HaMAYO: A Fault-Tolerant Reconfigurable Hardware Implementation of the MAYO Signature Scheme

Oussama Sayari[1], Soundes Marzougui[1,2], Thomas Aulbach[3], Juliane Krämer[3], and Jean-Pierre Seifert[1,4]

[1] Technical University of Berlin, Berlin, Germany
oussama_sayari@yahoo.fr, jean-pierre.seifert@tu-berlin.de
[2] STMicroelectronics, Diegem, Belgium
soundes.marzougui@st.com
[3] University of Regensburg, Regensburg, Germany
thomas.aulbach@ur.de, juliane.kraemer@ur.de
[4] Fraunhofer Institute SIT, Darmstadt, Germany

Abstract. MAYO is a topical modification of the established multivariate signature scheme UOV. Signer and Verifier locally enlarge the public key map, such that the dimension of the oil space and therefore, the parameter sizes in general, can be reduced. This significantly reduces the public key size while maintaining the appealing properties of UOV, like short signatures and fast verification. Therefore, MAYO is considered as an attractive candidate in the NIST call for additional digital signatures and might be an adequate solution for real-world deployment in resource-constrained devices.
When emerging to hardware implementation of multivariate schemes and specifically MAYO, different challenges are faced, namely resource utilization, which scales up with higher parameter sets. To accommodate this, we introduce a configurable hardware implementation designed for integration across various FPGA architectures. Our approach features adaptable configurations aligned with NIST-defined security levels and incorporates resources optimization modules. Our implementation is specifically tested on the Zynq ZedBoard with the Zynq-7020 SoC, with performance evaluations and comparisons made against previous hardware implementations of multivariate schemes.
Furthermore, we conducted a security analysis of the MAYO implementation highlighting potential physical attacks and implemented lightweight countermeasures.

Keywords: MAYO · Multivariate Cryptography · Post-Quantum Cryptography · Digital Signature · Hardware Implementation · Physical Security

1 Introduction

As quantum computing continues to advance, it is anticipated that quantum attacks can break many of the computational problems that classical cryptography

relies on, such as factorization and discrete logarithms used in RSA and ECDSA, respectively. To address this, researchers have proposed new mathematical assumptions and computational problems that are difficult to solve with quantum computers, resulting in the field of post-quantum cryptography. These new assumptions are grouped into different families, such as lattice-based, code-based, hash-based, and multivariate cryptography.

Multivariate schemes mainly rely on the difficulty of solving large systems of multivariate quadratic equations, known as the MQ Problem. As such, the signature scheme Rainbow [DS05] was a finalist in the third round of the NIST post-quantum cryptography (PQC) Standardization Process. Rainbow is a two-layered version of the UOV signature scheme [KPG99]. Hence, multivariate signature schemes based on the oil and vinegar principle received a lot of attention. They offer very short signatures and efficient verification, since the signature is mainly the solution to a system of multivariate quadratic equations, and verifying boils down to evaluating the polynomials at the presumed solution. Still, during the third round, Beullens developed an algebraic attack on Rainbow [Beu22a], targeting the layer structure that differentiates Rainbow from UOV. This led to the elimination of Rainbow from the ongoing process since it lost all its alleged advantages over the base scheme UOV.

Since mainly lattice-based signatures remained in the competition, NIST called for the submission of additional post-quantum digital signature schemes to enhance the given variety of signatures by prioritizing those that are not reliant on structured lattices, have short signatures and fast verification. The majority of the multivariate schemes submitted to this process are based on the oil and vinegar principle.

MAYO, introduced in [Beu22b], is one of them. It uses the same trapdoor - a secret oil space that is annihilated by the public key map - but is developed such that the signer and the verifier locally enlarge the public key matrices. Therefore, the dimension of the oil space can be reduced. That also allows to reduce other parameters like the number of variables in the quadratic equations since certain algebraic attacks get harder with a smaller oil space [KS06]. In total, this leads to significantly smaller public keys in MAYO, while keeping good performance numbers and signature sizes. For instance, with parameters targeting the first security level of the NIST process, the public key size of MAYO is 1,168 bytes, the secret key is 24 bytes, and the signature size is 321 bytes [BCC+23]. These results make the MAYO signature scheme even more compact than state-of-the-art lattice-based signature schemes such as Falcon and Dilithium [PQD23].

Contribution In this paper, we present an open source pure hardware implementation of the multivariate signature scheme MAYO. Our main target was a trade-off between SRAM/BRAM Consumption and FPGA Slides. In a second part, we investigated the physical security of MAYO implementation against side-channel analysis and fault-injection attacks. We, moreover, suggest lightweight countermeasures and implement them.

The contribution is summarized as follows:

- We manually settle a pure hardware implementation of MAYO. Our implementation is reconfigurable and can be easily integrated with different FPGA architectures and for different security levels.
- Certain functionalities used within key generation and signing are optimized, with a focus on low memory consumption.
- We present a new approach for the Gaussian solver and compare it to the well-known GSMITH approach of Rupp et al. in [REBG11].
- We considered threats emerging from possible fault injection and side channel analysis attacks, and cover them by employing low cost countermeasures.

The source code is available upon request.

Deployed Parameter Set When we started with the hardware implementation, there was only one proof of concept implementation available on https://github.com/WardBeullens/MAYO and it used the parameter set ($n = 62, m = 60, o = 6, k = 10, q = 31$) (see also [Beu22b, Section 8]). Thus, we also deployed these parameters in our work. In the meantime, the parameters were updated and as a main difference, MAYO also works over a field with even characteristic now, i.e., $q = 16$. This allows for higher efficiency and further implementation tricks, since now one field element occupies 4 bits instead of 5, and consequently, 2 field elements can be stored in one byte. The other parameters were also updated, but with minor impact. Thus, our work is one of the very few implementations of a multivariate schemes that utilizes a finite field with odd characteristic.

Related work At the time of writing this paper, there is a scarcity of complete hardware designs for post-quantum cryptographic schemes [ZZW+21, XL21, FG18, HZ18]. However, given that the NIST PQC reached the fourth round and started the call for additional digital signature schemes, it is expected that more dedicated hardware designs will emerge. These designs would be instrumental in showcasing the strength and inherent properties of specific protocols [NIS23a].

Multivariate schemes necessitate the development of comprehensive and extensive implementation designs to address the challenging gaps due to the schemes' large key sizes [DS05, KPG99]. These key sizes often pose challenges for devices with limited resources, as they may struggle to accommodate the storage requirements of these schemes. Moreover, multivariate schemes commonly involve memory and time-consuming blocks, with the Gaussian solver being a well-known performance bottleneck [REBG11]. Despite the above-mentioned challenges, there have been a few published hardware implementations that have reported results for multivariate schemes [TYD+11, HZ18, FG18].

In [FG18], Ferozpuri and Gaj present a high-speed FPGA implementation of Rainbow. Their hardware implementation uses a parameterized system solver where the execution time is proportional to the system dimension, i.e., it can solve an n-by-n system in n clock cycles. Moreover, their work reduces the number of required multipliers by almost half, speeds up execution as compared to

the previous state-of-the-art work, and implements Rainbow for higher security levels.

In [TYD+11], Tang et al. present another high-speed hardware implementation of Rainbow. The authors targeted similar functionalities for optimization as in [FG18], i.e., the Gaussian solver and the multipliers. They developed a new parallel hardware design for the Gaussian elimination and designed a novel multiplier to speed up the multiplication of three elements over a finite field. With Rainbow being broken [Beu22a], all its previously published software and hardware implementations needs to be revised and transferred to secure schemes for practical use. To address this issue, MAYO is seen as a viable alternative, showcasing improved performance results.

Simultaneous work During the preparation of this paper, hardware implementations of UOV [BCH+23] and MAYO [HSMR23] were published in 2023. The latter already features the updated parameter set of MAYO ($n = 66, m = 64, o = 8, k = 9, q = 16$), where m is chosen to be a multiple of 32 and q is a power of 2 to facilitate further implementation optimizations.

2 Preliminaries

The MAYO signature scheme [Beu22b] is a special modification of the UOV signature scheme [KPG99] and belongs to the field of multivariate cryptography. Herein, the main object is the multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ with m components and n variables. In more detail, it is a sequence $p_1(\mathbf{x}), \ldots, p_m(\mathbf{x})$ of m quadratic polynomials in n variables $\mathbf{x} = (x_1, \ldots, x_n)$, with coefficients in a finite field $\mathbb{F}_q$. Very abbreviated, multivariate cryptography is based on the hardness of finding a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of a target vector $\mathbf{t} \in \mathbb{F}_q^m$ under a given multivariate quadratic map $\mathcal{P}$, i.e., solving a multivariate system of quadratic equations. This task is often referred to as the MQ problem. One way that allows the signer to compute a signature $\mathbf{s}$ is to install a secret trapdoor into the public map $\mathcal{P}$.

2.1 The Trapdoor in UOV

In UOV, the trapdoor information is a basis of a secret linear subspace $\mathcal{O} \subset \mathbb{F}_q^n$ of dimension $\dim(\mathcal{O}) = m$, the so-called oil space [Beu21]. The multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ is then chosen in a way that it vanishes on this oil space, i.e., $\mathcal{P}(\mathbf{o}) = \mathbf{0}_m$ for all $\mathbf{o} \in \mathcal{O}$. For the multivariate quadratic polynomials $p_i(\mathbf{x})$, which constitute the map $\mathcal{P}$ via $\mathcal{P}(\mathbf{x}) = p_1(\mathbf{x}), \ldots, p_m(\mathbf{x})$, one can define their *polar form* or *differential* as

$$p_i'(\mathbf{x}, \mathbf{y}) \coloneqq p_i(\mathbf{x} + \mathbf{y}) - p_i(\mathbf{x}) - p_i(\mathbf{y}) + p_i(\mathbf{0}).$$

Since we commonly work with homogeneous polynomials, the term $p_i(\mathbf{0})$ will be omitted in the following. Similarly, we can define the polar form of $\mathcal{P}$ as

$$\mathcal{P}'(\mathbf{x}, \mathbf{y}) = p_1'(\mathbf{x}, \mathbf{y}), \ldots, p_m'(\mathbf{x}, \mathbf{y}).$$

As shown in [Beu21, Theorem 1], the map $\mathcal{P}' : \mathbb{F}_q^n \times \mathbb{F}_q^n \to \mathbb{F}_q^m$ is a symmetric and bilinear map. Furthermore, if one has knowledge of the secret oil space, it can be used to efficiently find preimages $\mathbf{x} \in \mathbb{F}_q^n$ of a given target $\mathbf{t} \in \mathbb{F}_q^m$ such that $\mathcal{P}(\mathbf{x}) = \mathbf{t}$. To do so, one can randomly pick a vinegar vector $\mathbf{v} \in \mathbb{F}_q^n$ and solve the system $P(\mathbf{v} + \mathbf{o}) = \mathbf{t}$ for $\mathbf{o} \in \mathcal{O}$. This is possible since in

$$\mathbf{t} = \mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathcal{P}(\mathbf{v}) + \mathcal{P}(\mathbf{o}) + \mathcal{P}'(\mathbf{v}, \mathbf{o}) \tag{1}$$

the term $\mathcal{P}(\mathbf{v})$ is constant and $\mathcal{P}(\mathbf{o})$ vanishes, so whenever the linear map $\mathcal{P}'(\mathbf{v}, \cdot)$ is non-singular, the system has a unique solution $\mathbf{o} \in \mathcal{O}$, which can be computed efficiently. This happens with probability roughly $\frac{q-1}{q}$. If this is not the case, one can simply pick a new value for $\mathbf{v}$ and try again. Without a description of the oil space $\mathcal{O}$, the term $\mathcal{P}(\mathbf{o})$ implies that Equation 1 constitutes a system of quadratic equations, which remains hard to solve.

Building a signature scheme directly from this setting has one big disadvantage. The oil space needs to be as large as the image space of the multivariate quadratic map $\mathcal{P}$, i.e., $\dim \mathcal{O} = m$. To counter the Kipnis-Shamir attack [KS06], the parameter n needs to be sufficiently larger than m, with $n \approx 2,5m$ being used in all currently considered implementations. The parameter m itself needs to be of a certain size as well, to provide security against direct attacks or the intersection attack [Beu21]. This leads to key pairs of enormous size, which is considered the main drawback of multivariate signatures. Recently, Beullens developed the signature scheme MAYO to tackle this problem.

2.2 Description of MAYO

The essential modification is the downsizing of the dimension of the oil space to $\dim \mathcal{O} = o < m$. Actually, this oil space is now too small to sample signatures, since the system $\mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathbf{t}$ given in Equation 1 consists consequently of m linear equations in o variables and is unlikely to have any solutions. Thus, the approach taken in [Beu22b] is to stretch the public key map into a larger whipped map $\mathcal{P}^* : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$, such that it accepts k input vectors $\mathbf{x} \in \mathbb{F}_q^n$. This is realized by defining

$$\mathcal{P}^*(\mathbf{x}_1, ..., \mathbf{x}_k) := \sum_{i=1}^{k} \mathbf{E}_{ii}\mathcal{P}(\mathbf{x}_i) + \sum_{1 \le i < j \le k} \mathbf{E}_{ij}(\mathcal{P}'(\mathbf{x}_i, \mathbf{x}_j)), \tag{2}$$

where the matrices $\mathbf{E}_{ij} \in \mathbb{F}_q^{m \times m}$ are fixed system parameters with the property that all their non-trivial linear combinations have rank m.

It is easy to see that $\mathcal{P}^*$ vanishes on the subspace $\mathcal{O}^k = \{(\mathbf{o}_1, \ldots, \mathbf{o}_k) | \text{ with } \mathbf{o}_i \in \mathcal{O} \text{ for all } i \in [k]\}$ of dimension ko. By choosing the parameters such that $ko \ge m$, the k copies of the oil space are large enough to construct preimages of a target vector $\mathbf{t} \in \mathbb{F}_q^m$ under the whipped map $\mathcal{P}^*$. In more detail, the signer randomly samples $(\mathbf{v}_1, \ldots, \mathbf{v}_k) \in \mathbb{F}_q^{kn}$, and then solves

$$\mathcal{P}^*(\mathbf{v}_1 + \mathbf{o}_1, ..., \mathbf{v}_k + \mathbf{o}_k) = \mathbf{t} \tag{3}$$

for $(\mathbf{o}_1, ...\mathbf{o}_k) \in \mathcal{O}^k$. Observe from Equation 2 that this system remains linear in the presence of the linear emulsifier maps $\mathbf{E}_{ij} \in \mathbb{F}_q^{m \times m}$. Thus, the signer can efficiently compute a preimage $\{\mathbf{s}_i = \mathbf{v}_i + \mathbf{o}_i\}_{i \in [k]}$ of $\mathbf{t}$. Similar to UOV, the verifier just needs to check if the given $\{\mathbf{s}_i\}_{i \in [k]}$ satisfy Equation 3.

Remark 1. Please note that both, the signer and the verifier, only locally whip up the public key map $\mathcal{P}$ to $\mathcal{P}^*$, so this modification comes with no additional cost in terms of key sizes. However, it entails additional computations during signing and verification. Furthermore, it increases signature size, since now a k-tuple of vectors in $\mathbb{F}_q^n$ constitute the signature. These negative effects are cushioned by the ability to reduce parameter sizes while maintaining the security level.

2.3 The implemented MAYO functionalities

The above descriptions remain rather high-level and abstract. Here we show more details about the main functionalities that need to be implemented, e.g., evaluations of (parts of) the public key map $\mathcal{P}$ via vector-matrix multiplications and finding solutions to the generated linear system via Gaussian elimination. Due to the page limit we do not present all the algorithms we implemented here, but refer to the MAYO specification [BCC+23, Section 2], specifically to the algorithms *MAYO.CompactKeyGen()*, *MAYO.ExpandSK(csk)* and *MAYO.Sign(esk,M)*. The latter will also play a major role in our security discussion in Section 4, so it is presented in Algorithm 1 below. The first few lines are used to sort the bit string of the expanded secret key to the respective matrices (line 1-5) and to derive a target vector $\mathbf{t} \in \mathbb{F}_q^m$ and salt (line 7-11). The main part of the signing process can be described by generating random variables (line 15-19), inserting the vinegar variables $\mathbf{v}_i$ into $\mathcal{P}$ to set up a linear system (line 21-35), solving the system (line 37-40) and adding the solution to the vinegar variables (line 42-45).

3 Hardware Design

In this section, we present the hardware design of our implementation. Our primary goal is to provide a reconfigurable hardware code that can be easily integrated with different FPGA architectures and for different security levels.

Although MAYO has keys of reduced size compared to other multivariate alternatives, it still necessities a large amount of internal memory to execute the key-generation and signing phase [Beu22b] in the order of several dozen KB. This is partially attributed to the fact that the keys are stored as seeds. During the signing the seed is expanded into large matrices, e.g., for the parameter set $(n, m, o, k, q) = (66, 64, 8, 9, 16)$, the public key of 1168B is expanded into 70KB.

For implementation and testing of our hardware design, we opted for the target board Zynq ZedBoard with the Zynq-7020 SoC [Xil23], which has 85K Logic Cells and 4.9MB Block RAM serving as an upper bound for the memory consumption.

Algorithm 1 MAYO.Sign(esk,M) [BCC+23]

Input: Expanded secret key esk $\in \mathcal{B}^{\text{esk bytes}}$, Message $M \in \mathcal{B}^*$
Output: Signature sig $\in \mathcal{B}^{\text{sig bytes}}$

// Decode esk
$\text{seed}_{sk} \leftarrow \text{esk}[0 : \texttt{sk\_seed\_bytes}]$
$\mathbf{O} \leftarrow \texttt{Decode}_{\mathbf{O}}(\text{esk}[\texttt{sk\_seed\_bytes} : \texttt{sk\_seed\_bytes} + \mathbf{O}\texttt{\_bytes}])$
$\{\mathbf{P}_i^{(1)}\}_{i\in[m]} \leftarrow \texttt{Decode}_{P^{(1)}}(\text{esk}[\texttt{sk\_seed\_bytes} + \mathbf{O}\texttt{\_bytes}] : \texttt{sk\_seed\_bytes} + \mathbf{O}\texttt{\_bytes}] + \texttt{P1\_bytes}])$
$\{\mathbf{L}_i\}_{i\in[m]} \leftarrow \texttt{Decode}_{L}(\text{esk}[\texttt{sk\_seed\_bytes} + \mathbf{O}\texttt{\_bytes}] + \texttt{P1\_bytes} : \texttt{esk\_bytes}])$

// Hash message and derive salt and $\mathbf{t}$
$M_{digest} \leftarrow \texttt{SHAKE256}(M, \texttt{digest\_bytes})$
$\text{R} \leftarrow \mathbf{0}_{R_{bytes}}$
$\text{salt} \leftarrow \texttt{SHAKE256}(M_{digest} \,\|\, R \,\|\, \text{seed}_{sk}, \texttt{salt\_bytes})$
$\mathbf{t} \leftarrow \texttt{Decode}_{vec}(m, \texttt{SHAKE256}(M_{digest} \,\|\, \text{salt}, \lceil (m \log(q))/8 \rceil))$

// Attempt to find a preimage for $\mathbf{t}$
for ctr from 0 to 255 **do**
    # Derive $\mathbf{v}_i$ and r
    $V \leftarrow \texttt{SHAKE256}(M_{digest} \,\|\, \text{salt} \,\|\, \text{seed}_{sk} \,\|\, \text{ctr}, k \cdot v_{bytes} + \lceil ko \log(q)/8 \rceil)$
    for i from 0 to $k-1$ **do**
        $\mathbf{v}_i \leftarrow \texttt{Decode}_{vec}(n-o, V[i \cdot v_{bytes} : (i+1) \cdot v_{bytes}])$
    $\mathbf{r} \leftarrow \texttt{Decode}_{vec}(ko, V[k \cdot v_{bytes} : k \cdot v_{bytes} + \lceil ko \log(q)/8 \rceil])$

    // Build linear system $Ax = y$.
    $\mathbf{A} \leftarrow \mathbf{0}_{m \times ko} \in \mathbb{F}_q^{m \times ko}$
    $\mathbf{y} \leftarrow \mathbf{t}$, $\ell \leftarrow 0$
    for i from 0 to $k-1$ **do**
        $\mathbf{M}_i \leftarrow \mathbf{0}_{m \times o} \in \mathbb{F}_q^{m \times o}$
        for j from 0 to $m-1$ **do**
            $\mathbf{M}_i[j, :] \leftarrow \mathbf{v}_i^\intercal \mathbf{L}_j$
        for j from $k-1$ to i **do**
            $\mathbf{u} \leftarrow \{\mathbf{v}_i^\intercal \mathbf{P}_a^{(1)} \mathbf{v}_i\}_{a\in[m]}$ **if** $i = j$
            $\mathbf{u} \leftarrow \{\mathbf{v}_i^\intercal \mathbf{P}_a^{(1)} \mathbf{v}_j + \mathbf{v}_j^\intercal \mathbf{P}_a \mathbf{v}_i\}_{a\in[m]}$ **if** $i \neq j$
            $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{E}^\ell \mathbf{u}$
            $\mathbf{A}[:, i \cdot o : (i+1) \cdot o] \leftarrow \mathbf{A}[:, i \cdot o : (i+1) \cdot o] + \mathbf{E}^\ell \mathbf{M}_j$
            if $i \neq j$ **then**
                $\mathbf{A}[:, j \cdot o : (j+1) \cdot o] \leftarrow \mathbf{A}[:, j \cdot o : (j+1) \cdot o] + \mathbf{E}^\ell \mathbf{M}_i$
            $\ell \leftarrow \ell + 1$

    // Try to solve the system
    $\mathbf{x} \leftarrow \texttt{SampleSolution}(\mathbf{A}, \mathbf{y}, \mathbf{r})$
    if $\mathbf{x} \neq \perp$ **then**
        break

// Finish and output the signature
$\mathbf{s} \leftarrow \mathbf{0}_{kn}$
for i from 0 to $k-1$ **do**
    $\mathbf{s}[i \cdot n : (i+1) \cdot n] \leftarrow (\mathbf{v}_i + \mathbf{O}\mathbf{x}[i \cdot o : (i+1) \cdot o] \,\|\, \mathbf{x}[i \cdot o : (i+1) \cdot o])$
return $\text{sig} = \texttt{Decode}_{vec}(s) \,\|\, \text{salt}$

The majority of the system architecture of our hardware design is described in VHDL, while a few modules are implemented using Verilog.

It is essential for the architecture to be encapsulated as an Intellectual Property (IP), to ensure design reuse. We developed Keygen and Sign IPs intended for use on an end-user device in diverse applications such as the authentication of bank transactions. It remains paramount that these two IPs guarantee compliance with the device's memory constraints, especially regarding time and memory utilization. In contrast, we expect that the verification process takes place within an environment boasting ample resources such as a dedicated server, where security measures are not as critical as those required for IPs operating directly on confidential data, i.e., Keygen and Sign.

It is possible to utilize one of the IPs on the target chip. Both cores are independent and capable of coexisting on the Programmable Logic operating at respectable frequencies.

The CPU-Peripheral communications between the built IPs are handled through AXI4-FULL, AXI-Lite, and interrupts. The provided firmware takes care of the AXI transactions, thanks to the Zynq hybrid architecture. Incidentally, the design focuses on maintaining high transfer bit-rates by extensively leveraging the CPU's 32-bit architecture. Frequencies and reset signals are also controlled by the hardcore and are propagated throughout the design.

Based on the proposed MAYO pseudo-code in [BCC+23], the scheme incorporates multiple helper functions that are implemented as sub-modules and arithmetic units within the hardware IPs. This approach fulfills another significant design requirement by minimizing unused module and minimizing the utilization of Flip-Flops (FFs) and Lookup Tables (LUTs). By avoiding code duplication in hardware and organizing the design into smaller, specialized modules, each capable of performing a single functionality, the overall efficiency and modularity of the design are improved.

Considering the scheme's parameter set, the memory is divided into *three* True Dual Port BRAMs, statically partitioned into 2×256KB BRAMs to store big matrices and large vectors like the $\mathcal{P}$ system and $\mathcal{O}^k$ subspaces, and 1×4KB BRAM designated for small scratch buffers and sensitive information such as the seed, signature, and secret key. Among these BRAMs, only one of the big BRAMs is exposed to CPU through the AXI bus. Detailed memory management and utilization is deliberated later in Section 3.4. As shown in Figure 1, most modules are connected to the BRAMs accordingly.

3.1 Hash Function

Our design employs the Keccak core [BDH+22] to generate seeds and expand the message as a first step of the signing process. For the first security level, SHAKE128 was used as an extendable-output function (XOF) based on the FIPS 202 standard [NIS23c]. We note that for higher security levels, it is necessary to adjust the parameters within the Keccak core accordingly. Nonetheless, the fundamental design of the hash sub-module remains applicable and does not require significant changes.

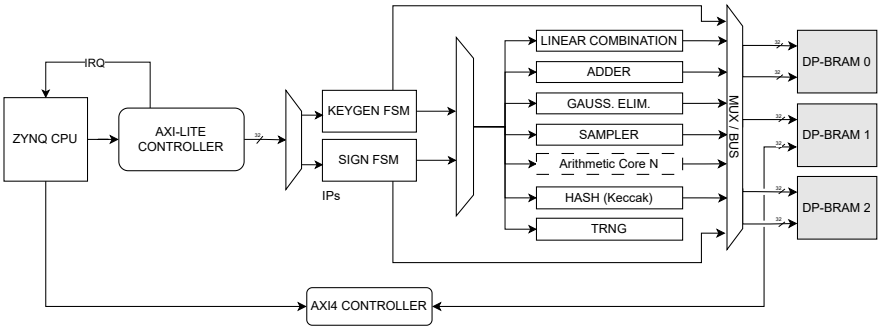

Figure 1: Block Diagram of the MAYO Core

The Keccak implementation in [BDH+22] streams data utilizing a different format compared to the proposed MAYO hardware 32-bit format. To address this discrepancy, we developed a wrapper around the core. The reasoning behind this is that MAYO algorithm requires a hash of approximately 120KB for the key generation. The hash is eventually stored in the inner 32-bit-wide block memory.

The proposed architecture stores the input seed and output message in separate descriptor-like registers. These intermediate registers are simultaneously accessed by the hash core and BRAM. The core itself takes care of BRAM communication and indexing, simplifying the architecture's state change and its modularity.

3.2 Random Number Generator

The random number generator leverages AES-128 in CTR Cipher mode, with the flexibility to seamlessly switch to AES-256 if necessary. Tinkering with key parameters like seed and counter interval (PRNG-Based) is effortlessly accomplished within the core. To optimize FPGA Slice utilization, the core's decryption functionalities have been deprecated, given the inherent independence of CTR-mode from such operations.

3.3 Vector-Matrix Multiplication

Referring to Section 2.2, it is evident that matrix-vector multiplication proceeded by a $\mathbb{F}_q$ space reduction, is a frequently utilized operation throughout the algorithm. Hence, its optimization will improve the performance of our design.

Compared to the initial MAYO Software C implementation[5], the vector-matrix multiplication iterates through a matrix stored in a row-wise manner, as seen in the left side of Figure 2, multiplying (using MULT operation) the content with a given series of coefficients and accumulating the results. Once this nested row/column loop concludes, another loop starts reducing the accumulated result

[5] Note here that we refer to the first implementation of MAYO scheme by Ward Beullens in [Beu22b]

through MOD operation. For instance, on an ARM Cortex-M3 with ARMv7-M instruction set, a single MULT operation with 8-bit operands takes around 2 to 3 clock cycles [ARM]. The reduction is done using the MOD operation that is usually translated to MULT and UDIV as Cortex-M3 lacks native modulo calculation. Consequently, the vector-matrix multiplication function could consume up to 6500 clock cycles, excluding the memory load and store operations.

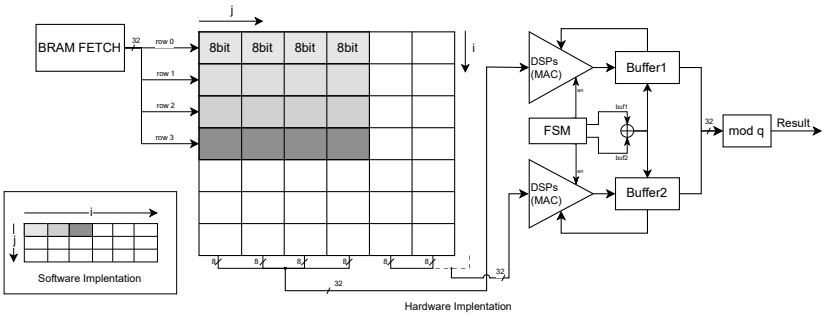

Figure 2: Matrix-Vector multiplication architecture; on the left side the vector-matrix multiplication iterates through a matrix stored in a row-wise manner as in the software implementation. In hardware design, we reversed the indexing order, and input four bytes to each DSP which executes 4 multiplications simultaneously.

In this paper, we process the multiplications differently. Firstly, our design offers four values on each memory read operation thanks to its 32-bit wide bus and executes 4 MULT operations from one row simultaneously. Secondly, we reversed the indexing of the input matrix, as shown in Figure 2.

As matrices are stored row-wise, each memory access returns $four$ sequential cells from *one* row. Note that the matrix is stored in BRAMs and not in an FF-layered structure.

Furthermore, the input of both Digital Signal Processors (DSPs) is composed of 4 bytes. This architecture helps increase the throughput and enables the parallelization of both MULT and MOD operations.

Once the accumulated data of a block of four columns begins the final MOD operation, the subsequent block is fetched and starts with MULT operation. The first row of the Matrix $\mathbf{M}$ and the first coefficient of the Vector $\mathbf{V}$ are fetched from the BRAMs. The read port then keeps feeding the system with blocks from each consequent row noted as $\mathbf{M}[rowIndex, columnBlock]$, until the accumulated result is ready to be stored through a different write-only port (WriteRES).

3.4 Memory Organization

The hardware implementation of MAYO mainly relies on BRAMs to store its vectors and matrices. To ensure that both cores, namely the KeyGen and Sign, have sufficient stack-like memory, 82% (4.03 Mb) of the available on-chip BRAM is allocated for the implementation. Thereby we provide enough headroom for potential parameter modification of MAYO that might increase memory usage, e.g., when changing the security level from 1 to 5, the expanded secret key size increases from 70KB to 557KB [BCC+23].

The design aligns itself with the 32-bit ARM multi-core processor architecture and uses a 32-bit data bus width. This approach simplifies data processing within each sub-module. In the case of MAYO, the values are usually stored in a 5 bits-wide reduced space. For the NIST security level 1, the scheme operates on values that are eventually reduced to $\mathbb{F}_q$, meaning that the results must be less or equal to $q = 31$. To store such numbers in the BRAM, $5 = \lceil \log_2(31) \rceil$ bits are mandatory. As a result, the design allocates 8 bits of memory (i.e., unsigned char) for each numerical unit. We, then, exploit the 32-bit architecture in various pipeline techniques by processing simultaneously four 8-bit values.

It is important to note that our implementation adapts the parameter set $(n = 62, m = 60, o = 6, k = 10, q = 31)$ and resulted in a public key and signatures have a size of 803B and 420B respectively. However for the NIST first level the parameters are $(66, 64, 8, 9, 16)$ and result in public key and signature size of 926B and 387B.
There exist different variants of the MAYO first security level where the public key size is increased at the expense of smaller signature. Precisely, these variants increase the n which is the number of variables in the multivariate quadratic polynomials in the public key at the expense of decreasing k which is the whipping parameter. This results in bigger public key size and smaller signature size as the whipping parameters are directly connected to the calculation of the signature.
In addition, the q does not have significant impact on the sizes of the public key and the signature itself but more on the stack-like memory during the key generation and the singing processes. On the other hand, if $q = 16$, one byte can be used to pack two elements as all elements are in $\mathbf{F}_{16}$. However, this is not the case for our implementation.

It is important to note, that not *all* the allocated memory is utilized for the first security level. In fact, only roughly 70% (2.8 Mb) of the allocated BRAM of the Zynq device is filled with data. The rest is left empty, but deemed necessary due to ARM's 32-bit memory alignment rules. The content of the BRAM cells is pre-allocated and statically organized since the sizes of most elements are pre-defined. In other words, all vectors and matrices' addresses are provided in a VHDL file to create a mapping. This file is then included in all sub-modules for better consistency. To eliminate dependency on vendor-specific SDKs, a set of Python scripts takes charge of memory template generation. These scripts meticulously analyze the VHDL file, dynamically determining the required depth of BRAMs. This approach not only fosters platform independence

but also enhances adaptability by allowing seamless adjustments to memory configurations based on the specifics of the VHDL code. The result is a more agile and versatile solution for memory management within the FPGA design, especially for various parameter sets.

The memory is partitioned into *three* dual port BRAMs, offering enhanced performance and flexibility. This configuration allows, for instance, efficient reading from one port while dedicating the other port for writing. Some sub-modules, such as vector-matrix multiplication, or vector addition, tend to utilize three ports for dual read and final write operations, therefore allowing better opportunities for parallelism within the sub-module.

Small buffers and vectors that are not meant to be accessed exclusively by programmable hardware are found in the smaller BRAM. The big BRAMs are indeed also shared with the CPU through AXI bus to stream input information such as the message and secret key to the MAYO core itself. Furthermore, since the key generation and the signing are not designed to operate synchronously but rather consecutively, multiple arrays and vector spaces overlap if one of their lifetimes expires. This approach helps the system avoid unnecessary increases in memory fingerprints.

3.5 Gaussian Elimination

Solving a System of Linear Equations (SLE) is evidently one of the primordial computations for the MAYO algorithm to generate a valid message signature as explained in Section 2.2. Several publications deal with hardware implementation of Gaussian elimination for various cryptographic applications, primarily focusing on $\mathbb{F}_2$. Among them, GSMITH [REBG11] has been widely recognized for efficiently handling $\mathbb{F}_{2^k}$ equations. Unfortunately, GSMITH's architecture only conforms with small and medium-sized matrices, whereas MAYO's SLE $m \times m$ shaped matrix is larger. This quadratic shape depends on the NIST security level. Not only would the proposed GSMITH architecture utilize costly resources, but also hinder the overall architecture's performance and increase the needed Look-up Tables (LUTs) when targeting $\mathbb{F}_{31}$. GSMITH describes, in fact, a systolic network composed of various types of tiny processors capable of specific Gaussian steps and propagating its values. Yet, since the source code was not open-sourced, we had to redesign GSMITH. The final architecture, however, fails to meet our resource requirements, depleting the Zynq's FFs and LUTs, due to the internal registers required in each GSMITH processor and its interconnection with the proposed BRAM. When considering the other needed arithmetic cores, we concluded it was unfeasible to fit GSMITH for the first security level.

To overcome this issue, we developed a state machine that fetches values directly from BRAM as the matrix is stored externally rather than within the core's FFs. Additionally, it was mandatory to allocate sufficient memory to accumulate every cell in the matrix. In other words, during the first step of the Gaussian elimination, multiplying rows with scalars may surpass the existing 8-bit limit. Hence, the targeted matrix is initially unpacked into 16-bit wide values

with added padding, meaning that every row in the BRAM now contains two instead of four values.

Moreover, to speed up the mod-inverse, which calculates the needed value to transform the pivot element into 1 throughout the first scale step, prefilled Read-Only Memory (ROM) with end results of this operation is utilized instead of performing the actual calculations on run-time. These optimizations contribute to the overall effectiveness of the MAYO core in solving an SLE. Although GSMITH might offer superior performance, this core certainly consumes less memory resources. Our architecture is theoretically compatible with other configuration sets, with a marginal difference in resource utilization. For instance, n, m, o control the SLE size which should affect BRAM consumption, while q modifies the LUT consumption, cell width, and the unpacking operation. The solver should support up to $\mathbb{F}_{2^8}$ and for smaller q values, unpacking the matrix might become unnecessary, as the result could still fit inside the original 8-bit vector.

3.6 Optimizations and Firmware

Besides resource utilization, the goal of our design is to achieve a reasonable time area trade-off. Therefore, we designed the sub-modules in a way that they share the same access to one of the BRAM ports. Nevertheless, the usage of each port, whether for reading, writing, or both differs. The core responsible for the vectors addition, for example, features multiple modes depending on the location of the input vectors in different BRAMs. It efficiently utilizes all available ports to leverage data throughput and synchronize the addition process accordingly.

Another notable design optimization lies in the polynomial reduction submodule where multiple arrays of scratch buffers are used to minimize memory interactions. Hence, the core is provided only with new values which are stored as final results.

Various functionalities of MAYO are divided into separate modules, each described individually. That said, each module still has access to header-like files that declare the security level parameters, the memory space allocations, utility functions required to fetch offsets or even ROM secret keys specifically intended for non-debugging purposes. Numerous bit vectors are built upon these constants. The code's style guide itself heavily discourages simple number inclusion, but instead, it is expected to utilize these pre-defined macro-like lines to improve code readability and ensure that the overall architecture can fit different configuration sets, i.e., different security levels.

In addition to the hardware implementation, the utilization of the MAYO core necessitates the development of accompanying firmware. This firmware serves as the interface between the hardware core and the software MAYO application, setting AXI/AXI-Lite transactions up. The existing C Bit-fields feature Control and Status Registers that can enable debug mode, interrupts, and supply the ARM CPU with the end of executions information besides the interrupt signal.

4 Mitigation of Physical Attack Vectors in MAYO

In Section 2.2, we stated that the secret key is solely given by the secret linear oil space O. Thus, an attacker is able to forge signatures, as soon as she recovered O. Even more, the description of the reconciliation attack in [Beu22b, Section 4.1] shows that it is enough to know a single vector $o_1 \in O$, to recover the remaining space O in polynomial time, since the first vector o_1 implies m linear equations via $\mathcal{P}'(o_1, o_2)$ on the entries of o_2. Consequently, we need to solve m quadratic equations in $n-m-o$ variables. Since in MAYO $n < m+o$ holds, the remaining basis vectors of O can be obtained just by solving linear equations.

Moreover, the randomly generated vinegar variables can also be used to recover the secret key. Recall, that a given MAYO signature has the form $s = (s_1, \ldots, s_k) = (v_1 + o_1, \ldots, v_k + o_k)$, so the knowledge of one of the v_i's together with the corresponding s_i leads the attacker to a vector of the oil space and thus, to the full secret key.

In the following, we show different scenarios where the attacker uses fault injection or side-channel attacks to reveal either a vinegar or an oil vector.

4.1 Fault Injection

The attacks suggested in the following are first-order fault injection attacks and assume an attacker to be able to skip one specific instruction during the signing process. The resulting faulted signature is used to recover the secret key.

Skip sampling of vinegar values (re-using) The main idea here is to insert an instruction skip during the sampling of the vinegar variables. In Algorithm 1, this corresponds to a jump over line 18, for one (or more) of the $i \in 1, \ldots, k$. This fault injection attack forces the same vinegar variable $v_i \in \mathbb{F}_q^n$ to be used for two consecutive signatures of different messages m and m'. We subtract the obtained correct (not faulted) signature s and the faulted signature s' and receive $s - s' = (s_1 - s'_1, \ldots, s_k - s'_k)$. Observe that for the entry i, where $v_i = v'_i$ holds, we have

$$s_i - s'_i = v_i + o_i - v'_i - o'_i = o_i - o'_i.$$

Since O forms a subspace, we know $o_i - o'_i \in O$ and thus, we found a vector in the secret subspace.

It has already been shown that UOV [KPG99] and Rainbow [DS05] are vulnerable to this kind of attack [AKKM22], so this can be seen as an extension of the approach to MAYO, which also works with vinegar and oil variables. Note, that the attack leads to valid signatures, and therefore, cannot be mitigated by a signature check.

Implemented countermeasure To mitigate this attack we shuffle the vinegar variables $v_i \in \mathbb{F}_q^n$ at the end of the signing algorithm. This is more secure than zeroing the respective variables since $v_i = 0$ could also lead to the leakage of oil variables in the next signing procedure. Thus, it is advisable to permute the entries of the

used variables instead, rendering them unknown to an attacker and ensuring $v_i \neq v'_i$.

Skip addition of oil values An attack vector that follows a similar reasoning, is to skip the addition of the oil variable o_i at the end of the signing process (see line 45 in Algorithm 1) for one (or more) $i \in 1, \ldots, k$. If the fault is injected correctly, this modifies the resulting signature to $s' = (v_1 + o_1, \ldots, v_i, \ldots, v_k + o_k)$. First, we see that s' is not a valid signature anymore, since $\mathcal{P}(s') \neq t$ with very high probability. Let s be the valid signature corresponding to the same message, then we can compute

$$s_i - s'_i = v_i + o_i - v'_i = o_i \in O.$$

Note that the signing is deterministic and the randomness that is used to generate the vinegar variable depends solely on the given message, which we have chosen to be identical. Therefore, $v_i = v'_i$. Again, we found a vector of the secret oilspace $o_i \in O$ and recover the remaining space with the reconciliation attack in negligible time.

Implemented countermeasure To avoid this attack we need to guarantee, that the vinegar and oil variables are really added, and neither of them are part of the signature by skipping their addition or the assignment of their values. Since the faulted signature is not valid anymore, one option is to verify the generated signature. However, this comes with a considerable performance overhead. Therefore, we rather chose to implement a check, that monitors if the entries of the computed signature s_i are different from the earlier generated vinegar variables v_i.

4.2 Side Channel Analysis

In this section, we focus on the leakage of the vector-matrix multiplication function. This function is called multiple times during key generation, secret key expansion and signing. It multiplies a secret vector by a known matrix (part of the public key), as shown in line 29 and 30 of Algorithm 1, as well as in line 16 of the algorithm MAYO.CompactKeyGen() and in line 17 of MAYO:ExpandSK(csk), for which we refer to [BCC+23, Section 2.1.5]. In MAYO, or more general, in UOV-based signature schemes, this is repeated for a considerable amount of public key matrices $P_i^{(1)}$.

An attacker is able to measure the power traces of the multiplication $(v_i)_j \cdot (P_a^{(1)})_{j,\cdot}$ for several $a \in [m]$, perform a profiling or a correlation attack, and predict the value $(v_i)_j$ which is supposed to remain unknown. This attack strategy was demonstrated in [ACK+23] ,where the authors attack an implementation of UOV, that incorporates similar operations as the one mentioned above. Again, the recovered values of v_i lead to efficient key recovery.

Implemented countermeasure In order to execute the SCA successfully, the attacker needs to know both, the value of the cofactor in $P_a^{(1)}$ and at which point in time the target $(v_i)_j$ is multiplied with this value. Thus, our approach to mitigate this attack, is to rearrange the order in which the multiplications are executed. In previous implementations optimized for efficiency a vinegar variable $(v_i)_j$ is picked and multiplied consecutively to the corresponding entry in all $P_a^{(1)}$ for $a \in \{1, \ldots, m\}$. This way, there is a certain interval in the power trace, that contains m multiplications of the sensitive value $(v_i)_j$ with public values. We treat the $P_a^{(1)}$ individually, and thus, the entry $(v_i)_j$ is only multiplied with $(P_a^{(1)})_{j.}$ before we move on to the next multiplication $(v_i)_{j+1} \cdot (P_a^{(1)})_{j+1,.}$. Consequently, on a 32-bit architecture, where at least 4 field elements are treated at once (even 8 if we move to the updated parameters $q = 16$), this massively increases the failure probability of a correlation attack, since the power trace is now related to 4 different secret field elements $((v_i)_j, (v_i)_{j+1}, (v_i)_{j+2}, (v_i)_{j+3})$ at once, and not only to the same secret element $(v_i)_j$ as previously. However, more advanced analysis methods that employ machine learning for the selection of point of interest might still pose a threat to this approach. This could require a vast amount of profiling traces and we leave a concrete analysis thereof as future work.

5 Results, Comparison, and Discussion

In Table 1, we show the resource consumption of the whole design and submodules for the first security level defined by the NIST PQC standardization process [NIS23b]. The parameters defining MAYO are q (the size of the finite field), n (the number of variables in the multivariate quadratic polynomials in the public key), m (the number of multivariate quadratic polynomials in the public key), o (the dimension of the oil space), and k (the whipping parameter, satisfying $ko \geq m$). For our results, these parameters are set to $q = 31$, $n = 62$, $m = 60$, $o = 6$, and $k = 10$.

Our design stands out as the most optimized among the current implementations of multivariate schemes concerning resource utilization. The proposed design effectively utilizes roughly 31% of the total logic resources available on the Zynq board, specifically accounting for 13K Flip-Flops (FFs) and 21K Look-Up Tables (LUTs). These resources are distributed among different sub-modules.

The dominance of the Keccak core is evident as it commands the majority of FPGA slices, enveloping nearly third of the entire design. This dominance arises from its expansive internal buffer and its' interwoven XOR network, crucial for generating the output hash. Additionally, the RNG Core, integrating AES-128, significantly contributes to resource consumption. Remarkably, the combined impact of these cores results in approximately 40% (9K LUTs) of the design's overall slice usage, underscoring the notion that the MAYO core in isolation represents a minimalist design.

Our Gaussian elimination proves an improvement in the memory utilization as compared to the previous work [REBG11]. In [REBG11], the FPGA imple-

mentation on Xilinx Spartan-3 XC3S1500 (300 MHz) consumes 7,384 and 2,574 LUTs and FFs, respectively, for a number of equations equal to 50. In our implementation on a Zynq Z-7020 (100MHz), for a number of equations equal to 60, the consumption in LUTs and FFs is 1,822 and 413, respectively.

	Resource Utilization		
Submodules	LUTs	FFs	DSP
Keccak (Hash)	6759	4453	0
RNG	2354	3208	0
Vector-Matrix multiplication	1035	528	8
Oil Space Sampling	176	289	0
Gaussian Elimination	1822	413	3
Vector Addition	485	300	0
Vector Negation	176	93	0
Vinegar Sampling	245/686*	277/614*	0
BRAMs Port management	448	0	0
FSM Signing	2871	1057	0
Combined Architectures	21000	13005	11

* Secure implementation

Table 1: Resource utilization of our hardware design on Zynq 7020 at a frequency of 100 MHz.

Implementation	Platform	LUT	FF	DSP	BR
Our	Z-7020 @ 100MHz	21,000	13,005	11	129
[HSMR23]	KC705 @ 100MHz	91,266	42,113	2	45
[HSMR23]	AU280 @ 225MHz	89,014	42,066	2	45
[BCH+23]	Artix-7 @ 90.8MHz	32,422	23,262	2	48

Implementation	Platform	Key Generation cylcles	Signing cycles
Our	Z-7020 @ 100MHz	996K	2,867K
[HSMR23]	KC705 @ 100MHz	12K	42K
[BCH+23]	Artix-7 @ 90.8MHz	11,072K	843K

Table 2: Comparison of our results with related work

We present in Table 1 the resource utilization of our implementation. While the implementation by [HSMR23] is highly optimized for efficiency, our implementation shows better performance in the direction of LUT and FF usage, as showed in Table 2. We use 4.3x less LUTs and 3.2x less FFs while our BRAM utilization is 2.8x more, we believe that this is due to the parameter set we follow specifically the choice of q =31. This also has a significant impact on the execution time, since we could not rely on optimized modules, but had to build some of them from scratch like the method for solving SLEs (see Section 3.5 for more details). Tuning our implementation to the new parameter set so that each two elements can be packed in one byte for example will result in reducing considerably the BRs utilization and execution time.

When compared to the implementation from [BCH+23] corresponding to a hardware implementation of the variant of ov-Ip with $n = 112$, $m = 44$, and $\mathbf{F_{256}}$, our implementation shows less consumption of LUTs. This is mainly due to the fact that their implementation LUTs utilization increases with higher q [BCH+23]. For example, for $\mathbf{F_{256}}$, the LUT is 8-in-8-out and requires 40 LUTs in the synthesis, while for $\mathbf{F_{16}}$, it requires 2 LUTs [BCH+23]. Our results show reduced LUTs and FFs which lead to faster logic operations, potentially resulting in improved clock speeds and reduced latency, especially for the key generation. The integration of 8 DSPs for vector-matrix multiplication shows potential for heightened parallel processing capabilities within the system architecture.

Our primary goal revolved around achieving an efficient usage of memory utilization and taking a first step towards physical security. Furthermore, our parameters choice proved the adaptability of the MAYO scheme for deployment in resource-constrained devices *even* in case the field is extended to $q = 31$ instead of 16. In fact, our implementation offers a commendable trade-off, showcasing an

adept combination of efficient resource utilization and operational speed. Furthermore, the implementation of the proposed countermeasures had hardly any impact on resource utilization as shown in Table 1. The increase in clock cycles that originates from the countermeasures lies in the order of hundreds and can be disregarded when considering the overall costs.

6 Conclusion

The implementation of multivariate signature schemes has faced challenges due to their large key sizes, impeding them from deployment on resource-constrained embedded devices. In response, the MAYO scheme was developed as a new modification of the mature UOV signature scheme. MAYO has successfully addressed the issue of large key sizes and can now be seen as one of the prominent candidates of NIST's call for additional digital signatures in regard of performance, key, and signature size. In this paper, we introduced a reconfigurable hardware implementation of MAYO, optimized to reduce the memory consumption during the key generation and the signing processes. Our implementation serves as evidence of MAYO's practicality for real-world deployment especially when deployed in resource-constraints devices. In fact, our design highlights the necessity of time area trade off. Moreover, we discussed a set of new security challenges brought by the deployment of MAYO in embedded systems, particularly in terms of defending against fault injection and side-channel attacks and suggest lightweight countermeasures.

Acknowledgments

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of the project Full Lifecycle Post-Quantum PKI - FLOQI (ID 16KIS1074). Furthermore, this work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - project number 505500359. Moreover, we would like to thank Amir Moradi for his valuable input which greatly improved the paper.

References

ACK+23. Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating Oil and Vinegar with a Single Trace: Side-Channel Assisted Kipnis-Shamir Attack on UOV. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 221–245, 2023.

AKKM22. Thomas Aulbach, Tobias Kovats, Juliane Krämer, and Soundes Marzougui. Recovering Rainbow's Secret Key with a First-Order Fault Attack. In *International Conference on Cryptology in Africa*, pages 348–368. Springer, 2022.

ARM. ARM. Armv7-m architecture reference manual. https://developer.arm.com/documentation/ddi0403/d/Application-Level-Architecture/The-ARMv7-M-Instruction-Set.

BCC+23. Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias Kannwischer. MAYO-algorithm specifications. MAYO team. https://pqmayo.org/assets/specs/mayo.pdf, 2023.

BCH+23. Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. Oil and Vinegar: Modern Parameters and Implementations. *Cryptology ePrint Archive*, 2023.

BDH+22. Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak open-source hardware implementation. https://keccak.team/index.html, 2022.

Beu21. Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I*, pages 348–373. Springer, 2021.

Beu22a. Ward Beullens. Breaking Rainbow Takes a Weekend on a Laptop. In *Annual International Cryptology Conference*, pages 464–479. Springer, 2022.

Beu22b. Ward Beullens. MAYO: Practical Post-Quantum Signatures from Oil-and-Vinegar Maps. In *Selected Areas in Cryptography: 28th International Conference, Virtual Event, September 29–October 1, 2021, Revised Selected Papers*, pages 355–376. Springer, 2022.

DS05. Jintai Ding and Dieter Schmidt. Rainbow, a new Multivariable Polynomial Signature Scheme. In *ACNS*, volume 5, pages 164–175. Springer, 2005.

FG18. Ahmed Ferozpuri and Kris Gaj. High-speed fpga implementation of the nist round 1 rainbow signature scheme. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2018.

HSMR23. Florian Hirner, Michael Streibl, Ahmet Can Mert, and Sujoy Sinha Roy. A hardware implementation of mayo signature scheme. *IACR Cryptol. ePrint Arch.*, 2023:1267, 2023.

HZ18. Yi Haibo and Nie Zhe. High-speed Hardware Architecture for Implementations of Multivariate Signature Generations on FPGAs. In *EURASIP Journal on Wireless Communications and Networking*, pages 1687–1499, 2018.

KPG99. Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced Oil and Vinegar Signature Schemes. In *Eurocrypt*, volume 99, pages 206–222. Springer, 1999.

KS06. Aviad Kipnis and Adi Shamir. Cryptanalysis of the Oil and Vinegar Signature Scheme. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings*, pages 257–266. Springer, 2006.

NIS23a. NIST. NIST post-quantum cryptography standardization. https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline, 2023.

NIS23b. NIST. NIST post-quantum cryptography standardization: Evaluation criteria. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria), 2023.

NIS23c. NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. https://csrc.nist.gov/publications/detail/fips/202/final, 2023.

PQD23. PQDB post-quantum data base. https://www.pqdb.info/, 2023.

REBG11. Andy Rupp, Thomas Eisenbarth, Andrey Bogdanov, and Oliver Grieb. Hardware SLE solvers: Efficient building blocks for cryptographic and cryptanalytic applications. *Integration*, 44(4):290–304, 2011.

TYD+11. Shaohua Tang, Haibo Yi, Jintai Ding, Huan Chen, and Guomin Chen. High-speed Hardware Implementation of Rainbow Signature on FPGAs. In *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011. Proceedings 4*, pages 228–243. Springer, 2011.

Xil23. AMD Xilinx. Zynq-7000 SoCs with Hardware and Software Programmability. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, 2023.

XL21. Yufei Xing and Shuguo Li. A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):328–356, Feb. 2021.

ZZW+21. Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):270–295, Nov. 2021.