Software Toolkit for HFE-based Multivariate Schemes

Jean-Charles Faugère, Ludovic Perret, Jocelyn Ryckeghem

Software Toolkit for HFE-based Multivariate Schemes

Jean-Charles Faugère[1,2], Ludovic Perret[1,2] and Jocelyn Ryckeghem[2]

[1] CryptoNext Security
[2] Sorbonne Université, CNRS, INRIA, LIP6, Équipe PolSys, F-75005 Paris, France

jcf@cryptonext-security.com, ludovic.perret@cryptonext-security.com,
jocelyn.ryckeghem@lip6.fr

Abstract. In 2017, NIST shook the cryptographic world by starting a process for standardizing post-quantum cryptography. Sixty-four submissions have been considered for the first round of the on-going NIST Post-Quantum Cryptography (PQC) process. Multivariate cryptography is a classical post-quantum candidate that turns to be the most represented in the signature category. At this stage of the process, it is of primary importance to investigate efficient implementations of the candidates. This article presents `MQsoft`, an efficient library which permits to implement `HFE`-based multivariate schemes submitted to the NIST PQC process such as G*e*MSS, `Gui` and DualModeMS. The library is implemented in `C` targeting Intel 64-bit processors and using `avx2` set instructions. We present performance results for our library and its application to G*e*MSS, `Gui` and DualModeMS. In particular, we optimize several crucial parts for these schemes. These include root finding for `HFE` polynomials and evaluation of multivariate quadratic systems in $\mathbb{F}_2$. We propose a new method which accelerates root finding for specific `HFE` polynomials by a factor of two. For G*e*MSS and `Gui`, we obtain a speed-up of a factor between 2 and 19 for the keypair generation, between 1.2 and 2.5 for the signature generation, and between 1.6 and 2 for the verifying process. We have also improved the arithmetic in $\mathbb{F}_{2^n}$ by a factor of 4 compared to the `NTL` library. Moreover, a large part of our implementation is protected against timing attacks.

Keywords: `MQsoft` · efficient software implementation · constant-time · `HFEv-` · G*e*MSS · `Gui` · DualModeMS · root finding · binary fields

Introduction

The recent progress on the development of quantum computers has motivated NIST [39] to start a standardization process for post-quantum cryptography. In this paper, we are interested in the category of multivariate signature schemes. The choice of the best candidate is based on its security and its performance. At this stage, the security parameters of the candidates are already fixed, but the implementations can be improved. In this article, we study the efficient implementation of multivariate schemes.
We present here software tools that allow the efficient implementation of `HFE`-based schemes (using arithmetic in $\mathbb{F}_{2^n}$). In particular, our software tools allow to speed-up the G*e*MSS [15], `Gui` [17] and DualModeMS [25] signature schemes, which are candidates submitted to the NIST post-quantum cryptography standardization process [39]. The advantage of $\mathbb{F}_{2^n}$ is that each element can be represented as a vector of bits, which corresponds to the architecture of binary computers and can be naturally improved by vector instructions. The signature generation requires arithmetic in $\mathbb{F}_{2^n}[X]$, and its

implementation is already provided by various libraries. Among the best, `NTL` [43] provides high quality implementations of state-of-the-art algorithms. But these algorithms are not specialized for the case of sparse polynomials in $\mathbb{F}_{2^n}[X]$. Moreover, the implementations are not constant-time and so are vulnerable to timing attacks. For these reasons, we need to adapt the algorithms used. We have chosen to create a new library, which is based on constant-time arithmetic in $\mathbb{F}_{2^n}$. Unlike `NTL`, which offers a general implementation, our library is specialized for a value of n, permitting more efficient arithmetic. Moreover, we exploit the sparse polynomial structure to improve the performance. More generally, our implementation uses the Intel vector instructions to obtain interesting speed-ups.
Our library also supports DualModeMS [25], which is a candidate in the NIST PQC standardization process. It is a modified `HFE`-based signature scheme which permits to decrease the size of the public-key, but by increasing the size of the signature. The parameters are chosen to minimize the sum of both sizes. By improving the implementation of `HFE`-based schemes, we automatically improve the implementation of DualModeMS.

Evaluation of multivariate quadratic systems. Many multivariate cryptosystems require to evaluate a multivariate quadratic system (MQS) to encrypt data or verify a signature (e.g. [17, 15, 19]). Encryption uses secret data and should be performed in constant-time, whereas verification is a public process and does not have this constraint. In `HFEv-` signature schemes, the evaluation step is the main part of verification. Efficient implementations of evaluation have been studied in [7, 16, 19, 20]. The authors of [7] propose different strategies for the evaluations in $\mathbb{F}_2, \mathbb{F}_{2^4}$ and $\mathbb{F}_{2^8}$. In [16], the evaluation is vectorized with `ssse3` instructions in $\mathbb{F}_{31}, \mathbb{F}_{16}$ and $\mathbb{F}_{256}$. In [19], the authors propose to optimize the evaluation in $\mathbb{F}_{31}$ and $\mathbb{F}_{2^{256}}$ by evaluating the public-key equations one-by-one. Their implementation is vectorized with the `avx2` instructions set. In [20], the authors present a faster evaluation with the same instructions set. To do so, they use a "*monomial representation*" of the public-key: for each monomial, the corresponding coefficients in each equation are stored together. We optimize the evaluation with this representation to obtain new speed records.

Root finding of a `HFE` polynomial. The main part of the signature generation in `Gui` and G*e*MSS is to find the roots of a polynomial F in $\mathbb{F}_{2^n}[X]$ with a specific form. Root finding is a fundamental problem in computer algebra with various applications in discrete mathematics. A survey of the main root finding methods can be found in [46]. Recently, the successive resultants algorithm (SRA) [41] has been proposed to find the roots of a polynomial in small characteristic, and this work has been extended for split polynomials in general finite fields. In [22], root finding is improved for split and separable polynomials, when the cardinality of multiplicative group is smooth.
In the case of the `HFE` polynomial F in $\mathbb{F}_{2^n}[X]$, F has a sparse structure and its coefficients are in a field of small characteristic. Moreover, the number of roots is generally small (it is almost always less than 10 for our parameters). The main challenge is to exploit the sparse structure of F to improve the complexity of the root finding: it should depend on the number of coefficients of F and not on its degree. In practice, the Berlekamp algorithm [46, Algorithm 14.15] is used, which computes $\texttt{GCD}(F, (X^{2^n} - X) \bmod F)$. The most costly task is the computation of $X^{2^n} \bmod F$, also called the Frobenius map, and the `HFE` structure can be exploited during the modular reduction by F. In [42], the authors propose a method to compute the Frobenius map with multi-squaring tables, which is interesting when the degree of F is (approximately) smaller than n. We study how to implement the Frobenius map efficiently, optimizing as a function of the parameters.

Arithmetic in $\mathbb{F}_{2^n}$. Arithmetic in $\mathbb{F}_{2^n}$ is a critical part of the root finding algorithm, because all operations in $\mathbb{F}_{2^n}[X]$ require it, and is studied in [4, 3, 45, 11]. In particular, multiplication in $\mathbb{F}_{2^n}$ is the most critical operation. This is a well-known task and is studied in [14, 23, 37, 18]. We choose here to use the `PCLMULQDQ` instruction (Section 1.7) to obtain an efficient implementation. This instruction computes the product of two binary polynomials, each of degree strictly less than 64.

Organization of the Paper and Main Results

We present `MQsoft` [2]: an efficient open-source library in `C` for `HFE`-based schemes such as G*e*MSS, `Gui` and DualModeMS. `MQsoft` is an improved version of the G*e*MSS additional implementation submitted to the NIST post-quantum cryptography competition [39]. Our library permits to improve the fastest known implementations for G*e*MSS and `Gui`, as well as the signature generation of DualModeMS. The performance results are studied in Section 5. Table 1 summarizes the obtained speed-ups. For the levels of security 128 and 192 bits of `Gui`, we modify slightly a security parameter to improve the performance (cf. Section 5.2).

Table 1: Speed-up of G*e*MSS, `Gui` and DualModeMS (best implementation provided for the NIST submissions versus our implementation). We use a Haswell processor (ServerH).

scheme	sec. level	key gen.	signature gen.	signature verif.
G*e*MSS	128	2.8	1.57	1.98
	192	2.4	1.25	1.83
	256	2.3	1.23	1.75
`Gui`	128	13	1.73	1.74
	192	14	1.21	1.59
	256	19	2.5	1.59
Inner.DualModeMS	128	2.2	1.30	2.1
DualModeMS	128	1.00	1.32	1.00

The structure of `MQsoft` is depicted in Figure 1 which summarizes the main tasks required for each cryptographic operation. The critical part of an operation is represented by a plain arrow, whereas less important operations are represented by dotted arrows.

It is clear from Figure 1, that `HFE`-based schemes require an efficient implementation of arithmetic in $\mathbb{F}_{2^n}[X]$ and so in $\mathbb{F}_{2^n}$. This is studied in Section 2. We have implemented state-of-the-art algorithms for arithmetic in $\mathbb{F}_{2^n}$ that use vectorization (`sse2` and `avx2`) and the `PCLMULQDQ` instruction to improve multiplication in $\mathbb{F}_{2^n}$ (Section 2.2). The multiplication is computed with the schoolbook algorithm in blocks of 64 bits for $n \leq 384$ and with Karatsuba otherwise. When `PCLMULQDQ` is not available, `MQsoft` uses the multiplication in $\mathbb{F}_2[X]$ of the `gf2x` library (Section 1.7). The modular inverse is computed with the Itoh-Tsujii Multiplicative Inversion Algorithm (Section 2.5) together with multi-squaring tables (Section 2.4).
To optimize the arithmetic in $\mathbb{F}_{2^n}$, the choice of n must be made before the compilation. This permits the specialization of the implementation. The library is flexible and allows to the choice of any $n \leq 576$. $\mathbb{F}_{2^n}$ is built as $\mathbb{F}_2$ quotiented by an irreducible polynomial f of degree n. When it is possible, we choose an irreducible trinomial for f to accelerate the modular reduction (Section 2.3). The modular reduction by f is vectorized for trinomials such that the degree of $f(x) - x^n$ is strictly less than 128, and for the parameters of studied schemes. We have vectorized the modular reduction by a pentanomial exclusively for $n \in \{184, 312, 448, 544\}$, because they are the parameters of `Gui` and `DualModeMS256`. Otherwise, the modular reduction is implemented for pentanomials such that the degree of $f(x) - x^n$ is strictly less than 33. For $n \leq 576$, 56% of the finite fields can be created

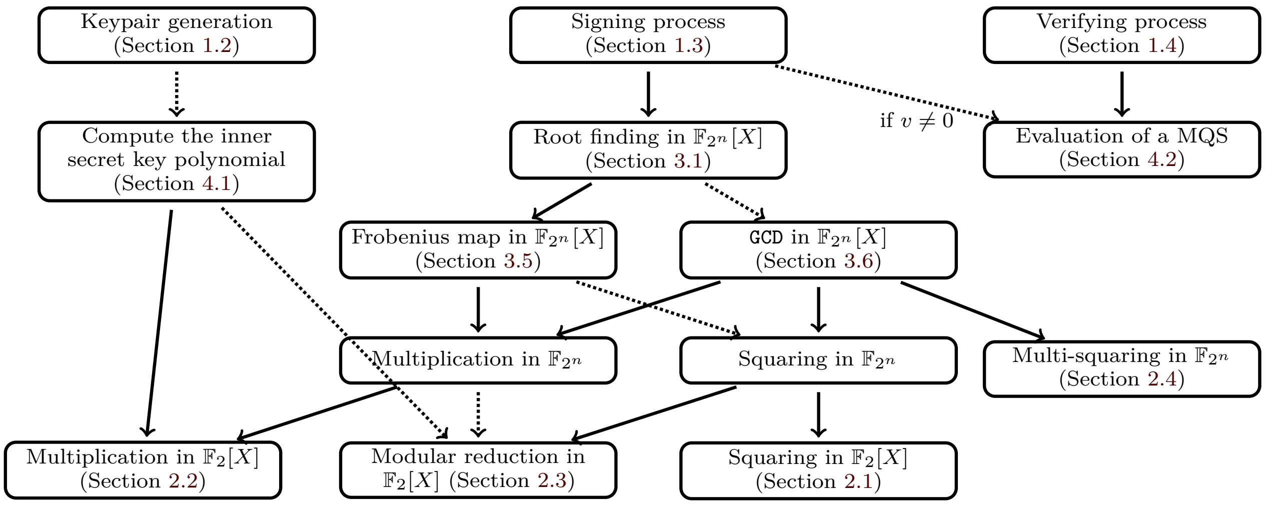
Keypair generation (Section 1.2)
Signing process (Section 1.3)
Verifying process (Section 1.4)
Compute the inner secret key polynomial (Section 4.1)
Root finding in $\mathbb{F}_{2^n}[X]$ (Section 3.1)
if $v \neq 0$
Evaluation of a MQS (Section 4.2)
Frobenius map in $\mathbb{F}_{2^n}[X]$ (Section 3.5)
GCD in $\mathbb{F}_{2^n}[X]$ (Section 3.6)
Multiplication in $\mathbb{F}_{2^n}$
Squaring in $\mathbb{F}_{2^n}$
Multi-squaring in $\mathbb{F}_{2^n}$ (Section 2.4)
Multiplication in $\mathbb{F}_2[X]$ (Section 2.2)
Modular reduction in $\mathbb{F}_2[X]$ (Section 2.3)
Squaring in $\mathbb{F}_2[X]$ (Section 2.1)

Figure 1: Dependencies between the different operations performed in `MQsoft`.

with an irreducible trinomial. Our library vectorizes modular reduction for 92% of these cases. We obtain approximately a speed-up of a factor of 4 compared to the arithmetic in $\mathbb{F}_{2^n}$ of `NTL`.

In Section 4.2, the verifying process is accelerated via an efficient evaluation of multivariate quadratic systems using `avx2` set instructions. We obtain new speed records for the constant-time and variable-time evaluations of binary multivariate polynomials. To do this, we have chosen to use the "*monomial representation*" as in [20]. For this, we have stored multivariate quadratic systems of m equations in $\mathbb{F}_2[x_1, \ldots, x_{n+v}]$ as a pair $(C, Q) \in \mathbb{F}_{2^m} \times \mathcal{M}_{n+v}(\mathbb{F}_{2^m})$, where Q is a upper triangular matrix such that $Q_{i,j}$ corresponds to the term $x_i x_j$, and C is the constant term. Since the multivariate quadratic systems will be evaluated in $\mathbb{F}_2^{n+v}$, $x_i^2 = x_i$ and so, the linear term x_i is stored with the term x_i^2 of Q. With this representation, the evaluation in $\mathbf{x} \in \mathbb{F}_2^{n+v}$ is computed as $C + \mathbf{x}Q\mathbf{x}^t$. For 256 equations and 256 variables in $\mathbb{F}_2$, our variable-time evaluation is 1.38 times faster than in [20]. To obtain this, we use unrolled loops and a specific way to extract the terms x_i. For the constant-time evaluation, we obtain a performance similar to [20], which targets Haswell processors. However, on Skylake processors, the evaluation can be faster by using vector instructions in a specific way, as explained in Section 4.2. This method saves a factor 1.1 on Skylake (for 256 equations and 256 variables).

The core of the signing process is to find the roots of a univariate `HFE` polynomial F in $\mathbb{F}_{2^n}[X]$, which has a special structure. In particular, F is in the following note:

$$\sum_{\substack{0 \leqslant j < i < n \\ 2^i + 2^j \leqslant D}} A_{i,j} X^{2^i + 2^j} + \sum_{\substack{0 \leqslant i < n \\ 2^i \leqslant D}} B_i X^{2^i} + C \in \mathbb{F}_{2^n}[X].$$

Our goal is to exploit this structure to accelerate the root finding. We address this question in Section 3. We have been able to tweak Berlekamp's algorithm [46, Algorithm 14.15] to take advantage of the sparse structure of F.
When $D > n$, the computation of $X^{2^n} \bmod F$ is done with the repeated squaring algorithm [46, Algorithm 4.8]. The core of this algorithm is to compute the modular reduction of the square of an element $B \in \mathbb{F}_{2^n}[X]$ by F. The classical Euclidean division of B^2

by F requires to compute $B^2 - QF$, where Q is the quotient of this division. With a naive implementation, the multiplication of Q by F costs $O(D^2)$ field multiplications. Using a sparse representation of F, the multiplication costs only $O(D \log_2(D)^2)$ field multiplications.
So, with a sparse polynomial, the computation of the roots is faster. This suggests to consider sparse `HFE` polynomials. In Theorem 1, we prove that making F more sparse improves the complexity. Because F is a part of the secret-key, the nature of this change requires a new analysis of security. We observe in practice that removing a small number of odd degree terms appears not to affect the security. However, the security of this method must be studied in depth. With Theorem 1, we can save 43.75% of the computations we would have done by removing only three terms having an odd degree in F. The general idea to make F more sparse has already been proposed in HFEBoost[1], but independently of this, the proof of Theorem 1 makes explicit a method to improve the complexity. It has the advantage of being in constant-time because the useless computations are known and so can be avoided.

Theorem 1. *Let H be a* `HFE` *polynomial of degree D in $\mathbb{F}_{2^n}[X]$ where the k-th terms of highest odd degree have been removed ($k \in [\![0, \lceil \log_2(D) \rceil - 1]\!]$), and let $A \in \mathbb{F}_{2^n}[X]$ be a square of degree at most $2D - 2$. If D is even, then the computation of the classical Euclidean division (Algorithm 8) of A by H can be accelerated by a factor $(D-1)/(\frac{D}{2} + \lfloor 2^{\lceil \log_2(D) \rceil - k - 2} \rfloor)$.*

When $D < n$, the strategy of [42] becomes more efficient for computing the Frobenius map. The idea is to compute a table of $X^{2i} \bmod F$ to accelerate the modular reduction. Thus, the squaring modulo F is computed by multiplying its i-th coefficient by the element $X^{2i} \bmod F$ from the table for $i \in [\![0, D-1]\!]$. The authors of [42] also suggest to do several squarings in one step, with multi-squaring tables. In Section 3.5, we make an explicit strategy for doing this efficiently (by exploiting the `HFE` structure of F when it is possible), and how to choose the number of squarings to do before the modular reduction.

The performance of both strategies described above depends on the required number of field multiplications. In Section 3.5, we compute accurately the number of multiplications of each method, in order to choose the best strategy as a function of the parameters.

1 Preliminaries

We briefly recall here the principle of a signature scheme based on `HFEv-` [34]. The public-key in `HFEv-` is given by a set of m quadratic equations in $\mathbb{F}_2[x_1, ..., x_{n+v}]$. These equations are derived from a single polynomial $F \in \mathbb{F}_{2^n}[X]$ (Section 1.2). Verification requires to evaluate the public polynomials (Section 1.4). We need to compute the roots of F in order to generate a signature (Section 1.3). From now, we always assume that the base field is $\mathbb{F}_2$. In Section 1.5, we introduce the specificities of `GeMSS` and `Gui` which are two submissions to the NIST PQC process based on `HFEv-`.

1.1 Main Parameters

The main parameters involved are:

- D, a positive integer that corresponds to the degree of a secret polynomial. D is such that $D = 2^i$ for $i \geq 0$, or $D = 2^i + 2^j$ for $i \neq j$, and $i, j \geq 0$,
- m, number of equations in the public-key,

[1] https://www-polsys.lip6.fr/Links/hfeboost.html

- n, the degree of a field extension of $\mathbb{F}_2$,
- v, the number of vinegar variables,
- Δ, the number of minus (the number of equations in the public-key is such that is $m = n - \Delta$),
- nb_ite ≥ 1, number of iterations in the verification and signature processes.

1.2 Keypair Generation

Secret-key. It is composed by a couple of invertible matrices[2] $(\mathbf{S}, \mathbf{T}) \in \mathrm{GL}_{n+v}(\mathbb{F}_2) \times \mathrm{GL}_n(\mathbb{F}_2)$ and a polynomial $F \in \mathbb{F}_{2^n}[X, v_1, \ldots, v_v]$ with the following structure:

$$\sum_{\substack{0\leqslant j<i<n\\ 2^i+2^j\leqslant D}} A_{i,j}\, X^{2^i+2^j} + \sum_{\substack{0\leqslant i<n\\ 2^i\leqslant D}} \beta_i(v_1,\ldots,v_v)\, X^{2^i} + \gamma(v_1,\ldots,v_v), \tag{1}$$

where $A_{i,j} \in \mathbb{F}_{2^n}, \forall i,j, 0 \leqslant j < i < n$, each $\beta_i : \mathbb{F}_2^v \to \mathbb{F}_{2^n}$ is linear and $\gamma(v_1,\ldots,v_v) : \mathbb{F}_2^v \to \mathbb{F}_{2^n}$ is quadratic. The variables $v_1,\ldots,v_v$ are called the *vinegar variables*. We shall say that a polynomial $F \in \mathbb{F}_{2^n}[X, v_1, \ldots, v_v]$ with the form of (1) has a `HFEv`-*shape*. The particularity of a polynomial $F(X, v_1, \ldots, v_v)$ with `HFEv`-shape is that for any specialization of the vinegar variables the polynomial F becomes a `HFE` polynomial [40], i.e. univariate polynomial of the following form:

$$\sum_{\substack{0\leqslant j<i<n\\ 2^i+2^j\leqslant D}} A_{i,j}\, X^{2^i+2^j} + \sum_{\substack{0\leqslant i<n\\ 2^i\leqslant D}} B_i\, X^{2^i} + C \in \mathbb{F}_{2^n}[X], \tag{2}$$

with $A_{i,j}, B_i, C \in \mathbb{F}_{2^n}, \forall i,j, 0 \leqslant j < i < n$. By abuse of notation, we will refer to D as the degree of the `HFEv` polynomial.
The special structure of (1) is chosen such that its *multivariate representation* over the base field $\mathbb{F}_2$ is composed by quadratic polynomials in $\mathbb{F}_2[x_1,\ldots,x_{n+v}]$. This is due to the special exponents chosen in X that have all a binary decomposition of Hamming weight at most 2.
Let $\theta = (\theta_1,\ldots,\theta_n) \in (\mathbb{F}_{2^n})^n$ be a basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$. We set

$$\varphi : E = \sum_{k=1}^{n} e_k \cdot \theta_k \in \mathbb{F}_{2^n} \longrightarrow \varphi(E) = (e_1,\ldots,e_n) \in \mathbb{F}_2^n.$$

We can now define a set of multivariate polynomials $\mathbf{f} = (f_1,\ldots,f_n) \in \mathbb{F}_2[x_1,\ldots,x_{n+v}]^n$ derived from a `HFEv` polynomial $F \in \mathbb{F}_{2^n}[X, v_1,\ldots,v_v]$ by:

$$F\left(\sum_{k=1}^{n} \theta_k x_k, v_1, \ldots, v_v\right) = \textstyle\sum_{k=1}^{n} \theta_k f_k\,. \tag{3}$$

For easing the notations, we now identify the vinegar variables $(v_1,\ldots,v_v) = (x_{n+1},\ldots,x_{n+v})$. Besides, we shall say that the polynomials $f_1,\ldots,f_n \in \mathbb{F}_2[x_1,\ldots,x_{n+v}]$ are the *components* of F over $\mathbb{F}_2$.
In the implementation, we compute the components of F by using directly Equation (3). We replace X by $\sum_{k=1}^{n} \theta_k x_k, v_1,\ldots,v_v$ in the expression of F, then we use a rearrangement of terms which minimizes the number of multiplications. This is detailed in Section 4.1.

[2]In full generality, one can have affine transformations. We choose linear transformations for the sake of simplicity.

Public-key. It is given by a set of m quadratic *square-free* non-linear polynomials $\mathbf{p} = (p_1, \ldots, p_m) \in \mathbb{F}_2[x_1, \ldots, x_{n+v}]^m$. It is obtained from the secret-key by taking the first $m = n - \Delta$ polynomials of:

$$\Big(f_1\big((x_1, \ldots, x_{n+v})\mathbf{S}\big), \ldots, f_n\big((x_1, \ldots, x_{n+v})\mathbf{S}\big)\Big)\mathbf{T}, \tag{4}$$

and reducing it modulo the field equations, i.e. modulo $\langle x_1^2 - x_1, \ldots, x_{n+v}^2 - x_{n+v}\rangle$.

We summarize the public-key/secret-key generation in Algorithm 1. In practice, we merge the steps 6 and 7 by removing the Δ last columns of $\mathbf{T}$ during the vector matrix product.

Algorithm 1 PK/SK generation in `HFEv-` schemes

```
1: procedure KeyGen
2:     Randomly sample $(\mathbf{S}, \mathbf{T}) \in \mathrm{GL}_{n+v}(\mathbb{F}_2) \times \mathrm{GL}_n(\mathbb{F}_2)$
3:     Randomly sample $F \in \mathbb{F}_{2^n}[X, v_1, \ldots, v_v]$ with HFEv-shape of degree $D$
4:     sk ← $(F, \mathbf{S}, \mathbf{T}) \in \mathbb{F}_{2^n}[X, v_1, \ldots, v_v] \times \mathrm{GL}_{n+v}(\mathbb{F}_2) \times \mathrm{GL}_n(\mathbb{F}_2)$
5:     Compute $\mathbf{f} = (f_1, \ldots, f_n) \in \mathbb{F}_2[x_1, \ldots, x_{n+v}]^n$ such that:
           $$F\left(\sum_{k=1}^{n} \theta_k x_k, v_1, \ldots, v_v\right) = \sum_{k=1}^{n} \theta_k f_k$$
                                                    ▷ See Section 4.1 for details on Step 5.
6:     Compute $(p_1, \ldots, p_n) =$
       $\Big(f_1\big((x_1, \ldots, x_{n+v})\mathbf{S}\big), \ldots, f_n\big((x_1, \ldots, x_{n+v})\mathbf{S}\big)\Big)\mathbf{T} \bmod \langle x_1^2 - x_1, \ldots, x_{n+v}^2 - x_{n+v}\rangle \in \mathbb{F}_2[x_1, \ldots, x_{n+v}]^n$
7:     pk ← $\mathbf{p} = (p_1, \ldots, p_m) \in \mathbb{F}_2[x_1, \ldots, x_{n+v}]^m$          ▷ Take the first $m = n - \Delta$
       polynomials computed in Step 6.
8:     return (sk, pk)
9: end procedure
```

1.3 Signing process

The main step of the signature process requires to invert the public-key, that is to say solving:

$$p_1(x_1, \ldots, x_{n+v}) - d_1 = 0, \ldots, p_m(x_1, \ldots, x_{n+v}) - d_m = 0. \tag{5}$$

for $\mathbf{d} = (d_1, \ldots, d_m) \in \mathbb{F}_2^m$.

To do so, we randomly sample $\mathbf{r} = (r_1, \ldots, r_\Delta) \in \mathbb{F}_2^\Delta$ and append it to $\mathbf{d}$. This gives $\mathbf{d}' = (\mathbf{d}, \mathbf{r}) \in \mathbb{F}_2^n$. We compute then $D' = \varphi^{-1}(\mathbf{d}' \times \mathbf{T}^{-1}) \in \mathbb{F}_{2^n}$ and try to find a root $(Z, z_1, \ldots, z_v) \in \mathbb{F}_{2^n} \times \mathbb{F}_2^v$ of the multivariate equation:

$$F(Z, z_1, \ldots, z_v) - D' = 0.$$

To solve this equation, we take advantage of the special `HFEv`-shape. That is why, we randomly sample $\mathbf{v} \in \mathbb{F}_2^v$ and consider the univariate polynomial $F(X, \mathbf{v}) \in \mathbb{F}_{2^n}[X]$. This yields a `HFE` polynomial according to Section 1.2. We then find the roots of the univariate equation:

$$F(X, \mathbf{v}) - D' = 0.$$

If there is a root $Z \in \mathbb{F}_{2^n}$, we return $(\varphi(Z), \mathbf{v}) \times \mathbf{S}^{-1} \in \mathbb{F}_2^{n+v}$.

A core part of the signature generation is then the computation of the roots of $F_{D'}(X) = F(X, \mathbf{v}) - D'$. To this end, we use the Berlekamp algorithm as described in [46, Algorithm 14.15], which requires mainly to compute:

$$\texttt{GCD}(X^{2^n} - X \bmod F_{D'}, F_{D'}).$$

We provide in Section 3.1 the best methods to compute $X^{2^n} \bmod F_{D'}$ and the `GCD` (both in function of n and D).

We can now present a way to define the inversion function (Algorithm 2):

Algorithm 2 Inverse map of the public-key

```
1: procedure $\mathrm{Inv}_{\mathbf{p}}(\mathbf{d} \in \mathbb{F}_2^m, \mathsf{sk} = (F, \mathbf{S}, \mathbf{T}) \in \mathbb{F}_{2^n}[X, v_1, \ldots, v_v] \times \mathrm{GL}_{n+v}(\mathbb{F}_2) \times \mathrm{GL}_n(\mathbb{F}_2))$
2:     repeat
3:         $\mathbf{r} \in_R \mathbb{F}_2^{\Delta}$                    ▷ The notation $\in_R$ stands for randomly sampling.
4:         $\mathbf{d}' \leftarrow (\mathbf{d}, \mathbf{r}) \in \mathbb{F}_2^n$
5:         $D' \leftarrow \varphi^{-1}(\mathbf{d}' \times \mathbf{T}^{-1}) \in \mathbb{F}_{2^n}$
6:         $\mathbf{v} \in_R \mathbb{F}_2^v$
7:         $F_{D'}(X) \leftarrow F(X, \mathbf{v}) - D'$
8:         $(\cdot, \mathrm{Roots}) \leftarrow \mathrm{FindRoots}(F_{D'})$              ▷ Call to Algorithm 6.
9:     until $\mathrm{Roots} \neq \emptyset$
10:    $Z \in_R \mathrm{Roots}$
11:    return $(\varphi(Z), \mathbf{v}) \times \mathbf{S}^{-1} \in \mathbb{F}_2^{n+v}$
12: end procedure
```

The signing algorithm in G*e*MSS and `Gui` is an iterative process. The basic idea is to call $\mathrm{Inv}_{\mathbf{p}}$ nb_ite times. More precisely, the signing process is in the following algorithm:

Algorithm 3 Signing process

```
1: procedure $\mathrm{SIGN}(\mathbf{M} \in \{0,1\}^*, \mathsf{sk} \in \mathbb{F}_{2^n}[X, v_1, \ldots, v_v] \times \mathrm{GL}_{n+v}(\mathbb{F}_2) \times \mathrm{GL}_n(\mathbb{F}_2), \mathrm{Inv}_{\mathbf{p}})$
2:     $\mathbf{H} \leftarrow \mathrm{HASH}(\mathbf{M})$
3:     $\mathbf{S}_0 \leftarrow \mathbf{0} \in \mathbb{F}_2^m$
4:     for $i$ from 1 to nb_ite do
5:         $\mathbf{D}_i \leftarrow$ first $m$ bits of $\mathbf{H}$          ▷ $\mathbf{S}_i \in \mathbb{F}_2^m$ and $\mathbf{X}_i \in \mathbb{F}_2^{\Delta+v}$
6:         $(\mathbf{S}_i, \mathbf{X}_i) \leftarrow \mathrm{Inv}_{\mathbf{p}}(\mathbf{D}_i \oplus \mathbf{S}_{i-1})$      ▷ $\oplus$ is the component-wise XOR
7:         $\mathbf{H} \leftarrow \mathrm{HASH}(\mathbf{H})$
8:     end for
9:     return $(\mathbf{S}_{\mathrm{nb\_ite}}, \mathbf{X}_{\mathrm{nb\_ite}}, \ldots, \mathbf{X}_1)$
10: end procedure
```

nb_ite is a parameter that can be easily computed from m and the level of security.

1.4 Verifying process

Naturally, the verifying process is also iterative as shows in Algorithm 4. The main part of this process is still to evaluate the public-key. We describe how to implement this step efficiently in Section 4.2.

Algorithm 4 Verifying process

```
1: procedure VERIFY($\mathbf{M} \in \{0,1\}^*$, nb_ite $> 0$, $\mathsf{sm} \in \mathbb{F}_2^{m+\text{nb\_ite}(\Delta+v)}$, $\mathsf{pk} = \mathbf{p} \in \mathbb{F}_2[x_1,\ldots,x_{n+v}]^m$)
2:     $(\mathbf{S}_{\text{nb\_ite}}, \mathbf{X}_{\text{nb\_ite}}, \ldots, \mathbf{X}_1) \leftarrow \mathsf{sm}$
3:     $\mathbf{H} \leftarrow \text{HASH}(\mathbf{M})$
4:     $\mathbf{D}_1 \leftarrow$ first $m$ bits of $\mathbf{H}$
5:     for $i$ from 2 to nb_ite do
6:         $\mathbf{H} \leftarrow \text{HASH}(\mathbf{H})$
7:         $\mathbf{D}_i \leftarrow$ first $m$ bits of $\mathbf{H}$
8:     end for
9:     for $i$ from nb_ite $- 1$ to 0 do
10:        $\mathbf{S}_i \leftarrow \mathbf{p}(\mathbf{S}_{i+1}, \mathbf{X}_{i+1}) \oplus \mathbf{D}_{i+1}$
11:    end for
12:    return VALID if $\mathbf{S}_0 = \mathbf{0}$ and INVALID otherwise.
13: end procedure
```

1.5 G*e*MSS and Gui

G*e*MSS and Gui are essentially based on the same principle. They still differ in the choice of parameters.

Choice of parameters. Table 26 summarizes the performance of the best implementations of G*e*MSS and Gui provided for the NIST submissions, in function of the parameters. For G*e*MSS, we consider the additional implementation. For Gui, we take the PCLMULQDQ additional implementation. Gui is implemented with the modified algorithms to achieve EUF-CMA security property [17, Section 1.6]. It is not the case for the G*e*MSS implementation. For this reason, we have modified the Gui additional implementation. This implementation provides the cryptographic operations of Gui with and without the EUF-CMA security property. We just replace the algorithms by their version without this property. Compared to the original implementation, this implies mainly a speed-up during the signature generation.
In Gui, the parameters are chosen to minimize the time of signing and verifying a signature. To do it, small values of D are chosen, and larger values of m are used. This choice implies to increase n, and so the size of the public-key. In G*e*MSS, it is the opposite. The goal is to minimize the size of the public-key. This leads to smaller values of n. This choice implies to take larger values of D.
As soon as m is fixed, the number of iterations required can be derived. In fact, the original parameters of Gui do not always provide the claimed security. The attack described in [21, Theorem 6.2.1] has a complexity $O(2^{m\frac{\text{nb\_ite}}{\text{nb\_ite}+1}})$. So, with $m = 168$ and nb_ite $= 2$, the original parameters of Gui-184 provide a security of only 112 bits. This problem has been mentioned in NIST's pqc-forum mailing list by W. Beullens the 04/27/2018. The Gui designers have answered the 06/15/2018, and choose to set the parameter nb_ite to 3. However, this provides only 126 bits of security. To reach 128 bits of security, we should set nb_ite to 4 (or to set m to 171). For completeness, we have measured Gui-184 for these three values of nb_ite. The modification slows down a factor $\frac{\text{nb\_ite}}{2}$ the signature generation and the verifying process of Gui-184.

To find a root of $F(X, \mathbf{v}) - D'$. During the signature generation, $F(X, \mathbf{v}) - D'$ cannot have solution. In G*e*MSS, $\mathbf{r}$ and $\mathbf{v}$ are changed while $F(X, \mathbf{v}) - D'$ does not have roots (cf. Algorithm 2). Then, when there are roots, one is deterministically chosen for fixed D', by using the SHA-3 of D' as a randombytes generator. In Gui, $\mathbf{v}$ is changed while

$F(X, \mathbf{v}) - D'$ does not have a unique root.
In practice, $\mathbf{d}$ is generated from the hash of a document. To do it, G*e*MSS uses `SHA-3`, whereas `Gui` uses `SHA-2`. The output size of the hash functions is the double of the level of security. In `MQsoft`, we have chosen the `SHA-3` function from the `Keccak Code Package` [26] and the `SHA-2` function from `OpenSSL`.

1.6 Data Structure

We describe here the data structure used to store elements of $\mathbb{F}_{2^n}$ and $\mathbb{F}_{2^n}[X]$. This representation is crucial for the efficiency of our implementation. This is especially true for binary fields since operations in $\mathbb{F}_2$ can be naturally vectorized. The arithmetic in $\mathbb{F}_{2^n}[X]$ is used during the root finding of a `HFE` polynomial. To be efficient, it is important to distinguish dense polynomials which appear during the computation of the Frobenius map and the `GCD`, from a `HFE` polynomial which is used to reduce $X^{2^n} - X$. We can notice that the `HFE` polynomial is sparse since it has only $K = O(\log_2(D)^2)$ non-zero coefficients.

Representation of elements in $\mathbb{F}_{2^n}$. The field $\mathbb{F}_{2^n}$ is defined as $\frac{\mathbb{F}_2[X]}{f(x)}$ with $f(x)$ being an irreducible polynomial of degree n in $\mathbb{F}_2[x]$. We have chosen the polynomial basis [31]. An element of $\mathbb{F}_{2^n}$ is represented by a polynomial in $\mathbb{F}_2[x]$ of degree at most $n-1$. The coefficients are stored as a vector of bits, requiring $\lceil \frac{n}{w} \rceil$ words, where w is the word size (in bits). The j-th bit of the i-th word is the coefficient of the term of degree $wi + j$, for $i \in [\![0, \lceil \frac{n}{w} \rceil - 1]\!]$ and $j \in [\![0, w-1]\!]$. It is setting to zero when $(wi + j) \geq n$.

Example 1. Let $w = 64$ and $P = x^{36} + x^4 \in \mathbb{F}_{2^{40}}$. To simplify the notations, we represent vectors of bits as 64-bit integers. P is stored as `0x0000001000000010`. In particular, the bits from 37 to 63 are setting to zero.

Representation of dense polynomials in $\mathbb{F}_{2^n}[X]$. An element of $\mathbb{F}_{2^n}[X]$ is represented by its degree δ and a vector of $(\delta + 1)$ coefficients. The coefficients are stored from lower to higher degree of the corresponding terms in a buffer. The degree is stored in a local variable, excepted for the implementation of the fast `GCD` in Section 3.6, because it requires matrices in $\mathcal{M}_2(\mathbb{F}_{2^n}[X])$. In this case, we use a `C` structure to store the degree and the pointer toward the coefficients buffer.

Example 2. Let $P = X^8 + \alpha X^7 + (\alpha+1)X^6 + X^5 + \alpha X \in \mathbb{F}_4[X]$ and $f(\alpha) = \alpha^2 + \alpha + 1$. P is stored as $8, (0, \alpha, 0, 0, 0, 1, \alpha + 1, \alpha, 1)$.

Representation of HFE polynomials in $\mathbb{F}_{2^n}[X]$. In `HFEv` scheme, the `HFEv` polynomial is a part of the secret-key. During the signature generation (cf. Section 1.3), the vinegar variables of the `HFEv` polynomial are evaluated to obtain a `HFE` polynomial. Its degree D is a parameter of security and is assumed to be known. It is defined by the `C` directive `#define`. A `HFE` polynomial in $\mathbb{F}_{2^n}[X]$ is represented as a vector of coefficients where only terms X^0, X^{2^i} and $X^{2^i+2^j}$ are stored. It is chosen monic and so the leading term is not stored. If P is in $\mathbb{F}_{2^n}[X]$, we denote by $P_{\texttt{HFE}}$ its `HFE` representation.

Example 3. Let $P = X^{16} + \alpha X^{12} + (\alpha+1)X^{10} + \alpha X \in \mathbb{F}_4[X]$ and $f(\alpha) = \alpha^2 + \alpha + 1$. $P_{\texttt{HFE}}$ is stored as $(0, \alpha, 0, 0, 0, 0, 0, 0, 0, \alpha + 1, \alpha)$. Only the coefficients of terms with a degree in $\{0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12\}$ are stored.

1.7 Experimental Platform and Benchmarked Libraries

Tables 2 and 3 summarize the main informations about the platform used in the experimental measurements. LaptopS is used for all measurements, excepted in Section 4.2

Table 2: Processors.

Computer	Processor	Frequency	Max freq.	Architecture
LaptopS	Intel(R) Core(TM) i7-6600U CPU	2.60 GHz	3.40 GHz	Skylake
ServerH	Intel(R) Xeon(R) CPU E3-1275 v3	3.50 GHz	3.90 GHz	Haswell
DesktopH	Intel(R) Core(TM) i7-4790 CPU	3.60 GHz	4.00 GHz	Haswell
DesktopS	Intel(R) Core(TM) i7-6700 CPU	3.40 GHz	4.00 GHz	Skylake

Table 3: OS and Memory.

Computer	OS	RAM	L1d	L1i	L2	L3
LaptopS	Ubuntu 16.04.5 LTS	32 GB	32 KB	32 KB	256 KB	4096 KB
ServerH	CentOS Linux 7 (Core)					8192 KB
DesktopH	Debian GNU/Linux 9	16 GB				
DesktopS						

and in Section 5. For this latter, we present the performance of `MQsoft` in function of the processor. Our implementation targets Intel 64-bit processors that support the `PCLMULQDQ` instruction. This allows to improve the performance of multiplication in $\mathbb{F}_{2^n}$ (cf. Section 2.1 and 2.2). We also take advantage of the `sse2`, `ssse3` and `avx2` instruction sets to speed-up the implementation of arithmetic in $\mathbb{F}_{2^n}$ (Section 2), the vector matrix product in $\mathbb{F}_2$ during the keypair generation (Section 5.4), and the evaluation of the public-key during the verifying process (Section 4.2). We explain here the main vector instructions[3] that `MQsoft` exploits:

- `PCLMULQDQ`: this instruction computes the product of two binary polynomials such that their degree is strictly less than 64.
- `PSHUFB`: this instruction from `ssse3` takes 16 indices on 4 bits, and searches the corresponding 8-bit elements in a table of size 128 bits.
- `VPSHUFB`: this instruction from `avx2` performs two times `PSHUFB`.
- `PSLLDQ` and `PSRLDQ`: these instructions from `sse2` computes respectively the left and right shift of a 128-bit register by a multiple of 8 bits.
- `PALIGNR`: this instruction from `ssse3` concatenates two registers 128 bits, shifts the concatenation at right by a multiple of 8 bits, then return the 128 lower bits of the result.
- `VPBROADCASTQ`: this instruction from `avx` duplicates four times a 64-bit integer on a 256-bit register.
- `VPERMQ`: this instruction from `avx2` permutes the 64-bit parts of a 256-bit register. In particular, it can duplicate one 64-bit part four times.
- `VPMASKMOVQ`: this instruction from `avx2` loads four contiguous 64-bit integers from a buffer, then applies a mask which permits to set to zero 64-bit parts.
- `POPCNT`: this instruction counts the number of bits set to 1 in 64-bit integers. It is used to speed-up the dot product of vectors in $\mathbb{F}_2$.

[3]For more informations, we refer to the Intel Intrinsics Guide (`https://software.intel.com/sites/landingpage/IntrinsicsGuide/#`).

We benchmarked our software toolkit against the following softwares or libraries.

- `Magma` ([12], version 2.23-6). `Magma` is a computer algebra software, well-known to be very efficient over finite fields.
- `NTL` ([43], version 10.5.0), installed with `GMP` ([28], version 6.1.2). `NTL` is one of the best library for number theory. It is implemented in `C++`.
- `gf2x` ([14],version 1.2). `gf2x` is a `C` implementation of the state-of-the-art multiplications in $\mathbb{F}_2[X]$. The multiplication algorithm depends on the degree of the operands and the vector instructions set available. When `PCLMULQDQ` is not available, `MQsoft` uses the `gf2x` multiplication.

`Magma` is running with `magma.avx64.dyn.exe` to take advantage of vector instructions. In our tests, the `avx64` version optimizes mainly the performance of multiplication in binary fields.
The measurements used one core of the CPU, and the `C` code was compiled with `gcc -O4 -mavx2 -mpclmul -mpopcnt -funroll-loops`. We use the version 6.4.0 of `gcc`. Turbo Boost and Enhanced Intel Speedstep Technology are disabled to have more accurate measurements, excepted when we use DesktopH and DesktopS. In practice, Turbo Boost generates a speed-up of 1.2 on LaptopS and 1.1 on ServerH.

2 Efficient Arithmetic in $\mathbb{F}_{2^n}$

Arithmetic in $\mathbb{F}_{2^n}$ is the core of the signature generation (Section 1.3) and the computation of $\mathbf{f}$ during the keypair generation (Section 1.2). The main involved parameters (Section 1.1) are n the degree of the field extension, v the number of vinegar variables, D the degree of the `HFEv` polynomial and nb_ite a constant between two and four in GeMSS and `Gui`. In Section 4.1, we explain how to generate efficiently the inner secret polynomials of $\mathbf{f}$ (Equation (3)). This requires $O(n\log_2(D))$ squarings in $\mathbb{F}_{2^n}$, $O(n\log_2(D)(n+v+\log_2(D))$ multiplications in $\mathbb{F}_2[X]$ and $O(n(n+v+\log_2(D)))$ modular reductions. In Section 3, the signature generation requires $O(\text{nb\_ite}\times(nD\log_2(D)^2+D^2))$ field multiplications, $O(\text{nb\_ite}\times nD)$ field squarings and $O(\text{nb\_ite}\times D)$ field inversions.
We use the polynomial representation defined in Section 1.6. It is the most efficient representation when `PCLMULQDQ` is available [45]. To compute the square (respectively the multiplication) of B in $\mathbb{F}_{2^n}$, we choose to compute the square (respectively the multiplication) of B in $\mathbb{F}_2[X]$ before to reduce the reduction by the univariate polynomial defining the extension.

Table 4: Performance of the `PCLMULQDQ` instruction in function of the architecture, as presented in the Intel Intrinsics Guide.

Architecture	Skylake	Broadwell	Haswell	Ivy Bridge
Latency	7	5	7	14
Throughput (CPI)	1	1	2	8

Table 4 presents the cost of `PCLMULQDQ` in function of the architecture. The choice of the best algorithm of multiplication in $\mathbb{F}_{2^n}$ depends on the processor. Our choices target the Skylake processors, which use only one CPI (cycle per instruction).

2.1 Squaring in $\mathbb{F}_{2^n}$

Squaring is used during the root finding algorithm (Section 3) which is the core of the signature generation (Section 1.3). It is also used during the so called Itoh-Tsujii algorithm

[33] which computes the modular inverse in $\mathbb{F}_{2^n}^\times$ (Section 2.5).
In binary fields, the squaring of $B = \sum_{i=0}^{n-1} \alpha_i x^i \in \mathbb{F}_{2^n}$ can be performed in linear time [35]. The linearity of the Frobenius endomorphism implies that $B^2 = \sum_{i=0}^{n-1} \alpha_i x^{2i}$. Since we have stored B as a vector of bits, squaring is the equivalent to insert a null bit between each bit of B.

Example 4. Let $B = x^3 + x^2 + 1 \in \mathbb{F}_{2^4}$. B is stored as the binary integer 1101. Its square is $B^2 = x^6 + x^4 + 1$, which is represented as 1010001.

To compute the square of a n-bit element, we divide it into words of 64 bits. For each one, the `PCLMULQDQ` instruction computes directly the binary polynomial multiplication of the 64-bit element by itself. This method requires $\lceil \frac{n}{64} \rceil$ calls to `PCLMULQDQ`.
We have compared this method to table lookups of square [3]. We have implemented Algorithm 1 of [3] which uses `sse2` instructions and `PSHUFB` from `ssse3`, and an `avx2` version that uses the `VPSHUFB` instruction. The `PSHUFB` instruction performs the search of the square of 16 elements on 4 bits in a table in constant-time, and the `VPSHUFB` instruction performs two times `PSHUFB`. On Skylake, both are less efficient than the `PCLMULQDQ` instruction, whereas on Haswell, we observe the opposite behavior because `PCLMULQDQ` is slower (Table 4).
Table 5 summarizes the performance of squaring functions which are proposed in our library. The experimental process consists to compute the square of elements from a small buffer, then to measure the average cost of one operation. In order to compute the square with table lookups, the `PSHUFB` instructions must be used by pair: one computes the square of lower 4-bit elements of each byte, whereas the second is used for the higher 4-bit elements. For this reason, the squaring performance using `PSHUFB` depends only on $\lceil \frac{n}{128} \rceil$.
For the squaring using `PCLMULQDQ`, the implementation for $\lceil \frac{n}{64} \rceil$ equals to 3 is slower than for $\lceil \frac{n}{64} \rceil$ equals to 4. Indeed, 3 is odd and the 128-bit load and store instructions are less efficient when they are used to load and store a 64-bit element. The best squaring is the one using the `PCLMULQDQ` instruction: on the Skylake processors, it costs only one cycle of throughput, but seven cycles of latency (Table 4). However, the latency can be used to do other instructions, which improves the performance.

Table 5: Number of cycles for computing the square of an element of $\mathbb{F}_2[X]$ of degree $n-1$, with `MQsoft`. We use a Skylake processor (LaptopS).

$\lceil \frac{n}{64} \rceil$	Squaring		
	PSHUFB	VPSHUFB	PCLMULQDQ
1	5.6	×	**2.2**
2	5.7	×	**4.6**
3	8.9	×	**6.6**
4	9.0	7.8	**5.8**
5	13.2	×	**6.9**
6	13.2	×	**7.9**
7	17.5	×	**9.2**
8	17.5	12.2	**10.5**
9	21.9	×	**11.5**

2.2 Multiplication in $\mathbb{F}_{2^n}$

The multiplication of two distinct elements in $\mathbb{F}_{2^n}$ is a central operation involved in the keypair generation and the signing process. In `MQsoft`, we adapt the multiplication algorithm in function of n. We target the Skylake processors.

When $n \leq 384$, we use a schoolbook multiplication by blocks of 64 bits. We use the `PCLMULQDQ` instruction for multiplying each block. Then, $\lceil \frac{n}{64} \rceil^2$ calls to `PCLMULQDQ` are required. This method is naturally in constant-time. Our implementation uses `PCLMULQDQ` which implies to use `sse2` instructions. We use also the `PALIGNR` instruction from `ssse3` to improve the implementation. This instruction concatenates two 128-bit registers and permits to extract 128 bits from the result. We use it to align on 128 bits the results of the multiplications.

When $385 \leq n \leq 576$, an element of $\mathbb{F}_{2^n}$ requires 7, 8 or 9 words, and Karatsuba multiplication [46, section 8.1] becomes faster than the schoolbook method. We split each input in two: the low bits create a 255 degree polynomial, and the remaining bits create a $n-257$ degree polynomial. Thus, the Karatsuba algorithm requires one multiplication of 255 degree polynomials, one multiplication of $n-257$ degree polynomials and one multiplication of $\max(255, n-257)$ degree polynomials. These multiplications are computed with the schoolbook multiplication, requiring $16 + \lceil \frac{n-256}{64} \rceil^2 + \max(16, \lceil \frac{n-256}{64} \rceil^2)$ calls to `PCLMULQDQ`.

The trade-off between the schoolbook multiplication and Karatsuba depends on the performance of `PCLMULQDQ` (Table 4). For the Skylake processors, this instruction costs one CPI, which makes schoolbook multiplication more efficient for $n \leq 384$. For the Haswell processors, `PCLMULQDQ` costs two CPI. This decreases the trade-off because each call to `PCLMULQDQ` is more penalizing. When $\lceil \frac{n}{64} \rceil$ is equals to 3, we remark that the Karatsuba multiplication in $\mathbb{F}_{2^n}$ is already faster on Haswell. In practice, we have compared the schoolbook multiplication to the three-term Karatsuba-like formulae described in [38, Equation 3 with $C = 0$]. The latter is slightly faster and requires only 6 calls to `PCLMULQDQ`.

Table 6 compares our multiplication with `gf2x`. As in Section 2.1, we measure the average cost to multiply elements from a small buffer. The multiplication of `gf2x` is sometimes abnormally slow. This probably dues to the fact that the implementation uses vector and no vector instructions in the same function, which penalizes it. This is probably the first reason to explain that our multiplication is faster. The second reason is that `gf2x` uses Karatsuba, which is slower than schoolbook multiplication for $n \leq 384$ on Skylake. We have also remarked that installing `NTL` with `gf2x` decreases slightly the performance. For this reason, `NTL` is not installed with `gf2x` on our experimental platform.

Table 6: Number of cycles to multiply two elements of $\mathbb{F}_2[X]$ of degree $n-1$. We use a Skylake processor (LaptopS).

$\lceil \frac{n}{64} \rceil$	`gf2x`	`MQsoft`
1	**3.4**	**3.4**
2	7.7	**6.8**
3	37.0	**15.5**
4	23.1	**21.9**
5	47.0	**34.9**
6	54.3	**45.7**
7	142.2	**59.2**
8	91.1	**65.5**
9	131.8	**93.3**

2.3 Modular Reduction and Field Operation in $\mathbb{F}_{2^n}$

In this section, we want to reduce $R = \sum_{i=0}^{2n-2} r_i x^i$ the result of the previous multiplication/squaring in $\mathbb{F}_{2^n}$. The choice of the irreducible polynomial f defining $\mathbb{F}_{2^n}$ (Section 1.6) is important for the modular reduction: it is faster when f is a trinomial or pentanomial [30, 3, 4].

For completeness, we explain here the principle of modular reduction by a trinomial. The method for pentanomials uses the same idea, and it is explained in Appendix A. Let $f_3(x) = x^n + x^{k_1} + 1$ such that $0 < k_1 \leq \lceil \frac{n}{2} \rceil$. Let $R_0 = \sum_{i=0}^{n-1} r_i x^i$, $R_{k_1} = \sum_{i=n}^{2n-k_1-1} r_i x^{i-n}$ and $S_{k_1} = \sum_{i=2n-k_1}^{2n-2} r_i x^{i-2n+k_1}$, we have:

$$R = R_0 + (R_{k_1} + S_{k_1} x^{n-k_1}) x^n. \tag{6}$$

We do a first step of reduction by f_3 by replacing x^n by $f_3(x) - x^n$ in Equation (6). We obtain:

$$R = R_0 + (R_{k_1} + S_{k_1} x^{n-k_1}) + (R_{k_1} x^{k_1} + S_{k_1} x^n) \bmod f_3.$$

We iterate a new step of reduction:

$$R = R_0 + R_{k_1} + S_{k_1} x^{n-k_1} + R_{k_1} x^{k_1} + S_{k_1}(f_3(x) - x^n) \bmod f_3. \tag{7}$$

In Equation (7), the degree of R is $\max(n-1, 2(k_1-1))$. So, R is reduced modulo f_3 only if $2(k_1-1) < n$. In two steps of reduction, we have then a method to compute the modular reduction for all trinomial such that $2(k_1-1) < n$.

To optimize the computation of (7), we factorize by $(f_3(x) - x^n)$. In this way, we can rewrite R as:

$$R = R_0 + S_{k_1} x^{n-k_1} + (R_{k_1} + S_{k_1})(f_3(x) - x^n) \bmod f_3. \tag{8}$$

There are mainly two methods [11] to compute (8). The first is the shift-and-add strategy: $(R_{k_1} + S_{k_1})(f_3(x) - x^n)$ is computed as $Q + Qx^{k_1}$ with $Q = R_{k_1} + S_{k_1}$. The second is the mul-and-add strategy: the multiplication by $(f_3(x) - x^n)$ is computed with the `PCLMULQDQ` instruction. In this case, it is recommended to choose k_1 strictly less than 64. In this way, $(f_3(x) - x^n)$ can be directly used as one of the operand of `PCLMULQDQ` instruction. We choose the first method because it requires a small number of low cost instructions.

These methods can be optimized for specific values of k_1 and n. Firstly, the case $k_1 = 1$ permits to avoid computations because $S_{k_1} = 0$. Secondly, a classical trick is the use of the `PSLLDQ` and `PSRLDQ` instructions which permit to shift 128-bit registers by a multiple of 8 bits. The `PALIGNR` instruction can also be used. As it is explained in Section 1.7, it permits to concatenate two registers 128 bits, then to extract 16 contiguous bytes. We list here cases where these instructions could be used:

- For the extraction of R_{k_1} from R when n is a multiple of 8. However, it does not exist irreducible trinomials such that n is a multiple of 8 [44].
- For the extraction of S_{k_1} from R when $2n - k_1$ is a multiple of 8.
- To obtain $S_{k_1} x^{n-k_1}$ from S_{k_1} when $n - k_1$ is a multiple of 8.
- For the multiplication by x^{k_1} when k_1 is a multiple of 8.

For the shift-and-add strategy, we can also compute $(R_{k_1} + S_{k_1})(f_3(x) - x^n)$ as $\frac{Q'}{x^{k_1}} + Q'$ with $Q' = R_{k_1}x^{k_1} + S_{k_1}x^{k_1}$. In this case, the 128-bit shifts improve the implementation in the following cases:

- For the extraction of $R_{k_1}x^{k_1}$ from R when $n - k_1$ is a multiple of 8.
- For the extraction of $S_{k_1}x^{k_1}$ from R when $2n - 2k_1$ is a multiple of 8.
- To obtain $S_{k_1}x^{n-k_1}$ from $S_{k_1}x^{k_1}$ when $n - 2k_1$ is a multiple of 8.
- For the division by x^{k_1} when k_1 is a multiple of 8.

We have implemented the shift-and-add method for trinomials with different SIMD instruction sets. `ssse3` is used to improve the implementation with the `PALIGNR` instruction. We have also implemented the shift-and-add method for pentanomials, but it is vectorized only for $n \in \{184, 312, 448, 544\}$, because they are the parameters of `Gui` and `DualModeMS256`. For the implementation of `HFE`-based schemes, we estimate that the best strategy is to choose n such that it exists an irreducible trinomial of degree n.
The performance of the modular reduction depends on the context. In Table 7, we reduce products from a small buffer, then we measure the cost of one modular reduction on average. We have removed the optimizations using `PSLLDQ` and `PSRLDQ` to have comparable measurements. In practice, they are used (for example, the `ssse3` modular reduction for $n = 375$ takes 12.2 cycles with these optimizations). The `ssse3` version is two times faster that without vector instructions because `sse2` permits to perform two 64-bit instructions in one instruction. The `avx2` implementation is slightly faster than `ssse3` version, probably because the `avx2` is faster to load and store data.

Table 7: Number of cycles to compute the modular reduction of an element $\mathbb{F}_2[X]$ of degree $2n - 2$ by f, with `MQsoft`. We use a Skylake processor (LaptopS).

(n, k_1)	Rem		
	Without SIMD	`ssse3`	`avx2`
$(62, 29)$	**6.6**	**6.6**	$\times$
$(126, 21)$	10.9	**7.2**	$\times$
$(191, 9)$	14.5	**10.7**	10.8
$(252, 15)$	19.2	11.1	**10.2**
$(314, 15)$	24.1	**14.6**	$\times$
$(375, 16)$	29.5	**14.9**	$\times$
$(441, 7)$	34.2	**17.8**	$\times$
$(511, 10)$	37.1	18.6	**16.9**
$(574, 13)$	40.3	**24.5**	$\times$

In Table 8, the modular reduction is also measured when it is used with multiplication in $\mathbb{F}_2[X]$. The performance of multiplication in $\mathbb{F}_{2^n}$ depends on the context. For this reason, we measure it in two ways:

- Left value: we measure the cost of one field multiplication on average during the computation of the naive exponentiation function (x^i is computed as $x^{i-1}x$). Each result depends on the previous result, and the data are already loaded.
- Right value: we measure the cost of one field multiplication on average to compute the multiplication of elements of two buffers. The data are independent but each multiplication requires to load input and to store output.

Table 8: Number of cycles to compute the multiplication in $\mathbb{F}_{2^n}$ in function of the modular reduction, with `MQsoft`. We use a Skylake processor (LaptopS).

(n, k_1)	Rem			
	`ssse3`		`avx2`	
	exp.	buffer	exp.	buffer
$(62, 29)$	**17.3**	**8.5**	×	×
$(126, 21)$	**26.6**	**14.8**	×	×
$(191, 9)$	**32.8**	**25.9**	44.7	27.2
$(252, 15)$	**40.2**	**35.6**	53.7	36.9
$(314, 15)$	**55.0**	**51.5**	×	×
$(375, 16)$	**65.9**	**65.8**	×	×
$(441, 7)$	**80.2**	**82.8**	×	×
$(511, 10)$	**90.7**	**91.5**	103.7	95.3
$(574, 13)$	**114.9**	**120.3**	×	×

We remark that for n less than 375, the multiplication on independent data is faster. It is probably caused by the latency of the `PCLMULQDQ` instruction. The field multiplication with modular reduction using `sse2` is the fastest, because the `PCLMULQDQ` instruction requires to use 128-bit registers. When `sse2` is used with `avx2`, the implementation pays a penalty. However, this problem will be solved with the `VPCLMULQDQ` instruction on the future Ice Lake processors [1].

In Table 9, the modular reduction is measured when it is used with squaring in $\mathbb{F}_2[X]$. As for the multiplication in $\mathbb{F}_{2^n}$, the performance of squaring depends on the context. For this reason, we measure it in two ways:

- Left value: we measure the cost of one field squaring on average during the raising of an element of $\mathbb{F}_{2^n}$ at the power 2^i (x^{2^i} is computed as $(x^{2^{i-1}})^2$). Each result depends on the previous result, and the data are already loaded.
- Right value: we measure the cost of one field squaring on average to compute the squaring of elements of one buffer. The data are independent but each squaring requires to load input and to store output.

Table 9: Number of cycles to compute the squaring in $\mathbb{F}_{2^n}$ in function of the enabled instructions, with `MQsoft`. We use a Skylake processor (LaptopS).

(n, k_1)	`PSHUFB`		`VPSHUFB`		`PCLMULQDQ`		`PCLMULQDQ`, `avx2`	
	multi-sqr	buffer	multi-sqr	buffer	multi-sqr	buffer	multi-sqr	buffer
$(62, 29)$	**14.9**	9.2	×	×	17.3	**7.6**	×	×
$(126, 21)$	**18.6**	11.8	×	×	21.5	**10.0**	×	×
$(191, 9)$	24.0	18.4	×	×	**23.3**	**14.1**	35.0	15.7
$(252, 15)$	25.2	20.0	30.8	16.8	**23.9**	**14.7**	37.4	16.3
$(314, 15)$	29.1	26.3	×	×	**25.2**	**22.0**	×	×
$(375, 16)$	28.9	27.0	×	×	**25.8**	**21.3**	×	×
$(441, 7)$	34.8	34.8	×	×	**27.4**	**25.1**	×	×
$(511, 10)$	38.4	36.4	35.7	28.7	**26.7**	**24.2**	39.8	28.0
$(574, 13)$	45.5	43.5	×	×	**29.7**	**31.2**	×	×

Table 9 shows the performance of squaring in $\mathbb{F}_{2^n}$. The squaring using `PCLMULQDQ` is the most efficient. For the same reasons that for the multiplication in $\mathbb{F}_{2^n}$, the best modular squaring is the one using only `sse2` modular reduction. This is the default setting in `MQsoft`.

All measured functions in this Section are available is our library. They are implemented in constant-time. The modular reduction by pentanomials is also available to make the library more complete, but is not yet vectorized (excepted for $n \in \{184, 312, 448, 544\}$).

2.4 Multi-squaring in $\mathbb{F}_{2^n}$

The multi-squaring [45] is an operation computing successively several squarings. This operation is important to compute the inverse in $\mathbb{F}_{2^n}^\times$ (in Section 2.5). Algorithm 5 requires to compute B^{2^i}, for $B \in \mathbb{F}_{2^n}$ and various values of i. For small i, the best way is to raise B at the power two i times (as in Section 2.3). For larger i, the best method is to use precomputed multi-squaring tables. Let $B = \sum_{k=0}^{n-1} \alpha_k x^k \in \mathbb{F}_{2^n}$ for $\mathbb{F}_{2^n} = \mathbb{F}_2/(f(x))$ (Section 1.6), then $B^{2^i} = (\sum_{k=0}^{n-1} \alpha_k x^{k2^i}) \bmod f$. The idea of multi-squaring tables is to store $x^{k2^i} \bmod f$ for $k \in [\![0, n-1]\!]$. Then, multi-squaring is equivalent to the dot product of the vectors $(\alpha_0, \dots, \alpha_{n-1})$ and $(1, x^{2^i} \bmod f, \dots, x^{(n-1)2^i} \bmod f)$. The table requires to store $n-1$ elements in $\mathbb{F}_{2^n}$ (1 is not formally stored), and the multi-squaring requires $n-1$ multiplications between elements of $\mathbb{F}_2$ and $\mathbb{F}_{2^n}$ and $n-1$ additions in $\mathbb{F}_{2^n}$. In a variable-time implementation, the multiplication by α_k can be done by a conditional statement. In a constant-time implementation, the value of α_k is duplicated in the mask variable, in the way to replace the multiplication by a bitwise `AND` with this mask. This process is explained in Section 4.2.

In variable-time implementation, the performance can be improved with larger tables [36]. Rather that to compute $\alpha_k x^{k2^i} \bmod f$ coefficient by coefficient, the coefficients can be grouped by block of B, and the 2^B possibility of $\sum_{k=0}^{B-1} \alpha_{jB+k} x^{(jB+k)2^i}$ can be precomputed for $j \in [\![0, \lceil \frac{n}{B} \rceil - 1]\!]$. This method cannot be used in a constant-time implementation because of the timing attack on the memory latency. It permits to attack the index of the precomputed table. In our implementation, we use a constant-time implementation of multi-squaring.

2.5 Modular Inverse in $\mathbb{F}_{2^n}^\times$

The computation of the inverse in $\mathbb{F}_{2^n}^\times$ is often required for the arithmetic in $\mathbb{F}_{2^n}[X]$. In our case, it is required to compute the `GCD` (Section 3.6). To compute the modular inverse of $A \in \mathbb{F}_{2^n}^\times$, there are mainly two methods. The first is to use extended Euclidean algorithm (EEA) [46, Algorithm 3.6]. This method is not constant-time. The second is to compute $A^{-1} = A^{2^n-2}$ by Fermat's little theorem. The exponentiation can be done with the square and multiply method [46, Algorithm 4.8], costing $n-1$ squarings and $n-2$ multiplications in $\mathbb{F}_{2^n}$. The Itoh-Tsujii Multiplicative Inversion Algorithm (ITMIA) [33] permits to modify the way to compute the power with an addition chain. It requires $n-1$ squarings and only $O(\log_2(n))$ multiplications in $\mathbb{F}_{2^n}$. The number of multiplications depends on the length of the chosen addition chains. ITMIA is described in Algorithm 5 for a specific addition chain which consists to read the bits of $n-1$ from MSB to LSB. At the end of the each iteration, the variable `inv` is always $A^{2^{\mathtt{val}}-1}$.

Algorithm 5 ITMIA for a specific addition chain.

```
function Inverse($A \in \mathbb{F}_{2^n}^{\times}$)
    val ← 1                                                    ▷ Case $i = \lfloor \log_2(n-1) \rfloor$
    inv ← $A$                                                  ▷ $A^{2^{\texttt{val}}-1} = A$
    for $i$ from $\lfloor \log_2(n-1) \rfloor - 1$ to 0 by $-1$ do
        tmp ← $(\texttt{inv})^{2^{\texttt{val}}}$              ▷ Multi-squaring to obtain $A^{2^{2\texttt{val}}-2^{\texttt{val}}}$.
        inv ← tmp × inv                                        ▷ $A^{2^{2\texttt{val}}-1}$.
        val ← $\left\lfloor \frac{n-1}{2^i} \right\rfloor$
        if ((val mod 2) == 1) then
            inv ← $(\texttt{inv})^2 \times A$                  ▷ $(A^{2^{\texttt{val}-1}-1})^2 \times A = A^{2^{\texttt{val}}-1}$
        end if
    end for                                                    ▷ $\texttt{val} = n-1$ and so $\texttt{inv} = A^{2^{n-1}-1}$
    return $(\texttt{inv})^2$                                  ▷ $A^{2^n-2} = A^{-1}$
end function
```

ITMIA requires to compute `val` successive squarings. The multi-squaring can be computed more quickly with precomputed tables (cf. Section 2.4).

Algorithm 5 is useful because it proposes automatically an addition chain for all values of n, but it is not always optimal. To choose the best addition chain is not easy, because it depends on the performance of multiplication, squaring and multi-squaring. Moreover, there is a large set of possible addition chains. This problem is studied in [36], which proposes a software generating an efficient `C++` inversion code. This software searches the addition chain which maximizes the performance of the generated code. However, the generator does not proposes implementation of multi-squaring tables in constant-time. For the moment, `MQsoft` uses Algorithm 5, but we propose in Appendix B examples of addition chains chosen to minimize the number of field multiplications. We have improved Algorithm 5 with multi-squaring tables to compute the variable `tmp` when i is zero or one. Because multi-squaring tables are huge, we use it only for the parameters of G*e*MSS, `Gui`, DualModeMS, and for the values of n used to evaluate the performance of `MQsoft`. The corresponding file in `MQsoft` requires 1.7 `MB` for 44 tables.

2.6 Performance of the Arithmetic in $\mathbb{F}_{2^n}$

Table 10 compares the performance of arithmetic operations in our library with respect to several open source libraries (listed in Section 1.7). We choose the irreducible trinomial $f(x) = x^n + x^{k_1} + 1$ with $k_1 \in [\![2, 32]\!]$ to create the field $\mathbb{F}_{2^n}$. All operations use modular reduction. We have measured the performance of `FLINT` [32, version 2.5.3] (it is a `C` library), but the times are not relevant in our context. It turns that for $n = 252$, `NTL` is 100 to 200 times faster than `FLINT`. The main reason is that `FLINT` does not have special implementation for binary fields. We have used the type `fq_nmod_t` which store each element of $\mathbb{F}_{2^n}$ as a polynomial in $\mathbb{F}_2[X]$ where each coefficient is stored on one word. `Magma` is also taken into account. The results are not significant because `Magma` is slowed down by its user interface. We remark that the squarings and multiplications of `NTL` are faster for $n = 126$ than for $n = 62$. It can be probably explained by the fact that `NTL` does not use trinomial for $n = 62$. Our implementation is 3.5 to 4.5 times faster than `NTL` for multiplication and 5 to 6 times faster for squaring. We think that `NTL` is slowed down by its `C++` interface. For the inversion, the measurements are not comparable because `NTL` is not in constant-time. However, we have a speed-up of two on average.
We compare now `MQsoft` to the constant-time arithmetic of [11], when trinomials are used to build $\mathbb{F}_{2^n}$. In $\mathbb{F}_{2^{233}}$, they compute the squaring in 18 cycles and the multiplication in 38 cycles. We have approximately the same performance for $n = 252$: `MQsoft` is slower with

Table 10: Number of cycles by operation in $\mathbb{F}_{2^n}$. We use a Skylake processor (LaptopS).

(n, k_1)	Operation	`Magma`	`NTL`	`MQsoft` (`PCLMULQDQ` + `avx2`)	
				dependencies	buffer
$(62, 29)$	Squaring	416	220	**14.9**	**7.6**
	Mul	444	231	**17.3**	**8.5**
	Inverse	13,183	1,868	$\times$	**1,154**
$(126, 21)$	Squaring	440	105	**18.6**	**10.0**
	Mul	494	124	**26.6**	**14.8**
	Inverse	27,353	3,457	$\times$	**1,731**
$(191, 9)$	Squaring	437	119	**23.3**	**14.1**
	Mul	529	144	**32.8**	**25.9**
	Inverse	40,200	4,918	$\times$	**2,706**
$(252, 15)$	Squaring	455	128	**23.9**	**14.7**
	Mul	558	169	**40.2**	**35.6**
	Inverse	51,720	7,809	$\times$	**3,647**
$(314, 15)$	Squaring	480	139	**25.2**	**22.0**
	Mul	629	211	**55.0**	**51.5**
	Inverse	66,220	9,515	$\times$	**5,096**
$(375, 16)$	Squaring	490	150	**25.8**	**21.3**
	Mul	653	238	**65.9**	**65.8**
	Inverse	76,704	11,813	$\times$	**6,364**
$(441, 7)$	Squaring	500	163	**27.4**	**25.1**
	Mul	714	286	**80.2**	**82.8**
	Inverse	94,846	14,974	$\times$	**7,755**
$(511, 10)$	Squaring	510	174	**26.7**	**24.2**
	Mul	761	320	**90.7**	**91.5**
	Inverse	115,681	18,601	$\times$	**8,825**
$(574, 13)$	Squaring	521	201	**29.7**	**31.2**
	Mul	922	579	**114.9**	**120.3**
	Inverse	129,266	23,185	$\times$	**11,329**

dependencies but faster with buffers. In $\mathbb{F}_{2^{409}}$, they compute the squaring in 28 cycles and the multiplication in 97 cycles. `MQsoft` is slightly faster (we compare to the measurements in $\mathbb{F}_{2^{441}}$). For the inversion, [11] is approximately two times slower. Our library takes advantage of using of multi-squaring tables. We replace $n - 1$ squarings by approximately $\frac{1}{4}n$ squarings and two multi-squarings.

3 Efficient Implementation of Root Finding in $\mathbb{F}_{2^n}[X]$

The most expensive part of the signature generation is to find the roots of a `HFE` polynomial $F \in \mathbb{F}_{2^n}[X]$ as defined in Equation (2). F is a D degree monic polynomial which is sparse because it has approximately $\frac{\log_2(D)^2}{2}$ non-zero coefficients. We have chosen to implement Berlekamp's algorithm [46, Algorithm 14.15] which finds the roots with an asymptotic complexity of $O(nD^2 + (n + \log(s))s^2 \log(s))$ operations in $\mathbb{F}_{2^n}$, where s is the number of roots of F [46, Theorem 14.11 adapted for $r = s$ and $d = 1$]. For `HFE` polynomials, the factor $O(nD^2)$ can be easily improved in $O(nD \log_2(D)^2 + D^2)$ operations in $\mathbb{F}_{2^n}$, by using the sparse structure of F (Section 3.5). Moreover, the `HFE` polynomial does not have many roots, so we can assume that s is negligible, yielding a final complexity of $O(nD \log_2(D)^2 + D^2)$ operations in $\mathbb{F}_{2^n}$.

For the general polynomials, the author of [41] proposes in 2014 the successive resultant algorithm (SRA). It requires $O(n^3D^2+n^4)$ operations in $\mathbb{F}_2$ to find roots, or $\widetilde{O}(n^2D+n^3)$ with the fast arithmetic. The step in $O(n^4)$ (or $\widetilde{O}(n^3)$) can be precomputed for a fixed finite field. In comparison to Berlekamp, SRA is interesting only when the polynomial has many roots. In [29] and [22], the root finding is improved for split and separable polynomials, and when the cardinality of multiplicative group is smooth. In our case, this method is not interesting because the HFE polynomials does not have many roots.
Improving root finding for sparse polynomials is a hard problem. In [10], the authors propose the first sub-linear (in q) algorithm which detects the existence of roots for t-monomials in $\mathbb{F}_q[X]$. The complexity is of $4^{t+o(1)}q^{\frac{t-2}{t-1}+o(1)}$ bit operations. This method is not interesting for a HFE polynomial because it is not enough sparse and also because in practice n is greater that the level of security. The algorithm costs approximately $4^{o(1)}2^{n+o(n)}D^{\log_2(D)}$ bit operations in our case.
In this section, we will remind the Berlekamp's algorithm. For each operation required, we study the different possible methods and compare their practical performance to choose the best. We specify how these methods can be tuned for a HFE polynomial.

3.1 Description of Berlekamp's Algorithm in $\mathbb{F}_{2^n}[X]$

Algorithm 6 describes Berlekamp's algorithm [46, Algorithm 14.15]. The main idea is to remark that all elements of $\mathbb{F}_{2^n}$ vanish on $X^{2^n}-X$. We can then compute G the GCD on F with $X^{2^n}-X$. G has the same roots than F but with a minimal degree (which is the number of roots). In general, the degree of G is small. The strategy is then to apply the so-called equal-degree factorization to find all roots. This is turned to be cheap. Indeed, let s be the degree of G, the equal-degree factorization costs $(n+\log(s))s^2\log(s)$ operations in $\mathbb{F}_{2^n}$ [46, Theorem 14.11 adapted for $r=s$ and $d=1$]. Because the degree of $X^{2^n}-X$ is big, we reduce $X^{2^n}-X$ by F by using the repeated squaring algorithm in $\mathbb{F}_{2^n}[X]$, before computing the GCD.

Algorithm 6 Algorithm to find the roots of a univariate polynomial.

```
function FindRoots($F \in \mathbb{F}_{2^n}[X]$)
    $X_n \leftarrow X^{2^n} - X \bmod F$        ▷ Step 1: Computation of the Frobenius map.
    $G \leftarrow \mathtt{GCD}(F, X_n)$        ▷ Step 2: Computation of the GCD.
    if degree($G$) $> 0$ then
        Roots ← List of all roots of $G$, computed by the equal-degree factorization
    algorithm described in [46, Section 14.3].        ▷ Call to Algorithm 7.
        return (degree($G$), Roots)
    end if
    return (degree($G$), $\emptyset$)
end function
```

Thereafter, we will study the choice of the algorithms only for the steps one and two. The computation of equal-degree factorization is negligible since G has a small degree. For the set of completeness, the equal-degree factorization algorithm is summarized in Algorithm 7. We will compare the classical and fast algorithms and we optimize them for a HFE polynomial which is sparse, i.e. which has $O(\log_2(D^2))$ coefficients in comparison to a dense polynomial that has $D+1$ coefficients.

3.2 Polynomial Squaring in $\mathbb{F}_{2^n}[X]$

Step 1 of Algorithm 6 requires to compute repeated squarings in $\mathbb{F}_{2^n}[X]$. In binary fields, squaring of $B=\sum_{i=0}^{D-1}\alpha_iX^i \in \mathbb{F}_{2^n}[X]$ is linear. Similarly to Section 2.1, it holds that

Algorithm 7 Algorithm to find the roots of a split monic univariate polynomial.

```
function FindRootsSplit($F \in \mathbb{F}_{2^n}[X]$)
    if degree($F$) $< 1$ then
        return $\emptyset$
    else if degree($F$) $== 1$ then
        return List($F.cst$)    ▷ $F.cst$ is the constant term of $F$, we create a list with the root of $F$ and return it.
    else
        repeat
            $r \in_R \mathbb{F}_{2^n}$    ▷ The notation $\in_R$ stands for randomly sampling.
            $H \leftarrow \sum_{i=1}^{n-1}((rX)^i \bmod F)$
            $G \leftarrow$ GCD($F, H$)    ▷ We can assumed that $G$ is chosen monic.
        until $G$ is a non trivial divisor of $F$
        return Concat(FindRootsSplit($G$),FindRootsSplit($\frac{F}{G}$))    ▷ The concatenation of the lists is returned.
    end if
end function
```

$B^2 = \sum_{i=0}^{D-1} \alpha_i^2 X^{2i}$. This operation requires D squaring in $\mathbb{F}_{2^n}$.

3.3 Euclidean Division in $\mathbb{F}_{2^n}[X]$

Let $P \in \mathbb{F}_{2^n}[X]$ be a univariate polynomial. The notation $\text{coef}_i(P)$ will denote the coefficient of the term of degree i in P. Step 1 of Algorithm 6 requires to compute the modular reduction of a polynomial P of degree at most $2D - 2$ by a monic polynomial F. The classical algorithm [46, Algorithm 2.5] uses $O(D^2)$ multiplications in $\mathbb{F}_{2^n}$.

Algorithm 8 computes the modular reduction specialized for HFE polynomials. Let K be the number of terms of F_{HFE} (Section 1.6). For a fixed i, each term of $(F_{\text{HFE}} - X^D)X^{i-D}$ is multiplied by $-\text{coef}_{i-D}(Q)$. Then, the result is added to R, requiring $K-1$ multiplications and additions in $\mathbb{F}_{2^n}$. Since $K - 1 = O(\log_2(D)^2)$ coefficients, the Euclidean division requires $(D-1)(K-1) = O(D\log_2(D)^2)$ multiplications in $\mathbb{F}_{2^n}$. This method is in constant-time because we compute the multiplication by $\text{coef}_{i-D}(Q)$ even when it is null.

Algorithm 8 Euclidean division by a monic polynomial.

```
function Euclidean($P, F_{\text{HFE}} \in \mathbb{F}_{2^n}[X]$)
    $R \leftarrow P$
    for $i$ from $2D - 2$ to $D$ by $-1$ do
        $\text{coef}_{i-D}(Q) \leftarrow \text{coef}_i(R)$
        $R \leftarrow R - \text{coef}_{i-D}(Q) \times (F_{\text{HFE}} - X^D)X^{i-D}$    ▷ Computation in constant-time.
        $\text{coef}_i(R) \leftarrow 0$    ▷ The new $R$ has a degree at most $i - 1$.
    end for
    return $Q, R$
end function
```

3.4 Improving Euclidean Division for Special HFE Polynomials

The previous method does not exploit that during Step 1 of Algorithm 6, the dividend is a square. Terms of odd degree are null. We show here that the complexity can be divided by two, in function of the term of higher odd degree of the divisor. For this, we introduce a new notation. Let $P \in \mathbb{F}_{2^n}[X]$, we denote by $\mathcal{D}(P)$ the largest odd integer i such that

$\mathrm{coef}_i(P) \neq 0$. If it does not exist, we set $\mathcal{D}(P) = -\infty$. The following lemma permits to demonstrate the main result (Theorem 1) of this part.

Lemma 1. *Let $A \in \mathbb{F}_{2^n}[X]$ be a polynomial of degree at most $2D-2$ such that $\mathcal{D}(A) = -\infty$, $H \in \mathbb{F}_{2^n}[X]$ be of degree D, $Q, R \in \mathbb{F}_{2^n}[X]$ be respectively the quotient and remainder of the Euclidean division of A by H and $d = \mathcal{D}(H)$. If D is even then $\mathcal{D}(Q) \leq d-2$.*

Proof. Let $H = \sum_{j=0}^{D} h_j X^j$ and $Q = \sum_{j=0}^{D-2} q_j X^j$. By definition of $\mathcal{D}$, $\mathcal{D}(Q) \leq d-2$ is equivalent to $q_i = 0$ for all odd i such that $i > d-2$. By definition of Q, $q_i = 0$ for $i < 0$ and $i > D-2$, so we show the lemma for the values of q_i such that $D-2 \geq i > \max(-1, d-2)$. For this, we use a proof by induction on an odd j such that $D-1 \geq j > \max(-1, d-2)$. The base case is trivial since $q_j = 0$ for $j > D-2$. Assume that $q_k = 0$ for all odd k such that $D-2 \geq k > j > \max(-1, d-2)$. To show that $q_j = 0$, firstly we show these two properties:

(1) $\mathrm{coef}_{D+j}(HQ) = 0$.

(2) $\mathrm{coef}_{D+j}(HQ) = q_j h_D$.

Proof of these two properties:

(1) By definition, $A = HQ + R$ and so $A - R = HQ$. Because $\mathcal{D}(A) = -\infty$ by hypothesis, $\mathcal{D}(A-R) = \mathcal{D}(R) \leq D-1 < D+j$ and $D+j$ is odd so $\mathrm{coef}_{D+j}(A-R) = 0$.

(2) $HQ = \sum_{r=0}^{2D-2} \sum_{\ell=0}^{r} q_\ell h_{r-\ell} X^r$, so $\mathrm{coef}_{D+j}(HQ) = \sum_{\ell=0}^{D+j} q_\ell h_{D+j-\ell}$. But $q_\ell = 0$ for $\ell > D-2$ and $h_{D+j-\ell} = 0$ for $D+j-\ell > D$, so $\mathrm{coef}_{D+j}(HQ) = \sum_{\ell=j}^{D-2} q_j h_{D+j-\ell}$. When $\ell > j$ is odd, $q_\ell = 0$ by induction hypothesis. When ℓ is even, $h_{D+j-\ell} = 0$ because $D+j-\ell$ is odd and $\mathcal{D}(H) = d < D+j-\ell$. So $\sum_{\ell=j}^{D-2} q_\ell h_{D+j-\ell} = q_j h_D$.

These two properties implies $\mathrm{coef}_{D+j}(HQ) = q_j h_D = 0$. Because $h_D \neq 0$, this implies that $q_j = 0$. □

We can now demonstrate Theorem 1.

Theorem 1. *Let H be a `HFE` polynomial of degree D in $\mathbb{F}_{2^n}[X]$ where the k-th terms of highest odd degree have been removed ($k \in [\![0, \lceil \log_2(D) \rceil - 1]\!]$), and let $A \in \mathbb{F}_{2^n}[X]$ be a square of degree at most $2D-2$. If D is even, then the computation of the classical Euclidean division (Algorithm 8) of A by H can be accelerated by a factor $(D-1)/(\frac{D}{2} + \lfloor 2^{\lceil \log_2(D) \rceil - k - 2} \rfloor)$.*

Proof. During Algorithm 8, A is a square so $\mathcal{D}(A) = -\infty$, and so Lemma 1 can be applied it. Let $d = \mathcal{D}(H)$, the iterations where i is odd and strictly greater than $D+d-2$ can be removed because $\mathrm{coef}_{i-D}(Q) = 0$. So, the number of iterations when i is odd is $\max(\frac{(D+d-2)-(D-1)}{2}, 0) = \max(\frac{d-1}{2}, 0)$, whereas the number of iterations when i is even is $\frac{D}{2}$. So, Algorithm 8 can be used with $\max(\frac{D+d-1}{2}, \frac{D}{2})$ iterations. Next, H is a `HFE` polynomial so $d = 1$ or $d = 2^j + 1$ for $j > 0$. By removing the $2^j + 1$ degree terms for j from $\lceil \log_2(D) \rceil - 1$ to $\lceil \log_2(D) \rceil - k$ by -1, d equals 1 or $2^{\lceil \log_2(D) \rceil - k - 1} + 1$. It implies that the number of iterations can be written as $\frac{D}{2} + \lfloor 2^{\lceil \log_2(D) \rceil - k - 2} \rfloor$. Algorithm 8 requires $D-1$ iterations, so the proposed modification accelerates it of a factor $(D-1)/(\frac{D}{2} + \lfloor 2^{\lceil \log_2(D) \rceil - k - 2} \rfloor)$. This factor is at most 2.

□

Let K be the number of terms of the `HFE` polynomial (without removed terms), and k be the number of removed terms. For $k = 0$, the modular reduction costs $(D-1)(K-1)$ multiplications in $\mathbb{F}_{2^n}$ (Section 3.3), whereas by removing terms (with even D), the cost is $\max(\frac{D+d-1}{2}, \frac{D}{2})(K-1-k)$ multiplications. The main gain comes from the fact to decrease the number of round loops during Algorithm 8. However, there is also a slight speed-up generated by the fact that terms are removed.

When F has exactly zero term of odd degree, we obtain that during the computation of $X^{2^n} \bmod F$, none of the odd degree terms appear because $R = A - FQ$ and A, F and Q do not have odd degree terms. This result allows to do computations only for even degree terms, dividing by 2 the cost of the squaring and of the modular reduction. But in practice, to remove all terms of odd degree of the `HFE` polynomial decreases the security. By applying to F the change of variable $X^2 = Y$, the degree of the result is only $\frac{D}{2}$. We will show in Table 11 that in this case, D must be multiplied by two to obtain the original security.

Table 11 studies the impact of $d = \mathcal{D}(F)$ on the theoretical speed-up over the classical Euclidean division compared to the case $D = 513$, and on the security. We have done an experimental test to analyse the degree of regularity [24, 5, 6, 9] in function of the number of removed terms. The degree of regularity is a tool to analyse the security of `HFE`-based scheme against Gröbner basis attacks. We measure it during the Gröbner basis attack on `HFE` for $n = m = 30$. We observe in practice that removing a small number of odd degree terms appears not to affect the security. The security is decreased when the second to last term is removed. The results confirm that the security to attack F of degree D without odd degree terms is the same that attack a `HFE` polynomial of degree $\frac{D}{2}$: the degree of regularity increases between $\frac{D}{2} = 512$ and $\frac{D}{2} = 513$. The case $d = 1$ seems to have the same behavior, but in the general case, the regularity degree does not decrease necessarily (for $D = 130$, the degree of regularity is 4 for $d = -\infty$ but 5 for $d = 1$).
The column speed-up on i corresponds to obtain speed-up by decreasing the number of iterations during Algorithm 8. The other column speed-up is the total speed-up, which uses the fact that remove terms decreases the number of multiplications for one iteration. To remove the higher terms generates the main part of the maximal speed-up. In practice, we propose to choose $d = 63$. This implies to remove one term when $D = 130$ and three terms when $D = 514$.

Table 11: Impact of d on the performance and on d_{reg} the degree of regularity.

D	d	removed terms	d_{reg}	nb. of iterations	speed-up on i	speed-up
512	257	none	5	384	25%	27%
513	513	none	6	512	Ref.	Ref.
514	257	X^{513}	6	385	25%	25%
	129	X^{513}, X^{257}		321	37%	39%
	65	$X^{513}, X^{257}, X^{129}$		289	44%	46%
	33	$X^{513}, X^{257}, X^{129}, X^{65}$		273	47%	50%
	17	$X^{513}, X^{257}, X^{129}, X^{65}, X^{33}$		265	48%	53%
	9	$X^{513}, X^{257}, X^{129}, X^{65}, X^{33}, X^{17}$		261	49%	54%
	5	all odds excepted X^5, X^3, X		259	49%	56%
	3	all odds excepted X^3, X		258	50%	57%
	1	all odds excepted X	5	257	50%	58%
	$-\infty$	all odds		257	50%	59%
1024	1	all odds excepted X	5	512	0%	0%
	$-\infty$	all odds		512	0%	2%
1026	1	all odds excepted X	6	513	0%	−4%
	$-\infty$	all odds		513	0%	−2%

3.5 Frobenius Map in $\mathbb{F}_{2^n}[X]$

The core of Algorithm 6 (Section 3.1) is to compute $X^{2^n} \bmod F$ during Step 1. As in Section 2.4, we compare the classical repeated squaring algorithm with the version using multi-squaring tables. The main differences with Section 2.4 is that the coefficients are not in $\mathbb{F}_2$ but in $\mathbb{F}_{2^n}$. So, the tables are too large to be precomputed. However, they can be computed more quickly by exploiting the HFE structure of F. Both presented methods are in constant-time.

Classical repeated squaring algorithm. We can compute $X^{2^n} \bmod F$ using repeated squaring algorithm [46, Algorithm 4.8]. This requires n steps of modular squaring. More precisely, the number of steps is $n - \lfloor \log_2(D) \rfloor$ because the modular reduction is useless when the degree of X^{2^i}, $1 \leq i \leq n$, is less than D. So, we compute $(X^{2^{\lfloor \log_2(D) \rfloor}})^{2^{n-\lfloor \log_2(D) \rfloor}} \bmod F$ (or $(F - X^D)^{2^{n-\lfloor \log_2(D) \rfloor}} \bmod F$ when D is a power of two). This remark permits to avoid useless computations in the constant-time implementations. This method requires $(n - \lfloor \log_2(D) \rfloor)D$ field squarings and $(n - \lfloor \log_2(D) \rfloor)$ call to Algorithm 8 (Section 3.3). It does $(n - \lfloor \log_2(D) \rfloor)(D-1)(K-1) = O(nD \log_2(D)^2)$ field multiplications. This method can be improved with the trick from Section 3.4. To improve the performance, we compute the repeated squaring in-place. To do it, we allocate a buffer of $2D - 1$ coefficients in $\mathbb{F}_{2^n}$. The squaring and the modular reduction modify directly the current result.

Repeated squaring algorithm with multi-squaring tables. The authors of [42] propose to compute several squarings before reducing by F. Set i this number of times. To compute i squarings creates a result of degree $(D-1)2^i$, but only the terms X^{j2^i} for $j \in \{0, \ldots, D-1\}$ are not null. To compute the reduction, firstly compute one time a table of $(X^{j2^i} \bmod F)$ for $j \in \{0, ..., D-1\}$, then multiply each coefficient by the corresponding element in the table. The table is computed one time for all and is re-used for each modular reduction.
To create the table, we compute each $X^{j2^i} \bmod F$ as $(X^{(j-1)2^i} \bmod F)X^{2^i} \bmod F$. The multiplication by X^{2^i} is just a shift of 2^i, and all terms of degree strictly greater than $D-1$ are reduced with Algorithm 8 (by replacing $2D-2$ by $D-1+2^i$). The table is useful only when X^{j2^i} is not already reduced by F, so when $2^i j \geq D$ and implies $j \geq \lceil \frac{D}{2^i} \rceil$. This table requires to store $(D - \lceil \frac{D}{2^i} \rceil)D$ elements of $\mathbb{F}_{2^n}$, and $D - \lceil \frac{D}{2^i} \rceil$ calls to Algorithm 8 are required to generate it, costing $O(2^i(K-1)(D - \lceil \frac{D}{2^i} \rceil))$ field multiplications.
To compute $X^{2^n} \bmod F$, we take $X^{2^{\lfloor \log_2(D-1) \rfloor + i}} \bmod F$ from the table, then we compute $\left\lfloor \frac{n-i-\lfloor \log_2(D-1) \rfloor}{i} \right\rfloor$ steps of modular multi-squarings. Each step requires to raise D elements of $\mathbb{F}_{2^n}$ at the power 2^i, then to multiply each by the corresponding elements of the table. It costs iD field squarings and $(D - \lceil \frac{D}{2^i} \rceil)D$ field multiplications. To obtain $X^{2^n} \bmod F$, we terminate by $((n - i - \lfloor \log_2(D-1) \rfloor) \bmod i)$ steps of modular squarings with the classical repeated squaring algorithm.
The final cost of this method is $(n - i - \lfloor \log_2(D-1) \rfloor)D = O(nD)$ field squarings and $(D - \lceil \frac{D}{2^i} \rceil)(2^i(K-1) + D \left\lfloor \frac{n-i-\lfloor \log_2(D-1) \rfloor}{i} \right\rfloor) + ((n-i-\lfloor \log_2(D-1) \rfloor) \bmod i)(D-1)(K-1) = O(2^i D \log_2(D)^2 + \frac{n}{i} D^2)$ field multiplications.

To choose the best algorithm for the Frobenius map, we just choose the one which minimizes the number of field multiplications. In practice, the repeated squaring algorithm is the best when approximately $D \geq n$, whereas the multi-squaring version is the best when $n > D$.

Table 12 summarizes the performance of both strategies for the Frobenius map, and compares our implementation to NTL and Magma. We use the Modexp function from Magma and the `PlainFrobeniusMap` function from `NTL`, both computing $X^{2^n} \bmod F$. We have studied also the strategy from Section 3.4 which permits to improve the Frobenius map by removing odd degree terms in the `HFE` polynomial. We choose $d = 65$, which requires to remove one term when $D = 130$ and three terms when $D = 514$. The results confirm the theoretical speed-ups: `MQsoft` saves approximately 25% and 44% of computations by removing respectively one and three terms for the first strategy. `Magma` is also improved by this trick, probably because it uses also the classical Euclidean division, and does not compute multiplication by zero. It is not the case for `NTL` because it uses the fast Euclidean division [46, Algorithm 9.5].

The multi-squaring strategy is the fastest when D is small compared to n. However, for $n = 354$ and $D = 129$, to set $d = 65$ is enough to change the trade-off between both strategies. To remove odd degree terms is interesting for `HFE`-based NIST submissions which uses D equals to 129 or more. However, this implies to increase by one the original parameters ($D = 129$ and $D = 513$). Without modifying it, our best Frobenius map is 7 to 12 times faster than `NTL`.

Table 12: Number of mega cycles to compute the Frobenius map of a `HFE` polynomial. We use a Skylake processor (LaptopS).

n	D	d	`Magma`	`NTL`	`MQsoft` (repeated squarings)	`MQsoft` (multi-squaring)
185	33	33	36.3	13.0	2.6	**1.4**
	129	129	169.3	136.5	**17.2**	18.1
	130	65	138.4	142.2	**12.9**	18.6
	513	513	1,000.8	1,092.7	**108.9**	223.7
	514	65	629.8	1,104.8	**59.0**	223.0
354	33	33	108.5	76.2	13.2	**6.2**
	129	129	559.0	551.0	93.3	**79.9**
	130	65	451.4	628.6	**70.9**	81.9
	513	513	3,371.0	4,107.0	**566.0**	1,061.5
	514	65	2,019.0	4,383.3	**312.1**	1,053.6

3.6 `GCD` in $\mathbb{F}_{2^n}[X]$

Step 2 of Algorithm 6 requires to compute the `GCD` of two degree D polynomials in $\mathbb{F}_{2^n}[X]$. The `NTL` library provides only the classical algorithm [46, Algorithm 3.5], which uses $O(D^2)$ field multiplications and $O(D)$ field inversions. We have implemented the half-`GCD` algorithm ([46, Algorithm 11.8],[13, Algorithm 6.8]), which uses $\widetilde{O}(D)$ multiplications in $\mathbb{F}_{2^n}$. Our implementation of the half-`GCD` is based on the Karatsuba polynomial multiplication (Appendix C) and on the fast Euclidean division (Appendix D). These algorithms are implemented with constant-time arithmetic, but the `GCD` is in variable-time because the number of successive remainders is variable.

Table 13 compares the performance of `GCD` algorithms. Our classical `GCD` is three to four times better than `NTL`. This results of the difference of performance between our operations in $\mathbb{F}_{2^n}$. The half-`GCD` becomes faster only for a high degree (approximately 8193).

Table 13: Number of mega cycles to compute the `GCD` in $\mathbb{F}_{2^n}[X]$ of a D degree polynomial by a $D-1$ degree polynomial. We use a Skylake processor (LaptopS).

n	D	`Magma`	`NTL`	`MQsoft` (gcd)	`MQsoft` (half-gcd)
185	33	2.2	0.620	**0.121**	0.535
	129	12.0	3.735	**0.719**	3.661
	513	129.9	39.8	**7.6**	30.2
	4097	3,060	2,057	**419**	726
	8193	9,485	8,131	**1,838**	2,168
	16385	27,168	32,175	7,304	**6,407**
354	33	4.1	1.350	**0.276**	1.019
	129	23.0	7.973	**1.870**	7.303
	513	229.4	82.4	**20.8**	64.2
	4097	8,914	4,078	**1,191**	1,548
	8193	26,035	16,096	4,691	**4,575**
	16385	70,170	64,135	18,734	**13,650**

3.7 Performance of the Root Finding Algorithm in $\mathbb{F}_{2^n}[X]$

Table 14 compares the best implementation of root finding of each library, and for the parameters of GeMSS and `Gui`. The results are similar to the performance of Frobenius map, which is the critical part of the root finding algorithm. `MQsoft` is five to nine times faster than `NTL`.

Table 14: Number of mega cycles to find the roots of a `HFE` polynomial. We use a Skylake processor (LaptopS).

n	D	d	`Magma`	`NTL`	`MQsoft`
174	513	513	1,130	1,089	**116.0**
	514	65	751.5	1,096	**67.1**
185	33	33	39.0	14.0	**1.7**
265	513	513	2,472	2,252	**361.0**
	514	65	1,559	2,304	**203.3**
313	129	129	451.3	352.3	**62.3**
	130	65	356.7	388.5	**53.1**
354	513	513	3,765	4,369	**589.1**
	514	65	2,341	4,392	**329.2**
448	513	513	4,673	5,313	**1,051.0**
	514	65	2,821	5,464	**581.5**

We have presented in this section the main algorithms that we have implemented in `MQsoft` to obtain an efficient root finding for `HFE` polynomials. However, our library proposes extra functions. The half-gcd requires to implement Karatsuba multiplication and fast Euclidean division, in both $\mathbb{F}_{2^n}[X]$. So, we propose a fast version of the root finding, based on a Frobenius map using the fast Euclidean division. This permits to have an efficient implementation of root finding for general applications, using a constant-time arithmetic in the base field.

4 Generation and Evaluation of the Public-key

In this section, we study how implement efficiently important steps of the keypair generation and verifying process. Both are based on multivariate quadratic systems that we represent as quadratic forms.

4.1 Generation of the Inner Secret-key Polynomial f

For the set of completeness, we explain here the method used to compute $\mathbf{f}$ during the keypair generation (Section 1.2). It requires to compute $F\left(\sum_{k=1}^{n}\theta_k x_k, v_1, \ldots, v_v\right)$ (cf. Equation (3)). As a first step, we assume that F does not have vinegar variables. We use two classical properties:

- The terms of degree strictly greater than 1 of F can be represented as a quadratic form $\mathbf{X}Q\mathbf{X}^t$, where $\mathbf{X} = (X, X^2, X^{2^2}, X^{2^3}, \ldots, X^{2^{\lfloor \log_2(D)\rfloor}})$ and $Q \in \mathcal{M}_{\lfloor \log_2(D)\rfloor+1}(\mathbb{F}_{2^n})$ is upper triangular such that:

$$Q_{i,j} = \begin{cases} B_{i+1} & \text{if } i = j \\ A_{j,i} & \text{if } i < j \\ 0 & \text{else.} \end{cases}$$

  We have the relation $F(X) = C + B_0 X + \mathbf{X}Q\mathbf{X}^t$. In particular, $Q_{i,j}$ corresponds to the term $X^{2^i+2^j}$ of F.

- The linearity of the Frobenius implies $X^{2^i} = (\sum_{k=1}^{n}\theta_k x_k)^{2^i} = \sum_{k=1}^{n}\theta_k^{2^i} x_k$.

With these two properties, it is easy to verify that $\mathbf{X} = \mathbf{x}\Gamma$ where $\mathbf{x} = (x_1, \ldots, x_n)$ and $\Gamma \in \mathcal{M}_{n,\lfloor \log_2(D)\rfloor+1}(\mathbb{F}_{2^n})$ is such that $\Gamma_{i,j} = \theta_{i+1}^{2^j}$. So, we deduce the multivariate quadratic form of F:

$$\mathbf{X}Q\mathbf{X}^t = \mathbf{x}Q'\mathbf{x}^t, \text{with } Q' = \Gamma Q \Gamma^t.$$

We have $F = \mathbf{x}Q'\mathbf{x}^t + B_0\theta\mathbf{x}^t + C$. In particular, $Q'_{i,j}$ corresponds to the term $x_i x_j$ of F. Since $x_i^2 = x_i$, we can add each $B_0\theta_i$ from the linear part to the term $Q'_{i,i}$ of Q'. We obtain:

$$F = \mathbf{x}Q''\mathbf{x}^t + C, \text{with } Q'' \text{ such that } \mathbf{x}Q''\mathbf{x}^t = (\mathbf{x}Q' + B_0\theta)\mathbf{x}^t.$$

To simplify the computation of $\mathbf{f}$, θ (Section 1.2) is the same basis that the one used to represent an element of $\mathbb{F}_{2^n}$ (Section 1.6). In this way, $\mathbf{f} = \varphi(\mathbf{x}Q''\mathbf{x}^t + C)$ and we store $\varphi^{-1}(\mathbf{f})$. This is a monomial representation of $\mathbf{f}$.

To compute $\mathbf{f}$, we compute first Γ with $O(n\log_2(D))$ field squarings. This matrix does not depend on F and so it can be precomputed. Then, we compute $\Gamma \times Q$ with classical matrix product, requiring $O(n\log_2(D)^2)$ multiplications in $\mathbb{F}_2[X]$ and $O(n\log_2(D))$ modular reductions. Finally, we multiply the previous result by Γ^t, requiring $O(n^2\log_2(D))$ multiplications in $\mathbb{F}_2[X]$ and $O(n^2)$ modular reductions.

Now, we consider vinegar variables. $\gamma(v_1, \ldots, v_v)$ (Equation (1)) is quadratic in vinegar variables. We store it as a upper triangular matrix $W \in \mathcal{M}_v(\mathbb{F}_{2^n})$ such that $\gamma(v_1, \ldots, v_v) = \mathbf{v}W\mathbf{v}^t + C$ where $\mathbf{v} = (v_1, \ldots, v_v)$ and $C \in \mathbb{F}_{2^n}$ is the constant term of F.
For the linear terms, we must compute $\sum_{i=0}^{\lfloor \log_2(D)\rfloor}\beta_i X^{2^i} = \mathbf{X}\beta^t$ with $\beta = (\beta_0, \ldots, \beta_{\lfloor \log_2(D)\rfloor})$. Let $V \in \mathcal{M}_{\lfloor \log_2(D)\rfloor+1,v}(\mathbb{F}_{2^n})$ and $\beta_i = \sum_{k=1}^{v} V_{i,k}v_k$. We have $\beta = \mathbf{v}V^t$, and so $\mathbf{X}\beta^t = \mathbf{x}\Gamma V\mathbf{v}^t$. ΓV is the matrix where the coefficient (i,j) corresponds to the term $x_i v_j$.

The final result is:

$$\varphi^{-1}(\mathbf{f}) = (\mathbf{x}\ \mathbf{v}) \begin{pmatrix} Q'' & \Gamma V \\ 0 & W \end{pmatrix} (\mathbf{x}\ \mathbf{v})^t + C.$$

The computation of ΓV requires $O(nv\log_2(D))$ multiplications in $\mathbb{F}_2[X]$ and $O(nv)$ modular reductions. So, the generation of the inner secret polynomials of $\mathbf{f}$ requires $O(n\log_2(D))$ squarings in $\mathbb{F}_{2^n}$, $O(n\log_2(D)(n+v+\log_2(D))$ multiplications in $\mathbb{F}_2[X]$ and $O(n(n+v+\log_2(D)))$ modular reductions.

We can notice that this representation of $\mathbf{f}$ permits easily to apply the linear change of variable by the matrix $\mathbf{S}$. Just replace $(\mathbf{x}\ \mathbf{v})$ by $(\mathbf{x}\ \mathbf{v})\mathbf{S}$ and we obtain:

$$(\mathbf{x}\ \mathbf{v})\mathbf{S} \begin{pmatrix} Q'' & \Gamma V \\ 0 & W \end{pmatrix} \mathbf{S}^t (\mathbf{x}\ \mathbf{v})^t + C.$$

4.2 Evaluation of Multivariate Quadratic Systems in $\mathbb{F}_2$

The evaluation of the public-key is the main part of the verifying process (Section 1.4). It is iterated nb_ite times. Since the verification is a public process, it does not require to be protected against timing attacks. So, we can exploit the fact that for a random input, the evaluation of a monomial $x_i x_j$ in $\mathbb{F}_2$ has a probability of 75% to be null, and so to avoid 75% of computations. However, the evaluation in constant-time is required during the signature generation to evaluate the constant of the `HFEv` polynomial, which is quadratic in the vinegar variables. It is also used in other contexts, for example to encrypt a message for the `HFE`-based encryption scheme. In this section, we both study, variable-time and constant-time evaluation.

To evaluate the public-key $\mathbf{p}$, we can use different representations. The representation by equation consists to store the m equations of $\mathbf{p}$ separately ($\mathbf{p} \in (\mathbb{F}_2[x_1,\dots,x_{n+v}])^m$), whereas the representation by monomial consists to store the monomials of $\mathbf{p}$ separately ($\mathbf{p} \in \mathbb{F}_{2^m}[x_1,\dots,x_{n+v}]$, cf. Section 4.1).
The authors of [19] proposes a fast evaluation of the public-key with the representation by equation. In [20], the authors present a faster evaluation. To do so, they use a monomial representation of the public-key. Both, [19] and [20], have used the `avx2` instructions set. We have chosen the monomial representation as in [20], because it exploits naturally the fact that on average, 75% of monomials are null.

Our variable-time evaluation uses only the classical method [7]: we initialize an accumulator `acc` to the constant term of $\mathbf{p}$, and for each term $p_{i,j}x_i x_j$ with $p_{i,j} \in \mathbb{F}_{2^m}$, we add $p_{i,j}$ to `acc` only if $x_i = x_j = 1$. This process is described in Algorithm 9.

We have vectorized Algorithm 9. To do it, we just store `acc` with 256-bit registers, and we use 256-bit load, store and bitwise `XOR` to do vectorial computations. When $\lceil \frac{m}{64} \rceil$ is not a multiple of 4, we sometimes add the use of 64-bit or 128-bit registers, to speed-up the implementation. Algorithm 9 is vulnerable to timing attacks, since the addition is done only if $x_i = x_j = 1$. The traditional way to avoid this attack is to replace the conditional statement by a multiplication by x_i (respectively x_j). But x_i and x_j are in $\mathbb{F}_2$, so the multiplication can be accelerated: it is equivalent to apply a mask which is the duplication of x_i (respectively x_j) m times. With this strategy, we obtain Algorithm 10.

To vectorize Algorithm 10, `acc` and `acc_i` are stored in 256-bit registers. However, the optimal choice to put each mask in a 256-bit register is not trivial. On the one hand, we can store 256-bit masks in the buffer `mask`. In this way, one load permits to create the

Algorithm 9 Evaluation of the public-key in variable-time.

```
function General_evaluation($\mathbf{p} \in \mathbb{F}_{2^m}[x_1, \ldots, x_{n+v}], \mathbf{x} = (x_1, \ldots, x_{n+v}) \in \mathbb{F}_2^{n+v}$)
    acc ← $\mathbf{p}.cst$                          ▷ $\mathbf{p}.cst$ is the constant term of $\mathbf{p}$.
    for $i$ from 1 to $n+v$ do
        if $x_i == 1$ then
            acc+=$p_{i,i}$                          ▷ Addition in $\mathbb{F}_{2^m}$ (bitwise XOR).
            for $j$ from $i+1$ to $n+v$ do
                if $x_j == 1$ then
                    acc+=$p_{i,j}$
                end if
            end for
        end if
    end for
    return acc
end function
```

Algorithm 10 Evaluation of the public-key in constant-time.

```
function Constant_evaluation($\mathbf{p} \in \mathbb{F}_{2^m}[x_1, \ldots, x_{n+v}], \mathbf{x} = (x_1, \ldots, x_{n+v}) \in \mathbb{F}_2^{n+v}$)
    for $i$ from 1 to $n+v$ do
        mask[i] ← -$x_i$                          ▷ Duplicate the bit $x_i$ to create a mask.
    end for
    acc ← $\mathbf{p}.cst$                          ▷ $\mathbf{p}.cst$ is the constant term of $\mathbf{p}$.
    for $i$ from 1 to $n+v$ do
        acc_i ← $p_{i,i}$
        for $j$ from $i+1$ to $n+v$ do
            acc_i+=AND(mask[j],$p_{i,j}$)         ▷ Apply the mask on $p_{i,j}$ (compute $p_{i,j}x_j$).
        end for
        acc+=AND(mask[i],acc_i)  ▷ Apply the mask on acc_i (compute acc_i $\times x_i$).
    end for
    return acc
end function
```

256-bit register. On the other hand, we can store 64-bit masks in the buffer `mask`. The creation of the 256-bit register is done by one call to `VPBROADCASTQ`, which duplicates a 64-bit mask in a 256-bit register. This idea is described in Algorithm 11. We propose a new idea, described in Algorithm 12. Firstly, we unroll with a depth four the loop in j. Then, we store 64-bit masks in the buffer `mask`, but we load four 64-bit masks in one 256-bit register. Then, to create a 256-bit mask from one of the four 64-bit masks, we use the `VPERMQ` instruction. It permits to create a 256-bit register where each 64-bit part is one of the four 64-bit part of the input. In particular, we use it to duplicate one 64-bit part of the input (which is a mask) in a 256-bit register. This method is the best: it requires only one load for four masks, unlike the two previous methods which require four loads (four 256-bit loads for the first method and four 64-bit loads for the second method).

Then, to apply the mask to $p_{i,j}$, we remark that the `VPMASKMOVQ` instruction permits to load data and to apply the mask in only one instruction. It permits to accelerate the evaluation.

More generally, this new method permits to improve the constant-time vector matrix product in $\mathbb{F}_2$, but is interesting only when the variable `buffer` is computed one time for several products using the same vector. We remark that this method is faster on Skylake processors, but on Haswell processors, the use of `VPBROADCASTQ` remains faster.

Algorithm 11 Improvement of Algorithm 10 with avx2, VPMASKMOVQ and VPBROADCASTQ.

```
for $j$ from $i+1$ to $n+v$ do
    acc_i+=VPMASKMOVQ($p_{i,j}$,VPBROADCASTQ(mask[j]))        ▷ Compute $p_{i,j}x_j$.
end for
```

Algorithm 12 Improvement of Algorithm 10 with avx2, VPMASKMOVQ and VPERMQ.

```
for $j$ from $i+1$ to $n+v-3$ by 4 do
    x_256 $\leftarrow$ VMOVDQU(mask+j)    ▷ Load mask[j],mask[j+1],mask[j+2],mask[j+3].
    mask_256 $\leftarrow$ VPERMQ(x_256,0x00)                    ▷ Duplicate mask[j].
    acc_i+=VPMASKMOVQ($p_{i,j}$,mask_256)           ▷ Load $p_{i,j}$ and apply the mask.
    mask_256 $\leftarrow$ VPERMQ(x_256,0x55)                  ▷ Duplicate mask[j+1].
    acc_i+=VPMASKMOVQ($p_{i,j+1}$,mask_256)               ▷ Compute $p_{i,j+1}x_{j+1}$.
    mask_256 $\leftarrow$ VPERMQ(x_256,0xAA)                  ▷ Duplicate mask[j+2].
    acc_i+=VPMASKMOVQ($p_{i,j+2}$,mask_256)               ▷ Compute $p_{i,j+2}x_{j+2}$.
    mask_256 $\leftarrow$ VPERMQ(x_256,0xFF)                  ▷ Duplicate mask[j+3].
    acc_i+=VPMASKMOVQ($p_{i,j+3}$,mask_256)               ▷ Compute $p_{i,j+3}x_{j+3}$.
end for
for $j$ until $n+v$ do
    acc_i+=VPMASKMOVQ($p_{i,j}$,VPBROADCASTQ(mask[j]))        ▷ Compute $p_{i,j}x_j$.
end for
```

Table 15 shows the performance of the evaluation that uses avx2. To improve the performance, we use the option `-funroll-loops` of `gcc` which unrolls loops to improve the use of the pipeline. The factor of performance between variable-time and constant-time implementation depends on m: the factor is two for small values of m and four for high values. The performance is affected by cache penalty when the public-key is too large. For $m = n + v = 256$, we compare our code with the efficient implementation of [20], by using a similar processor (ServerH). We have similar times for constant-time implementation, and a speed-up of 1.38 for variable-time implementation. This speed-up is mainly dued to unrolled loops. Moreover, we have splitted the loop i (respectively the loop j) in two loops with an Euclidean division by 64: the first is a loop for $iq \in [\![0, \lfloor \frac{i}{64} \rfloor]\!]$, and the second is a loop for $ir \in [\![0, 63]\!]$. In this way, for extracting x_i which is the i-th bit from a vector of word, we take the ir-th bits of the iq-th word. It permits to simplify the extraction of bits from 64-bit variables.

For the constant-time evaluation, we have obtained our best times on Haswell by using Algorithm 11. However, on Skylake, Algorithm 12 is faster. For 256 equations and 256 variables (cf. Table 16), we obtain 61.4 Kc with Algorithm 11 against 55.5 Kc with Algorithm 12. Since Algorithm 11 is state-of-the-art on Haswell, we have obtained a new speeding record on Skylake, by a factor 1.1. For comparison, we obtain 23.2 Kc for the variable-time evaluation.

For m requiring one word (respectively two words), we use the 256-bit registers to do computations in $\mathbb{F}_{2^m}$ by pack of four elements (respectively two elements). This method implies to use masks to compute $q_{i,j}x_j$ for four (respectively two) successive values of j. To optimize the cases m requiring one or two words is important because it permits to use a new strategy of parallelization: with k cores, the public-key can be splitted in k packets of 64 equations (respectively 128 equations), and each core can apply one time the evaluation for its part of the public-key. This method is interesting because the number of miss in the cache is decreased since each core has just a part of the public-key. In a general way, m can be splitted in the way to use evaluation algorithms for smaller number of equations.

Table 15: Number of kilo cycles to evaluate the public-key with `MQsoft`, for $m = n + v$. We use a Haswell processor (ServerH) with the `avx2` instructions set. Turbo Boost is not used.

category	$m = 64$	128	192	256	320	384	448	512
constant-time	2.01	14.1	44.7	89.1	196	318	478	853
variable-time	1.15	6.46	17.1	37.3	74.5	120	191	205

Table 16: Number of kilo cycles to evaluate the public-key with `MQsoft`, for $m = n + v$. We use a Skylake processor (DesktopS) with the `avx2` instructions set. Turbo Boost is used.

category	$m = 64$	128	192	256	320	384	448	512
constant-time	1.49	7.04	30.1	55.5	142	202	341	610
variable-time	0.841	3.87	12.3	23.2	51.0	75.5	133	144

Hybrid representation of the multivariate quadratic systems. When m mod 64 is small, the monomial representation of the public key is not optimal for the variable-time evaluation. The addition in $\mathbb{F}_{2^m}$ can be computed with 64-bit, 128-bit and 256-bit `XOR`, so when m mod 64 is small, many bits are unused during the computations. In memory, we use $\lceil \frac{m}{64} \rceil$ words to store each monomial. The unused bits are set to zero. When m mod 64 is small, to store the m mod 64 last equations separately is more efficient. This permits to obtain an optimal representation for a large part of equations, then to optimize the representation of the last remaining equations. We have applied this idea to G*e*MSS256, for which m is 324 and so m mod 64 is equal to 4. The 320 first equations are stored by using the monomial representation, whereas the four last equations are stored one-by-one. With this method, we save 18% of the pratical size of the public key, and we obtain a slight speed-up of 5% during the verifying process.

5 Performance of G*e*MSS, `Gui` and DualModeMS

In this section, we show the speed-ups obtained for G*e*MSS, `Gui` and DualModeMS thanks to `MQsoft`. The difference between `Gui` and G*e*MSS are explained in Section 1.5. `MQsoft` uses the `SHA-3` function from the `Keccak Code Package` [26] and the `SHA-2` function from `OpenSSL`. Random elements ar generated by the determinist random bytes generator provided by the NIST during the competition (which is based on `AES` from `OpenSSL`). The original implementation of G*e*MSS requires `NTL` library to compute the root finding. In `MQsoft`, `NTL` has been completely removed. DualModeMS is based on G*e*MSS implementation and so is naturally supported by `MQsoft`.

5.1 Performance of NIST Implementations

Table 17 summarizes the performance measurements of G*e*MSS additional (best) implementation and `Gui` `PCLMULQDQ` implementation which have been submitted to NIST PQC standardization process. The measurements of G*e*MSS have been corrected since the submission. The parameter D had been set by error at 512 in the implementation (and during the measurements). Because the Frobenius map has not been implemented in constant-time, $D = 512$ allows to save 27% of computations in the critical part of the signature generation (cf. Table 11). As explained in Section 1.5, we have removed the EUF-CMA security property of `Gui` in the `PCLMULQDQ` implementation, because this property is not available in G*e*MSS original implementation. We have also added the performance of DualModeMS, which will be studied in Section 5.5. Measurements more detailed are available in Appendix E.

Table 17: Number of mega cycles (Mc) for each cryptographic operation with a Haswell processor (DesktopH), followed by the speed-up between Haswell and Skylake processors (DesktopH versus DesktopS). For example, $\boxed{110 \quad S\colon +21\%}$ means a performance of 110 Mc on Haswell, and a performance of $\frac{110}{1.21} = 91$ Mc on Skylake.

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	110 S: +21%	1400 S: +39%	0.14 S: +15%
G*e*MSS192	$(265, 513, 22, 20, 4)$	510 S: +19%	3500 S: +35%	0.39 S: +13%
G*e*MSS256	$(354, 513, 30, 33, 4)$	1500 S: +27%	6600 S: +63%	0.91 S: +3.2%
`Gui-184`	$(184, 33, 16, 16, 4)$	670 S: +9.5%	31 S: +38%	0.16 S: +13%
`Gui-312`	$(312, 129, 24, 20, 2)$	4700 S: +13%	450 S: +40%	0.25 S: +7.2%
`Gui-448`	$(448, 513, 32, 28, 2)$	27000 S: +3.4%	16000 S: +45%	0.65 S: +6.7%
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	410 S: +16%	130 S: +34%	0.083 S: +9.9%
DualModeMS128	$(266, 129, 10, 11, 1)$	1700000 S: +5.3%	8800 S: +34%	8.2 S: +5.7%

The main difference between Haswell and Skylake processors is the performance of the `PCLMULQDQ` instruction (cf. Table 4). This impacts mainly the performance of the signature generation, but also the one of the keypair generation.

5.2 Performance of `MQsoft`

We present in Tables 18 and 19 new performance results for G*e*MSS and `Gui`. More detailed measurements are available in Appendix F. The choice of the parameter n in `Gui` requires, for all levels of security, to choose irreducible pentanomials to define $\mathbb{F}_{2^n}$. This affects the performance in comparison to G*e*MSS which uses only irreducible trinomials. For this reason, we propose to sligthly increase the parameter n in `Gui` to use trinomials. This choice does not change the security. For $n = 184$ and $n = 312$, we propose respectively to take $n = 185$ and $n = 313$. However, for $n = 448$, we keep this value since $n = 449$ increases the cost of the multiplication in $\mathbb{F}_{2^n}$. Indeed, the cost of such multiplication is a function of $\lceil \frac{n}{64} \rceil$ (Section 2.2). In practice, the impact of the modular reduction is important for small values of n (as $n = 185$). The obtained speed-up decreases when n increases, because the multiplication is asymptotically coster than the modular reduction.

Table 18 and 19 summarize also the speed-up between NIST submissions and `MQsoft`. For G*e*MSS, the main part of the implementation which was in variable-time has been modified to be in constant-time. To obtain these results, we have improved the complexity of the generation of $\mathbf{f}$ by using the method explained in Section 4.1, and we have vectorized the multiplication by $\mathbf{T}$ (cf. Section 5.4). We have improved the arithmetic in $\mathbb{F}_{2^n}$ with a better multiplication, a vectorial modular reduction and an inverse computed with ITMIA. In the first implementation, the `GCD` was computed by `NTL`. With our efficient computation of the inverse in $\mathbb{F}_{2^n}^{\times}$, the `GCD` becomes negligible compared to the Frobenius map. The verifying process is accelerated by a factor between 1.7 and 2, thanks to our vectorial implementation of the evaluation of the public-key.
For our implementation of `Gui`, we obtain at least a speed-up of a factor 12 for the keypair generation, probably because the original implementation uses evaluations of F then an interpolation to compute $\mathbf{f}$. The signature generation is faster by a factor between 1.2 and 2.5, thanks to our efficient implementation of the arithmetic in $\mathbb{F}_{2^n}$. We target the Skylake processors, whereas the original implementation targets the Haswell processors. For this

Table 18: Number of mega cycles (Mc) for each cryptographic operation with our library for a Haswell processor (DesktopH), followed by the speed-up between the best implementation provided for the NIST submissions versus our implementation. For example, $\boxed{40 \quad \delta\colon +180\%}$ means a performance of 40 Mc with `MQsoft`, and a performance of $40 \times 2.8 = 110$ Mc for the NIST implementations.

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	40 δ: +180%	860 δ: +57%	0.072 δ: +100%
G*e*MSS192	$(265, 513, 22, 20, 4)$	210 δ: +140%	2800 δ: +25%	0.21 δ: +84%
G*e*MSS256	$(354, 513, 30, 33, 4)$	620 δ: +130%	5300 δ: +23%	0.52 δ: +76%
`Gui-184`	$(184, 33, 16, 16, 4)$	51 δ: +1200%	22 δ: +39%	0.088 δ: +81%
`Gui-185`	$(185, 33, 16, 16, 4)$	52 δ: +1200%	18 δ: +72%	0.09 δ: +77%
`Gui-312`	$(312, 129, 24, 20, 2)$	340 δ: +1300%	400 δ: +13%	0.15 δ: +64%
`Gui-313`	$(313, 129, 24, 20, 2)$	340 δ: +1300%	370 δ: +21%	0.15 δ: +61%
`Gui-448`	$(448, 513, 32, 28, 2)$	1400 δ: +1800%	6400 δ: +150%	0.41 δ: +60%
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	190 δ: +120%	100 δ: +30%	0.04 δ: +110%
DualModeMS128	$(266, 129, 10, 11, 1)$	1700000 δ: +0%	6600 δ: +33%	8.2 δ: +0%

reason, the speed-up should be less advantageous on the Haswell processors. However, the modular reduction of the NIST submission uses `PCLMULQDQ`. This method is probably slower on Haswell than the shift-and-add strategy. Moreover, the original implementation uses multi-squaring tables to compute the Frobenius map (Section 3.5). For the 256-bit level of security, this is inefficient compared to the classical repeated squaring algorithm (cf. Table 12 for $D = 513$). But this error has been corrected in libpqcrypto[4], and so in SUPERCOP[5] (SUPERCOP uses the implementation of libpqcrypto). The verifying process is faster by a factor between 1.6 and 1.9.

5.3 Statistics on the Performance of G*e*MSS and `Gui`

We propose here an other method to evaluate the performance of G*e*MSS and `Gui`. The experimental protocol is the following. Firstly, we measure the performance of a small number of keypair generation (between 10 and 100). The set of the measurements will permit to compute the average, the standard deviation, the median as well as the first and third quartiles. Secondly, for three keypairs, we measure the performance of the signing and verifying processes on a small number of documents (between 256 and 2048). To do this, we set the length of each document to 59 bytes, then we generate randomly a small number of documents, stored in a buffer. Then, we measure the performance of the signing process on these documents, and each signature is stored in a second buffer. Finally, we use this buffer to measure the performance of the verifying process. The code was compiled with `gcc -O4 -mavx2 -mpclmul -mpopcnt -funroll-loops`. Here, we use the version 6.5.0 of `gcc`. In Table 20, we present the results for the 128-bit security level, with three significant digits. The complete table is on our website [2].

In this section, we only measure the `Gui` implementation from libpqcrypto. The main difference with the original implementation is the correction of a mistake in the way to compute the Frobenius map (as explained in the previous section). This change impacts only the performance of the signing process of `Gui-448`.

Then, we have updated `MQsoft` to achieve the EUF-CMA security property in order to be comparable to libpqcrypto. The EUF-CMA version implies a slow down of the signing

[4] `https://libpqcrypto.org`
[5] `https://bench.cr.yp.to/supercop.html`

Table 19: Number of mega cycles (Mc) for each cryptographic operation with our library for a Skylake processor (DesktopS), followed by the speed-up between the best implementation provided for the NIST submissions versus our implementation. For example, $\boxed{36 \quad \delta\text{: } +160\%}$ means a performance of 36 Mc with `MQsoft`, and a performance of $36 \times 2.6 = 94$ Mc for the NIST implementations.

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	36 δ: +160%	600 δ: +64%	0.067 δ: +88%
G*e*MSS192	$(265, 513, 22, 20, 4)$	190 δ: +120%	1900 δ: +37%	0.2 δ: +78%
G*e*MSS256	$(354, 513, 30, 33, 4)$	560 δ: +100%	3100 δ: +30%	0.45 δ: +96%
`Gui-184`	$(184, 33, 16, 16, 4)$	49 δ: +1200%	17 δ: +32%	0.081 δ: +74%
`Gui-185`	$(185, 33, 16, 16, 4)$	49 δ: +1100%	13 δ: +68%	0.081 δ: +75%
`Gui-312`	$(312, 129, 24, 20, 2)$	310 δ: +1200%	290 δ: +13%	0.13 δ: +85%
`Gui-313`	$(313, 129, 24, 20, 2)$	310 δ: +1200%	260 δ: +24%	0.13 δ: +83%
`Gui-448`	$(448, 513, 32, 28, 2)$	1300 δ: +1900%	4400 δ: +150%	0.33 δ: +87%
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	170 δ: +100%	69 δ: +43%	0.03 δ: +150%
DualModeMS128	$(266, 129, 10, 11, 1)$	1600000 δ: +0%	4600 δ: +43%	7.8 δ: +0%

process. We do not measure the performance of the NIST submission of G*e*MSS because it does not achieve this property.
In Table 20, we remark that the timings for the signing process are really unstable. This is explained by the fact that during Algorithm 2, the root finding algorithm is reiterated when any root is found. The verifying process is slightly unstable, since the evaluation of the public-key is implemented in variable-time.

SUPERCOP benchmarks. We have integrated `MQsoft` in SUPERCOP. The results are summarized in Table 21. Here again, we have benchmarked the EUF-CMA versions, and we compare `MQsoft` to the `Gui` implementation of SUPERCOP (which is identical to libpqcrypto). We give only the measurements to sign and verify a document of length 59 bytes. The complete table is on our website [2].

In this section, we give the results for Skylake processors. We have also made similar experiments on Haswell and obtained similar ratios. As in the previous section, the signing process is impacted by the performance of `PCLMULQDQ` (Table 4).

5.4 Performance of Keypair Generation

Table 22 summarizes the time of most important steps of the keypair generation (Section 1.2). These steps are achieved in constant-time. The generation of $\mathbf{f}$ is computed as explained in Section 4.1. We have vectorized the multiplication by $\mathbf{T}$, which is based on a vector matrix product in $\mathbb{F}_2$ implemented with `avx2` instructions set. For the moment, the multiplication by $\mathbf{S}$ is not vectorized. This is the crucial part of the keypair generation when `PCLMULQDQ` is available. For the multiplication by $\mathbf{T}$, we have obtained approximately a factor two with the vectorization. So, we can hope to obtain the same factor for the multiplication by $\mathbf{S}$.

5.5 Performance of DualModeMS

DualModeMS [25] scheme is a candidate to NIST PQC standardization process. It is composed by two distinct layers. The first one (Inner.DualModeMS) is a re-parametrization of G*e*MSS. The second one (DualModeMS) is a modified `HFE`-based signature scheme

Table 20: Statistics in mega cycles (Mc) for each cryptographic operation, for a Skylake process (LaptopS). For three different keypairs, documents of length 59 bytes are signed then verified. The EUF-CMA property is implemented for all schemes.

scheme	operation	average	std. deviation	first quartile	median	third quartile
`Gui-184` (libpqcrypto)	keygen	771	9.62	764	769	776
	sign 1	215	201	66.8	154	301
	sign 2	225	217	69.4	162	303
	sign 3	214	211	67.3	147	288
	verify 1	0.298	0.0281	0.285	0.292	0.3
	verify 2	0.295	0.02	0.285	0.292	0.299
	verify 3	0.296	0.0222	0.285	0.292	0.299
`Gui-184` (`MQsoft`)	keygen	59.5	4.1	56.4	58.4	62
	sign 1	153	152	44.2	107	208
	sign 2	155	146	49.7	109	219
	sign 3	154	151	50	108	208
	verify 1	0.207	0.0188	0.193	0.205	0.218
	verify 2	0.206	0.0196	0.192	0.204	0.217
	verify 3	0.207	0.0337	0.192	0.204	0.217
`Gui-185` (`MQsoft`)	keygen	59.4	3.46	56.5	59.1	60.7
	sign 1	120	115	38.8	86	165
	sign 2	125	123	38.7	87.4	165
	sign 3	124	120	40.1	88.9	172
	verify 1	0.209	0.0199	0.195	0.207	0.22
	verify 2	0.21	0.0208	0.195	0.208	0.221
	verify 3	0.21	0.0211	0.195	0.208	0.221
G*e*MSS128 (`MQsoft`)	keygen	45.5	1.02	45.2	45.3	45.3
	sign 1	1670	1340	694	1260	2180
	sign 2	1700	1400	697	1280	2210
	sign 3	1640	1330	689	1260	2070
	verify 1	0.182	0.0168	0.17	0.18	0.191
	verify 2	0.182	0.0164	0.17	0.18	0.19
	verify 3	0.182	0.017	0.17	0.18	0.191

which permits to decrease the size of the public-key, but by increasing the size of the signature. These parameters are chosen to minimize the sum of both sizes. Since the NIST submission DualModeMS is based on the G*e*MSS implementation, `MQsoft` supports naturally DualModeMS. The original implementation supported only the 128-bit level of security. For Inner.DualModeMS, we obtain the original implementation by modifying only the security parameters in the G*e*MSS128 implementation. Our new library permits to support the three levels of security. Table 23 and 24 compare the performance of DualModeMS between the additional implementation of the NIST submission and `MQsoft`. The main part of signature generation is to compute a large number (between 64 and 256) of `HFE`-based signatures with Inner.DualModeMS. For a 128-bit level of security, `MQsoft` permits to obtain a factor 1.33 on Haswell and 1.43 on Skylake (cf. Table 18 and 19). This is directly obtained by the improvement of Inner.DualModeMS. These factors are mainly obtained by using the multi-squaring tables to compute the Frobenius map (Section 3.5). The original implementation used the classical repeated squaring algorithm.

Table 21: Median in mega cycles (Mc) for each cryptographic operation, for a Skylake process (LaptopS). We compare the `Gui` implementation from SUPERCOP (S) with `MQsoft`. For three different keypairs, documents of length 59 bytes are signed then verified. The EUF-CMA property is implemented for all schemes.

scheme	keygen 1	keygen 2	keygen 3	sign 1	sign 2	sign 3	verify 1	verify 2	verify 3
`Gui-184` (S)	485	487	488	208	178	130	0.279	0.273	0.285
`Gui-184`	53.7	54	53.7	73.8	122	129	0.101	0.0998	0.107
`Gui-185`	55.2	54.7	54.7	105	102	132	0.107	0.102	0.107
G*e*MSS128	44.8	44.8	44.7	1380	1390	1270	0.0893	0.0961	0.088

Table 22: Number of mega cycles for main steps of keypair generations with `MQsoft`. We use a Skylake processor (DesktopS).

scheme	sec. level	(n, D, Δ, v)	gen. $\mathbf{f}$	apply $\mathbf{S}$	apply $\mathbf{T}$	keypair gen.
G*e*MSS	128	$(174, 513, 12, 12)$	4.04	26.5	5.78	36.2
	192	$(265, 513, 22, 20)$	21.0	144	20.2	189
	256	$(354, 513, 30, 33)$	49.6	446	61.9	564
`Gui`	128	$(184, 33, 16, 16)$	2.73	33.1	7.11	48.6
		$(185, 33, 16, 16)$	2.57	33.6	7.16	48.9
	192	$(312, 129, 24, 20)$	22.7	229	39.6	307
		$(313, 129, 24, 20)$	22.7	231	40.3	310
	256	$(448, 513, 32, 28)$	100	1080	120	1340

6 Conclusion

`MQsoft` is an efficient library to do `HFE`-based multivariate cryptography. We obtain interesting speed-ups for G*e*MSS, `Gui` and for the signature generation of DualModeMS. Our library provides an efficient constant-time arithmetic, which is on average four times faster than `NTL`. We have proposed a new method to improve the root finding for specific `HFE` polynomials, but the security analysis must be studied in depth. We have exploited the architecture to obtain efficient implementations for the evaluation of multivariate quadratic systems in $\mathbb{F}_2$.

However, our library can be improved again. In fact, the generation of keypair is not completely vectorized. The library is in constant-time for a large part, but is not completely protected against timing attacks. The generation and inversion of random invertible matrices require to use constant-time Gaussian elimination [8]. It is not implemented in `MQsoft` for the moment. Moreover, the `GCD` is implemented in variable-time since the number of iterations is variable. To solve this problem, the `Gui` submission proposes a constant-time implementation which returns a correct `GCD` only if its degree is one.

During the review process, G*e*MSS was selected in the second round of the NIST PQC standardization process. `MQsoft` will permit to improve the performance of G*e*MSS, as well as to propose other trade-offs between security and performance.

Acknowledgements

We thank the reviewers for their very useful comments, and Peter Schwabe for the time spent to confirm the performance of `MQsoft`. This work was financially supported by the French Ministère des armées - Direction Générale de l'Armement, and prepared with the

Table 23: Number of mega cycles for each cryptographic operation for DualModeMS. We use a Haswell processor (DesktopH).

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	sign. gen.	sign. verif.
Inner.DualModeMS128 NIST	$(266, 129, 10, 11, 1)$	410	130	0.083
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	190	100	0.04
Inner.DualModeMS192	$(402, 129, 18, 18, 1)$	930	220	0.13
Inner.DualModeMS256	$(544, 129, 32, 32, 1)$	3600	440	0.24
DualModeMS128 NIST	$(266, 129, 10, 11, 1)$	1.7 M	8800	8.2
DualModeMS128	$(266, 129, 10, 11, 1)$	1.7 M	6600	8.2
DualModeMS192	$(402, 129, 18, 18, 1)$	6.4 M	22000	15
DualModeMS256	$(544, 129, 32, 32, 1)$	16 M	120000	26

Table 24: Number of mega cycles for each cryptographic operation for DualModeMS. We use a Skylake processor (DesktopS).

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	sign. gen.	sign. verif.
Inner.DualModeMS128 NIST	$(266, 129, 10, 11, 1)$	350	98	0.076
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	170	69	0.03
Inner.DualModeMS192	$(402, 129, 18, 18, 1)$	850	150	0.098
Inner.DualModeMS256	$(544, 129, 32, 32, 1)$	3400	290	0.2
DualModeMS128 NIST	$(266, 129, 10, 11, 1)$	1.6 M	6500	7.8
DualModeMS128	$(266, 129, 10, 11, 1)$	1.6 M	4600	7.8
DualModeMS192	$(402, 129, 18, 18, 1)$	5.7 M	16000	15
DualModeMS256	$(544, 129, 32, 32, 1)$	14 M	81000	25

support of the French Programme d'Investissement d'Avenir under national project RISQ[6] P141580.

References

[1] Intel architecture instruction set extensions programming reference, May 2018. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[2] MQsoft: a fast multivariate cryptography library, December 2018. https://www-polsys.lip6.fr/Links/NIST/MQsoft.html.

[3] Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient software implementation of binary field arithmetic using vector instruction sets. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *Progress in Cryptology - LATINCRYPT 2010, First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, Proceedings*, volume 6212 of *Lecture Notes in Computer Science*, pages 144–161. Springer, 2010.

[4] Diego F. Aranha, Julio López, and Darrel Hankerson. High-speed parallel software implementation of the ηt pairing. In Josef Pieprzyk, editor, *Topics in Cryptology - CT-RSA 2010, The Cryptographers' Track at the RSA Conference 2010, San Francisco,*

[6] https://risq.fr/?page_id=31&lang=en

CA, USA, March 1-5, 2010. Proceedings, volume 5985 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.

[5] Magali Bardet, Jean-Charles Faugère, and Bruno Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *International Conference on Polynomial System Solving – ICPSS*, pages 71–75, 2004.

[6] Magali Bardet, Jean-Charles Faugère, Bruno Salvy, and Bo-Yin Yang. Asymptotic behaviour of the degree of regularity of semi-regular polynomial systems. In *The Effective Methods in Algebraic Geometry Conference – MEGA 2005*, pages 1–14, 2005.

[7] Côme Berbain, Olivier Billet, and Henri Gilbert. Efficient implementations of multivariate quadratic systems. In *Selected Areas in Cryptography, 13th International Workshop, SAC 2006, Montreal, Canada, August 17-18, 2006 Revised Selected Papers*, pages 174–187, 2006.

[8] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer, 2013.

[9] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Cryptanalysis of hfe, multi-hfe and variants for odd and even characteristic. *Des. Codes Cryptography*, 69(1):1–52, 2013.

[10] Jingguo Bi, Qi Cheng, and J. Maurice Rojas. Sub-linear root detection, and new hardness results, for sparse polynomials over finite fields. In Manuel Kauers, editor, *International Symposium on Symbolic and Algebraic Computation, ISSAC'13, Boston, MA, USA, June 26-29, 2013*, pages 61–68. ACM, 2013.

[11] Manuel Bluhm and Shay Gueron. Fast software implementation of binary elliptic curve cryptography. *IACR Cryptology ePrint Archive*, 2013:741, 2013.

[12] Wieb Bosma, John J. Cannon, and Graham Matthews. Programming with algebraic structures: Design of the MAGMA language. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '94, Oxford, UK, July 20-22, 1994*, pages 52–57, 1994.

[13] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, and Éric Schost. Algorithmes efficaces en calcul formel, August 2017. 686 pages. Édition 1.0.

[14] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in gf(2)[x]. In *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*, pages 153–166, 2008.

[15] A. Casanova, J.-C. Faugère, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem. G*e*MSS: A great multivariate short signature. Submission to NIST Post-Quantum Cryptography Standardization Process, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[16] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate pkcs on modern x86 cpus. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 33–48, 2009.

[17] M. Chen, J. Ding, A. Petzoldt, D. Schmidt, and B. Yang. Gui. Submission to NIST Post-Quantum Cryptography Standardization Process, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[18] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. *CoRR*, abs/1708.09746, 2017.

[19] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass $\mathcal{MQ}$ -based identification to $\mathcal{MQ}$ -based signatures. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, pages 135–165, 2016.

[20] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure MPKC signatures. *IEICE Transactions*, 101-A(3):553–569, 2018.

[21] Nicolas Courtois. Generic attacks and the security of quartz. In *Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 351–364. Springer, 2003.

[22] James H. Davenport, Christophe Petit, and Benjamin Pring. A generalised successive resultants algorithm. In Sylvain Duquesne and Svetla Petkova-Nikova, editors, *Arithmetic of Finite Fields - 6th International Workshop, WAIFI 2016, Ghent, Belgium, July 13-15, 2016, Revised Selected Papers*, volume 10064 of *Lecture Notes in Computer Science*, pages 105–124, 2016.

[23] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *IACR Cryptology ePrint Archive*, 2017:1251, 2017.

[24] Jean-Charles Faugère and Antoine Joux. Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using gröbner bases. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2003.

[25] J.-C. Faugère, L. Perret, and J. Ryckeghem. DualModeMS: A dual mode for multivariate-based signature. Submission to NIST Post-Quantum Cryptography Standardization Process, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[26] M. Peeters G. Bertoni, J. Daemen and G. Van Assche. A software interface for keccak. 2013.

[27] Shuhong Gao and Todd D. Mateer. Additive fast fourier transforms over finite fields. *IEEE Trans. Information Theory*, 56(12):6265–6272, 2010.

[28] Torbjörn Granlund and al. GNU Multiple Precision Arithmetic Library 6.1.2, December 2002. https://gmplib.org/.

[29] Bruno Grenet, Joris van der Hoeven, and Grégoire Lecerf. Randomized root finding over finite fft-fields using tangent graeffe transforms. In Kazuhiro Yokoyama, Steve Linton, and Daniel Robertz, editors, *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, Bath, United Kingdom, July 06 - 09, 2015*, pages 197–204. ACM, 2015.

[30] Johann Großschädl and Guy-Armand Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields gf(2m). In *14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2003), 24-26 June 2003, The Hague, The Netherlands*, page 455. IEEE Computer Society, 2003.

[31] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[32] William B. Hart. Fast library for number theory: An introduction. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings*, pages 88–91, 2010.

[33] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in gf(2^{m}) using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[34] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 1999.

[35] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2 edition, 1996.

[36] Jeremy Maitin-Shepard. Optimal software-implemented itoh-tsujii inversion for gf(2^{m}). *IACR Cryptology ePrint Archive*, 2015:28, 2015.

[37] Alexander Maximov and Helena Sjoberg. On fast multiplication in binary finite fields and optimal primitive polynomials over GF(2). *IACR Cryptology ePrint Archive*, 2017:889, 2017.

[38] Peter L. Montgomery. Five, six, and seven-term karatsuba-like formulae. *IEEE Trans. Computers*, 54(3):362–369, 2005.

[39] National Institute of Standards and Technology. Round 1 submissions - post-quantum cryptography | csrc, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[40] Jacques Patarin. Hidden fields equations (HFE) and isomorphisms of polynomials (IP): two new families of asymmetric algorithms. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 1996.

[41] Christophe Petit. Finding roots in gf(p^{n}) with the successive resultant algorithm. *IACR Cryptology ePrint Archive*, 2014:506, 2014.

[42] Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, Chengdong Tao, and Jintai Ding. Design principles for hfev- based multivariate signature schemes. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2015.

[43] Victor Shoup. Ntl: A library for doing number theory. 01 2003. http://www.shoup.net/ntl/.

[44] Richard G. Swan. Factorization of polynomials over finite fields. *Pacific J. Math.*, 12(3):1099–1106, 1962.

[45] Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson, and Julio López. Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication. *IACR Cryptology ePrint Archive*, 2011:170, 2011.

[46] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (3. ed).* Cambridge University Press, 2013.

Appendix

A Modular Reduction by a Pentanomial

For completeness, we explain here the principle of modular reduction by a pentanomial. Let $f_5(x) = x^n + x^{k_3} + x^{k_2} + x^{k_1} + 1$ such that $0 < k_1 < k_2 < k_3 \leq \lceil \frac{n}{2} \rceil$. Let $R_0 = \sum_{i=0}^{n-1} r_i x^i$, $R_{k_j} = \sum_{i=n}^{2n-k_j-1} r_i x^{i-n}$ and $S_{k_j} = \sum_{i=2n-k_j}^{2n-2} r_i x^{i-2n+k_j}$ for $j \in [\![1, 3]\!]$, we have:

$$R = R_0 + (R_{k_j} + S_{k_j} x^{n-k_j}) x^n. \tag{9}$$

We do a first step of reduction by f_5 by replacing x^n by $f_5(x) - x^n$ in Equation (9). To compute $(R_{k_j} + S_{k_j} x^{n-k_j}) \times (1 + x^{k_1} + x^{k_2} + x^{k_3})$, we multiply the left operand by x^{k_j} for $j \in [\![1, 3]\!]$, and we choose $j = 1$ when we multiply the left operand by 1. We obtain:

$$R = R_0 + (R_{k_1} + S_{k_1} x^{n-k_1}) + (R_{k_1} x^{k_1} + S_{k_1} x^n) + (R_{k_2} x^{k_2} + S_{k_2} x^n) + (R_{k_3} x^{k_3} + S_{k_3} x^n) \bmod f_5.$$

We iterate a new step of reduction:

$$R = R_0 + R_{k_1} + S_{k_1} x^{n-k_1} + R_{k_1} x^{k_1} + R_{k_2} x^{k_2} + R_{k_3} x^{k_3} + (S_{k_1} + S_{k_2} + S_{k_3})(f_5(x) - x^n) \bmod f_5. \tag{10}$$

In Equation (10), the degree of R is $\max(n-1, 2(k_3-1))$. So, R is reduced modulo f_5 only if $2(k_3 - 1) < n$. In two steps of reduction, we have then a method to compute the modular reduction for all pentanomials such that $2(k_3 - 1) < n$.

To optimize the computation of (10), we compute $R_{k_1} + R_{k_1} x^{k_1} + R_{k_2} x^{k_2} + R_{k_3} x^{k_3}$ as $R_{k_1}(f_5(x) - x^n) \bmod x^n$. In this way, we can rewrite R as:

$$R = R_0 + S_{k_1} x^{n-k_1} + ((R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3})(f_5(x) - x^n) \bmod x^n) \bmod f_5. \tag{11}$$

The product $(R_{k_1} + S_{k_1} + S_{k_2} + S_{k_3})(f_5(x) - x^n)$ can be computed with same methods that in Section 2.3: directly with several calls to `PCLMULQDQ` instructions, or else with the shift-and-add strategy. For the moment, our library uses the shift-and-add strategy which has the advantage to be portable since it does not require `PCLMULQDQ`.

The parameters of `Gui` and `DualModeMS256` require the use of irreducible pentanomials. For $n \in \{184, 312, 448, 544\}$, we have choose respectively $x^{184} + x^{27} + x^{24} + x + 1$, $x^{312} + x^{128} + x^{15} + x^5 + 1$, $x^{448} + x^{64} + x^{39} + x^{33} + 1$ and $x^{544} + x^{128} + x^3 + x + 1$. For these values of n, n is a multiple of 8. We speed-up the extraction of R_{k_1} from R by using shifts of bytes (`PSRLDQ` and `PSHUFB` instructions). For $n = 184$, we choose $k_2 = 24$ because it is a multiple of 8. This permits to improve the multiplication by x^{24} by using shifts of bytes (`PSLLDQ` and `PSHUFB` instructions). For $n = 312$ and $n = 544$, we choose $k_3 = 128$. This improve the multiplication of R_{k_1} by x^{128}, which does not require shifts when the data are stored on 64-bit or 128-bit registers. For $n = 448$, we choose $k_3 = 64$ because n is a multiple of 64. We use this to simplify the multiplication of R_{k_1} by x^{64}: instead of doing this computation, we remark that the result is already available in the input. The result is aligned on 128 bits in R. For $n = 184$ and $n = 544$, the choice of $k_1 = 1$ implies $S_{k_1} = 0$. This saves one instruction in our implementation.

B Addition Chains for the ITMIA

Table 25: Proposed addition chains to minimize the number of multiplications in $\mathbb{F}_{2^n}$. The bold numbers are used to create some numbers of the chain.

n	addition chains	nb of mul.	used in Algorithm 5
174	$\mathbf{1}, 2, 4, \mathbf{5}, 10, 20, 21, 42, 84, 168, 173$	10	no
184	$\mathbf{1}, 2, \mathbf{3}, \mathbf{4}, \mathbf{7}, 11, 22, 44, 88, 176, 183$	10	no
185	$\mathbf{1}, 2, 4, 5, 10, 11, 22, 23, 46, 92, 184$	10	yes
265	$\mathbf{1}, 2, 4, 8, 16, 32, 33, 66, 132, 264$	9	yes
266	$\mathbf{1}, 2, 4, 8, 16, 32, 33, 66, 132, 264, 265$	10	yes
312	$\mathbf{1}, \mathbf{2}, 4, \mathbf{5}, \mathbf{7}, 14, 19, 38, 76, 152, 304, 311$	11	no
313	$\mathbf{1}, 2, 3, \mathbf{6}, 12, 18, 36, 72, 78, 156, 312$	10	no
354	$\mathbf{1}, 2, 4, 5, 10, 11, 22, 44, 88, 176, 352, 353$	11	yes
402	$\mathbf{1}, 2, 3, 6, 12, 24, 25, 50, 100, 200, 400, 401$	11	yes
448	$\mathbf{1}, 2, \mathbf{3}, 6, 12, 24, 27, 54, 108, 111, 222, 444, 447$	12	no
544	$\mathbf{1}, 2, \mathbf{3}, 6, 12, \mathbf{15}, 30, 33, 66, 132, 264, 528, 543$	12	no

C Polynomial Multiplication in $\mathbb{F}_{2^n}[X]$

Fast algorithms which are presented in Appendix D and Section 3.6 require fast multiplications in $\mathbb{F}_{2^n}[X]$. The main fast multiplication is the Karatsuba method [46, Section 8.1]. The fast convolution algorithm [46, Algorithm 8.16] that uses Fast Fourier Transform (FFT) cannot be used here because primitive 2^k-th roots of unity do not exist in the binary field. However, efficient additive FFT in characteristic two have been proposed [27, 8], but they are probably not invertible or not enough efficient to be used here.

Karatsuba multiplication is well known when the degree of polynomials is a power of two [46, Algorithm 8.1], and can be easily extended for all degrees. However, fast algorithms require multiplications of two polynomials having different degrees. To multiply $A, B \in \mathbb{F}_{2^n}[X]$ respectively of degree da, db such that $da \geq db$, we just split A by block of size $db + 1$ in order to apply the multiplication of each block by B. The classical and Karatsuba multiplications in $\mathbb{F}_{2^n}[X]$ are available in `MQsoft`.

D Fast Euclidean Division in $\mathbb{F}_{2^n}[X]$

In this section, we define $Rec_i(P)$ the reciprocal polynomial of $P \in \mathbb{F}_{2^n}[X]$, such that $Rec_i(P) = X^i P(\frac{1}{X})$. The fast Euclidean division [46, Algorithm 9.5] of A by F consists to write $A = FQ + R$ with $Q, R \in \mathbb{F}_{2^n}[X]$, and to remark that:

$$Rec_{2D-2}(A) = Rec_D(F)Rec_{D-2}(Q) + X^{D-1}Rec_{D-1}(R). \tag{12}$$

Because the degree of Q is at most $D - 2$, we can compute Equation (12) modulo X^{D-1}. So, we obtain the following formula for Q:

$$Rec_{D-2}(Q) = Rec_{2D-2}(A)Rec_D(F)^{-1} \bmod X^{D-1}. \tag{13}$$

Once Q is known, R is easily obtained. This process can be summarized in three steps:

- Computation of $Rec_D(F)^{-1} \bmod X^{D-1}$ with Newton iterations [46, Algorithm 9.3].
- Computation of the quotient with Equation (13).

- Computation of the remainder: $R = A - QF \bmod X^D$.

The fast Euclidean division can be used to compute the Frobenius map (Section 3.5). In the case of `HFE-`, $Rec_D(F)^{-1} \bmod X^{D-1}$ can be precomputed and stored with the secret-key, because the computation does not have dependences with the constant of F (in Section 1.3, we are looking for a root of $F(X, \mathbf{v}) - D'$). For the vinegar variant, the linear terms of F depend on the choice of vinegars variables and so $Rec_D(F)^{-1} \bmod X^{D-1}$ must be computed for each of these choices. So, a precomputed table requires $2^v(D-1)n$ bits, which huge.
However, when D is odd, $D-1$ is a power of two and so the computation of $Rec_D(F)^{-1}$ mod X^{D-1} can be improved. Because F has a `HFE` structure, we have that:

$$Rec_D(F) = 1 + \mathrm{coef}_D(F) \bmod X^{\frac{D-1}{4}},$$

and so:

$$Rec_D(F)^{-1} = \sum_{i=0}^{\frac{D-1}{4}-1} \mathrm{coef}_D(F)^i X^i \bmod X^{\frac{D-1}{4}}.$$

With this formula, only two iterations of Newton are required to compute $Rec_D(F)^{-1}$ mod X^{D-1} (by computing $Rec_D(F)^{-1}$ modulo $X^{\frac{D-1}{2}}$ then modulo X^{D-1}). The two others steps require only two multiplications in $\mathbb{F}_{2^n}[X]$ modulo a power of X. With a enough fast multiplication, fast algorithm is better asymptotically than classical Euclidean division (Section 3.3). However, a fast Euclidean division based on Karatsuba multiplication (Appendix C) is inefficient compared to the classical Euclidean division which exploits the sparse structure of F. This explains why `NTL` is slow to compute the Frobenius map in Section 3.5.

In the general case, the fast Euclidean division algorithm is interesting. It permits to improve the fast `GCD` algorithm in Section 3.6. The fast Euclidean division is implemented in `MQsoft`, as well as the fast Frobenius map, the fast `GCD`, and so the fast root finding algorithm. This is slower for `HFE` polynomials, but is efficient in the general case.

E Performance of the NIST submissions

Table 26: Number of mega cycles for each cryptographic operation (best implementation provided for the NIST submissions). We use a Haswell processor (ServerH). Turbo Boost is not used.

scheme	sec. level	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	128	$(174, 513, 12, 12, 4)$	125	1510	0.161
G*e*MSS192	192	$(265, 513, 22, 20, 4)$	562	3870	0.439
G*e*MSS256	256	$(354, 513, 30, 33, 4)$	1620	7300	1.01
`Gui-184`	112	$(184, 33, 16, 16, 2)$	744	17.1	0.0869
	126	$(184, 33, 16, 16, 3)$	744	26.1	0.131
	128	$(184, 33, 16, 16, 4)$	744	34.1	0.177
`Gui-312`	192	$(312, 129, 24, 20, 2)$	5180	505	0.278
`Gui-448`	256	$(448, 513, 32, 28, 2)$	30500	17600	0.715
Inner.DualModeMS128	128	$(266, 129, 10, 11, 1)$	451	146	0.0925
DualModeMS128	128	$(266, 129, 10, 11, 1)$	1850000	9740	9.16

Table 27: Number of mega cycles for each cryptographic operation (best implementation provided for the NIST submissions). We use a Haswell processor (DesktopH).

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	112	1360	0.145
G*e*MSS192	$(265, 513, 22, 20, 4)$	505	3480	0.394
G*e*MSS256	$(354, 513, 30, 33, 4)$	1460	6580	0.911
`Gui-184`	$(184, 33, 16, 16, 4)$	670	30.5	0.16
`Gui-312`	$(312, 129, 24, 20, 2)$	4660	452	0.25
`Gui-448`	$(448, 513, 32, 28, 2)$	27500	15800	0.653
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	406	131	0.0831
DualModeMS128	$(266, 129, 10, 11, 1)$	1670000	8780	8.23

Table 28: Number of mega cycles for each cryptographic operation (best implementation provided for the NIST submissions). We use a Skylake processor (DesktopS).

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	92.6	978	0.125
G*e*MSS192	$(265, 513, 22, 20, 4)$	424	2570	0.35
G*e*MSS256	$(354, 513, 30, 33, 4)$	1150	4040	0.883
`Gui-184`	$(184, 33, 16, 16, 4)$	612	22.1	0.141
`Gui-312`	$(312, 129, 24, 20, 2)$	4110	323	0.233
`Gui-448`	$(448, 513, 32, 28, 2)$	26600	10900	0.612
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	349	97.7	0.0756
DualModeMS128	$(266, 129, 10, 11, 1)$	1580000	6530	7.79

F Performance of `MQsoft`

Table 29: Number of mega cycles (Mc) for each cryptographic operation with our library for a Haswell processor (ServerH), followed by the speed-up between the best implementation provided for the NIST submissions versus our implementation. For example, $\boxed{45 \quad \delta\text{: } +180\%}$ means a performance of 45 Mc with `MQsoft`, and a performance of $45 \times 2.8 = 130$ Mc for the NIST implementations.

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	45 δ: +180%	960 δ: +57%	0.081 δ: +98%
G*e*MSS192	$(265, 513, 22, 20, 4)$	230 δ: +140%	3100 δ: +25%	0.24 δ: +83%
G*e*MSS256	$(354, 513, 30, 33, 4)$	690 δ: +130%	5900 δ: +23%	0.58 δ: +75%
`Gui-184`	$(184, 33, 16, 16, 4)$	57 δ: +1200%	24 δ: +40%	0.11 δ: +67%
`Gui-185`	$(185, 33, 16, 16, 4)$	58 δ: +1200%	20 δ: +73%	0.1 δ: +74%
`Gui-312`	$(312, 129, 24, 20, 2)$	370 δ: +1300%	440 δ: +14%	0.17 δ: +61%
`Gui-313`	$(313, 129, 24, 20, 2)$	380 δ: +1300%	420 δ: +21%	0.17 δ: +59%
`Gui-448`	$(448, 513, 32, 28, 2)$	1600 δ: +1800%	7100 δ: +150%	0.45 δ: +59%
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	210 δ: +120%	110 δ: +30%	0.045 δ: +110%
DualModeMS128	$(266, 129, 10, 11, 1)$	1900000 δ: +0%	7400 δ: +32%	9.2 δ: +0%

Table 30: Number of mega cycles for each cryptographic operation with `MQsoft`. We use a Haswell processor (ServerH). Turbo Boost is not used.

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	44.9	962	0.0814
G*e*MSS192	$(265, 513, 22, 20, 4)$	235	3080	0.24
G*e*MSS256	$(354, 513, 30, 33, 4)$	694	5930	0.577
`Gui-184`	$(184, 33, 16, 16, 4)$	57.2	24.4	0.106
`Gui-185`	$(185, 33, 16, 16, 4)$	58.4	19.8	0.102
`Gui-312`	$(312, 129, 24, 20, 2)$	373	444	0.172
`Gui-313`	$(313, 129, 24, 20, 2)$	380	416	0.174
`Gui-448`	$(448, 513, 32, 28, 2)$	1570	7070	0.451
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	210	112	0.0446
Inner.DualModeMS192	$(402, 129, 18, 18, 1)$	1030	245	0.143
Inner.DualModeMS256	$(544, 129, 32, 32, 1)$	4000	487	0.263
DualModeMS128	$(266, 129, 10, 11, 1)$	1.85 M	7360	9.16
DualModeMS192	$(402, 129, 18, 18, 1)$	7.14 M	24700	17.1
DualModeMS256	$(544, 129, 32, 32, 1)$	18 M	131000	28.5

Table 31: Number of mega cycles for each cryptographic operation with `MQsoft`. We use a Haswell processor (DesktopH).

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	40.5	864	0.0722
G*e*MSS192	$(265, 513, 22, 20, 4)$	211	2770	0.215
G*e*MSS256	$(354, 513, 30, 33, 4)$	625	5330	0.518
`Gui-184`	$(184, 33, 16, 16, 4)$	51.4	21.9	0.088
`Gui-185`	$(185, 33, 16, 16, 4)$	52.4	17.8	0.0899
`Gui-312`	$(312, 129, 24, 20, 2)$	336	399	0.152
`Gui-313`	$(313, 129, 24, 20, 2)$	344	374	0.155
`Gui-448`	$(448, 513, 32, 28, 2)$	1430	6350	0.409
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	189	101	0.0396
Inner.DualModeMS192	$(402, 129, 18, 18, 1)$	928	221	0.129
Inner.DualModeMS256	$(544, 129, 32, 32, 1)$	3640	438	0.241
DualModeMS128	$(266, 129, 10, 11, 1)$	1.67 M	6610	8.23
DualModeMS192	$(402, 129, 18, 18, 1)$	6.42 M	22200	15.3
DualModeMS256	$(544, 129, 32, 32, 1)$	16.2 M	119000	25.7

Table 32: Number of mega cycles for each cryptographic operation with `MQsoft`. We use a Skylake processor (DesktopS).

scheme	$(n, D, \Delta, v, \text{nb\_ite})$	key gen.	signature gen.	signature verif.
G*e*MSS128	$(174, 513, 12, 12, 4)$	36.2	597	0.0666
G*e*MSS192	$(265, 513, 22, 20, 4)$	189	1880	0.197
G*e*MSS256	$(354, 513, 30, 33, 4)$	564	3100	0.451
`Gui-184`	$(184, 33, 16, 16, 4)$	48.6	16.7	0.0811
`Gui-185`	$(185, 33, 16, 16, 4)$	48.9	13.1	0.0811
`Gui-312`	$(312, 129, 24, 20, 2)$	307	285	0.126
`Gui-313`	$(313, 129, 24, 20, 2)$	310	260	0.128
`Gui-448`	$(448, 513, 32, 28, 2)$	1340	4390	0.327
Inner.DualModeMS128	$(266, 129, 10, 11, 1)$	171	68.5	0.0297
Inner.DualModeMS192	$(402, 129, 18, 18, 1)$	850	153	0.0982
Inner.DualModeMS256	$(544, 129, 32, 32, 1)$	3400	292	0.204
DualModeMS128	$(266, 129, 10, 11, 1)$	1.58 M	4570	7.79
DualModeMS192	$(402, 129, 18, 18, 1)$	5.72 M	15800	15.2
DualModeMS256	$(544, 129, 32, 32, 1)$	14.4 M	80600	24.9