MAYO Lite: a Low-RAM Implementation of the MAYO Signature Scheme

Sven Bauer, Fabrizio De Santis and Florian Wilde

Siemens AG – Foundational Technologies – Research and Pre-Development
{svenbauer,fabrizio.desantis,florian.wilde}@siemens.com

Abstract. MAYO is a signature scheme based on the Unbalanced Oil and Vinegar (UOV) construction and a third round candidate in the NIST standardization process for additional post-quantum signature schemes. We present a memory-optimized pure-C implementation of MAYO signature verification that reduces RAM consumption by 97–99% compared to the reference implementation provided by the PQM4 project [KPR+] at the cost of increasing runtime by 50–200% and while maintaining code size. This reduction finally enables verifying MAYO signatures on smart cards and small microcontrollers with only a few kilobytes of RAM. We achieve it through three coupled design choices: a) we expand the public key vector-by-vector on the fly rather than all at once at the start; b) we reorder the calculation of the matrix product SPS so that each vector of P is used exactly once, which avoids repeated expansions; and c) we add precomputed multiples of each P-vector directly onto the result instead of accumulating P-vectors before multiplication. We provide results using the PQM4 framework for all parameter sets listed in the specification and supported by PQM4 on our platform, including three main parameter sets $\text{MAYO}_{\{1,2,3\}}$. This enables direct comparison with other post-quantum digital signature schemes. Further parameter sets not supported by PQM4, including the fourth main parameter set MAYO_5, are measured on our own framework, which supports key and signature generation on the host. Our results for non-standard MAYO parameter sets offer insights into the performance and scalability of the proposed approach that may inform ongoing standardization efforts.

Keywords: Post-Quantum Signatures · MAYO · UOV · Embedded · Cortex-M4

1 Introduction

Post-quantum signature schemes face a dual challenge on constrained embedded devices: they must meet demanding security requirements while operating within severe memory and computational limitations. Among the candidates in the NIST additional digital signature standardization process [NIS], multivariate-based schemes such as MAYO [BCC+25] stand out for their compact signatures (a few hundred bytes) and small public keys (a few kilobytes). These properties make MAYO attractive for use cases such as firmware signature verification on microcontrollers, where the signature and public key must fit into memory alongside the application code.

However, existing MAYO implementations exhibit a critical bottleneck that prevents deployment on precisely such devices: their verification routine requires hundreds of kilobytes of RAM, which is orders of magnitude more than the public key itself, because the entire quadratic public key map must be fully expanded into a matrix representation in memory before evaluation can begin [BCC+24, KPR+]. For MAYO_1, the smallest NIST security category 1 parameter set, the PQM4 implementation consumes over 200 KiB of stack for verification. This RAM footprint excludes MAYO from the large class of

resource-constrained devices like smart cards, sensor nodes, and small microcontrollers, whose total RAM budget is measured in single-digit kilobytes and for which signature verification is a critical security capability.

In this work, we show that this limitation is not inherent to the MAYO scheme. By restructuring the core quadratic map evaluation as a single streaming pass over the public key coefficients, we eliminate the need to hold the expanded public key in memory. Each coefficient vector is expanded on demand from its seed, consumed exactly once, and discarded, with the result accumulated in a small, fixed-size buffer. For the variants $\text{MAYO}_{\{1,2,3\}}$, this reduces RAM consumption for verification to 2–3% of that of the most stack-efficient existing implementation at the cost of increasing runtime to 150–300%. RAM consumption for verifying a MAYO_1 signature, for example, is reduced from over 200 KiB down to under 7 KiB. This reduction finally makes MAYO viable for verifying signed firmware updates on microcontrollers with only a few kilobytes of RAM.

Contributions. This work makes the following contributions:

1. We identify the two dominant RAM contributors in existing MAYO verification: the fully expanded public key and the matrix-multiplication accumulators, together accounting for over 95% of the total RAM footprint. We quantify their sizes across all standardized parameter sets (Section 3.1). Then, we propose a RAM-optimized MAYO verification that reorders the operations in the core matrix multiplication so that each public-key depth-vector is expanded on demand, consumed once via precomputed multiples, and discarded, eliminating both the expanded-key storage and the large accumulator arrays that prior implementations require (Section 3).

2. We evaluate our implementation on a Cortex-M4 platform using the PQM4 framework and a custom framework, demonstrating measured RAM reductions of 97–99% with 50–200% runtime increase, while maintaining, and in most cases reducing, code size (Section 4).

3. Finally, we additionally report results for non-standard parameter sets of MAYO. These results may provide useful insights for ongoing standardization efforts and help assess the scalability of the proposed approach beyond the currently defined parameter sets.

Related work. Prior work on MAYO implementation and optimizations falls into two streams: speed-optimized software implementations and hardware acceleration. In the software stream, Beullens et al. [BCC+24] introduced nibble-sliced representations and the Method of the Four Russians (M4R) for efficient matrix multiplication on Cortex-M4 and AVX2 platforms, achieving signing times up to 21% faster than prior bitsliced approaches on Cortex-M4. Their implementation prioritizes speed, but at the cost of expanding the entire public key into RAM and maintaining large accumulator arrays, resulting in a verification stack footprint of over 150 KiB for all standardized parameter sets. The reference C implementation [BCC+23] provides a `HAVE_STACKEFFICIENT` compile-time flag that reduces the accumulator size by a factor of k at the cost of reading each coefficient of P k times and thus still holds the fully expanded public key in memory. The PQM4 project [KPR+] ports this implementation to Cortex-M4 and provides the baseline against which we compare in Section 4. Across all of these software implementations, the speed-oriented matrix multiplication scheme is fundamentally coupled to the fully expanded public key, keeping RAM usage in the hundreds of kilobytes. This coupling is the specific gap our work breaks.

In the hardware stream, Hirner et al. [HSK+23, Str24] describe expanding $\ddot{\mathbf{P}}_i^{(1)}$ and $\ddot{\mathbf{P}}_i^{(2)}$ on the fly in a hardware MAYO accelerator targeting high-throughput ASIC and FPGA platforms. Their approach processes coefficients in a different order than ours,

which leads to either repeated unpacking of the same coefficients (with a performance penalty) or RAM requirements that remain significantly larger than our solution. The HaMAYO hardware implementation [SMA+23] focuses on fault-tolerant reconfigurable hardware and does not implement verification. Whereas these hardware works demonstrate that on-demand expansion is feasible in custom logic, they require an FPGA or ASIC, i.e., resources unavailable on general-purpose microcontrollers, and thus do not address the pure-software verification challenge we solve.

Outline. Section 2 introduces the notation, Unbalanced Oil and Vinegar (UOV) signatures, and the MAYO signature scheme. Section 3 presents our RAM-optimized verification together with its correctness and execution-time analysis. Section 4 reports our measurement results. Section 5 concludes and outlines future work.

2 Preliminaries

2.1 Notation

For an integer $n > 0$, we write $[n]$ for the set $\{0, \ldots, n-1\}$. We denote the finite field with q elements by $\mathbb{F}_q$. Non-scalars are written in bold with the number of dimensions indicated by the number of dots above. E.g., $\dot{\mathbf{x}} \in \mathbb{F}_q^n$ is a vector, $\ddot{\mathbf{M}} \in \mathbb{F}_q^{n \times k}$ is a matrix with n rows and k columns, and $\dddot{\mathbf{P}} \in \mathbb{F}_q^{n \times k \times m}$ is a three-dimensional array stacking m matrices together.

For a square matrix $\ddot{\mathbf{A}} \in \mathbb{F}_q^{n \times n}$, we define $\mathsf{Upper}(\ddot{\mathbf{A}})$ as

$$\mathsf{Upper}(\ddot{\mathbf{A}}) = (b_{ij})_{i,j \in [n]} \text{ where } b_{ij} = \begin{cases} a_{ij} + a_{ji} & \text{if } i < j \\ a_{ii} & \text{if } i = j \\ 0 & \text{if } i > j \end{cases}. \tag{1}$$

Note that $\mathsf{Upper}(\ddot{\mathbf{A}})$ is an upper triangular matrix and that

$$\dot{\mathbf{x}}^T \ddot{\mathbf{A}} \dot{\mathbf{x}} = \dot{\mathbf{x}}^T \mathsf{Upper}(\ddot{\mathbf{A}}) \dot{\mathbf{x}} \quad \text{for all } \dot{\mathbf{x}} \in \mathbb{F}_q^n. \tag{2}$$

The set of byte strings of arbitrary length is denoted by $\mathcal{B}^*$.

2.2 Unbalanced Oil and Vinegar signatures

The MAYO signature scheme [BCC+25] is based on the Unbalanced Oil and Vinegar (UOV) signature scheme first described in [KPG99], which itself originates from the balanced Oil and Vinegar construction of Patarin [Pat95] (broken shortly thereafter by Kipnis and Shamir [KS98]). We therefore introduce UOV signatures first. A UOV signature scheme has system parameters consisting of a prime power q and integers n, m, and o such that $n > 2m$ and $m > o$. It further uses a hash function $H : \mathcal{B}^* \times \mathcal{B}^* \to \mathbb{F}_q^m$ that maps a message and a random salt to a vector in $\mathbb{F}_q^m$.

The public key is a map

$$\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m, \mathcal{P}(\dot{\mathbf{x}}) := (p_0(\dot{\mathbf{x}}), \ldots, p_{m-1}(\dot{\mathbf{x}})), \tag{3}$$

where each of the p_i is a homogeneous quadratic polynomial. Hence, each p_i can be written as

$$p_i(\dot{\mathbf{x}}) = p_i(x_0, \ldots, x_{n-1}) = \sum_{0 \le r \le s < n} p_{i,r,s} x_r x_s = \dot{\mathbf{x}}^T \ddot{\mathbf{P}}_i \dot{\mathbf{x}}, \tag{4}$$

where $\ddot{\mathbf{P}}_i \in \mathbb{F}_q^{n \times n}$ is an upper triangular matrix.

The secret key is a vector subspace $O \subset \mathbb{F}_q^n$ of dimension o such that $\mathcal{P}(\dot{\mathbf{x}}) = 0$ for all $\dot{\mathbf{x}} \in O$.

A valid signature of a message M is a pair $(\dot{\mathbf{s}}, \mathsf{salt}) \in \mathbb{F}_q^n \times \mathcal{B}^*$ such that $\mathcal{P}(\dot{\mathbf{s}}) = H(\mathsf{M}, \mathsf{salt})$.

Key generation for UOV is shown in Algorithm 1. The public key consists of $(\mathsf{seed}_{\mathrm{pk}}, \ddot{\dot{\mathbf{P}}}^{(3)})$. The secret key consists of $(\mathsf{seed}_{\mathrm{sk}}, \ddot{\mathbf{O}})$. We now show that this description is equivalent to the description of public and secret key given above. The map $\mathcal{P}$ as in Eq. (3) is given by

$$p_i(\dot{\mathbf{x}}) = \dot{\mathbf{x}}^T \begin{pmatrix} \ddot{\mathbf{P}}_i^{(1)} & \ddot{\mathbf{P}}_i^{(2)} \\ 0 & \ddot{\mathbf{P}}_i^{(3)} \end{pmatrix} \dot{\mathbf{x}}. \tag{5}$$

The secret linear subspace O is the column space of the matrix $\begin{pmatrix} \ddot{\mathbf{O}} \\ \ddot{\mathbf{I}}_o \end{pmatrix}$. We have to show that $\mathcal{P}(\dot{\mathbf{x}}) = 0$ for all $\dot{\mathbf{x}} \in O$. So, let $\dot{\mathbf{x}} \in O$. Then, there is a $\dot{\mathbf{y}} \in \mathbb{F}_q^o$ such that $\dot{\mathbf{x}} = \begin{pmatrix} \ddot{\mathbf{O}} \\ \ddot{\mathbf{I}}_o \end{pmatrix} \dot{\mathbf{y}}$. Hence,

$$\begin{aligned} p_i(\dot{\mathbf{x}}) =& \dot{\mathbf{x}}^T \begin{pmatrix} \ddot{\mathbf{P}}_i^{(1)} & \ddot{\mathbf{P}}_i^{(2)} \\ 0 & \ddot{\mathbf{P}}_i^{(3)} \end{pmatrix} \dot{\mathbf{x}} = \dot{\mathbf{y}}^T \ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(1)} \ddot{\mathbf{O}} \dot{\mathbf{y}} + \dot{\mathbf{y}}^T \ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(2)} \dot{\mathbf{y}} + \dot{\mathbf{y}}^T \ddot{\mathbf{P}}_i^{(3)} \dot{\mathbf{y}} \\ =& \dot{\mathbf{y}}^T (\ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(1)} \ddot{\mathbf{O}} + \ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(2)}) \dot{\mathbf{y}} - \dot{\mathbf{y}}^T \mathsf{Upper}(\ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(1)} \ddot{\mathbf{O}} + \ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(2)}) \dot{\mathbf{y}} \\ =& 0 \end{aligned} \tag{6}$$

for all $i \in [m]$ by Eq. (2). So, $\mathcal{P}(\dot{\mathbf{x}}) = 0$.

The idea of compressing a UOV key pair by pseudo-randomly expanding $\dddot{\mathbf{P}}^{(1)}$ and $\dddot{\mathbf{P}}^{(2)}$ as in Algorithm 1 seems to have been introduced first by Petzoldt et al. in [PTBW11], who showed that the $\dddot{\mathbf{P}}^{(1)}$ and $\dddot{\mathbf{P}}^{(2)}$ components can be generated from a short seed rather than stored explicitly.

Algorithm 1 UOV.KeyGeneration

Output: Data to construct $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ as in Eq. (3) (the public key)
Output: Base of a linear subspace $O \subset \mathbb{F}_q^n$ with $\dim O = o$ and $\mathcal{P}(\dot{\mathbf{x}}) = 0$ for all $\dot{\mathbf{x}} \in O$ (the secret key)

1: Initialize a pseudo-random number generator PRNG with a random $\mathsf{seed}_{\mathrm{pk}}$.
2: Generate prism $\dddot{\mathbf{P}}^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)\times m}$ containing the upper triangular matrices $\ddot{\mathbf{P}}_i^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)}$ from PRNG.
3: Generate cuboid $\dddot{\mathbf{P}}^{(2)} \in \mathbb{F}_q^{(n-o)\times o\times m}$ containing the matrices $\ddot{\mathbf{P}}_i^{(2)} \in \mathbb{F}_q^{(n-o)\times o}$ from PRNG.
4: Initialize the pseudo-random number generator PRNG with a random $\mathsf{seed}_{\mathrm{sk}}$.
5: Generate a matrix $\ddot{\mathbf{O}} \in \mathbb{F}_q^{(n-o)\times o}$ from PRNG.
6: $\ddot{\mathbf{P}}_i^{(3)} \leftarrow -\mathsf{Upper}(\ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(1)} \ddot{\mathbf{O}} + \ddot{\mathbf{O}}^T \ddot{\mathbf{P}}_i^{(2)})$ for all $i \in [m]$.
7: **return** $(\mathsf{seed}_{\mathrm{pk}}, \ddot{\dot{\mathbf{P}}}^{(3)})$ as the public key and $(\mathsf{seed}_{\mathrm{sk}}, \ddot{\mathbf{O}})$ as the secret key.

To describe the signing process, we need the observation that the map

$$\mathcal{P}' : \mathbb{F}_q^n \times \mathbb{F}_q^n \to \mathbb{F}_q^m, \mathcal{P}'(\dot{\mathbf{x}}, \dot{\mathbf{y}}) := \mathcal{P}(\dot{\mathbf{x}} + \dot{\mathbf{y}}) - \mathcal{P}(\dot{\mathbf{x}}) - \mathcal{P}(\dot{\mathbf{y}}) \tag{7}$$

is bilinear. This can be verified by direct calculation using the description $p_i(\dot{\mathbf{x}}) = \dot{\mathbf{x}}^T \ddot{\mathbf{P}}_i \dot{\mathbf{x}}$.

Remark 1. Choosing a linear complement V we can split $\mathbb{F}_q^n$ as a direct sum $\mathbb{F}_q^n = V \oplus O$ with a "vinegar space" V and the "oil space" O. We have an unbalanced oil and vinegar scheme because $\dim V > \dim O$. Now we can write the public key map as

$$\mathcal{P}(\dot{\mathbf{v}} + \dot{\mathbf{o}}) = \mathcal{P}(\dot{\mathbf{v}}) + \mathcal{P}'(\dot{\mathbf{v}}, \dot{\mathbf{o}}). \tag{8}$$

Because $\mathcal{P}'$ is bilinear, for fixed $\dot{\mathbf{v}}$, this is an affine map in $\dot{\mathbf{o}}$. So, the secret key can also be seen as a coordinate system in which the public key map is affine in the "oil" variables.

The signing algorithm is shown in Algorithm 2. Observe that for its output the following holds:

$$\begin{aligned}\mathcal{P}(\dot{\mathbf{s}}) =& \mathcal{P}(\dot{\mathbf{v}} + \dot{\mathbf{x}}) = \mathcal{P}'(\dot{\mathbf{v}}, \dot{\mathbf{x}}) + \mathcal{P}(\dot{\mathbf{v}}) + \mathcal{P}(\dot{\mathbf{x}}) \\ =& (\dot{\mathbf{t}} - \mathcal{P}(\dot{\mathbf{v}})) + \mathcal{P}(\dot{\mathbf{v}}) + 0 = \dot{\mathbf{t}} = H(\mathsf{M}, \mathsf{salt}),\end{aligned} \tag{9}$$

where we have used the fact that $\dot{\mathbf{x}} \in O$ and that by definition of O we have $\mathcal{P}(\dot{\mathbf{x}}) = 0$. Hence, the output is indeed a valid signature that can successfully be verified with Algorithm 3.

Algorithm 2 $\mathsf{UOV.Sign}(O, \mathcal{P}, M)$

Input: Secret linear subspace $O \subset \mathbb{F}_q^n$ (the secret key)
Input: Public map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ (the public key)
Input: Message $\mathsf{M} \in \mathcal{B}^*$
Output: Signature $(\dot{\mathbf{s}}, \mathsf{salt})$ such that $\mathcal{P}(\dot{\mathbf{s}}) = H(\mathsf{M}, \mathsf{salt})$

1: Choose a random salt.
2: $\dot{\mathbf{t}} \leftarrow H(\mathsf{M}, \mathsf{salt}) \in \mathbb{F}_q^m$.
3: **repeat**
4: Choose random $\dot{\mathbf{v}} \in \mathbb{F}_q^n$.
5: Solve the linear equation $\mathcal{P}'(\dot{\mathbf{v}}, \dot{\mathbf{x}}) = \dot{\mathbf{t}} - \mathcal{P}(\dot{\mathbf{v}})$ for $\dot{\mathbf{x}} \in O$.
6: **until** $\dot{\mathbf{x}} \neq \perp$
7: $\dot{\mathbf{s}} \leftarrow \dot{\mathbf{v}} + \dot{\mathbf{x}}$
8: **return** $(\dot{\mathbf{s}}, \mathsf{salt})$

Algorithm 3 $\mathsf{UOV.Verify}(\mathcal{P}, \mathsf{M}, (\dot{\mathbf{s}}, \mathsf{salt}))$

Input: Public map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ (the public key)
Input: Message $\mathsf{M} \in \mathcal{B}^*$
Input: Signature $(\dot{\mathbf{s}}, \mathsf{salt}) \in \mathbb{F}_q^n \times \mathcal{B}^*$
Output: Return `true` if the signature is valid, `false` otherwise

1: $\dot{\mathbf{t}} \leftarrow H(\mathsf{M}, \mathsf{salt})$
2: **return** $\mathcal{P}(\dot{\mathbf{s}}) == \dot{\mathbf{t}}$

2.3 The MAYO signature scheme

The MAYO signature scheme [BCC+25] is based on the UOV construction. It has been submitted to the NIST standardization process for additional post-quantum signature schemes [NIS]. UOV signature schemes offer compact signatures—typically a few hundred bytes—but have large public and secret keys—typically several hundred kilobytes. The important distinguishing feature of MAYO is a specific technique to compress the public and secret keys. In MAYO the parameters n and o are "too small", but a technique referred to as "whipping up" is used to obtain a map $\mathcal{P}^* : \mathbb{F}_q^{kn} \to \mathbb{F}_q^m$ from a UOV public key map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ as in Eq. (3). More precisely, $\mathcal{P}^*$ is defined as

$$\mathcal{P}^*(\dot{\mathbf{x}}_0, \ldots, \dot{\mathbf{x}}_{k-1}) = \sum_{i=0}^{k-1} \ddot{\mathbf{E}}_{ii} \mathcal{P}(\dot{\mathbf{x}}_i) + \sum_{0 \leq i < j < k} \ddot{\mathbf{E}}_{ij} \mathcal{P}'(\dot{\mathbf{x}}_i, \dot{\mathbf{x}}_j). \tag{10}$$

The $\ddot{\mathbf{E}}_{ij} \in \mathbb{F}_q^{m \times m}$ are matrices defined in [BCC+25].

Remark 2. Continuing in the spirit of oil and vinegar, one can view the matrices $\ddot{\mathbf{E}}_{ij}$ as egg, the third main ingredient in mayonnaise [Beu21].

Parameter sets for the main MAYO variants defined in [BCC+25] are shown in Table 1. It is worth noting that [BCC+25] actually defines several additional parameter sets, offering different trade-offs between the sizes of public key, secret key and signatures. As we will analyze our proposed implementation also for most of these additional parameter sets, we came up with a more generic naming scheme: We write mayo in lower case to distinguish it from the official names, followed by one digit indicating the NIST security category of the parameter set, followed by two digits for the value of k in this parameter set. For examples of our naming scheme, see Table 1, which lists the official name, our name, and the parameters for the main MAYO variants.

Key generation in MAYO is essentially the same as in Algorithm 1. The public seed $\mathsf{seed}_{\mathrm{pk}}$ is derived pseudo-randomly from the secret seed $\mathsf{seed}_{\mathrm{sk}}$. Because we focus on the use case of verifying firmware updates on small microcontrollers, the exact particulars of the MAYO key generation process and signing with MAYO are not important in the present context, and we refer the reader to the MAYO specification [BCC+25] for details.

Signature verification in MAYO is shown in Algorithm 4. The various functions called Decode convert byte strings into matrices or vectors with coefficients in $\mathbb{F}_{16}$. The two nibbles of each byte encode two elements of $\mathbb{F}_{16}$. The bit-exact representation of $\mathbb{F}_{16}$ is not important in the present context and can be found in the MAYO specification [BCC+25]. In cases where the order of coefficients matters we will explicitly mention this. The subroutine Compute_y handles the "whipping up" described in Eq. (10): It applies the fixed matrices $\ddot{\mathbf{E}}_{ij}$ to combine the k^2 depth-vectors of $\dddot{\mathbf{Y}} \in \mathbb{F}_q^{k\times k\times m}$ into the single output vector $\dot{\mathbf{y}} \in \mathbb{F}_q^m$. Being part of `compute_rhs()` in all considered implementations, it operates in place on the already-allocated memory for the result $\dddot{\mathbf{Y}}$ of matrix multiplication and therefore does not contribute significantly to RAM usage.

Algorithm 4 MAYO.Verify(cpk, M, sig)

Input: Compact public key $\mathsf{cpk} = (\mathsf{seed}_{\mathrm{pk}}, \mathsf{P3_bytes})$
Input: Message M
Input: Signature $\mathsf{sig} = (\mathsf{salt}, \mathsf{S_bytes})$
Output: Return `true` if the signature is valid, `false` otherwise

1: ▷ *Expand public key components* $\dddot{\mathbf{P}}^{(1)} \in \mathbb{F}_q^{(n-o)\times(n-o)\times m}$, $\dddot{\mathbf{P}}^{(2)} \in \mathbb{F}_q^{(n-o)\times o\times m}$. ◁
2: $\{\dddot{\mathbf{P}}^{(1)}, \dddot{\mathbf{P}}^{(2)}\} \leftarrow \mathsf{AES128_CTR}(\mathsf{seed}_{\mathrm{pk}}, 0, m(n-o)(n+o+1)/2)$
3: ▷ *Read public key component* $\dddot{\mathbf{P}}^{(3)} \in \mathbb{F}_q^{o\times o\times m}$. ◁
4: $\dddot{\mathbf{P}}^{(3)} \leftarrow \mathsf{Decode}_{\dddot{\mathbf{P}}^{(3)}}(\mathsf{P3_bytes})$
5: ▷ *Decode signature.* $\ddot{\mathbf{S}} \in \mathbb{F}_q^{k\times n}$ ◁
6: $\ddot{\mathbf{S}} \leftarrow \mathsf{Decode}_{\ddot{\mathbf{S}}}(\mathsf{S_bytes})$
7: ▷ *Hash message.* $\dot{\mathbf{t}} \in \mathbb{F}_q^m$ ◁
8: $\dot{\mathbf{t}} \leftarrow \mathsf{Decode}_{\dot{\mathbf{t}}}(\mathsf{SHAKE256}(\mathsf{M}\|\mathsf{salt}, \lceil m/2\rceil))$
9: ▷ *Evaluate public map on signature.* $\dddot{\mathbf{Y}} \in \mathbb{F}_q^{k\times k\times m}$, $\dot{\mathbf{y}} \in \mathbb{F}_q^m$ ◁
10: **for all** $i \in [m]$ **do**
11: $\quad \ddot{\mathbf{Y}}_i \leftarrow \ddot{\mathbf{S}} \begin{pmatrix} \ddot{\mathbf{P}}_i^{(1)} & \ddot{\mathbf{P}}_i^{(2)} \\ 0 & \ddot{\mathbf{P}}_i^{(3)} \end{pmatrix} \ddot{\mathbf{S}}^T$
12: $\dot{\mathbf{y}} \leftarrow \mathsf{Compute_y}(\dddot{\mathbf{Y}})$
13: ▷ *Check equality and return result* ◁
14: **return** $\dot{\mathbf{y}} == \dot{\mathbf{t}}$

We now describe public key expansion as in Algorithm 4, Line 2 in detail. Each matrix $\ddot{\mathbf{P}}_i^{(1)}$ has $(n-o)(n-o+1)/2$ coefficients. Each matrix $\ddot{\mathbf{P}}_i^{(2)}$ has $(n-o)o$ coefficients. Each coefficient is an element of $\mathbb{F}_{16}$ and hence requires 4 bit = 0.5 B. Therefore, $\lceil m((n-o)(n-o+1)/2+(n-o)o)/2\rceil$ B = $\lceil m(n-o)(n+o+1)/4\rceil$ B must be generated to expand $\dddot{\mathbf{P}}^{(1)}$ and

Algorithm 5 AES128_CTR(k, s, ℓ)

Input: AES128 key $\mathsf{k} \in \mathcal{B}^{16}$
Input: Starting nibble index s.
Input: Length ℓ of output in nibbles.
Output: Return nibbles at index $s, s+1, \ldots, s+\ell-1$ of the pseudo-random byte string $\mathsf{AES128}(\mathsf{k}, 0) \| \mathsf{AES128}(\mathsf{k}, 1) \| \ldots$

```
out ← empty byte string
for i = 0, ..., ⌈ℓ/32⌉ − 1 do
    counter ← ⌊s/32⌋ + i as a big-endian 16-byte integer
    block ← AES128(k, counter)          ▷ Encrypt counter under key k
    out ← out‖block
return first ℓ nibbles of out
```

$\ddot{\mathbf{P}}^{(2)}$. This is done by operating AES128 in counter mode as shown in Algorithm 5 with the key $\mathsf{seed}_{\mathrm{pk}}$ and an initial counter value of 0 encoded as a big-endian 16-byte integer. Using AES in this way is a standard solution for expanding seeds into pseudorandom values. The resulting byte stream is decoded as follows: The lower nibble of the first byte is the coefficient $P^{(1)}_{0,0,0}$, the upper nibble of the first byte is $P^{(1)}_{0,0,1}$. When all m top left coefficients $P^{(1)}_{0,0,i}$ have been decoded, the decoding continues with $P^{(1)}_{0,1,0}$, $P^{(1)}_{0,1,1}$ and so on. Figure 1 shows this geometrically. Expansion proceeds first along the depth-axis, then along the rows and then along the columns. When all elements of $\ddot{\mathbf{P}}^{(1)}$ have been decoded, decoding continues with $\ddot{\mathbf{P}}^{(2)}$ in the same way.

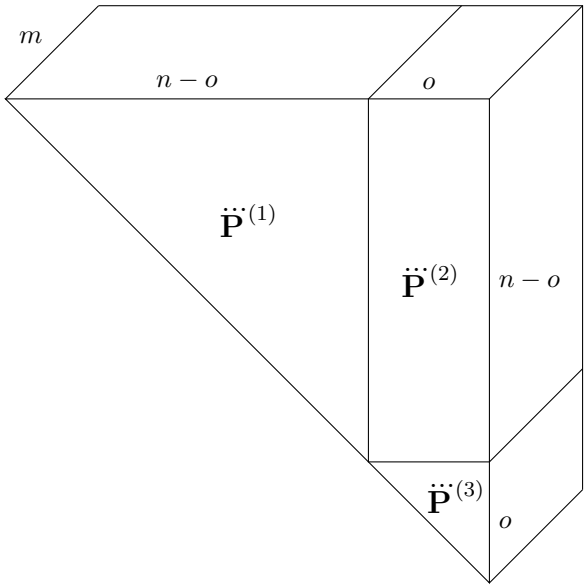


Figure 1: Geometric depiction of the MAYO public key. The front plane represents the matrices $\ddot{\mathbf{P}}^{(i)}_0$ with $i \in \{1, 2, 3\}$. AES in counter mode is used to generate the coefficients of $\ddot{\mathbf{P}}^{(1)}$ and $\ddot{\mathbf{P}}^{(2)}$, filling $\ddot{\mathbf{P}}^{(1)}$ front to back, left to right, and top to bottom, then continuing with $\ddot{\mathbf{P}}^{(2)}$ likewise.

Table 1: The proposed standard MAYO parameter sets [BCC+25]

official name	MAYO_1	MAYO_2	MAYO_3	MAYO_5
our name	mayo110	mayo104	mayo311	mayo512
NIST security category	1	1	3	5
n	86	81	118	154
m	78	64	108	142
o	8	17	10	12
k	10	4	11	12
q	16	16	16	16
signature size	454 B	186 B	681 B	964 B
compact public key size	1420 B	4912 B	2986 B	5554 B
expanded public key size	142 KiB	104 KiB	370 KiB	828 KiB

3 RAM optimized MAYO signature verification

A major drawback of existing MAYO implementations, even those in the PQM4 project [KPR+] tailored to embedded microcontroller platforms, is that they use a speed-optimized matrix multiplication for $\ddot{\mathbf{S}}\ddot{\mathbf{P}}\ddot{\mathbf{S}}^T$ that requires large accumulator arrays and a fully expanded public key $\dddot{\mathbf{P}}$ in RAM, filling more than 100 KiB even for the smallest parameter sets. This approach couples the required amount of working memory to the expanded public key size, making it unsuitable for microcontrollers with only a few kilobytes of RAM. In this section, we analyze which implementation choices led to this coupling and propose a streaming approach that breaks it: each public-key coefficient vector is expanded on demand, consumed exactly once in a reordered matrix multiplication, and discarded before the next vector is fetched. At the end of the section, we further show that our approach provides a substantially better time-memory trade-off than existing implementations.

3.1 Analysis of RAM consumption

Let us first analyze the RAM consumption of a direct implementation of Algorithm 4. The public key part $\ddot{\mathbf{P}}^{(3)}$ is stored uncompressed in the compact public key and hence does not require additional RAM. Similarly, the signature $\ddot{\mathbf{S}}$ can be read directly from the signature byte string. Expanding $\ddot{\mathbf{P}}^{(1)}$ and $\ddot{\mathbf{P}}^{(2)}$ requires $\lceil m(n-o)(n+o+1)/4 \rceil$ B of RAM.

Storing the target vector $\dot{\mathbf{t}}$ requires $\lceil m/2 \rceil$ B, storing $\ddot{\mathbf{Y}}$ requires $\lceil k^2 m/2 \rceil$ B, and storing $\dot{\mathbf{y}}$ requires another $\lceil m/2 \rceil$ B of RAM. Some additional RAM is required for computing the matrix multiplication in Line 10 of Algorithm 4. This amount is highly implementation dependent and is the key differentiator between implementations.

Let us now look at two real-world implementations: the reference implementation from [BCC+23] and the similar C-based implementation that is part of the PQM4 project [KPR+][1]. Their major contributors to RAM requirement are shown in Table 2 based on their C-code, i.e. not considering compiler optimizations. Storage for loop variables and other temporaries is negligible in comparison. Note that to efficiently work on m-element vectors, both use the `uint64_t` data type, which means memory consumption is quantized to multiples of 8 B for involved variables. Both expand not just $\ddot{\mathbf{P}}^{(1)}$ and $\ddot{\mathbf{P}}^{(2)}$ but also convert $\ddot{\mathbf{P}}^{(3)}$ into the same vector format, which increases speed, but also RAM usage. For the matrix multiplications, both implementations optimize for speed by using

[1]The PQM4 project includes the C implementation at https://github.com/mupq/mupq/tree/ddcccedb2db9d0250856bd58ee5c46c61e506c5d.

Table 2: Required RAM (in byte) for selected MAYO verification implementations. Reference corresponds to the original implementation in [BCC+23], Stack-efficient to the `HAVE_STACKEFFICIENT` variant of [BCC+23] and the implementation in [KPR+]. $\hat{m} = 8\lceil m/16 \rceil$ denotes the number of bytes required to store m nibbles using the `uint64_t` data type.

Function	Variable	Reference	Stack-efficient	This Work
`mayo_verify()`	`tEnc`	$\lceil m/2 \rceil$	$\lceil m/2 \rceil$	$\lceil m/2 \rceil$
	`t`	m	m	m
	`y`	$2m$	$2m$	$2m$
	`s`	kn	kn	kn
	`pk`	$\hat{m}n(n+1)/2$	$\hat{m}n(n+1)/2$	733
`eval_public_map()`	`SPS`	$k^2\hat{m}$	$k^2\hat{m}$	$k^2\hat{m}$
	`zero`	m	m	m
matrix multiplication in $\mathbb{F}_q^m$	–	$17kn\hat{m}$	$(k+17n)\hat{m}$	$15\hat{m}$
`compute_rhs()`	`temp`	$\hat{m}$	$\hat{m}$	$\hat{m}$

accumulator arrays. In those arrays, summands to be multiplied with the same factor are added up until eventually being multiplied out and added onto the final result, which reduces the number of scalar multiplications in $\mathbb{F}_q^m$. Their size is determined by the number of possible factors, which in case of MAYO is $q = 16$. The reference implementation without the `HAVE_STACKEFFICIENT` flag requires an accumulator of size $16kmn/2$ B for it, because it calculates the entire products $\ddot{\mathbf{P}}_{[m]}\ddot{\mathbf{S}}^T$ before multiplying $\ddot{\mathbf{S}}$ onto each from the left-hand side. Setting the `HAVE_STACKEFFICIENT` flag or using the PQM4 implementation slightly optimizes for RAM by calculating the m matrix products $\ddot{\mathbf{S}}\ddot{\mathbf{P}}_{[m]}\ddot{\mathbf{S}}^T$ partially, one column of the result after the other, which allows to shrink the accumulator by a factor of k to $16mn/2$ B. The price for this optimization, however, is accessing all elements of $\dddot{\mathbf{P}}$ k times, which makes it only viable with fully expanded $\dddot{\mathbf{P}}^{(1)}$, $\dddot{\mathbf{P}}^{(2)}$. Concretely, both implementations realize Line 11 of Algorithm 4 as two nested passes. An inner pass forms the m products $\ddot{\mathbf{P}}_i\ddot{\mathbf{S}}^T$ column by column into an accumulator indexed by the $q = 16$ possible field factors, and an outer pass multiplies $\ddot{\mathbf{S}}$ onto the partial results from the left. Because the factor-indexed accumulator must persist across the entire inner pass over $\dddot{\mathbf{P}}$, its footprint scales with mn (or kmn without the flag). As a further consequence, each coefficient of $\dddot{\mathbf{P}}$ is re-read on every one of the k outer iterations. This is precisely the coupling between accumulator size and expansion count that our reordering breaks. Hence, for the main MAYO variants defined in Table 1, the PQM4 implementation is estimated to require about 209 KiB for MAYO_1, 148 KiB for MAYO_2, 503 KiB for MAYO_3 and 1037 KiB for MAYO_5.

An implementation that aims to significantly reduce RAM usage must therefore address the two main contributors: the fully expanded public key created in Line 2 of Algorithm 4 and the buffers needed for matrix multiplications in Lines 10f. of Algorithm 4. An obvious idea to save RAM would be to simply expand $\dddot{\mathbf{P}}^{(1)}$, $\dddot{\mathbf{P}}^{(2)}$ and convert $\dddot{\mathbf{P}}^{(3)}$ partially as needed within the loop in Lines 10f. However, with PQM4's current implementation of matrix multiplication, this would severely degrade performance as each element of the public key would need to be expanded k times. Avoiding repeated expansion by first calculating $\ddot{\mathbf{P}}_{[m]}\ddot{\mathbf{S}}^T$ entirely similar to the reference implementation would require an accumulator array much larger than the fully expanded public key and therefore is neither an option.

Table 3: Number of operations in $\mathbb{F}_q^m$ required for matrix multiplication in our implementation compared to the existing stack-efficient implementations ([KPR+] and the `HAVE_STACKEFFICIENT` variant of [BCC+23]).

Implementation	additions in $\mathbb{F}_q^m$	scalar multiplications in $\mathbb{F}_q^m$
Stack-efficient	$(k/2)n^2 + (k^2 + 16.5k)n + 16k$	$16kn + 16k$
This Work	$(k^2/2 + 5.5)n^2 + (k^2/2 + 5.5)n$	$1.5n^2 + 1.5n$

The key to unlock the potential RAM savings from on-demand expansion/conversion of the public key is thus an alternative implementation of the matrix multiplications, which accesses the elements of $\dddot{\mathbf{P}}$ only once without requiring large accumulator arrays.

Our solution to this is shown in Algorithm 6. While matrix multiplication is not commutative, rewriting Line 10f. of Algorithm 4 as a sum of products of matrix elements

$$\dot{\mathbf{Y}}_{i,j,[m]} = \sum_{h,\ell} \dot{\mathbf{P}}_{h,\ell,[m]} \left(S_{i,h} \cdot S_{j,\ell}\right) \tag{11}$$

allows us to change the order of multiplications. Since expansion of $\dddot{\mathbf{P}}^{(1)}$, $\dddot{\mathbf{P}}^{(2)}$ and storage order of $\dddot{\mathbf{P}}^{(3)}$ all go along the m-axis first and $\ddot{\mathbf{S}}$ is a k by n planar matrix without the m-axis, we can work on m-element vectors at once. Any m-element vector $\dot{\mathbf{P}}_{h,\ell,[m]}$ can be efficiently obtained from a single call to Algorithm 5 respectively $\mathsf{Decode}_{\dddot{\mathbf{P}}^{(3)}}$ and requires $\hat{m}\,\mathrm{B}$. Each vector is then added on the $k \times k$ depth-vectors in $\ddot{\mathbf{Y}}$ with an individual factor determined by the product in parentheses. As MAYO operates in $\mathbb{F}_{16}$, there can be at most 15 non-zero factors. This means $15\hat{m}\,\mathrm{B}$ are sufficient to provide all pre-calculated multiples of $\dot{\mathbf{P}}_{h,\ell,[m]}$. The product in parentheses can be efficiently implemented by a 256 B table look-up using two nibbles as indices and returning another nibble.

In summary, our solution manages to implement the entire matrix multiplication with only $15\hat{m}\,\mathrm{B}$ of buffer RAM and a 256 B look-up table in flash, cf. Lines 18–33 in Algorithm 6. Hence, we estimate RAM consumption to be about 6.39 KiB for MAYO_1, 2.28 KiB for MAYO_2, 9.90 KiB for MAYO_3 and 14.3 KiB for MAYO_5. This means reductions in RAM consumption of 97% to 99% compared to the most stack-efficient existing implementation in [KPR+]. Measurements to verify these savings on actual hardware will be presented in Section 4.

3.2 Analysis of execution time

Let us now analyze the effect of our RAM optimizations on execution time. On-demand expansion of the public key comes at basically no additional cost for two reasons: First, we reimplemented matrix multiplication such that no element of $\dddot{\mathbf{P}}$ needs to be expanded twice and the elements are consumed in the order they are output by Algorithm 5. Second, our implementation of Algorithm 5 saves tail bytes exceeding the requested number of nibbles ℓ but created by AES128 due to AES block size until the next call, so AES128 is only called once for the same inputs.

For the change in matrix multiplication, we use additions, denoted A, and scalar multiplications, denoted M, in $\mathbb{F}_q^m$ as units of measurement. From Algorithm 6, we see that our implementation requires $(15\,\mathrm{M} + k^2\,\mathrm{A})$ for procedure `ADD`, which is called $n(n+1)/2$ times. The effort for procedure `ADD` can be reduced to $(3\,\mathrm{M} + (k^2 + 11)\,\mathrm{A})$, though, by optimizing Line 10 of Algorithm 6 to use scalar multiplications for $2\dot{\mathbf{P}}$, $4\dot{\mathbf{P}}$, and $8\dot{\mathbf{P}}$ and obtaining the other multiples of $\dot{\mathbf{P}}$ through additions. In comparison, for the C-implementation from [KPR+] we count $kn(n+1)/2\,\mathrm{A} + 16kn(\mathrm{M} + \mathrm{A}) + k^2 n\,\mathrm{A} + 16k(\mathrm{M} + \mathrm{A})$. Given that $n \gg k$,

Algorithm 6 RAM optimized MAYO signature verification

Input: Compact public key $\mathsf{cpk} = (\mathsf{seed}_{\mathrm{pk}}, \mathsf{P3_bytes})$
Input: Message M
Input: Signature $\mathsf{sig} = (\mathsf{salt}, \mathsf{S_bytes})$
Output: Return `true` if the signature is valid, `false` otherwise

▷ *Decode signature.* $\ddot{\mathbf{S}} \in \mathbb{F}_q^{k \times n}$ ◁
$\ddot{\mathbf{S}} \leftarrow \mathsf{Decode}_{\ddot{\mathbf{S}}}(\mathsf{S_bytes})$
▷ *Hash message.* $\dot{\mathbf{t}} \in \mathbb{F}_q^m$ ◁
$\dot{\mathbf{t}} \leftarrow \mathsf{Decode}_{\dot{\mathbf{t}}}(\mathsf{SHAKE256}(\mathsf{M} \| \mathsf{salt}, \lceil m/2 \rceil))$
▷ *Evaluate public map on signature.* $\dddot{\mathbf{Y}} \in \mathbb{F}_q^{k \times k \times m}$, $\dot{\mathbf{y}} \in \mathbb{F}_q^m$ ◁
$\dddot{\mathbf{Y}} \in \mathbb{F}_q^{k \times k \times m}, \dddot{\mathbf{Y}} \leftarrow \ddot{\mathbf{0}}$
procedure ADD(h, ℓ, $\dot{\mathbf{P}}$)
    ▷ *Precompute non-zero multiples of* $\dot{\mathbf{P}}$ *and store them in a table.* ◁
    for all $a \in \mathbb{F}_q \setminus \{0\}$ **do**
        $\dot{\mathbf{T}}_a \leftarrow a\dot{\mathbf{P}}$
    for all $i \in [k]$ **do**
        for all $j \in [k]$ **do**
            $a \leftarrow S_{i,h} \cdot S_{j,\ell}$ ▷ *Multiplication in* $\mathbb{F}_q$ *can be implemented efficiently with a 256 Byte look-up table.*
            if $a \neq 0$ **then**
                $\dddot{\mathbf{Y}}_{i,j,[m]} \leftarrow \dddot{\mathbf{Y}}_{i,j,[m]} + \dot{\mathbf{T}}_a$ ▷ *Add* $S_{i,h}S_{j,\ell}\dot{\mathbf{P}}$
$c \leftarrow 0$
▷ *Iterate over* $\dddot{\mathbf{P}}^{(1)}$ ◁
for $h = 0, \ldots, n-o-1$ **do**
    for $\ell = h, \ldots, n-o-1$ **do**
        $\dot{\mathbf{P}} \leftarrow \mathsf{AES128_CTR}(\mathsf{seed}_{\mathrm{pk}}, c, m)$ ▷ $\dot{\mathbf{P}} \leftarrow P^{(1)}_{h,\ell,[m]}$
        $c \leftarrow c + m$
        ADD(h, ℓ, $\dot{\mathbf{P}}$)
▷ *Iterate over* $\dddot{\mathbf{P}}^{(2)}$ ◁
for $h = 0, \ldots, n-o-1$ **do**
    for $\ell = n-o, \ldots, n-1$ **do**
        $\dot{\mathbf{P}} \leftarrow \mathsf{AES128_CTR}(\mathsf{seed}_{\mathrm{pk}}, c, m)$ ▷ $\dot{\mathbf{P}} \leftarrow P^{(2)}_{h,\ell,[m]}$
        $c \leftarrow c + m$
        ADD(h, ℓ, $\dot{\mathbf{P}}$)
▷ *Iterate over* $\dddot{\mathbf{P}}^{(3)}$ ◁
for $h = n-o, \ldots, n-1$ **do**
    for $\ell = n-o, \ldots, n-1$ **do**
        $\dot{\mathbf{P}} \leftarrow \dot{\mathbf{P}}^{(3)}_{h-(n-o),\ell-(n-o),[m]}$ ▷ *Decode directly from* P3_bytes.
        ADD(h, ℓ, $\dot{\mathbf{P}}$)
$\dot{\mathbf{y}} \leftarrow \mathsf{Compute_y}(\dddot{\mathbf{Y}})$
▷ *Check equality and return result* ◁
return $\dot{\mathbf{y}} == \dot{\mathbf{t}}$

Table 4: Results on the Nucleo-L4R5ZI platform obtained using the PQM4 framework. ref denotes the C-implementation in [KPR+], TW denotes This Work. Memory refers to peak stack consumption; all implementations use 8 B in the data and 16 B in the bss section. Speed displays mean, min, and max runtime over 10 executions. Symmetric indicates the fraction of runtime spent on symmetric primitives (AES and SHAKE).

	Memory [KiB]		Speed [Mcycle]		Symmetric [%]		Code [KiB]	
Scheme	ref	TW	ref	TW	ref	TW	ref	TW
mayo102	213	2.32	8.3	$11.9^{+0.1}_{-0.1}$	77	53	28.1	27.4
mayo103	165	2.52	7.0	$10.0^{+0.1}_{-0.1}$	71	49	28.0	26.7
mayo104	149	2.80	6.8	$10.3^{+0.1}_{-0.1}$	67	44	27.2	26.2
mayo105	183	3.54	8.0	$13.7^{+0.1}_{-0.1}$	58	34	29.1	27.8
mayo106	183	4.05	8.8	$16.1^{+0.2}_{-0.2}$	54	30	29.9	29.1
mayo107	188	4.54	9.9	$20.2^{+0.2}_{-0.3}$	52	26	29.9	27.2
mayo108	200	5.36	11.6	$25.5^{+0.5}_{-0.4}$	50	23	29.7	27.0
mayo109	205	6.01	12.6	$30.9^{+0.7}_{-0.6}$	49	21	30.1	27.4
mayo110	210	6.89	14.0	$36.0^{+0.5}_{-0.6}$	46	19	29.3	26.6
mayo307	477	6.11	25.1	$54.7^{+0.9}_{-0.7}$	54	27	35.2	31.1
mayo308	478	7.05	27.1	$64.9^{+0.8}_{-1.3}$	51	24	35.3	31.1
mayo309	486	8.12	29.0	$77.2^{+0.9}_{-1.5}$	50	21	36.2	32.1
mayo310	495	9.29	31.7	$92.2^{+2.0}_{-1.3}$	47	18	36.2	32.1
mayo311	504	10.6	34.0	$106.7^{+1.2}_{-1.6}$	46	16	35.8	31.7
mayo313	622	15.0	42.9	$159.3^{+1.8}_{-3.0}$	42	13	39.5	34.6

we can infer from Table 3 that our implementation requires about $n/10k$ times more scalar multiplications and $k + 11/k$ times more additions than the C-implementation from [KPR+]. For MAYO_1, MAYO_3, and MAYO_5, $n \approx 10k$, so the number of scalar multiplications does not change much, leaving only a ten to twelve times increase in number of additions. For MAYO_2, we expect twice as many scalar multiplications and about four times as many additions. Assuming approximately half of the overall runtime is spent on matrix multiplication and this time is split equally on scalar multiplications and additions, we expect the overall runtime to double or at most triple. Given that our implementation reduced RAM usage by a factor of more than thirty, the time-memory-product is expected to be significantly better than with the existing implementations, still. We will present measurements on actual hardware in Section 4.

4 Implementation results

To verify the expected RAM savings and measure the actual runtime increase in a manner comparable to existing literature, we used the PQM4 framework [KPR+] on a NUCLEO-L4R5ZI evaluation board. It features a Cortex-M4 based MCU with 640 KiB of RAM and is clocked by the framework at 24 MHz according to its documentation. The host compiling the C code and taking the measurements was a ThinkPad T580 running Ubuntu 24.04 and arm-none-eabi-gcc (15:13.2.rel1-2) 13.2.1 20231009. Table 4 shows the results for all parameter sets the framework supports on this evaluation board. Because it performs key generation and signing on the board itself, we could not use it for parameter sets where those operations, which are not affected by our optimizations, exceed the resources of the evaluation board.

Inspecting the memory columns in Table 4, we find our predictions on stack usage

in Section 3.1 being validated. The deviation of at most 1 KiB most likely results from saved registers, loop variables, etc. not considered in our predictions. The runtime increased by a factor correlated to parameter k, which is in line with our estimation in Section 3.2. For MAYO_1, where $k = 10$, our implementation trades 2.57 times the runtime for $1/30.5$ times (equals 3.28% of) the memory compared to PQM4. For MAYO_2, where $k = 4$, our implementation trades 1.51 times the runtime for $1/53.2$ times the memory (1.88%). For MAYO_3, where $k = 11$, our implementation trades 3.14 times the runtime for $1/47.5$ times the memory (2.10%). Contrary to the PQM4 implementation, the runtime for our implementation varies slightly, which is depicted in the speed column by the accuracy markers. As signature verification does not operate on secret data, this variation in runtime does not present a security issue. Finally, our implementation achieves a 3% to 12% smaller code size with an average reduction of 8%.

To be able to test our implementation for the remaining parameter sets, we wrote our own evaluation framework that performs key generation and signing on the host and only signature verification on the target platform. The results obtained from this framework can be found in Table 5. Apart from a speed-up in testing, this allowed us to evaluate different message sizes, the same message with multiple keys per parameter set, and multiple messages with the same keys. We thus created four random messages of 512 B, representing a firmware manifest file being signed, and four messages of 256 KiB, representing a firmware blob being signed directly. For each parameter set, we created eight key pairs and signed all messages with all keys. For mayo302 and all mayo5xx parameter sets, the stack usage of PQM4's C-implementation exceeded the board's available RAM during signature verification and therefore could not be tested. For the remaining mayo3xx parameter sets, the message reception buffer had to be shrunk to leave enough space for the stack, allowing only the 512 B manifest examples to be measured but not the 256 KiB firmware examples. Note that runtime differs from the values obtained with the PQM4 framework, because our framework clocks the MCU at 40 MHz and enables the flash prefetch unit to avoid wait states when executing code directly from flash. We therefore also included the parameter sets already evaluated with the PQM4 framework in our tests to ensure that results are comparable. Though we aimed to analyze all parameter sets given in [BCC+25], we could not find reduction polynomials compatible with the implementation in [KPR+] for mayo502, mayo503 and mayo504. The reduction polynomials are required for the matrices $\ddot{\mathbf{E}}_{ij}$ in Eq. (10) and are given in [BCC+25] only for the main parameter sets, but not for the additional parameter sets. Enhancing the implementation to support reduction polynomials for mayo502, mayo503 and mayo504 was out of scope of this work.

The results in Table 5 prove that signatures of MAYO parameter sets in NIST security category 5 can successfully be verified even on a microcontroller. Furthermore, our predictions on stack usage in Section 3.1 can now also be validated for MAYO_5, where the deviation of 0.8 KiB stems again most probably from saved registers and loop variables.

To visualize the substantial improvement in time-memory-product (TMP), the results from Table 5 are plotted in Figs. 2 to 4 together with isolines depicting locations of equal TMP. The plots highlight another interesting aspect: Using the implementation from [KPR+], $k = 4$ results in the best TMP for both mayo104 and mayo304, with the TMP deteriorating monotonically the further k deviates from 4 in either direction. For our implementation, the optimum k is 3, with 4 being almost as good. Again, the TMP deteriorates with increasing values of k, but even for the worst choice of k the TMP remains much better than for the implementation from [KPR+] with optimal choice of k.

5 Conclusion

We presented a memory-optimized pure-C implementation of MAYO signature verification. By expanding the public key vector-by-vector on the fly, reordering the SPS matrix product

Table 5: Results on the Nucleo-L4R5ZI platform obtained with our own framework. ref denotes the C-implementation in [KPR+], TW denotes This Work. Memory refers to peak stack consumption. Speed displays mean, min, and max runtime over 32 executions — from 8 keys and 4 messages — per message size.

	Memory [KiB]		Speed [Mcycle]			
			512 B		256 KiB	
Scheme	ref	TW	ref	TW	ref	TW
mayo102	213	2.39	8.1	$14.9^{+0.1}_{-0.1}$	29.3	$36.0^{+0.1}_{-0.1}$
mayo103	165	2.59	6.8	$12.3^{+0.1}_{-0.2}$	27.9	$33.4^{+0.1}_{-0.1}$
mayo104	149	2.88	6.7	$12.4^{+0.1}_{-0.1}$	27.8	$33.5^{+0.2}_{-0.2}$
mayo105	183	3.61	7.9	$16.1^{+0.3}_{-0.2}$	29.0	$37.2^{+0.3}_{-0.3}$
mayo106	183	4.12	8.7	$18.5^{+0.3}_{-0.6}$	29.8	$39.6^{+0.3}_{-0.2}$
mayo107	188	4.73	9.7	$22.6^{+0.5}_{-0.6}$	30.8	$43.7^{+0.6}_{-0.5}$
mayo108	200	5.43	11.4	$28.6^{+0.4}_{-0.7}$	32.5	$49.6^{+0.5}_{-0.5}$
mayo109	205	6.20	12.5	$33.3^{+0.7}_{-0.7}$	33.6	$54.4^{+0.9}_{-0.5}$
mayo110	210	7.04	13.7	$39.6^{+0.9}_{-1.4}$	34.8	$60.6^{+0.7}_{-1.0}$
mayo302		3.06		$48.4^{+0.2}_{-0.3}$		$69.5^{+0.3}_{-0.3}$
mayo303	498	3.36	21.2	$39.8^{+0.3}_{-0.4}$		$61.1^{+0.3}_{-0.4}$
mayo304	445	3.78	21.6	$39.9^{+0.3}_{-0.4}$		$61.1^{+0.4}_{-0.5}$
mayo305	497	4.69	22.3	$47.8^{+0.5}_{-0.5}$		$68.8^{+0.9}_{-0.8}$
mayo306	476	5.41	22.9	$54.0^{+1.1}_{-0.8}$		$75.1^{+0.8}_{-1.6}$
mayo307	477	6.23	24.6	$61.9^{+0.8}_{-1.1}$		$83.0^{+1.1}_{-0.9}$
mayo308	478	7.16	26.5	$72.2^{+1.3}_{-1.5}$		$93.5^{+1.6}_{-2.1}$
mayo309	486	8.23	28.6	$84.5^{+1.4}_{-1.8}$		$105.7^{+1.3}_{-1.9}$
mayo310	495	9.40	31.1	$98.8^{+2.0}_{-2.2}$		$119.8^{+1.3}_{-2.8}$
mayo311	504	10.7	33.4	$114.9^{+2.0}_{-2.1}$		$136.2^{+2.1}_{-3.5}$
mayo313	622	15.1	42.2	$168.5^{+3.8}_{-3.2}$		$189.2^{+2.4}_{-2.9}$
mayo505		5.72		$110.7^{+1.4}_{-1.3}$		$131.6^{+1.5}_{-1.1}$
mayo506		6.63		$124.1^{+1.8}_{-1.4}$		$144.7^{+2.3}_{-1.8}$
mayo507		7.67		$136.1^{+2.5}_{-2.2}$		$157.5^{+1.0}_{-1.3}$
mayo508		8.88		$155.2^{+4.0}_{-2.1}$		$176.0^{+2.5}_{-2.4}$
mayo509		10.2		$181.0^{+2.7}_{-3.4}$		$202.1^{+2.7}_{-2.4}$
mayo510		11.7		$206.8^{+3.2}_{-5.0}$		$228.4^{+4.7}_{-4.6}$
mayo511		13.4		$238.9^{+4.6}_{-4.5}$		$260.1^{+3.0}_{-4.5}$
mayo512		15.1		$273.9^{+4.6}_{-5.3}$		$295.2^{+4.3}_{-2.9}$
mayo514		19.1		$354.3^{+6.1}_{-5.7}$		$375.0^{+5.8}_{-6.4}$
mayo515		23.4		$449.5^{+19.3}_{-14.8}$		$465.1^{+13.7}_{-14.8}$

so that each public-key vector is consumed only once, and adding precomputed multiples of each vector instead of accumulating vectors before multiplication, our implementation cuts verification RAM by up to two orders of magnitude while maintaining, and typically reducing, code size. Concretely, for MAYO_1 this reduces verification RAM from over 210 KiB to under 7 KiB, for MAYO_2 from 149 KiB to 2.8 KiB, and for MAYO_3 from 504 KiB to 10.6 KiB.

The key trade-off insight is that on-demand public-key expansion becomes practical precisely when the matrix multiplication is restructured to touch each coefficient exactly once, which trades a modest, k-correlated runtime increase for a dramatic reduction in RAM. This breaks the coupling between key size and RAM usage in UOV-style verification, and finally enables verifying MAYO signatures on smart cards, sensor nodes, and small microcontrollers with only a few kilobytes of RAM. At these RAM levels, MAYO verification becomes competitive with lattice-based schemes on the same class of devices, while retaining MAYO's advantages of compact signatures and simple arithmetic over $\mathbb{F}_{16}$.

RAM reduction for key generation and signing. The MAYO key generation involves computing $\ddot{\mathbf{O}}^T\ddot{\mathbf{P}}_i^{(1)}\ddot{\mathbf{O}}$ and $\ddot{\mathbf{O}}^T\ddot{\mathbf{P}}_i^{(2)}$ (Algorithm 4, Line 16 in [BCC+25]) for all $i \in [m]$. This is similar to the matrix multiplications in the loop Algorithm 4, Line 10. Hence, similar techniques as described here can be applied to avoid expanding the public key matrices $\ddot{\mathbf{P}}_i^{(1)}$ and $\ddot{\mathbf{P}}_i^{(2)}$ completely in RAM during key generation. This is a straightforward adaptation of the methods described in this work and we leave an implementation and exact evaluation as future work. To see how our techniques can be applied to signing with MAYO, we use the description of the signing process in [BCC+24]. As described in [BCC+24], Algorithm 2, Lines 9–11, signing with MAYO requires computing $\ddot{\mathbf{V}}((\ddot{\mathbf{P}}_i^{(1)} + \ddot{\mathbf{P}}_i^{(1)T})\ddot{\mathbf{O}} + \ddot{\mathbf{P}}_i^{(2)})$ and $\ddot{\mathbf{V}}\ddot{\mathbf{P}}_i^{(1)}\ddot{\mathbf{V}}^T$ for all $i \in [m]$ and a randomly chosen matrix $\ddot{\mathbf{V}} \in \mathbb{F}_q^{k\times(n-o)}$. Again, it seems straightforward to apply our techniques in this context and we leave this as future work.

Transfer to other signature schemes. The signature scheme UOV [BCD+25] is an instance of the classic unbalanced oil and vinegar signature scheme originally proposed in [KPG99] and described in Section 2.2. Like MAYO, it has been submitted to the NIST standardization process for additional post-quantum signature schemes [NIS]. Unlike MAYO, it does not use specific techniques such as the "whipping up" to reduce the size of the public key. However, it does use the same pseudo-random expansion of the public key matrices as MAYO, specifically using AES in counter mode. Even the order in which the coefficients are generated is the same as in MAYO. Hence, the techniques described in this work to reduce RAM consumption during MAYO signature verification can be applied to UOV signature verification as described in [BCD+25], too. We leave an implementation and exact evaluation similar to the results for MAYO presented in Section 4 as future work. SNOVA [WCD+25] and QR-UOV [FIH+25] are two further UOV-based signature schemes submitted to the NIST standardization process for additional post-quantum signature schemes [NIS]. Transferring the techniques described in this paper to these schemes would also be interesting future work.

Acknowledgments

This work was partly funded by the German Federal Ministry of Education and Research in the project Quoryptan through grant number 16KIS2033.

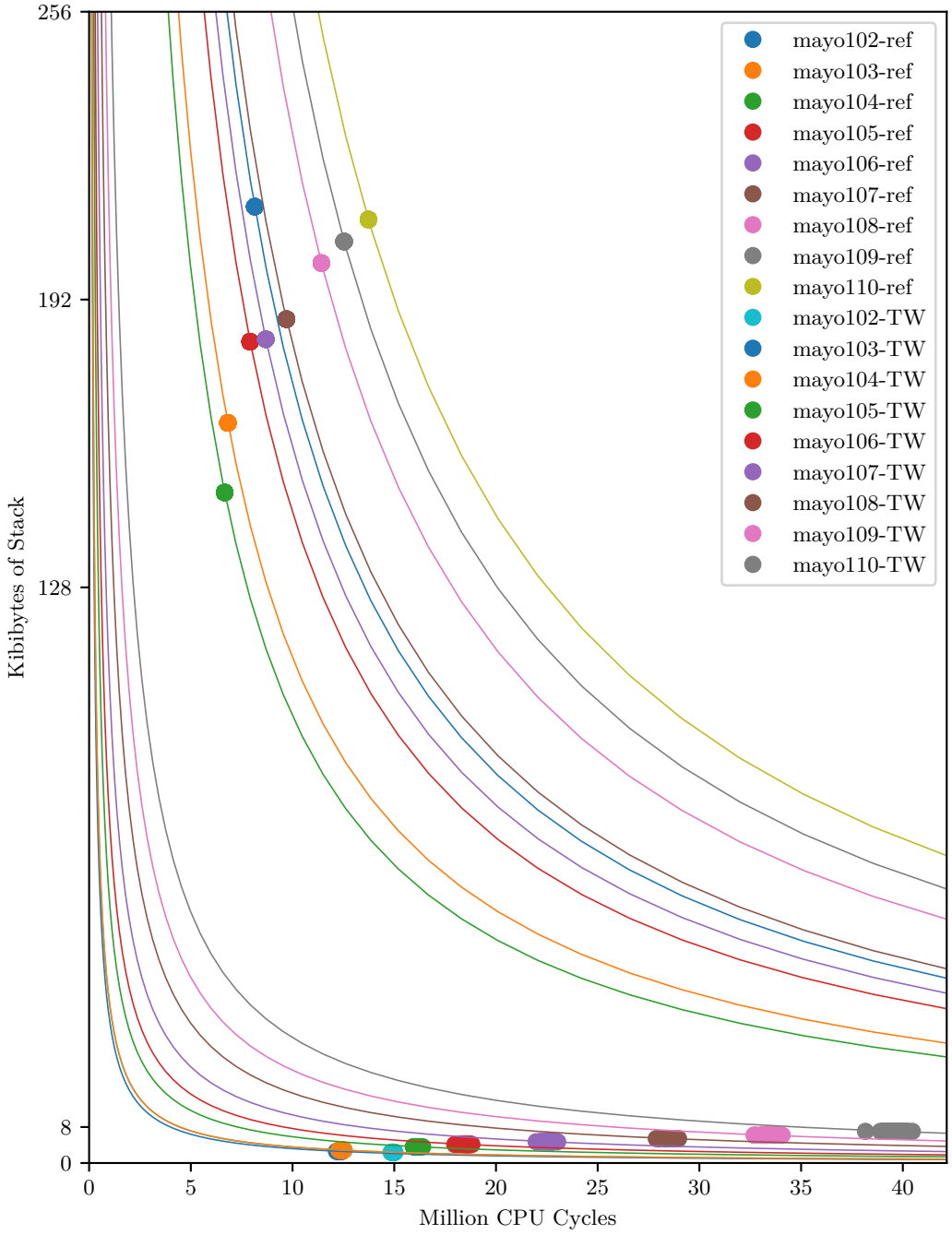


Figure 2: Time-memory-product of PQM4's C-implementation [KPR+] (denoted "ref") and our work ("TW") using 512 B messages for parameter sets of NIST security category 1. Note that for our work the markers for mayo103 and mayo104 are overlapping.

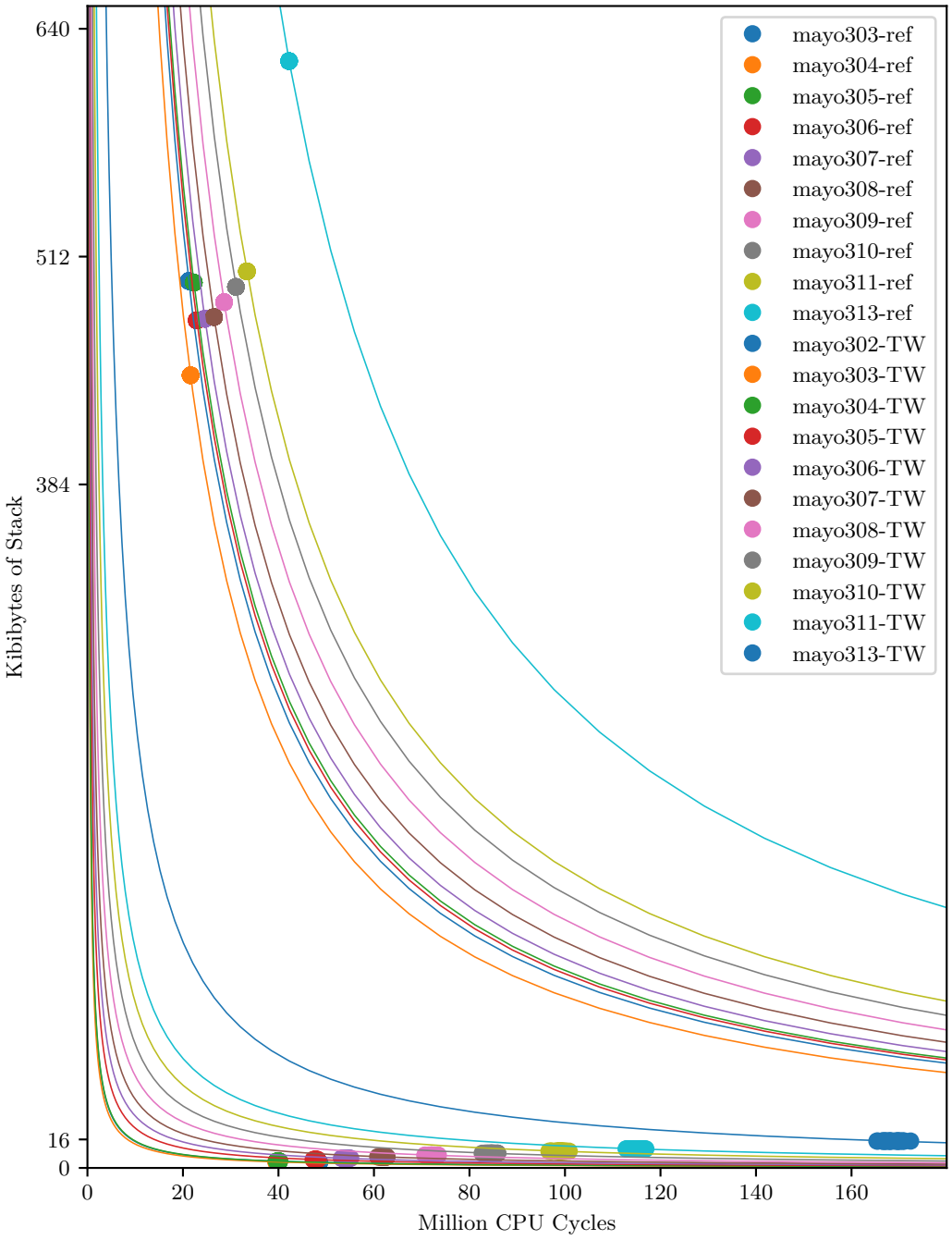


Figure 3: Time-memory-product of PQM4's C-implementation [KPR+] (denoted "ref") and our work ("TW") using 512 B messages for parameter sets of NIST security category 3. Note that mayo302 is not depicted for PQM4 as it exceeded the available RAM. Furthermore, for our work the markers for mayo302 and mayo305 are overlapping, as well as the markers for mayo303 and mayo304.

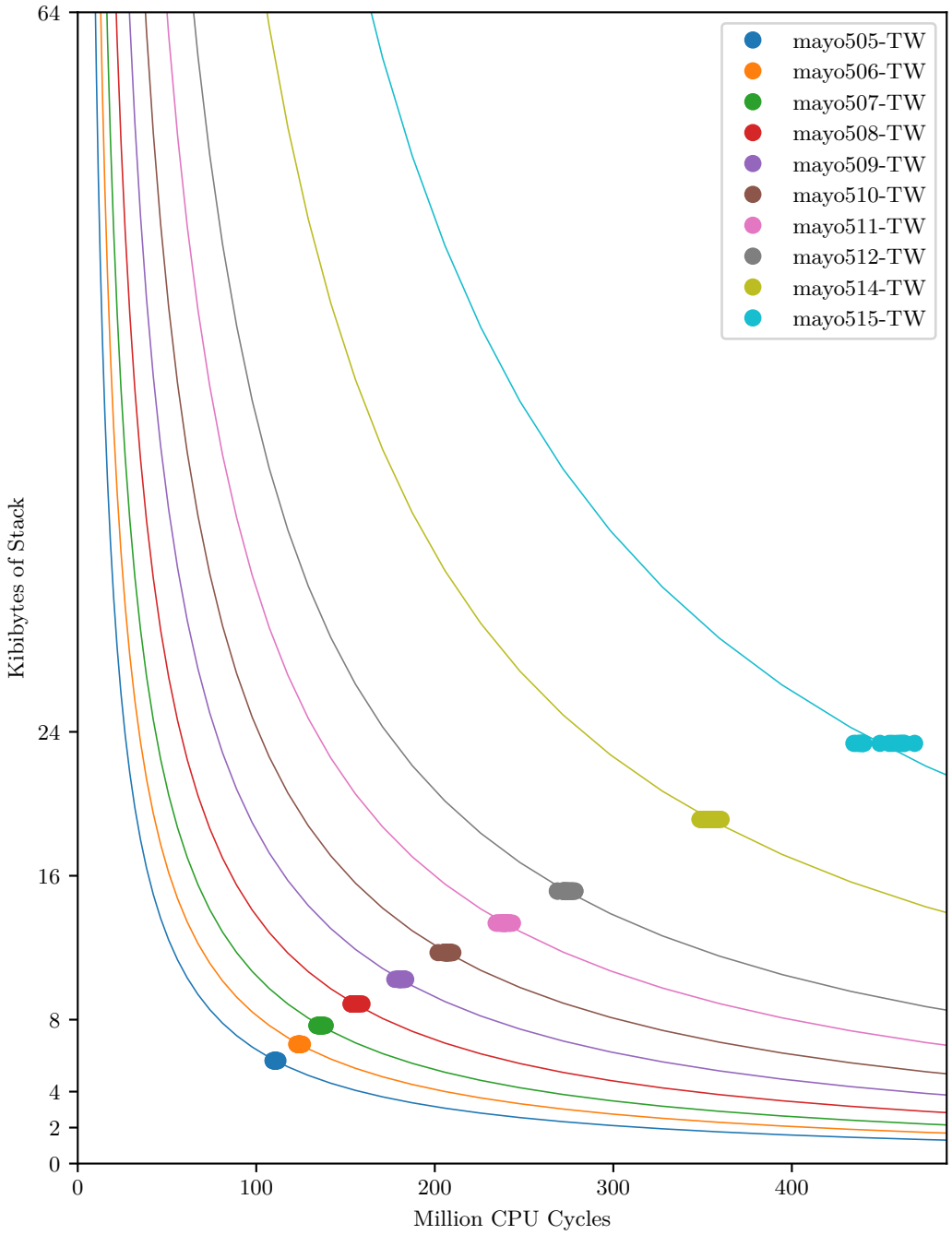


Figure 4: Time-memory-product of our work ("TW") using 512 B messages for parameter sets of NIST security category 5. Note that PQM4's C-implementation [KPR+] exceeded the available RAM for all parameter sets in this category.

References

[BCC+23] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO C implementation, June 2023. "commit id 4b7cd94c96b9522864efe40c6ad1fa269584a807". URL: https://github.com/PQCMayo/MAYO-C.

[BCC+24] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: Optimized implementations for AVX2 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):252–275, Mar. 2024. URL: https://tches.iacr.org/index.php/TCHES/article/view/11427, doi:10.46586/tches.v2024.i2.252-275.

[BCC+25] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO round 2 version. Technical report, NIST, 2025. URL: https://pqmayo.org/assets/specs/mayo-round2.pdf.

[BCD+25] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. UOV: Unbalanced oil and vinegar algorithm specifications and supporting documentation version 2.0. Technical report, NIST, 2025. URL: https://www.uovsig.org/.

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

[FIH+25] Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Haruhisa Kosuge Tsuyoshi Takagi, Kimihiro Yamakoshi, Satoshi Nakamura Rika Akiyama, Shingo Orihara, and Koha Kinjo. QR-UOV. Technical report, NIST, 2025. URL: https://info.isl.ntt.co.jp/crypt/qruov/.

[HSK+23] Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. Whipping the MAYO signature scheme using hardware platforms. Cryptology ePrint Archive, Paper 2023/1267, 2023. URL: https://eprint.iacr.org/2023/1267.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 206–222, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. commit id a24bb4b. URL: https://github.com/mupq/pqm4.

[KS98] Aviad Kipnis and Adi Shamir. Cryptanalysis of the oil and vinegar signature scheme. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 257–266. Springer, 1998. doi:10.1007/BFb0055733.

[NIS] Post-quantum cryptography: Additional digital signature schemes. URL: https://csrc.nist.gov/projects/pqc-dig-sig.

[Pat95] Jacques Patarin. The oil and vinegar signature scheme. In *Dagstuhl Workshop on Cryptography*, 1995.

[PTBW11] Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. Small public keys and fast verification for $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic public key systems. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 475–490, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

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

[Str24] Michael Helmut Streibl. Design and analysis of the MAYO signature scheme with focus on hardware platforms. Master's thesis, TU Graz, 2024. URL: https://repository.tugraz.at/publications/c1kgw-ph417.

[WCD+25] Lih-Chung Wang, Chun-Yen Chou, Jintai Ding, Jan Adriaan Leegwater Yen-Liang Kuan, Ming-Siou Li, Po-En Tseng Bo-Shu Tseng, and Chia-Chun Wang. SNOVA, proposal for NISTPQC: Additional digital signature schemes, version 2.0. Technical report, NIST, 2025. URL: https://snova.pqclab.org/.