0$
25: $\quad\quad \hat{\mathbf{A}}_i \in \mathbb{F}_q^{m \times m} \cup \{\perp\}, \hat{\mathbf{B}}_i \in \mathbb{F}_q^{n \times n} \cup \{\perp\} \leftarrow \text{Solve}(\hat{\mathbf{C}}_i)$
26: $\quad\quad$ **if** $(\hat{\mathbf{A}}_i = \perp$ **and** $\hat{\mathbf{B}}_i = \perp)$**or** $\hat{\mathbf{A}}_i \notin \text{GL}_m(q)$ **or** $\hat{\mathrm{B}}_i \notin \text{GL}_n(q)$ **then**
27: $\quad\quad\quad$ **goto** line 21
28: $\quad\quad \hat{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \leftarrow \pi_{\hat{\mathbf{A}}_i, \hat{\mathbf{B}}_i}(\mathbf{G}_0)$
29: $\quad\quad \hat{\mathbf{G}}_i \in \mathbb{F}_q^{k \times mn} \cup \{\perp\} \leftarrow \text{SF}(\hat{\mathbf{G}}_i)$
30: $\quad\quad$ **if** $\hat{\mathbf{G}}_i = \perp$ **then**
31: $\quad\quad\quad$ **goto** line 21
32: $d' \in \mathcal{B}^{\ell_{\text{digest}}} \leftarrow \text{H}(\text{Compress}(\hat{\mathbf{G}}_0[; k, mn-1]) \mid \ldots$
33: $\quad\quad\quad\quad \mid \text{Compress}(\hat{\mathbf{G}}_{t-1}[; k, mn-1]) \mid m)$
34: **if** $d = d'$ **then**
35: $\quad$ **return** m
36: **else**
37: $\quad$ **return** $\perp$

A.3 Supplemental algorithms

The MEDS key generation, signing, and verification algorithms require some additional algorithms to function. In this section, we present a few algorithms that are used by the three MEDS algorithms. Note that we do not list all supplemental algorithms, but only those that are relevant to our research.

Algorithm A.4 MEDS matrix multiplication

function MATRIX_MUL($A \in \mathbb{F}_{4093}^{m \times n}, B \in \mathbb{F}_{4093}^{n \times o}$)
  $C \leftarrow$ zero matrix of size $m \times o$
  for $c \leftarrow 0$ to m **do**
    for $r \leftarrow 0$ to o **do**
      for $k \leftarrow 0$ to n **do**
        $C[c][r] \leftarrow C[c][r] + A[c][k] \cdot B[k][r]$
      $C[c][r] \leftarrow C[c][r] \mod 4093$
  return C

Algorithm A.5 MEDS 'pi' function $\pi_{\mathbf{A},\mathbf{B}}(\mathbf{G})$

function PI($\mathbf{A} \in \mathbb{F}_{4093}^{m \times m}, \mathbf{B} \in \mathbb{F}_{4093}^{n \times n}, \mathbf{G} \in \mathbb{F}_{4093}^{k \times mn}$)
  $G' \leftarrow$ matrix (array) of size $k \times mn$
  for $i \leftarrow 0$ to k **do**
    $G'[i \cdot mn, (i+1) \cdot mn] \leftarrow$ matrix_mul($\mathbf{A}, \mathbf{G}[i \cdot mn : (i+1) \cdot mn]$)
    $G'[i \cdot mn, (i+1) \cdot mn] \leftarrow$ matrix_mul($\mathbf{G}[i \cdot mn : (i+1) \cdot mn], \mathbf{B}$)
  return G'

Algorithm A.6 MEDS 'SF' function

function SF($\mathbf{G} \in \mathbb{F}_{4093}^{k \times mn}$)
  $M \leftarrow$ matrix of size $k \times k$
  for $r \leftarrow 0$ to k **do**
    $M[r \cdot k, (r+1) \cdot k] \leftarrow \mathbf{G}[r \cdot mn, r \cdot mn + k]$
  $M^{-1} \leftarrow$ mat_inv(M)
  if $M^{-1} = \perp$ **then**
    return $\perp$
  return matrix_mul($M^{-1}, \mathbf{G}$)

Algorithm A.7 MEDS matrix systemizer

function SYSTEMIZE($A \in \mathbb{F}_{4093}^{m \times n}$, $r_{\max}$, do_swap, do_backsub)
  $ret \leftarrow m \cdot$ do_swap
  for $r \leftarrow 0$ to $r_{\max}$ **do**
    `// Attempt to make the diagonal element non-zero`
    if do_swap **then**
      $z \leftarrow 0$
      for $r_2 \leftarrow r$ to m **do**
        $z \leftarrow z$ or $A[r_2][r]$
      if $z = 0$ **then**
        $ret \leftarrow r$
        for $i \leftarrow 0$ to r **do**
          $A[i][r], A[i][n-1] \leftarrow A[i][n-1], A[i][r]$
    for $r_2 \leftarrow r+1$ to m **do**
      if $A[r][r] = 0$ **then**
        for $c \leftarrow r$ to n **do**
          $A[r][c] \leftarrow (A[r][c] + A[r_2][c]) \mod 4093$
    if $A[r][r] = 0$ **then**
      return -1
    `// Normalize row r such that A[r][r] = 1`
    $v \leftarrow \text{GF\_inv}(A[r][r])$
    for $c \leftarrow r$ to n **do**
      $A[r][c] \leftarrow (A[r][c] \cdot v) \mod 4093$
    `// Eliminate A[r2][r] for r2 > r`
    for $r_2 \leftarrow r+1$ to m **do**
      for $c \leftarrow r$ to n **do**
        $v \leftarrow (A[r][c] \cdot A[r_2][r]) \mod 4093$
        $A[r2][c] \leftarrow ((A[r2][c] - \text{v}) + 4093) \mod 4093$
  `// Return if we do not need to do back substitution`
  if !do_backsub **then**
    return ret
  `// Perform back substitution`
  for $r \leftarrow r_{\max} - 1$ to 0 **do**
    for $r_2 \leftarrow 0$ to r **do**
      $v \leftarrow (A[r][r] \cdot A[r_2][r]) \mod 4093$
      $A[r_2][r] \leftarrow ((A[r_2][r] - \text{v}) + 4093) \mod 4093$
      for $c \leftarrow r_{\max}$ to n **do**
        $v \leftarrow (A[r][c] \cdot A[r_2][r]) \mod 4093$
        $A[r_2][c] \leftarrow ((A[r_2][c] - \text{v}) + 4093) \mod 4093$
  return ret

Appendix B

Benchmark Results

In this appendix, we present the benchmarking results for all implementations of MEDS on two different platforms: the ARM Cortex-A72 and the Apple M2. We benchmark each of the three parameter sets that we consider: MEDS-21595, MEDS-55520, and MEDS-122000. For each benchmark, we consider the reference, low-level optimized, and high-level optimized implementations. For each implementation, we also consider the optimized hash structure variant described in Section 4.5.

The ARM Cortex-A72 measurements are taken on a Raspberry Pi 4 Model B with the CPU running at 1.5 GHz. Frequency scaling is disabled to ensure consistent results. The results are shown in Tables B.1, B.2, and B.3.

The Apple M2 measurements are taken on a Mac Mini (2023). All benchmarks are run on one of the four performance cores, clocked at 3.49 GHz. The results are shown in Tables B.4, B.5, and B.6.

Table B.1: MEDS-21595 benchmarking results on the ARM Cortex-A72 for all implementations (reference and optimized variants). The values represent the number of megacycles required to execute that algorithm. 'New Hash Structure' refers to the optimized hash structure described in Section 4.5.

Variant	Keygen	Sign	Verify
Reference	8.0 (×1.0)	890.7 (×1.0)	889.8 (×1.0)
Reference (New Hash Structure)	8.0 (×1.0)	892.0 (×1.0)	889.5 (×1.0)
Low-Level	2.8 (×2.9)	248.2 (×3.6)	229.8 (×3.9)
Low-Level (New Hash Structure)	2.8 (×2.9)	235.5 (×3.8)	217.1 (×4.1)
High-Level	8.0 (×1.0)	255.4 (×3.5)	252.2 (×3.5)
High-Level (New Hash Structure)	8.0 (×1.0)	242.2 (×3.7)	237.6 (×3.7)

Table B.2: MEDS-55520 benchmarking results on the ARM Cortex-A72 for all implementations (reference and optimized variants). The values represent the number of megacycles required to execute that algorithm. 'New Hash Structure' refers to the optimized hash structure described in Section 4.5.

Variant	Keygen	Sign	Verify
Reference	22.7 (×1.0)	3623.8 (×1.0)	3628.2 (×1.0)
Reference (New Hash Structure)	22.6 (×1.0)	3622.2 (×1.0)	3633.0 (×1.0)
Low-Level	7.2 (×3.2)	927.9 (×3.9)	878.3 (×4.1)
Low-Level (New Hash Structure)	7.2 (×3.2)	880.4 (×4.1)	832.1 (×4.4)
High-Level	22.6 (×1.0)	1166.9 (×3.1)	1163.4 (×3.1)
High-Level (New Hash Structure)	22.6 (×1.0)	1120.7 (×3.2)	1116.8 (×3.2)

Table B.3: MEDS-122000 benchmarking results on the ARM Cortex-A72 for all implementations (reference and optimized variants). The values represent the number of megacycles required to execute that algorithm. 'New Hash Structure' refers to the optimized hash structure described in Section 4.5.

Variant	Keygen	Sign	Verify
Reference	71.0 (×1.0)	13748.9 (×1.0)	13731.4 (×1.0)
Reference (New Hash Structure)	71.0 (×1.0)	13749.6 (×1.0)	13723.3 (×1.0)
Low-Level	16.6 (×4.3)	2794.5 (×4.9)	2693.8 (×5.1)
Low-Level (New Hash Structure)	16.7 (×4.3)	2653.6 (×5.2)	2550.3 (×5.4)
High-Level	71.2 (×1.0)	3820.6 (×3.6)	4130.7 (×3.3)
High-Level (New Hash Structure)	71.2 (×1.0)	3672.1 (×3.7)	3991.3 (×3.4)

Table B.4: MEDS-21595 benchmarking results on the Apple M2 for all implementations (reference and optimized variants). The values represent the number of megacycles required to execute that algorithm. 'New Hash Structure' refers to the optimized hash structure described in Section 4.5.

Variant	Keygen	Sign	Verify
Reference	3.0 (×1.0)	345.6 (×1.0)	345.5 (×1.0)
Reference (New Hash Structure)	3.0 (×1.0)	352.7 (×1.0)	352.6 (×1.0)
Low-Level	1.0 (×3.0)	93.9 (×3.7)	84.6 (×4.1)
Low-Level (New Hash Structure)	1.0 (×3.0)	72.3 (×4.8)	62.7 (×5.5)
High-Level	3.0 (×1.0)	89.8 (×3.8)	93.3 (×3.7)
High-Level (New Hash Structure)	3.0 (×1.0)	67.9 (×5.1)	71.1 (×4.9)

Table B.5: MEDS-55520 benchmarking results the Apple M2 for all implementations (reference and optimized variants). The values represent the number of megacycles required to execute that algorithm. 'New Hash Structure' refers to the optimized hash structure described in Section 4.5.

Variant	Keygen	Sign	Verify
Reference	8.8 (×1.0)	1469.8 (×1.0)	1469.0 (×1.0)
Reference (New Hash Structure)	8.8 (×1.0)	1469.8 (×1.0)	1469.5 (×1.0)
Low-Level	2.3 (×3.8)	335.3 (×4.4)	314.8 (×4.7)
Low-Level (New Hash Structure)	2.4 (×3.7)	255.7 (×5.7)	234.2 (×6.3)
High-Level	8.8 (×1.0)	372.2 (×3.9)	386.8 (×3.8)
High-Level (New Hash Structure)	8.8 (×1.0)	291.8 (×5.0)	309.3 (×4.7)

Table B.6: MEDS-122000 benchmarking results the Apple M2 for all implementations (reference and optimized variants). The values represent the number of megacycles required to execute that algorithm. 'New Hash Structure' refers to the optimized hash structure described in Section 4.5.

Variant	Keygen	Sign	Verify
Reference	22.6 (×1.0)	4877.7 (×1.0)	4878.4 (×1.0)
Reference (New Hash Structure)	22.6 (×1.0)	4970.7 (×1.0)	4876.4 (×1.0)
Low-Level	5.1 (×4.4)	948.6 (×5.1)	899.4 (×5.4)
Low-Level (New Hash Structure)	5.1 (×4.4)	719.9 (×6.8)	671.6 (×7.3)
High-Level	22.6 (×1.0)	1193.6 (×4.1)	1247.8 (×3.9)
High-Level (New Hash Structure)	22.6 (×1.0)	963.1 (×5.1)	10160.3 (×0.5)