Michael Helmut Streibl, BEd BSc

Design and Analysis of the MAYO Signature Scheme with Focus on Hardware Platforms

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science (UF 066 921)

submitted to

Graz University of Technology

Supervisors

Sujoy Sinha Roy, Ass.Prof. PhD

Ahmet Can Mert, Lis. PhD Y.Lis.

Florian Hirner, Dipl.-Ing. BSc

Institute of Applied Information Processing and Communications

Inffeldgasse 16a, 8010 Graz, Austria

Graz, March 2024

Acknowledgements

I would like to express my sincere gratitude to all those who have supported and guided me throughout the journey of completing this thesis. First and foremost, I extend my deepest appreciation to my supervisors Sujoy Sinha Roy, Ahmet Can Mert, and Florian Hirner for their invaluable guidance and continuous assistance during the research and writing process. Their expertise and constructive feedback have been instrumental in shaping this thesis. Working with you was an exceptional experience and being involved in the writing of a paper is certainly one of the highlights in my academic career.

I am indebted to my family for their unwavering love, encouragement, and support throughout this academic and personal journey. Your belief in me has been a constant source of motivation and strength and I would not have been able to get to this point without you.

Finally, I would like to express my heartfelt appreciation to all my friends and colleagues who have shared this path with me. My way through university is filled with lasting memories and I will always look back on this time with joy.

This thesis would not have been possible without the support and contributions of all those mentioned above and many more. Thank you all for being part of this journey.

Abstract

Research in post-quantum cryptography attracts great interest due to rapid progress in the development of quantum computers. The initial NIST call regarding post-quantum cryptography led to the standardization of the first quantum-safe cryptographic algorithms. However, the majority of these new schemes are based on lattice constructions, prompting NIST to issue a new call to expand their portfolio with schemes founded on non-lattice problems. One of the new candidates is MAYO, which falls under the multivariate-based cryptography category and is a modification of the Unbalanced Oil and Vinegar (UOV) scheme. In general, UOV offers short signatures and fast verification, but has the disadvantage of large public keys. MAYO addresses this drawback by introducing a whipping technique to substantially reduce the size of the public key, making it a promising candidate of the current NIST call.

This thesis thoroughly analyzes the MAYO scheme and presents multiple approaches for an efficient hardware design that can be used for both, low-area and high-performance implementations. The first major design approach is the introduction of on-the-fly generation for pseudo-random data, leading to a substantial reduction of on-chip memory. We propose a memory design that enables the parallelization of the matrix computations within the MAYO scheme to increase the performance of a hardware implementation. Additionally, we examine the performance effects of different pseudo-random number generators on hardware and software platforms. While the utilization of SHAKE128 enhances the scheme's performance on FPGAs, a decrease on modern CPUs is observed. Consequently, we present a modification of the original MAYO scheme to parallelize the generation of pseudo-random data. Comparisons show that a hardware implementation which follows the presented design approaches significantly outperforms previous works by one to three orders of magnitude, while simultaneously reducing memory consumption by 30 to 50 %. Furthermore, most of the presented design techniques can be applied to UOV schemes in general, thereby extending their applications to a wide range of post-quantum cryptographic schemes.

Keywords: MAYO, Unbalanced Oil and Vinegar, PQC, Multivariate Cryptography, FPGA, Digital Signatures

Kurzfassung

Die Forschung im Bereich der Post-Quanten-Kryptographie stößt aufgrund der schnellen Fortschritte bei der Entwicklung von Quantencomputern auf großes Interesse. Der erste NIST-Aufruf zur Post-Quanten-Kryptographie führte zur Standardisierung der ersten quantensicheren kryptographischen Algorithmen. Die meisten dieser neuen Verfahren basieren jedoch auf Gitterkonstruktionen, was das NIST dazu veranlasste einen neuen Aufruf zur Erweiterung seines Portfolios zu starten. Ziel dieses Aufrufes ist es Verfahren zu standardisieren, die nicht auf Gitterproblemen basieren. Einer der neuen Kandidaten ist MAYO, der in die Kategorie der multivariaten Kryptographie fällt und eine Abwandlung des "Unbalanced Oil and Vinegar" (UOV) Schemas ist. Im Allgemeinen bietet UOV kurze Signaturen und eine schnelle Verifizierung, leidet aber unter großen öffentlichen Schlüsseln. MAYO behebt diesen Nachteil durch die Einführung einer neuartigen "Whipping"-Technik, mit der die Schlüsselgröße erheblich reduziert werden kann, was es zu einem vielversprechenden Kandidaten für die aktuelle NIST-Ausschreibung macht.

In dieser Arbeit wird das MAYO-Verfahren gründlich analysiert und es werden mehrere Herangehensweisen für ein effizientes Hardwaredesign vorgestellt, die sowohl für Implementierungen mit geringem Ressourcenbedarf als auch für Hochleistungsimplementierungen verwendet werden können. Der erste wichtige Entwurfsansatz ist die "on-the-fly" Generierung von Pseudo-Zufallsdaten, was zu einer erheblichen Reduzierung des On-Chip-Speichers führt. Wir schlagen ein neuartiges Speicherdesign vor, das in der Lage ist die Matrixberechnungen innerhalb des MAYO-Schemas zu parallelisieren, um die Leistung einer Hardwareimplementierung zu steigern. Außerdem untersuchen wir die Leistungseffekte verschiedener Pseudozufallszahlengeneratoren auf Hardware- und Software-Plattformen. Während die Verwendung von SHAKE128 die Leistung des Schemas auf FPGAs steigert, wird auf modernen CPUs ein Leistungsabfall beobachtet. Folglich präsentieren wir eine Modifikation des ursprünglichen MAYO-Schemas zur Parallelisierung der Pseudo-Zufallsdatengenerierung. Vergleiche zeigen, dass eine Hardware-Implementierung, die den vorgestellten Designansätzen folgt, frühere Arbeiten um eine bis drei Größenordnungen übertrifft, während gleichzeitig der Speicherverbrauch um 30 bis 50 % reduziert wird. Darüber hinaus können die meisten der vorgestellten Entwurfstechniken auch auf UOV-Schemata angewendet werden, was ihre Anwendung auf eine breite Palette von kryptographischen Post-Quanten-Schemata erweitert.

Schlagwörter: MAYO, Unbalanced Oil and Vinegar, PQC, Multivariate Kryptographie, FPGA, Digitale Signaturen

Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard	11
AES-NI	Advanced Encryption Standard New Instructions	11
ALU	arithmetic logic unit	30
AVX	Advanced Vector Extensions	12
BRAM	Block RAM	36
CTR	counter	11
DH	Diffie-Hellmann	1
DSP	Digital Signal Processing	58
FF	flip-flop	58
FPGA	Field Programmable Gate Array	28
IV	initialization vector	52
LUT	look-up table	57
MAC	multiply-and-accumulate	41
MQ	Multivariate Quadratic	8
NIST	National Institute of Standards and Technology	1
PQC	Post-Quantum Cryptography	1
PRNG	pseudorandom number generator	17
SHA	Secure Hash Algorithm	12
SIMD	Single Instruction Multiple Data	12
UOV	Unbalanced Oil and Vinegar	2

Chapter 1.

Introduction

1.1. Motivation

Communication in our digital world heavily depends on public-key cryptography to ensure the authenticity of shared information and to perform key agreements. Commonly employed public-key algorithms rely on problems such as integer factorization or discrete logarithms, which are assumed to be computationally infeasible to solve with current computers. However, the introduction of large-scale quantum computers poses a significant risk to these cryptographic algorithms, as Shor's quantum algorithm [Sho94] can solve the underlying mathematical problems in polynomial time, resulting in the break of the widely adopted RSA and Diffie-Hellmann (DH) schemes. Nevertheless, Shor's algorithm is not the only quantum algorithm endangering current cryptographic schemes, since Grover's algorithm [Gro96] enhances the efficiency of key and preimage search for current ciphers and hash functions. While its impact is not as severe as in the case of Shor's algorithm, Grover still needs to be considered for the development of future cryptographic schemes. Recent years have witnessed rapid progress in quantum computing, with notable developments such as Google's Bristlecone (72 qubits) in 2018 or IBM's Condor (1121 qubits) in 2023 [Wikc]. In light of this technological progress, there is a growing demand for novel cryptographic algorithms, specifically designed to withstand quantum attacks, in order to facilitate a seamless transition from the current schemes.

This new form of public-key cryptography, often referred to as Post-Quantum Cryptography (PQC), can be classified into five primary classes based on their underlying mathematical problem: lattice-based, hash-based, code-based, isogeny-based, and multivariate-based. Each class exhibits its own set of distinct characteristics, strengths, and limitations, including factors such as public key sizes and performance. In 2016, the US National Institute of Standards and Technology (NIST) started the "Post-Quantum Cryptography Standardization" process, inviting proposals from researchers and experts in the field to establish a new standard for PQC algorithms. In 2022, following three evaluation rounds, NIST standardized one key-encapsulation mechanism, Crystals-Kyber [Sch+22], and three signature schemes, Crystals-Dilithium [Bai+22], SPHINCS+ [Hul+22], and Falcon [Pre+22]. Notably, SPHINCS+ is the only algorithm categorized under the class of hash-based algorithms, while the other three algorithms are

based on lattice constructions. However, due to insufficient diversity among the selected algorithms, NIST initiated a new call in 2022 [NIS22] asking for additional quantum-safe signature schemes. The submissions regarding this call encompass digital signature algorithms utilizing a wide range of constructions, among them the remaining classes of hash-based, code-based, isogeny-based, and multivariate-based approaches.

One of the ten candidates in the multivariate signature class is MAYO [Beu22b; Beu+23a], a novel post-quantum scheme founded on the Unbalanced Oil and Vinegar (UOV) concept [KPG99]. The UOV algorithm is considered as one of the most comprehensively studied multivariate quadratic signature algorithms. It is based on the the NP-hardness of solving multivariate quadratic systems. However, MAYO is not the first multivariate signature scheme submitted to a NIST standardization process. The Rainbow signature scheme [DS05] advanced to the third round of the initial NIST post-quantum cryptography call. As a result, multivariate signature schemes that are based on the Oil and Vinegar concept have attracted considerable attention, due to their short signatures and fast verification process. However, Rainbow was broken during the third round [Beu22a], resulting in the elimination from the standardization process. The MAYO scheme adapts the original UOV algorithm to address the inherent large key sizes. It employs an oil space considered too small for the original scheme and introduces a special *whipping* technique to facilitate signature sampling. This results in MAYO signatures being more compact than those of the recently standardized Falcon and Dilithium.

To achieve widespread adoption in practical settings, a new signature scheme must demonstrate performance not only on software but also on hardware platforms. NIST mandates that every submitted scheme is optimized for usage with AVX. Such a requirement is significantly more challenging to formulate for hardware implementations, as optimizations heavily rely on the scheme's specifications and the targeted platform. Therefore, implementation techniques and optimizations strategies must be explored for each candidate individually. Due to the novelty of MAYO, publications focusing on its implementation are relatively rare. However, should MAYO emerge as promising candidate of the NIST call, it is essential that efficient hardware designs exist.

1.2. Contribution

Paper: Florian Hirner, Michael Streibl, Ahmet Can Mert, and Sujoy Sinha Roy. *"Whipping the MAYO Signature Scheme using Hardware Platforms"* Submitted for review in IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) 2024. [Hir+23]

Contribution: Contributed to the analysis of MAYO and its mathematical background, the design of the optimization strategies, the implementation of the software modification, and the writing of the paper

The following listing outlines the most notable contributions of this thesis.

1. We give a extensive description of the MAYO scheme and its foundational concept UOV. Given the various forms of UOV and the revisions undergone by MAYO, the information regarding the scheme is scattered across different publications with varying specifications We consolidated the available information to create an extensive description featuring the latest parameters and algorithm descriptions. Additionally, we included essential details to understand the transition from UOV to MAYO, along with the necessary mathematical background to comprehend the scheme's operations.

2. We address the large memory demand of MAYO by introducing an on-the-fly generation design approach. This optimization facilitates the implementation of MAYO on hardware platforms, preventing potential memory constraint issues that could arise from the public key exceeding the available memory. We examine the consequences of dynamically generating data and present solutions for its integration into the scheme. Furthermore, we demonstrate how this technique can be utilized to achieve either a high-performance or a low-area design.

3. We present a novel memory design approach specifically tailored for MAYO. This design facilitates the efficient computation of matrix operations by introducing two distinct memory formats for storing matrix elements. The proposed design is compatible with on-the-fly generation and offers customization to target high-performance or low-area implementations.

4. We introduce a finite field arithmetic unit design that relies exclusively on bit-wise AND and XOR operations. This design keeps the resource utilization of the ALU relatively low, enabling the instantiation of multiple units. This capability facilitates the parallelization of matrix multiplication, resulting in a significant performance increase.

5. We propose a modification of the original MAYO specifications, employing SHAKE as sole PRNG. The resulting software implementation no longer relies on AES-NI for performance enhancement. This revised version exhibits more hardware-friendly characteristics, making it easier to implement on constrained platforms such as FPGAs and microprocessors.

1.3. Related Work

Given the recent publication of MAYO, available implementations are rare. The authors of MAYO included a software implementation in their submission supporting different operation modes. The optimized mode enhances performance by employing AES-NI and AVX2 instructions [Beu+23b]. In addition, the MAYO team released a modified variant for ARM platforms based on a nibble-sliced representation instead of a bitsliced approach [Beu+23c; Beu+23d]. There exists a second work on porting MAYO to the

ARM architecture [Gri+23], however, this implementation does not adhere to the current specifications [Beu+23a] and employs a flawed linear system solving algorithm.

Regarding multivariate-based cryptography in general, the majority of publications targeting FPGA platforms focus on Rainbow [FG18; Tan+11]. However, a comparison with these implementations is obsolete due to the break of Rainbow. HaMAYO [Say+23] is currently the only work specifically aimed at MAYO. Yet, their implementation only supports key generation and signature computation for one parameter set. The scheme most closely related to MAYO is UOV, which serves as the foundation of MAYO. Different variants of UOV were released since its publication in 1995 [Pat97]. Among these variants, [Beu+23e] is particularly noteworthy as it resembles MAYO by employing a similar matrix representation for the public and private key. Additionally, this work discusses implementation techniques for ARM and FPGAs. The similarity between many operations in MAYO and UOV renders this work a excellent candidate for comparison. Moreover, in contrast to HaMayo, it provides a complete implementation of the scheme for hardware platforms.

1.4. Outline

This thesis is organized as follows. **Chapter 2** presents the used notation and provides the necessary background for the MAYO scheme, including discussions on finite field arithmetic, multivariate quadratic systems, and the UOV scheme. **Chapter 3** expands on this background and offers a comprehensive description of MAYO, including its whipping technique, the specified parameter sets, and attacks targeting it. Then, **Chapter 4** introduces several key approaches for hardware designs, among them on-the-fly generation, the memory design and potential techniques for parallelization. **Chapter 5** shifts the focus to software platforms and proposes a modified variant of MAYO that utilizes a multi-seed approach. **Chapter 6** presents hardware and software implementation results and compares them to previous works. Finally, **Chapter 7** provides a summary of our findings and future research opportunities in the area of MAYO and Oil and Vinegar schemes.

Chapter 2.

Background

This chapter provides the background required to comprehend the Unbalanced Oil and Vinegar (UOV) [Beu+23e] and MAYO [Beu22b] schemes. First, a description of the used notation in this thesis is given, followed by a discussion on finite field arithmetic. Subsequently, the third and fourth sections elaborate on fundamental building blocks of UOV and MAYO. The fifth section introduces the security foundation of multivariate cryptography before a high level overview of Oil and Vinegar schemes is given. Section 7 describes Gaussian Elimination in a finite field, which is used to solve related linear systems in multivariate cryptography. In the subsequent sections, we explore how AES and SHAKE can be used to generate pseudo-random data. Finally, the last section offers a brief explanation of the AVX instruction set and the advantages it offers to MAYO. This chapter extents the background provided in [Hir+23].

2.1. Notation

Throughout this thesis finite field arithmetic is often used. A finite field with q elements is denoted by $\mathbb{F}_q$, where q is either a prime number or a power of a prime. Elements of this field are written as lowercase letters (i.e., $a \in \mathbb{F}_q$). A vector with k field elements is represented by $\mathbf{v} \in \mathbb{F}_q^k$. Similarly, a matrix with m rows and n columns is denoted by $\mathbf{M} \in \mathbb{F}_q^{m \times n}$. The i-th row of a matrix is expressed as $\mathbf{M}[i,:]$, while the j-th column is represented as $\mathbf{M}[:,j]$. Consequently, $\mathbf{M}[i,j]$ refers to the matrix element located at row i and column j. Over the course of this thesis, we often switch between polynomials, denoted as lowercase letters (i.e., p) and their matrix counterparts, indicated as the same letter in uppercase bold font. $\wedge$ and $\oplus$ represent the bitwise AND and XOR operation, respectively.

2.2. Finite field arithmetic over $\mathbb{F}_{16}$

The main parts of the MAYO signature arithmetic take place in $\mathbb{F}_{16}$. As mentioned before, q needs to be a prime number or a power of a prime and, thus, $\mathbb{F}_{16}$ is the shorthand notion for $\mathbb{F}_{2^4}$. Elements within this domain can be expressed as a cubic polynomial, for instance, $a = a_3x^3 + a_2x^2 + a_1x + a_0$, where a_3, a_2, a_1, a_0 belong to the field $\mathbb{F}_2$.

Throughout the thesis, we encode elements of $\mathbb{F}_{16}$ as unsigned 4-bit integers, whose bits are the coefficients of the polynomial, such that $\textsc{Encode}(a = a_3x^3 + a_2x^2 + a_1x + a_0) = (a_3a_2a_1a_0)_2$. For instance, $\textsc{Encode}(1x^3 + 0x^2 + 1x + 1) = (1011)_2 = (11)_{10}$.

2.2.1. $\mathbb{F}_{16}$ addition and subtraction

Building on the polynomial representation of $\mathbb{F}_{16}$, the addition and subtraction of two field elements, represented as polynomials $a = a_3x^3 + a_2x^2 + a_1x + a_0$ and $b = b_3x^3 + b_2x^2 + b_1x + b_0$, is defined as standard polynomial addition and subtraction, respectively. Given that the coefficients of the polynomial belong to the field $\mathbb{F}_2$ where addition is equivalent to subtraction, we can utilize a single operation for both. Thus, we implement $\mathbb{F}_{16}$ addition and subtraction as depicted in Equation (2.1).

$$a \pm b = (a_3 \pm b_3)x^3 + (a_2 \pm b_2)x^2 + (a_1 \pm b_1)x + (a_0 \pm b_0) = a \oplus b \tag{2.1}$$

2.2.2. $\mathbb{F}_{16}$ multiplication

As before, multiplication of two field elements $a = a_3x^3 + a_2x^2 + a_1x + a_0$ and $b = b_3x^3 + b_2x^2 + b_1x + b_0$ can be represented similar to its polynomial counterpart. However, in this case a standard multiplication can result in a polynomial c with a degree greater than three, which is not part of the finite field anymore. Therefore, a reduction operation has to be performed to produce a valid $\mathbb{F}_{16}$ element. The result of this reduction operation is defined as the remainder of the Euclidean division of c by an irreducible polynomial p. The MAYO scheme uses $p = x^4 + x + 1$ as the reduction polynomial. The finite field multiplication with this specific reduction polynomial can be implemented using only bitwise AND and XOR operations as shown in Equation (2.2).

$$\begin{aligned}
c =& a \times b = (c_3c_2c_1c_0)_2, \quad \text{where} \\
c_0 =& (a_0 \wedge b_0) \oplus (a_1 \wedge b_3) \oplus (a_2 \wedge b_2) \oplus (a_3 \wedge b_1) \\
c_1 =& (a_0 \wedge b_1) \oplus (a_1 \wedge b_0) \oplus (a_1 \wedge b_3) \oplus (a_2 \wedge b_2) \oplus (a_3 \wedge b_1) \oplus (a_2 \wedge b_3) \oplus (a_3 \wedge b_2) \\
c_2 =& (a_0 \wedge b_2) \oplus (a_1 \wedge b_1) \oplus (a_2 \wedge b_0) \oplus (a_2 \wedge b_3) \oplus (a_3 \wedge b_2) \oplus (a_3 \wedge b_3) \\
c_3 =& (a_0 \wedge b_3) \oplus (a_1 \wedge b_2) \oplus (a_2 \wedge b_1) \oplus (a_3 \wedge b_0) \oplus (a_3 \wedge b_3)
\end{aligned} \tag{2.2}$$

This bitsliced approach allows implementing $\mathbb{F}_{16}$ multiplication in a simple and efficient manner on hardware platforms, due to the the fast bitselection capability compared to software.

2.3. Multivariate Quadratic Maps

The fundamental component of the Oil and Vinegar [KPG99] and the MAYO scheme is the multivariate quadratic map. We adhere to the definition and notation outlined in [Beu22b]. A multivariate quadratic map $P(\mathbf{x}) = (p_1, \ldots, p_m) : \mathbb{F}_q^n \to \mathbb{F}_q^m$ comprises m multivariate quadratic polynomials in n variables with coefficients from $\mathbb{F}_q$. To evaluate this map at a specific value $\mathbf{a} \in \mathbb{F}_q^n$, every polynomial p_i is evaluated at $\mathbf{a}$. Hence, the result of the multivariate quadratic map is specified as $P(\mathbf{a}) = \mathbf{b}$ with $\mathbf{b} = (p_1(\mathbf{a}), \ldots, p_m(\mathbf{a}))$.

UOV as well as MAYO employ homogeneous multivariate maps, where every polynomial is in homogeneous form. In the quadratic case this results in polynomials where every nonzero term is of degree two, also known as quadratic form. One property of the quadratic form is that every polynomial has an associated matrix. Let c_{ij} denote the coefficient of the quadratic term $x_i x_j$. Then, a polynomial can be represented as

$$p(\mathbf{x}) = p(x_1 \ldots x_n) = \sum_{1 \leq i \leq j \leq n} c_{ij} x_i x_j. \tag{2.3}$$

Since $i \leq j$, every term $x_i x_j$ appears only once. This does not restrict the total number of polynomials. Due to commutativity the $x_j x_i$ terms can be expressed by $x_i x_j$. This allows us to set the lower triangular part to 0 and, therefore, Eq. (2.3) can be rewritten into

$$p(\mathbf{x}) = \mathbf{x}^\top \begin{pmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ 0 & c_{22} & \ldots & c_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & c_{nn} \end{pmatrix} \mathbf{x}. \tag{2.4}$$

MAYO adopts the upper triangular matrix form as depicted in Equation (2.4) and defines polynomial evaluation as

$$p_i(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_i \mathbf{x} = \mathbf{x}^\top \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} \mathbf{x}. \tag{2.5}$$

Due to the multivariate quadratic map comprising m different polynomials, we are left with m distinct $\mathbf{P}_i$ matrices that require evaluation.

2.4. Polar Form

To understand how UOV and MAYO sample signatures efficiently the polar form of polynomials is essential. Every homogeneous multivariate quadratic polynomial is linked to a symmetric and bilinear form

$$p'(\mathbf{x}, \mathbf{y}) = p(\mathbf{x} + \mathbf{y}) - p(\mathbf{x}) - p(\mathbf{y}). \tag{2.6}$$

Since the multivariate quadratic maps used in the two signature schemes are homogeneous, the polar form of a map can be defined similarly as

$$P'(\mathbf{x}, \mathbf{y}) = P(\mathbf{x} + \mathbf{y}) - P(\mathbf{x}) - P(\mathbf{y}). \tag{2.7}$$

2.5. Multivariate Quadratic Problem

As described in Section 2.3, multivariate quadratic maps are the foundation of the Oil and Vinegar and the MAYO scheme. This design choice is motivated by the underlying Multivariate Quadratic (MQ) problem. Given a multivariate map P and a target $\mathbf{t}$, the problem is defined as finding a preimage $\mathbf{s}$ such that $P(\mathbf{s}) = \mathbf{t}$. The MQ problem is the basis for the computational hardness of the two schemes, since it is proven to be NP-hard [GJ79]. To our current knowledge, there is no algorithm to solve it in polynomial time if $n \sim m$, even for quantum computers [Beu22b].

2.6. Oil and Vinegar

Using a set of multivariate quadratic equations as central part of a public key scheme has been discussed by multiple authors over several years [FD85; MI88; Pat96]. In 1997, Patarin presented the original Oil and Vinegar algorithm [Pat97]. The motivation was to create a cryptographic scheme based on a set of multivariate quadratic equations to utilize the computational hardness of the MQ problem as described in Section 2.5. The fundamental concept behind these schemes is to incorporate a trapdoor into the set of equations, facilitating efficient signature sampling. Although the original scheme was broken by Kipnis and Shamir [KS98], its adapted version, Unbalanced Oil and Vinegar [KPG99], still appears to remain secure in the present context. Hence, it was selected as the basis for the MAYO scheme.

2.6.1. Scheme Description

The description of the Oil and Vinegar signature scheme is adapted from [Beu22b]. The fundamental component of this scheme is the multivariate quadratic map $P : \mathbb{F}_q^n \to \mathbb{F}_q^m$, which serves as the public key. Signing a message M involves obtaining its digest through a cryptographic hash function H and a randomly generated salt. Subsequently, the signature $\mathbf{s}$ is defined as the preimage under the multivariate quadratic map of the particular digest, satisfying $P(\mathbf{s}) = H(\text{M} \mid\mid \text{salt})$. However, due to the computational hardness of sampling preimages of multivariate quadratic maps, the scheme incorporates a trapdoor to obtain them efficiently. The trapdoor information in the Oil and Vinegar

scheme is denoted as oil space, a linear subspace $O \subset \mathbb{F}_q^n$ of dimension m, where P vanishes in the sense that

$$P(\mathbf{o}) = 0 \quad \text{for all } \mathbf{o} \in O. \tag{2.8}$$

Knowledge of the oil space enables efficient preimage sampling for P and, thus, a basis of the oil space serves as the secret key. Attacks specifically targeting Oil and Vinegar schemes aim to uncover the oil space from the public key, which led to the break of the original Oil and Vinegar version. Since O is hidden in $\mathbb{F}_q^n$, increasing the value of n for a fixed m enhances the difficulty of these attacks. The Unbalanced Oil and Vinegar scheme adheres to this principle by setting n to $3m$ to render oil space recovery attempts unsuccessful.

For a given target $\mathbf{t} \in \mathbb{F}_q^m$, one chooses a vector $\mathbf{v} \in \mathbb{F}_q^n$ and solves the equation $P(\mathbf{v}+\mathbf{o}) = \mathbf{t}$ for $\mathbf{o} \in O$. Using the polar form from Equation. (2.7), it follows that

$$P(\mathbf{v} + \mathbf{o}) = P(\mathbf{v}) + P(\mathbf{o}) + P'(\mathbf{v}, \mathbf{o}) = \mathbf{t}. \tag{2.9}$$

Since $P(\mathbf{v})$ is fixed due to the choice of $\mathbf{v}$ and $P(\mathbf{o})$ evaluates to 0 by the definition of the oil space in Equation (2.8), only the linear system $P'(\mathbf{v}, \mathbf{o}) = \mathbf{t} - P(\mathbf{v})$ remains to be solved for $\mathbf{o}$ and the signature is computed via $\mathbf{s} = \mathbf{v} + \mathbf{o}$. Although Oil and Vinegar is a well-researched and relatively old scheme, its use cases as a practical signing algorithm are limited, since it suffers from large public key sizes in the order of 50 KB.

2.7. Gaussian Elimination in $\mathbb{F}_{16}$

Both, UOV and MAYO, rely on the matrix representation of multivariate polynomials and signature sampling is accomplished by solving a linear system, which is obtained using the introduced trapdoor. One of the best known algorithms to solve linear systems in matrix representation is Gaussian Elimination. It takes an augmented matrix as input and transforms it into echelon form. A matrix is defined to be in echelon form if all zero rows are at the bottom, the leading non zero entry of every row has to be 1, and every leading 1 has to be on the left of the leading 1 of the row below it. An example of a matrix in echelon form is shown in Equation 2.10.

$$\begin{pmatrix} 1 & c_{12} & c_{13} & c_{14} & c_{15} \\ 0 & 0 & 1 & c_{24} & c_{25} \\ 0 & 0 & 0 & 1 & c_{35} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.10}$$

To transform a matrix into echelon form, Gaussian Elimination iterates over all rows and columns from top to bottom and left to right, respectively, to find a non zero entry. If this entry is found, the whole row is multiplied by its inverse to set it to 1. This row is then used to annihilate all entries in the same column of the rows below. A detailed description of the algorithm can be found in Algorithm 1. Since the coefficients of the

Value	Inverse	Value	Inverse	Value	Inverse
1	1	6	7	11	5
2	9	7	6	12	10
3	14	8	15	13	4
4	13	9	2	14	3
5	11	10	12	15	8

Table 2.1.: Multiplicative Inverses of $\mathbb{F}_{16}$

linear system are in a finite field, the inverse depends on the choice of $\mathbb{F}_q$. The concrete values for $\mathbb{F}_{16}$ are listed in Table 2.1.

After transforming a linear system into echelon form, its solution can be obtained using back-substitution by computing every single component of the solution vector in a bottom-up fashion. Compared to other methods, Gaussian elimination has the benefit that it does not require the original matrix to be in a specific form and it is more efficient than similar algorithms like Gauss-Jordan Elimination and matrix inversion [Hea18].

Algorithm 1 Gaussian Elimination [Beu+23a]

GaussianElimination($\mathbf{A}$):
Input: Matrix $\mathbf{A} \in \mathbb{F}_q^{m \times n}$
Output: Matrix $\mathbf{A}' \in \mathbb{F}_q^{m \times n}$ in echelon form

pivot_row $\leftarrow 0$, pivot_column $\leftarrow 0$
while pivot_row $< m$ and pivot_column $< n$ **do**
    next_pivot_row $\leftarrow$ pivot_row
    while next_pivot_row $< m$ and $\mathbf{A}$[next_pivot_row, pivot_column] $= 0$ **do**
        if next_pivot_row $= m$ **then**
            pivot_column $\leftarrow$ pivot_column $+ 1$
        else
            if next_pivot_row $>$ pivot_row **then**
                Swap($\mathbf{A}$[pivot_row, :], $\mathbf{A}$[next_pivot_row, :])
            $\mathbf{A}$[pivot_row, :] $\leftarrow \mathbf{A}$[pivot_row, pivot_column]$^{-1}\mathbf{A}$[pivot_row, :]
            for row from pivot_row $+ 1$ to $m - 1$ **do**
                $\mathbf{A}$[row, :] $\leftarrow \mathbf{A}$[row, :] $- \mathbf{A}$[row, pivot_column]$\mathbf{A}$[pivot_row, :]
            pivot_row $\leftarrow$ pivot_row $+ 1$
            pivot_column $\leftarrow$ pivot_column $+ 1$
return A

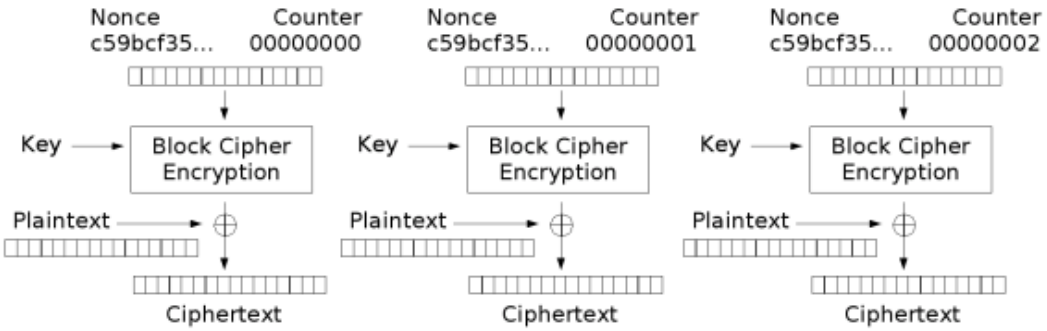

Figure 2.1.: AES-CTR mode

2.8. AES

Advanced Encryption Standard (AES) is a symmetric key encryption algorithm established by NIST in 2001 [Dwo+01]. It is based on Rijndael by Daemen and Rijmen [DR99]. Aes operates on blocks of data, each block being 128 bits in size, and supports three key lengths: 128, 192, and 256 bits. The algorithm consists of several rounds of substitution, permutation, and mixing operations, known as the SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations, which are applied sequentially.

Instead of encryption, Aes is used to generate pseudo-random data in the original MAYO version. To achieve that, counter (CTR) [Dwo01] is used as mode of operation. In this mode Aes is transformed into a stream cipher by utilizing the cipher's output as a keystream and performing XOR operations between the plaintext and the keystream. As illustrated by Figure 2.1, a static nonce and an increasing counter is reencrypted multiple times under the same key to generate a keystream of the desired length. The individual keystream blocks can be generated independently, since the outputs of Aes are not connected. Thus, a user is able to generate a certain block without computing the preceding blocks by setting the appropriate counter value.

To accelerate the performance of the algorithm, MAYO utilizes Advanced Encryption Standard New Instructions (AES-NI), which is an extension to the x86 instruction set architecture introduced by Intel in 2010 [Gue10]. It aims to accelerate the Aes encryption and decryption operations by implementing dedicated hardware instructions specifically designed for Aes operations. Aes-Ni provides a set of new instructions that significantly enhance the performance of Aes encryption and decryption algorithms on processors that support this feature. These instructions support different modes of operation for Aes, among them CTR. By offloading Aes operations to specialized hardware, Aes-Ni improves the efficiency and speed of cryptographic tasks that rely on Aes encryption, as it is the case for MAYO.

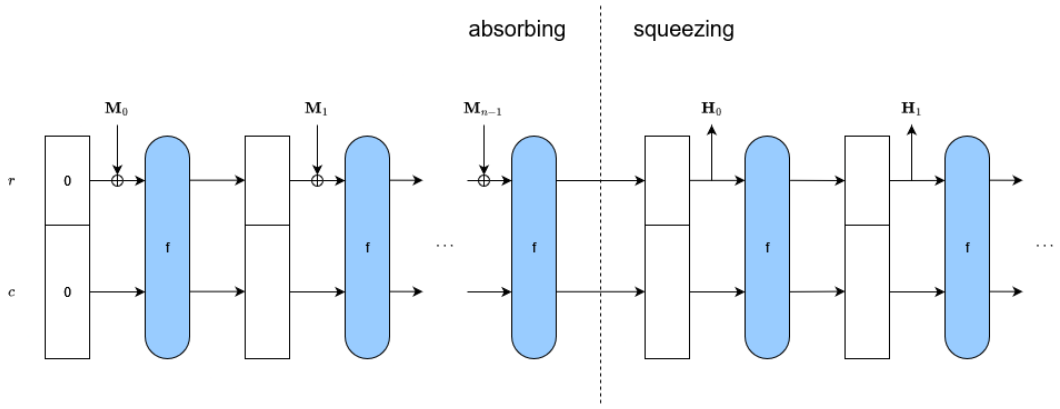

Figure 2.2.: Sponge construction

2.9. SHAKE

MAYO uses AES-128 in counter mode to sample pseudo-random data. In our hardware design we exchange AES with SHAKE. SHAKE is part of the SECURE HASH ALGORITHM (SHA) 3 family standardized by NIST [Dwo15]. SHA-3 is based on the cryptographic suite KECCAK by Bertoni *et al.* [Ber+11]. NIST established this new standard to have an alternative to previous hash functions, which were based on the Merkle-Damgård construction. KECCAK uses a novel approach instead, called sponge construction. Hence the name, this construction allows to absorb an arbitrary amount of input data and squeezing an arbitrary amount of output data.

As shown in Figure 2.2, the sponge constructions utilizes an internal state consisting of $r + c$ bits. The input is partitioned into blocks which are successively XORed with r bits of the state. Upon absorbing the entire input, the sponge construction squeezes out blocks until the desired output length is reached. Between every step of the absorbing and squeezing phase the function f is applied to the whole state. In the case of SHA-3, the state size is $r + c = 1600$, with r and c depending on the specific SHA instance. The entire state is divided into a 5×5 array of 64-bit state variables. Additionally to common hash functions, which compute a fixed-length digest, SHA-3 also includes SHAKE128 and SHAKE256, which allow an arbitrary output length. This property renders the SHAKE algorithms as ideal choice to sample pseudo-random data in the MAYO scheme.

2.10. AVX

Advanced Vector Extensions (AVX) is an extension for the instruction set architecture for microprocessor proposed by Intel in 2008 [Gep17]. They allow the execution of Single Instruction Multiple Data (SIMD) operations [Lom11]. AVX uses special registers to carry out a instruction on a larger amount of data in parallel. However, AVX exclu-

sively enables 256-bit data processing for floating points, whereas its successor, AVX2, extended the instruction set in 2013 to support 256 bits for integers as well. Today, these standards are compatible with a wide range of CPU families [Wika]. In 2016, Intel released the first CPU equipped with the newest update AVX-512, further expanding the instruction set to accommodate 512-bit data [Rei17]. Despite providing an additional performance increase, AVX-512 is presently not as widely adopted as its two predecessor standards [Wikb].

Given that numerous sections of the MAYO scheme use large amounts of data with identical arithmetic instructions, AVX emerges as a good choice for enhancing implementation performance. Instead of performing operations on single vector or matrix elements sequentially, they can be loaded into an AVX register and processed in parallel. Since one element of $\mathbb{F}_{16}$ is represented by 4 bits, 64 elements fit into one AVX register.

Chapter 3.

MAYO Scheme

In this chapter, we give a detailed explanation of the MAYO scheme extending the description provided in [Hir+23]. The description is adapted from [Beu22b], incorporating the latest specifications and algorithms outlined in [Beu+23a]. The MAYO scheme alters the original Oil and Vinegar scheme to address the challenge of large public key sizes. The design philosophy closely aligns with UOV, adhering to identical principles. Security is established on the MQ-problem, with signatures representing preimages of the hashed message under a multivariate quadratic map. Key and signature generation, as well as signature verification, closely resemble Oil and Vinegar. The primary distinction lies in the selection of the oil space dimension. MAYO employs an oil space deemed too small for the original scheme. As the oil space is hidden in $\mathbb{F}_q^n$, a smaller size makes it more challenging to recover. Consequently, the other parameters can be reduced without compromising the scheme's security. Nevertheless, this modification makes signature sampling impossible in most cases using the Oil and Vinegar algorithm. The authors of MAYO, however, have identified a solution for this challenge. They introduce a *whipping* mechanism that transforms the multivariate quadratic map $P : \mathbb{F}_q^n \to \mathbb{F}_q^m$ into an expanded map $P^* : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$. This approach enables the selection of a smaller oil space, consequently resulting in a significant reduction in the key size. To delve into the details of the whipping construction, it is essential to first understand why the dimension of the oil space plays a crucial role in determining the size of the public key.

3.1. Public Key Size

In the Oil and Vinegar scheme, the public key comprises the multivariate quadratic map P, consisting of m multivariate quadratic polynomials in n variables. As a result, the memory needed to store P is $\mathcal{O}(mn^2 \log q)$ bits, attributable to the matrix form of a polynomial defined in Equation (2.5). Petzoldt *et al.*[Pet+11] demonstrated that $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$ can be generated pseudo-randomly. Consequently, only $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ needs to be stored as the public key, leading to a reduction in key size to $\mathcal{O}(mo^2 \log q)$ bits. Since $\mathbf{P}_i^{(3)}$ is upper triangular the exact public key size is

$$|pk| = m(\frac{o(o-1)}{2} + o) \log q + |\text{seed}_{\text{pk}}|, \tag{3.1}$$

where seed_{pk} denotes the seed which is used to generate $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. Nevertheless, the original Oil and Vinegar scheme mandates that o must be at least as large as m, otherwise, the linear system derived from Equation (2.9) becomes unsolvable with high probability. The MAYO scheme presents a novel whipping technique to further decrease the size of the public key by reducing the dimension of the oil space while maintaining the solvability of the linear system.

3.2. Whipping Technique

As outlined in introduction of Chapter 3, MAYO undergoes a transformation of P into a larger map denoted as P^*. This whipping transformation must possess the property that if P vanishes on a subspace $O \subset \mathbb{F}_q^n$, then P^* must also be zero on $O^k \subset \mathbb{F}_q^{kn}$, where k serves as the whipping parameter determining the size of the oil space, with $o = \lceil m/k \rceil$. While such a transformation is easy to achieve, the resulting map P^* needs to preserve the preimage resistance of P. To illustrate this challenge, we inspect the transformation

$$P^*(\mathbf{x}_1, \ldots, \mathbf{x}_k) = P(\mathbf{x}_1) + \cdots + P(\mathbf{x}_k). \tag{3.2}$$

The map P^* obtained from Equation (3.2) clearly fulfills the requirements, that it maps from $\mathbb{F}_q^{kn}$ to $\mathbb{F}_q^m$ and if P vanishes on O, then the same holds for P^* on O^k. However, it is not preimage resistant. Assume there exists $\alpha \in \mathbb{F}_q$ such that $\alpha^2 = -1$. Then $P(\alpha \mathbf{x}_1) = -P(\mathbf{x}_1)$, since P consists only of homogeneous polynomials. Subsequently, an adversary can randomly select $\delta \in \mathbb{F}_q^n$, set $\mathbf{x}_2 = \alpha \mathbf{x}_1 + \delta$, and assign $\mathbf{x}_i = 0$ for $i > 2$. As a result, we obtain

$$\begin{aligned} P^*(\mathbf{x}_1, \mathbf{x}_2, \mathbf{0}, \ldots, \mathbf{0}) &= P(\mathbf{x}_1) + P(\mathbf{x}_2) \\ &= P(\mathbf{x}_1) + P(\alpha \mathbf{x}_1 + \delta) \\ &= P(\mathbf{x}_1) + P(\alpha \mathbf{x}_1) + P(\delta) + P'(\alpha \mathbf{x}_1, \delta) \\ &= P(\mathbf{x}_1) - P(\mathbf{x}_1) + P(\delta) + P'(\alpha \mathbf{x}_1, \delta) \\ &= \underbrace{P(\delta)}_{\text{fixed by choice of } \delta} + \underbrace{P'(\alpha \mathbf{x}_1, \delta)}_{\text{linear}}, \end{aligned} \tag{3.3}$$

by using Equation (2.7). Given a message digest $\mathbf{t} \in \mathbb{F}_q^m$, an attacker can effectively determine a solution for $\mathbf{x}_1$, such that $P^*(\mathbf{x}_1, \alpha \mathbf{x}_1 + \delta, \mathbf{0}, \ldots, \mathbf{0}) = \mathbf{t}$ by solving the linear system shown in Equation (3.3). Thus, anyone is able to sample preimages to given messages without knowledge of the private key and, consequently, can forge signatures. As demonstrated by this example, finding transformations in a way that the security properties of multivariate quadratic maps are maintained is not trivial.

To preserve the preimage resistance, the authors of MAYO propose the so-called whipping operation defined as

$$P^*(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{x}_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{x}_i, \mathbf{x}_j). \tag{3.4}$$

The matrices denoted as $\mathbf{E}_{ij} \in \mathbb{F}_q^{m \times m}$ are named emulsifier maps and have a crucial role, serving as the fundamental element for the security of the whipping technique. Further details about these emulsifier maps can be found in Section 3.4. Since the parameters are chosen to satisfy $ko > m$ and given the property that P^* vanishes on O^k, the signature of MAYO can be sampled similarly to Equation (2.9) of UOV, by solving the linear system

$$P^*(\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k) = \mathbf{t}, \tag{3.5}$$

which has m equations in ko variables.

3.3. Scheme Description

This section provides a comprehensive description of the key generation, signature computation, and signature verification algorithms of MAYO. The description and algorithms are adapted from [Beu22b; Beu+23a].

3.3.1. Key Generation

To generate a key pair, a randomly generated secret seed is expanded via SHAKE256, and the resulting output is utilized as the matrix $\mathbf{O} \in \mathbb{F}_q^{(n-o) \times o}$ and the public seed. This matrix $\mathbf{O}$ serves as the secret key, and the corresponding oil space O is defined as the rowspace of $(\mathbf{O}^\top \mathbf{I}_o)$, where $\mathbf{I}_o$ represents the identity matrix of size o. As outlined in Equation (2.8), the multivariate quadratic map P has to vanish on O. Consequently, every polynomial $p_i(\mathbf{x})$ of the map P is required to satisfy the condition that

$$(\mathbf{O}^\top \mathbf{I}_o) \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} (\mathbf{O}^\top \mathbf{I}_o)^\top = \mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{O}^\top \mathbf{P}_i^{(2)} + \mathbf{P}_i^{(3)} = 0. \tag{3.6}$$

It follows, that $\mathbf{P}_i^{(3)} = -\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^\top \mathbf{P}_i^{(2)}$. However, $\mathbf{P}_i^{(3)}$ has to be in upper triangular form, which is not necessarily the case if this formula is applied. Nevertheless, every polynomial matrix can be transformed into upper triangular form, as outlined in Section 2.3, without changing the underlying polynomial by simply adding the lower half to the upper half. Hence, it is feasible to randomly generate $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ using a public seed, and set $\mathbf{P}_i^{(3)}$ to $\text{UPPER}(-\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^\top \mathbf{P}_i^{(2)})$, where $\text{UPPER}(\cdot)$ is defined as $\text{Upper}(\mathbf{M}_{ii}) = \mathbf{M}_{ii}$ and $\text{UPPER}(\mathbf{M}_{ij}) = \mathbf{M}_{ij} + \mathbf{M}_{ji}$ for $i < j$. The key generation algorithm is shown in Algorithm 2 and the described matrix generation technique is applied in Lines 7 and 8.

Algorithm 2 KEY GENERATION [Beu+23a]

CompactKeyGen():

Output: Compact public key cpk $\in \mathcal{B}^{\text{cpk\_bytes}}$ and secret key csk $\in \mathcal{B}^{\text{csk\_bytes}}$

1: $\text{seed}_{\text{sk}} \leftarrow \text{RANDOM}(\text{sk\_seed\_bytes})$ ▷ Pick at random
2:
3: //Expand seed_{sk} to get seed_{pk} and $\mathbf{O}$
4: $\text{seed}_{\text{pk}}, \mathbf{O} \leftarrow \text{SHAKE256}(\text{seed}_{\text{sk}})$ ▷ $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$
5:
6: //Expand seed_{pk} to get $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
7: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \text{AES-128-CTR}(\text{seed}_{\text{pk}})$
8: $\{\mathbf{P}_i^{(3)}\}_{i\in[m]} \leftarrow \text{Upper}(-\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^\top \mathbf{P}_i^{(2)})_{i\in[m]}$ ▷ Compute $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$
9:
10: $\text{cpk} \leftarrow \text{seed}_{\text{pk}} \;||\; \{\mathbf{P}_i^{(3)}\}_{i\in[m]}$ ▷ Public Key
11: $\text{csk} \leftarrow \text{seed}_{\text{sk}}$ ▷ Secret Key
12: **return** (cpk, csk)

The generation of substantial matrix parts through pseudorandom number generators (PRNGs) facilitates a significant reduction in key size, as there is no necessity to store the entire key information. Instead, we generate segments of both the public and private keys based on their respective seeds. For the private key, it is now sufficient to store only the private seed, while the public key is composed of the public seed and $\mathbf{P}_i^{(3)}$. Furthermore, the whipping transformation outlined in Section 3.2 enables the reduction of the size of $\mathbf{P}_i^{(3)}$ from $m \times m$ to $o \times o$.

3.3.2. Signature Computation

To obtain a signature for a message M, a random salt is generated, and the digest $\mathbf{t} = H(H(\text{M}) \;||\; \text{salt})$ is computed. Subsequently, a set of vectors $(\mathbf{v}_1, \ldots, \mathbf{v}_k)$ is chosen randomly, and the linear system for $(\mathbf{o}_1, \ldots, \mathbf{o}_k)$ is solved as depicted in Equation (3.5). Following the approach outlined by Beullens *et al.* [Beu+23a], the last o entries of $\mathbf{v}_i$ can be safely set to 0 without altering the distribution of the signing output. Consequently, one generates a random vector $\tilde{\mathbf{v}}_i \in \mathbb{F}_q^{(n-o)}$ and sets to $\mathbf{v}_i$ to $(\tilde{\mathbf{v}}_i, 0)$. This choice ensures that only $\mathbf{P}_i^{(1)}$ is necessary for the signature computation. Analogous to Equation (2.9), the oil space trapdoor information enables the partitioning of Equation (3.5) into a

constant and a linear part, leading to

$$\begin{aligned}
P^*(\mathbf{v}_1+\mathbf{o}_1,\ldots,\mathbf{v}_k+\mathbf{o}_k) &= \sum_{i=1}^{k}\mathbf{E}_{ii}P(\mathbf{v}_i+\mathbf{o}_i)+\sum_{i=1}^{k}\sum_{j=i+1}^{k}\mathbf{E}_{ij}P'(\mathbf{v}_i+\mathbf{o}_i,\mathbf{v}_j+\mathbf{o}_j)\\
&= \sum_{i=1}^{k}\mathbf{E}_{ii}(P(\mathbf{v}_i)+P'(\mathbf{v}_i,\mathbf{o}_i))+\sum_{i=1}^{k}\sum_{j=i+1}^{k}\mathbf{E}_{ij}(P'(\mathbf{v}_i,\mathbf{v}_j)+P'(\mathbf{v}_i,\mathbf{o}_j)+P'(\mathbf{v}_j,\mathbf{o}_i))\\
&= \sum_{i=1}^{k}\mathbf{E}_{ii}P(\mathbf{v}_i)+\sum_{i=1}^{k}\sum_{j=i+1}^{k}\mathbf{E}_{ij}P'(\mathbf{v}_i,\mathbf{v}_j) \quad \text{(constant)}\\
&+\sum_{i=1}^{k}\mathbf{E}_{ii}P'(\mathbf{v}_i,\mathbf{o}_i))+\sum_{i=1}^{k}\sum_{j=i+1}^{k}\mathbf{E}_{ij}(P'(\mathbf{v}_i,\mathbf{o}_j)+P'(\mathbf{v}_j,\mathbf{o}_i)) \quad \text{(linear)}\\
&= \mathbf{t}.
\end{aligned} \tag{3.7}$$

Computation of the constant part can be achieved using

$$\begin{aligned}
p_i(\mathbf{v}_k) &= \tilde{\mathbf{v}_k}^{\top}\mathbf{P}_i^{(1)}\tilde{\mathbf{v}_k},\\
p_i'(\mathbf{v}_k,\mathbf{v}_l) &= \tilde{\mathbf{v}_k}^{\top}\mathbf{P}_i^{(1)}\tilde{\mathbf{v}_l}+\tilde{\mathbf{v}_l}^{\top}\mathbf{P}_i^{(1)}\tilde{\mathbf{v}_k}.
\end{aligned} \tag{3.8}$$

To compute the linear part, it is necessary to perform the evaluation of the linear transformation $P'(\mathbf{v}_k,\cdot)$. This can be accomplished by utilizing the matrix representation of the linear transformation, defined as follows:

$$\mathbf{L}_i = (\mathbf{P}_i^{(1)}+{\mathbf{P}_i^{(1)}}^{\top})\mathbf{O}+\mathbf{P}_i^{(2)}. \tag{3.9}$$

Subsequently, each component $p_i'(\mathbf{v}_k,\cdot)$ in P' can be written as $\tilde{\mathbf{v}_k}^{\top}\mathbf{L}_i$. Applying Equation (3.8) and Equation (3.9) to Equation (3.7) yields an augmented matrix that must be solved for $\mathbf{o}_i$ to determine the signature. Hence, MAYO builds a linear system $\mathbf{Ax}=\mathbf{y}$, where $\mathbf{A}$ consists of the linear part of Equation (3.7) and $\mathbf{y}$ is the vectorized message digest $\mathbf{t}$ minus the respective constant part. Algorithm 3 shows the signing algorithm in detail.

In Lines 21 and 22, the described computation of the constant part is applied, whereas in Lines 18, and 23 to 25 the evaluation of the linear part and the construction of $\mathbf{A}$ occurs. Solving the linear system can be accomplished using various algorithms, with Gaussian Elimination being one example. The process of solving the system takes place inside the SampleSolution function which is explained in the following section.

3.3.3. Sample Solution

To sample a solution, the constructed matrix $\mathbf{A}\in\mathbb{F}_q^{m\times ko}$ of Algorithm 3 needs to be solved such that the solution vector $\mathbf{x}$ satisfies $\mathbf{Ax}=\mathbf{y}$. Since $ko\geq m$, the system

Algorithm 3 SIGNATURE COMPUTATION [Beu+23a]

Sign(csk, M) with incorporated ExpandSK:
Input: Compact secret key csk $\in \mathcal{B}^{\text{csk\_bytes}}$ and message M $\in \mathcal{B}^*$
Output: Signature sig $\in \mathcal{B}^{\text{sig\_bytes}}$

$\text{seed}_{\text{sk}} \leftarrow$ csk ▷ csk is seed_{sk}
//Expand seed_{sk} to get seed_{pk} and $\mathbf{O}$
$\text{seed}_{\text{pk}}, \mathbf{O} \leftarrow \text{SHAKE256}(\text{seed}_{\text{sk}})$ ▷ $\mathbf{O} \in \mathbb{F}_q^{(n-o)\times o}$
//Expand seed_{pk} to get $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
$\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \text{AES-128-CTR}(\text{seed}_{\text{pk}})$
$\{\mathbf{L}_i\}_{i\in[m]} \leftarrow \{(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top})\mathbf{O} + \mathbf{P}_i^{(2)}\}_{i\in[m]}$ ▷ Compute linear part $\mathbf{L}_i \in \mathbb{F}_q^{(n-o)\times o}$

M_digest $\leftarrow \text{SHAKE256}(\text{M})$
R $\leftarrow \text{RANDOM}(\text{R\_bytes})$ ▷ Pick at random
salt $\leftarrow \text{SHAKE256}(\text{M\_digest} \,||\, \text{R} \,||\, \text{seed}_{\text{sk}})$
$\mathbf{t} \leftarrow \text{SHAKE256}(\text{M\_digest} \,||\, \text{salt})$ ▷ $\mathbf{t} \in \mathbb{F}_q^m$

for ctr from 0 to 255 **do** ▷ Find preimage of $\mathbf{t}$
  $\{\mathbf{v}_i\}_{i\in[k]}, \mathbf{r} \leftarrow \text{SHAKE256}(\text{M\_digest} \,||\, \text{salt} \,||\, \text{seed}_{\text{sk}} \,||\, \text{ctr})$ ▷ $\mathbf{v}_i \in \mathbb{F}_q^{n-o}, \mathbf{r} \in \mathbb{F}_q^{ko}$

  $\mathbf{A} \leftarrow \mathbf{0}, l \leftarrow 0, \mathbf{y} \leftarrow \mathbf{t}$ ▷ $\mathbf{A} \in \mathbb{F}_q^{m\times ko}$
  for i from 0 to $k-1$ **do**
    $\{\mathbf{M}_i[j,:]\}_{j\in[m]} \leftarrow \{\mathbf{v}_i^\top \mathbf{L}_j\}_{j\in[m]}$
  for i from 0 to $k-1$ **do** ▷ Build linear system $\mathbf{Ax} = \mathbf{y}$
    for j from $k-1$ to i **do**
      $\mathbf{u} \leftarrow \begin{cases} \{\mathbf{v}_i^\top \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a\in[m]} & \text{if } i = j \\ \{\mathbf{v}_i^\top \mathbf{P}_a^{(1)} \mathbf{v}_j + \mathbf{v}_j^\top \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a\in[m]} & \text{if } i \neq j \end{cases}$ ▷ $\mathbf{u} \in \mathbb{F}_q^m$
      $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}^l \mathbf{u}$
      $\mathbf{A}[:, i*o : (i+1)*o] \leftarrow \mathbf{A}[:, i*o : (i+1)*o] + \mathbf{E}^l \mathbf{M}_j$
      if $i \neq j$ **then**
        $\mathbf{A}[:, j*o : (j+1)*o] \leftarrow \mathbf{A}[:, j*o : (j+1)*o] + \mathbf{E}^l \mathbf{M}_i$
      $l \leftarrow l + 1$
  $\mathbf{x} \leftarrow \text{SampleSolution}(\mathbf{A}, \mathbf{y}, \mathbf{r})$ ▷ $\mathbf{x} \in \mathbb{F}_q^{ko} \cup \perp$
  if $\mathbf{x} \neq \perp$ **then**
    break
$\mathbf{s} \leftarrow \mathbf{0}$ ▷ $\mathbf{s} \in \mathbb{F}_q^{kn}$
for i from 0 to $k-1$ **do** ▷ Compute signature
  $\mathbf{s}[i*n : (i+1)*n] \leftarrow (\mathbf{v}_i + \mathbf{O}\mathbf{x}[i*o : (i+1)*o]) \,||\, \mathbf{x}[i*o : (i+1)*o]$
return sig $= \mathbf{s} \,||$ salt

is underdetermined and multiple solutions exist. Precisely, the solution space exhibits a dimension of $ko - m$. Thus, the random vector $\mathbf{r}$ is employed to select each of the q^{ko-m} solutions with uniform probability. This is achieved by solving the corresponding system $\mathbf{Ax}' = \mathbf{y} - \mathbf{Ar}$ using the conventional Gaussian Elimination method described in Section 2.7 and returning $\mathbf{x} = \mathbf{x}' + \mathbf{r}$. No solution exists if the input matrix $\mathbf{A}$ is of rank smaller than m, which is the case if the last row of the matrix in echelon form consists only of zeros. Then SampleSolution outputs $\perp$. The full algorithm is shown in Algorithm 4.

Algorithm 4 Sampling Solutions [Beu+23a]

SampleSolution$(\mathbf{A}, \mathbf{y}, \mathbf{r})$:

Input: Linear system in matrix form $\mathbf{A} \in \mathbb{F}_q^{m \times ko}$
Target vector $\mathbf{y} \in \mathbb{F}_q^m$
Randomization vector $\mathbf{r} \in \mathbb{F}_q^{ko}$

Output: Solution $\mathbf{x} \in \mathbb{F}_q^{ko}$ or $\perp$ if system is unsolvable

```
1:  x ← r                                              ▷ Randomize system
2:  y ← y − Ar
3:
4:  (A||y) ← GaussianElimination((A||y))               ▷ Compute Echelon Form of (A||y)
5:
6:  if A[m − 1, :] = 0 then                            ▷ Check if A has rank m
7:      return ⊥
8:
9:  for r from m − 1 to 0 do                           ▷ Back substitution
10:     c ← 0
11:     while A[r, c] = 0 do        ▷ Get index of first non-zero element of current row
12:         c ← c + 1
13:     x_c ← x_c + y_r
14:     y ← y − y_r A[:, c]
15: return x
```

Line 1 illustrates the randomization of the solution, Line 6 demonstrates the solvability check, and Lines 9 to 11 involve the application of back substitution to extract a solution from the echelon form matrix.

3.3.4. Signature Verification

To verify a given message M along with its signature sig = (salt $||$ $\mathbf{s}_1, \ldots, \mathbf{s}_k$), the computation of the hash $\mathbf{t} = H(M \,||\, \text{salt})$ is required, followed by the assessment of the whipped up map $P^*(\mathbf{s}_1, \ldots, \mathbf{s}_k) = \mathbf{y}$. If $\mathbf{y} = \mathbf{t}$, the signature is valid. To evaluate $P^*(\mathbf{s}_1, \ldots \mathbf{s}_k)$, Equation (3.4) needs to be followed. A straightforward approach to accomplish this is to evaluate the $P(\mathbf{s}_i)$ and $P'(\mathbf{s}_i, \mathbf{s}_j)$ segments separately and subsequently combine the intermediate results using the $\mathbf{E}_{ij}$ matrices. The complete verification process is outlined

in Algorithm 5.

Algorithm 5 Signature Verification [Beu+23a]

Verify(cpk, M, sig) with incorporated ExpandPK:
Input: Compact public key cpk $\in \mathcal{B}^{\text{cpk\_bytes}}$, message M $\in \mathcal{B}^*$, and sig $\in \mathcal{B}^{\text{sig\_bytes}}$
Output: 0 if signature is valid, -1 otherwise

1: $\text{seed}_{\text{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]} \leftarrow$ cpk ▷ Extract seed_{pk} and $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ from cpk
2: //Expand seed_{pk} to get $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
3: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow$ Aes-128-Ctr(seed_{pk})
4:
5: salt, $\{\mathbf{s}_i\}_{i\in[k]} \leftarrow$ sig ▷ $\mathbf{s}_i \in \mathbb{F}_q^n$
6: M_digest $\leftarrow$ Shake256(M)
7: $\mathbf{t} \leftarrow$ Shake256(M_digest || salt) ▷ $\mathbf{t} \in \mathbb{F}_q^m$
8: $\mathbf{y} \leftarrow \mathbf{0}, l \leftarrow 0$ ▷ $\mathbf{y} \in \mathbb{F}_q^m$
9: **for** i from 0 to $k-1$ **do** ▷ Evaluate $P^*(\mathbf{s}_1, \dots \mathbf{s}_k)$
10: **for** j from $k-1$ to i **do**
11: $\mathbf{u} \leftarrow \begin{cases} \{\mathbf{s}_i^\top \mathbf{P}_a \mathbf{s}_i\}_{a\in[m]} & \text{if } i = j \\ \{\mathbf{s}_i^\top \mathbf{P}_a \mathbf{s}_j + \mathbf{s}_j^\top \mathbf{P}_a \mathbf{s}_i\}_{a\in[m]} & \text{if } i \neq j \end{cases}$ ▷ $\mathbf{u} \in \mathbb{F}_q^m$
12: $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{E}^l\mathbf{u}$
13: $l \leftarrow l + 1$
14:
15: **if** $\mathbf{y} = \mathbf{t}$ **then** ▷ Accept signature if $\mathbf{y} = \mathbf{t}$
16: **return** 0
17: **return** -1

In Line 11, the calculation of the individual outcomes $P(\mathbf{s}_i)$ and $P'(\mathbf{s}_i, \mathbf{s}_j)$ is performed, while Line 12 illustrates the merging of these results into the final outcome.

3.4. Emulsifier maps

The MAYO signature scheme incorporates a new vital element known as emulsifier maps, denoted as $\mathbf{E} \in \mathbb{F}_q^{m\times m}$. They play a major role in the whipping transformation, which distinguishes it from the original Oil and Vinegar algorithm and ultimately contributes to a more compact public key size. In the context of MAYO, the matrix $\mathbf{E}$ performs multiplication by z in the finite field $\mathbb{F}_q[z]/f(z)$ and it is employed in computations of the form $\mathbf{E}^l\mathbf{u}$, where $\mathbf{u}$ represents a vector of length m and l ranges from 0 to $\frac{k(k+1)}{2} - 1$. Rather than explicitly calculating the matrix multiplications, it is more efficient, particularly considering hardware memory access limitations, to treat $\mathbf{u}$ as a single polynomial and perform the reduction mod $f(z)$ once. This operation resembles a multiplication in the finite field $\mathbb{F}_{16^m}$. Similar to the finite field described in Section 2.2, elements of $\mathbb{F}_{16^m}$ can be represented as a polynomial, but now of degree $m-1$ with coefficients in $\mathbb{F}_{16}$.

Thus, an element $a \in \mathbb{F}_{16^m}$ takes the form

$$a = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_1 z + a_0. \tag{3.10}$$

The emulsifier map $\mathbf{E}$ is now represented by a multiplication by z. Similar to the field multiplication discussed in Section 2.2.2, it is necessary to reduce the resulting polynomial to obtain a valid element in $\mathbb{F}_{16^m}$ once again. In this case, the reduction polynomials for the different security levels MAYO_1, MAYO_3, and MAYO_5, are given by $f_{64}(z) = z^{64}+8z^3+2z^2+8$, $f_{96}(z) = z^{96}+2z^3+2z+2$, and $f_{128}(z) = z^{128}+2z^4+4z^3+8z+4$ respectively. To apply $\mathbf{E}$ to a vector $\mathbf{a}$, we interpret $\mathbf{a}$ as a polynomial in the form described in Equation (3.10) and perform the subsequent computations:

$$\begin{gathered}
\text{MAYO}_1 \\
b = \mathbf{E}a, \quad \text{with} \quad b_i = a_{i-1} \quad \text{for } i \notin \{0,2,3\}. \\
b_0 = 8a_{m-1}, \quad b_2 = 2a_{m-1} + a_1 \quad b_3 = 8a_{m-1} + a_2 \\
\text{MAYO}_3 \\
b = \mathbf{E}a, \quad \text{with} \quad b_i = a_{i-1} \quad \text{for } i \notin \{0,1,3\}. \\
b_0 = 2a_{m-1}, \quad b_1 = 2a_{m-1} + a_0 \quad b_3 = 2a_{m-1} + a_2 \\
\text{MAYO}_5 \\
b = \mathbf{E}a, \quad \text{with} \quad b_i = a_{i-1} \quad \text{for } i \notin \{0,1,3,4\}. \\
b_0 = 4a_{m-1}, \quad b_1 = 8a_{m-1} + a_0 \quad b_3 = 4a_{m-1} + a_2 \quad b_4 = 2a_{m-1} + a_3
\end{gathered} \tag{3.11}$$

It is essential to emphasize that the operations involving addition and multiplication in Equation (3.11) specifically refer to actions within the field $\mathbb{F}_{16}$. This methodology seamlessly aligns with our packed format detailed in Section 4.2, allowing the simultaneous loading of m values and, consequently, an entire $\mathbb{F}_{16^m}$ element within a single hardware cycle. The computation of $\mathbf{E}^l\mathbf{u}$ involves repeating this process l times.

3.5. Attacks on MAYO

This section gives a short overview over attacks on MAYO and closely follows the description in [Beu+23a]. Attacks on MAYO can be classified into two main categories, first targeting MAYO itself and, second, targeting the underlying Oil and Vinegar problem.

3.5.1. Attacks targeting MAYO

This sections describes the possible attacks which are specific for MAYO.

Direct Attack: One method to potentially breach the MAYO signature scheme is to disregard the oil space entirely and instead attempt to solve the system $P^*(\mathbf{s}) = \mathbf{t}$, directly in order to generate a signature for the message M. In order to achieve that,

the attacker selects a salt at random, computes $\mathbf{t} = \text{SHAKE256}(\text{SHAKE256}(\text{M}) \,||\, \text{salt})$, as described in Algorithm 3, and tries to solve the quadratic system. Currently, there are no known algorithms capable of exploiting the whipping structure behind $P^*(s)$ to determine a solution more efficiently than employing a standard system solving algorithm. Without taking advantage of the whipping technique, the problem of solving $P^*(s)$ resembles the MQ problem presented in Section 2.5.

The primary method for solving nonlinear systems over finite fields relies on computing Gröbner bases. One of the most renowned algorithm for this task is the hybrid Wiedemann XL algorithm [BFP09; Yan+07]. Since P^* consists of m equations in ko variables, it is highly underdetermined using the proposed parameter sets of Table 3.1. For this type of systems, Furue *et al.* [FNT21] present the most effective method, which integrates the hybrid technique with the contributions of [TW12]. This method initially condenses the underdetermined system into a series of q^k smaller overdetermined systems before employing the Wiedemann XL algorithm to solve them. The bit complexities for direct attacks in Table 3.3 are based on this method.

Guessing the secret key: Since the secret key is the SHAKE256 expansion of the secret seed_{sk}, as shown by Line 4 of Algorithm 2, an attacker may attempt to deduce the secret key by guessing the random seed bytes. A successful seed guess would require approximately $2^{8\text{sk\_seed\_bytes - 1}}$ attempts on average. Thus, the length of the secret seed is chosen to fulfill the respective NIST level requirements with an additional security margin of 64 bits.

Forging signatures: MAYO signature can be easily forged under a chosen message attack by discovering a collision for SHAKE256. If an attacker obtains two distinct messages $M_1 \neq M_2$ such that $\text{SHAKE256}(M_1) = \text{SHAKE256}(M_2)$, they can request a signature for M_1 from the signing oracle and then present it as a forgery for M_2. To mitigate the risk of hash collisions, the digest output lengths for the different NIST levels are chosen such that SHAKE256 meets the necessary security requirements.

Claw Finding Attack: An adversary can obtain $P^*(\mathbf{s}_i)$ for X random inputs $\{\mathbf{s}_i\}_{i\in[X]}$ and determine the digest $H(H(\text{M}) \,||\, \text{salt}_j)$ for Y random salts $\{\text{salt}_j\}_{j\in[Y]}$. If $XY = q^m$, a collision $P^*(\mathbf{s}_i) = H(H(\text{M}) \,||\, \text{salt}_j)$ occurs with a probability of approximately $1 - e^{-1}$, allowing the adversary to produce the signature $(\mathbf{s}_i, \text{salt}_j)$ for message M. The computational effort of this attack is $36mX + Y2^{17}$, which is equivalent to $12\sqrt{q^m m 2^{17}}$ when X and Y are optimally chosen.

3.5.2. Attacks targeting the Oil and Vinegar problem

The public map P^* is composed of a classic Oil and Vinegar map $P : \mathbb{F}_{16}^n \to \mathbb{F}_{16}^m$, which vanishes on an o-dimensional linear subspace $O \subset \mathbb{F}_{16}^n$. As mentioned in Section 2.6.1, the oil space serves as the secret key and is hidden in $\mathbb{F}_{16}^n$. Consequently, the security of MAYO relies on the inability of an attacker to reconstruct O from P. This challenge has been extensively explored in literature, since it corresponds to a Oil and Vinegar key

recovery attack. While the specific algorithm is different to enable signature computation with the altered parameter set, the MAYO public key itself is essentially an UOV key. The description of the attacks in this section are based on [Beu21] with the modifications of [Beu22b; Beu+23a] to apply them on MAYO.

Kipnis-Shamir attack: Kipnis and Shamir [KS98] first proposed a successful attack on the original Oil and Vinegar problem of Patarin [Pat97], which was extended by [KPG99] to apply to the unbalanced case. This attack aims to identify vectors within the oil space O by leveraging the increased likelihood of these vectors being eigenvectors of certain publicly-known matrices. These matrices are composed of the linear part of the public key map $P'(\mathbf{x}, \mathbf{y})$. Every component $p'_i(\mathbf{x}, \mathbf{y})$ of P' is associated with a matrix M_i such that $p'_i(\mathbf{x}, \mathbf{y}) = \mathbf{x}^\top M_i \mathbf{y}$. The matrices used for the eigenvector computation are then defined as $M_j^{-1} M_i$. The primary challenge of this attack lies in calculating the eigenvectors of approximately q^{n-2o} matrices of size $n \times n$. Asymptotically, the computational expense of computing these eigenvectors equals that of matrix multiplication. Although this attack operates within polynomial time when $n = 2o$, it rapidly becomes computationally impractical for unbalanced Oil and Vinegar instances. For the Kipnis-Shamir data in Table 3.3, Beullens *et al.* [Beu+23a] utilized a lower bound of $36q^{n-2o}n^{2.8}$ as the bit cost of the attack.

Reconciliation attack: The Reconciliation attack [Din+08] attempts to identify vectors within the oil space O until a complete basis for O is obtained. It exploits the property that $P(\mathbf{o}) = 0$ for every $\mathbf{o} \in O$. In general, it is expected that a random quadratic map P would contain roughly q^{n-m} zeros and Oil and Vinegar maps include additional q^o fabricated zeros in O. As shown in Table 3.1, $o > n - m$, and, thus, the majority of zeros in P lie in O. Therefore, to find a vector in O, an attacker can solve $P(\mathbf{x}) = 0$ for $\mathbf{x}$ using the method described in the Direct Attack section. Since O is of dimension o, an attacker can employ o random affine constraints to reduce the variable set to $n - o$, and, consequently, the resulting system is likely to have a unique solution, corresponding to a vector in O. Hence, identifying a vector in O simplifies to finding a solution for a quadratic system with $n - o$ variables and m inhomogeneous multivariate equations. Upon a single vector $\mathbf{x}$ in O is discovered, determining the remaining vectors in O becomes a significantly simpler task. This is due to the requirement that $P'(\mathbf{x}, \mathbf{y}) = 0$ needs to hold in addition to $P(\mathbf{y}) = 0$ for the second vector $\mathbf{y}$, resulting in a quadratic system with a smaller amount of variables. Thus, finding the first vector is the dominating factor in the attack complexity of the Reconciliation attack.

Intersection attack: The intersection attack [Beu21] extends the reconciliation attack by incorporating elements from the Kipnis-Shamir approach. Its principle lies in exploring multiple vectors within the oil space concurrently. The attack aims to identify $k \geq 2$ vectors within O by solving a quadratic system with $\binom{k+1}{2}m - 2\binom{k}{2}$ equations involving $\min(n, nk - (2k-1)m)$ variables. In the case of MAYO, the most efficient outcomes occur when $k = 2$. However, the attack's success is only assured if $3o > n$, a condition not satisfied by the parameters of MAYO. When $3o \leq n$, the probability to succeed is $q^{-n+3o-1}$, requiring an average of q^{n-3o+1} repetitions. Therefore, the total

attack complexity is q^{n-3o+1} times the cost of solving a quadratic system with $3m-2$ equations in n variables. Upon a successful attempt, the k found vectors can be extended to a complete basis of O as outlined in the Reconciliation attack section. Given the small value of o in MAYO, the intersection attack exhibits notably low success rates, rendering it considerably less efficient than in the conventional Oil and Vinegar setup where $o = m$.

3.6. Parameter Sets

NIST mandates that every submitted algorithm satisfies certain security levels [NIS22] and, thus, lists 5 different categories, also referred to as levels, in increasing order of strength. Levels 1, 2, and 5 are categorized in regard to the computational complexity required to carry out a key search on a block cipher, such as AES, with key lengths of 128, 192, and 256 bits, respectively. Levels 2 and 4, on the other hand, are defined by the computational resources needed for collision search on a hash function with either 256 or 384 bits, for instance SHA-3.

It is recommended to meet the requirements of levels 1 to 3 as well as one of the higher levels 4 or 5. MAYO aims to satisfy levels 1, 3, and 5 by providing different parameter sets, denoted as MAYO_1, MAYO_3, MAYO_5, respectively, to achieve that. The definitions of the parameter sets are shown in Table 3.1. Furthermore, the authors of MAYO specify MAYO_2, an additional parameter set reaching security level 1, however, we omitted this set from this thesis because it does not target small public key sizes, the original objective of MAYO.

To demonstrate the effectiveness of the whipping technique, we compare the key sizes of MAYO with an implementation of UOV. Beullens *et al.* [Beu+23e] specify parameter sets for UOV also targeting NIST security levels 1, 3, and 5. These parameter sets are shown in Table 3.2.

As demonstrated by the table, whipping accomplishes a reduction of the public key size by approximately 98 % across all security levels. Accordingly, the sizes of the expanded keys are reduced by at least 80 % for the public key and 77 % for the secret key. This reduction comes with an increased signature size by a factor of 3. Nevertheless, this increase may be neglected as the signatures typically consist of only a few hundred bytes, whereas the public key size decrease falls within the kilobyte range.

To provide proof that the respective security level requirements are fulfilled, MAYO [Beu+23a] gives the attack complexity in bits, shown in Table 3.3, for the different attacks described in Section 3.5. It is important to note that the bit complexity attacks in the table are lower bounds, since they do not account for the cost of memory accesses, which provides an additional security margin. The targeted security levels 1, 3, and 5 are specified in regard to the computational complexity of key search on AES. As shown by Table 3.3 attacks on the chosen parameter sets exceed these security requirements.

Table 3.1.: MAYO parameter sets for NIST security levels 1, 3, and 5. Adapted from [Beu+23a]

Parameter Set	**MAYO_1**	**MAYO_3**	**MAYO_5**
Security Level	**1**	**3**	**5**
n	66	99	133
m	64	96	128
o	8	10	12
k	9	11	12
q	16	16	16
Salt Bytes	24	32	40
Digest Bytes	32	48	64
PK Seed Bytes	16	16	16
$f(z)$	$f_{64}(z)$	$f_{96}(z)$	$f_{128}(z)$
Secret Key Size	24 B	32 B	40 B
Public Key Size	1168 B	2656 B	5008 B
Signature Size	321 B	577 B	838 B
Expanded SK Size	69 KB	230 KB	553 KB
Expanded PK Size	70 KB	233 KB	557 KB

Table 3.2.: UOV parameter sets for NIST security levels 1, 3, and 5. Adapted from [Beu+23e]

Parameter Set	**ov-Is**	**ov-III**	**ov-V**
Security Level	**1**	**3**	**5**
n	160	184	244
m	64	72	96
q	16	256	256
Secret Key Size	48 B	48 B	48 B
Public Key Size	65 KB	185 KB	437 KB
Signature Size	96 B	200 B	260 B
Expanded SK Size	341 KB	1020 KB	2380 KB
Expanded PK Size	403 KB	1197 KB	2802 KB

Table 3.3.: Lower bound complexity in bits for different attacks against chosen parameter sets Adapted from [Beu+23a]

Parameter Set	**Kipnis-Shamir**	**Reconciliation**	**Intersection**	**Direct Attack**	**Claw-finding**
MAYO_1	222	143	255	145	143
MAYO_3	340	209	390	210	207
MAYO_5	461	276	525	275	272

3.7. MAYO Implementations

The MAYO team created three different software implementations of their algorithm, which are available under [Beu+23b]. They provide C implementations with an increasing degree of optimization in the following order: Reference, Optimized, and AVX2 optimized.

Reference implementation: In most instances of the reference implementation, matrix-matrix and matrix-vector multiplications are executed using a bitsliced representation. The choice of m being divisible by 32 enables the computation of 32 $\mathbb{F}_{16}$-additions or $\mathbb{F}_{16}$-multiplications in parallel utilizing four 32-bit variables. All parameter sets are supported by a single library through the use of runtime parameters.

Optimized implementation: The optimized implementation deviates from the reference implementation in two key aspects. To begin with, the MAYO parameters are defined at compile-time, leading to the creation of distinct libraries for each set. Modern compilers efficiently unroll matrix arithmetic operations, eliminating the need for manual loop unrolling. Furthermore, distinct bitsliced arithmetic functions are introduced for different values of m. For all parameter sets m computations can be carried out in parallel using four 64-bit, twelve 32-bit, or eight 64-bit variables for the respective value of m. A significant portion of the computational time involved in key expansion is consumed by AES, allowing for substantial performance enhancement through the utilization of an AES library with AES-NI support. Nevertheless, the extent of acceleration achieved may vary depending on the particular AES implementation. AES can either be done in software or in hardware via intrinsic function calls of AES-NI, however, this depends on the CPU support. Thus, the usage of AES-NI is optional.

AVX2 implementation: The AVX2 implementation employs compiler assembly intrinsics to utilize SIMD and AVX instructions. Different strategies are employed to optimize bitsliced arithmetic for every value of m. Similar to before, m multiplications can be carried out in parallel, however, in this case vector registers are used. Among the values of m, 64 holds a distinct position since all of the values are encoded within a single 256-bit vector and multiplication is executed using vector permute and shuffle instructions. On the other hand, for 96 and 128, three 128-bit and two 256-bit registers are utilized, respectively. Additional optimizations involve the unrolling of matrix multiplication loops for every operation in MAYO, the interleaving of multiple bitsliced arithmetic operations, and the reuse of intermediate values. The efficiency of multiplications in the computation of the echelon form is enhanced through the utilization of optimized AVX2 shuffle instructions. Additionally, AES-NI can be used to increase the PRNG performance, as described in the optimized implementation. Similar to the previous case, the utilization of AES-NI is optional also for the AVX2 implementation.

Chapter 4.

Key Approaches for a Hardware Design

The MAYO signature scheme exhibits several properties which render an efficient hardware design challenging. At first glance, the primary arithmetic operations employed offer hardware-friendly characteristics. Nonetheless, the large memory demand requires a sophisticated memory design to store the matrix elements and retrieve them efficiently. This ensures that the relatively straightforward arithmetic operations can be effectively utilized. Furthermore, it is challenging to cope with the pseudo-random generation of matrix coefficients due to the high memory demand. Thus, the seamless integration of coefficient generation into the memory design is essential to facilitate high-performance implementations.

Given the novelty of MAYO and the challenges it presents, HaMAYO [Say+23] is the only Field Programmable Gate Array (FPGA) implementation currently available. However, HaMAYO is limited to performing key generation and signature computation solely for security level 1. Should MAYO emerge as a promising candidate of the post-quantum signature call, it is essential that efficient implementations of the complete algorithm exist not only for software but also for hardware platforms.

The goal of this thesis is not to present a complete hardware design or to provide exhaustive descriptions of individual modules, given the complexity of the signature scheme. Instead, we identify and analyse the main challenges inherited by the algorithm specification and propose design approaches which allow for an efficient hardware design. The subsequent sections introduce the following techniques:

1. On-the-fly Coefficient Generation
2. Memory Design
3. Parallelizing Matrix Multiplication
4. Coefficient Generation via Shake128
5. Gaussian Elimination

The presented techniques are flexible in a sense that they can be used in both, high-performance and low-area implementations. The intended purpose is to reduce the barrier for future hardware designs, not only for MAYO but also for UOV schemes in general, as the majority of the approaches presented here can be applied to both.

Table 4.1.: Memory consumption of the $\mathbf{P}_i$ matrices sizes for security levels 1, 3 and 5

Matrix	Dimension	MAYO_1	MAYO_3	MAYO_5
$\mathbf{P}^{(1)}$	$(n-o) \times (n-o)$	58×58 856 B	89×89 2003 B	121×121 3691 B
$\mathbf{P}^{(2)}$	$(n-o) \times o$	58×8 232 B	89×10 445 B	121×12 726 B
$\mathbf{P}^{(3)}$	$o \times o$	8×8 18 B	10×10 28 B	12×12 39 B
$\mathbf{P}$	$n \times n$	66×66 1106 B	99×99 2476 B	133×133 4456 B

4.1. On-the-fly Coefficient Generation

This section describes the on-the-fly coefficient generation design that addresses the large memory demand of MAYO. First, we examine the factors contributing to this demand, followed by the presentation of the on-the-fly generation technique. Subsequent sections analyze the implications of this approach and its effect on the algorithm, offering potential solutions. Finally, we illustrate how this technique allows for the optimization of either performance or memory usage.

Challenge 1: The $\mathbf{P}_{i\in[m]}$ matrices are involved in the majority of the operations and the main reason for the large memory demand of MAYO. As shown in Equation (2.5), each $\mathbf{P}_i$ matrix comprises three submatrices, $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$. The total memory consumption is shown in Table 4.1. The number of matrix elements corresponds to the parameters of the different security levels as listed in Table 3.1. To calculate the size in bytes, we simply multiply the number of elements by the field element size, which is 4 bits for MAYO. It is important to note that $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(3)}$ are in upper triangular form and, thus, the listed byte sizes do not account for the zero elements to give a lower bound for the memory demand. Since there exist m $\mathbf{P}_i$ matrices, the total memory consumption is 69, 232, and 557 KB for the respective security level. Although the memory consumption may appear insignificant in the context of software implementations on modern computers, it poses challenges when considering memory-constrained platforms such as microcontrollers. These memory sizes could potentially cause issues and even render the scheme unimplementable on such platforms.

Design Approach: The large size of the $\mathbf{P}_i$ matrices results in significant memory consumption, making it an important point that needs to be considered during the design phase. It is not feasible to store all the matrices in memory. Thus, we need an approach to reduce the memory demand. $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ are generated pseudo-randomly from the public seed seed_{pk}, as outlined in Section 3.3.1. $\mathbf{P}_i^{(3)}$ is the only matrix in the key generation process that is not directly derived from a seed, as shown in Algorithm 2. Therefore, we need to store it in on-chip memory to avoid the recomputation of $\mathbf{P}_i^{(3)}$,

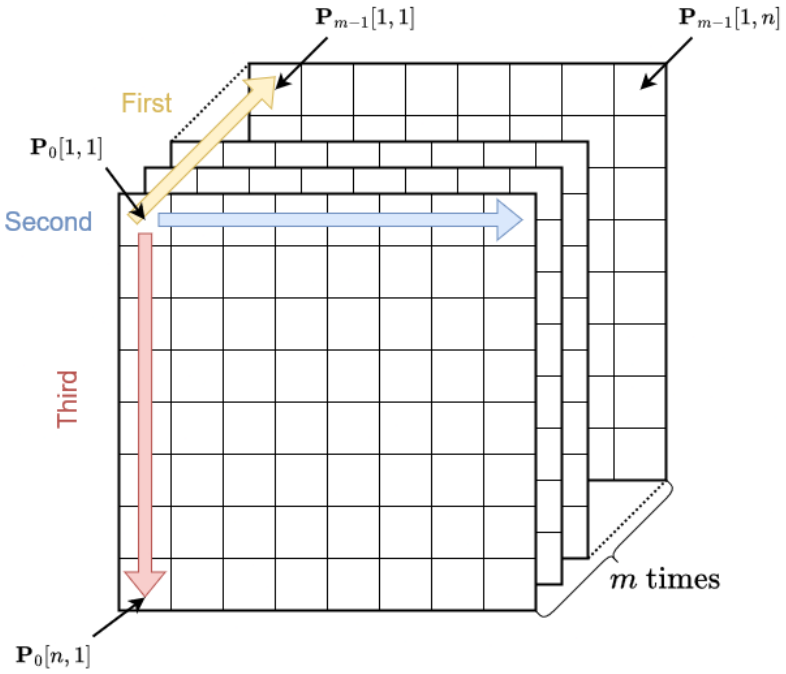

Figure 4.1.: PRNG generation order

however, $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ can be regenerated anytime. Additionally, as shown by Table 4.1, the cost of storing $\mathbf{P}_i^{(3)}$ is significantly less compared to the other two matrices. By storing only $\mathbf{P}_i^{(3)}$, the memory demand can be reduced from 69, 232, and 557 KB to 1152, 2688, and 4992 B, respectively. This corresponds to a reduction in terms of memory by 99 % across all security levels. These numbers underlie the effectiveness of on-the-fly coefficient generation, which reduces the memory demand of each $\mathbf{P}_i$ from $(n \times n)/2$ to only $(o \times o)/2$ elements.

Important Consideration: As a result of this approach, every time some $\mathbf{P}_i^{(1)}$ or $\mathbf{P}_i^{(2)}$ matrix is needed in an operation, the PRNG needs to regenerate the matrices. To ensure that the memory reduction does not compromise the performance, on-the-fly generation needs to be considered during the design of the the arithmetical units. As soon as the PRNG generates coefficients, the arithmetical units should be able to consume the input and carry out the operations. Ideally, the PRNG generates data as fast as the arithmetic logic units (ALUs) can consume them to avoid underutilization. This enables the possibility of customization. Depending on the generation rate of the PRNG and the number of available ALUs, a designer can either aim for a high-performance implementation or a low-area approach. Therefore, we have to identify in which order the coefficients of $\mathbf{P}_i$ are generated and in which operations the matrices are involved to come up with a well-integrated design approach.

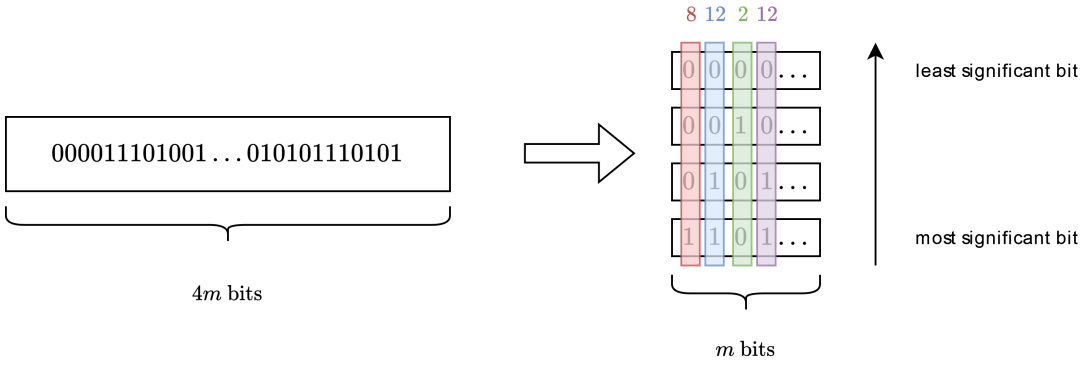

Figure 4.2.: Bitsliced decoding of PRNG output

A hardware implementation should be compliant with the official MAYO specification. Due to this, we need to adhere to the same generation order. In the case of MAYO, the coefficients are generated in m-row-column order, meaning that the PRNG first outputs the first element of all m matrices followed by the next row element of all m matrices. This order is illustrated by Figure 4.1. Additionally, all $\mathbf{P}_i^{(1)}$ are generated before $\mathbf{P}_i^{(2)}$. MAYO uses a bitslicing technique to decode the PRNG output into actual $\mathbb{F}_{16}$ elements. This technique is shown in Figure 4.2. It waits until $4m$ bits have been generated and splits them into 4 packets of length m. Since a $\mathbb{F}_{16}$ element consists of 4 bits, each packet contains a single bit of an element. Subsequently, one bit from each packet is concatenated to form a field element. The most significant bit is located in the fourth packet, hence, the reading order is from bottom to top. In this case, we obtain the elements $(1000)_2 = (8)_{10}$, $(1100)_2 = (12)_{10}$, $(0010)_2 = (2)_{10}$, and $(1100)_2 = (12)_{10}$. Consequently, the 4 packets yield m field elements.

As shown by Algorithms 2, 3, and 5, in the majority of cases $\mathbf{P}_i$ is part of a matrix-matrix or matrix-vector multiplication. The benefit of this operation types is that intermediate result can be computed without knowledge about subsequent values, as depicted by Equation (4.1).

$$\mathbf{c} = \mathbf{ab} = \begin{pmatrix} a_1 & a_2 & a_3 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 b_1 + a_2 b_2 + a_3 b_3 \end{pmatrix} \tag{4.1}$$

The first part of the result, colored in red, can be computed before the blue and orange part. Hence, it is relatively straightforward to combine the on-the-fly generation approach with the involved operations of the algorithm. The PRNG stores the generated coefficients in a buffer where the $\mathbb{F}_{16}$-ALUs consume them. The second operand is retrieved from memory and the intermediate result is stored inside the until a complete

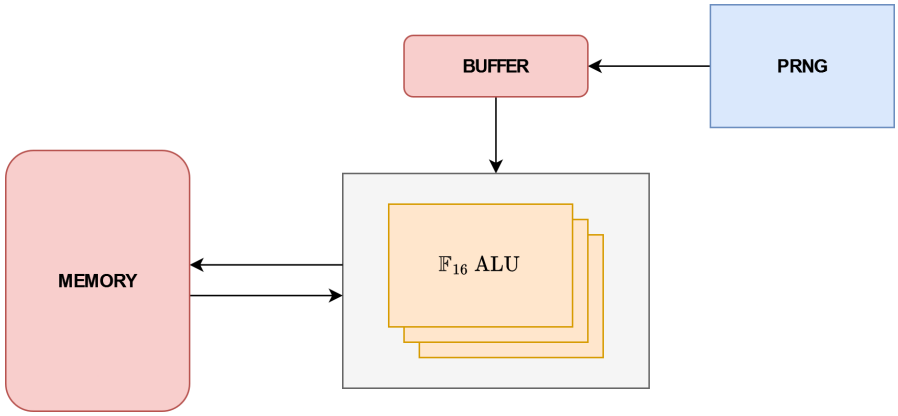

Figure 4.3.: On-the-fly coefficient generation layout

element of the result is computed. The final element is then written back to memory. A simplified design of this approach is shown in Figure 4.3. First, the PRNG generates the bits representing the matrix elements, which are then subject to the decoding process as previously described. Second, after the decoding is completed, the resulting field elements are fed into the buffer, where they await processing by the $\mathbb{F}_{16}$-ALUs. Third, the final computation results are stored in memory for subsequent steps of the respective algorithm. Thus, the buffer needs to have a capacity of at least $4m$ bits to accommodate all elements from a single decoding cycle. Nevertheless, the specific size of the buffer can be selected based on design considerations. A larger buffer enables the utilization of more ALUs, thereby enhancing performance. Conversely, a smaller buffer size reduces the required area since fewer ALUs have to be instantiated.

Although, for this to work, the generated $\mathbf{P}_i$ matrices need to be the left-hand operand, because otherwise, the generation order of the coefficients does not align with the order of matrix multiplication, since the right-hand side is accessed in column order. Terms of type $\mathbf{v}_i^\top \mathbf{P}_a^{(1)} \mathbf{v}_i$ (i.e. Line 21 of Algorithm 3) are not an issue, since matrix multiplication is associative it can be computed as $\mathbf{v}_i^\top (\mathbf{P}_a^{(1)} \mathbf{v}_i)$. However, the computation of $\mathbf{P}_i^{(3)}$ in Line 8 of Algorithm 2 is not as straightforward. Using the distributivity of matrix multiplication we can rewrite it to

$$\mathbf{P}_i^{(3)} = \text{Upper}(-\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^\top \mathbf{P}_i^{(2)}) = \text{Upper}(-\mathbf{O}^\top (\mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{P}_i^{(2)})). \tag{4.2}$$

We want to avoid that the generated matrix is on the right-hand side of a matrix multiplication. Using this modification eliminates that. As shown by Equation 4.2, the original computation comprises six suboperations: three matrix multiplications, one matrix subtraction, one negation, and one upper triangulation. It is notable that multiplication with $\mathbf{O}^\top$ occurs twice. Hence, this can be reduced to a single multiplication by factoring out the multiplication of $\mathbf{O}^\top$. This adjustment, omitting one matrix multiplication,

saves a total of $m(o \times (n-o))(n-o)$ multiplications and $m(o \times (n-o))(n-o-1)$ additions, thereby enhancing the performance of key generation.

The last location, where on-the-fly generation can not be applied in a straightforward manner is located in Line 11 of Algorithm 5. In the computation of $\mathbf{u}$, the complete matrix $\mathbf{P}_a = \begin{pmatrix} \mathbf{P}_a^{(1)} & \mathbf{P}_a^{(2)} \\ 0 & \mathbf{P}_a^{(3)} \end{pmatrix}$ is required. However, the PRNG generates $\mathbf{P}_a^{(1)}$ before $\mathbf{P}_a^{(2)}$ as described before, and, thus, we do not receive complete rows of the $\mathbf{P}_a$ matrices. Consequently, we are not able to directly feed the $\mathbb{F}_{16}$-ALUs with the matrix coefficients. To cope with this issue, we perform block matrix multiplication by dividing $\mathbf{P}_a$ and $\mathbf{s}_i$ into blocks of a certain dimension such that it adheres to the generation order. As a result, $\mathbf{h} = \mathbf{P}_a \mathbf{s}_i$ is computed using

$$\begin{aligned} \mathbf{a} &= \mathbf{P}_a^{(1)} \mathbf{s}_i[0 : n-o] \\ \mathbf{b} &= \mathbf{P}_a^{(2)} \mathbf{s}_i[n-o : n] \\ \mathbf{c} &= \mathbf{P}_a^{(3)} \mathbf{s}_i[n-o : n] \end{aligned} \tag{4.3}$$

$$\mathbf{h} = \begin{pmatrix} \mathbf{a} + \mathbf{b} \\ \mathbf{c} \end{pmatrix}.$$

Since $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are the results of matrix-vector multiplications, we only need to store three vectors with sizes $n-o$ and o as intermediate results, which is easily possible.

Challenge 2: To preserve the memory saving effect of on-the-fly generation, the intermediate results have to be smaller than the involved $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices. This is obvious by taking a look at Line 6 of Algorithm 3. It is not reasonable to compute $\mathbf{M} = \mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top}$ using on-the-fly generation and then storing the intermediate result $\mathbf{M}$. Since $\mathbf{P}_i^{(1)}$ is upper triangular and $\mathbf{P}_i^{(1)^\top}$ is lower triangular, each matrix consists of $(n-o)(n-o-1)/2$ elements. Therefore, $\mathbf{M}$ would contain $(n-o)(n-o)$ elements and need twice the memory of $\mathbf{P}_i^{(1)}$. This massive increase in required memory would annihilate the benefits of our memory saving technique.

Design Approach: To reduce the size of intermediate values, the $\mathbf{P}_i$ matrices have to be multiplied with a smaller matrix to obtain a result with smaller dimensions. Most importantly, we need to avoid additions involving only plain $\mathbf{P}_i$ matrices. In the current example, we are able to take advantage of matrix product distributivity and modify Line 6 to

$$\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top})\mathbf{O} + \mathbf{P}_i^{(2)} = \underbrace{\mathbf{P}_i^{(1)}\mathbf{O}}_{\mathbb{F}_{16}^{(n-o)\times o}} + \underbrace{\mathbf{P}_i^{(1)^\top}\mathbf{O}}_{\mathbb{F}_{16}^{(n-o)\times o}} + \mathbf{P}_i^{(2)}. \tag{4.4}$$

While $\mathbf{P}_i^{(1)}\mathbf{O}$ and ${\mathbf{P}_i^{(1)}}^\top\mathbf{O}$ consist of $(n-o)\times o$ elements each, they require significantly less memory than $\mathbf{P}_i^{(1)} + {\mathbf{P}_i^{(1)}}^\top$. o is smaller than $n-o$ by at least a factor of 7 across all security levels. For MAYO_1, this results in storing 2×464 instead of 1×3364 matrix elements, reducing the the required memory from 13 KB to 3.6 KB.

To achieve the maximum memory reduction, the scheme needs to be adapted to minimize the storage of intermediate values and exploit properties of matrix operations to compress them efficiently. Lines 17 through 26 of Algorithm 3 provide a good example of this approach. The computations of $\mathbf{y}$ and $\mathbf{A}$ are independent, thus, their intermediate values do not need to be held in memory simultaneously. A memory-efficient adaption of this section is shown in Algorithm 6.

Algorithm 6 Adapted Signature Computation [Beu+23a]

17: //Build right-hand side vector $\mathbf{y}$ of linear system
18: **for** i from 0 to $k-1$ **do**
19: **for** j from $k-1$ to i **do**
20: $\mathbf{u} \leftarrow \begin{cases} \{\mathbf{v}_i^\top \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a\in[m]} & \text{if } i = j \\ \{\mathbf{v}_i^\top \mathbf{P}_a^{(1)} \mathbf{v}_j + \mathbf{v}_j^\top \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a\in[m]} & \text{if } i \neq j \end{cases}$
21: $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}^l\mathbf{u}$
22:
23: //Build left-hand side matrix $\mathbf{A}$ of linear system
24: **for** i from 0 to $k-1$ **do**
25: $\{\mathbf{M}_i[j,:]\}_{j\in[m]} \leftarrow \{\mathbf{v}_i^\top \mathbf{L}_j\}_{j\in[m]}$
26: **for** i from 0 to $k-1$ **do**
27: **for** j from $k-1$ to i **do**
28: $\mathbf{A}[:, i*o : (i+1)*o] \leftarrow \mathbf{A}[:, i*o : (i+1)*o] + \mathbf{E}^l\mathbf{M}_j$
29: **if** $i \neq j$ **then**
30: $\mathbf{A}[:, j*o : (j+1)*o] \leftarrow \mathbf{A}[:, j*o : (j+1)*o] + \mathbf{E}^l\mathbf{M}_i$
31: $l \leftarrow l+1$
32: $\mathbf{x} \leftarrow \text{SampleSolution}(\mathbf{A}, \mathbf{y}, \mathbf{r})$

By separating the computation of $\mathbf{A}$ and $\mathbf{y}$ we are able to reduce the memory needed for intermediate values significantly. After the final result of $\mathbf{y} \in \mathbb{F}_q^m$ is obtained, we only need to store its m elements, since none of the depended variables are shared with $\mathbf{A}$. Consequently, we can use the freed up memory for the computation of the linear system matrix. A significant advantage of this separation is that it incurs minimal performance cost. The only added overhead is the inclusion of an extra for loop with identical limits, which has negligible impact on performance.

To this point on-the-fly generation was used to decrease the memory requirements of MAYO. Nevertheless, this approach comes with a performance trade-off. Typically, retrieving coefficients from on-chip-memory is faster than regenerating them repeatedly. However, this is not an feasible option due to the large size of the $\mathbf{P}_i$ matrices.

Nonetheless, we can sacrifice a small amount of the saved memory to compensate for this performance loss. By applying precomputations at certain locations in signature computation and signature verification, we increase the number of intermediate variables in memory, but enable a performance increase in exchange. For instance, $\mathbf{g}_i = \mathbf{P}_a^{(1)}\mathbf{v}_i$ (Line 21 of Algorithm 3) and $\mathbf{h}_i = \mathbf{P}_a\mathbf{s}_i$ (Line 11 of Algorithm 5) are computed repeatedly. Consequently, the PRNG has to regenerate the matrix coefficients several times for the same operation, leading to multiple executions of matrix multiplication sharing identical operands. This can be avoided by precomputing $\mathbf{g}_i$ and $\mathbf{h}_i$ for all $i \in [k]$ and storing them in memory. Using verification as an example, the adapted version would take the form shown in Algorithm 7.

Algorithm 7 Adapted Signature Verification [Beu+23a]

9: **for** i from 0 to $k-1$ **do**
10: $\quad \mathbf{h}_i \leftarrow \{\mathbf{P}_a\mathbf{s}_i\}_{a\in[m]}$
11:
12: **for** i from 0 to $k-1$ **do**
13: $\quad$ **for** j from $k-1$ to i **do**
14: $\quad\quad \mathbf{u} \leftarrow \begin{cases} \mathbf{s}_i^\top \mathbf{h}_i & \text{if } i = j \\ \mathbf{s}_i^\top \mathbf{h}_j + \mathbf{s}_j^\top \mathbf{h}_i\} & \text{if } i \neq j \end{cases}$
15: $\quad\quad \mathbf{y} \leftarrow \mathbf{y} + \mathbf{E}^l\mathbf{u}$
16: $\quad\quad l \leftarrow l + 1$

Using this adaption increases the required on-chip storage for the $\mathbf{h}_i$ variable by a factor of k, since the values for all $i \in [k]$ are stored concurrently. A single $\mathbf{h}_i$ is of dimension $m \times n$, thus, the precomputation adaption needs approximately 18, 50, and 100 KB instead of 2, 4.6, and 9.3 KB for the respective security levels. This is a significant increase in memory demand, but on the other hand, we save $mk(k-1)$ matrix-vector multiplications. Therefore, the performance of this operation is increased by 4608, 10560, and 16896 times the latency of a single matrix-vector multiplication. This example illustrates the flexibility of on-the-flight generation in regard to performance and memory demand.

To conclude, on-the-fly generation enables the implementation of MAYO on hardware platforms. Otherwise, the memory of constraint platforms would already be exhausted by the public key matrices $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$. Nevertheless, developers retain the ability to make decisions regarding their design choices. On one hand, they can prioritize low memory usage and minimize the storage for intermediate results. On the other hand, they have the option to increase memory and enhance the performance instead.

4.2. Memory Design

This section describes a potential memory design aimed at offering a low-latency solution for accessing memory elements in MAYO. A well-designed memory layout is one of the key factors for an efficient hardware implementation. First, we introduce two distinct data formats and elaborate on their identification process. Then, we describe the impact of these formats on various types of operations. Lastly, we present a memory layout that demonstrates how to implement the memory formats on a FPGA utilizing Block RAMs (BRAMs).

Challenge: The majority of arithmetic operations in MAYO involves the evaluation of the multivariate quadratic map P. This map comprises the three submatrices $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$. Since P consists of m multivariate quadratic polynomials, evaluating the map involves the multiplication of all m $\mathbf{P}_i$ matrices. Although the elements of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ are generated on-the-fly, $\mathbf{P}_i^{(3)}$ and subsequent computation results have to be retained in memory. Hence, the memory design needs to support fast loading and storing of these elements. Thus, using an appropriate format for matrices and vectors is a decisive factor in the design. Otherwise, accessing and storing elements has a negative impact on the performance of the hardware implementation.

Design approach: We need to analyze the MAYO scheme and its algorithms to understand the memory requirement. The scheme's computations mainly evolve around the $\mathbf{P}_i$ matrices making them a crucial point when it comes to hardware design. Additionally, the $\mathbf{P}_i$ matrices employ the largest amount of elements. Therefore, our memory layout needs to focus on accelerating computations involving the $\mathbf{P}_i$ matrices. An important detail is that all of the m $\mathbf{P}_i$ are used every time. There is no instance where a single $\mathbf{P}_i$ is involved without the others. Consequently, all m matrices share the second operand. Therefore, it is necessary to store all m coefficients with identical indices close to each other, as they are simultaneously used in the same operation. This fact leads to our memory design. We introduce two different formats to store vectors and matrices, defined as followed:

1. **Unpacked**: One entry of a BRAM holds an entire vector $\mathbf{v}$ or a row of a matrix $\mathbf{M}$. This format is mainly used during the computation of the matrix $\mathbf{A}$ in Algorithm 3. Therefore, we are able to load a whole vector or matrix row from memory in a single cycle.

2. **Packed**: Each BRAM entry holds m elements $\mathbf{P}_i[x, y]$ for $i \in [m]$. $\mathbf{P}_i[x, y]$ is the coefficient located at row x and column y of the i-th $\mathbf{P}$ matrix. Thus, loading one entry provides us with m elements from m different matrices, all sharing the identical index. This format is used whenever a matrix of the form $\mathbf{P}_i$ is involved. This choice is motivated by the fact that the same operation needs to be applied to elements with identical indices in each $\mathbf{P}_i$ matrix. Therefore, subsequent BRAM entries contain the elements of all m matrices in row-major order. Figure 4.4 illustrates the packed memory format of the $\mathbf{P}_i$ matrices. One can interpret the packed

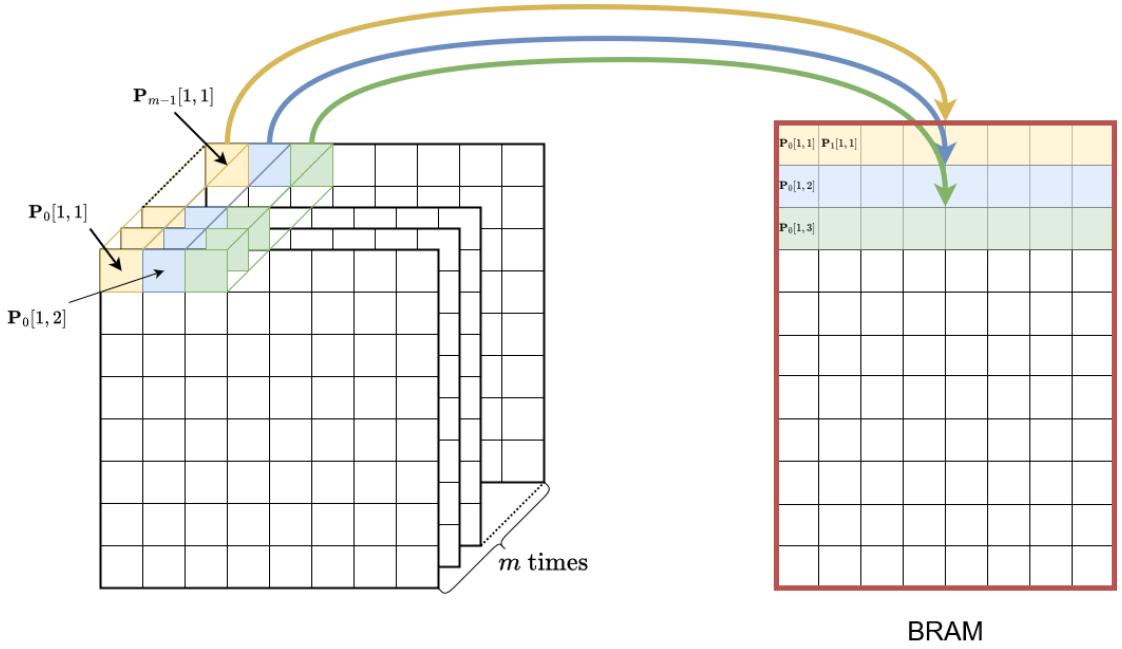

Figure 4.4.: Packed Format

memory format as storing a single matrix entry per memory location, but instead of a 4-bit element we store larger elements of size $4m$. This interpretation also aligns with the involved operations since all m elements are used simultaneously. As an additional benefit, the row order inside the BRAM is not relevant, since we can load the individual elements in any desired order.

Consequently, these memory formats impact the arithmetical operations of the scheme. Now, also the format has to be considered. Three different operation types can be identified:

1. **Packed-Unpacked Multiplication**: An element in packed format is multiplied with an element in unpacked format (e.g. $\{\mathbf{P}_i^{(1)}\mathbf{O}\}_{i\in[m]}$). Since the right-hand side operand is unpacked, all m elements of a single memory location are multiplied with the same value of the unpacked variable. Thus, we are able to load all the relevant elements to start with the multiplication in a single cycle. Due the row-major order of the packed format, we only need to access the memory row by row to adhere to the order of matrix multiplication The obtained result is stored in packed format.

2. **Unpacked-Packed Multiplication**: An element in unpacked format is multiplied with an element in packed format (e.g. $-\mathbf{O}^\top(\mathbf{P}_i^{(1)}\mathbf{O}+\mathbf{P}_i^{(2)})$). Similar to the packed-unpacked multiplication, all m elements of a single memory location are

multiplied with the same value of the unpacked variable and we are able load these elements within one cycle. However, the packed variable is on the right-hand side of the operation and we need to retrieve the elements in column order. However, since one BRAM entry stores only one element, the memory access order is not important. Instead of retrieving subsequent BRAM entries, we need to compute the correct index using the number of columns.

3. **Packed-Packed Addition**: An element in packed format is added to an element in packed format (e.g. $\mathbf{P}_i^{(1)}\mathbf{O} + \mathbf{P}_i^{(2)}$). This operation type is the easiest to implement. Since matrix-matrix addition is only defined for matrices of the same dimension, we only have to iterate over the memory entries of both matrices and add the respective values.

The access pattern to load packed variables in row-major and column-major order is illustrated in Figure 4.5. This pattern enables us to perform computations with packed variables either on the left or right-hand side and, additionally, the transposition of matrices with no additional cost.

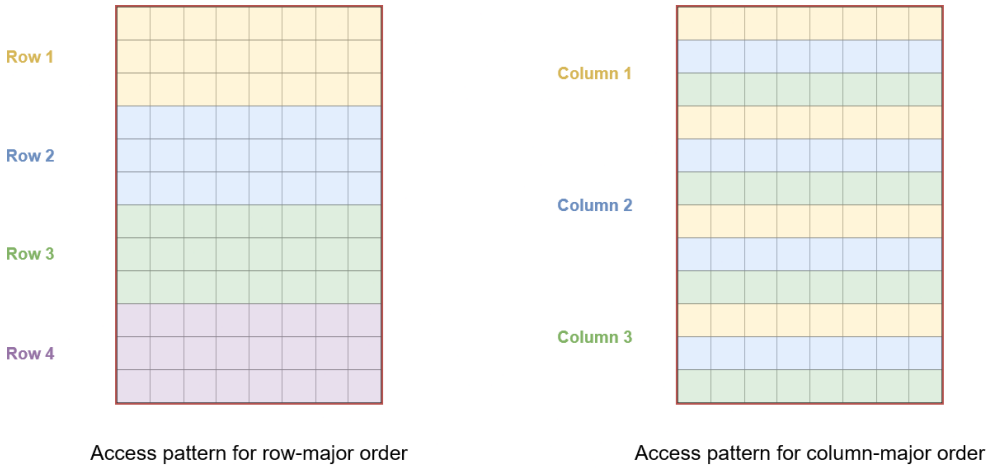

Figure 4.5.: Access pattern for loading matrices in row-major and column-major order

In contrast to the packed format, transposition and column-major order access of unpacked variables poses a challenge, since an entire matrix row is stored in a single BRAM entry. While loading a row only accounts for one cycle, retrieving a column requires as many cycles as there are matrix rows. However, analysis of the algorithms shows that the only variables in unpacked format besides $\mathbf{A}$ are $\mathbf{O}$, $\mathbf{v}_i$, and $\mathbf{s}_i$. These variables are of dimension $(n-o) \times o$, $(n-o)$, and n, respectively. Thus, their total memory demand is 294, 540, and 854 B across all security levels. Additionally, not all three are used in the same algorithms. $\mathbf{v}_i$ is needed only in Algorithm 3 and $\mathbf{s}_i$ only in Algorithm 5. Thus, it is feasible to employ a data cache for this three variables utilizing registers instead of storing them in BRAM. This allows us to access all elements of these variables within one cycle and eliminates the overhead of transposition and column order access.

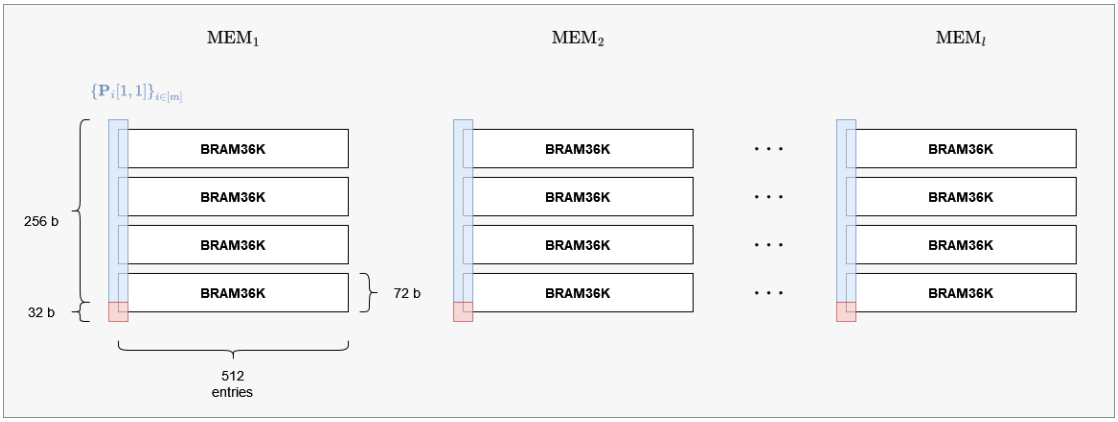

Figure 4.6.: Memory layout for MAYO$_1$

To support the two formats we need to align our BRAMs accordingly. We target the Xilinx 7 Series for the following description, nevertheless this design approach can be applied to different FPGAs by adapting to the specific Block RAM size. The Xilinx 7 Series comes with two different BRAM sizes, 18 Kb and 36 Kb. The 36 Kb BRAM supports a maximum width of 72 bits and a maximum depth of 512. The 16 Kb BRAM allows a width of up to 36 bits and a depth of 512 [Xil19]. As described before, we spread m 4-bit values across the width of the BRAMs to store and load all of them within on cycle. However, m is too large to store the values in a single BRAM. The maximum width of 72 is already exceeded in security level 1 with $4m = 256$. Thus, we need to employ multiple BRAMs to implement our packed format. For every security level we need to align $\lceil 4m/72 \rceil$ BRAMs to store all m values in a single entry. To accommodate a packed entry for MAYO$_1$, 4 BRAMS in width are required. We denote such a block of $\lceil 4m/72 \rceil$ BRAMs as memory bank. The required number of banks depends on the amount of intermediate values and the design choice of the developer. As described in Section 4.1, one can aim for a high-performance or a low-area implementation.

Our memory design for the security level 1 configuration is depicted in Figure 4.6. The design comprises a total of l memory banks, denoted as MEM$_{i\in[l]}$, responsible for storing packed data during the operations involved in MAYO. The value of l depends on the number of intermediate vales as mentioned before. Each memory bank entry contains one element in packed format spread across the vertically arranged BRAMs, as highlighted in blue. The decision if 36K or 18K BRAMS are used depends on the specific security level. The objective is to approach the required entry width as closely as possible. In MAYO$_1$, a single packed element consists of 256 bits and each memory bank is able to store elements of up to 4×72 bits. Thus, the last 32 bits of every memory bank entry are unused, as marked in red. However, this overhead is reasonable and cannot be easily avoided. Up to this point, we focused on the storage of packed variable. We leave a thorough discussion of the memory design for unpacked variables to Section 4.5, as

this format is primarily used in GaussianElimination and SampleSolution. Both of these algorithms are addressed in that section.

In conclusion, our memory design aims to minimize the latency associated with storing and loading matrix and vector elements. It introduces two memory formats, denoted as packed and unpacked, to handle the m-fold matrices $\mathbf{P}_i$ and the outcomes of their respective operations. The design provides flexibility by allowing developers to choose the number of memory banks, thereby either enhancing the performance or reducing the memory demand.

4.3. Parallelizing Matrix Multiplication

This section describes an ALU design to carry out matrix-vector and matrix-matrix multiplications. The proposed design is fully compatible with the previously presented on-the-fly generation and the memory design. First, we analyze matrix multiplication to come up with a design for the responsible ALU. Then, we describe how this ALUs can be grouped together to deal with the packed format. Finally, we describe how to further parallelize the matrix multiplications to increase the performance of the design.

Challenge 1: The majority of operations in MAYO are matrix-matrix and matrix-vector multiplications involving a large number of elements. Thus, the performance of a hardware implementation highly depends how fast these operations can be carried out. In addition to this, elements are not only retrieved from memory, as in the case of intermediate values, but are also generated pseudo-randomly. As a result, the employed ALUs have to be designed carefully to operate well in these different scenarios. Otherwise, the overall performance of a hardware implementation is compromised.

Design Approach: We need to analyze matrix-matrix and matrix-vector multiplications to understand the requirements for an arithmetical unit design. Furthermore, we have to consider on-the-fly generation and the memory design to align the ALU accordingly with these two optimizations. Equation 4.5 shows an example of a matrix-matrix multiplication.

$$\underbrace{\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{pmatrix}}_{\mathbf{A}} \times \underbrace{\begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{pmatrix}}_{\mathbf{B}} = \underbrace{\begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}}_{\mathbf{C}} \tag{4.5}$$

To compute an element of matrix $\mathbf{C}$, the respective row of matrix $\mathbf{A}$ is multiplied with the respective column of matrix $\mathbf{B}$. For instance, $c_{2,1}$ is the result of the second row of $\mathbf{A}$ times the first column of $\mathbf{B}$. Thus, matrix-matrix multiplications as well as matrix-vector multiplication can be broken down to a number of vector-vector multiplications. Equation 4.6 shows this interpretation in more detail.

$$\mathbf{Ab} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_{1,1}b_1 + a_{1,2}b_2 + a_{1,3}b_3 \\ a_{2,1}b_1 + a_{2,2}b_2 + a_{2,3}b_3 \\ a_{3,1}b_1 + a_{3,2}b_2 + a_{3,3}b_3 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \mathbf{c} \tag{4.6}$$

The elements of $\mathbf{c}$ can be computed completely independent of each other. The row colored in red only influences c_1, while the blue and orange rows only influence c_2 and c_3, respectively. Additionally, we see how vector-vector multiplication is carried out. The subsequent elements of a row and a column are multiplied and then added together until a complete element of $\mathbf{c}$ is computed. Thus, vector-vector multiplication is essentially a multiply-and-accumulate (MAC) operation. This enables us to come up with a design for our $\mathbb{F}_{16}$-ALUs. The computations colored in red are accumulated within a MAC unit until all elements of the corresponding row in $\mathbf{A}$ are consumed. This process is then repeated for each row in $\mathbf{A}$, which is highlighted in blue and orange.

The required components for this procedure are only one adder, one multiplier, and one register to store the accumulation result. However, the matrix elements are in $\mathbb{F}_{16}$ and, thus, the addition and multiplication needs to be implemented as described in Section 2.2. An illustration of the outlined $\mathbb{F}_{16}$-MAC ALU is shown in Figure 4.7. The multiplier consumes one element of each side of the multiplication and feeds the result to the adder. The adder accumulates the multiplication and the intermediate result from the register until the final element is obtained.

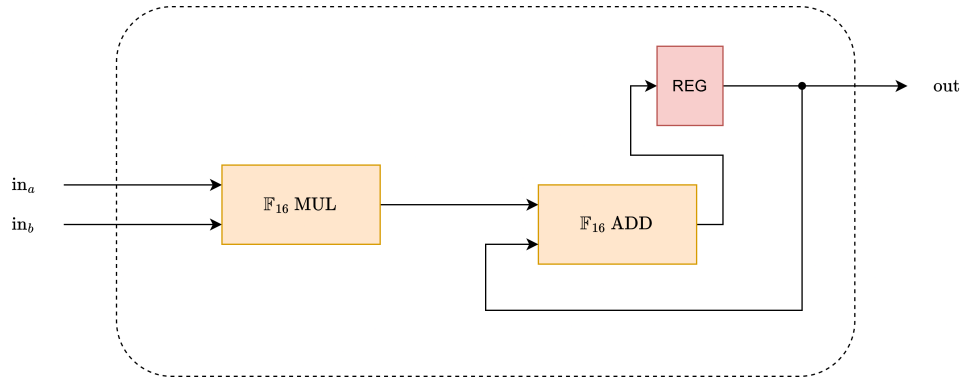

Figure 4.7.: $\mathbb{F}_{16}$ MAC unit

This ALU design aligns well with the on-the-fly generation and the memory design described in Section 4.1 and Section 4.2. Both, the element generation of the PRNG and the storage in memory, follow a row-major order. Additionally, we modified the scheme such that generated matrices are always on the left-hand side of a computation. Therefore, the sequence in which the ALU consumes the elements is in accordance with on-the-fly generation and the memory design. This is demonstrated once more by Equation 4.6, where the elements utilized in the computation of $\mathbf{c}$ precisely match the elements of $\mathbf{A}$ in row-major order.

Challenge 2: The proposed ALU is compatible with the memory design and the generation order of matrix elements. However, one ALU is only able to compute a single element of the desired result but we receive m elements in a single load cycle from the BRAMs. Therefore, if only one ALU is used we would need to iterate over the memory entry m times, resulting in poor performance and extra logic since the ALU needs to select the individual elements according to their index. Thus, the consumption of one BRAM entry using a single $\mathbb{F}_{16}$-ALU requires at least m cycles. As a result, we would encounter a strong underutilization. Furthermore, the advantages of our memory design cannot be fully utilized and the performance as a whole is decreased.

Design Approach: One $\mathbb{F}_{16}$-ALU module is relatively cheap. We only need a combination of XOR and AND operations to implement the field arithmetic described in Section 2.2. Therefore, we can enhance the performance by enabling simultaneous computations through the instantiation of multiple units and utilizing the parallelism capabilities of hardware platforms. Since the memory design provides us with m elements within one cycle, a straightforward design choice is to instantiate m $\mathbb{F}_{16}$-ALUs. This allows us to consume all elements within one cycle and we introduce no additional latency. This blends in well also with the on-the-fly generation, since the PRNG generates more than one element at once. We only have to wait until m elements are stored in the PRNG buffer to start with the computation. As a result, we are able to reduce the latency for the computation of matrix and vector multiplications by a factor of m compared to a single ALU approach. For the remaining part of this section we denote the group of m $\mathbb{F}_{16}$ modules as ALU block.

The majority of multiplications in the MAYO scheme are of the packed-unpacked type, for instance $\mathbf{P}_i^{(1)}\mathbf{O}$. We take this matrix-matrix multiplication as example to give an estimate for the latency using the presented ALU block.

$$\mathbf{P}_i^{(1)}\mathbf{O} = \underbrace{\begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n-o} \\ \vdots & \vdots & & \vdots \\ p_{n-o,1} & p_{n-o,2} & \cdots & p_{n-o,n-o} \end{pmatrix}}_{(n-o)\times(n-o)} \underbrace{\begin{pmatrix} o_{1,1} & \cdots & o_{1,o} \\ o_{2,1} & \cdots & o_{1,1} \\ \vdots & & \vdots \\ o_{n-o,1} & \cdots & o_{n-o,o} \end{pmatrix}}_{(n-o)\times o} \tag{4.7}$$

$$= \begin{pmatrix} p_{1,1}o_{1,1} + \cdots + p_{1,n-o}o_{n-o,1} & \cdots & p_{1,1}o_{1,o} + \cdots + p_{1,n-o}o_{n-o,o} \\ \vdots & & \vdots \\ p_{n-o,1}o_{1,1} + \cdots + p_{n-o,n-o}o_{n-o,1} & \cdots & p_{n-o,1}o_{1,o} + \cdots + p_{n-o,n-o}o_{n-o,o} \end{pmatrix}$$

Our ALU block consumes one element of each $\mathbf{P}_i^{(1)}$ matrix simultaneously as outlined in the paragraph before. Equation 4.7 shows that the first row of $\mathbf{P}_i^{(1)}$, colored in red, is multiplied with every column of $\mathbf{O}$. Thus, we need to hold the generated row in memory

until the ALU block processed all the columns of $\mathbf{O}$. As a result, the latency to compute a full row of $\mathbf{P}_i^{(1)}\mathbf{O}$ is $o \times latency_{\text{ALU}}$. However, this enables us to further parallelize matrix-matrix multiplication. The matrix $\mathbf{O}$ is stored in a data cache as described in Section 4.2. Thus, we are able to access the elements of $\mathbf{O}$ instantly. The left-hand side of the multiplication is shared across all columns. Therefore, we can instantiate o more ALU blocks to compute the final elements of all columns concurrently. This new approach allows simultaneous computation of multiple columns and, thus, reduces the latency for a full matrix-matrix multiplication by a factor of o to $(n-o) \times latency_{\text{ALU}}$. It is the responsibility of the developer to determine the specific number of ALU blocks. Employing separate blocks for each column maximizes the performance, while processing the columns sequentially aims for a design with reduced area requirements.

To conclude, our ALU design aims to minimize the latency associated with performing matrix multiplications while maintaining the compatibility with on-the-fly generation and the memory design. We introduced two parallelization approaches: first, grouping m $\mathbb{F}_{16}$-ALUs to efficiently manage the packed format by enabling simultaneous computation, and second, concurrent processing of multiple columns to improve performance.

4.4. Coefficient Generation via SHAKE128

This section describes how to improve the performance of on-the-fly generation and at the same time lower the area demand by substituting AES with SHAKE. First, we explain the rationale behind this substitution and its advantages. Subsequently, we address the challenge associated with using SHAKE instead of AES. Finally, we introduce a novel type of arithmetic unit designed to overcome this challenge and discuss potential opportunities for parallelization.

Challenge 1: The performance of the scheme is highly dependent on the generation speed of the employed PRNG. Furthermore, the limited area of hardware platforms restricts the simultaneous utilization of different cryptographic primitives. The newest specifications of MAYO use AES and SHAKE, which can present challenges in hardware implementations related to both area usage and performance.

Design Approach: In the first MAYO paper [Beu22b], SHAKE128 is used exclusively for hashing and pseudo-random coefficient generation. In [Beu+23a] this was changed to AES-128-CTR for the generation of the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ elements while SHAKE256 is employed for computing the message digest and generating $\mathbf{O}$. The reasoning behind this modification is to use AES-128 for the primary data generation tasks. Thus, the fast AES-NI instruction of modern CPUs can be applied to enhance the performance of the scheme. A detailed explanation of AES-NI and the resulting optimization is available in Section 2.8 and Section 3.7. However, this specification change presents challenges in hardware implementations as previously indicated, specifically:

1. **Area usage**: To follow the newest MAYO specifications in a hardware implementation, it is necessary to integrate two cores, one for AES-128 and another for SHAKE128. Consequently, a significant portion of the area requirement would stem from these two cores. Given that both primitives serve the same purpose of generating pseudo-random values, this method introduces redundancy.
2. **Performance**: The advantage of AES in a software implementation is based on the acceleration provided by the NI extension. However, hardware platforms cannot benefit from this instruction set. Thus, the reasoning for this change is no longer applicable. Additionally, our tested SHAKE core performs significantly better compared to AES. Specifically, SHAKE128 generates 1344 bits every 26 cycles compared to AES-128-CTR with 128 bits every 12 cycles. This results in a speedup by a factor of 4.84. Therefore, the transition to AES not only fails to improve performance but actually slows down the scheme on hardware platforms.

Hence, using SHAKE exclusively for random data generation can enhance the scheme's performance on hardware and also decrease the area requirement of the implementation. We decided to use SHAKE128 to generate the elements of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$, while maintaining SHAKE256 in the other cases as specified in the latest specification. Thus, we are able to benefit of the faster SHAKE128 in the generation of the large $\mathbf{P}_i$ matrices, while adhering closely to the original version. Importantly, this has no impact on the area demand since both utilize KECCAK, as described in Section 2.9.

This modification becomes even more important in combination with on-the-fly generation, as it significantly increases the frequency of PRNG usage. Consequently, the speedup attained through SHAKE has a substantial impact on the overall performance of the hardware implementation. Although, it is important to note that SHAKE has a drawback in case of software implementations, as the optimizations outlined at the beginning of the design approach can no longer be applied. Yet, we present a solution to tackle this problem by introducing an AVX2-based SHAKE adaption in Chapter 5.

Challenge 2: The transition from AES to SHAKE is fully compatible with the hardware optimizations presented in this chapter except one case. Unlike with AES, a challenge arises in the computation of $\mathbf{L}_i$ in Line 6 of Algorithm 3. We transformed the original definition to $\mathbf{L}_i = \mathbf{P}_i^{(1)}\mathbf{O} + \mathbf{P}_i^{(1)^\top}\mathbf{O} + \mathbf{P}_i^{(2)}$ to ensure compatibility with on-the-fly generation as outlined in Section 4.1. Computing $\mathbf{P}_i^{(1)^\top}\mathbf{O}$ poses a challenge when using SHAKE. One major difference between AES and SHAKE is the reason for this. In AES individual keystream blocks can be generated independently by controlling the counter. However, this is not possible with SHAKE, as subsequent output blocks depend on each other. In the case of AES, generating $\mathbf{P}_i^{(1)^\top}$ is feasible by setting the counter appropriately, resulting in generation of the matrix in its transposed form. With SHAKE, we are limited to the row-major generation order and cannot directly obtain $\mathbf{P}_i^{(1)^\top}$.

Design Approach: As we are bound to the standard generation order of the matrix,

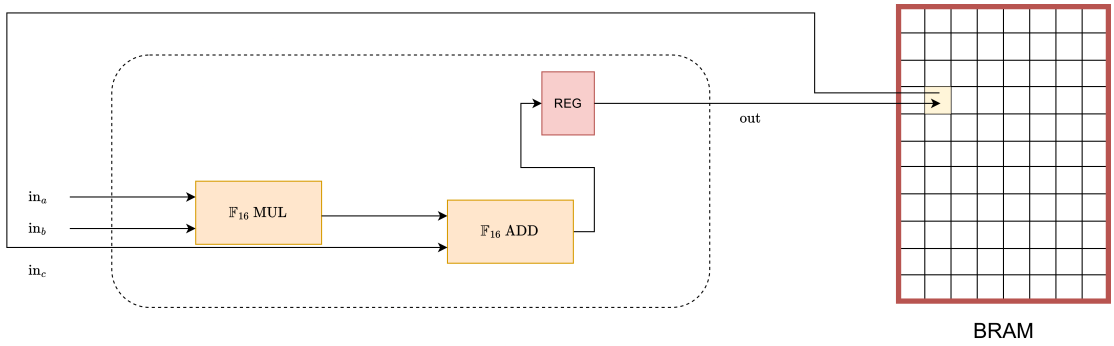

Figure 4.8.: $\mathbb{F}_{16}$ BMAC unit

we need to modify the computation of $\mathbf{P}_i^{(1)\top}\mathbf{O}$ to make it compatible with SHAKE. Therefore, we analyze the operation similar to Equation 4.6 by transposing the left-hand side matrix and observe how the computation of the final elements changes.

$$\mathbf{A}^\top\mathbf{b} = \begin{pmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_{1,1}b_1 + a_{2,1}b_2 + a_{3,1}b_3 \\ a_{1,2}b_1 + a_{2,2}b_2 + a_{3,2}b_3 \\ a_{1,3}b_1 + a_{2,3}b_2 + a_{3,3}b_3 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \mathbf{c} \tag{4.8}$$

As shown by Equation 4.8, the elements of $\mathbf{A}^\top$ are not consumed in row-major order anymore. The generation sequence is indicated by the colors, first, the red values are generated by the PRNG from top to bottom followed by the blue and orange ones. Thus, subsequent values contribute to the whole result vector and are not isolated within a single entry anymore. As a result, we are not able to feed our ALU blocks directly with the generated data and receive a complete element after processing a whole row, since the necessary elements for computing a single c_i are no longer generated one after another.

Therefore, we have to design a modified $\mathbb{F}_{16}$-ALU to support multiplication of a transposed matrix without actually transposing it. Due to the fact that subsequent elements to not belong to the same c_i, we are not able to accumulate the intermediate values inside the arithmetical unit. Instead, we can store the intermediate results for every result entry inside a BRAM until the next element is generated and proceed with the accumulation. For instance, we compute $a_{1,1}b_1$ and store its result in a BRAM. When $a_{2,1}$ is generated, we retrieve $a_{1,1}b_1$ from memory and continue with the computation of c_1. We denote this approach as BMAC. An illustration of the outlined $\mathbb{F}_{16}$-BMAC ALU is shown in Figure 4.8.

The multiplier consumes one element of each side of the multiplication and feeds the result to the adder. However, the second input for the addition is not retrieved from

the register as in the case of standard MAC but is loaded from a BRAM. Then, the adder accumulates the multiplication and the retrieved intermediate result and stores it in BRAM again. This process is repeated until the final element is obtained. Ideally, we allocate the appropriate BRAM entry already for the computation to avoid any relocating afterwards. This means that the BRAM entry used to store the intermediate results should align with where the final result should be located.

BMAC offers the same parallelization possibilities as the standard MAC operation described in Section 4.3. We are able to combine m BMAC-ALUs to a block to compute ${\mathbf{P}_i^{(1)}}^\top \mathbf{O}$ for all $i \in [m]$ simultaneously. Additionally, we can also instantiate o BMAC blocks to obtain the elements of all columns of $\mathbf{O}$ in parallel. One advantage of BMAC is that $\mathbf{P}_i^{(1)}\mathbf{O}$ and ${\mathbf{P}_i^{(1)}}^\top \mathbf{O}$ consume the elements in the same order. Thus, we can decide if we integrate BMAC in the standard $\mathbb{F}_{16}$-ALU or instantiate it separately to compute the two matrix multiplications in parallel. A very important advantage of this approach is that we would save one full PRNG round by avoiding the need to regenerate the $\mathbf{P}_i^{(1)}$ elements a second time. Therefore, we are able to enhance the performance even more.

In conclusion, incorporating Shake significantly enhances the scheme's performance, particularly when coupled with on-the-fly generation. Additionally, we are able to lower the area demand by omitting the Aes core. Despite introducing a new challenge, the utilization of BMAC proves effective in overcoming this issue. Moreover, BMAC offers additional opportunities for optimizing the performance on hardware platforms and conserving PRNG resources.

4.5. Gaussian Elimination

This section covers our approaches to solve the challenges associated with performing Gaussian Elimination in hardware. Although the actual process of computing the echelon form is relatively straightforward, it heavily relies on the memory format utilized for matrix $\mathbf{A}$ upon which Gaussian Elimination is applied. Consequently, most challenges are related to the computation of $\mathbf{A}$ and the necessary transformation into the appropriate memory format. First, we analyze the construction of the matrix $\mathbf{A}$ and how the involved $\mathbf{M}_i$ matrices require a modification. Second, we describe the reasoning why $\mathbf{A}$ needs to transition between different representations in memory to enable a high performance design, along with the technique to accomplish these transitions.

Challenge 1: Gaussian Elimination is performed on the matrix $\mathbf{A}$. We aim not only for a fast computation of the echelon form but also for an efficient building process of $\mathbf{A}$. Lines 23 and 25 of Algorithm 3 indicate that $\mathbf{A}$ depends only on the result of $\mathbf{E}^l\mathbf{M}_i$, as shown by Equation 4.9.

$$\begin{aligned} \mathbf{A}[:, i*o:(i+1)*o] &= \mathbf{A}[:, i*o:(i+1)*o] + \mathbf{E}^l\mathbf{M}_j \\ \mathbf{A}[:, j*o:(j+1)*o] &= \mathbf{A}[:, j*o:(j+1)*o] + \mathbf{E}^l\mathbf{M}_i \end{aligned} \tag{4.9}$$

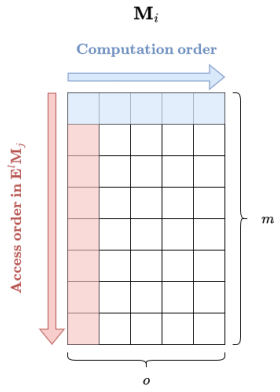

Figure 4.9.: Original computation of $\mathbf{M}_i$

Furthermore, $\mathbf{M}_i$ is computed as $\mathbf{v}_i^\top \mathbf{L}_j$ as shown by Line 18 of Algorithm 3. Thus, we have to analyze $\mathbf{E}^l\mathbf{M}_i$ first. $\mathbf{E}^l\mathbf{M}_i$ is defined as applying $\mathbf{E}^l$ to every column of $\mathbf{M}_i$ using the approach described in Section 3.4. Here we encounter the issue that $\mathbf{M}_i$ is stored in unpacked format in row-major order but its elements are accessed column-wise. Since $\mathbf{M}_i$ is of dimension $m \times o$, we already need m cycles just to load the necessary elements for the computation of $\mathbf{E}^l\mathbf{M}_i$. Additionally, we need extra logic to select the appropriate elements of $\mathbf{M}_i$.

Design Approach: Algorithm 3 considers $\mathbf{M}_i$ as unpacked matrix as illustrated by Figure 4.9. The blue arrow indicates the computation order which would occur if we follow the definition in Line 18. The matrix is filled up from top to bottom in a row-wise fashion. In general, this is not an issue since unpacked matrices are stored in row-major order and storing the rows requires only cycle. The problem arises in Lines 23 and 25 of the same algorithm, when we access $\mathbf{M}_i$ column-wise as indicated by the red arrow. Since the matrices are stored in unpacked format we have to load the whole matrix just to access a single column and, thus, we need m cycles. The biggest advantage of the unpacked format, loading and storing whole rows in one cycle, is also its biggest downside when it comes to column access. Since the computations of Lines 23 and 25 are inside the two loops of Line 19 and 20, the impact on the performance is severe. In total, we spend $mk(k-1)$ cycles just to load the elements of $\mathbf{M}_i$.

Therefore, it is very important to modify the computation of $\mathbf{M}_i$ to improve the memory access times. This can be achieved by utilizing the packed format. Instead of treating $\mathbf{M}_i$ as unpacked matrix, we perform an unpacked-packed multiplication as it is known from the $\mathbf{P}_i$ matrices. Given that $\mathbf{v}_i \in \mathbb{F}_{16}^{n-o}$ and $\mathbf{L}_j \in \mathbb{F}_{16}^{(n-o)\times o}$, we obtain a packed vector $\mathbf{m}_i = \mathbf{v}_i\{\mathbf{L}_j\}_{j\in[m]} \in \mathbb{F}_{16}^{o}$. $\mathbf{m}_i$ consists of the same elements as $\mathbf{M}_i$, the only difference is in our internal representation in memory where, $\mathbf{m}_i$ is stored in packed format and $\mathbf{M}_i$ is stored in unpacked format. Instead of considering the k $\mathbf{m}_i$ as single vector, we can combine them into one packed matrix $\mathbf{M}$. This approach is illustrated in Figure 4.10.

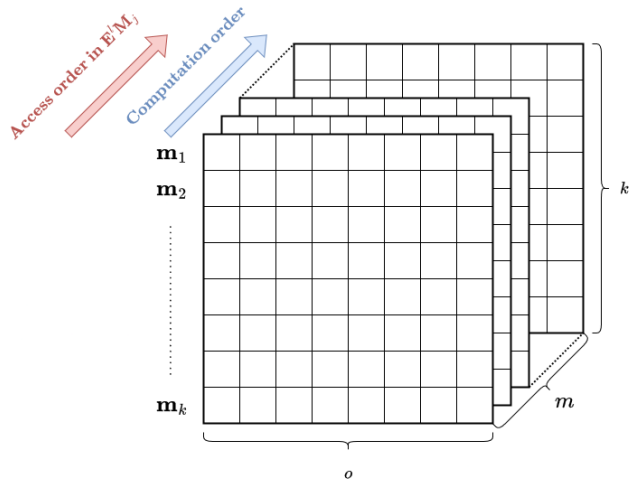

Figure 4.10.: Packed computation of $\mathbf{M}$

The packed multiplication essentially computes the columns of $\mathbf{M}_i$ before the rows. Thus, the computation and access order is now identical, highlighted again by the blue and red arrow. The different $\mathbf{M}_i$ are transformed into packed vectors $\mathbf{m}_i$ which reside in the rows of the packed matrix $\mathbf{M}$. This approach enables us to store and load the elements of $\mathbf{m}_i$ within one cycle. A very important advantage of this packed processing is that we can utilize the $\mathbb{F}_{16}$-ALU blocks as described in Section 4.3. As a result, a whole column of $\mathbf{M}_i$ is computed simultaneously. Thus, we do not only improve the access time but also decrease the computational latency. Instead of spending $mk(k-1)$ cycles to load the elements of $\mathbf{M}_i$, we reduced it to $ok(k-1)$. This corresponds to a reduction by a factor of 8.0, 9.6, and 10.6 for the respective security levels.

Challenge 2: The utilized algorithm to solve the linear system is Gaussian Elimination. It is employed to transform the augmented matrix $(\mathbf{A}||\mathbf{y}) \in \mathbb{F}_{16}^{m \times ko+1}$ into echelon form to subsequently sample a signature. As shown by Algorithm 1, Gaussian Elimination heavily relies on row operations, among them swapping rows, multiplying by a constant, or adding two rows. Thus, it is essential to load and store rows of $(\mathbf{A}||\mathbf{y})$ in BRAM as fast as possible. To achieve this, we introduced the unpacked format in Section 4.2. This format allows us to load and store rows within one cycle since one BRAM entry holds a full row of $(\mathbf{A}||\mathbf{y})$. However, the signature computation algorithm constructs $\mathbf{A}$ in a column-major order as shown by Line 23 and 25 of Algorithm 3. Building $\mathbf{A}$ in this fashion slows down the performance of the signature computation since writing a full column to memory requires at least m cycles. Additionally, this computation is nested within two loops, which further amplifies the slowdown. Consequently, the building process of $\mathbf{A}$ would result in a significant bottleneck for the signing algorithm.

Design Approach: The foundation for a fast Gaussian Elimination algorithm is the utilization of the unpacked format. However, this poses a challenge in the building process of $\mathbf{A}$, which is performed in a column-wise order. Therefore, we have to modify the computation of $\mathbf{A}$ in Lines 23 to 25 of Algorithm 3. This problem is similar to the previous one of the $\mathbf{M}_i$ matrices. Unfortunately, the approach used for $\mathbf{M}_i$ is not applicable for $\mathbf{A}$ since the computation of $\mathbf{A}$ is not related to a packed multiplication in any way. However, our solution is relatively straightforward. Instead of building $\mathbf{A}$ directly, we compute its transpose $\mathbf{A}^\top$. Therefore, the column-wise computation order of $\mathbf{A}$ is transformed into a sequence of row computations for $\mathbf{A}^\top$. Consequently, we need to alter the sign algorithm accordingly. The modified part is shown in Algorithm 8.

Algorithm 8 Adapted Signature Computation [Beu+23a]

```
26: for i from 0 to k − 1 do
27:     for j from k − 1 to i do
28:         A⊤[i ∗ o : (i + 1) ∗ o, :] ← A⊤[i ∗ o : (i + 1) ∗ o, :] + E^l M_j
29:         if i ≠ j then
30:             A⊤[j ∗ o : (j + 1) ∗ o, :] ← A⊤[j ∗ o : (j + 1) ∗ o, :] + E^l M_i
31:         l ← l + 1
32: x ← SampleSolution(A⊤, y, r)
```

The transposed version of $\mathbf{A}$ is denoted as $\mathbf{A}_\top$. Lines 28 and 30 demonstrate the row-wise access instead of the original column-wise order in Algorithm 3. Although the issue of accessing $\mathbf{A}$ in a wrong order is solved for its computation, we probably introduced new problems since $\mathbf{A}$ is passed to SampleSolution in its transposed form. We have to inspect all the operations where $\mathbf{A}$ is involved and examine the impact of using $\mathbf{A}_\top$ instead.

The first relevant computation occurs in Line 2 of Algorithm 4, where $\mathbf{y} = \mathbf{y} - \mathbf{Ar}$. $\mathbf{Ar}$ cannot be computed in this manner, since the dimensions of $\mathbf{A}_\top$ and $\mathbf{r}$ do not align. Nevertheless, this is easily resolved due to the fact that $\mathbf{Ar} = \mathbf{r}^\top \mathbf{A}_\top$. The only difference is that the result is a row vector rather than a column one. However, the dimension of a vector is not important regarding our memory design, since vectors in unpacked format are always stored in a single BRAM entry, regardless if they are in row or column form. In Line 4, $\mathbf{A}$ and $\mathbf{y}$ are concatenated to create the augmented matrix for the Gaussian elimination. This essentially adds $\mathbf{y}$ as a column to $\mathbf{A}$. Since $\mathbf{A}_\top$ is in unpacked format, we have fast column write and read. Thus, we are able to append $\mathbf{y}$ in one cycle.

The major issue arises in Line 4 when we hand over $\mathbf{A}_\top$ to GaussianElimination. The computation of the echelon form requires $\mathbf{A}$ in unpacked format to perform efficiently. Thus, there is no alternative than transposing $\mathbf{A}_\top$ before performing Gaussian Elimination. Given that $\mathbf{A}_\top$ is of dimension $ko \times m$ and, hence, rather large, it is not reasonable to retrieve the whole matrix from memory, except a developer completely disregards area requirements. Instead, we transpose the matrix in parts. We iterate over all BRAM entries of $\mathbf{A}_\top$, which store the rows of $\mathbf{A}_\top$, and select r elements. After

loading the last BRAM entry of $\mathbf{A}_\top$, we end up with r rows of $\mathbf{A}$ and store them in a new memory location. This procedure is repeated for all columns of $\mathbf{A}_\top$ until the transposition is completed. Consequently, we have to iterate over the BRAM entries of $\mathbf{A}_\top$ $\lceil m/r \rceil$ times. Now we are able to pass $\mathbf{A}$ in unpacked format to GaussianElimination and compute the echelon form as described in Algorithm 1.

After GaussianElimination is finished, we obtain $\mathbf{A}$ in echelon form. Hence, we can proceed with sampling the solution. Line 11 of Algorithm 4 performs a search within a row until the first non-zero element is found. This aligns with the unpacked format of $\mathbf{A}$ since we are able to load a whole row within a cycle and carry out the search. However, the computation in Line 14 accesses a column of $\mathbf{A}$. Thus, we would need m cycles to load all column elements. Therefore, we perform the same transposition as presented in the previous paragraph. As a result, we obtained the echelon form of the matrices $\mathbf{A}$ and $\mathbf{A}_\top$ in unpacked format. Consequently, we have fast row and column access for $\mathbf{A}$ to perform the computations of Lines 11 and 14 efficiently. Algorithm 9 incorporates all the presented modifications for SampleSignature.

Algorithm 9 Adapted Sampling Solutions [Beu+23a]

SampleSolution$(\mathbf{A}_\top, \mathbf{y}, \mathbf{r})$:

1: $\mathbf{x} \leftarrow \mathbf{r}$
2: $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{r}^\top \mathbf{A}_\top$
3:
4: $\mathbf{B}_\top \leftarrow \begin{pmatrix} \mathbf{A}_\top \\ == \\ \mathbf{y} \end{pmatrix}$ ▷ Append $\mathbf{y}$ to $\mathbf{A}_\top$ as row
5:
6: $\mathbf{B} \leftarrow (\mathbf{B}_\top)^\top$ ▷ Transpose augmented matrix $\mathbf{B}_\top$ to enable fast computation of GaussianElimination
7:
8: $(\mathbf{A}||\mathbf{y}) \leftarrow \text{GaussianElimination}(\mathbf{B})$
9:
10: **if** $\mathbf{A}[m-1,:] = \mathbf{0}$ **then**
11: **return** $\perp$
12:
13: $\mathbf{A}_\top \leftarrow \mathbf{A}^\top$ ▷ Transpose echelon form of $\mathbf{A}$ to enable fast column access
14:
15: **for** r from $m-1$ to 0 **do**
16: $c \leftarrow 0$
17: **while** $\mathbf{A}[r,c] = 0$ **do**
18: $c \leftarrow c + 1$
19: $\mathbf{x}_c \leftarrow \mathbf{x}_c + \mathbf{y}_r$
20: $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{y}_r \mathbf{A}_\top[c,:]$
21: **return** $\mathbf{x}$

Line 2 shows the modified computation of $\mathbf{y}$. Appending $\mathbf{y}$ to $\mathbf{A}_\top$ and transposing the augmented matrix occurs in Line 4 and 6. The second transposition is shown in Line 13 to enable fast columns access of $\mathbf{A}$ in the computation of Line 20. Using this modifications allows us to fully utilize the unpacked format for computing the echelon form and sampling the solution. We reduced the latency for accessing rows and columns of $\mathbf{A}$ to a minimum. In exchange we introduced additional overhead by adding the two transpositions. The total transposing latency is

$$Transpose\ latency = \underbrace{(ko+1) \times \lceil m/r \rceil}_{\text{Transposing } \mathbf{B}_\top} + \underbrace{m \times \lceil ko/r \rceil}_{\text{Transposing } \mathbf{A}}. \tag{4.10}$$

Given $\mathbf{B}_\top$ is in $\mathbb{F}_q^{(ko+1)\times m}$, we need to iterate over all $(ko+1)$ BRAM entries $\lceil m/r \rceil$ times. Likewise, we have to access all m entries of $\mathbf{A}$ $\lceil ko/r \rceil$ times. By transposing $\mathbf{A}$ we achieve the maximum performance in regard to memory access for the operations involved in Algorithms 3, 4, and 1. Additionally, the latency shown in Equation 4.10 is lower compared to the latency which would occur without transposing since the wrong access order in the described loops reduces the performance significantly.

To conclude, we modified the computation of $\mathbf{M}_i$ by utilizing the packed memory format and the available $\mathbb{F}_{16}$-ALU blocks leading to a significant improvement in both, computation and memory access time. Additionally, we changed the signature computation algorithm to build $\mathbf{A}_\top$ instead of $\mathbf{A}$ to benefit from the resulting row-order access. To provide Gaussian Elimination with $\mathbf{A}$ in its required format, we introduced a flexible transposing technique. The same procedure is then used to transform the obtained echelon form into its transposed version, enabling the sample solution algorithm to efficiently access rows and columns in memory.

Chapter 5.

Software Modifications

In this chapter, we examine the consequences of substituting AES with SHAKE as PRNG in software. The design strategy outlined in Section 4.4 leads to an incompatibility between a hardware implementation and the reference software implementation. First, we provide a brief overview of PRNG usage in the reference version of MAYO as submitted to NIST, along with the setup involved in utilizing AES and SHAKE for generating pseudo-random data. Subsequently, we adapt the software implementation to exclusively employ SHAKE as the PRNG in accordance with our hardware design approach. We then describe the various software configurations within MAYO and explain their relevance for a comparison with our modified software. Finally, we propose an approach to enhance the performance of our SHAKE adaption to demonstrate its practicality on software platforms

Challenge 1: The specification of MAYO utilizes AES-128-CTR as a PRNG to generate the elements of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. In Section 4.4 of our hardware design approaches, we replaced AES-128-CTR with SHAKE128 to enhance the performance and reduce the area requirements. However, this modification renders a hardware implementation incompatible with the original specifications. Therefore, it is of interest to apply the same adaptation for the software implementation to assess its performance and determine if employing SHAKE is also reasonable for software platforms.

Design Approach: The C-implementation of MAYO is publicly available on Github [Beu+23b]. We modify this code and adapt the PRNG call to use SHAKE128 instead of AES-128-CTR. The detailed explanation of AES and its corresponding counter mode is provided in Section 2.8. The required parameters are the seed seed_{pk} upon which expansion is performed and the desired output length in bytes, hence $P1\_bytes + P2\_bytes$. The PRNG output has to be deterministic to generate the same keys in every run if the identical seed is used. Thus, the initialization vector (IV), also referred to as nonce, is set to 0. seed_{pk} is employed as key and the counter is repeatedly encrypted by AES until the specified output length is obtained. Fixing the IV to zero essentially transforms AES from a symmetric cipher to a hash function with variable digest length. This configuration is illustrated in Figure 5.1. Due to the 128-bit block length of AES, the input and output maintain the same length. Given that the IV is set to zero, the counter value serves as the only input to the cipher. For each execution of AES, seed_{pk} serves as the

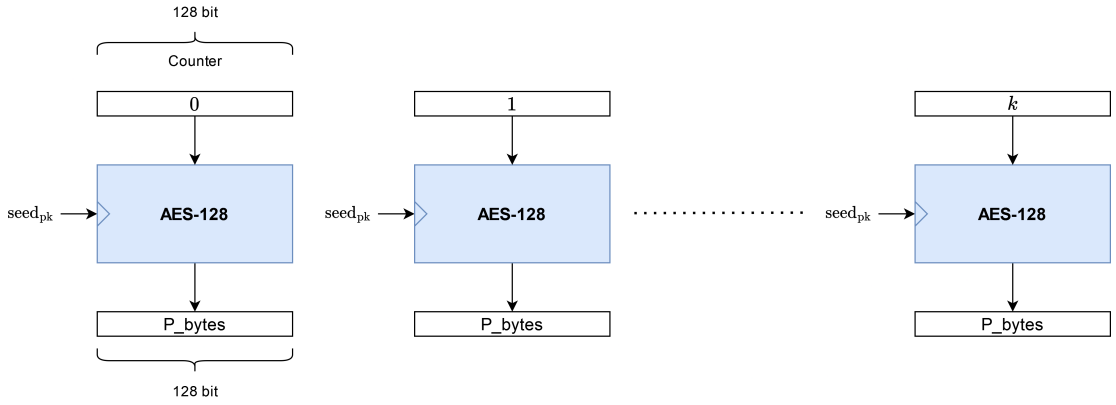

Figure 5.1.: PRNG setup of AES128

key and every run produces 16 bytes of data for $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. Encryption is repeated k times until sufficient data is generated, thus k equals $\lceil (P1\_bytes + P2\_bytes)/16 \rceil$.

SHAKE on the other hand is already an extendable output hash function, thus, it allows arbitrarily long output lengths. It only requires the input and the desired output length as parameters. Consequently, both primitives fulfill the requirements for being used as a PRNG in MAYO. Therefore, we only have to replace the AES-128-CTR call in Line 7 of Algorithm 2, Line 5 of Algorithm 3, and Line 3 of Algorithm 5 with SHAKE128. As a result, we obtain a software implementation of MAYO which is compatible to our proposed hardware approach of Section 4.4.

The C-implementation of MAYO is available in different configurations as outlined in Section 3.7. Specifically, there are 4 configurations:

1. **Reference mode**: The reference implementation comes with the lowest degree of optimization. It does not rely on any instruction set support and is essentially a one-to-one translation of the algorithms described in Section 3.3.
2. **Optimized mode**: The optimized implementation is based on the reference mode but with loop unrolling and improved bitslice arithmetic. Same as before, it does not require any instruction set support.
3. **Optimized mode with AES-NI**: The optimized implementation is extended by the utilization of AES-NI. While the performance is increased significantly, this mode is only available on CPU platforms with AES-NI instruction set support.
4. **AVX2 mode**: In addition to AES-NI, AVX2 is employed to parallelize matrix computations. The AVX2 mode comes with the highest degree of optimization and, thus, offers the best performance. However, AES-NI and AVX2 support is required for this configuration.

We compare the performance of these 4 configurations with the SHAKE version of MAYO to test the capability of a pure SHAKE adaption. The results are listed in Table 6.4 of Chapter 6. Summarized, the SHAKE version outperforms configurations (1) and (2), but fails to keep up with (3) and (4).

Challenge 2: The SHAKE version of MAYO shows a performance decrease in configurations (3) and (4). Although, there is an increase in the first two modes, the decrease in the last two is more significant, because in general, modern CPUs come with support for AES-NI and AVX2 as described in Section 2.8 and 2.10. Thus, we have to enhance our SHAKE adaption to proof its practicability on software platforms.

Design Approach: In search of opportunities to improve performance, we explored analogous schemes that also utilize SHAKE to generate matrix elements. Two candidates that follow a similar approach are Kyber [Bos+18a] and Saber [D'A+18]. Both schemes utilize SHAKE128 for generating a matrix. They employ multiple independent SHAKE instances to generate the elements in parallel using AVX. Consequently, multiple seeds are required for the different SHAKE calls to generate distinct output streams. SHAKE employs 64-bit state variables as described in Section 2.9. Thus, four state variables can be packed into one AVX-256 vector. As a result, it is feasible to execute four parallel instances of SHAKE128. We modify the C-implementation of MAYO to adopt a multi-seed approach, enabling the generation of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ through four parallel SHAKE128 calls. We denote this variant as SHAKE128X4

First, we have to modify the MAYO scheme to use four different seeds for generating $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. To maintain the security margin, we use the same length for the individual seeds as in the original specifications, specifically 4×16 bytes. Thus, the public key seed length is increased to 64, as shown in Table 5.1. As a result, the public key size rises from 1168, 2656, and 5008 B to 1216, 2704, and 5056 B for the respective security levels. This increase is neglectable.

Table 5.1.: MAYO public key sizes for single and multi-seed implementation

Parameter Set	**MAYO$_1$**	**MAYO$_3$**	**MAYO$_5$**
Reference Implementation			
PK Seed Bytes	16	16	16
Public Key Size	1168 B	2656 B	5008 B
Multi-seed Implementation			
PK Seed Bytes	64	64	64
Public Key Size	1216 B	2704 B	5056 B

In the C-implementation the PRNG generates 69600 B for MAYO$_1$, 93120 B for MAYO$_3$, and 565312 B for MAYO$_5$ for all $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices. Given that all three numbers share 4 as a common factor, we can evenly distribute the generation among 4 SHAKE128 instances. This approach is illustrated in Figure 5.2.

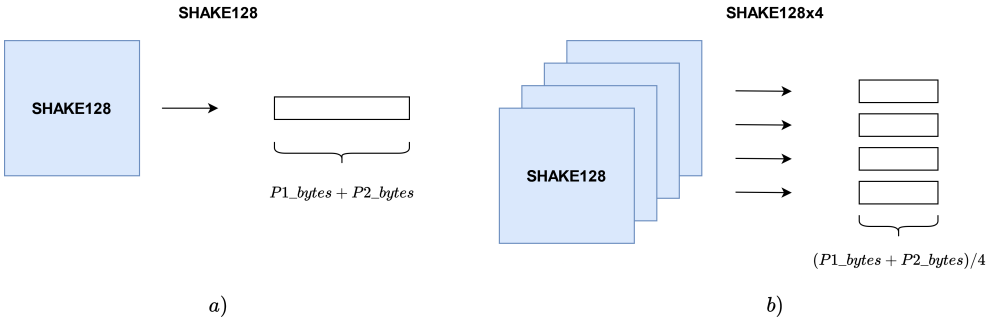

Figure 5.2.: Comparison of single and multi-seed approach

In the original variant, a single SHAKE128 call is used to generate the data for $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ as depicted by a). In the SHAKE128x4 modification, 4 individual SHAKE128 instances generate one fourth of the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ elements in parallel, as shown in b). Due to the determinism of SHAKE128, the 4 instances use 4 different seeds as input to generate distinct elements. Once the generation is completed, the 4 output streams are concatenated and the scheme can proceed as usual. The obtained performance increase is listed in Table 6.4 and will be further discussed in Section 6.2.

The C-implementation of Kyber is available at [Bos+18b]. We adapted the code responsible for the parallel execution of SHAKE to use it for our SHAKE128x4 configuration. Integrating SHAKE128x4 into MAYO only requires the modification of the PRNG calls in Line 7 of Algorithm 2, Line 5 of Algorithm 3, and Line 3 of Algorithm 5. The adapted version of key generation is shown in Algorithm 10.

Algorithm 10 ADAPTED KEY GENERATION [Beu+23a]

CompactKeyGen():
Output: Compact public key cpk $\in \mathcal{B}^{\text{cpk\_bytes}}$ and secret key csk $\in \mathcal{B}^{\text{csk\_bytes}}$

1: $\text{seed}_{\text{sk}} \leftarrow \text{RANDOM}(\text{sk\_seed\_bytes})$
2:
3: //Expand seed_{sk} to get 4 public seeds and $\mathbf{O}$
4: $\text{seed}_{\text{pk}}^1, \text{seed}_{\text{pk}}^2, \text{seed}_{\text{pk}}^3, \text{seed}_{\text{pk}}^4, \mathbf{O} \leftarrow \text{SHAKE256}(\text{seed}_{\text{sk}})$
5:
6: //Expand $\text{seed}_{\text{pk}}^k$ to get $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$
7: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \text{SHAKE128x4}(\text{seed}_{\text{pk}}^1, \text{seed}_{\text{pk}}^2, \text{seed}_{\text{pk}}^3, \text{seed}_{\text{pk}}^4)$
8: $\{\mathbf{P}_i^{(3)}\}_{i\in[m]} \leftarrow \text{Upper}(-\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^\top \mathbf{P}_i^{(2)})_{i\in[m]}$
9:
10: $\text{cpk} \leftarrow \text{seed}_{\text{pk}}^1 \,||\, \text{seed}_{\text{pk}}^2 \,||\, \text{seed}_{\text{pk}}^3 \,||\, \text{seed}_{\text{pk}}^4 \,||\, \{\mathbf{P}_i^{(3)}\}_{i\in[m]}$
11: $\text{csk} \leftarrow \text{seed}_{\text{sk}}$
12: **return** (cpk, csk)

In Line 4, Shake256 is utilized to generate 4 public seeds instead of one. Subsequently, these 4 seeds are passed to Shake128x4 to parallelize the generation of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. To reconstruct the $\mathbf{P}_i$ matrices in the other algorithms, the public key now needs to comprise the 4 seeds, as shown in Line 10.

To ensure compatibility with the non-AVX2 configurations, specifically (1), (2), and (3), Shake128x4 has to be replaced by four subsequent Shake128 calls using the four different seeds and a smaller output length. Thus, it is relatively straightforward to transition the MAYO specifications from a single to a multi-seed scheme. This adaption is independent of Shake being employed as PRNG. In the case of Aes, the scheme is modified such that four different keys are used instead of one. Consequently, the value of the counter has to be reset upon key switching.

In conclusion, we adapted the software implementation of MAYO to accommodate the use of Shake as the only PRNG. We analyzed the impact on the scheme's performance and presented an approach to enhance its efficiency, demonstrating that our modification is applicable to software platforms as well.

Chapter 6.

Results

In this chapter, we present our implementation results and compare them to related work. The chapter comprises two sections. First, we outline the results of our hardware design approaches. Second, we compare our modified software variant with the original implementation.

6.1. Hardware Results

The objective of this thesis is to propose several design approaches which render a hardware implementation of the complete MAYO scheme possible. As stated in the introduction of Chapter 4, giving a full hardware design is beyond the scope of this thesis. However, we have developed a hardware implementation of MAYO that follows the design approaches of Chapter 4 as part of our paper [Hir+23] to prove the effectiveness of the presented hardware design techniques. The tables in this section originate from this work and include only the parts required to demonstrate the capability of our design approaches. For further details regarding the full hardware design, we refer readers to [Hir+23].

The area and performance results are given for two different variants. One variant utilizes AES-128-CTR as PRNG, while the other variant employs SHAKE128. Therefore, we provide results for a version that is compliant with the official MAYO specifications and another to demonstrate the improvements achievable by using SHAKE128. The area and performance metrics are acquired using Xilinx Vivado 2022.2 for the Alveo U280 (XCU280). This board is chosen to provide a consistent comparison across all security levels. Although it is feasible to synthesize MAYO_1 on a low-end Xilinx Artix-7 FPGA, this does not extend to MAYO_3 and MAYO_5. Area and performance results of the hardware implementation are shown in Table 6.1. All computations are performed exclusively on hardware without the need for any communication with software. Initially we discuss the area results followed by the performance results.

Area results: Our FPGA hardware implementations exhibit relatively high look-up table (LUT) utilization, primarily due to the handling of data from multiple memory banks as described in Section 4.2 and the utilization of multiple $\mathbb{F}_{16}$-ALU blocks as outlined in Section 4.3. While this approach allows our design to perform the majority of

Table 6.1.: Area and performance results

Design	Platform	Latency (in cc/ms)			Area
		KeyGen	Sign	Verify	LUT/FF/DSP/BR
MAYO with AES-128-CTR based seed expansion					
$MAYO_1$	AU280 @ 200MHz	29,287/0.15	94,686/0.47	30,150/0.15	93,841/32,890/2/45.5
$MAYO_3$	AU280 @ 200MHz	65,551/0.32	213,845/1.06	66,939/0.33	157,125/52,975/2/96
$MAYO_5$	AU280 @ 200MHz	118,027/0.59	384,488/1.92	19,603/0.98	219,363/73,900/2/194.5
MAYO with SHAKE128 based seed expansion					
$MAYO_1$	AU280 @ 200MHz	12,182/0.06	49,926/0.25	12,722/0.05	91,480/32,007/2/45.5
$MAYO_3$	AU280 @ 200MHz	38,325/0.19	137,358/0.69	39,740/0.20	153,195/51,650/2/96
$MAYO_5$	AU280 @ 200MHz	90,743/0.45	241,310/1.21	92,339/0.46	222,822/71,913/2/194.5

AU280: Xilinx Alveo U280.

computations in parallel and avoids redundant data loading from memory, it does result in increased implementation complexity. The ALU blocks also contribute to the flip-flop (FF) utilization due to their internal register. Another factor of the FF utilization are the employed data caches for $\mathbf{O}$, $\mathbf{v}_i$, and $\mathbf{s}_i$ and the PRNG buffer responsible for storing the generated elements. Noteworthy is the low Digital Signal Processing (DSP) block utilization in all hardware implementations. Only two DSPs are required because the ALU design exclusively utilizes bitsliced AND and XOR operations. The BRAM utilization is relatively low due to our on-the-fly generation technique, as described in Section 4.1, and the optimized memory design. The combination of these approaches enables us to minimize the number of required BRAMs compared to previous works in literature, as we will demonstrate later in this section. This is primarily because our implementation only requires the storage of intermediate results rather than the large $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices, which can occupy up to several thousand kilobytes of memory.

A comparison between the AES-128 and SHAKE128 variant reveals a slight advantage for the latter one regarding area demands. In the case of $MAYO_1$ and $MAYO_3$, the implementations with SHAKE128 as only PRNG exhibit a slightly decreased LUT utilization compared to the implementations that additionally employ AES-128. The detailed resource utilization of the two PRNGs is given in Table 6.2. We employ multiple AES cores to match the faster performance of SHAKE128 as described in Section 4.4. It is important to note that the AES variants also need to incorporate a KECCAK core, since seed_{pk} and the elements of matrix $\mathbf{O}$ are generated using SHAKE256 as shown in Line 4 of Algorithm 2. As demonstrated by the table, omitting AES and using SHAKE as sole PRNG results in a reduction of the area requirements. Specifically, the LUT utilization is reduced by 22, 24, and 27 % across all security levels. Similarly, the FF utilization is reduced by 23, 30, and 36 %, respectively.

Performance results: The performance results in terms of clock cycles and milliseconds are presented in Table 6.1. The AES-128 variant completes key generation, signature computation, and verification in 0.15/0.32/0.59 ms, 0.47/1.06/1.92 ms, and

Table 6.2.: Resource utilization of AES-128 and KECCAK

Design	Resources	AES128	Keccak	Total
$MAYO_1$	LUTs	2,981	11,025	14,006
(2×AES)	FFs	916	3,200	4,116
$MAYO_1$	LUTs	-	10,895	10,895
(SHAKE)	FFs	-	3,186	3,186
$MAYO_3$	LUTs	4,078	10,077	14,155
(3×AES)	FFs	1,368	3,179	4,547
$MAYO_3$	LUTs	-	10,716	10,716
(SHAKE)	FFs	-	3,183	3,183
$MAYO_5$	LUTs	9,895	5,418	15,313
(4×AES)	FFs	3,214	1,808	5,022
$MAYO_5$	LUTs	-	11,179	11,179
(SHAKE)	FFs	-	3,208	3,208

0.15/0.33/0.98 *ms* for $MAYO_1$/$MAYO_3$/$MAYO_5$, respectively. For the SHAKE128 variant, the required time for $MAYO_1$/$MAYO_3$/$MAYO_5$ is 0.06/0.19/0.46 *ms* for key generation, 0.25/0.69/1.20 *ms* for signature computation, and 0.06/0.15/0.46 *ms* for verification.

Our SHAKE-based hardware implementation requires a total of 12128, and 49926, and 12722 cycles for keygen, sign, and verify in the first security level. For the third security level, the latency increases to 38325, 137358, and 39740 cycles. This represents an increase of 3.15×, 2.75×, and 3.12×, respectively. This increase in latency is primarily attributed to the larger amount of pseudo-random data that has to be generated. The total size of $\mathbf{P}_i^{(1)}$ combined with $\mathbf{P}_i^{(2)}$ has grown from 2175 to 4895 elements from $MAYO_1$ to $MAYO_3$. Thus, transitioning from security level 1 to 3 results in a 2.25× increase in pseudo-random data generation. In case of security level 5, the latency of each operation increases by a factor of 7.45×, 4.83×, and 7.26× compared to security level 1. The total size of the two $\mathbf{P}_i$ matrices changes from 2175 to 8833 elements, resulting in an 4.06× increase in pseudo-random data generation. This clearly indicates that the most significant factor influencing the latency across the security levels is the amount of pseudo-random generated data for $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. A similar result can be observed for the AES-based variant.

A comparison between the AES-128 and SHAKE128 variant reveals a significant advantage for the latter one in terms of performance. This is attributed to the faster generation rate of SHAKE, which enhances the pseudo-random data generation by a factor of 4.84, as outlined in Section 4.4. The latency values in Table 6.1 indicate a performance increase of key generation, signature computation, and verification up to 2.5×, 1.88×, and 3×, respectively. Thus, the SHAKE variant clearly outperforms the AES version on hardware, despite instantiating multiple AES cores, as shown in Table 6.2.

Table 6.3.: Comparison with related works

	Works	Platform	Latency (cc/ms)			Area
			KeyGen	Sign	Verify	LUT/FF/BR
UOV_1	[Beu+23e][a]	Artix-7 @ 90.3MHz[c]	11.0M/121.91	779K/8.63	115K/1.27	34,208/26,974/66
$MAYO_1$	[Beu+23a]	AC-M4 @ 1GHz	5.24M/5.24	9.18M/9.18	4.88M/4.88	-
		IXG 6338 @ 2GHz[b]	110K/0.05	460K/0.23	175K/0.08	-
	[Say+23]	Z-7020 @ 100MHz	996K/9.96	3.49M/34.92	-	23,356/24,645/136
	Our (2×AES)	AU280 @ 200MHz	29,287/0.15	94,686/0.47	30,150/0.15	93,841/32,890/45.5
	Our (SHAKE)	AU280 @ 200MHz	15,291/0.08	58,305/0.29	16,013/0.07	91,480/32,007/45.5
UOV_3	[Beu+23e][a]	Artix-7 @ 94.1MHz[c]	16.4M/174.94	1.19M/12.75	195K/2.07	43,166/31,928/184.5
$MAYO_3$	[Beu+23a]	IXG 6338 @ 2GHz[b]	508K/0.25	1.66M/0.83	610K/0.30	-
	Our (3×AES)	AU280 @ 200MHz	65,551/0.32	213,845/1.06	66,939/0.33	157,125/52,975/96
	Our (SHAKE)	AU280 @ 200MHz	38,325/0.19	137,358/0.69	39,740/0.20	153,195/51,650/96
UOV_5	[Beu+23e][a]	Artix-7 @ 92.6MHz[c]	38.4M/414.73	2.64M/28.56	364K/3.93	83,444/40,597/359
$MAYO_5$	[Beu+23a]	IXG 6338 @ 2GHz[b]	1.21M/0.60	4.14M/2.07	1.18M/0.59	-
	Our (4×AES)	AU280 @ 200MHz	118,027/0.59	384,488/1.92	119,603/0.60	219,363/73,900/194,5
	Our (SHAKE)	AU280 @ 200MHz	90,743/0.45	241,310/1.20	92,339/0.16	222,822/71,913/194.5

Z-7020: Xilinx Zynq-7020. IXG 6338: Intel Xeon Gold 6338. AC-M4: ARM Cortex-M4.

[a]: Targets UOV scheme. [b]: Uses AVX2 with AES-NI. [c]: Uses pipelined AES128. K and M are used as an abbreviation for $\times 10^3$ and $\times 10^6$, respectively.

Comparison with related works: Given the novelty of MAYO, available implementations are rare. We reference two works that address implementation aspects for either CPU [Beu+23a], ARM [Beu+23a], or FPGA [Say+23]. However, the FPGA publication only covers key generation and signature computation exclusively for $MAYO_1$. Therefore, we also include a work that describes an FPGA design for UOV [Beu+23e] to provide a comprehensive comparison for every algorithm across all security levels. Since UOV closely resembles MAYO in its construction and computations, a comparison is possible. Table 6.3 provides the area and performance results of the aforementioned publications and our work. We use the latency of our AES variant for a fair comparison.

First, we compare our work with existing FPGA works, as this represents our primary focus. As far as we are aware, only one FPGA implementation of MAYO exists in the literature, namely, HaMAYO [Say+23]. However, HaMAYO is limited to performing key generation and signature computation operations for security level 1 and uses the outdated parameter set of the MAYO scheme. In contrast to HaMAYO, we support all operations and security levels 1, 3, and 5 of the MAYO scheme. Our hardware design outperforms HaMAYO by 66× and 74× for key generation and signature computation, respectively. Our enhanced performance comes with the trade-off of 4× higher LUT

and 1.3× higher FF utilization, although we require 2.99× fewer BRAMs compared to HaMAYO.

To present a comparison for security level 3 and 5, we included [Beu+23e]. This work presents an FPGA design of the UOV scheme and provides area and performance results for all operations and security levels. We use their high-performance implementation with pipelined AES and $\mathbb{F}_{16}$ arithmetic for the comparison, as it closely resembles the MAYO specifications. Our design surpasses key generation/signature computation/verification by up to 813×/18×/8×, 547×/12×/6×, and 703×/15×/7× for security level 1, 3, and 5, respectively. This is attributed to our extensive parallelization of the matrix-matrix multiplications. However, the high degree of parallelism comes at the cost of increased resource utilization, resulting in 2.7×/3.6×/2.6× higher LUT and 1.2×/1.7×/1.8× higher FF usage for security levels 1/3/5. Nevertheless, their BRAM utilization is increased by 1.5×/1.9×/1.8× for security levels 1/3/5.

These results clearly demonstrate that our design outperforms HaMAYO [Say+23] and [Beu+23e] by one to three orders of magnitude. Moreover, besides performance enhancements, our on-the-fly generation technique combined with the memory design illustrates how to reduce the consumption of BRAMs by half for UOV and MAYO schemes on hardware.

Next, we compare our work with the ARM design presented in [Beu+23a]. Similar to HaMAYO, results are provided only for MAYO_1, but in this case, for all algorithms. Our implementation outperforms key generation, signature computation, and verification by 35×, 20×, and 33×, respectively.

Finally, we conduct a comparison with the high-end CPU implementation of [Beu+23a]. We opted for their most performant AVX2 implementation utilizing AES-NI. The results are obtained on an Intel Xeon Gold 6338 CPU (Ice Lake). Our hardware design performs similarly or worse depending on the security level. The software implementation outperforms our work in key generation/signature computation/verification by 3×/2×/1.9× and 1.3×/1.3×/1.1× for security level 1 and 3, respectively. The latency of our work and the software implementation is at a similar level for MAYO_5. However, as shown by Table 6.3, the superior performance of our SHAKE variant is capable of narrowing this gap.

In conclusion, our hardware design outperforms most of the related implementations while utilizing more LUTs and FFs. MAYO_1 and MAYO_3 on a high-end CPU platform is the sole exception in terms of performance. This clearly demonstrates the capability of our presented hardware design approaches.

6.2. Software Results

The specification of MAYO utilizes AES-128-CTR as PRNG to generate the elements of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. The C-implementation of MAYO is available in various configura-

tions. Specifically, there are four configurations: Reference mode (1), Optimized mode (2), Optimized mode with AES-NI (3), and AVX2 mode (4). No instruction set support is necessary for the first two configurations. However, AES-NI support is required for the third configuration and the fourth configuration requires AVX2 support in addition to AES-NI. If both AES-NI and AVX2 instruction sets are supported, AES-NI accelerates the generation of pseudo-random data and AVX2 reduces the computational latency.

Table 6.4.: MAYO performance in CPU cycles on an Intel i5-7300U at 2.60GHz. The library was compiled on Ubuntu with gcc version 11.4.0-1ubuntu1 22.04. Results are the median of 1000 benchmark runs

Scheme	KeyGen	ExpandSK	ExpandPK	ExpandSK + Sign	ExpandPK + Verify
Configuration 1: Reference Implementation:					
$MAYO_1$ - AES-128	1,662,549	1,752,476	1,351,339	2,439,583	1,556,351
$MAYO_1$ - SHAKE128	713,854	801,117	403,174	1,486,906	605,872
$MAYO_3$ - AES-128	7,639,649	6,277,750	4,605,025	10,980,786	6,084,326
$MAYO_3$ - SHAKE128	4,441,638	3,021,492	1,357,584	7,734,692	2,747,314
$MAYO_5$ - AES-128	15,447,969	16,174,596	11,126,708	23,099,671	12,798,116
$MAYO_5$ - SHAKE128	7,482,941	8,160,095	3,293,238	15,247,473	4,836,765
Configuration 2: Optimized Implementation without AES-NI:					
$MAYO_1$ - AES-128	1,663,461	1,753,371	1,351,716	2,464,112	1,559,883
$MAYO_1$ - SHAKE128	713,982	802,221	403,182	1,486,190	607,556
$MAYO_3$ - AES-128	7,659,665	6,292,588	4,615,400	11,010,849	6,007,949
$MAYO_3$ - SHAKE128	4,389,432	3,030,337	1,358,121	7,751,360	2,758,221
$MAYO_5$ - AES-128	15,457,654	16,340,242	11,141,203	23,517,764	12,803,594
$MAYO_5$ - SHAKE128	7,504,031	8,218,936	3,295,467	15,362,409	4,847,094
Configuration 3: Optimized Implementation with AES-NI:					
$MAYO_1$ - AES-128	344,252	431,652	34,152	1,117,312	237,247
$MAYO_1$ - SHAKE128	713,842	802,209	403,518	1,487,726	603,638
$MAYO_3$ - AES-128	3,123,392	1,759,719	114,751	6,464,676	1,474,894
$MAYO_3$ - SHAKE128	4,400,519	3,069,822	1,357,976	7,744,110	2,754,690
$MAYO_5$ - AES-128	4,466,331	5,092,942	275,632	12,231,455	1,721,823
$MAYO_5$ - SHAKE128	7,499,027	8,204,599	3,302,239	15,313,877	4,898,631
Configuration 4: AVX2 Optimized Implementation with AES-NI:					
$MAYO_1$ - AES-128	129,760	134,258	34,160	395,064	163,274
$MAYO_1$ - SHAKE128	499,020	501,111	403,239	763,103	533,731
$MAYO_1$ - SHAKE128X4	255,462	254,256	158,642	520,868	288,566
$MAYO_3$ - AES-128	629,795	846,652	114,665	1,979,929	707,485
$MAYO_3$ - SHAKE128	1,873,847	2,095,504	1,357,915	3,257,199	1,957,071
$MAYO_3$ - SHAKE128X4	1,047,088	1,267,037	532,676	2,432,315	1,133,366
$MAYO_5$ - AES-128	1,432,013	1,954,131	275,627	4,252,660	1,203,565
$MAYO_5$ - SHAKE128	4,561,166	5,074,846	3,311,075	7,696,547	4,394,867
$MAYO_5$ - SHAKE128X4	2,498,755	3,009,640	1,281,901	5,352,631	2,238,094

We modified the software implementation of MAYO [Beu+23b] to include Shake128 alongside Aes-128 as PRNG for generating the pseudo-random data of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. This modification enables the software implementation to reflect the hardware design approach of Chapter 4.4. Table 6.4 presents the median latency in software for all security levels of MAYO using either Aes or Shake as PRNG.

In configuration (1) without Aes-Ni and AVX2 support, the Shake128 variant outperforms the Aes-128 version in key generation by 2.33×, 1.72×, and 2.06× for every security level, respectively. Regarding signature computation and verification, the selection of Shake128 as the PRNG results in a performance increase ranging from 1.41× to 1.64× and from 2.21× to 2.64×, respectively. Configuration (2) achieves a performance boost compared to the Aes-128 PRNG from 1.75× to 2.33× for key generation, from 1.42× to 1.66× for signature computation, and from 2.18× to 2.64× for verification, depending on the security level. In configuration (3), Aes-Ni is utilized for the first time. As a result, the Aes variant outperforms the Shake version in key generation, signature computation, and verification by up to 2.07×, 1.33×, and 2.85× across all security levels. Configuration (4) employs AVX2 in addition to Aes-Ni. In this case, the performance of the Aes implementation surpasses the Shake variant by a factor of up to 3.85×, 1.93×, and 3.65× for key generation, signature computation and verification, respectively.

The configuration (4) part in Table 6.4 includes additional values for our multi-seed Shake128x4 modification, as described in Chapter 5. This modification enables us to narrow the gap compared to the fully optimized Aes PRNG. The performance improvement of Shake128x4 over Shake128 is by a factor of up to 1.95×, 1.47×, and 1.96× for key generation, signature computation and verification, respectively. Therefore, the performance advantage of Aes in key generation, signature computation, and verification is reduced to approximately 1.66×, 1.23×, and 1.6×, respectively.

To conclude, our modified software implementation design outperforms the original version in configuration (1) and (2). However, the utilization of Aes-Ni leads to a significant performance increase in Configuration (3) and (4). While Shake128x4 enhances the performance compared to normal Shake128, it falls slightly short of the performance achieved with Aes.

Chapter 7.

Conclusion

The objective of thesis was to present several hardware design approaches for the post-quantum signature scheme MAYO. The presented techniques optimize the resource utilization and performance of MAYO on constraint platforms. Most notably is the on-the-fly coefficient generation that addresses the large memory demand of MAYO and the combination of the memory design with the arithmetical units which enable extensive parallelization to improve the performance. Additionally, the presented techniques provide the flexibility to target either low-area or high-performance implementations. Unlike previous works, these design approaches led to the first FPGA implementation that supports the complete scheme including all security levels. In terms of performance, an improvement of one to three orders of magnitude compared to related works was achieved.

In addition to this, we analyzed the impact of utilizing Shake128 to generate pseudo-random data on software platforms. While the performance is superior compared to Aes without intruction set support, it is not able to match the performance of Aes enhanced by Aes-Ni.

The presented results highlight the effectiveness of our design approaches. Furthermore, the majority of the presented techniques can be applied to UOV schemes in general, which serve as the foundation for most post-quantum schemes based on multivariate quadratic systems. This flexibility leads to a wide range of possible applications for future multivariate schemes.

Further Research

An interesting area for further research is the exploration of the multi-seed approach to utilize AVX2 for a parallel Shake128 computation. The potential introduction of Sha-3 intrinsics may finally shift the PRNG choice in favor of Shake on software platforms. Additionally, the multi-seed approach can be adopted to further improve the performance on hardware platforms by parallelizing the pseudo-random data generation. In terms of security, more research has to be invested into the whipping technique of MAYO. While UOV serves as a robust foundation for the scheme, the whipping technique may be susceptible to attacks and requires further research to ensure its resilience.

Bibliography

[Bai+22] Shi Bai et al. *CRYSTALS-Dilithium*. Selected Algorithms 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023. 2022.

[Ber+11] Guido Bertoni et al. *The KECCAK reference*. https://keccak.team/files/Keccak-reference-3.0.pdf. Online; accessed January 17, 2024. 2011.

[Beu21] Ward Beullens. "Improved Cryptanalysis of UOV and Rainbow". In: *Advances in Cryptology – EUROCRYPT 2021*. Springer. 2021, pp. 348–373.

[Beu22a] Ward Beullens. "Breaking Rainbow Takes a Weekend on a Laptop". In: *Advances in Cryptology – CRYPTO 2022*. Springer. 2022, pp. 464–479.

[Beu22b] Ward Beullens. "MAYO: Practical Post-quantum Signatures from Oil-and-Vinegar Maps". In: *Selected Areas in Cryptography*. Springer. 2022, pp. 355–376.

[Beu+23a] Ward Beullens et al. *MAYO*. MAYO Website. https://pqmayo.org/assets/specs/mayo.pdf. 2023. URL: https://pqmayo.org/assets/specs/mayo.pdf.

[Beu+23b] Ward Beullens et al. *MAYO-C*. https://github.com/PQCMayo/MAYO-C. Online; accessed January 30, 2024. 2023.

[Beu+23c] Ward Beullens et al. *MAYO-M4*. https://github.com/PQCMayo/MAYO-M4. Online; accessed February 28, 2024. 2023.

[Beu+23d] Ward Beullens et al. *Nibbling MAYO: Optimized Implementations for AVX2 and Cortex-M4*. Cryptology ePrint Archive, Paper 2023/1683. https://eprint.iacr.org/2023/1683. 2023. URL: https://eprint.iacr.org/2023/1683.

[Beu+23e] Ward Beullens et al. "Oil and Vinegar: Modern Parameters and Implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (2023), pp. 321–365.

[BFP09] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. "Hybrid approach for solving multivariate systems over finite fields". In: *Journal of Mathematical Cryptology* 3.3 (2009), pp. 177–197.

[Bos+18a] Joppe Bos et al. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 353–367.

[Bos+18b] Joppe Bos et al. *Kyber*. https://github.com/pq-crystals/kyber. Online; accessed February 25, 2024. 2018.

[D'A+18] Jan-Pieter D'Anvers et al. "Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM". In: *Progress in Cryptology – AFRICACRYPT 2018*. Springer. 2018, pp. 282–305.

[Din+08] Jintai Ding et al. "New Differential-Algebraic Attacks and Reparametrization of Rainbow". In: *Applied Cryptography and Network Security*. Springer. 2008, pp. 242–257.

[DR99] Joan Daemen and Vincent Rijmen. *AES proposal: Rijndael*. 1999.

[DS05] Jintai Ding and Dieter Schmidt. "Rainbow, a New Multivariable Polynomial Signature Scheme". In: *Applied Cryptography and Network Security*. Springer. 2005, pp. 164–175.

[Dwo01] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation. Methods and Techniques*. 2001. DOI: https://doi.org/10.6028/NIST.SP.800-38A.

[Dwo+01] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[Dwo15] Morris Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202.

[FD85] Harriet Fell and Whitfield Diffie. "Analysis of a Public Key Approach Based on Polynomial Substitution". In: *Advances in Cryptology – CRYPTO '85 Proceedings*. Springer. 1985, pp. 340–349.

[FG18] Ahmed Ferozpuri and Kris Gaj. "High-speed FPGA Implementation of the NIST Round 1 Rainbow Signature Scheme". In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2018, pp. 1–8.

[FNT21] Hiroki Furue, Shuhei Nakamura, and Tsuyoshi Takagi. "Improving Thomae-Wolf Algorithm for Solving Underdetermined Multivariate Quadratic Polynomial Problem". In: *Post-Quantum Cryptography*. Springer. 2021, pp. 65–78.

[Gep17] Pawel Gepner. "Using AVX2 Instruction Set to Increase Performance of High Performance Computing Code". In: *Computing and Informatics* 36.5 (2017), pp. 1001–1018.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[Gri+23] Arianna Gringiani et al. *MAYO: Optimized Implementation with Revised Parameters for ARMv7-M*. Cryptology ePrint Archive, Paper 2023/540. https://eprint.iacr.org/2023/540. 2023. URL: https://eprint.iacr.org/2023/540.

[Gro96] Lov K. Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. 1996, pp. 212–219.

[Gue10] Shay Gueron. *Intel advanced encryption standard (AES) new instructions set*. Intel White Paper. 2010.

[Hea18] Michael T. Heath. *Scientific Computing: An Introductory Survey, Revised Second Edition*. Society for Industrial and Applied Mathematics, 2018.

[Hir+23] Florian Hirner et al. *Whipping the MAYO Signature Scheme using Hardware Platforms*. Cryptology ePrint Archive, Paper 2023/1267. https://eprint.iacr.org/2023/1267. 2023. URL: https://eprint.iacr.org/2023/1267.

[Hul+22] Andreas Hulsing et al. *SPHINCS+*. Selected Algorithms 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023. 2022.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. "Unbalanced oil and vinegar signature schemes". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1999, pp. 206–222.

[KS98] Aviad Kipnis and Adi Shamir. "Cryptanalysis of the oil and vinegar signature scheme". In: *Advances in Cryptology – CRYPTO '98"*. Springer. 1998, pp. 257–266.

[Lom11] Chris Lomont. *Introduction to Intel Advanced Vector Extensions*. Intel White Paper. 2011.

[MI88] Tsutomu Matsumoto and Hideki Imai. "Public Quadratic Polynomial-Tuples for Efficient Signature-Verification and Message-Encryption". In: *Advances in Cryptology – EUROCRYPT'88*. Springer. 1988, pp. 419–453.

[NIS22] NIST. *Call for Additional Digital Signature Schemes for the Post-Quantum Cryptography Standardization Process*. https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf. Online; accessed January 22, 2024. 2022.

[Pat96] Jacques Patarin. "Asymmetric Cryptography with a Hidden Monomial". In: *Advances in Cryptology – CRYPTO'96*. Springer. 1996, pp. 45–60.

[Pat97] Jacques Patarin. "The Oil and Vinegar signature scheme". In: *Dagstuhl Workshop on Cryptography*. Sept. 1997.

[Pet+11] Albrecht Petzoldt et al. "Small Public Keys and Fast Verification for Multivariate Quadratic Public Key Systems". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Springer. 2011, pp. 475–490.

[Pre+22] Thomas Prest et al. *FALCON*. Selected Algorithms 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023. 2022.

[Rei17] James R Reinders. *Intel AVX-512 Instructions*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html. Online; accessed February 01, 2024. 2017.

[Say+23] Oussama Sayari et al. *HaMAYO: A Reconfigurable Hardware Implementation of the Post-Quantum Signature Scheme MAYO*. Cryptology ePrint Archive, Paper 2023/1135. https://eprint.iacr.org/2023/1135. 2023. URL: https://eprint.iacr.org/2023/1135.

[Sch+22] Peter Schwabe et al. *CRYSTALS-KYBER*. Selected Algorithms 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023. 2022.

[Sho94] P. W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings of the 35th Annual Symposium on Founda-*

tions of Computer Science. SFCS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 124–134. ISBN: 0-8186-6580-7. DOI: 10.1109/SFCS.1994.365700. URL: http://dx.doi.org/10.1109/SFCS.1994.365700.

[Tan+11] Shaohua Tang et al. "High-Speed Hardware Implementation of Rainbow Signature on FPGAs". In: *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011*. Springer. 2011, pp. 228–243.

[TW12] Enrico Thomae and Christopher Wolf. "Solving Underdetermined Systems of Multivariate Quadratic Equations Revisited". In: *Public Key Cryptography – PKC 2012*. Springer. 2012, pp. 156–171.

[Wika] Wikipedia contributors. *Advanced Vector Extensions – CPUs with AVX2*. https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#CPUs_with_AVX2. Online; accessed February 01, 2024.

[Wikb] Wikipedia contributors. *AVX-512 – CPUs with AVX-512*. https://en.wikipedia.org/wiki/AVX-512#CPUs_with_AVX-512. Online; accessed February 01, 2024.

[Wikc] Wikipedia contributors. *List of quantum processors*. https://en.wikipedia.org/wiki/List_of_quantum_processors [Online; accessed 05-May-2023]. URL: https://en.wikipedia.org/wiki/List_of_quantum_processors.

[Xil19] Xilinx, Inc. *7 Series FPGAs Memory Resources*. https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources. Online; accessed February 08, 2024. 2019.

[Yan+07] Bo-Yin Yang et al. "Analysis of QUAD". In: *Fast Software Encryption*. Springer. 2007, pp. 290–308.