Bitsliced Jasmin Implementation of the Mayo Signature Scheme

Samyuktha M[1,2], Pallavi Borkar[1], and Chester Rebeiro[1]

[1] Indian Institute of Technology, Madras
[2] Society for Electronic Transactions and Security, Chennai
samyuktha@setsindia.net
{pallavi,chester}@cse.iitm.ac.in

Abstract. We present a Jasmin bitsliced implementation of Mayo2, a multivariate quadratic(MQ) based signature scheme. Mayo overcomes the disadvantage of the Unbalanced oil and vinegar(UOV) scheme by whipping the UOV map to produce public keys of sizes comparable to ML-DSA. The Round 2 C reference implementation employs a nibble-sliced representation of GF(16), packing two field elements per byte across four 64-bit limbs. In contrast, our Jasmin implementation adopts a bitsliced representation, decomposing each GF(16) element across four separate 64-bit bit planes, enabling 64 simultaneous field operations. Our Jasmin implementation of Mayo2 takes 10.55 ms for keygen, 8.21 ms for sign, 0.17 ms for verify based on an average of 1000 runs of the implementation on a 13th Gen Intel Core i7-13700 processor (max turbo frequency 5.2 GHz) with 16 GB RAM. To this end, we have a multivariate quadratic based signature implementation that is amenable for verification of constant-time, correctness, proof of equivalence properties using Easycrypt. Subsequently, the results of this endeavor can be extended to other MQ based schemes including UOV.

Keywords: Formal Verification · Post Quantum Cryptography · Mayo · Jasmin · Easycrypt

1 Introduction

With the widespread migration of security protocols to post-quantum, various efficient and architecture-specific optimized implementations have emerged. These implementations can be considerably complex and their correctness not easily verifiable due to larger input spaces. Thus, these implementations need to be proven correct and equivalent to their corresponding algorithmic specification in order to achieve the high levels of assurance needed for critical embedded system applications like root of trust, remote attestation and secure communication.

In the context of the NIST PQC competition, it is essential to formally investigate the security of the submissions and their implementations against side-channel attacks. These attacks can compromise a mathematically strong algorithm by exploiting information leaked through side channels such as power

consumption and electromagnetic radiation of the device executing the cryptographic operation. A potent side-channel variant called the timing attack utilizes execution differences to reveal information about the secret key. Thus, it is crucial not only to verify the correctness of the cryptographic implementation but also ensure constant time execution.

Multivariate cryptographic schemes based on the oil and vinegar (OV) problem have been well studied and are of interest to the research community as they offer shorter signatures and faster verification times. With NIST standardizing ML-DSA, ML-KEM and an additional call for proposals to focus on signatures based on hardness other than lattices, evaluation of these schemes is relevant.

We target Mayo2 [7], a multivariate quadratic (MQ) signature scheme submitted to the NIST's additional call for post-quantum signature proposals. The round 2 C reference implementation of Mayo2 employs a nibble-sliced representation of GF(16), where two field elements are packed per byte and vectors of m=64 elements span four 64-bit limbs [8], [9], [10]. Our work departs from this by implementing Mayo2 in Jasmin using a fully bitsliced GF(16) representation, where each of the four bits of a field element occupies a separate 64-bit register. This enables 64 simultaneous GF(16) operations and produces code that is directly amenable to machine-checked proofs of functional correctness and constant-time execution in EasyCrypt.

We plan to carry out the following three-phased approach: (i) formalising the specification of the signature scheme in Easycrypt (ii) developing bitsliced Jasmin implementation of Mayo2, (iii) Proofs for functional correctness of the Jasmin implementation with respect to the specification. In line with the strategy, we have implemented the Keygen, Sign and Verify primitives of a MQ based signature scheme (Mayo2) in Jasmin, the specifics of which will be detailed in the later sections. Hence we make the following contributions in this article:

1. Bitsliced Jasmin implementation of GF(16) arithmetic, contrasting with the nibble-sliced representation of the Round 2 C reference
2. Bitsliced implementation of the linear algebra operations over GF(16)
3. Bitsliced gaussian elimination implementation to solve the linear system of equations arising during signing
4. Jasmin implementation of the Keygen, Sign and Verify primitives of Mayo2 (Round 2)

The code is made available at
`https://github.com/samyukthasets/Round-2-Mayo-Jasmin`

Related Work. Formally verified implementations, often called high-assurance cryptographic software, exist for classical algorithms and have been widely adopted in libraries. It is imperative to extend these notions to post-quantum schemes for arguing robust security guarantees. Frameworks like Easycrypt [6] utilize interactive proof assistant based on probabilistic relational Hoare logic and have demonstrated effectiveness in the formal verification process. In their research on a formally verified implementation of SHA-3, Almeida et al. [3] advocate for mechanized proofs of functional correctness, provable security, and resistance to

timing attacks, employing a toolchain that combines Jasmin [4] and Easycrypt. Furthermore, Almeida et al in [5] present a formally verified proof of the functional correctness and IND-CCA security of ML-KEM, the Kyber-based Key Encapsulation Mechanism (KEM).

Beullens et al. [8] present optimized implementations of Mayo for AVX2 and Arm Cortex-M4. The authors show that switching the internal representation of GF(16) elements from bitsliced to nibble-sliced combined with the Method of Four Russians (M4R) for matrix multiplication yields substantial performance gains. On AVX2, they use vpshufb-based lookups with single 4-bit indices, so each lookup returns the result of two field multiplications, 32 parallel lookups per instruction therefore yield 64 multiply-accumulate operations per vpshufb, with a theoretical peak of 96 multiply-accumulate operations per cycle on Ice Lake. Constant-time execution of their AVX2 implementations is also empirically validated using Valgrind and the ctgrind method, rather than through machine-checked proof.

Our Jasmin implementation originated from a bitsliced Round 1 implementation of Mayo2, rather than porting to the nibble-sliced representation adopted in Round 2. We retained the bitsliced internal representation and instead ensured that our serialized key and signature outputs match the nibble-sliced Round 2 wire format exactly, thereby preserving interoperability with Round 2 implementations.

However, formally verified implementations of multivariate quadratic based schemes remain largely unexplored. To the best of our knowledge, no prior work presents a formally verified implementation of a Mayo or UOV based signature scheme. This work addresses that gap by providing the first Jasmin implementation for Mayo 2 based on NIST additional call for proposals round 2 specification.

2 Preliminaries

Multivariate Quadratic based Schemes. Signature schemes based on the hardness of multivariate quadratic(MQ) problem offer shorter signature sizes and faster verification times. UOV is one such well-studied MQ based scheme but it suffers from the disadvantage of larger public key sizes. Mayo can be considered as a variant of UOV that offers public keys whose size are comparable to that of ML-DSA. However, it is to be noted that the cryptographic security of Mayo depends on the security of UOV. The Mayo2 Round 2 parameters used in this work are m = 64, n = 81, k = 4, o = 17, with a secret key of 24 bytes, public key of 4912 bytes and signature of 186 bytes.

Mayo. The keygen(1^λ) of Mayo starts by sampling a random matrix $\mathbf{O}$ and the column space of $\begin{bmatrix}\mathbf{O}\\ I_o\end{bmatrix}$ forms the secret oil space $\mathcal{O}$ with $\dim(O) = o$; number of variables 'n', number of equations 'm' and $o < m$. A random multivariate

quadratic map $\mathcal{P}(x)$: $\mathbb{F}_q^n \to \mathbb{F}_q^m$ which vanishes on this subspace $\mathcal{O}$ is generated as the public key.

In a smaller subspace where o $<$ m, with larger probability the system will not have any solution. Addressing this during sign, a parameter 'k' is fixed with ko $\geq$ m, the UOV map $\mathcal{P}(x)$: $\mathbb{F}_q^n \to \mathbb{F}_q^m$ is whipped to $\mathcal{P}^*(x_1, x_2, ..x_k)$: $\mathbb{F}_q^{kn} \to \mathbb{F}_q^m$ using public matrices called emulsifier maps E_{ij} as

$$\mathcal{P}^*(x_1, \ldots, x_k) := \sum_{i=1}^{k} E_{ii} P(x_i) + \sum_{i=1}^{k} \sum_{j=i+1}^{k} E_{ij} P'(x_i, x_j)$$

With ko degrees of freedom, a solution can be found with overwhelming probability. The whipped map $\mathcal{P}^*$ is constructed in such a way that it vanishes on the subspace $O^k = (o_1, ..., o_k)$, $\forall$ i $\in \{1, \ldots, k\}$, $o_i \in \mathcal{O}$. The target t is computed as $\mathcal{H}(M||Salt)$. The signer at random chooses $(v_1, ..., v_k) \in \mathbb{F}_q^{kn}$ and solves for $(o_1, ..., o_k) \in O_k$ such that $\mathcal{P}^*(v_1 + o_1, ..., v_k + o_k) =$ t. The solution ($s_i = v_i + o_i$ where i $\in \{1, \ldots, k\}$) along with the salt is shared as signature. For verification, message M with the salt is hashed to obtain t and the signature is accepted if and only if

$$\mathcal{P}^*(s_1, \ldots, s_k) = t.$$

3 Implementation Details

The implementation targets Mayo2 as specified in the NIST Round 2 submission. The Round 2 C reference implementation adopts a nibble-sliced representation of GF(16), where two field elements are packed per byte and a vector of $m = 64$ field elements spans four 64-bit limbs (256 bits total). Scalar-vector multiplication `gf16v_mul_u64` computes a double-and-add scalar multiplication in parallel across all 16 packed elements of a limb using only shifts, masks and XOR. Our Jasmin implementation departs from this by adopting a fully bitsliced representation throughout, where each of the four bits of a GF(16) element occupies a separate 64-bit register (bit plane), enabling 64 simultaneous field multiply-accumulate operations.

Finite Field Implementation. At the bottom is the implementation of the GF(16) arithmetic. The Round 2 C reference (as built with `ENABLE_PARAMS_DYNAMIC=ON` matching our benchmarked configuration) stores a vector of $m = 64$ GF(16) elements in nibble-sliced form across four 64-bit limbs with 16 elements packed per limb. Scalar-vector multiplication (`gf16v_mul_u64`) computes across all 16 packed elements of a limb in parallel. The limb is repeatedly doubled in GF(16) by extracting each nibble's top bit via the mask `0x8888888888888888` clearing it, shifting the limb left by one and XORing back a correction term $(a_msb \gg 3) \cdot 3$ encoding reduction modulo $x^4 + x + 1$. This is then conditionally accumulated into the result based on each of the four bits of a using only shifts, masks, multiplication by 0 or 1 and XOR.

In our Jasmin implementation, the same vector of 64 GF(16) elements is stored in bitsliced form across four 64-bit registers, one per bit plane of the field

element. The core routine `bitsliced_64_vec_mul_add` multiplies such a vector by a scalar by extracting each of the four bits of scalar as a sign-extended all-ones or all-zeros mask (via `-(a & 1)`, `-(a » 1 & 1)`) and conditionally XORing the input bit-planes into an accumulator with modular reduction with respect to x^4+x+1 folded in. A single call to this routine performs 64 GF(16) multiply-accumulate operations simultaneously with no memory table accesses. Where the reference implementation's `gf16v_mul_u64` keeps all 16 elements of a limb packed together and processes them via repeated word-level doubling, our approach instead separates each element's bits across four independent registers, computing all 64 multiply-accumulates through direct bitwise selection rather than iterative doubling.

A non-sliced scalar GF(16) implementation (`gf16_mul` in `ffarith.jazz`) is additionally provided for operations where the parallelism of bitslicing is not exploitable.

Linear Algebra Operations. This layer implements the matrix-matrix and matrix-vector products required by KeyGen, Sign, and Verify. KeyGen uses the bitsliced routine `bitsliced_64_vec_mul_add` (via `process_k` and `process_k_trans` in `expand_sk.jazz` and `mayo2_keypair_NIST.jazz`) to compute $P^{(1)} \cdot O$ and the subsequent $O^T \cdot (P^{(1)} \cdot O + P^{(2)})$ product that yields $P^{(3)}$. Sign uses a related family of 32-bit-limb bitsliced multiply-accumulate routines (`bitsliced_m_vec_mul_add_2`, `bitsliced_m_vec_mul_add_slice`) within higher-level routines such as `P1_times_Vt` and `mul_add_mat_x_bitsliced_m_mat` to construct the whipped linear system, and within `echelon_form.jazz` for Gaussian elimination. Matrix addition of byte-encoded vinegar/oil components (`mat_add`) is performed element-wise via scalar `gf16_add` as this step operates on already decoded data rather than the large bitsliced matrices.

SHAKE256 Implementation. The Mayo specification uses SHAKE256 at multiple points: expanding the secret key seed to the public key seed bytes, generating the matrix **O** and hashing the message to produce the digest. The SHAKE256 Jasmin implementation available in the libjade library [1] has been adapted to suit the required input and output lengths of each call. Wrapper functions have been written in Jasmin accordingly.

PK-PRF Implementation. An AES-128-CTR based pseudorandom function (PRF) is used, as specified in the Round 2 submission, to expand the block matrices P_1 and P_2 in the public key. The PRF is constructed in Jasmin on top of the AES-128-CTR implementation available from the Formosa-Crypto project [2]. A counter initialised to zero is successively encrypted under AES-128 and the outputs are concatenated to fill the required buffer. The AES-128-CTR implementation uses AES-NI instructions for its round computations, ensuring constant-time execution.

Randombytes Generation. The random bytes for the initial seed_sk and the salt for randomising signatures are using the #randombytes primitive in Jasmin which derives random bytes from the linux CSPRNG /dev/urandom.

Mayo Primitives. All three primitives are implemented in Jasmin using the bit-sliced arithmetic described above.

Keygen: The compact secret key seed is expanded via SHAKE256 to produce the oil space matrix $\mathbf{O}$ and the public key seed. The public key matrices P_1, P_2 are expanded using the AES-128-CTR based PK-PRF, and P_3 is computed from P_1, P_2, and $\mathbf{O}$.

Sign: The UOV map is whipped to the larger map $\mathcal{P}^*$ using the emulsifier maps E_{ij}. Vinegar values $(v_1, \ldots, v_k)$ are sampled at random, and the resulting linear system is solved for the oil variables $(o_1, \ldots, o_k)$ using Gaussian elimination, implemented in bit-sliced form. The final signature $(s_i = v_i + o_i)$ is assembled using the scalar GF(16) arithmetic.

Verify: The signature is re-evaluated under $\mathcal{P}^*$ and accepted if and only if $\mathcal{P}^*(s_1, \ldots, s_k) = t$, where $t = \mathcal{H}(M \| \mathsf{Salt})$.

Benchmarking The Mayo2 parameters according to NIST Round 2, as elaborated in Section 2, utilized in our implementation are $m = 64$, $n = 81$, $k = 4$, $o = 17$, with a secret key size of 24B, public key size of 4912B, and signature size of 186B.

Table 1 reports the average execution time over 1000 runs of each primitive, measured on a 13th Generation Intel Core i7-13700 processor (16 cores, 24 threads) with a maximum turbo frequency of 5.2 GHz and 16 GB RAM.

The Jasmin implementation was compiled with the Jasmin Compiler, version 2025.06.3 using the `-lea` flag, which is linked together with the test harness using `gcc -g -O3 -I`.

The Round 2 C Reference implementation was built from the official NIST submission package with `-DMAYO_BUILD_TYPE=ref -DCMAKE_BUILD_TYPE=Release`, yielding `-O3 -DNDEBUG`; the build system additionally applies `-march=native` for GCC on x86-64, targeting the exact benchmarking CPU's instruction set. The `ref` build type sets `ENABLE_PARAMS_DYNAMIC=ON` by default, selecting the generic, runtime-parameterized arithmetic path (`arithmetic_dynamic.h`, using `gf16v_mul_u64`) rather than the compile-time-fixed path (`arithmetic_fixed.h`, using `mul_table`). Two configurations were built with this same optimization level, overriding the `ENABLE_AESNI` flag (`OFF` by default for `ref` builds) explicitly: `-DENABLE_AESNI=ON` and `-DENABLE_AESNI=OFF`, corresponding to the two "Nibble-sliced C Ref" columns in Table 1.

We plan to perform detailed benchmarking of the Jasmin implementation of Mayo2 against the C Reference implementation with and without AES-NI enabled over 1,00,000 runs.

Table 1. Execution time for Jasmin and Reference Implementations (ms).

Primitive	Bitsliced Jasmin	Nibble-sliced C Ref (AES-NI Disabled)	Nibble-sliced C Ref (AES-NI Enabled)
KeyGen	10.55	4.06	2.94
Sign	8.21	3.07	2.80
Verify	0.17	1.66	1.87

4 Specification and Proof of Correctness in Easycrypt

Finite Field Arithmetic. We specify the operations of GF(16) addition and multiplication within the EasyCrypt proof assistant, with respect to the non-sliced, scalar GF(16) arithmetic (`gf16_mul` in `ffarith.jazz`), rather than the bitsliced routines described above. Field elements are represented as 4-tuple boolean values, encoding polynomials of degree at most three over GF(16). The addition operation is defined as component-wise XOR, corresponding to finite field addition. Finite field multiplication is implemented through the simplified expression, performing polynomial multiplication followed by modular reduction with respect to the irreducible polynomial x^4+x+1. This formalization facilitates verification of algebraic correctness and structural properties of GF(16)-based cryptographic components.

The proof establishes a functional equivalence between the Jasmin procedure and the abstract addition and multiplication over $\mathbb{F}_{2^4}$ in the specification by showing that the output bits of the Jasmin procedure correspond to the expected expression derived from polynomial multiplication modulo $x^4 + x + 1$. The proof makes use of lemmas relating word-level bit manipulation to field-level logic and relies on axioms capturing the semantics of Jasmin bitwise operations.

The EasyCrypt specification and proof scripts are not currently included in the public GitHub repository and will be made available in a future release.

5 Future Work

We plan to (i) provide a specification of other building blocks including bitsliced finite field arithmetic, linear algebra operations, keygen sign and verify primitived (ii) formally verify the Jasmin implementation for functional correctness (iii) formally verify the implementation for constant-time execution and memory safety through logically reasoned machine-checked proofs using Easycrypt at the software level.

References

1. `https://github.com/formosa-crypto/libjade`, libjade
2. `https://formosa-crypto.org/news/2022-06-07/sibenik`, formosa Crypto

3. Almeida, J.B., et. al.: Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In: Proceedings of the 2019 ACM SIGSAC Conference (2019)
4. Almeida, J.B., Barbosa, M.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823 (2017)
5. Almeida, J.B., Barbosa, M., Barthe, G.: Formally verifying kyber episode iv: Implementation correctness (2023)
6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.Y.: EasyCrypt: A Tutorial, pp. 146–166. Springer International Publishing, Cham (2014)
7. Beullens, W.: Mayo: Practical post-quantum signatures from oil-and-vinegar maps. In: International Conference on Selected Areas in Cryptography. pp. 355–376. Springer (2021)
8. Beullens, W., Campos, F., Celi, S., Hess, B., Kannwischer, M.J.: Nibbling MAYO: Optimized implementations for AVX2 and Cortex-M4. vol. 2024, pp. 252–275 (2024). https://doi.org/10.46586/tches.v2024.i2.252-275, https://eprint.iacr.org/2023/1683
9. Beullens, W., Campos, F., Celi, S., Hess, B., Kannwischer, M.J.: MAYO: Specification document. Nist pqc digital signature on-ramp, round 2 specification, National Institute of Standards and Technology (2025), https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/mayo-spec-round2-web.pdf
10. MAYO Team: MAYO round 2 reference implementation. NIST PQC Digital Signature On-Ramp, Round 2 submission package (2025), available at https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/round-2/spec-files/mayo-round2-submission.zip