Whipping the Multivariate-based MAYO Signature Scheme using Hardware Platforms

Florian Hirner
Graz University of Technology
Graz, Austria

Michael Streibl
Graz University of Technology
Graz, Austria

Florian Krieger
Graz University of Technology
Graz, Austria

Ahmet Can Mert
Graz University of Technology
Graz, Austria

Sujoy Sinha Roy
Graz University of Technology
Graz, Austria

Abstract

NIST issued a new call in 2023 to diversify the portfolio of quantum-resistant digital signature schemes since the current portfolio relies on lattice problems. The MAYO scheme, which builds on the Unbalanced Oil and Vinegar (UOV) problem, is a promising candidate for this new call. MAYO introduces emulsifier maps and a novel 'whipping' technique to significantly reduce the key sizes compared to previous UOV schemes. This paper provides a comprehensive analysis of the implementation aspects of MAYO and proposes several optimization techniques that we use to implement a high-speed hardware accelerator. The first optimization technique is the partial unrolling of the emulsification process to increase parallelization. The second proposed optimization is a novel memory structure enabling the parallelization of significant bottlenecks in the MAYO scheme. In addition to this, we present a flexible transposing technique for the data format used in MAYO that can be expanded to other UOV-based schemes. We use these techniques to design the first high-speed ASIC and FPGA accelerator that supports all operations of the MAYO scheme for different NIST security levels. Compared with state-of-the-art, like HaMAYO [24] and UOV [7], our FPGA design shows a performance benefit of up to three orders of magnitude in both latency and area-time-product. Furthermore, we lower the BRAM consumption by up to 2.8× compared to these FPGA implementations. Compared to high-end CPU implementations, our ASIC design allows between 2.81× and 60.14× higher throughputs. This increases the number of signing operations per second from 483 to 13424, thereby fostering performant deployment of the MAYO scheme in time-critical applications.

CCS Concepts

• **Hardware → VLSI design**; • **Security and privacy → Cryptography**; • **Computer systems organization → Architectures**.

Keywords

Post-Quantum, Digital Signature, MAYO, FPGA, ASIC

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690258

ACM Reference Format:
Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. 2024. Whipping the Multivariate-based MAYO Signature Scheme using Hardware Platforms. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690258

1 Introduction

Public-key cryptography encompasses essential cryptographic primitives for key exchange, public-key encryption, key encapsulation, and digital signatures. Widely used public-key cryptographic algorithms are based on integer factorization or discrete logarithm problems, which 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 [26] 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, Xanadu's 216-qubit Borealis and IBM's 433-qubit Osprey in 2022, among others. Considering such rapid advancements, agencies, industries, and research institutions strive to facilitate a smooth transition to quantum-resistant public-key cryptography, commonly known as Post-Quantum Cryptography (PQC).

Existing PQC algorithms are grouped into five main categories based on their foundational mathematical problems: code-based, hash-based, isogeny-based, lattice-based, and multivariate-based problems. Each category has 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. In July 2022, NIST selected one key encapsulation mechanism, namely Crystals-Kyber [25], and three signature algorithms, namely Crystals-Dilithium [2], Falcon [23], and SPHINCS+ [13] for standardization. The first three of them use lattice-based constructions, which leads to a need for more diversity in the PQC portfolio. Hence, in 2022, NIST issued a new call [17] 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 [4, 5] is a new post-quantum digital signature scheme based on the Unbalanced Oil and Vinegar (UOV) construction [16], a multivariate quadratic signature scheme. MAYO has

also been submitted to NIST's new diversification call for quantum-resistant digital signatures, and it is one of eleven signature schemes based on multivariate cryptography. MAYO reduces the key size significantly by using a minimal oil space and employs 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.

A new cryptographic signature scheme must be efficiently computable on diverse software and hardware platforms for broad adoption in real-world applications. Several implementation methods must be researched to examine the schemes' speed, memory, and energy efficiency while considering specific application and platform requirements. This was also the case in the first three rounds of the NIST standardization project. There, we have seen numerous papers investigating the secure and efficient implementation [3, 8, 14, 15, 19] of novel PQC algorithms on high-end software, resource-constrained microcontroller, FPGA and ASIC hardware, and other platforms. Just few works are available in literature that consider implementations of UOV-based schemes, like MAYO.

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 [22]. Another work [10] focuses on porting and optimizing the MAYO scheme for ARM microcontrollers and proposes new parameters to improve the signing and verification processes. Recently, an FPGA implementation of the MAYO scheme is proposed [24] that implements the MAYO scheme but does not support signature verification. Apart from MAYO works, some publications target the closely related UOV scheme [7] which the MAYO scheme is based on. Many operations performed in the MAYO scheme are similar to the UOV scheme. Hence we consider UOV works for comparison. At the time of writing this paper, there is one UOV work [7] that fully implements the UOV scheme. This work analyses techniques to port UOV to microcontrollers and FPGAs.

Application-area: Integrating multivariate signature schemes such as MAYO in high-performance real-time systems is constrained by the computationally intensive UOV mathematics. On the resource constrained Arm M4 microcontroller [5] running at 24MHz, a single signature generation at the lowest security level takes approximately 0.38 seconds. Furthermore, a highly optimized AVX2 software implementation [5] on an Intel Xeon Gold 6338 CPU at 2.0 GHz achieves only 4348, 1205, and 483 signature generations per second at security level 1, 3, and 5, respectively. This level of performance is insufficient for real-time applications (e.g., servers), particularly when compared to Hardware Security Modules (HSMs) like the Thales HSM [11], which can perform up to $10,000$ RSA and $20,000$ ECC operations per second. Our proposed hardware accelerator at 1.5 GHz achieves 48188, 22840, and 13424 signature generations per second for each security level.

Contributions: This paper investigates the potential of hardware acceleration to significantly improve the throughput and efficiency of the MAYO algorithm. We deploy several techniques dedicated to hardware platforms to enhance the performance of the MAYO scheme. In addition, our techniques also apply generally to the base UOV scheme [7]. Our contributions are as follows:

- First, we propose partial loop unrolling to reduce the latency of the emulsification process in the MAYO scheme. Our highly parallelized architecture allows the reconfiguration of the arithmetical block to perform the emulsification of different iterations simultaneously. This accelerates the emulsification in both signing and verification from $(k \cdot k + 1)/2$ to k iterations.
- Second, we present a novel memory structure that is both performant and memory-saving simultaneously. The memory structure allows a high degree of parallelization of matrix multiplications, which enables us to increase the performance. The performance benefits especially affect matrix multiplications involving oil space and vinegar maps. These are the major computation bottlenecks. Additionally, the memory structure allows us to perform the loop unrolling of nested loops during signature generation and verification.
- Third, 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.
- Fourth, we optimize the efficiency of our design by reducing the amount of memory to reach a low area-time product (ATP). Contrasting to other works, we do not store pseudo-randomly generated data in memory but immediately consume it for computations. This allows us to halve the required on-chip memory consumption on FPGAs. The impact on memory savings on ASIC platforms is even higher.

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. We tested 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 [16]. It also contains a description of the MAYO signature scheme, which gives 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 and Section 4 shows how they are Incorporated. Section 5 we discuss the scheduling of the key generation, signing and verify operation. Section 6 gives an in-depth ablation study of our optimization strategies. Section 7 presents the evaluation and results compared to related works and in Section 8, we discuss the security aspects and implications of our work.

2 Background

This section covers the background necessary to understand arithmetic used in the UOV [7] and MAYO [4] scheme.

2.1 Finite field arithmetic 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 $\mathbb{F}_{16} = \mathrm{GF}(2^4)$.

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 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.

2.2 Multivariate Quadratic Maps

The core of the Oil and Vinegar [16] and the MAYO scheme are multivariate quadratic maps. We follow the definition and notation presented in [4]. 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 and the coefficients of the polynomials are in $\mathbb{F}_q$, the finite field with q elements. This map is evaluated by simply evaluating each polynomial p_i. Both, UOV and MAYO use homogeneous multivariate maps in the sense that every polynomial is in homogeneous form. In the quadratic case this results in polynomials where every nonzero term is quadratic. Let c_{ij} be the coefficient of the quadratic term x_ix_j. Then, a polynomial can be expressed as

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

Eq. 1 can be rewritten into

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

MAYO uses this upper triangular matrix form of Eq. 2 and defines polynomial evaluation as

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

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 original Oil and Vinegar algorithm was presented by Patarin [20] in 1997. The motivation was to create a cryptographic scheme based on a set of multivariate quadratic equations, since solving such equations is NP-hard [16]. The main idea behind these schemes is to introduce a trapdoor into the set of equations to allow efficient sampling of preimages. While the original scheme was broken, its modified version Unbalanced Oil and Vinegar still seems to be secure at the current state and, therefore, was chosen as the foundation of the MAYO scheme.

The description and notation of the Oil and Vinegar signature scheme is adapted from [4]. 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 m dimensional 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 and, thus, a basis of the oil space acts as the secret key. Specific attacks against Oil and Vinegar schemes try to recover the oil space from the public key, which led to the break of the original Oil and Vinegar version. Since O is hidden in $\mathbb{F}_q^n$, increasing n for a fixed m renders these attacks more difficult. The Unbalanced Oil and Vinegar scheme follows this principle and uses $n = 3m$ to prevent the recovery of the oil space.

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 [4]. However, the Oil and Vinegar scheme suffers from large public key sizes in the order of 50 kB, which limits the use case as a practical signing algorithm.

2.4 MAYO Signature Algorithm

In this section, we give a short description of the MAYO scheme. The description and notation is adapted from [4] according to the latest specifications described in [5]. Readers may refer to [4, 5] for more details. MAYO modifies the original Oil and Vinegar scheme to tackle the problem of large key sizes. The design philosophy is closely related to UOV and follows the same principles. Its security is also based on the MQ problem, and signatures are preimages of the hashed message under a multivariate quadratic map. The key generation, signing, and signature verification are similar to those of Oil and Vinegar. The main difference lies in the choice of the oil space dimension. MAYO uses an oil space that is too small for the original scheme. Since the oil space is hidden in $\mathbb{F}_q^n$, the smaller its size, the harder a search for it. Thus, the other parameter can be reduced without compromising the scheme's security. However, this change renders the signature sampling impossible in most cases using the Oil and Vinegar algorithm. As a solution to the problem, MAYO uses 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 for a smaller oil space and significantly reduces the key size. 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.* [21] 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

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

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

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

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

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

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.* [5], 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

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

The constant part can be calculated using

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

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

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

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 $(salt||\mathbf{s}_1, \ldots, \mathbf{s}_k)$, 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.

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

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

Table 1: $\mathbf{P}_i$ matrix sizes for different NIST security levels

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

The emulsifier map $\mathbf{E}$ now represents a multiplication by z. Analog to field multiplication in $GF((2^4)^m)$, we need to reduce the resulting polynomial, to receive a valid $GF((2^4)^m)$ 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:

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

It is important to note that the additions and multiplications in Eq. 14 are $GF(2^4)$ operations. This approach blends well with our packed format described in Section 4.3, as we are able to load m values and, therefore, a whole $GF((2^4)^m)$ element in one cycle in hardware. To evaluate $\mathbf{E}^l\mathbf{u}$, we perform this computation l times.

3 Optimization Strategies

This section presents several high-level optimizations to improve MAYO's performance and memory consumption on hardware.

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. Table 1 shows the dimension of the $\mathbf{P}_i$ and its sub-matrices for NIST security levels 1 ($MAYO_1$), 3 ($MAYO_3$), and 5 ($MAYO_5$). The table indicates the large size of the $\mathbf{P}_i$ matrices which 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 $GF(2^4)$ and we need 4 bit to represent each element, leading to 136 kB that would need to be stored in memory. We optimize the efficiency of our design by reducing the memory consumption of each $\mathbf{P}_i$ from $n \times n$ elements to just $o \times o$ elements. This is possible since 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 required to store the $\mathbf{P}_i^{(3)}$ matrices. This measure reduces the overall memory demand while not affecting the performance.

3.2 Parallel Matrix Column Multiplication

Generating the coefficient of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ on the fly significantly reduces the memory usage. However, it still allows us to carry out the matrix multiplications 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. To take advantage of this observation, we instantiate as many MAC units as there are columns in $\mathbf{B}$. In the 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.

3.3 Parallelizing Computation of $\mathbf{L}_i$

The on-the-fly generation of the $\mathbf{P}_i^{(1)}$ matrices demands proper treatment in various locations of MAYO. One such location is the MAYO.EXPANDSK() operation, shown in Eq. 16. The shown addition of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(1)\top}$ is not easily possible since we generate the $\mathbf{P}_i^{(1)}$ matrix in a row-wise manner to simplify on-demand seed expansion. The row-wise generation affects the computation of $\mathbf{L}_i$ in MAYO.EXPANDSK() (Step 17 of Algorithm 6 of [5]). Hence, 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. The row-wise generation is exactly the required order for a straightforward matrix-matrix multiplication as discussed in Sec. 3.2. Thus, we use a simple multiply and accumulate (MAC) unit for this computation. Yet, the row-wise generation order of $\mathbf{P}_i^{(1)}$ corresponds to a column-wise generation of $\mathbf{P}_i^{(1)\top}$, which hinders a straightforward matrix-matrix multiplication in the second multiplication. The following examples in Eq. 17 and Eq. 18 give an impression of this limitation and present 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.

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

Eq. 17 shows a standard matrix multiplication of $\mathbf{Po}$. 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

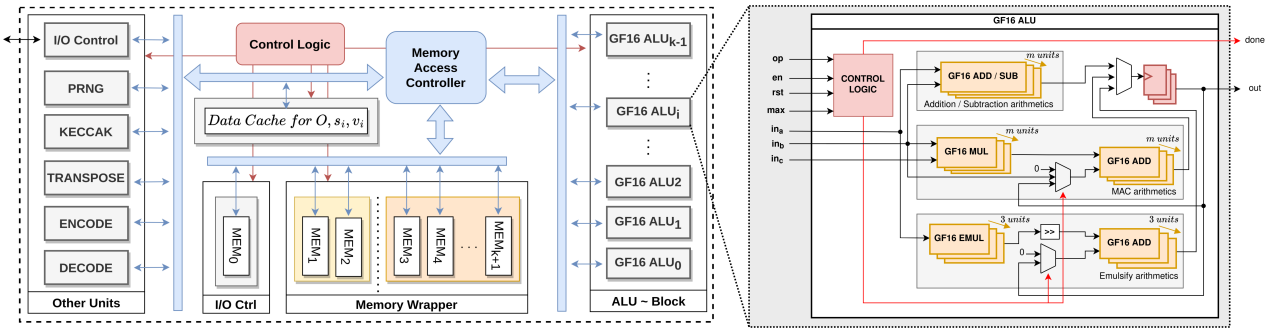


Figure 1: Overview of our high-speed architecture and its sub-modules

of the respective row of $\mathbf{P}$ are consumed. This procedure is repeated for each row in $\mathbf{P}$ (colored blue and orange).

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

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 of $\mathbf{P}_i^{(1)}$ through the computations.

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. 3.2. 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 $\mathbf{P}_i^{(1)\top}\mathbf{O}$. Both MAC and BMAC operations take the same elements of $\mathbf{P}_i^{(1)}$ and $\mathbf{O}$ as input during the matrix multiplications.

3.4 Block Matrix Multiplication during Signature Verification

In the signature verification, we need to compute line 24 in Alg. 9 MAYO.VERIFY() [5].

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 $\mathbf{P}_i^{(1)}$, when AES128 is used as pseudo-random function. Therefore, we apply a block matrix multiplication to line 24 in Alg. 9 MAYO.VERIFY() [5] 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.

4 The Proposed Hardware

This section explains the proposed hardware architecture and the main arithmetic blocks in a bottom-up fashion. We start with the overall design of our architecture to give an overview. Then, we present our pseudo-random data sampling, arithmetical units, memory management, and transpose units.

4.1 Overall Design of Processor

This section shows how we combine all the components into one core. The overall architecture of the core is shown in Fig. 1. The right side of Fig. 1 shows the arithmetical block unit discussed in Sec. 4.4. It instantiates k many GF(16) 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 GF(16) ALU. These memory banks are part of the MEMORY WRAPPER discussed in Sec. 4.3, as shown in the bottom of Fig. 1. 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. The control logic uses a Finite-State-Machine (FSM) approach 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.1. The FSM is already designed to be easily replaceable with an instruction-set architecture (ISA) to allow fast adaption to possible changes in the scheme.

4.2 Hashing and Pseudo-Random Data

In the latest specifications of MAYO [5], SHAKE256 is used for hashing and AES128 in counter mode to generate data from a public seed. An analysis of the MAYO software implementation shows that only a small time-share of the execution is spent on hashing via SHAKE256. However, the major time-share is spent on pseudo-random data sampling via AES128. The sampling based on AES128 in software benefits from the AES-NI instruction-set extension [1]. 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.

There are two approaches to implement AES128 in counter mode. One can use either an iterative or a fully pipelined unrolled approach [27]. We will refer to the iterative approach as AES128-R and to the fully pipelined unrolled one as AES128-P through the

Table 2: Latency of pseudo-random sampling of $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ via iterative and fully unrolled AES128

Sec. Level	Matrix	Elements	AES128-R (cc)	AES128-P (cc)
$MAYO_1$	$\mathbf{P}_i^{(1)}$ / $\mathbf{P}_i^{(2)}$	1,711 / 464	22,243 / 6,043	1,722 / 475
$MAYO_3$	$\mathbf{P}_i^{(1)}$ / $\mathbf{P}_i^{(2)}$	4,005 / 890	52,065 / 11,581	4,016 / 902
$MAYO_5$	$\mathbf{P}_i^{(1)}$ / $\mathbf{P}_i^{(2)}$	7,381 / 1,452	95,953 / 18,786	7,392 / 1,464

rest of the paper. The iterative approach (AES128-R) instantiates one full AES round function that is iterated multiple times to perform one whole encryption. The advantage of this approach is the low area utilization since modules, like SUBBYTES(), SHIFTROWS(), MIXCOLUMNS(), and ADDROUNDKEY(), are reused multiple times. The main disadvantage of AES128-R is the lower throughput as each encryption occupies the hardware instance completely. In our architecture, the AES128-R requires around 1.2k LUTs, 600 FFs, and outputs 128 encrypted bits after 12 cycles.In contrast to the iterative approach, the pipelined and fully unrolled approach (AES128-P) gives high throughput but causes high resource consumption. Compared to the AES128-R, AES128-P consumes around 10k LUTs, 6k FFs, and delivers 128 bits in each cycle after filling the pipeline.

In MAYO, AES128 is mainly responsible for generating the two matrices $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$. The size of these two matrices depends on the selected NIST security level, which is either 1, 3, or 5. In case of the $\mathbf{P}_i^{(1)}$ matrix, its size is either 1711, 4005, or 7381 elements while the size of $\mathbf{P}_i^{(2)}$ 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 $\lceil s_m \cdot s_e / r \rceil \cdot l$, 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 that the bit-rate for AES128 is 128 bit per 12 cycles in AES128-R and 128 bit per cycle in AES128-P once the pipeline is filled. Table 2 shows the total latency that both approaches require for sampling. Comparing the latency shows that AES128-P is around 12× to 13× faster than AES128-R while requiring around 9× more resources. Note, that we use $n = wordsize/128$b many AES cores for each security level, where the $wordsize$ is either 256b, 384b, or 512b.

4.3 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 follows:

(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. 2a. 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 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. 2b. 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.

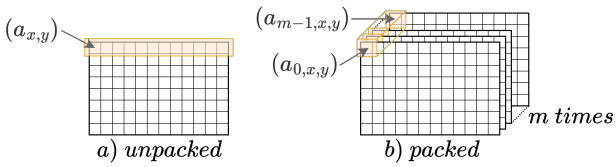


Figure 2: Overview of packed and unpacked memory format

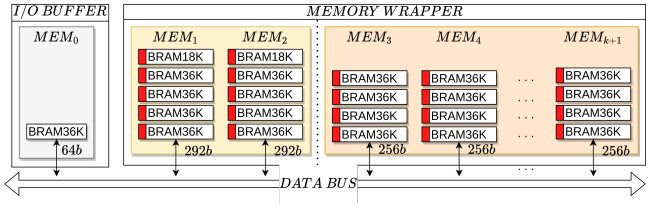


Figure 3: Memory grid layout of our $MAYO_1$ core

The m multivariate quadratic polynomials operate on input values with the same indices, as shown in Fig. 2b. 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. 3, for its configuration in security level 1. The architecture of our core consists of an I/O BUFFER and the actual 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. 3. 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. 3 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.

4.3.1 Parallel Computation using our Memory Layout. The described memory structure allows for the full parallel computation of various operations. These operations include matrix-vector and vector-vector multiplication, which are parallelized over the parameters k and o respectively, as explained in Sec 3.2. To give an example of the parallelization, we consider the computation of $\mathbf{u}$ during MAYO.SIGN() (line 24 in Alg. 8 in [5]). We can simultaneously perform the k computations of $\mathbf{P}_a^{(1)}\mathbf{v}_0, \mathbf{P}_a^{(1)}\mathbf{v}_1, \ldots, \mathbf{P}_a^{(1)}\mathbf{v}_{k-1}$

through our dedicated memory structure. Here, each $MEM_{3...k+1}$ memory stores the data of a single $\mathbf{v}_i$ vector to allow parallel computation of all $\mathbf{v}_i$. Note that the operands involved in such parallel computations are either $\mathbf{O}$, $\mathbf{s}_i$, $\mathbf{v}_i$, $\mathbf{L}_i$, $\mathbf{P}_a^{(1)}\mathbf{v}_i$, $\mathbf{P}_a\mathbf{s}_i$, etc. The sizes of these operands are between $n-o$ and n elements. Therefore, the memory banks $MEM_{3...k+1}$ need a depth of $max(n, n-o) = n$ elements to store the required data. This leads to a depth of $n = 66$ when considering MAYO_1 security level.

4.3.2 Memory Consumption. Table 1 shows the memory consumption of $\mathbf{P}_i$, $\mathbf{P}_i^{(1)}$, $\mathbf{P}_i^{(2)}$, $\mathbf{P}_i^{(3)}$, and $\mathbf{L}_i$ for MAYO_1. Not taking advantage of the upper triangular form of the $\mathbf{P}_i$ matrix would require a total of 136.125 kB. Under consideration of the upper triangular form, the memory demand for $\mathbf{P}_i$ reduces to 69.094 kB. However, our architecture only stores $\mathbf{P}_i^{(3)}$ and $\mathbf{L}_i$ in memory. The other components $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ are generated on-the-fly, as discussed in Sec. 3.1.

The optimizations mentioned above, combined with the customized memory layout discussed in Sec.4.3, result in reduced BRAM consumption. Specifically, for MAYO_1, the required number of BRAM36K units can be calculated as $1+2\cdot4.5+(k-1)\cdot4+3.5 = 45.5$. As illustrated in Fig.3, one BRAM36K unit is designated as an IO Buffer, while $2\cdot4.5+(k-1)\cdot4 = 41$ BRAM36K units are allocated for parallel computation to supply the ALU, as elaborated in Sec. 4.3.1. Additionally, 3.5 BRAM36K units are utilized to cache the oil space and vinegar maps. Thus, the total on-chip memory requirement for MAYO_1 amounts to 45.5 BRAM36K units.

The memory blocks $MEM_{3...k+1}$ marked orange in Fig. 3 use a depth of $max(n, n-o) = n$ elements. This is due to the utilization of these BRAMs for temporal values like $\mathbf{P}_a^{(1)}\mathbf{v}_i$ and $\mathbf{P}_a\mathbf{s}_i$. As, the FPGA platform provides BRAMs with a fixed depth of 512 words only, $512-n$ memory locations remain unused. In contrast, on ASIC we reduce the memory requirement to a depth of n by utilizing memories with a finer granularity. This allows us to save memory in the $k-1$ memory units in ASIC compared to FPGA.

4.3.3 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 swapping 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.

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.

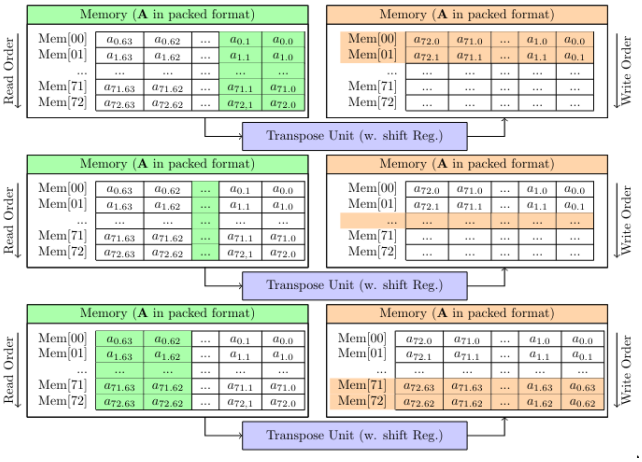


Figure 4: Example of the transpose operation with $\tau = 2$

We developed a scaleable 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 $\tau = 2$. Fig. 4 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. 4. 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.

$$transpose\ latency = \lceil \#rows/\tau \rceil \times \#columns \tag{19}$$

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, as shown in Eq. 19 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 transpose unit depending on the available resources in hardware.

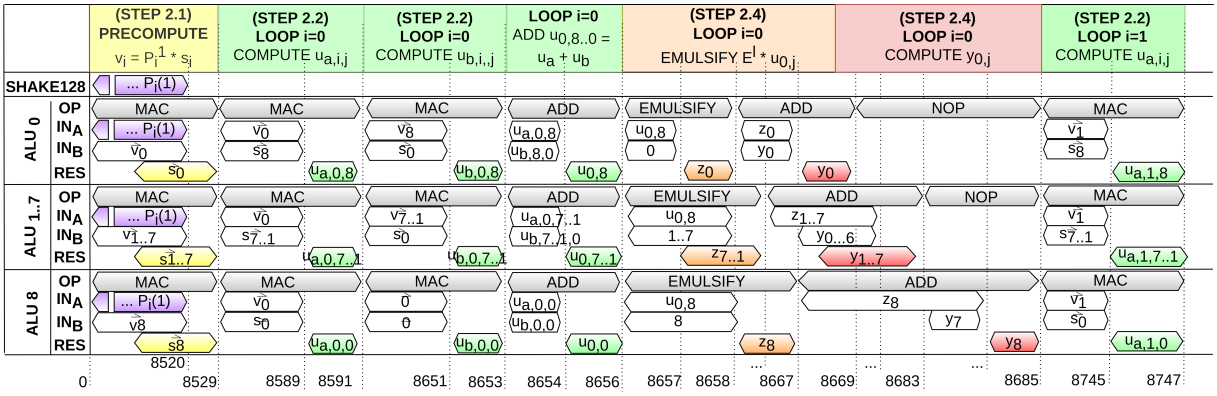


Figure 5: Scheduling overview of computation loop to compute y of MAYO.Sign()

4.4 Arithmetic 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 multiply-accumulate. 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. The architecture of our ALU-Block is shown in the center of Fig. 1. It consists of k many instances of our arithmetical units (GF(16) ALU$_i$). The internal architecture of one GF(16) ALU is shown on the right side in Fig. 1. The ALU 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 only process one GF(16) 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.

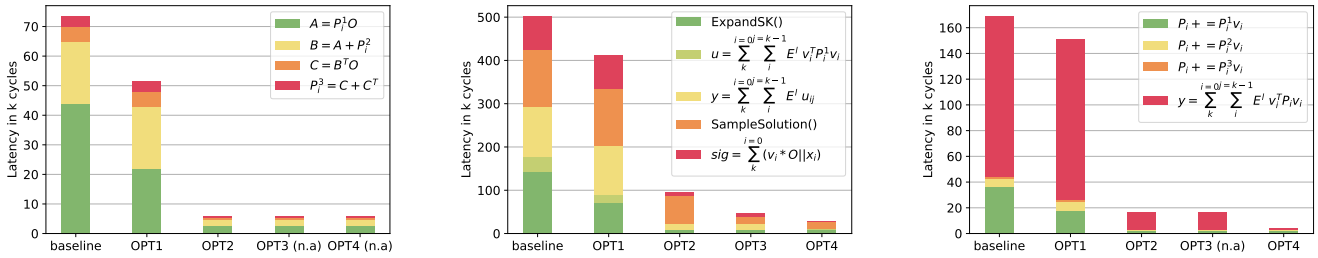
5 Scheduling of Operation

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 the original key generation operation is shown in Alg. 5 in [5]. The main computation happens in lines 2 and 3. 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 : 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 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. 3.2.

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 Aes128 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. 3.3. 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 used for signature generation and follows Mayo.ExpandSK(). The function follows Alg. 8 of [5]. The MAYO.Sign() function needs to find a preimage for a given hash $\mathbf{t}$ of the digested message and a given salt. Obtaining a preimage of $\mathbf{t}$ requires solving a linear system ($\mathbf{Ax} = \mathbf{y}$), which is quite hard. 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 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 [5]. The counter ctr keeps track of the number of



(a) Mayo1.KeyGen (b) Mayo1.Sign + Mayo1.ExpandSK (c) Mayo1.Verify + Mayo1.ExpandPK

Figure 6: Performance impact of our techniques in terms of overall latency (n.a. means not affected by this optimization).

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. 3.2. 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 lines 24-34 of Alg. 8 in [5]. 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 unrolling the inner loop. Therefore, our Alu Block uses up to k many GF(16) ALUs. Each GF(16) ALU is responsible for computing one of the $i-(k-1)$ loop iterations in the inner loop. Fig. 5 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() (Alg. 2 in [5]) 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. The probability that the sampling of a solution fails is increasingly low, however, if it happens the loop has to run again (step 1-4). In case the sampling of a solution is successful, the computations can 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 shown in Alg. 9 in [5] 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 $\mathbf{P}_i^{(3)}$. We split the computation of $\mathbf{P}_i\mathbf{s}_i$ into three blocks for each of the sub-matrices as described in Sec. 3.4.

6 Detailed Ablation Study

This section analyses the effect of each of our optimization techniques presented in Sec. 4 and discusses the impact on the overall performance and memory usage. The analysis includes on-the-fly coefficient generation, parallel multiplication, and other critical methods for security level MAYO_1 and Aes128-P. The impact of these optimizations on the latency of Mayo.KeyGen, Mayo.Verify, and Mayo.Sign is visually represented in Fig. 6a, Fig. 6b, and Fig. 6c, respectively. The first bar of each diagram represents the baseline implementation without any of our optimization techniques. The following bars add one optimization per bar to illustrate the impact of each optimization.

We first consider the on-the-fly coefficient generation method (OPT1) as discussed in Sec. 4.2. This optimization reduces total memory consumption and lowers latency. Without this technique, memory consumption increases from 2KB to 130KB at the MAYO_1 security level, with even higher demands at increased security levels. Furthermore, state-of-the-art works [24] first generate data like $P^{(1)}$, and store it in memory. Once all data is generated the actual computation is started. In contrast to that, our approach directly consumes the generated data which reduces the latency. This latency reduction affects step 16 in key generation, step 30 in sign, and step 24 in verify, as shown in the OPT1 bars in Fig. 6.

The design of our memory structure (OPT2), explained in Sec. 4.3, enables parallel read and write operations during computation. Without this optimization, the low memory bandwidth would lead to an under-utilization of our parallel GF16 ALU cores. All computation steps of each algorithm are affected by this memory structure that allows high parallelization. The impact of OPT2 is illustrated in Fig. 6. The large difference between OPT1 and OPT2 in all three operations shows the significant latency reduction enabled by this optimized memory structure.

The signature generation requires a transpose operation of the $\mathbf{A}$ matrix during sample solution. Compared to state of the art works [7, 24] that use index based transposition, we choose a pipelined multi-pass approach that operates on multiple columns simultaneous due to the unpacked data format that is used to store the matrix $\mathbf{A}$ during sample solution. Transpositing $\mathbf{A}$ is particularly challenging due to the sub-byte granularity of the unpacked data, as explained in Sec. 4.3.3. Our scalable and pipelined approach of the transpose operation improves the efficiency and area utilization. The pipelined loading, transposing, and storing of data minimize the latency of transpose operations, as explained in detailed in

Sec. 4.3.3. Fig. 6b shows the impact of OPT3 on the performance of signature generation.

The partial-unrolling technique (OPT4) enhances the performance of the emulsification process in both signature generation and verification by allowing existing arithmetic blocks to execute multiple iterations simultaneously (as described in Sec. 5). This optimization reduces the required number of iterations from $\frac{k(k+1)}{2}$ to just k for both signing and verification processes as illustrated in Fig. 6b and 6c. Additionally, it conserves memory resources, as data is processed immediately in subsequent operations, avoiding unnecessary interaction with memory.

In conclusion, this detailed study in addition with Fig. 6a, Fig. 6b, and Fig. 6c clearly demonstrate the substantial benefits of our optimization techniques. These include drastic reductions in memory usage, computational latency, and overall system overhead, underlining the importance of each method in achieving a highly efficient hardware implementation.

7 Evaluation and Results

In this section, we present the area and performance results of our hardware architectures with round-based AES128-R and fully-unrolled AES128-P seed expansions for different security levels of MAYO. We implemented our design in SystemVerilog and synthesized the ASIC design for 28nm technology. Furthermore, we verified our MAYO accelerator by synthesizing, implementing, and running it on FPGAs. For that, we used the Vivado 2022.2 tool and targeted Artix7 (xc7a200t) and Kintex7 (xc7k410t) FPGAs. We chose these FPGA platforms to allow a fair comparison to related works. Related works [7, 24] present results for different Artix7 platforms. However, our high-speed design for $MAYO_3$ and $MAYO_5$ does not fit on the Artix7 FPGAs. Hence, we implemented $MAYO_3$ and $MAYO_5$ on the larger Kintex7 FPGA. Kintex7 and Artix7 FPGAs use the same AMD 28nm [29] High-Performance, LowPower (HPL) technology [28, 30], but Kintex7 has a higher resource budget. Still, the performance figures are the same, which allows a fair comparison between these two architectures.

7.1 Area and Performance Results

In the first part of this section, we present the performance results of our hardware architectures with both round-based (AES128-R) and fully-unrolled (AES128-P) seed expansions. In the second part, we provide an in-detail analysis of the area utilization of all the sub-modules used to build our high-speed architecture.

Performance results: The performance results of our high-speed hardware architectures with AES128-R and AES128-P are shown in Table 5. The table shows the latency of each operation in cycles and *ms* for different security levels. The architecture with AES128-P achieves up to 6.0×, 8.3× and 3.5× better latency compared to the architecture with AES128-R for MAYO.KEYGEN(), MAYO.VERIFY() and MAYO.SIGN(), respectively, across different security levels. These speedups are less than the sole PRNG performance difference of 13× between AES128-R and AES128-P primitives. However, this is expected since not all operations depend on the performance of the PRNG that generates data on-demand through seed expansion. It is noteworthy that the architecture with AES128-P requires between 10% and 20% more LUTs and FFs compared to the architecture with AES128-R. However, this area overhead is negligible compared to the increase in performance between 3.0× and 8.3× across all operations and security levels of the MAYO scheme.

Table 3: Resource utilization of sub-modules on FPGA.

Design	Resources	Share in %										Count
		Memory	Caches	AES128	Keccak	ALU-Block	KeyGen	ExpandSK	Sign	ExpandPK	Verify	Total
MAYO with seed expansion using AES128-R												
$MAYO_1$ (2×AES128-R)	LUTs	15.4	0.1	3.2	12.3	18.9	2.9	7.4	34.5	0.0	5.5	91,146
	FFs	0.1	24.1	2.2	9.8	37.2	1.0	1.3	22.4	0.0	1.9	33,060
	BRs	98.9	1.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	45.5
$MAYO_3$ (3×AES128-R)	LUTs	25.1	0.1	2.6	6.0	22.2	3.3	5.6	27.7	0.0	7.4	158,537
	FFs	0.1	12.6	1.8	6.0	41.6	6.3	9.2	20.5	0.0	1.9	55,388
	BRs	99.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	96
$MAYO_5$ (4×AES128-R)	LUTs	31.2	0.9	4.7	2.5	21.5	8.6	4.9	25.0	0.0	5.8	218,607
	FFs	0.1	12.6	1.8	6.0	41.6	6.3	9.2	20.5	0.0	1.9	77,336
	BRs	99.7	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	194.5
MAYO with seed expansion using AES128-P												
$MAYO_1$ (2×AES128-P)	LUTs	16.4	0.1	19.1	10.3	15.5	1.1	6.3	26.6	0.0	4.6	109,503
	FFs	0.6	19.9	15.4	8.1	30.8	0.8	1.2	18.6	0.0	4.5	38,473
	BRs	98.9	1.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	45.5
$MAYO_3$ (3×AES128-P)	LUTs	17.5	0.1	17.3	5.1	18.6	2.9	4.9	27.8	0.0	5.9	181,632
	FFs	0.4	22.0	14.2	5.1	34.8	0.6	0.8	17.4	0.0	4.8	62,247
	BRs	99.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	96
$MAYO_5$ (4×AES128-P)	LUTs	20.4	0.0	16.1	4.0	19.1	3.4	5.8	20.4	0.0	10.7	256,450
	FFs	0.1	24.6	14.0	3.8	37.3	0.4	0.6	16.6	0.0	2.7	85,360
	BRs	99.7	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	194.5

Area utilization: Our architecture has a rather high LUT utilization on FPGA, as Table 3 shows. This is due to the extensive parallelization of computations. The MEMORY-UNIT incorporates a complex data bus, which allows reading and writing data from multiple memories to feed k parallel ALU-BLOCKS. Although this enables our design to perform most computations in parallel and avoid loading the same data from memory multiple times, it increases implementation complexity. The inputs and outputs of all ALU-BLOCKS are buffered to improve the frequency of the design. This also increases the FF utilization since about one-third of the overall FF utilization is used for the ALU-BLOCKS. The following example gives an impression of the FF consumption of the ALU-BLOCK. Every ALU-BLOCK has k ALU-UNITS and each ALU-UNIT has three inputs, one accumulation buffer, and one output. Each input, accumulation buffer, and output has a size of $(k \cdot o+1) \cdot \log_2(16) \in \{292\text{b}, 444\text{b}, 580\text{b}\}$, depending on the chosen security level. This leads to an FF consumption of $k \cdot (3+1+1) \cdot (k \cdot o+1) \cdot \log_2(16) \in \{13140\text{b}, 24420\text{b}, 34800\text{b}\}$ just for buffering in the ALU-BLOCK. Note that the implementation results in Table 3 are slightly lower due to the optimization strategy of Vivado. The rest of the FF in our architecture is mainly used within modules like CACHES, KECCAK, and SIGN.

In our architecture, BRAM utilization is relatively low thanks to on-demand seed expansion and optimized memory structure with careful computation scheduling. Combining these techniques allows us to lower the required BRAMs compared to 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, as Table 5 shows. For ASIC implementations of our design, on-chip SRAMs use the most area, namely between 59% and 76% of the total area. For the architecture with AES128-R for $MAYO_1$, $MAYO_3$ and $MAYO_5$, 71%, 68%, and 76% of the total

area are consumed by on-chip SRAMs, respectively. Similarly, for $MAYO_1$, $MAYO_3$ and $MAYO_5$, 59%, 59%, and 69% of the total area of the architecture with AES128-P are consumed by on-chip SRAMs, respectively. On ASIC, the on-chip SRAM consumption takes up the majority of the area, which shows that our memory optimization strategy has a much higher impact on ASIC compared to FPGA platforms.

Area utilization of sub-modules: Our high-speed hardware architecture includes multiple sub-modules as shown in Fig. 1. In addition to this, Table 3 shows the detailed resource distribution over all sub-modules, like MEMORY, CACHES, AES128-PRNG, KECCAK, ALU-BLOCK, and more for the architectures with AES128-R and AES128-P. The largest share of LUT consumption, around 21% to 34%, is dedicated to the module that performs the signature generation. This is due to the sampling of the solution which includes units to perform transpose, echelon, and back substation. The ALU-BLOCK and MEMORY-UNIT are the second and third largest modules in terms of LUT. These two modules are required to feed and compute data in a parallel manner. The remaining modules only cause a minor share of LUT consumption.

7.2 Comparisons of Optimization Techniques

Our proposed optimization techniques target a high-performance hardware architecture. Hence, we enhance the parallelism in our system which is a core feature in hardware designs. The inherent operations in the MAYO scheme suit this design paradigm well as they rely on multiple independent matrix multiplications. Our design parallelizes these computations over the parameters m and k, as discussed in Sec. 4.3.1 and 3.2, leading to a two-dimensional parallelization. This allows a performant overall computation of the MAYO operations. In addition, we solved arising issues caused by this parallelism, as explained in detail in Sec. 3.3 and 4.3.3.

Although the superior performance, this type of 2-dimensional parallelization has not been used in prior work. HaMAYO [24] does not report parallel computation on a set of matrix-vector multiplications or during the evaluation of the quadratic maps, which was explained in Sec. 2.2. On the other hand, the UOV scheme does not use multiple parallel matrices or the emulsification technique. Therefore, the work [7] cannot benefit from the same parallelization level. However, the Gaussian elimination operation in UOV and MAYO is similar. The work of [24] uses a serial computation while [7] uses a systolic array to perform the Gaussian elimination. In contrast to that, we reuse our parallelization technique over m to concurrently compute over a whole row of the matrix $\mathbf{A}$ to bring it to Echelon form. Thereby we reuse our GF16 ALUs to lower area consumption compared to the systolic array.

In addition to this, we feed our unrolled architecture with on-the-fly generated data during the expansion of the seed. This technique has not been used in HaMAYO and UOV works which reduces their performance and increases memory consumption. Moreover, we show how to partial loop unrolling the emulsification process which reduces the latency of signature generation and verification.

7.3 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 [7, 24], microcontroller [5, 7, 10] and high-end CPU [5] implementations of MAYO with our implementations on FPGA and ASIC. Further, we also included FPGA and microcontroller implementations of the UOV scheme [7] which uses similar construction and computations as the MAYO, e.g., using emulsifier maps to reduce the size of the signature. These similarities make it possible to present a comparison between [7] and our work. To the best of our knowledge, there are no ASIC implementation results for the MAYO schemes and we present the first according results.

Table 4: Throughput comparison with related works

	Works	Platform	Throughput per second			Speedup
			KeyGen	Sign	Verify	
$MAYO_1$	[5]^b	IXG 6338 @ 2GHz	18,182	4,348	11,429	-/-/-
	Our^e	A-28nm @ 1.5GHz	51,142	15,757	49,751	2.81/3.62/4.35
	Our^f	A-28nm @ 1.5GHz	258,665	48,188	327,011	14.22/11.08/28.61
$MAYO_3$	[5]^b	IXG 6338 @ 2GHz	3,973	1,205	3,279	-/-/-
	Our^e	A-28nm @ 1.5GHz	22,844	6,984	22,401	5.74/5.79/6.83
	Our^f	A-28nm @ 1.5GHz	127,877	22,840	166,021	32.18/18.95/50.63
$MAYO_5$	[5]^b	IXG 6338 @ 2GHz	1,653	483	1,695	-/-/-
	Our^e	A-28nm @ 1.5GHz	12,708	3,901	12,541	7.68/8.07/7.39
	Our^f	A-28nm @ 1.5GHz	75,388	13,424	101,950	45.60/27.79/60.14

A-28nm: ASIC with 28nm library. IXG 6338: Intel Xeon Gold 6338. ^{b}: Uses AVX2 with AES-NI. ^{e}: Uses 2×/3×/4× (AES128-R) for $MAYO_{1/3/5}$. ^{f}: Uses 2×/3×/4× (AES128-P) for $MAYO_{1/3/5}$.

Comparisons with high-end server CPU: The MAYO team provides a reference implementation as well as an optimized C version using Avx2 and AES-NI [22]. For comparison, we use this most optimized implementation on the high-end Intel Xeon Gold 6338 CPU (Ice Lake) with 2GHz [5], as shown in Table 4 and 5. We will first elaborate on the technology difference between the CPU and ASIC then we will analyze the throughput per second of both, CPU and ASIC platforms. The CPU used to collect the reference values runs on 2GHz and uses 10nm technology, whereas our ASIC implementation runs on 1.5GHz and uses 28nm technology from TSMC. Hence, the high-end CPU uses superior fabrication technologies. Nevertheless, our MAYO ASIC clearly outperforms the CPU, as discussed in the remainder of this section. In addition to that, our ASIC design includes all required SRAM memory (which consumes between 58% and 76% of die area) and can readily be integrated into larger products. This is another difference to the CPU requiring a significant amount of peripheral components such as DRAM. These components cause higher energy consumption and cost effort. An important aspect of high-speed architecture is the comparison of throughput per second. Table 4 shows the *throughput per second* of different security levels. We use the reference values of the optimized version from the MAYO team [5] that uses both Avx2 and AES-NI as a basis to illustrate our speedup. The reference implementation performs best in the case of security level 1, where it can perform either 18, 182 key generations, 4, 348 signature generations, or 11, 429 signature verifications per second. Yet, the *throughput per second* decreases by a factor of around 4 with each increase in the security level. Compared to that, we present two architectures with either AES128-R (round-based) or AES128-P (fully unrolled) seed expansion, as explained in Section 4.2. In comparison to [5], we achieve a speedup of 2.81×, 3.62×, and 4, 35× for AES128-R and 14.22×, 11.08×, and 28.61× for AES128-P in security level 1. Furthermore, our *throughput per second* decreases only by a factor of 2

Table 5: Comparison of Impl. results with related works

	Works	Platform	Latency (cc/ms)			Area (mm^2 or LUT/FF/DSP/BR)
			KeyGen	Sign	Verify	
UOV_1	[7][a]	Artix7 @ 90.8MHz[c]	11.0M/121.94	843k/9.29	284k/3.13	28k/24k/2/66
		Artix7 @ 90.3MHz[d]	11.0M/121.91	779k/8.63	115k/1.27	34k/27k/2/66
		AC-A72 @ 1.8GHz	28.3M/15.73	13.3M/7.40	2.20M/1.25	-
$MAYO_1$	[5]	AC-M4 @ 24MHz	5.24M/21.84	9.18M/38.25	4.88M/20.33	-
		IXG 6338 @ 2GHz[b]	110k/0.05	460k/0.23	175k/0.08	-
	[10]	AC-M7 @ 480MHz	-	42.9M/89.43	5.70M/11.88	-
	[24]	Artix7* @ 100MHz	996k/9.96	2.867M/28.67	-	21k/13k/11/129
	Our[e]	Artix7 @ 75MHz	29,330/0.40	95,191/1.28	30,150/0.40	92k/33k/2/45.5
		Kintex7 @ 100MHz	29,330/0.30	95,191/0.96	30,150/0.30	94k/32k/2/45.5
		A-28nm @ 1.5GHz	29,330/0.02	95,191/0.06	30,150/0.02	$1.02mm^2$
	Our[f]	Artix7 @ 75MHz	5,799/0.08	31,128/0.43	4,587/0.05	106k/38k/2/45.5
		Kintex7 @ 100MHz	5,799/0.06	31,128/0.32	4,587/0.04	112k/38k/2/45.5
		A-28nm @ 1.5GHz	5,799/0.004	31,128/0.02	4,587/0.003	$1.24mm^2$
UOV_3	[7][a]	Artix7 @ 96MHz[c]	16.7M/174.24	1.46M/15.26	823k/8.57	38k/29k/2/184.5
		Artix7 @ 94.1MHz[d]	16.4M/174.94	1.19M/12.75	195k/2.07	43k/35k/2/184.5
		AC-A72 @ 1.8GHz	56.8M/31.56	34.5M/19.18	8.31M/4.62	-
$MAYO_3$	[5]	IXG 6338 @ 2GHz[b]	508k/0.25	1.66M/0.83	610k/0.30	-
	Our[e]	Kintex7 @ 100MHz	65,660/0.65	214,751/2.15	66,959/0.67	147k/52k/2/96
		A-28nm @ 1.5GHz	65,660/0.04	214,751/0.14	66,959/0.04	$1.71mm^2$
	Our[f]	Kintex7 @ 100MHz	11,730/0.11	65,673/0.65	9,035/0.10	169k/60k/2/96
		A-28nm @ 1.5GHz	11,730/0.008	65,673/0.04	9,035/0.006	$1.97mm^2$
UOV_5	[7][a]	Artix7 @ 82.5MHz[c]	39.0M/437.53	3.30M/40.09	1.9M/23.29	77k/38k/2/356
		Artix7 @ 92.6MHz[d]	38.4M/414.73	2.64M/28.56	364k/3.93	83k/41k/2/359
		AC-A72 @ 1.8GHz	291M/161.91	86.7M/48.18	18.6M/10.33	-
$MAYO_5$	[5]	IXG 6338 @ 2GHz[b]	1.21M/0.60	4.14M/2.07	1.18M/0.59	-
	Our[e]	Kintex7 @ 100MHz	118,027/1.19	384,488/3.89	119,603/1.19	215k/73k/2/194.5
		A-28nm @ 1.5GHz	118,027/0.08	384,488/0.26	119,603/0.08	$3.09mm^2$
	Our[f]	Kintex7 @ 100MHz	19,897/0.20	111,736/1.12	14,713/0.14	246k/83k/2/194.5
		A-28nm @ 1.5GHz	19,897/0.01	111,736/0.07	14,713/0.01	$3.45mm^2$

A-28nm: ASIC with 28nm library. IXG 6338: Intel Xeon Gold 6338. AC-M4/M7/A72: ARM Cortex-M4/M7/A72. k and M are used as an abbreviation for $\times 10^3$ and $\times 10^6$, respectively.
*: Xilinx Zynq-7020 with Artix7 PL. [a]: Targets UOV scheme. [b]: Uses AVX2 with AES-NI.
[c]: ov-(I/III/V)s-pkc-skc (round-based AES128). [d]: ov-(I/III/V)s-pkc-skc (pipelined AES128).
[e]: Uses 2×/3×/4× (AES128-R) for $MAYO_{1/3/5}$. [f]: Uses 2×/3×/4× (AES128-P) for $MAYO_{1/3/5}$.

with each increase in the security level. This is a clear improvement compared to the factor of 4 in [5]. In the highest security level 5, our ASIC with AES128-P can perform 75388, 13424, and 101950 key generations, signature generations, and signature verifications per second, respectively. Compared to [5], which achieves 1653, 483, and 1695 operations per seconds, we reach a speedup of 45.60, 27.79, and 60.14, respectively. These results show that our proposed optimization techniques support the performant deployment of multi-multivariate signature schemes in time-critical applications like HSMs.

Comparisons with FPGA implementations for MAYO: To the best of our knowledge, there is only one FPGA implementation of the MAYO in the literature, HaMAYO [24], which is a reconfigurable and fault-tolerant hardware implementation of the MAYO scheme that uses the outdated parameters of the MAYO scheme. Note, that HaMAYO is only capable of performing key generation and signature generation operations for security level 1. In contrast to HaMAYO, we support all operations and security levels 1, 3, and 5 of the MAYO scheme. Compared to HaMAYO, our architecture with AES128-R requires 4.4×/2.5× more LUT/FF, but 5.5×/2.8× less DSP/BRAM. We achieve a speedup of 25× and 22× for key generation and signature generation compared to HaMAYO for security level 1 on Artix7 FPGA. In the case of our architecture with AES128-P, the speedup is even higher, namely 125× and 67× for key generation and signature generation. It is noteworthy to mention that the target of our architecture is high speed, while HaMAYO seems to be more targeted towards low consumption. To that end, we also provide a comparison for Area-Time-Product (ATP) using the formula ($ATP = (LUT + 100 \cdot DSP + 300 \cdot BRAM) \cdot Latency$) presented in [31]. Fig. 7 shows the ATP of our work, HaMAYO [24],

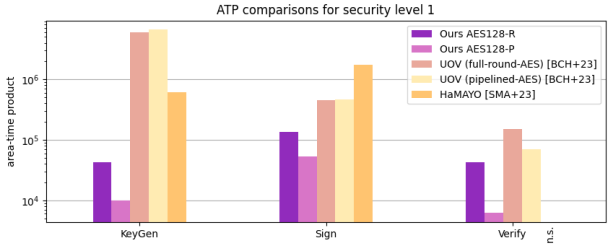


Figure 7: ATP comparison for FPGA implementations of MAYO/UOV security level 1. Signature verification is not supported in HaMAYO (n.s.).

and UOV [7], for each operation of the security level 1. Our architectures with AES128-R and AES128-P show 14×/61× and 13×/33× lower ATP, respectively, compared to the HaMAYO for key generation/signature generation operations of $MAYO_1$. One would argue that our speedup of one order of magnitude solely comes from the high area utilization of our architecture; however, the ATP comparison shows that this is not the case. Consuming a similar number of resources, other implementations will not match our speed, e.g., doubling the resources of HaMAYO may double its speed but cannot bridge the present speed gap. Note that the MAYO team updated the field operations of the scheme from GF(256) (8-bit) to GF(16) (4-bit) to enable faster computation as well as smaller keys. We already adopted this change within our design while HaMAYO operates on the previous GF(256) field. In addition, one major contribution of HaMAYO is their focus on low memory consumption, however, our high-speed architecture still consumes 3× less memory.

Comparisons with FPGA implementations for UOV: The work in [7] presents an implementation of the UOV scheme and provides area/performance results for all operations and security levels on Artix7 FPGA. [7] presents implementation results for different configurations such as arithmetic with GF(256) and GF(16) 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 GF(16) arithmetic that uses seed expansion with AES128-R and AES128-P since they best resemble our parameters and design methodology. We will use the results of Artix7 FPGA for security level 1 and Kintex7 FPGA for security levels 3 and 5. Our architectures with AES128-R outperform their comparable architecture for key generation/signature generation/signature verification by up to 305×/7×/8×, 270×/7×/13×, and 368×/10×/20× for security levels 1, 3, and 5, respectively. Our architectures with AES128-P outperform their comparable architecture by up to 1524×/20×/24×, 1535×/20×/22×, and 2095×/26×/27× for security levels 1, 3, and 5, respectively. This shows that we outperform them by up to three orders of magnitude for the key generation operation due to our massive parallelization of the matrix-matrix multiplications. However, our high parallelism comes at a price of resource utilization leading to 3.2×/4.1×/2.8× and 1.3×/1.9×/2.0× higher LUTs and FFs usages, respectively, for security levels 1/3/5 with the architecture using AES128-R. Similar results can be seen in the architecture that uses AES128-P. For signature generation and verification operations, our designs still outperform theirs by one order of magnitude. The decrease in speedup is because they utilize

Table 6: Power consumption on different Platforms

Platform	PRNG Type	$MAYO_1$	$MAYO_3$	$MAYO_5$
A-28nm @ 1.5GHz	AES128-R	2.94 W	3.95 W	6.99 W
A-28nm @ 1.5GHz	AES128-P	3.21 W	4.39 W	7.61 W
Kintex7 @ 100MHz	AES128-R	0.37 W	0.55 W	0.98 W
Kintex7 @ 100MHz	AES128-P	0.69 W	0.81 W	1,20 W

more memory to buffer temporary data in the key generation that can be reused during signing and verification. Yet, their design uses 1.6×/1.9×/1.8× 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 [24] and UOV [7] by up to three orders of magnitude in both speed and ATP comparison. In addition to performance improvement, our optimized memory management shows how to reduce the consumption of BRAMs for multivariate schemes on hardware.

Comparisons with ARM-based implementations: There are two ARM-based microcontroller implementations of MAYO scheme [5, 10] and both works target only security level 1 of the MAYO scheme. Compared to [5], our FPGA architecture on Kintex7 using AES128-R outperforms the key generation, signing, and verification operation by 55×, 30×, and 51×, respectively. In the case of AES128-P, our architecture outperforms [5] by 273×, 90×, and 381×, respectively. The implementation in [10] presents results only for signature generation and verification. Our FPGA design with AES128-R/AES128-P outperforms their performance by 93×/279× and 40×/297×, for signing and verifying, respectively.

7.4 Power consumption

Power consumption is a critical parameter in the design and optimization of hardware platforms. Table 6 shows our power consumption on both FPGA and ASIC for all configurations. The power consumption falls in the range of 0.37 W and 1.20 W on FPGA and between 2.94 W and 7.61 W on ASIC depending on the security level. The higher power consumption on ASIC is mainly due to the high frequency (15× higher) compared to FPGA, whereas the energy consumption per operation is lower in the ASIC case. Considering the energy consumption for $MAYO_1$ with AES128-R, our design requires 1.74mJ and 0.17mJ per signature computation for FPGA and ASIC respectively. A comparison of power consumption to related hardware accelerators is not possible since [7, 24] do not provide power results.

8 Discussion of Security Aspects

The MAYO team provides an in-depth mathematical security analysis of their cryptographic construction [5]. After presenting and evaluating our optimizations for a high-performance MAYO implementation, we now examine its *physical security* aspects that are particularly relevant to hardware platforms.

Secret-independent control flow: Secret-dependent branching can introduce variations in execution time that are exploitable through timing attacks. These attacks allow an attacker to deduce secret information based on the duration of specific operations.

By default, all other operations within MAYO's signing, except solving a linear equation, are constant-time. Therefore, our hardware design executes them with secret-independent timing. The signing operation involves solving a linear equation $\mathbf{Ax} = \mathbf{y}$ using Gaussian elimination. A straightforward implementation of Gaussian elimination is not constant-time due to the conditional swapping of rows during the pivot search. The authors of [6] address this issue by proposing a constant-time implementation of Gaussian elimination. Our work includes a similar approach for constant-time Gaussian elimination. The overhead introduced by the constant-time implementation is at most 3% of the signing latency.

On-the-Fly Data Generation: The baseline MAYO scheme uses AES-128-CTR with a public seed to expand the $\mathbf{P}_i$ matrices. Prior works follow this approach and store the expanded $\mathbf{P}_i$ matrices in memory before further computations. In contrast, our proposed on-the-fly generation does not store the expansion result in memory, but directly feeds the output of the AES-128-CTR to the GF16 ALUs. Therefore, we omit storing the data in memory, which does not open additional attack surfaces compared to non-optimized architectures. However, we acknowledge the possibility of fault injections to the seed expansion. Yet, fault attacks cannot gain additional advantages from our on-the-fly seed expansion optimization. Fault attacks would also be possible in all implementations unless countermeasures are incorporated.

Parallel Computation and Hardware Noise: One of the significant advantages of hardware implementations over software is the ability to perform parallel computations in a greater context. This shift from sequential or multi-threaded computation to parallel processing not only enhances performance but also makes certain advanced side-channel attacks, such as DPA, harder. In a parallel computation environment, the power profiles become more complex and less predictable, making it difficult for attackers to correlate power consumption with secret data. Therefore, it becomes more challenging to extract useful information related to secret data from the noisy observed data.

However, using parallelization as an argument for security is not always valid, as shown in surveys like [9, 18]. Furthermore, attackers can mitigate the higher noise level in parallel computing by collecting more power traces. Resistance against differential power analysis-based attacks is achievable by applying masking techniques. We leave the design and analysis of masking schemes for MAYO as a topic for future work.

9 Conclusion

In this paper, we proposed and implemented several optimization techniques for the MAYO post-quantum signature scheme for each NIST security level. On FPGA platforms we achieve a general speed-up of up to three orders of magnitude compared to similar works while reducing the memory consumption by up to 2.8×. In the context of throughput per second, our ASIC shows a speed-up of up to one order of magnitude compared to the most optimized C implementation on CPUs with instruction set extensions like AES-NI and Avx2. These results show that our proposed optimization techniques support the performant deployment of multi-multivariate signature schemes in a post-quantum world.

Artifact

We provide an artifact including two bitstreams ($MAYO_1$ with AES128-Rand AES128-P) for Genesys2 Kintex7 FPGA in [12]. It also includes the source code to interface with both architectures and a manual on how to set up and run the project.

Acknowledgments

This work was supported by the State Government of Styria, Austria – Department Zukunftsfonds Steiermark and the FWF ISOCrystals project. This work has benefitted from the fourth and fifth author's participation in Dagstuhl Seminar 23152 "Secure and Efficient Post-Quantum Cryptography in Hardware and Software". Finally, we want to thank the reviewers for their feedback, as well as Daniel Gruss from Graz University of Technology for his encouragement and feedback.

References

[1] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2012. Intel® Advanced Encryption Standard (Intel® AES) Instructions Set. https://www.intel.com/content/www/us/en/developer/articles/tool/intel-advanced-encryption-standard-aes-instructions-set.html

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

[3] Utsav Banerjee, Tenzin S. Ukyab, and Anantha P. Chandrakasan. 2019. Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 4 (Aug. 2019), 17–61. https://doi.org/10.13154/tches.v2019.i4.17-61

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

[5] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias Kannwischer. 2023. MAYO. MAYO Website. https://pqmayo.org/assets/specs/mayo.pdf. Accessed on August 3rd 2023..

[6] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. 2023. Nibbling MAYO: Optimized Implementations for AVX2 and Cortex-M4. Cryptology ePrint Archive, Paper 2023/1683. https://eprint.iacr.org/2023/1683 https://eprint.iacr.org/2023/1683.

[7] Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. 2023. Oil and Vinegar: Modern Parameters and Implementations. Cryptology ePrint Archive, Paper 2023/059. https://eprint.iacr.org/2023/059..

[8] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. 2022. Fast and Efficient Hardware Implementation of HQC. Cryptology ePrint Archive, Paper 2022/1183. https://eprint.iacr.org/2022/1183..

[9] Yiwen Gao, Yongbin Zhou, and Wei Cheng. 2018. How Does Strict Parallelism Affect Security? A Case Study on the Side-Channel Attacks against GPU-based Bitsliced AES Implementation. Cryptology ePrint Archive, Paper 2018/1080. https://eprint.iacr.org/2018/1080 https://eprint.iacr.org/2018/1080.

[10] Arianna Gringiani, Alessio Meneghetti, Edoardo Signorini, and Ruggero Susella. 2023. MAYO: Optimized Implementation with Revised Parameters for ARMv7-M. Cryptology ePrint Archive, Paper 2023/540. https://eprint.iacr.org/2023/540..

[11] Thales Group. 2024. Luna PCIe HSM. Thales Group documentation Website. https://cpl.thalesgroup.com/sites/default/files/content/product_briefs/field_document/2024-04/Thales-Luna-PCIe-HSM-pb.pdf - Accessed: 2024-04-29.

[12] Florian Hirner. 2024. Artifact. https://zenodo.org/records/12779188?preview=1&token=eyJhbGciOiJIUzUxMiJ9\let\begingroup\escapechar\m@ne\let\MT@subst@\T1/LinuxLibertineT-TLF/m/n/7\def{\@@par}.

[13] 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. 2022. SPHINCS+. Selected Algorithms 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022..

[14] Tendayi Kamucheka, Alexander Nelson, David Andrews, and Miaoqing Huang. 2022. A Masked Pure-Hardware Implementation of Kyber Cryptographic Algorithm. Cryptology ePrint Archive, Paper 2022/1547. https://eprint.iacr.org/2022/1547..

[15] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. 2024. Post-Quantum Signatures on RISC-V with Hardware Acceleration. *ACM Trans. Embed. Comput. Syst.* 23, 2 (2024). https://doi.org/10.1145/3579092

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

[17] Kris Gaj. 2020. Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using FPGAs. Research seminar for the NIST Post-Quantum Cryptography Standardization project. https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/seminars/oct-2020-gaj-kris-presentation.pdf..

[18] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. 2022. Microarchitectural Attacks in Heterogeneous Systems: A Survey. *ACM Comput. Surv.* 55, 7, Article 142 (dec 2022). https://doi.org/10.1145/3544102

[19] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. 2021. A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. *IEEE Access* 9 (2021), 150798–150808. https://doi.org/10.1109/ACCESS.2021.3126208

[20] Jacques Patarin. 1997. The Oil and Vinegar signature scheme. In *Dagstuhl Workshop on Cryptography*.

[21] Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. 2011. Small Public Keys and Fast Verification for Multivariate Quadratic Public Key Systems. In *Cryptographic Hardware and Embedded Systems – CHES 2011*. Springer, 475–490.

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

[23] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2022. FALCON. Selected Algorithms 2022.

[24] Oussama Sayari, Soundes Marzougui, Thomas Aulbach, Juliane Krämer, and Jean-Pierre Seifert. 2023. HaMAYO: A Reconfigurable Hardware Implementation of the Post-Quantum Signature Scheme MAYO. Cryptology ePrint Archive, Paper 2023/1135. https://eprint.iacr.org/2023/1135..

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

[26] P. W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (SFCS '94)*. IEEE Computer Society, Washington, DC, USA, 124–134. https://doi.org/10.1109/SFCS.1994.365700

[27] Abolfazl Soltani and Saeed Sharifian. 2015. An ultra-high throughput and fully pipelined implementation of AES algorithm on FPGA. *Microprocessors and Microsystems* 39, 7 (2015), 480–493. https://doi.org/10.1016/j.micpro.2015.07.005

[28] AMD Xilinx. 2012. FPGA Leadership Across Multiple Process Nodes. https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/kintex7-product-brief.pdf. Accessed: 2024-04-29.

[29] AMD Xilinx. 2021. AMD Xilinx 7 Series Product Selection Guid. https://docs.amd.com/v/u/en-US/7-series-product-selection-guide. Accessed: 2024-04-29.

[30] AMD Xilinx. 2021. FPGA Leadership Across Multiple Process Nodes. https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/artix7-product-brief.pdf. Accessed: 2024-04-29.

[31] Zewen Ye, Ray C. C. Cheung, and Kejie Huang. 2022. PipeNTT: A Pipelined Number Theoretic Transform Architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs* 69, 10 (2022), 4068–4072. https://doi.org/10.1109/TCSII.2022.3184703