Whipping the MAYO Signature Scheme using Hardware Platforms

Florian Hirner[1], Michael Streibl[2], Ahmet Can Mert[1] and Sujoy Sinha Roy[1]

[1] IAIK, Graz University of Technology, Graz, Austria
{florian.hirner,ahmet.mert,sujoy.sinharoy}@iaik.tugraz.at
[2] Graz University of Technology, Graz, Austria
michael.streibl@student.tugraz.at

Abstract.
NIST recently issued a new call to diversify the portfolio of quantum-resistant digital signature schemes since the current portfolio solely relies on lattice problems. A promising candidate for this new call is the MAYO scheme that builds on the Unbalanced Oil and Vinegar (UOV) problem. The MAYO scheme introduces emulsifier maps and a novel whipping technique to significantly reduce the signature and key sizes compared to previous UOV schemes. This paper provides a comprehensive analysis of the MAYO scheme and proposes multiple adaption and optimization techniques for an efficient hardware accelerator. The first proposed adaption is that we sample data on-the-fly and immediately use it for computation which saves a significant amount of memory. The second adaption is the replacement of the slow data sampling via AES128 by the faster SHAKE128. This improves the overall performance of data sampling in hardware while reducing resource consumption. We further increase the performance of our architecture via a novel memory structure capable of parallelizing major computations in the MAYO scheme. In addition, we also present a flexible transposing technique for the data format used in MAYO. We use these techniques to design a hardware accelerator that supports all operations of the MAYO scheme. The supported operations include key generation, signing, and verification for different NIST security levels. Comparisons show that our design massively outperforms HaMAYO [SMA+23] and UOV [BCH+23] by one to three orders of magnitude. HaMAYO has a 83× and 71× higher latency for key generation and signature generation, respectively. Comparisons with UOV show a performance increase of 1016×, 460×, and 607× in key generation for NIST security levels 1, 3, and 5, respectively. Furthermore, our signature generation and verification show a performance benefit of two orders of magnitude compared to both works. In addition to performance improvement, the presented optimized memory management shows a 2× to 3× lower BRAM consumption for multivariate schemes on FPGA platforms.

Keywords: MAYO, PQC, FPGA, ASIC, Digital Signatures

1 Introduction

Public-key cryptography encompasses essential cryptographic primitives for key exchange, public-key encryption, key encapsulation mechanism, and digital signature algorithm. Widely used public-key cryptographic algorithms are based on integer factorization or discrete logarithm problems. These problems are presumed to be computationally infeasible to solve using present-day computers. However, the emergence of a large-scale quantum computer poses a tangible threat to cryptographic primitives based on the above-mentioned mathematical problems as Shor's quantum algorithm [Sho94] can solve them in polynomial time. Over the last few years, quantum computer designs have seen accelerated

advancements. Prominent developments in this field include IBM's 5-qubit Tenerife in 2016, Google's 53-qubit-effective Sycamore in 2019, USTC's 76-qubit Jiuzhang in 2020, IBM's 127-qubit Eagle in 2021, Xanadu's 216-qubit Borealis and IBM's 433-qubit Osprey in 2022, among others [Wik]. Considering such rapid advancements, cybersecurity agencies, industries, and research institutions strive to facilitate a smooth transition to quantum-resistant public-key cryptography, commonly known as Post-Quantum Cryptography (PQC).

PQC algorithms can be grouped into five major categories based on their foundational mathematical problems: code-based, hash-based, isogeny-based, lattice-based, and multivariate-based. Each category has its unique mathematical and practical characteristics, strengths, and constraints. To standardize PQC algorithms, the US standardization organization NIST initiated the project "Post Quantum Cryptography Standardization" in 2016 and called for proposals. After three rounds of evaluations, in July 2022, NIST selected one key-encapsulation mechanism, namely Crystals-Kyber [SAB+22], and three signature algorithms, namely Crystals-Dilithium [BDK+22], Falcon [PFH+22], and SPHINCS+ [HBD+22] for standardization. Of these algorithms, the first four use lattice-based constructions. As the selected algorithms lacked sufficient diversity, in 2022, NIST issued a new call [NIS] specifically for additional post-quantum signature schemes. The list of submissions to this call has various signature algorithms relying on code-based, multivariate-based, MPC-based, and isogeny-based constructions.

MAYO [Beu22, BCC+23] is a new post-quantum digital signature scheme based on the Unbalanced Oil and Vinegar (UOV) construction [KPG99], a multivariate quadratic signature scheme. MAYO is also submitted to NIST's new diversification call for quantum-resistant digital signatures, and it is one of eleven signature schemes using multivariate cryptography. MAYO reduces the key size significantly by using a minimal oil space. Furthermore, it requires using a special *whipping up* technique to avoid falling out of the oil and vinegar map. This technique makes MAYO more compact than state-of-the-art lattice-based signature schemes such as Falcon and Dilithium.

For a new public-key cryptographic signature scheme to be viable for real-world applications, it must be efficiently computable on diverse software and hardware platforms. To examine the speed, memory, and energy/power efficiency of the cryptographic primitive, implementation methods must be researched, considering specific application and platform requirements. In the first three rounds of the NIST standardization project, we have seen numerous papers investigating the secure and efficient implementation aspects of novel PQC algorithms on high-end software, resource-constrained microcontroller, FPGA and ASIC hardware, and other platforms. When MAYO's implementations are considered, only a few available implementations are available in the literature.

The MAYO team provides a reference software implementation and an optimized version. The optimized version boosts the performance by utilizing AES-NI and AVX2 instructions during computations [PQM]. Another work [GMSS23] focuses on porting and optimizing the MAYO scheme for ARM microcontrollers, where they propose new parameters to improve the signing and verification processes. Recently, an FPGA implementation of the MAYO scheme is proposed [SMA+23] that implements a part of the scheme. Next to this, there is one work [BCH+23] that implements the underlying UOV scheme on which MAYO is based. This work analyses implementation techniques to port UOV to microcontrollers and FPGAs. Most of the operations performed in MAYO are similar to UOV. In contrast, the main difference is that the emulsification operation reduces the overall sizes of the public and private keys. Due to this, the UOV work is essential for comparisons since it is the only work that fully implements this scheme.

Contributions: Our contributions are summarized as follows. First, we show how a hardware implementation of the MAYO scheme benefits from on-the-fly data generation. Our approach does not store the generated data in memory but immediately consumes it for computations, allowing us to halve the required on-chip memory consumption on

FPGAs. The impact on memory savings on ASIC platforms is even higher due to the finer granularity of memory size. Second, we present a different approach to sample pseudo-random data faster by using SHAKE128 instead of AES128. In contrast to software using AES-NI instructions to accelerate AES128, a hardware accelerator benefits from omitting dedicated hardware to perform AES128. The MAYO scheme already requires data hashing via SHAKE256, which can also be used for SHAKE128. Hence, it becomes possible in the hardware to support SHAKE128 without any significant increase in logic instead of supporting AES128. Another advantage of replacing AES128 with SHAKE128 is the higher data sampling rate of almost 5× for round-based hardware implementations. Our third contribution presents a novel memory structure that is both performant and memory-saving simultaneously. The memory structure allows a high degree of parallelization of required computations. This parallelization further allows us to increase the performance by unrolling nested loops during signature generation and verification. Fourth, we propose a novel flexible matrix transpose module design that operates on our optimized memory structure. The transpose unit is flexible regarding throughput, meaning that a trade-off between latency and resource utilization is possible at design time. Hence, a higher throughput leads to a lower latency and higher area consumption and vice versa. We combine all these techniques to design a hardware accelerator that supports all operations of the MAYO scheme. The supported operations include key generation, signing, and verification for different NIST security levels. Our hardware accelerator decreases the latency of key generation by two orders of magnitude and signature generation and verification by one order of magnitude. We tested and verified our optimized design on FPGA and verified its functionality via the reference implementation of the MAYO team. In addition to this, we also give implementation results for ASIC using 28nm technology.

Outline: In Section 2, we provide the background, such as finite field arithmetic, multivariate quadratic maps, and the Oil and Vinegar signature scheme [KPG99]. In Section 2.4, we describe the MAYO signature scheme and give a detailed explanation of its specifications, like their whipping technique, emulsifier maps, and more. Section 3 gives an in-depth explanation of our optimization strategies. Section 4 gives an in-depth explanation of our hardware implementation and in Section 5, we present the results. Moreover, in Section 3, we present several optimizations to further improve hardware implementations and Section 6 concludes the paper.

2 Background

This section covers the background necessary to understand arithmetic used in the UOV [BCH+23] and MAYO [Beu21] scheme.

2.1 Finite field arithmetic's over $\mathbf{GF(2^4)}$

The arithmetic in the MAYO digital signature algorithm is mainly based on vector and matrix operation in the finite field $\mathrm{GF}(2^4)$. Elements in this field can be represented as a polynomial of degree 3, e.g., $a = a_3x^3 + a_2x^2 + a_1x + a_0$, where a_3, a_2, a_1, a_0 are elements of $\mathrm{GF}(2)$. For the rest of the paper, we use the following encoding, an element $a \in \mathrm{GF}(2^4)$ is encoded as an unsigned 4-bit integer, whose 4 bits are the coefficients of the polynomial, e.g., $\mathrm{Encode}(a = a_3x^3 + a_2x^2 + a_1x + a_0) = (a_3a_2a_1a_0)_2$. For example, $\mathrm{Encode}(1x^3 + 0x^2 + 1x + 0)$ is equal to $(1010)_2$, which is 10 in decimal.

2.1.1 $\mathbf{GF(2^4)}$ addition and subtraction

Addition and subtraction 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 as polynomial addition and subtraction, respectively.

Therefore, we implement $\mathrm{GF}(2^4)$ addition and subtraction as shown in Eq. (1), where $\oplus$ represents bit-wise XOR operation. Since the coefficients of the $\mathrm{GF}(2^4)$ elements are in $\mathrm{GF}(2)$ and addition is equivalent to subtraction in this field, we are able to use a single operation for both.

$$a \pm b = (a_3 \pm b_3)x^3 + (a_2 \pm b_2)x^2 + (a_1 \pm b_1)x + (a_0 \pm b_0) = a \oplus b \tag{1}$$

2.1.2 $\mathrm{GF}(2^4)$ multiplication

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 as a polynomial multiplication. However, a standard polynomial multiplication can result in a polynomial with a degree greater than 3, which is not an element of $\mathrm{GF}(2^4)$. Therefore, a reduction operation is required to bring the resulting polynomial to $\mathrm{GF}(2^4)$. The MAYO scheme uses $x^4 + x + 1$ as the reduction polynomial. The $\mathrm{GF}(2^4)$ multiplication with x^4+x+1 reduction polynomial is shown in Eq. (2), where $\wedge$ represents bit-wise AND operation.

$$\begin{aligned}
c =& a \times b = (c_3c_2c_1c_0)_2, \quad \text{where} \\
c_0 =& (a_0 \wedge b_0) \oplus (a_1 \wedge b_3) \oplus (a_2 \wedge b_2) \oplus (a_3 \wedge b_1) \\
c_1 =& (a_0 \wedge b_1) \oplus (a_1 \wedge b_0) \oplus (a_1 \wedge b_3) \oplus (a_2 \wedge b_2) \oplus (a_3 \wedge b_1) \oplus (a_2 \wedge b_3) \oplus (a_3 \wedge b_2) \\
c_2 =& (a_0 \wedge b_2) \oplus (a_1 \wedge b_1) \oplus (a_2 \wedge b_0) \oplus (a_2 \wedge b_3) \oplus (a_3 \wedge b_2) \oplus (a_3 \wedge b_3) \\
c_3 =& (a_0 \wedge b_3) \oplus (a_1 \wedge b_2) \oplus (a_2 \wedge b_1) \oplus (a_3 \wedge b_0) \oplus (a_3 \wedge b_3)
\end{aligned} \tag{2}$$

This bitsliced approach, with the fast bitselection capability of hardware compared to software, enables implementing $\mathrm{GF}(2^4)$ multiplication in an efficient but still simple form in hardware.

2.2 Multivariate Quadratic Maps

The core of the Oil and Vinegar [KPG99] and the MAYO scheme are multivariate quadratic maps. We follow the definition and notation presented in [Beu22]. Such a map $P(\mathbf{x}) = (p_1, \ldots, p_m) : \mathbb{F}_q^n \to \mathbb{F}_q^m$ consists of m multivariate quadratic polynomials in n variables. This map is evaluated by simply evaluating each polynomial p_i. MAYO uses the upper triangular matrix form of multivariate quadratic polynomials. Therefore, polynomial evaluation is defined as

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

Since there are m different multivariate quadratic polynomials, we end up with m different $\mathbf{P}_i$ matrices, which need to be evaluated. Therefore, the result of the multivariate quadratic map is defined as $P(\mathbf{a}) = \mathbf{b}$ with $\mathbf{b} = (p_1(\mathbf{a}), \ldots, p_m(\mathbf{a}))$.

2.3 Oil and Vinegar

The foundation of the MAYO scheme is the so-called Oil and Vinegar scheme. The description and notation of the Oil and Vinegar signature scheme is adapted from [Beu22]. The central object of this scheme is the multivariate quadratic map, which acts as a public key in the scheme. To sign a message M, it first obtains its digest using a cryptographic hash function H and a random *salt*. Then, the signature $\mathbf{s}$ is the preimage under the multivariate quadratic map P of the specific digest value such that $P(\mathbf{s}) = H(M||salt)$. However, since sampling preimages for multivariate quadratic maps, known as MQ problem, is considered hard, we need a trapdoor to obtain them efficiently. The trapdoor information

in the Oil and Vinegar scheme is the so-called Oil space, a linear subspace $O \subset \mathbb{F}_q^n$ where P vanishes, meaning that

$$P(\mathbf{o}) = 0 \quad \text{for all } \mathbf{o} \in O. \tag{4}$$

Knowledge of the oil space allows to efficiently sample preimages of P. To understand how this information helps to generate the signature, the polar form of quadratic polynomials is needed. Every homogeneous multivariate quadratic polynomial has an associated symmetric and bilinear form $p'(\mathbf{x}, \mathbf{y}) = p(\mathbf{x} + \mathbf{y}) - p(\mathbf{x}) - p(\mathbf{y})$. Similarly, the polar form of a multivariate quadratic polynomial map consisting of m polynomials is defined as

$$P'(\mathbf{x}, \mathbf{y}) = P(\mathbf{x} + \mathbf{y}) - P(\mathbf{x}) - P(\mathbf{y}). \tag{5}$$

Given a target $\mathbf{t} \in \mathbb{F}_q^m$, one selects a vector $\mathbf{v} \in \mathbb{F}_q^n$ and solves $P(\mathbf{v} + \mathbf{o}) = \mathbf{t}$ for $\mathbf{o} \in O$. From Eq. (5), it follows that

$$P(\mathbf{v} + \mathbf{o}) = P(\mathbf{v}) + P(\mathbf{o}) + P'(\mathbf{v}, \mathbf{o}) = \mathbf{t}. \tag{6}$$

Since $P(\mathbf{v})$ is fixed and due to Eq. (4), 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 $\mathbf{s} = \mathbf{v} + \mathbf{o}$. The security of the signature algorithm is based on the MQ problem, which is considered NP-hard if $n \sim m$, even for quantum computers [Beu22]. However, the Oil and Vinegar scheme suffers from large public key sizes in the order of 50 KB, which renders the scheme unsuitable as a practical signing algorithm.

2.4 MAYO Scheme

In this section, we give a short description of MAYO scheme. The description and notation of MAYO scheme is adapted from [Beu22] according to the latest specifications described in [BCC+23]. Readers may refer to [Beu22, BCC+23] for more details. The MAYO schemes modifies the original Oil and Vinegar scheme to tackle the problem of large key sizes. It introduces a *whipping* mechanism, which transforms the multivariate quadratic map $P : \mathbb{F}_q^n \to \mathbb{F}_q^m$ into a larger map $P^* : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$. This construction allows to choose a smaller oil space and as a consequence reduces the key size significantly. Before we explain the whipping construction in detail, we need to examine why the dimension of the oil space is the determining factor in the size of the public key.

2.4.1 Public Key Size

The public key in the Oil and Vinegar scheme is the multivariate quadratic map P consisting of m multivariate quadratic polynomials in n variables. Thus, the memory requirement for storing P is $mn^2 \log q$ due to the upper triangular matrix form of a polynomial defined in Eq. (3). Petzoldt *et al.* [PTBW11] showed 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 and, as a result, only $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{o\times o}$ needs to be stored as public key. This method reduces the key size to $mo^2 \log q$. However, the original Oil and Vinegar scheme requires o to be at least as large as m, otherwise the linear system obtained from Eq. (6) is unsolvable with high probability. The MAYO scheme proposes a novel whipping technique to allow a further reduction of the public key by reducing the dimension of the oil space.

2.4.2 Whipping Technique

As mentioned in Section 2.4, MAYO transforms P up into a larger map P^*. This whipping transformation must have the property that if P vanishes on a subspace $O \subset \mathbb{F}_q^n$ then P^* needs to vanish on $O^k \subset \mathbb{F}_q^{kn}$, where k is the whipping parameter which controls the size of the oil space with $o = \lceil m/k \rceil$. The concrete whipping operation is defined as

$$P^*(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{x}_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{x}_i, \mathbf{x}_j). \tag{7}$$

The matrices $\mathbf{E}_{ij} \in \mathbb{F}_q^{m \times m}$ are the so-called emulsifier maps and fundamental for the security of the whipping technique. These emulsifier maps are described in-detail in Sec.2.5. Further, the signature of MAYO can be sampled similar to Eq. (6) of UOV by solving the linear system of Eq. (8)

$$P^*(\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k) = \mathbf{t}, \tag{8}$$

which has m equations in ko variables.

2.4.3 Scheme Description

In this section, we briefly describe the key generation, signature generation and signature verification algorithms of MAYO.

Key Generation: To generate a key-pair, a randomly-generated seed is expanded and its output is used as matrix $\mathbf{O} \in \mathbb{F}_q^{(n-o) \times o}$. $\mathbf{O}$ is the secret key and the according oil space O is the rowspace of $(\mathbf{O}^\top \mathbf{I}_o)$, where $\mathbf{I}_o$ denotes the identity matrix of size o. As described in Eq. (4), the multivariate quadratic map P must vanish on O. Thus, a polynomial $p_i(\mathbf{x})$ of P has to fulfill

$$(\mathbf{O}^\top \mathbf{I}_o) \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} (\mathbf{O}^\top \mathbf{I}_o)^\top = \mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{O}^\top \mathbf{P}_i^{(2)} + \mathbf{P}_i^{(3)} = 0. \tag{9}$$

Therefore, it is possible to generate $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ pseudo-randomly from a seed and set $\mathbf{P}_i^{(3)}$ to $\textsc{Upper}(\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{O}^\top \mathbf{P}_i^{(2)})$, where $\textsc{Upper}(\cdot)$ is defined as $\text{Upper}(\mathbf{M}_{ii}) = \mathbf{M}_{ii}$ and $\textsc{Upper}(\mathbf{M}_{ij}) = \mathbf{M}_{ij} + \mathbf{M}_{ji}$ for $i < j$. Generating large parts of the matrices pseudo-randomly enables the significant key size reduction since we do not need to store the whole key information. Instead we generate parts of the public and private key based on the respective seed. In case of private key we now only need to store the private seed while the public key consists of public seed and $\mathbf{P}_i^{(3)}$. Additionally, the whipping transformation described in Section 2.4.2 reduced the size of $\mathbf{P}_i^{(3)}$ from $m \times m$ to $o \times o$.

Signature Generation: To compute a signature of a message M, a random salt is generated and the digest $\mathbf{t} = H(M||salt)$ is computed. Afterwards, one chooses vectors $(\mathbf{v}_1, \ldots \mathbf{v}_k)$ randomly and solves the linear system for $(\mathbf{o}_1, \ldots \mathbf{o}_k)$ as shown in Eq. (8). As described by Beullens *et al.* [BCC+23], the last o entries of $\mathbf{v}_i$ can be set to 0 without affecting the distribution of the signing output. Thus, one generates $\tilde{\mathbf{v}}_i \in \mathbb{F}_q^{(n-o)}$ randomly and sets $\mathbf{v}_i$ to $(\tilde{\mathbf{v}}_i, 0)$. As a result of this choice, only $\mathbf{P}_i^{(1)}$ is needed for the signature computation. Similar to Eq. (6), the oil space trapdoor information enables the partition of Eq. (8) into a constant and a linear part, which leads to

$$\begin{aligned} P^*(\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k) &= \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{v}_i + \mathbf{o}_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{v}_i + \mathbf{o}_i, \mathbf{v}_j + \mathbf{o}_j) \\ &= \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{v}_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{v}_i, \mathbf{v}_j) \quad \text{(constant)} \\ &+ \sum_{i=1}^{k} \mathbf{E}_{ii} P'(\mathbf{v}_i, \mathbf{o}_i)) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathbf{E}_{ij} (P'(\mathbf{v}_i, \mathbf{o}_j) + P'(\mathbf{v}_j, \mathbf{o}_i)) \quad \text{(linear)} \\ &= \mathbf{t}. \end{aligned} \tag{10}$$

The constant part can be calculated using

$$\begin{aligned} p_i(\mathbf{v}_k) &= \tilde{\mathbf{v}_k}^\top \mathbf{P}_i^{(1)} \tilde{\mathbf{v}_k}, \\ p'_i(\mathbf{v}_k, \mathbf{v}_l) &= \tilde{\mathbf{v}_k}^\top \mathbf{P}_i^{(1)} \tilde{\mathbf{v}_l} + \tilde{\mathbf{v}_l}^\top \mathbf{P}_i^{(1)} \tilde{\mathbf{v}_k}. \end{aligned} \tag{11}$$

For the computation of the linear part, the evaluation of the linear transformation $P'(\mathbf{v}_k, \cdot)$ has to be carried out. To achieve that, the matrix representation of the linear transformation can be used, which is defined as

$$\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top})\mathbf{O} + \mathbf{P}_i^{(2)}. \tag{12}$$

Then, each component $p'_i(\mathbf{v}_k, \cdot)$ of P' is defined as $\tilde{\mathbf{v}_k}^\top \mathbf{L}_i$. Applying Eq. (11) and Eq. (12) to Eq. (10) results in the augmented matrix which needs to be solved for $\mathbf{o}_i$ to compute the signature. The linear system can be solved using one of the many available algorithms, e.g., Gaussian elimination.

Signature Verification: Given a message M and a signature $(salt||\mathbf{s}_1, \dots \mathbf{s}_k)$, only the digest $\tilde{\mathbf{t}} = H(M||\text{salt})$ is obtained and the whipped up map $P^*(\mathbf{s}_1, \dots \mathbf{s}_k) = \mathbf{t}$ is evaluated. If $\mathbf{t} = \tilde{\mathbf{t}}$, the signature is accepted, otherwise rejected.

2.5 Emulsifier maps

One vital component of the MAYO signature scheme is the so-called emulsifier maps $\mathbf{E} \in \mathbb{F}_q^{m \times m}$. Their usage is the main difference to the original Oil and Vinegar algorithm and the reason for the compact public key size. $\mathbf{E}$ corresponds to a multiplication by z in a finite field $\mathbb{F}_q[z]/f(z)$ and they are used in computations of the form $\mathbf{E}^l\mathbf{u}$, where $\mathbf{u}$ denotes a vector of length m and l takes values from 0 to $\frac{k(k+1)}{2} - 1$. However, instead of computing the matrix multiplications explicitly, it is more efficient, especially regarding memory access limits in hardware, to interpret $\mathbf{u}$ as single polynomial and perform the reduction mod $f(z)$ once, which resembles a multiplication in the finite field $\text{GF}((2^4)^m)$. Similar to the finite field described in Section 2.1, elements of $\text{GF}((2^4)^m)$ can be represented as a polynomial, however, this time of degree $m-1$ and with coefficients in $\text{GF}(2^4)$. Therefore, $a \in \text{GF}((2^4)^m)$ is of the form

$$a = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_1 z + a_0. \tag{13}$$

The emulsifier map $\mathbf{E}$ now represents a multiplication by z. Analog to the field multiplication in Section 2.1.2, we need to reduce the resulting polynomial, to receive a valid $\text{GF}((2^4)^m)$ element again. In this case, the reduction polynomial is $z^{64} + 8z^3 + 2z^2 + 8$. To apply $\mathbf{E}$ to a vector $\mathbf{a}$, we interpret $\mathbf{a}$ as polynomial of the form seen in Eq. (13), and perform the following computations:

$$\begin{aligned} b = \mathbf{E}a, \quad \text{with} \quad b_i = a_{i-1} \quad \text{for } i \notin \{0, 2, 3\}. \\ b_0 = 8a_{m-1}, \quad b_2 = 2a_{m-1} + a_1 \quad b_3 = 8a_{m-1} + a_2 \end{aligned} \tag{14}$$

It is important to note that the additions and multiplications in Eq. (14) are $\text{GF}(2^4)$ operations. This approach blends well with our packed format described in Section 4.2, as we are able to load m values and, therefore, a whole $\text{GF}((2^4)^m)$ element in one cycle in hardware. To evaluate $\mathbf{E}^l\mathbf{u}$, we perform this computation l times.

3 Optimization Strategies

In this section, we show several high-level optimizations on an algorithmic level to improve the performance and memory consumption for the MAYO scheme. These optimizations are focused on lowering memory consumption as well as improving latency for MAYO on hardware platforms.

Table 1: $\mathbf{P}$ matrix sizes for different NIST security levels (1, 3, 5)

Matrix	MAYO_1	MAYO_3	MAYO_5
$\mathbf{P}^{(1)}$	58×58	89×89	121×121
$\mathbf{P}^{(2)}$	58×8	89×10	121×12
$\mathbf{P}^{(3)}$	8×8	10×10	12×12
$\mathbf{P}$	66×66	99×99	133×133

3.1 On-the-fly Coefficient Generation

The $\mathbf{P}_{i \in m}$ matrices are the fundamental building block of the MAYO scheme. MAYO splits each $\mathbf{P}_i$ matrix into three sub-matrices $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$ as shown in Eq. (3). As the sizes for each security levels vary we show the corresponding matrices sizes in Table 1. The large size of the $\mathbf{P}_i$ matrices lead to a high memory consumption making it an important aspect in the designing stage. Let us consider the smallest level MAYO_1 with parameters $(n = 66; m = 64; o = 8; k = 9; q = 16)$ as an example to give an impression of the total memory consumption. There are m $\mathbf{P}_i$ matrices each with size $n \times n$ leading to a total of $(n \times n) \times m = 278,784$ elements. Each matrix element is in $\text{GF}(2^4)$ and we need 4 bit to represent each element, we would have to store 136KB in memory for $\mathbf{P}_i$ matrices. In regards to constraint platforms this size exceeds the memory capacity. Hence, we reduce the required memory consumption of each $\mathbf{P}_i$ from $n \times n$ elements to just $o \times o$ elements. This is possible due to the fact that the coefficients of the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices can be generated pseudo-randomly based on the public seed. To be precise, every time some $\mathbf{P}_i^{(1)}$ or $\mathbf{P}_i^{(2)}$ matrix is needed in an operation, it is possible to generate the matrix element instead of retrieving their elements from on-chip storage. Thus, it is only desired to store the $\mathbf{P}_i^{(3)}$ matrices, which reduces the memory demand from 136KB to 2KB for MAYO_1.

3.2 Switching Coefficient Generation from AES128-CTR to SHAKE128

In the latest specifications of MAYO [BCC+23], both Aes128 and Shake256 are used to generate data for $\mathbf{P}_i^{(1)}$ or $\mathbf{P}_i^{(2)}$. The rationale behind this decision is to use Aes128-CTR for the major part of data generation so the fast AES-NI extension of modern CPUs can be utilized to improve the performance of signature generation. However, this approach poses problems in a hardware implementation when it comes to area usage and performance.
Area usage: To implement the MAYO scheme with original specifications on an FPGA, we need to incorporate two cores, one for Aes and one for Shake. Thus, a large part of the area demand of our current version is caused by these two cores. Since both cores share the same use case, namely generating pseudo-random data, this approach creates redundancy on the hardware level.
Performance: The benefit of Aes in software relies on Aes-NI instruction-set support on modern CPUs. Hardware platforms can not benefit from these meaning that a round based Shake outperforms Aes significantly on FPGA platforms. Shake generates 1344 bit every 26 cycles, which is $4,84\times$ faster compared to 128 bit every 12 cycles of Aes.

Therefore, using solely Shake instead of using both SHAKE and Aes for generating random data can increase the performance of the algorithm on hardware and, furthermore, reduce the area demand of the implementation. Additionally, the optimizations described in Section 3.1 and Section 3.3 are also compatible with SHAKE.

3.3 Parallel Matrix Column Multiplication

Generating the coefficient of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ on-the-fly enables reducing the memory usage significantly. This also allows us to carry out the matrix operations efficiently. Matrix multiplication can be broken down into several vector-vector multiplications, as shown in

Eq. (15). Each row vector of matrix $\mathbf{A}$ is multiplied with each column vector of matrix $\mathbf{B}$ in a multiply-and-accumulate (MAC) fashion. Every vector-vector multiplication obtains one element of the result matrix $\mathbf{C}$.

$$\underbrace{\begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \end{pmatrix}}_{\mathbf{A}} \times \underbrace{\begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \\ b_4 & b_5 \end{pmatrix}}_{\mathbf{B}} = \underbrace{\begin{pmatrix} c_0 & c_1 \\ c_2 & c_3 \end{pmatrix}}_{\mathbf{C}} \tag{15}$$

The elements sharing the same color can be multiplied in parallel. Therefore, we can parallelize the multiplication of one left-hand coefficient (f.e a_0) with the according row coefficients (b_0 and b_1) of the right-hand side. Hence we instantiate as many MAC units as there are columns in $\mathbf{B}$. In case of MAYO, the number of columns is fixed to k, meaning that either 9, 11, or 12 units are needed depending on the security level. This optimization reduces the latency of these operations by a factor of k.

3.4 Parallelizing Normal and Transpose Computation of $\mathbf{L}_i$

Generating coefficient of $\mathbf{P}_i^{(1)}$ in a row-wise order, as described in Sec. 3.2, requires an adaption of the computations in MAYO.EXPANDSK(). This affects the computation of $\mathbf{L}_i$ in step 17 of Alg. 6 of [BCC+23]. We adapt the computation of $\mathbf{L}_i$ as shown in Eq. (16).

$$\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top})\mathbf{O} + \mathbf{P}_i^{(2)} = \underbrace{\mathbf{P}_i^{(1)}\mathbf{O}}_{\text{MAC}} + \underbrace{\mathbf{P}_i^{(1)^\top}\mathbf{O}}_{\text{BMAC}} + \mathbf{P}_i^{(2)}. \tag{16}$$

We can see that two matrix multiplications are involved in our adapted computation. The left operands of each multiplication $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(1)^\top}$ are generated pseudo-randomly via SHAKE128, as described in detail Sec. 3.2. The first multiplication uses $\mathbf{P}_i^{(1)}$ as left operand which is generated in a row-wise order. This row-wise order is exactly the required order for a straightforward matrix-matrix multiplication as discussed in Sec. 3.3. Thus, we use a simple multiply and accumulate (MAC) unit for this computation. Yet, the row-wise generation order of $\mathbf{P}_i^{(1)}$ poses a challenge in the second matrix multiplication. The row-wise generation of $\mathbf{P}_i^{(1)}$ corresponds to a column-wise generation of $\mathbf{P}_i^{(1)^\top}$, which hinders a straightforward matrix-matrix multiplication. The following examples in Eq. (17) and Eq. (18) give an impression of this limitation and presents our adapted algorithm to solve this challenge. We consider a simple matrix-vector multiplication of $\mathbf{P}$ and $\mathbf{o}$ where $\mathbf{P}$ is generated in a row-wise manner.

$$\mathbf{u} = \mathbf{P}\mathbf{o} = \begin{pmatrix} p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,1} & p_{3,2} & p_{3,3} \end{pmatrix} \begin{pmatrix} o_1 \\ o_2 \\ o_3 \end{pmatrix} = \begin{pmatrix} p_{1,1}o_1 + p_{1,2}o_1 + p_{1,3}o_1 \\ p_{2,1}o_2 + p_{2,2}o_2 + p_{2,3}o_2 \\ p_{3,1}o_3 + p_{3,2}o_3 + p_{3,3}o_3 \end{pmatrix}. \tag{17}$$

Eq. (17) shows a standard matrix multiplication of $\mathbf{P}\mathbf{o}$. We obtain one element of the resulting vector $\mathbf{u}$ after consuming one full row of $\mathbf{P}$ and the column of $\mathbf{o}$, as shown in Eq. (17). Hence, we accumulate the computations colored in red inside a MAC unit until all elements of the respective row of $\mathbf{P}$ are consumed. This procedure is repeated for each row in $\mathbf{P}$ (colored blue and orange).

$$\mathbf{v} = \mathbf{P}^\top\mathbf{o} = \begin{pmatrix} p_{1,1} & p_{2,1} & p_{3,1} \\ p_{1,2} & p_{2,2} & p_{3,2} \\ p_{1,3} & p_{2,3} & p_{3,3} \end{pmatrix} \begin{pmatrix} o_1 \\ o_2 \\ o_3 \end{pmatrix} = \begin{pmatrix} p_{1,1}o_1 + p_{2,1}o_2 + p_{3,1}o_3 \\ p_{1,2}o_1 + p_{2,2}o_2 + p_{3,2}o_3 \\ p_{1,3}o_1 + p_{2,3}o_2 + p_{3,3}o_3 \end{pmatrix}. \tag{18}$$

Eq. (18) shows a similar matrix multiplication just with transposed $\mathbf{P}^\top$ as the first operand. Yet, we cannot simply use the same MAC unit as in the previous case since the

required elements for computing one element of $\mathbf{v}$ are no longer generated directly after each other. Therefore, we have to store the intermediate MAC results for each element of $\mathbf{v}$ in memory and retrieve them again for MAC-ing the following generated coefficients to carry out the accumulation. This approach referred to as BMAC enables us to compute the transposed matrix multiplication while maintaining the row-wise generation order.

Our presented BMAC approach is extendable to matrix-matrix multiplications. Each column vector of the right-hand side operand is processed in parallel by one dedicated BMAC unit, as discussed in Sec. 3.3. We apply this optimization to the computation of $\mathbf{P}_i^{(1)}\mathbf{O}$ and ${\mathbf{P}_i^{(1)}}^\top\mathbf{O}$ within the MAYO.EXPANDSK() function. Additionally, it is possible to parallelize the calculation of $\mathbf{P}_i^{(1)}\mathbf{O}$ and ${\mathbf{P}_i^{(1)}}^\top\mathbf{O}$ due to the fact that both matrix multiplications use the same elements during computation.

3.5 Block Matrix Multiplication during Signature Verification

In the signature verification, we need to compute Eq. (19). Due to the optimization described in Section 3.1, it is not possible to perform the matrix multiplication using the standard approach as the elements of $\mathbf{P}_i^{(2)}$ are generated after $\mathbf{P}_i^{(1)}$. Therefore, we apply a block matrix multiplication to Eq. (19) to calculate the results of $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$ individually. Afterward, we combine the intermediate results accordingly using vector addition. Thus, only the intermediate results of the multiplication $\mathbf{P}_i\mathbf{s}_i$ need to be stored, which are much smaller than the $\mathbf{P}$ matrices.

$$\mathbf{s}_i^\top \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} \mathbf{s}_i \tag{19}$$

4 The Proposed Hardware

In this section, the proposed hardware architectures and all of the main arithmetic blocks are explained in a bottom-up fashion. We start with the pseudo-random data sampling via SHAKE128. Then, we present our arithmetical units, memory management, and the overall design. Finally, the scheduling of operations during key generation, key expansion, signature generation, and signature verification is presented. Note that we use the parameters shown in Table 2 for each of the respected security levels during designing our hardware.

4.1 Hashing and Pseudo-Random Data Sampling

In the latest specifications of MAYO [BCC+23] both SHAKE256 and AES128-CTR are used. An analysis of the MAYO software implementation shows that only a small time-share of the execution is spent on hashing via SHAKE256. Yet, the major time-share is spent on pseudo-random data sampling via AES128-CTR. The sampling based on AES128-CTR in software benefits from the AES-NI instruction-set extension [ADF+12]. The AES-NI instructions invoke a built-in hardware accelerator for AES on high-end CPUs. This accelerates the major share of pseudo-random data sampling in MAYO that relies on AES128-CTR. It is not required to use AES128-CTR for pseudo-random data sampling, but other primitives are suitable as well. Further, supporting both AES128-CTR and SHAKE256 in a hardware implementation introduces a high overhead. In contrast to a hardware-based AES128-CTR implementation that produces an output of 128 bit every 12 cycles, a SHAKE128 hardware generates 1344 pseudo-random bits every 26 cycles. This is more than 4× faster compared to AES128-CTR. Hence, we can see that supporting both AES128-CTR and SHAKE256 in hardware is neither required nor optimal. The MAYO scheme already requires SHAKE256 meaning that supporting SHAKE128 does not introduce

Table 2: Overview of selected parameter set for each security level

Security Level	Parameter Set (n, m, o, k, q)	pk size	sig size	esk size	epk size
$MAYO_1$	(66, 64, 8, 9, 16)	1168 B	321 B	68 KB	70 KB
$MAYO_3$	(99, 96, 10, 11, 16)	2656 B	576 B	230 KB	233 KB
$MAYO_5$	(133, 128, 12, 12, 16)	5008 B	838 B	553 KB	557 KB

Table 3: Latency of pseudo-random sampling of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ for NISTs security levels

Sec. Level	Matrix	Generated elements	AES128 (cc)	SHAKE128 (cc)
$MAYO_1$	$\mathbf{P}^{(1)}$ / $\mathbf{P}^{(2)}$	1,711 / 464	41,064 / 11,136	8,473 / 2,314
$MAYO_3$	$\mathbf{P}^{(1)}$ / $\mathbf{P}^{(2)}$	4,005 / 890	144,180 / 32,040	29,744 / 4,420
$MAYO_5$	$\mathbf{P}^{(1)}$ / $\mathbf{P}^{(2)}$	7,381 / 1,452	354,288 / 69,696	73,112 / 7,202

any overhead. Therefore, we replace the AES128-CTR primitive with SHAKE128. This increases the performance of MAYO on hardware and reduces the area demand of the implementation at the same time.

The pseudo-random number generation in MAYO is mainly responsible to generate the two matrices $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. The reference software implementation of the MAYO team [BCC+23] uses AES128 to generate the pseudo-random data of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. In contrast to this, we use SHAKE128 to generate both of these matrices. The remaining part of this paragraph focuses on the performance benefit we achieve through this measure. The size of these two matrices depends on the selected security level defined by NIST, which is either 1, 3, or 5. In case of $\mathbf{P}_i^{(1)}$ the sizes is either 1711, 4005, or 7381 elements while the size of $\mathbf{P}_i^{(1)}$ is either 464, 890, or 1452 for each respective security level. It is possible to calculate the amount of time it takes to generate these matrices by using the equation $\lceil s_m \times s_e / r \rceil \times l$, where s_m, s_e, r, and l represents the size of the matrix, size of each element, the bit-rate, and the latency of the bit-rate of each generation cycle. Note, the bit-rate for AES128-CTR is 128 bit per 12 cycles and 1344 bit per 26 cycles for SHAKE128. Table 3 shows the total latency required for sampling via AES128-CTR and SHAKE128.

Compared both primitives shows that using SHAKE128 instead of AES128-CTR can be beneficial, when it comes to pseudo-random data sampling on hardware platforms. Further, a switch yields a speedup of 4.84× through all security levels. It would be possible to use multiple AES128-CTR modules concurrently to match the output rate of SHAKE128 due to the counter mode which making data round-independent. Yet, It would take a total of $\lceil 41064/8473 \rceil = 5$ AES128-CTR modules to match the rate of SHAKE128. This approach, however, is not practical in hardware due to a massive overhead in terms of area utilization.

4.2 Organization of On-Chip Memory

One key factor of an efficient hardware implementation is a well-designed memory layout. A major factor of our memory layout is that it needs to support fast loading of relevant elements since the MAYO scheme mainly consists of matrix and vector operations. These operations are either performed on packed or unpacked data, explained as followed:

1. **Unpacked**: One memory location of a BRAM stores a whole vector $\mathbf{v}$ or a row of the matrix $\mathbf{A}$, marked in orange in Fig. 1a. This format is used when computing the matrix $\mathbf{A}$ that is used in MAYO.SIGN() as well as in SAMPLESOLUTION() and EF(). This allows us to load a whole row within a single load cycle.

2. **Packed**: One memory location of a BRAM stores m elements $(p_{i,(x,y)})$ for $i = 1...m$,

marked in orange in Fig. 1b. The elements $(p_{i,(x,y)})$ is the (x, y)-th coefficient of the matrix $\mathbf{P}_{i\in m}$. The packed memory format is used each time a matrix in the form $\mathbf{P}_i$ is involved. This means that loading one BRAM location gives us m elements with the same indices from m different matrices.

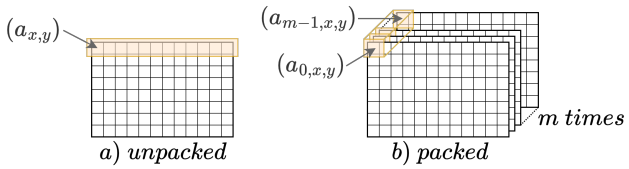

Figure 1: Overview of packed and unpacked memory format

The m multivariate quadratic polynomials operate on input values with the same indices, as shown in Fig. 1b. Therefore, we simply pack all the m different matrix elements with the same index into one BRAM entry as they share the same input value. Thus, the packed format allows us to efficiently load and evaluate the m different polynomials in parallel. Consequently, all vectors and matrices to the multivariate quadratic map are stored in the packed format, while the remaining are stored as unpacked.

Our parametric memory wrapper design is shown in Fig. 2, for its configuration in security level 1. The architecture of our core consists of an I/O buffer and a memory wrapper. The memory wrapper contains a total of $k+1$ many memory banks $(MEM_{i\in(k+1)})$, which are responsible for storing our packed and unpacked data during the operations in MAYO. One memory location of each memory bank is spread over the vertically arranged BRAMS, as marked in red. This memory location stores either data in packed or unpacked form as described in the listing above. We split the memory bank into two regions marked in yellow and orange, as shown in Fig. 2. The two memory banks MEM_1 and MEM_2 in the yellow region have a different word size. This is caused by the fact that it can store both packed and unpacked data. The word size of the packed data is 256b, which results from the size of an element in the GF(2^4) field and the number of m elements used in $\mathbf{P}_{i\in m}$. Yet, the unpacked data requires a word size of 292b instead of 256b, which is used to store a whole row of a matrix $\mathbf{A} \in \mathbb{F}_q^{m\times ko+1}$. The orange region on the right size of Fig. 2 supports a word size of 256b to store only packed data. In addition to that, the number of memory banks within this region depends on the parameter k of the chosen security level.

4.2.1 Transpose of Packed and Unpacked Matrix

Our architecture needs to support two different types of transpose operation due to the packed and unpacked data format. Transposing data in packed format is trivial since it only requires switching the data at certain indexes inside a BRAM. Meaning that we need to load an element a from index i_a and another element b from index i_b and store a on

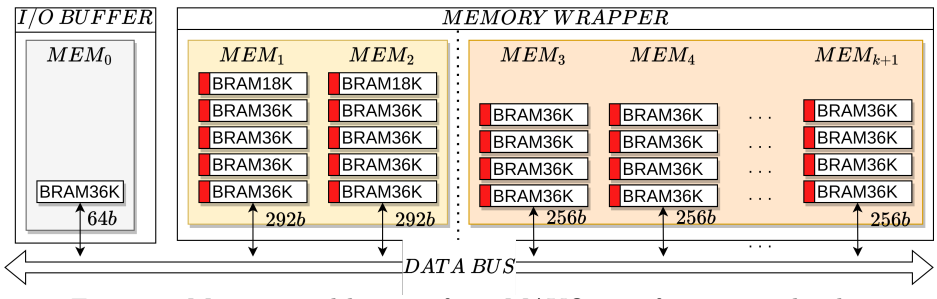

Figure 2: Memory grid layout of our MAYO core for security level 1

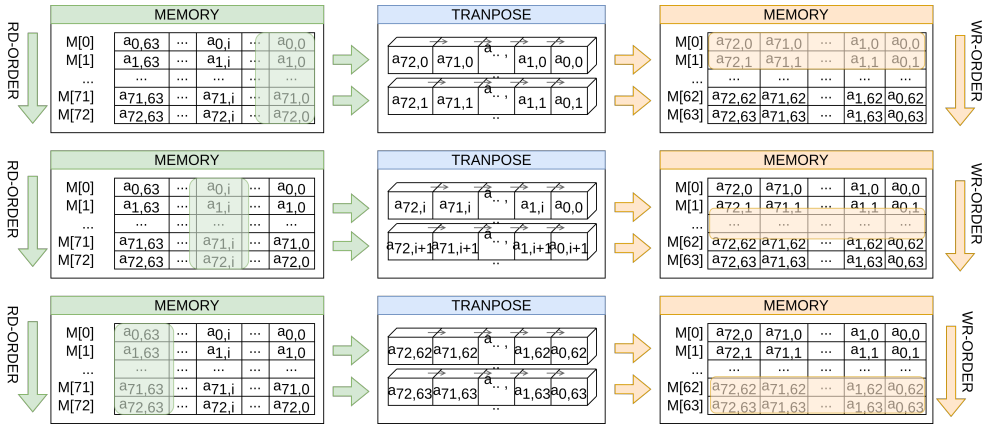

Figure 3: Example of the transpose operation with parameter $\tau = 2$

index i_b and b on index i_a. This indicates that a transpose operation on packed data is relatively simple.

A transpose operation on an unpacked data format is more complex since the data of a matrix is stored differently. Compared to the packed format that stores each element in a separate BRAM slot the unpacked format stores all elements of a row in one slot. This allows us to load and store a whole row of an unpacked matrix in one cycle. However, a transposing operation on unpacked data is much more complex, since we need to split a row into its elements and store these elements at different addresses of the BRAM. This spreading of data to different memory slots leads to a longer latency during the store operation. The logic for the store operation needs to compensate that each element of the matrix is a small chunk of 4 bit data. This 4 bit chunk needs to be written into a specific part of a memory slot of our memory bank, while the remaining data of the memory slot needs to be preserved. In the case of security level 1, each memory element has a size of 292 bit which means that 4 bits at a certain location need to be updated as the remaining 288 bits stay the same.

We developed a scalable method to transpose an unpacked matrix in a pipelined fashion. This method uses three modules to load, transpose, and store data of a given unpacked matrix $\mathbf{A}$. The parametric transpose module instantiates τ many parallel shift registers which allows tuning the throughput of the transpose unit depending on τ. The following will explain the transpose operation on a matrix $\mathbf{A}$ with the dimensions 73×64 as used in security level 1. The number of parallel shift registers in this example is $\tau = 2$. Figure 3 shows the data flow during the transpose operation. The matrix $\mathbf{A}$ is stored in a row-wise manner in MEM_1 (green) and the goal is to get the transpose $\mathbf{A}^\top$ into MEM_2 (orange). The load logic iterates through MEM_1 to load each row of matrix $\mathbf{A}$. It then selects $\tau = 2$ elements of the loaded row depending on the currently targeted row of $\mathbf{A}^\top$. This means that in the first iteration, $a_{0,0}$ and $a_{0,1}$ are selected from the first row of $\mathbf{A}$ and forwarded to the shift register. This is repeated for all rows in $\mathbf{A}$ which fully fills the shift registers. Hence, after the first iteration over all rows of $\mathbf{A}$, *shift register 0* will store $(a_{0,0}, a_{1,0}, a_{2,0}, ..., a_{72,0})$ while *shift register 1* will store $(a_{0,1}, a_{1,1}, a_{2,1}, ..., a_{72,1})$, as shown on top of Fig. 3. This behavior mimics a transpose of the first $\tau = 2$ columns of $\mathbf{A}$ and gives us the first $\tau = 2$ rows of $\mathbf{A}^\top$ which are stored to MEM_2. This procedure is repeated until all columns of $\mathbf{A}$ are handled which yields the transposed matrix $\mathbf{A}^\top$ in MEM_2.

Note, that the number of parallel running shift registers τ can be changed freely by adapting a parameter within our design. This number directly influences the latency of the operation as well as the resource utilization. This means that a lower τ will result in a high latency and low flip-flop utilization while a higher τ will decrease the latency and increase the total number of required flip-flops. This allows us to flexibly adapt the

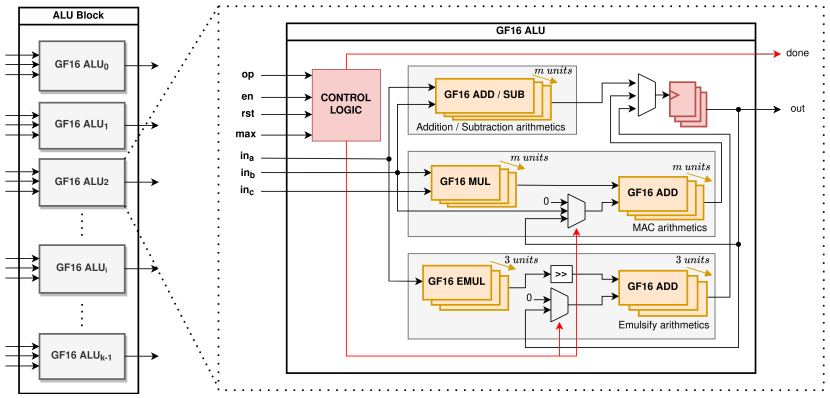

Figure 4: Overview of arithmetical block that consists of k many arithmetical units

transpose unit depending on the available resources in hardware.

4.3 Arithmetical Units

The MAYO scheme operates on the finite field GF(2^4). The field GF(2^4) defines the addition and multiplication operations, as described in Sec. 2.1. Our design needs to support these operations in hardware and uses them for more advanced operations like field accumulation. In addition to this, we also need to support reduction in GF($(2^4)^m$) as described in Sec. 2.5. We first explain addition, subtraction, and multiplication operations on the finite field of GF(2^4). These can be done via a combination of bitwise AND and XOR operations. Second, we show how all functionalities of the MAYO scheme can be implemented by using the basic building blocks of field addition and multiplication. As an example, the MAYO schemes requires a accumulation operation during vector or matrix multiplications. This accumulation can be done by combining a multiplication and addition with a accumulator register, as shown in Fig. 4.

Arithmetical Block The architecture of our arithmetical block is shown in Fig. 4. It consists of k many instances of our arithmetical units, as shown on the left side of Fig. 4. The internal architecture of such a GF16 ALU is shown on the right side. It contains the control logic and three arithmetical units marked in grey. These three units are responsible for addition, accumulation (MAC), and emulsification. Each of these arithmetical units does not process just one GF16 element but m elements concurrently. This allows us to compute on m elements with the same indices from all m matrices $\mathbf{P}_{i \in m}$ in parallel.

Finite Field Addition/Subtraction. An addition and a subtraction on the finite field GF(2^4) is equivalent, which allows us to perform both operations with the same hardware unit. The addition operation consists only of a bit-wise XOR operation, as shown in Section 2.1.1, which are relatively cheap to perform in hardware. We use a total of m adder within our *addition arithmetic unit*, as mentioned in the first paragraph.

Finite Field Multiplication. A multiplication on the finite field in GF(2^4) is different than an integer multiplication. The multiplication operation consists of several bit-wise AND and XOR operations, as shown in Section 2.1.2. Compared to addition, multiplication requires more bit-wise evaluations and therefore consumes more resources in terms of LUTs. However, it is still relatively cheap to perform in hardware. Similar to in the *addition arithmetic unit* we again use a total of m multipliers.

Finite Field Accumulation. In the case of a vector or matrix multiplication, an accumulation operation needs to be performed when multiplying a row with a column. This accumulation operation requires both multiplication and addition. The multiplication is placed before the addition block, as shown in the center of Fig. 4. The output of the multiplication is fed into the addition block which is used to accumulate the intermediate

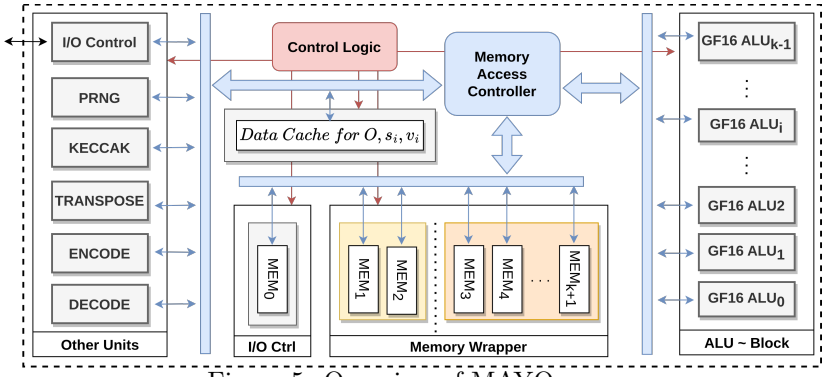

Figure 5: Overview of MAYO core

results. The accumulated value is stored in a register marked in red in Fig. 4 until a reset signal is set. The reset signal requires some extra logic to clear or keep the accumulated data inside the unit. Similar to in the *addition arithmetic unit* we again use a total of m addition and multiplication units for our *MAC arithmetic unit*.

Matrix Multiplications by z in a Finite Field. During the computation and verification of the signature, a multiplication of $\mathbf{E}^l\mathbf{u}$ is required, where $\mathbf{E} \in \mathbb{F}_q^{m \times m}$ is a matrix that represents a multiplication by z in the finite field $\mathbb{F}_q[z]/f(x)$. Our core performs this operation by an iterative reduction of the polynomial by $f(x)$. This reduction operation, however, depends on the security level and its corresponding irreducible polynomial, since each level requires a polynomial of different form due to the parameter m. In the case of security level 1, the irreducible polynomial $f_{64}(z) = z^{64} + x^3z^3 + xz^2 + x^3$ is used during the computation of $\mathbf{E}^l\mathbf{u}$. This reduction by $\pmod{xf(x)}$ is implemented by using just three multiplication and three addition units. First, the m element is used as a scaling factor and multiplied by the polynomial $z = 8z^3 + 2z^2 + 8z$. The result of the three multiplications with the scaling factor is then added to the original data, which needs to be shifted by one element to the right. In contrast to addition, multiplication, and accumulation, the reduction operation uses three addition and multiplication units instead of a full grouping of m units for its computation.

4.4 Overall Design of Processor

This section shows how we combine all the previously discussed components into one core. The overall architecture of the core is shown in Fig. 5. The right side of Fig. 5 shows the arithmetical block unit as discussed in Sec. 4.3. It instantiates k many GF16 ALUs, to allow a parallel computation of $\mathbf{O}$, $\mathbf{v}$, and $\mathbf{s}$. This parallel processing requires a total of k many memory banks, one for each GF16 ALU. These memory banks are part of the Memory Wrapper discussed in Sec. 4.2, as shown in the bottom of Fig. 5. The *Memory Access Controller* is responsible for the data transfer between ALU, Memory Wrapper, and all other units. All of the other supplementary units required for Mayo are shown on the left. All of the units within the core are controlled via the *Control Logic* marked in red. The control logic uses a Finite-State-Machine (FSM) approach to to run the subroutines of Mayo, namely key generation, secret key expansion, signature generation, public key expansion, and signature generation by reusing the same compute units, as shown in Fig.5. The FSM is already designed to be easily replaceable with a instruction-set architecture (ISA) to allow fast adaption to possible changes in the scheme.

4.5 Scheduling of Operation

This section gives an overview of how we schedule the computation of each operation of MAYO within our architecture. The purpose of this section is to show how the MAYO scheme benefits from our optimizations when it comes to key generation, signature generation, and signature verification. Note that we include expansion operations of secret and public key within sign and verification respectively.

Key Generation: An algorithmic representation of original key generation operation is shown in Algorithm 5 of [BCC+23]. The main computation happens in line 10 and 16. First, both $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ are sampled via the public seed $seed_{pk}$. In the next step each sub-matrix of $\mathbf{P}_i^{(3)}$ is computed by line 4 : $\text{UPPER}(-(\mathbf{O}^T\mathbf{P}_i^{(1)}\mathbf{O} + \mathbf{O}^T\mathbf{P}_i^{(2)}))$. This computation consists of six sub-operations in total; three matrix-matrix multiplications, one matrix-matrix addition, one negation, and one upper triangulation operation. We can see that a multiplication with $\mathbf{O}^T$ happens twice, therefore, it can be reduced to just one multiplication by pulling out the multiplication of $\mathbf{O}^T$. Hence the computation of $\mathbf{P}_i^{(3)}$ in line 4 changes to $\text{UPPER}(-\mathbf{O}^T(\mathbf{P}_i^{(1)}\mathbf{O}+\mathbf{P}_i^{(2)}))$. This change from two to one matrix-matrix multiplication saves a total of $(n-o)(o \cdot o - 1)(o \cdot o)$ operations. In addition to this, we apply our parallel matrix column multiplication strategy that operates on the whole $\mathbf{O}$ concurrently, as described in Sec. 3.3.

Expansion of Secret Key: The MAYO scheme splits the computation of the signature into two operations, called MAYO.EXPANDSK() and MAYO.SIGN(). First, the expand function takes both public and private seeds and computes the matrix representation of the linear part $\mathbf{L}_i$, defined as $\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top})\mathbf{O} + \mathbf{P}_i^{(2)}$. Whereas $\mathbf{L}_i$ is needed during evaluation of the linear transformation $P'(\mathbf{v}_k, \cdot)$. Our on-the-fly data sampling via SHAKE128 makes it challenging to compute $\mathbf{L}_i$ since $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(1)^\top}$ needs to be added. This challenge is discussed in detail in Sec. 3.4. We use the described adaption for computing $\mathbf{L}_i$ by utilizing a combination of MAC and BMAC. The MAC and BMAC operations allow us to perform both computations in a pipelined manner without any transpose of $\mathbf{P}_i^{(1)}$. The respective execution flow is as follows: (1) we generate one row of $\mathbf{P}_i^{(1)}$ via on-the-fly data sampling, (2) we perform a MAC operation on this row of $\mathbf{P}_i^{(1)}$, which is followed by (3) a BMAC operation. This procedure (1-3) is repeated for all rows of $\mathbf{P}_i^{(1)}$. Finally, all partial computation results are accumulated to yield $\mathbf{L}_i$.

Signature Computation: The MAYO.SIGN() function is the second operation in signature generation and follows MAYO.EXPANDSK(). The MAYO.SIGN() function follows Algorithm 8 of [BCC+23]. The main effort within the MAYO.SIGN() function is spent on finding a preimage for a given hash $\mathbf{t}$ of the digested message and a given salt. Obtaining a preimage of $\mathbf{t}$ requires us to find a linear system ($\mathbf{Ax} = \mathbf{y}$) that is solvable. The preimage is then used to generate the signature according to Eq. (10). We split this process into five parts, namely (1) deriving $\mathbf{v}$ and $\mathbf{r}$, (2) calculating $\mathbf{y}$, (3) calculating $\mathbf{A}$, (4) check if the resulting system of $\mathbf{Ax} = \mathbf{y}$ is actually solvable, and (5) computing the signature. These four steps are repeated as long as no solvable equation system is found. In the first step, we derive $\mathbf{v}$ and $\mathbf{r}$ by hashing a combination of M_{digest}, salt $salt$, secret key $seed_{sk}$, and a counter ctr via SHAKE256. This step corresponds to line 16 in Alg. 8 in [BCC+23]. The counter ctr keeps track of the number of unsuccessful attempts to find a preimage for $\mathbf{t}$. The resulting $\mathbf{v}$ and $\mathbf{r}$ are stored in the on-chip *data cache* to allow parallel computation, as explained in Sec. 3.3. Second, we calculate $\mathbf{y}$ that is used as the right side of the linear equation system $\mathbf{Ax} = \mathbf{y}$. The computation of $\mathbf{y}$ is done in a nested loop, as in line 24-34 of Alg. 8 in [BCC+23]. We first perform a precomputation of $\mathbf{s}_i = \mathbf{P}_i^{(1)}\mathbf{v}_i$ which result is used in the following nested loop computation. The main purpose of the precomputation is to avoid repeating identical operations within the loop. After the precomputation, we perform the computations within the nested loop by fully

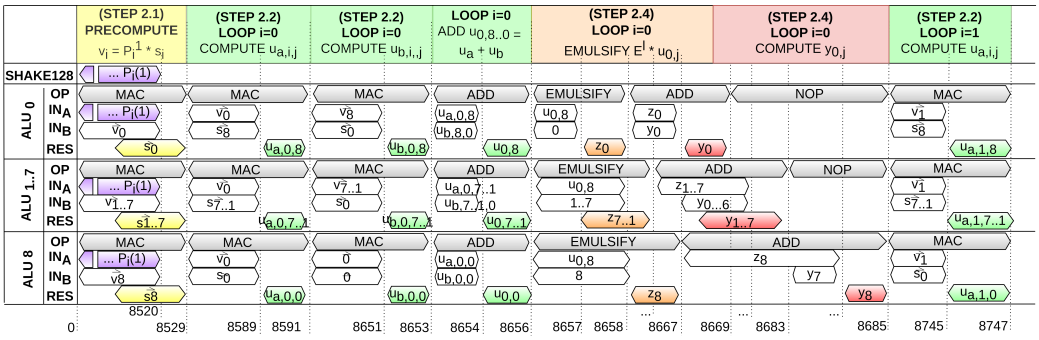

Figure 6: Scheduling overview of computation loop to compute y of MAYO.Sign()

unrolling the inner loop. Therefore, our ALU Block uses up to k many GF16 ALUs. Each GF16 ALU is responsible for computing one of the $i-(k-1)$ loop iterations in the inner loop. Figure 6 shows the unrolled data flow during the computation of the innermost loop. It shows how $\mathbf{u}$ is computed in two phases depending on values of the loop variables i and j, marked in green. The following emulsification of $\mathbf{E}^l\mathbf{u}$ and the accumulation of $\mathbf{y}$ are marked orange and red respectively. The operation (3) of the Mayo.Sign() is the computation of $\mathbf{A}$. This step is very similar as described above for computing $\mathbf{y}$ in step (2). The only difference is that the procedure for computing $\mathbf{y}$ is repeated several times since $\mathbf{A}$ is a matrix. After finishing both computations $\mathbf{A}$ and $\mathbf{y}$, we move on to step (4). This step uses SampleSolution() to check whether the linear system is solvable. If the system is solvable, we start computing the signature. Otherwise, we need to repeat all steps (1-4) with an incremented counter Ctr. Finally, we move to step (5) to compute the signature by using the solution $\mathbf{x}$ that was calculated in step (4).

Signature Verification: The Mayo.Verify() function is used to verify whether a given signature in combination with a message is valid. The steps in Mayo.Verify() is relatively similar to the computation of $\mathbf{y}$ in Mayo.Sign(), as Alg. 9 in [BCC+23] shows. Hence, we use the same loop unrolling technique as in the signature generation to accelerate the verification process. The main difference between signing and verifying is in the pre-computation step. In contrast to signing, which computes $\mathbf{s}_i = \mathbf{P}_i^{(1)}\mathbf{v}_i$, the verification needs to compute $\mathbf{w}_i = \mathbf{P}_i\mathbf{s}_i$. The $\mathbf{P}_i$ matrix is a collection of $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$ as shown in Eq. (19). We split the computation of $\mathbf{P}_i\mathbf{s}_i$ into three blocks for each of the sub-matrices as described in Sec. 3.5.

5 Results

In this section, we present the area and performance results of our hardware and provide a comparison with related works in the literature. We coded the architectural units of our MAYO cores using SystemVerilog. The area and performance results are obtained using Xilinx Vivado 2022.2 for Kintex-7 KC705 and Alveo U280 with default synthesis and place & route settings. Furthermore, we verified the functionality of each operation (key generation, signing, and verification) for all security levels on actual FPGAs. Specifically, our MAYO cores for security levels 1 and 3 are verified on the Xilinx Kintex-7 KC705 board while the core for security level 5 is verified on the Xilinx Alveo U280 board. We also synthesized our MAYO_1, MAYO_3 and MAYO_5 cores using a 28nm library with the Cadence tool. Area and performance results of the proposed architectures are shown in Table 4. The proposed hardware architecture performs all computations solely on hardware without requiring any communication with the software during computations.

Area results: Our architectures on FPGA have a rather high LUT utilization and this is mainly contributed by the complex data bus, which reads data from multiple memories

Table 4: Area and Performance of the Proposed Architectures

Design	Platform	Latency (in cc/ms)			Area (mm^2 for ASIC)
		KeyGen	Sign	Verify	LUT/FF/DSP/BR
$MAYO_1$	KC705 @ 100MHz	12,182/0.12	49,926/0.49	12,722/0.12	91,266/42,113/2/45
	AU280 @ 225MHz	12,182/0.05	49,926/0.22	12,722/0.05	89,014/42,066/2/45
	A-28nm @ 1GHz	12,182/0.01	49,926/0.05	12,722/0.01	$1.02mm^2$
$MAYO_3$	KC705 @ 100MHz	38,325/0.38	137,358/1.37	39,740/0.39	150,839/69,968/2/95
	AU280 @ 175MHz	38,325/0.21	137,358/0.78	39,740/0.22	148,307/69,900/2/95
	A-28nm @ 1GHz	38,325/0.04	137,358/0.14	39,740/0.04	$1.69mm^2$
$MAYO_5$	AU280 @ 125MHz	90,743/0.72	241,310/1.93	92,339/0.73	222,979/98,844/2/193.5
	A-28nm @ 1GHz	90,743/0.09	241,310/0.24	92,339/0.09	$3.04mm^2$

A-28nm: ASIC with 28nm library. KC705: Xilinx Kintex-7. AU280: Xilinx Alveo U280.

and feeds k parallel ALU blocks. Although this enables our design to perform most computations in parallel and avoid loading the same data from the memory multiple times, it increases implementation complexity. Inputs and output of all ALU blocks are buffered to improve the frequency of the design. This also increases the FF utilization and half of the overall FF utilization is used for the ALU blocks. Each ALU within our ALU-Block has three inputs and one output, each of a size of $k \cdot o \cdot log_2(16)$, either 292 bit, 444 bit, or 580 bit depending on the chosen security level. In our architecture, BRAM utilization is relatively low thanks to our on-the-fly data sampling and optimized memory structure with careful computation scheduling. A combination of these techniques allows us to keep the number of required BRAM low compared to the related works in the literature, as we will show later in this section. This is because our architecture only needs to store interim results and not the large $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices that require up to a few thousand KB of memory. For ASIC implementations, on-chip SRAMs use the most area. For the $MAYO_1$, $MAYO_3$ and $MAYO_5$ architectures, $0.72mm^2$ of $1.02mm^2$ total area, $1.15mm^2$ of $1.69mm^2$ total area and $2.34mm^2$ of $3.04mm^2$ total area are consumed by on-chip SRAMs, respectively.

Performance results: All performance numbers for FPGA implementations are collected using actual measurements on real FPGAs. For the implementations on Xilinx Alveo U280, we can finish key generation, signature generation and verification in $0.05/0.21/0.72ms$, $0.22/0.78/1.93ms$ and $0.05/0.22/0.73ms$ for $MAYO_1/MAYO_3/MAYO_5$, respectively. As shown in Table 4, cycle counts of key generation, signature generation and verification for $MAYO_3$ show 3.15×, 2.75× and 3.12× increase, respectively, compared to $MAYO_1$. This increase in latency is mostly due to the increased amount of pseudo-random data that needs to be sampled, as explained in Sec. 3. Note that $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ sizes change from 1711 to 4005 and 464 to 890, respectively. This means that switching the security level from 1 to 3 leads to an increase of 2.34× and 1.91× for the required pseudo-random data. We observe a similar result for the cycle counts of $MAYO_3$ and $MAYO_5$ as well. The ASIC implementations can finish key generation, signature generation and verification in $0.01/0.04/0.09ms$, $0.05/0.14/0.24ms$ and $0.01/0.04/0.09ms$ for $MAYO_1/MAYO_3/MAYO_5$, respectively.

5.1 Comparisons with Related Works

There are only a few works in literature implementing the MAYO digital signature scheme. Table 5 provides area and performance comparisons of FPGA [BCH+23, SMA+23], microcontroller [BCH+23, GMSS23, BCC+23] and high-end CPU [BCC+23] implementations of MAYO with our implementations on FPGA and ASIC. Further, we also included FPGA and microcontroller implementations of UOV scheme [BCH+23] which uses similar construction and computations as the MAYO such as using emulsifier maps to reduce the size of the signature. These similarities make it possible to present a comparison between [BCH+23] and our work. To the best of our knowledge, there are no ASIC imple-

Table 5: Comparison with Related Works

	Works	Platform	Latency (cc/ms)			Area (mm^2 for ASIC)
			KeyGen	Sign	Verify	LUT/FF/DSP/BR
$MAYO_1$	[BCH+23][a]	Artix-7 @ 90.8MHz[c]	11,072K/121.94	843K/9.29	284K/3.13	28,497/24,444/2/66
		Artix-7 @ 90.3MHz[d]	11,008K/121.91	779K/8.63	115K/1.27	23,208/26,974/2/66
		AC-A72 @ 1.8GHz	28,324K/15.73	13,333K/7.40	2,266K/1.25	-
	[BCC+23][b]	AC-M4 @ 1GHz	5,245K/5.24	9,183K/9.18	4,886K/4.88	-
		IXG 6338 @ 2GHz	110K/0.05	460K/0.23	175K/0.08	-
	[GMSS23]	AC-M7 @ 480MHz	-	42,927K/89.43	5,703K/11.88	-
	[SMA+23]	Z-7020 @ 100MHz	996K/9.96	3,491K/34.92	-	23,356/24,645/11/136
	Our	KC705 @ 100MHz	12,182/0.12	49,926/0.49	12,722/0.12	91,266/42,113/2/45
		AU280 @ 225MHz	12,182/0.05	49,926/0.22	12,722/0.05	89,014/42,066/2/45
		A-28nm @ 1GHz	12,182/0.01	49,926/0.05	12,722/0.01	$1.02mm^2$
$MAYO_3$	[BCH+23][a]	Artix-7 @ 96MHz[c]	16,727K/174.24	1,465K/15.26	823K/8.57	38,352/19,446/2/184.5
		Artix-7 @ 94.1MHz[d]	16,462K/174.94	1,199K/12.75	195K/2.07	43,166/31,928/2/184.5
		AC-A72 @ 1.8GHz	56,815K/31.56	34,533K/19.18	8,318K/4.62	-
	[BCC+23][b]	IXG 6338 @ 2GHz	508K/0.25	1,663K/0.83	610K/0.30	-
	Our	KC705 @ 100MHz	38,325/0.38	137,358/1.37	39,740/0.39	150,839/69,968/2/95
		AU280 @ 175MHz	38,325/0.21	137,358/0.78	39,740/0.22	148,307/69,900/2/95
		A-28nm @ 1GHz	38,325/0.04	137,358/0.14	39,740/0.04	$1.69mm^2$
$MAYO_5$	[BCH+23][a]	Artix-7 @ 82.5MHz[c]	39,066K/437.53	3,308K/40.09	1,921K/23.29	77,352/38,217/2/356
		Artix-7 @ 92.6MHz[d]	38,404K/414.73	2,645K/28.56	364K/3.93	83,444/40,597/2/359
		AC-A72 @ 1.8GHz	291,438K/161.91	86,727K/48.18	18,602K/10.33	-
	[BCC+23][b]	IXG 6338 @ 2GHz	1,210K/0.60	4,149K/2.07	1,186K/0.59	-
	Our	AU280 @ 125MHz	90,743/0.72	241,310/1.93	92,339/0.73	222,979/98,844/2/193.5
		A-28nm @ 1GHz	90,743/0.09	241,310/0.24	92,339/0.09	$3.04mm^2$

Z-7020: Xilinx Zynq-7020. IXG 6338: Intel Xeon Gold 6338. AC-M4/M7/A72: ARM Cortex-M4/M7/A72.
[a]: Targets UOV scheme. [b]: Uses AVX2 with AES-NI. [c]: Uses AES without pipelining. [d]: Uses pipelined AES.

mentation results for the MAYO schemes and we present the first ASIC implementation results for it.

Comparisons with FPGA implementations: To the best of our knowledge, there is only one FPGA implementation of the MAYO in the literature, HaMAYO [SMA+23], which is a reconfigurable hardware implementation of the MAYO scheme that uses the outdated parameters of the MAYO scheme. The implementation is only capable of performing key generation and signature generation operations for security level 1. Compared to HaMAYO, we support all operations and security levels of the MAYO scheme. Table 5 shows that our implementation on Kintex-7 outperforms HaMAYO by 83× and 71× for key generation and signature generation, respectively. We show better performance at the expense of using 3.9× more LUTs and 1.7× more FFs while our BRAM utilization is 3× less compared to HaMAYO. Our implementations on Alveo U280 FPGA and ASIC show 199×/158× and 996×/698× speedup compared to HaMAYO's key generation/signature generation, respectively. Note that the MAYO team updated the field operations of the scheme from GF256 (8-bit) [Beu21] to GF16 (4-bit) [BCC+23] to enable faster computation as well as smaller keys. We already adopted this change within our design while HaMAYO operates on the old ones. The work in [BCH+23] presents an Artix-7 FPGA implementation of the UOV scheme and provides area/performance results for all operations and security levels. [BCH+23] presents implementation results for different configurations such as arithmetic with GF256 and GF16 as well as implementations using either a full-round or reduced-round based AES128 for pseudo-random data sampling. We use two of their implementations with GF16 arithmetic and full-round AES (pipelined and non-pipelined) since they best resemble our parameters and design methodology. Our implementations on Kintex-7 and Alveo U280 FPGAs outperform key generation/signature generation/verification generation of both implementations by up to 1016×/19×/26×, 460×/11×/22×, and 607×/20×/32× for security 1, 3, and 5, respectively. This shows that we outperform them by up to three orders of magnitude when it comes to key generation operation due to our massive parallelization of the matrix-matrix multiplications. However, the high parallelism comes at a price of resources utilization leading to a 3.2×/4.4×/3.37× and 1.72×/1.64×/2.59× higher LUTs and FFs usages, respectively, for security levels 1/3/5. For signature generation and

verification operations, our designs still outperform theirs by an order of magnitude. The decrease in performance is because they utilize more memory to buffer temporary data in the key generation that can be reused during signing and verification. Yet, their design designs use 1.57×/1.94×/1.84× more BRAMs for security levels 1/3/5. These results clearly show that despite the increase in the utilization of LUTs and FFs, our design still outperforms HaMAYO [SMA+23] and [BCH+23] by one to three orders of magnitude. In addition to performance improvement, our optimized memory management shows how to halve the consumption of BRAMs for multivariate schemes on hardware.

Comparisons with ARM-based implementations: There are two ARM-based microcontroller implementations of MAYO scheme [GMSS23, BCC+23] and both works target only security level 1 of the MAYO. Compared to [BCC+23], our implementation on Kintex-7 FPGA outperforms its key generation, signing, and verification performance by 43×, 18×, and 38×, respectively. The implementation in [GMSS23] presents results only for signature generation and verification, and our implementation on Kintex-7 outperforms their performance by 182.5× and 99×, respectively. The ARM Cortex-A72 implementation for the UOV scheme in [BCH+23] implements all operations for security levels 1, 3 and 5. Compared to [BCH+23], our implementations on Kintex-7 FPGA for $MAYO_1$ and $MAYO_3$ outperform their ARM implementations by 131×/15×/10× and 83×/14×/12× for key generation/signing/verification, respectively. On Alveo U280, our implementation shows 224×/25×/17× better performance.

Comparisons with high-end CPU implementations: The MAYO team provides reference, optimized and Intel AVX2 optimized C implementations [PQM]. For comparison, we report their best AVX2 optimized implementation with AES-NI on Intel Xeon Gold 6338 CPU (Ice Lake) with 2GHz [BCC+23], as shown in Table 5. Our implementations on Alveo U280 show similar or slightly better performance for some operations. Specifically, our $MAYO_1$ implementation shows 1.6× better performance for signature verification. Similarly, our $MAYO_3$ implementation shows 1.2×, 1.06× and 1.3× performance improvement for key generation, signature generation and signature verification, respectively. Our implementations on Kintex-7 FPGA are $1.3\times - 2.4\times$ slower due to low operating frequency. Our ASIC implementations of $MAYO_1$, $MAYO_3$ and $MAYO_5$ outperform the optimized CPU implementation [BCC+23] by 5×/4.6×/8×, 6.2×/5.9×/7.5× and 6.6×/8.6×/6.5×, respectively, for key generation/signature generation/signature verification. It is not easy to provide a fair comparison for the area cost of ASIC and FPGA works. Our ASIC implementations for $MAYO_1$, $MAYO_3$ and $MAYO_5$ consume ≈8.1M, ≈16.2M and ≈38.9 transistors, respectively in 28nm technology while high-end Intel CPUs can use up to billions of transistors. Thus, our ASIC implementations can provide significant speedup for less area cost.

6 Conclusion

In this paper, we propose and implement a hardware architecture for the MAYO post-quantum signature scheme for all NIST security levels. The proposed architecture can perform key generation/signature generation/signature verification operations in $0.12ms/0.49ms/0.12ms$, $0.38ms/1.37ms/1.39ms$, and $0.72ms/1.93ms/0.73ms$ for security levels 1, 3, and 5, respectively. We also propose several optimization techniques to improve resource utilization and performance of MAYO implementations on hardware platforms like FPGA and ASIC. Our hardware shows a general speed-up of one to three orders of magnitude compared to similar work for FPGA. The most significant improvement can be seen when it comes to key generation, where we achieve a speedup of 1016×, 460×, and 607×. In addition to this, our design shows an increase in performance of 19×, 11×, and 20× for signature generation as well as 26×, 22×, and 32× for verification.

Acknowledgement

This work was supported in part by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark. This work has benefitted from the third and fourth author's participation in Dagstuhl Seminar 23152 "Secure and Efficient Post-Quantum Cryptography in Hardware and Software". We thank Florian Krieger for his careful review and feedback that helped us to improve the paper.

References

[ADF+12] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Intel® advanced encryption standard (intel® aes) instructions set, 2012.

[BCC+23] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias Kannwischer. Mayo. MAYO Website, 2023. https://pqmayo.org/assets/specs/mayo.pdf.

[BCH+23] Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. Oil and vinegar: Modern parameters and implementations. Cryptology ePrint Archive, Paper 2023/059, 2023. https://eprint.iacr.org/2023/059.

[BDK+22] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium. Selected Algorithms 2022, 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023.

[Beu21] Ward Beullens. Mayo: Practical post-quantum signatures from oil-and-vinegar maps. Cryptology ePrint Archive, Paper 2021/1144, 2021. https://eprint.iacr.org/2021/1144.

[Beu22] Ward Beullens. Mayo: Practical post-quantum signatures from oil-and-vinegar maps. In *Selected Areas in Cryptography*, pages 355–376. Springer, 2022.

[GMSS23] Arianna Gringiani, Alessio Meneghetti, Edoardo Signorini, and Ruggero Susella. Mayo: Optimized implementation with revised parameters for armv7-m. Cryptology ePrint Archive, Paper 2023/540, 2023. https://eprint.iacr.org/2023/540.

[HBD+22] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Selected Algorithms 2022, 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023.

[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*, pages 206–222. Springer, 1999.

[NIS] 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. Accessed August 3rd 2023.

[PFH+22] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. FALCON. Selected Algorithms 2022, 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023.

[PQM] PQMayo. MAYO-C. https://github.com/PQCMayo/MAYO-C. Accessed August 3rd 2023.

[PTBW11] Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. Small public keys and fast verification for multivariate quadratic public key systems. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 475–490. Springer, 2011.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Leo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehle. CRYSTALS-KYBER. Selected Algorithms 2022, 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022. Accessed August 3rd 2023.

[Sho94] P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, Washington, DC, USA, 1994. IEEE Computer Society.

[SMA+23] Oussama Sayari, Soundes Marzougui, Thomas Aulbach, Juliane Krämer, and Jean-Pierre Seifert. Hamayo: A reconfigurable hardware implementation of the post-quantum signature scheme mayo. Cryptology ePrint Archive, Paper 2023/1135, 2023. https://eprint.iacr.org/2023/1135.

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