A Hardware Implementation of MAYO Signature Scheme

Florian Hirner, Michael Streibl, Ahmet Can Mert and Sujoy Sinha Roy

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

Abstract.
We present a work-in-progress hardware implementation for the MAYO post-quantum digital signature scheme, which is submitted to the American National Institute of Standards and Technology's call for diversification of quantum-resistant public key cryptographic standards. The scheme is based on the Unbalanced Oil and Vinegar signature scheme, which operates on the fact that solving systems of multivariate polynomial equations is NP-complete. MAYO utilizes a unique whipping technique in combination with emulsifier maps to offer a significant reduction in key size compared to other Unbalanced Oil and Vinegar signature schemes. In this paper, we demonstrate how to design a hardware architecture for the MAYO post-quantum signature scheme. We also provide a comprehensive analysis and propose multiple optimization techniques to reduce resource utilization and accelerate computation on hardware platforms.

Keywords: Hardware, MAYO, PQC, FPGA.

1 Introduction

The American National Institute of Standards and Technology (NIST) recently issued a new call to diversify quantum-resistant digital signature schemes selected for standardization [NIS]. In 2022, NIST selected one key-exchange mechanism (KEM), CRYSTALS-KYBER (Kyber) [SAB+22], and three digital signature algorithms, Falcon [PFH+22], SPHINCS+ [HBD+22], CRYSTALS-DILITHIUM (Dilithium) [BDK+22], for standardization. Most of these utilize different types of lattice problems to ensure post-quantum security, therefore, a diversification of the underlying computation assumptions is desired by NIST. 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 an unusually small oil space, furthermore, it requires the usage of 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.

Related Works. There are only a few available MAYO implementations in the literature. The MAYO team provides a reference software implementation as well as 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 MAYO scheme is proposed [SMA+23] that implements a part of the scheme.

Our Contributions. We present a comprehensive analysis of the MAYO scheme and propose multiple optimization techniques to port it efficiently to hardware. A comparison with existing software and embedded implementations shows the potential benefits that can be achieved by using a pure hardware solution. We designed and implemented hardware for the MAYO scheme targeting NIST security level 1. The proposed hardware can perform key generation, signature generation, and signature verification operations in 0.60 ms, 1.45 ms, and 0.77 ms, respectively. To the best of our knowledge, this is the first FPGA implementation of MAYO supporting all three operations, key generation, signature generation, and verification, within one architecture for NIST security level 1. Moreover, we present the first in-detail analysis of all required computations, which shows that data generation and matrix multiplication operations are the most suitable for optimizations, and propose several optimizations for the MAYO implementations targeting hardware platforms. We tested and verified our optimization techniques with the reference implementation to guarantee the functionality of the scheme.

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 3, we describe the MAYO signature scheme and give a detailed explanation of its specifications, like their whipping technique, emulsifier maps, and more. Section 4 gives an in-depth explanation of our hardware implementation and in Section 5, we present the results. Moreover, in Section 6, we present several optimizations to further improve hardware implementations and Section 7 concludes the paper.

2 Background

This section covers the background necessary to understand arithmetic used in MAYO scheme.

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

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

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

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

$$\mathbf{b} = (p_1(\mathbf{a}), \ldots, p_m(\mathbf{a})). \tag{4}$$

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

$$P(\mathbf{o}) = 0 \quad \text{for all } \mathbf{o} \in O. \tag{5}$$

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

$$P'(\mathbf{x}, \mathbf{y}) = P(\mathbf{x}+\mathbf{y}) - P(\mathbf{x}) - P(\mathbf{y}). \tag{6}$$

Given a target $\mathbf{t} \in \mathbb{F}_q^m$, one selects a vector $\mathbf{v} \in \mathbb{F}_q^n$ and solves $P(\mathbf{v}+\mathbf{o}) = t$ for $\mathbf{o} \in O$. From Eq. (6), it follows that

$$P(\mathbf{v}+\mathbf{o}) = P(\mathbf{v}) + P(\mathbf{o}) + P'(\mathbf{v}, \mathbf{o}) = \mathbf{t}. \tag{7}$$

Since $P(\mathbf{v})$ is fixed and due to Eq. (5), 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.

3 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. To tackle the problem of large key sizes Beullens *et al.* modifies the original Oil and Vinegar scheme by introducing 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.

3.1 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. The memory requirement for storing P is $mn^2 \log q$. Recall 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. (7) is unsolvable with high probability. Hence, by reducing the oil space dimension the public key size decreases accordingly and the proposed whipping achieves exactly that.

3.2 Whipping Technique

As mentioned in Section 3, 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

$$P^*(\mathbf{x}_1, \ldots, \mathbf{x}_k) = \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{x}_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{x}_i, \mathbf{x}_j). \tag{8}$$

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. Due to the property that P^* vanishes on O^k, the signature can be sampled similar to Eq. (7) by solving the linear system

$$P^*(\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k) = \mathbf{t}, \tag{9}$$

which has m equations in ko variables.

3.3 Scheme Description

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

3.3.1 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. (5), the multivariate quadratic map P must vanish on O. Thus, a polynomial $p_i(\mathbf{x})$ of P has to fulfill

$$(\mathbf{O}^\top \mathbf{I}_o) \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} (\mathbf{O}^\top \mathbf{I}_o)^\top = \mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{O}^\top \mathbf{P}_i^{(2)} + \mathbf{P}_i^{(3)} = 0. \tag{10}$$

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 $\text{Upper}(\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{O}^\top \mathbf{P}_i^{(2)})$, where Upper is defined as $\text{Upper}(\mathbf{M}_{ii}) = \mathbf{M}_{ii}$ and $\text{Upper}(\mathbf{M}_{ij}) = \mathbf{M}_{ij} + \mathbf{M}_{ji}$ for $i < j$.

Generating large parts of the matrices pseudo-randomly enables the significant key size reduction, since only $\mathbf{P}_i^{(3)}$, the private and the public seed needs to be stored. Additionally, the whipping transformation described in Section 3.2 reduced the size of $\mathbf{P}_i^{(3)}$ from $m \times m$ to $o \times o$.

3.3.2 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. (9). 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. (7), the oil space trapdoor information enables the partition of Eq. (9) into a constant and a linear part, which leads to

$$\begin{aligned}
&P^*(\mathbf{v}_1 + \mathbf{o}_1, \ldots, \mathbf{v}_k + \mathbf{o}_k)\\
&= \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{v}_i + \mathbf{o}_i) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{v}_i + \mathbf{o}_i, \mathbf{v}_j + \mathbf{o}_j)\\
&= \sum_{i=1}^{k} \mathbf{E}_{ii} (P(\mathbf{v}_i) + P'(\mathbf{v}_i, \mathbf{o}_i)) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} \mathbf{E}_{ij} (P'(\mathbf{v}_i, \mathbf{v}_j) + P'(\mathbf{v}_i, \mathbf{o}_j) + P'(\mathbf{v}_j, \mathbf{o}_i))\\
&= \sum_{i=1}^{k} \mathbf{E}_{ii} P(\mathbf{v}_i) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} \mathbf{E}_{ij} P'(\mathbf{v}_i, \mathbf{v}_j) \quad \text{(constant)}\\
&+ \sum_{i=1}^{k} \mathbf{E}_{ii} P'(\mathbf{v}_i, \mathbf{o}_i)) + \sum_{i=1}^{k}\sum_{j=i+1}^{k} \mathbf{E}_{ij} (P'(\mathbf{v}_i, \mathbf{o}_j) + P'(\mathbf{v}_j, \mathbf{o}_i)) \quad \text{(linear)}\\
&= \mathbf{t}.
\end{aligned} \tag{11}$$

The constant part can be calculated using

$$\begin{aligned}
p_i(\mathbf{v}_k) &= \tilde{\mathbf{v}_k}^\top \mathbf{P}_i^{(1)} \tilde{\mathbf{v}_k},\\
p'_i(\mathbf{v}_k, \mathbf{v}_l) &= \tilde{\mathbf{v}_k}^\top \mathbf{P}_i^{(1)} \tilde{\mathbf{v}_l} + \tilde{\mathbf{v}_l}^\top \mathbf{P}_i^{(1)} \tilde{\mathbf{v}_k}.
\end{aligned} \tag{12}$$

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

$$\mathbf{L}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)^\top})\mathbf{O} + \mathbf{P}_i^{(2)}. \tag{13}$$

Then, each component $p'_i(\mathbf{v}_k, \cdot)$ of P' is defined as $\tilde{\mathbf{v}_k}^\top \mathbf{L}_i$. Applying Eq. (12) and Eq. (13) to Eq. (11) 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.

3.3.3 Signature Verification

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

3.4 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 $\mathrm{GF}((2^4)^m)$.

Similar to the finite field described in Section 2.1, elements of $\mathrm{GF}((2^4)^m)$ can be represented as a polynomial, however, this time of degree $m-1$ and with coefficients in $\mathrm{GF}(2^4)$. Therefore, $a \in \mathrm{GF}((2^4)^m)$ is of the form

$$a = a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_1 z + a_0. \tag{14}$$

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 $\mathrm{GF}((2^4)^m)$ 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. (14), and perform the following computations:

$$\begin{aligned} b &= \mathbf{E}a, \quad \text{with} \\ b_0 &= 8a_{m-1} \\ b_2 &= 2a_{m-1} + a_1 \\ b_3 &= 8a_{m-1} + a_2 \\ b_i &= a_{i-1} \quad \text{for } i \notin \{0, 2, 3\}. \end{aligned} \tag{15}$$

It is important to note that the additions and multiplications in Eq. (15) are $\mathrm{GF}(2^4)$ 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 $\mathrm{GF}((2^4)^m)$ element in one cycle in hardware. To evaluate $\mathbf{E}^l\mathbf{u}$, we perform this computation l times.

4 The Proposed Hardware

The MAYO scheme has different operations (key generation, signing, and verification) and they share similar arithmetic computations. Since we target a hardware architecture supporting all operations, we choose an instruction-set based architecture that allows a flexible mode of operation for different parameter sets. The overall architecture consists of an instruction ROM, an instruction decoder, a memory grid, a data bus, a Keccak core, an AES core, a PRNG, and computation units for finite field arithmetic. The overall design of the MAYO core is shown in Fig. 1. Our design employs specific ALU blocks that are capable of performing all the aforementioned operations by dividing them into vector-vector addition, multiplication and accumulation operations. All of them can be computed via our ALU blocks by sending an instruction that keeps information such as memory location and operation type. The core can be programmed using dedicated instruction ROM, which stores instruction that are decoded to control the data bus and modules. This makes our core flexible in terms of changes in the schemes as well as in different sizes in regards to different security levels.

4.1 Instruction Set

Each instruction consists of multiple arguments, an opcode and three arguments (OPCODE, A, B, C). The opcode represents the mode of operation that the hardware should perform, like reading input, sending output, aes128_ctr, keccak256, addition, multiplication, and many more. The three arguments after the opcode (A, B, C) are used as pointers to the respective data in the memory wrapper, e.g., an addition of data stored at memory locations A and B, and storing the result to memory location C. Each instruction keeps the information $< opcode >< A >< B >< C >$ (or $< opcode >< source_0 >< source_1 >< destination >$), which will be decoded by an instruction decoder.

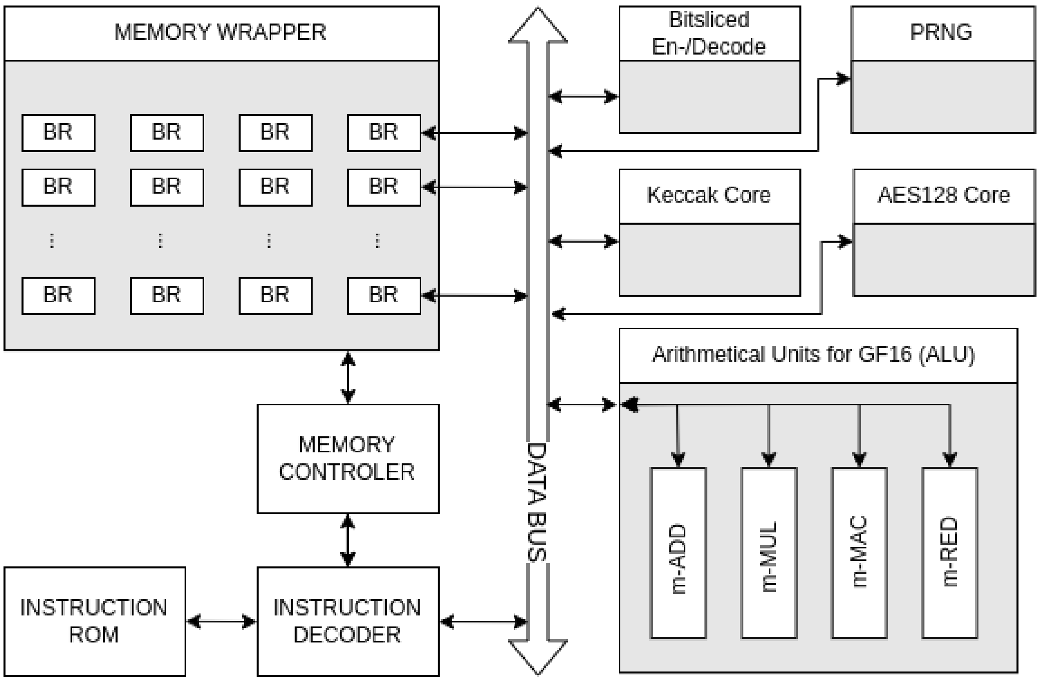

Figure 1: High-level architecture of the proposed hardware for MAYO

4.2 Layout of Memory Grid

One key factor of an efficient hardware implementation is a well-designed memory layout. Since MAYO mainly consists of matrix and vector operations, we have to be able to load the relevant elements fast. The operations of MAYO include the evaluation of a multivariate quadratic map, which consists of multiple polynomial evaluations as described in Section 2.2. Therefore, our memory layout focuses on accelerating these evaluations. Advantageously, the input to all m elements of polynomials is the same in each occurrence, which allows us to parallelize these operations. Thus, we introduce two major formats for storing vectors and matrices.

1. **Unpacked**: The elements of a vector or matrix are stored in row-major order in BRAMs. One entry in the BRAM corresponds to one vector or matrix element.
2. **Packed**: The elements of a vector or matrix are again stored in row-major order in BRAMs. However, one BRAM entry stores not a single vector or matrix element anymore, but m elements. More specifically, we take all m 4-bit long field elements and concatenate it to one larger value of length $4 \times m$-bit. This value is then stored into one entry of the BRAM.

The m multivariate quadratic polynomials operate on the same input values. 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 which are related to the multivariate quadratic map are stored in the packed format, while the remaining ones are stored as unpacked. Due to the fact that m is at least of size 64, we need a minimum of 4 BRAMs in width to store such a packed entry, considering a Xilinx BRAM36K unit, which can store 512 elements each 72-bit.

The memory layout is designed as a grid-like structure to support all security levels (1, 2, 3, 5) of MAYO. In the case of security level 1, the width is defined by the number of samples and their corresponding size in the GF(2^4) field, which is 64 and 4 respectively, leading to a total of 256 bits. The height, however, depends on the total size of the data required during the evaluation of the functions (key generation, sign, and verity). These functions use quadratic polynomials, denoted as $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, $\mathbf{P}_i^{(3)}$, described in Section 3 that depend on the parameters n and o, as well as other data, like O, L_i and many more. We calculated the maximum size of memory that is needed to hold the required data during computations which led to a memory height of 16 BRAMS. All in all, the total requirement for BRAMS in our memory layout is $16 \cdot 4 = 64$ BRAMS. The overall layout of the memory grid is represented in Fig. 2, whereas the last row has an extra BRAM due to the transpose that is required for solving the system of equations during the computation of the signature. Another important challenge is transposing matrices during or after computations if they are in a packed or unpacked format.

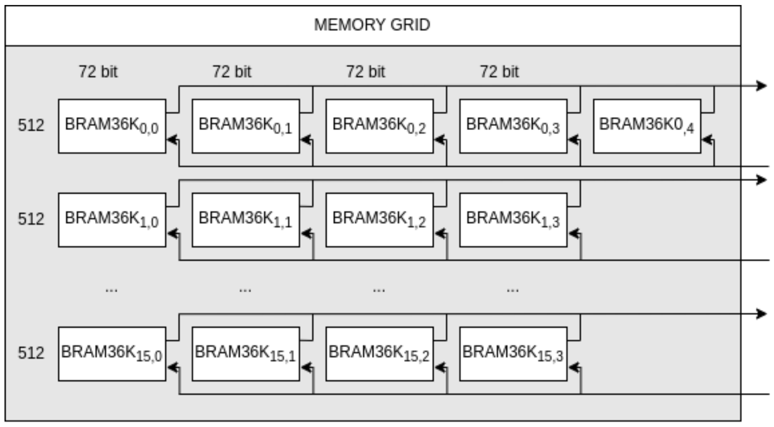

Figure 2: Memory grid layout of MAYO core

4.2.1 Packed/Unpacked Matrix Transpose

There are two different types of transpose that our core needs 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 index i_b and b on index i_a. This indicates that a transpose operation on packed data is relatively simple. However, a transpose operation on an unpacked data format is much 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 means that we can load and store a whole row of a matrix in one cycle, however, this benefit comes with a downside. The transposing operation is much more complex in this situation since we need to split a row into its elements and store them at different addresses. This spreading of data to different memory slots leads to a longer latency during the store operation. Moreover, the logic for the store operation needs to compensate for 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 element inside a BRAM. In case of security level 1, each memory element has a size of 292-bit which means that 4 bits at a certain location needs to be updated as the remaining 282 bits stay the same.

4.3 Arithmetic Units

The MAYO scheme operates on the finite field GF(2^4) and requires therefore units to perform addition, multiplication, and accumulation operations in this field as well as a specific unit to perform GF($(2^4)^m$) reduction. As shown in Section 2.1, the addition, subtraction and multiplication operations on the finite field of GF(2^4) can be done via a combination of bitwise AND and XOR operations. Note that addition and subtraction in GF(2^4) gives the same result leading to the fact that only one of them needs to be implemented. A throughout study of all functionalities of the MAYO scheme shows that we can reduce all of the necessary operations to just two base operations, addition and multiplication, and all other can be build through a combination of these two. As an example, the MAYO schemes requires a accumulation operation during vector or matrix multiplications, and this can be done by combining a multiplication, addition and a accumulator register as shown in Fig. 3.

Finite Field Addition/Subtraction. A addition and a subtraction on the finite field in GF(2^4) is equivalent, which enables performing both operations with one unit. The addition operation consists only of bit wise AND and XOR operations, as shown in Section 2.1.1, which are relatively cheap to perform in hardware. Further, we use a total of m addition units to for a m-ADD computation unit, where m is defined depending on the chosen security level. The grouping of multiple addition units to a m computation block is shown in Fig. 3.

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 bitwise AND and XOR operations, as shown in Section 2.1.2. Compared to addition, multiplication requires more bitwise evaluations and therefore consumes more resources in terms of LUTs. However, it is still relatively cheap to perform in hardware. Due to the packed data format, our core has to support m multiplication simultaneously to accelerate the computations. In total, m multiplication units form the m-MUL computation unit (shown in Fig. 3), where m is defined depending on the chosen security level.

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. The output of the multiplication is fed into the addition block that accumulates the output until a reset signal is set. The reset signal requires some extra logic to clear or keep the accumulated data inside the unit. Let's consider

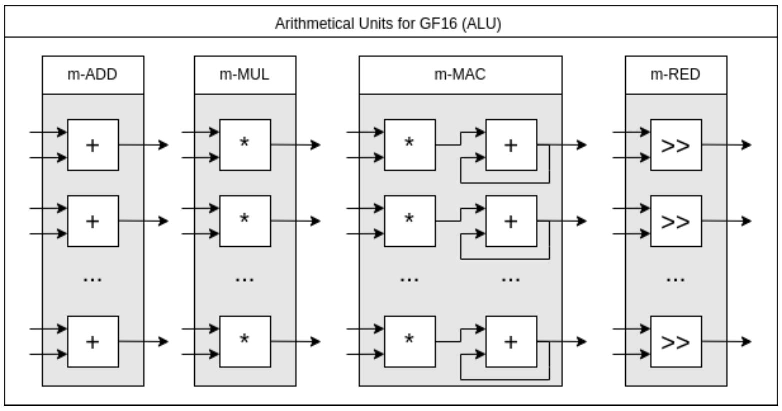

Figure 3: Overview of arithmetical units of MAYO core

a multiplication of two vectors $a =< a_0, a_1 >$ and $b =< b_0, b_1 >$, where both consist of two coefficients. The result c is the accumulated product of all coefficients, which can be written as $c = \sum_{i=0}^{n} a_i b_i$ or $c = a_0 b_0 + a_1 b_1$. As in addition and multiplication the accumulation is grouped into an m-MAC computation units, where m is defined depending on the chosen security level. MAC unit has one cycle of latency.

Matrix Multiplications by z in a Finite Field. During the computation and verification of the signature, a multiplication of $E^l y$ is required, where $E \in \mathbb{F}_q^{m\times m}$ 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, where m is 64, the irreducible polynomial $f_{64}(z) = z^{64} + x^3 z^3 + xz^2 + x^3$ is used during the computation of $E^l y$. 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. The resulting unit is called the m-RED unit and shown in Fig. 3.

4.4 Remaining Building Blocks

In previous sections, we discussed how we implemented the arithmetic and memory units required by the MAYO scheme as shown in Fig. 1. The scheme also needs some extra building blocks to perform operations, like bit-sliced encoding/decoding, sampling of random data, hashing with SHAKE256, and pseudo data generation using AES128 in counter mode (CTR).

Bitsliced Encoding and Decoding. The MAYO scheme employs a bitslicing technique to accelerate computations by utilizing AVX2 instructions which are supported by modern processors. The bitlicing is done by taking $s = m \times log_2(q)$ bits and shaping them into a matrix with dimensions $a \times b$, where $a = 64$ and $b = s/a$. After shaping the data into a matrix with dimensions $a \times b$, the bitsliced values can be read row-wise in four-bit chunks. Our implementation also uses this bitslicing technique to make logic placement easier since it allows us to perform all arithmetic operations by chaining bitwise AND and XOR.

Hashing via SHAKE256. The MAYO scheme requires SHAKE256 to hash the given message as well as its secret and public seed. In our implementation, we adapted the Keccak core presented in [AMI+22], which is capable of performing SHAKE128 and SHAKE256.

Pseudo-data generation via AES128. Another time intensive task is pseudo-random data generation. The MAYO scheme uses AES128 in counter mode to generate pseudo-random data from the public seed. This generated data is used for the two matrices $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$, which have a combined size of around 120 KB for security level 1. This size indicates that a fast generator is needed to avoid stalls during execution. Due to this, the software version employs AES128 in counter mode since it enables parallel computation of multiple data points via the AES-NI instructions. In our implementation, we employ two AES128 in counter mode to be able to generate one data point per cycle.

Table 1: Resource Utilization of the Proposed Architecture for MAYO_1

Unit	**LUTs**	**FFs**	**DSPs**	**BRAMs**
MAYO Core	22,527	6,930	-	81
⌊ Memory Wrapper	5,891	-	-	75
⌊ ALU Block	3,840	3,196	-	6
⌊ Keccak Core	9,488	3,162	-	-
⌊ AES128 Core (CTR[0])	1,508	282	-	-
⌊ AES128 Core (CTR[1])	1,227	284	-	-

Table 2: Performance of the Proposed Architecture for MAYO_1

Operation	**Latency** (in cc)	**Latency** (in *ms*)
Key Generation	60,154	0.6
Sign + ExpandSK	145,785	1.45
Verify + ExpandPK	77,755	0.77

5 Results

In this section, we present the area and performance results of our hardware architecture of the MAYO scheme with NIST security level 1, called MAYO_1. The proposed architecture performs all computations solely on hardware without requiring any software interaction. We coded the architectural units of our MAYO core using Verilog/SystemVerilog and verified functionality of the operations using behavioral simulations. We obtained area and performance results using Xilinx Vivado 2019.1 for PYNQ-Z2 Zynq-7020 with default synthesis and place & route settings at a frequency of 100 MHz.

Resource Utilization. In Table 1, we show the area utilization of our core for MAYO_1 and each of its sub-units. There are five sub-units, the memory wrapper, ALU blocks, Keccak core, and two AES128 cores, which are described in detail in Section 4. The core requires a total of 72 BRAMs, whereas the memory wrapper occupies the most with a total of 65 BRAMs and the ALU block with 7 BRAMs. The memory wrapper uses a grid of $16 \cdot 4 + 1 = 65$ BRAMs to store the data and all other temporary values, however, most of the memory is occupied by $\mathbf{P}_i$, $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, $\mathbf{L}_i$, and the pre-computed data required for $\mathbf{u}$, which is either $\mathbf{P}_i^{(1)} \cdot \mathbf{v}_i$ or $\mathbf{P}_i \cdot \mathbf{s}_i$. The remaining 7 BRAMs are used inside our ALU block to perform transposition of packed and unpacked matrix $\mathbf{A}$, which is required to solve the system of linear equations in the *SampleSolution*() function in Algorithm 2 in [BCC+23]. In terms of LUT usage, the Keccak core and the memory wrapper require the most, due to their high complexity. Note that the Keccak core employed (adapted from [AMI+22]) is able to perform SHAKE128, SHAKE256, SHA256, and SHA512, however, we only use SHAKE256 in our current design. The AES128-CTR cores and the ALU block do not require much LUTs compared to the other logic, which indicates that it would be possible to instantiate multiple of them to reduce the latency of intensive tasks, such as matrix multiplication and data generation. It is also noteworthy to mention that the high LUT usage of the memory wrapper is due to the complex data bus.

Performance Evaluation. In this section, we present the latency of each operation of the MAYO scheme for MAYO_1. In Table 2, we present the latency in clock cycles and *ms* for the key generation, signature generation, and signature verification. The overall latency is 60K, 146K, and 78K cycles for the aforementioned operations, respectively. Moreover, Table 3, 4, and 5 show a in-detail listing of every operations performed in each

Table 3: Breakdown of key generation latency for MAYO$_1$, which implements Algorithm 5 (MAYO.CompactKeyGen( )) of [BCC+23]

Operation	**Latency** (in cc)	**Perc.** (%)
Key Generation	60,154	(100%)
⌊ Generate $seed_{sk}$, $seed_{pk}$ and O	499	(1%)
⌊ Generate $P_i^{(1)}$ via AES128-CTR	22,244	(37%)
⌊ Generate $P_i^{(2)}$ via AES128-CTR	6,033	(10%)
⌊ Compute $A_i = P_i^{(1)} * O$	26,913	(45%)
⌊ Compute $B_i = A_i + P_2^{(1)}$	466	(1%)
⌊ Compute $C_i = B_i^T * O$	3,713	(6%)
⌊ Upper $P_2^{(3)} = Upper(B_i)$	65	(0%)
⌊ Output csk and cpk	65	(0%)

Table 4: Breakdown of signing latency for MAYO$_1$, which implements Algorithm 6 (MAYO.ExpandSK( )) and Algorithm 7 (MAYO.ExpandPK( )) of [BCC+23]

Operation	**Latency** (in cc)	**Perc.** (%)
Signature Computation	145,785	(100%)
⌊ Generate $seed_{pk}$ and O	499	(1%)
⌊ Generate $P_i^{(1)}$ via AES128-CTR	22,244	(15%)
⌊ Generate $P_i^{(2)}$ via AES128-CTR	6,033	(4%)
⌊ ExpandSK() $\rightarrow$ compute L_i	29,119	(20%)
⌊ **Loop:** Find Preimage for t		
⌊ Derive v_i and r	533	(1%)
⌊ Build linear System $Ax = y$		
⌊ Compute y	33,337	(25%)
⌊ Compute A	20,990	(15%)
⌊ SampleSolution(A, y, r)	22,827	(15%)
⌊ Compute signature	4,977	(3%)
⌊ Output signature	611	(1%)

function and their corresponding latency as well as their overall consumed runtime in percent. It can be seen that the operations with the highest latency are data generation via AES128-CTR and matrix multiplication to compute values like $\mathbf{P}_i^3$ in KeyGeneration(), $\mathbf{L}_i$ in ExpandSK(), $v_i^T \cdot \mathbf{P}_i^1 \cdot v_i$ in Sign(), and $s_i^T \cdot \mathbf{P}_i \cdot s_i$ in Verify(). Another high latency operation is searching for a *preimage* by building and solving a system of linear equations during the signature computation. This clearly shows that first a special analysis needs to be performed to find ways to accelerate the data generation of $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$. The second most important task is to search for techniques to accelerate large matrix multiplications, like the ones necessary to compute $\mathbf{P}_i^3$, $\mathbf{L}_i$, and more, since these multiplications are currently performed element by element. This is clearly not the ideal way as it would be possible to perform several operations in parallel.

Comparison with Related Works. MAYO team provides results for four different implementations of their scheme, (*i*) a generic C-implementation, (*ii*) an implementation with AES-NI, (*iii*) an implementation with AES-NI and AVX2, and (*iv*) an ARM implementation. Besides these, there is another ARM implementation [GMSS23] and one partial FPGA implementation [SMA+23] in the literature. We give a detailed comparison

Table 5: Breakdown of verify latency for MAYO_1, which implements Algorithm 7 (MAYO.ExpandPK( )) and Algorithm 8 (MAYO.Sign( )) of [BCC+23]

Operation	**Latency** (in cc)	**Perc.** (%)
Signature Verification	77,755	(100%)
⌊ Generate $seed_{sk}$, $seed_{pk}$ and O	494	(1%)
⌊ Generate $P_i^{(1)}$ via AES128-CTR	22,244	(29%)
⌊ Generate $P_i^{(2)}$ via AES128-CTR	6,033	(7%)
⌊ Decode signature s_i	622	(1%)
⌊ Hash Message M	127	(0%)
⌊ Compute $P^*(s)$		
⌊ Pre-Compute $P_i * s_i$	39,214	(50%)
⌊ Compute $u = s_i^T * P_i * s_i$	5,307	(7%)
⌊ Compute $y = \sum E^l u$	3,713	(5%)
⌊ Verify signature if $y = t$	1	(0%)

of these works with our implementation in Table 6. In comparison to the MAYO generic C-implementation, our work outperforms it by 2.09×, 1.92×, and 1.61×, respectively, for key generation, sign and verification operations. However, when it comes to their optimized version that utilizes AES-NI and AVX2, our implementation is 5.54×, 2.68×, and 7.77× and 12.00×, 2.3×, and 9.63× slower, respectively. However, when comparing these numbers one must take into account the difference in operating frequency of the target platforms, 2 GHz for Intel Xeon CPU and 100 MHz for Zynq-7020 FPGA. A more realistic comparison of our work with their ARM Cortex-M4 implementation, which runs at 120 MHz, shows a speed up of 72.84×, 52.77×, and 52.87×. Another implementation targeting ARMv7-M running at 480 MHz is presented in [GMSS23]; however, it does not support key generation. For the signature and verification functions, our work shows 61.75× and 15.43× speedups compared to their implementation with the configuration ($o = 7, k = 10$). Compared to their second configuration ($o = 7, k = 14$), our performance gain is even higher with 123.10× and 29.16×.

All the mentioned works show results for software implementation either on Intel processors or embedded platforms like ARM. To the best our knowledge, there is only one partial MAYO implementation for FPGAs, called HaMAYO [SMA+23], that gives results for the key generation and signing functions of MAYO_1. Compared to [SMA+23], our implement not only implements all functionalities, it also shows speedups 16.6× and 24.07 for key generation and signing, respectively. The overall good performance of our implementation can also be attributed to the parallel computation of all $m = 64$ data during all operations. Our solution consistently computes 64 values at once boosting overall performance compared to other works.

6 Proposed Optimizations

In Section 4, we present a proof-of-concept hardware implementation that provides useful insights on how to map the MAYO scheme to a hardware platform efficiently. In this section, we introduce several optimizations to improve the performance and area of the hardware implementation presented in Section 4. Note that we are already working on an implementation with the following optimization techniques.

Table 6: Results of Related Works for MAYO_1

Works (for MAYO_1)	**Platform**	**Freq.** (MHz)	**KeyGen** (cc/ms)	**Sign**[a] (cc/ms)	**Verify**[b] (cc/ms)
[BCC+23] (generic)	Intel Xeon	2,000	2,507K/1.25	5,569K/2.78	2,472K/1.23
[BCC+23] (AES-NI)	Intel Xeon	2,000	222K/0.11	1,087K/0.54	205K/0.10
[BCC+23] (AES-NI, AVX2)	Intel Xeon	2,000	110K/0.05	460K/0.23	175K/0.08
[BCC+23]	ARM C-M4	120	5,245K/43.70	9,183K/76.52	4,886K/40.71
[GMSS23] (o=7, k=10)	ARMv7-M	480	-	42,981K/89.54	5,703K/11.88
[GMSS23] (o=7, k=14)	ARMv7-M	480	-	85,682K/178.50	10,776K/22.45
[SMA+23]	Zynq-7020	100	996K/9.96	3,491K/34.91	-
Ours	Zynq-7020	100	60K/0.60	145K/1.45	77K/0.77

[a]: Including ExpandSK. [b]: including ExpandPK.

6.1 On-the-fly Coefficient Generation

The $\mathbf{P}$ matrices are the fundamental building block of the MAYO scheme. In total, MAYO uses three different $\mathbf{P}$ matrices, $\mathbf{P}^{(1)}$, $\mathbf{P}^{(2)}$, and $\mathbf{P}^{(3)}$. In Table 7, the sizes of the matrices with different security levels are shown.

Table 7: $\mathbf{P}$ matrix sizes for different NIST security levels (1, 2, 3, 5)

Matrix	MAYO_1	MAYO_2	MAYO_3	MAYO_5
$\mathbf{P}^{(1)}$	58×58	60×60	89×89	121×121
$\mathbf{P}^{(2)}$	58×8	60×18	89×10	121×12
$\mathbf{P}^{(3)}$	8×8	18×18	10×10	12×12
$\mathbf{P}$	66×66	78×78	99×99	133×133

Since we have m $\mathbf{P}$ matrices, there are $66 \times 66 \times 64 = 278784$ matrix elements for the smallest parameter set, MAYO_1. Following that every 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 $\mathbf{P}$ matrices. This would lead to extensive BRAM usage. The coefficients of the $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$ matrices are generated using AES128 with counter mode. Thus, it is possible to not store the matrix elements of those $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$ in memory but generate them on-the-fly instead. To be precise, every time some $\mathbf{P}^{(1)}$ or $\mathbf{P}^{(2)}$ matrix is needed in an operation, it is possible to use AES128 again to generate the matrix element instead of retrieving their elements from on-chip storage. Thus, it is only required to store the $\mathbf{P}^{(3)}$ matrices, which reduces the memory demand from 136KB to 2KB for MAYO_1.

6.2 Parallel Matrix Column Multiplication

Generating the coefficients on-the-fly enables reducing memory usage significantly, however, it raises the question of how to carry out the matrix operations efficiently. Essentially, matrix multiplication can be broken down into several vector-vector multiplications, where each row vector of the first matrix operand is multiplied with each column vector of the second matrix operand as shown in Eq. (16). These multiplications are common multiply-and-accumulate (MAC) operations. Therefore, the computations of the result matrix entries are independent of each other and we can parallelize them.

$$\begin{pmatrix} \cdots & a_1 & \cdots \\ \cdots & a_2 & \cdots \end{pmatrix} \times \begin{pmatrix} \vdots & \vdots \\ b_1 & b_2 \\ \vdots & \vdots \end{pmatrix} \tag{16}$$

The **P** matrices are generated in row-major order and due to our on-the-fly approach, we are not able to operate on more than one row simultaneously. However, the small size of the other matrices allows us to store them in registers and, thus, we can access all columns concurrently. Therefore, we can parallelize the multiplication of a single row with all columns by instancing multiple MAC units to carry out the computations.

6.3 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. The rationale behind this decision is to use AES128 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 two problems in a hardware implementation:

1. **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, generating random data, this approach creates some redundancy on the hardware level.

2. **Performance**: While the use of AES allows a significant speed-up on software level, its opposite is achieved on hardware. Our SHAKE core outperforms AES significantly. SHAKE128 generates 1344 bit every 26 cycles, which is more than $4\times$ faster compared to AES128 with an output of 128 bit every 12 cycles.

Therefore, using solely SHAKE instead of using both SHAKE and AES128 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 6.1 and Section 6.2 are also compatible with SHAKE.

6.4 Block Matrix Multiplication during Signature Verification

In the signature verification, we need to compute

$$\mathbf{s}_i^\top \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} \mathbf{s}_i. \tag{17}$$

Due to the optimization described in Section 6.1, 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, block matrix multiplication could be applied to Eq. (17), calculating the results of $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, and $\mathbf{P}_i^{(3)}$ individually and combining them accordingly using vector addition. Thus, only the intermediate results need to be stored, which are much smaller than the **P** matrices.

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

As a consequence of optimization presented in Section 6.3, the matrix elements are generated in row-major order. Therefore, we have to modify Eq. (13), the computation of $\mathbf{L}_i$ in the secret key expansion. Calculating the linear part using $\mathbf{L}_i = (\mathbf{P}_i^{(1)}\mathbf{O} + \mathbf{P}_i^{(1)\top}\mathbf{O}) + \mathbf{P}_i^{(2)}$ enables us to apply the optimizations described before. We generate the $\mathbf{P}_i^{(1)}$ elements on-the-fly and compute $\mathbf{P}_i^{(1)}\mathbf{O}$ and $\mathbf{P}_i^{(1)\top}\mathbf{O}$ in parallel. For the calculation of $\mathbf{P}_i^{(1)\top}\mathbf{O}$, we have to modify our MAC units. Instead of storing the intermediate values directly inside the unit, we need to store them in a BRAM and retrieve them for accumulation to deal with the transposition.

7 Conclusion

In this paper, we propose and implement a hardware architecture for the MAYO post-quantum signature scheme with NIST security level 1. The proposed architecture can perform key generation, signature generation, and signature verification operations in $0.6ms$, $1.45ms$, and $0.77ms$, respectively. We also propose several optimization techniques to improve resource utilization and performance of MAYO implementations on hardware platforms. We are already working on a hardware architecture with the proposed optimizations and plan to present it as future work.

Acknowledgement

This work was supported in part by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark.

References

[AMI+22] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. Kali: A crystal for post-quantum security using kyber and dilithium. Cryptology ePrint Archive, Paper 2022/1086, 2022. https://eprint.iacr.org/2022/1086.

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

[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.

[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.

[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.