G*e*MSS: A Gr*e*at Multivariate Short Signature

Principal submitter

This submission is from the following team, listed in alphabetical order:

- A. Casanova, CS
- J.-C. Faugère, CryptoNext, INRIA and Sorbonne University
- G. Macario-Rat, Orange
- J. Patarin, University of Versailles
- L. Perret, CryptoNext, Sorbonne University and INRIA
- J. Ryckeghem, Sorbonne University and INRIA

E-mail address: `ludovic.perret@lip6.fr`

Telephone : +33-1-44-27-88-35

Postal address:
Ludovic Perret
Sorbonne Université
LIP6 - Équipe projet INRIA/SU POLSYS
Boite courrier 169
4 place Jussieu
F-75252 Paris cedex 5, France

Auxiliary submitters: There are no auxiliary submitters. The principal submitter is the team listed above.

Inventors/developers: The inventors/developers of this submission are the same as the principal submitter. Relevant prior work is credited below where appropriate.

Owner: Same as submitter.

Signature: $\times$. See also printed version of “Statement by Each Submitter”.

1 Changes for the second round (April 1$^{\text{st}}$, 2019)

The goal of this part is to summarize the modifications done on G*e*MSS for the second round.

1.1 Large set of parameters

The parameters proposed for G*e*MSS in the first round were very conservative in term of security. In [5], it was suggested to explore different parameters in order to improve efficiency. We address this comment as follows.

- We added a new Section 9 to discuss the parameters.
- In Section 9.6, we present an exhaustive table including possible parameters and the corresponding timings.
- In Section 9.5, we explore the use of sparse polynomials in G*e*MSS to improve the efficiency of the signing process.
- We then suggest 3 sets of parameters for each security level with several trade-offs. This includes the initial parameters of G*e*MSS proposed in the first round, and two new more aggressive parameters (BlueG*e*MSS and RedG*e*MSS).
  - RedG*e*MSS128 is 269 times faster than G*e*MSS128.
  - BlueG*e*MSS128 is 7.08 times faster than G*e*MSS128.
- We design a family of possible values that depends on only one parameter n. We call this family FG*e*MSS(n) (Section 9.4).

1.2 Further details on known attacks

The paper [2] was published at about the same time than the deadline for the first round. In this revision, we further details [2] in Section 8 (Analysis of known attacks). We added two new sub-sections (8.3.4 and 8.4.2) to explain the attacks from [2]. This attack permits to give more insight on how to balance the number of modifiers. These attacks tend to confirm that taking the same number of minus and same number of vinegar variables is a safe choice.

1.3 Implementation and Performance

Since the submission, we also drastically improved the keypair generation, the signing process as well as the root finding. These are key steps for the efficiency of G*e*MSS. We have a paper accepted at CHES on this topic [4, 1]. This paper presents also `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. `MQsoft` permits, in particular, to

- perform an efficient constant-time arithmetic in $\mathbb{F}_{2^n}$.

- to find the roots of a univariate polynomial in $\mathbb{F}_{2^n}[X]$. We have specialized algorithms for the `HFE` polynomials.
- to evaluate efficiently multivariate quadratic systems in $\mathbb{F}_2$ (in constant-time and in variable-time).
- to implement the dual mode of Matsumoto-Imai based multivariate signature schemes (cf. DualModeMS [3]).

Our new submitted implementations are more secure against timing attacks.

- We have removed `NTL`[1] in the optimized and additional implementations. We have added a constant-time implementation of the inverse in $\mathbb{F}_{2^n}$ (which replaces the use of `NTL`).
- During the keypair generation, the determinant and the inversion of matrices in $M_n(\mathbb{F}_2)$ are achieved in constant-time. In particular, we have implemented a constant-time Gaussian elimination. Now, the keypair generation is immune against timing attacks.
- During the signing process, the Frobenius map is achieved in constant-time. We assume the degree of the current polynomial in $\mathbb{F}_{2^n}[X]$ is $D-1$ and its square has a degree $2D-2$. In particular, we compute the square of the zero coefficients.

Our practical sizes are shorter.

- We have added an algorithm to pack and unpack the bits of the signature. Now, the theoretical size is the same than the practice size.
- When the public-key has 324 equations, we have added "an hybrid representation" [4] which permits to have a practice size similar to the theoretical size. The practical size of the 320 first equations is exactly the theoretical size. The 4 last equations are stored one by one, and are unpacked.

Several algorithms are improved:

- The computation of the components of F is faster. When $v = 0$, the old complexity was $O(n^2 \log_2(D)^2)$ multiplications in $\mathbb{F}_{2^n}$. The new complexity is $O(n \log_2(D)(n + \log_2(D)))$ multiplications in $\mathbb{F}_2[X]$ and $O(n(n + \log_2(D)))$ modular reductions.
- When n is large compared to D, we compute the Frobenius map by using multi-squaring tables.

Our implementations use only the `ranbombytes` function for the random generation. In particular, we have modified the equal-degree factorization algorithm from `NTL`, for using `ranbombytes`.

[1] `http://www.shoup.net/ntl/`

2 New changes for the second round (April 15, 2020)

The goal of this part is to summarize the modifications done on GeMSS after the beginning of the second round. We have decreased the size of the keys. For the secret-key, it is now generated from a small secret seed (Section 2.1). For the public-key, we have improved the implementation to reach the theoretical size. We introduce a so-called ”hybrid representation” [4], which allows to keep an efficient evaluation (Section 2.3). The changes in our implementation are enumerated in Section 2.4. In particular, the old KATs files are no longer valid since we have changed the keys format.

2.1 Secret-key generated from a seed and new sizes

We have drastically reduced the size of the secret-key. For that, we expand the secret-key from a random seed. This is classical and implies to consider a new attack: the exhaustive research of the seed. Thus, we set the size of the seed to λ bits to reach a λ-bit security level. This change increases the cost of the signing process, since the secret-key has to be generated for each operation. However, the expansion of the seed is negligible compared to the cost of the root finding. The timings are not really impacted by this modification (just slightly for RedGeMSS which has a fast signing process).

The use of a seed is controlled with the `ENABLED_SEED_SK` macro (set to 1 by default) from `config_HFE.h`. When enabled, the seed is expanded with `SHAKE`.

In Table 1, we provide the updated sizes of the public-key, secret-key and signature. From now on, the implementation optimizes the sizes. The theoretical and practical sizes are the same. Since the secret-key is generated from a seed, the secret-key is very small: just several hundreds of bits. In contrast, the decompressed secret-key size is between 10 and 80 KB. For the public-key, we have decreased the practical size. This part will be further explained in Section 2.3. We save 18% for $\lambda = 128$, 5% for $\lambda = 192$ and 0.2% for $\lambda = 256$ (the latter was already optimized since the beginning of the second round).

scheme	$(\lambda, D, n, \Delta, v, \text{nb\_ite})$	$\lvert pk \rvert$ (KB)	$\lvert sk \rvert$ (B)	sign (B)
GeMSS128	$(128, 513, 174, 12, 12, 4)$	352.188	16	32.25
BlueGeMSS128	$(128, 129, 175, 13, 14, 4)$	363.609	16	33.75
RedGeMSS128	$(128, 17, 177, 15, 15, 4)$	375.21225	16	35.25
GeMSS192	$(192, 513, 265, 22, 20, 4)$	1237.9635	24	51.375
BlueGeMSS192	$(192, 129, 265, 22, 23, 4)$	1264.116375	24	52.875
RedGeMSS192	$(192, 17, 266, 23, 25, 4)$	1290.542625	24	54.375
GeMSS256	$(256, 513, 354, 30, 33, 4)$	3040.6995	32	72
BlueGeMSS256	$(256, 129, 358, 34, 32, 4)$	3087.963	32	73.5
RedGeMSS256	$(256, 17, 358, 34, 35, 4)$	3135.591	32	75

Table 1: Memory cost. 1 KB is 1000 bytes.

2.2 Time

The following measurements were realized under the same conditions as at the beginning of the second round, excepted for the compilation of the reference implementation. The latter was compiled with `gcc -O2 -msse2 -msse3 -mssse3 -msse4.1 -mpclmul`. The SIMD is enabled only to inline the (potential) vector multiplication functions from the `gf2x` library[2]. The reference implementation does not exploit these instructions sets. The used machines (LaptopS, ServerH) have not changed and are described in the specification.

2.2.1 Reference implementation

For the second round, we had removed the use of `NTL` in the optimized and additional implementations (Section 1.3). This allowed to remove the use of `C++` in the implementation. The code is easier to use, more portable and more standalone. In our new implementation, we have also removed `NTL` from the reference implementation. However, the performance of the multiplication in $\mathbb{F}_2[x]$ is crucial for G*e*MSS. The latter was performed by `NTL`. So, we propose to switch to the `gf2x` library, which is specialized in multiplication in $\mathbb{F}_2[x]$.

These choices explain the new performances summarized in Table 2. The verifying process is more than 100 times faster, whereas the keypair generation is 13 times faster. The performance of the signing process depends on D. Indeed, `NTL` uses classical modular reductions when $D = 17$, whereas fast modular reductions are used for $D = 129$ and $D = 513$. The fast modular reduction is slower than the classical method when the input is a sparse `HFE` polynomial. So, we conclude that the vector arithmetic from `NTL` is faster than the vector multiplication from `gf2x` coupled to our reference arithmetic (without vector instructions).

scheme	$(\lambda, D, n, \Delta, v, \text{nb\_ite})$	key gen. (MC)	sign (MC)	verify (KC)
G*e*MSS128	$(128, 513, 174, 12, 12, 4)$	145 / ×13	2730 / ×2.5	211 / ×140
BlueG*e*MSS128	$(128, 129, 175, 13, 14, 4)$	118 / ×13	530 / ×1.46	228 / ×130
RedG*e*MSS128	$(128, 17, 177, 15, 15, 4)$	91.1 / ×13	52 / ×0.34	239 / ×110
G*e*MSS192	$(192, 513, 265, 22, 20, 4)$	619 / ×13	6510 / ×2.3	585 / ×150
BlueG*e*MSS192	$(192, 129, 265, 22, 23, 4)$	520 / ×13	1290 / ×0.99	592 / ×150
RedG*e*MSS192	$(192, 17, 266, 23, 25, 4)$	423 / ×14	126 / ×0.22	627 / ×120
G*e*MSS256	$(256, 513, 354, 30, 33, 4)$	1660 / ×12	10500 / ×2.4	1160 / ×150
BlueG*e*MSS256	$(256, 129, 358, 34, 32, 4)$	1510 / ×13	2080 / ×0.79	1190 / ×150
RedG*e*MSS256	$(256, 17, 358, 34, 35, 4)$	1310 / ×14	203 / ×0.18	1190 / ×120

Table 2: Performance of the reference implementation, followed by the speed-up between the new and the previous implementation. We use a Skylake processor (LaptopS). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits. For example, $\boxed{145\ /\ \times 13}$ means a performance of 145 MC with the new code, and a performance of $145 \times 13 = 1880$ MC with the old code.

[2] http://gf2x.gforge.inria.fr/

2.2.2 Optimized (Haswell) implementation

Since the original submission of the second round, the verifying process is between 3 and 10% slower. This is due to the fact that the public-key is stored with a packed representation. The signing process is up to 43% faster, since we have adapted the multiplication and squaring in $\mathbb{F}_2[x]$ for the Haswell processors. This counterbalances the slight cost of the secret-key decompression. The new arithmetic in $\mathbb{F}_2[x]$ improves slightly the keypair generation.

scheme	$(\lambda, D, n, \Delta, v, \text{nb\_ite})$	key gen. (MC)	sign (MC)	verify (KC)
GeMSS128	$(128, 513, 174, 12, 12, 4)$	51.6 / ×1.01	1240 / ×0.98	163 / ×0.92
BlueGeMSS128	$(128, 129, 175, 13, 14, 4)$	52.1 / ×1.02	198 / ×1.02	170 / ×0.93
RedGeMSS128	$(128, 17, 177, 15, 15, 4)$	52.4 / ×1.06	5.72 / ×0.97	178 / ×0.91
GeMSS192	$(192, 513, 265, 22, 20, 4)$	270 / ×1.01	3320 / ×1.08	459 / ×0.96
BlueGeMSS192	$(192, 129, 265, 22, 23, 4)$	268 / ×1.07	481 / ×1.09	468 / ×0.94
RedGeMSS192	$(192, 17, 266, 23, 25, 4)$	264 / ×1.03	13.7 / ×1.01	474 / ×0.96
GeMSS256	$(256, 513, 354, 30, 33, 4)$	814 / ×1.04	5380 / ×1.32	973 / ×0.97
BlueGeMSS256	$(256, 129, 358, 34, 32, 4)$	810 / ×1.08	733 / ×1.43	989 / ×0.97
RedGeMSS256	$(256, 17, 358, 34, 35, 4)$	805 / ×1.07	22.1 / ×1.17	1010 / ×0.97

Table 3: Performance of the optimized implementation, followed by the speed-up between the new and the previous implementation. We use a Haswell processor (ServerH). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits. For example, $\boxed{163\ /\ \times 0.92}$ means a performance of 163 KC with the new code, and a performance of $163 \times 0.92 = 150$ KC with the old code.

2.2.3 Additional (Skylake) implementation

The additional and the optimized implementations are based on the same implementation. We have only set the macro `PROC_SKYLAKE` to 1, whereas in the optimized implementation, we set the macro `PROC_HASWELL` to 1. This macro impacts mainly the multiplication in $\mathbb{F}_{2^n}$. Since the original submission of the second round, the verifying process is between 11 and 16% slower, for the same reason as before. The signing process is up to 17% slower, since the secret-key must be decompressed. The keypair generation is slightly slower because the secret-key must be decompressed and the public-key must be packed. Finally, the performance is not really impacted by our new updates, whereas keys size is smaller.

scheme	$(\lambda, D, n, \Delta, v, \text{nb\_ite})$	key gen. (MC)	sign (MC)	verify (KC)
G*e*MSS128	$(128, 513, 174, 12, 12, 4)$	52.6 / ×0.97	1040 / ×0.9	164 / ×0.89
BlueG*e*MSS128	$(128, 129, 175, 13, 14, 4)$	53.8 / ×0.97	164 / ×0.97	176 / ×0.88
RedG*e*MSS128	$(128, 17, 177, 15, 15, 4)$	54.3 / ×0.98	5.24 / ×0.88	185 / ×0.86
G*e*MSS192	$(192, 513, 265, 22, 20, 4)$	275 / ×0.96	2960 / ×0.98	501 / ×0.87
BlueG*e*MSS192	$(192, 129, 265, 22, 23, 4)$	278 / ×0.96	448 / ×0.96	512 / ×0.86
RedG*e*MSS192	$(192, 17, 266, 23, 25, 4)$	277 / ×0.96	13.1 / ×0.9	518 / ×0.87
G*e*MSS256	$(256, 513, 354, 30, 33, 4)$	916 / ×0.95	4940 / ×0.98	1120 / ×0.91
BlueG*e*MSS256	$(256, 129, 358, 34, 32, 4)$	923 / ×0.96	653 / ×1.06	1140 / ×0.89
RedG*e*MSS256	$(256, 17, 358, 34, 35, 4)$	921 / ×0.97	21.4 / ×0.86	1170 / ×0.9

Table 4: Performance of the additional implementation, followed by the speed-up between the new and the previous implementation. We use a Skylake processor (LaptopS). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits. For example, $\boxed{164\ /\ \times 0.89}$ means a performance of 164 KC with the new code, and a performance of $164 \times 0.89 = 146$ KC with the old code.

2.2.4 `MQsoft`

We give here the times with the latest version of `MQsoft` [4] that uses `sse2`, `ssse3` and the `avx2` instructions sets to be faster. Since the original submission of the second round, the verifying process is between 17 and 31% slower, for the same reason as before. The signing process is between 20 and 41% faster, thanks to some optimizations. The keypair generation is not impacted.

scheme	$(\lambda, D, n, \Delta, v, \text{nb\_ite})$	key gen. (MC)	sign (MC)	verify (KC)
G*e*MSS128	$(128, 513, 174, 12, 12, 4)$	38.7 / ×0.99	531 / ×1.41	106 / ×0.77
BlueG*e*MSS128	$(128, 129, 175, 13, 14, 4)$	39.2 / ×1.00	81.3 / ×1.3	136 / ×0.82
RedG*e*MSS128	$(128, 17, 177, 15, 15, 4)$	39.5 / ×0.99	2.33 / ×1.2	141 / ×0.77
G*e*MSS192	$(192, 513, 265, 22, 20, 4)$	175 / ×1.00	1800 / ×1.29	304 / ×0.79
BlueG*e*MSS192	$(192, 129, 265, 22, 23, 4)$	174 / ×0.99	252 / ×1.31	325 / ×0.78
RedG*e*MSS192	$(192, 17, 266, 23, 25, 4)$	173 / ×0.99	5.97 / ×1.4	334 / ×0.76
G*e*MSS256	$(256, 513, 354, 30, 33, 4)$	530 / ×1.00	3020 / ×1.21	678 / ×0.83
BlueG*e*MSS256	$(256, 129, 358, 34, 32, 4)$	530 / ×1.00	399 / ×1.37	684 / ×0.85
RedG*e*MSS256	$(256, 17, 358, 34, 35, 4)$	534 / ×0.98	9.82 / ×1.31	704 / ×0.84

Table 5: Performance of `MQsoft`, followed by the speed-up between the new and the previous implementation. We use a Skylake processor (LaptopS). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits. For example, $\boxed{106\ /\ \times 0.77}$ means a performance of 106 KC with the new code, and a performance of $106 \times 0.77 = 82$ KC with the old code.

2.3 Packed representation of the public-key

The proposed implementation for the second round does not reached the theoretical size of the public-key. We solve this problem in our new implementation. We use a public-key format allowing to pack the bits of the public-key, while maintaining a fast use during the verifying process. On one hand, we save up to 18% of the public-key size. On the other hand, the verifying process is slightly slower (up to 31%). This change does not impact the security.

This format is based on the so-called "hybrid representation" [4]. Let $m = 8 \times k + r$ be the Euclidean division of m by 8. We store the $8k$ first equations with the monomial representation, then we store the r last equations one by one. This process is illustrated by Figure 1. Firstly, we pack the coefficients of the $8k$ first equations monomial by monomial. This corresponds to take the vertical rectangles from left to right, then to take coefficients from up to down. Secondly, we pack the coefficients of each of the r last equations. This corresponds to take the horizontal rectangles from up to down, then to take coefficients from left to right.

$$\begin{array}{l}
c^{(1)} + p_{1,1}^{(1)}x_1^2 + p_{1,2}^{(1)}x_1x_2 + p_{1,3}^{(1)}x_1x_3 + p_{2,2}^{(1)}x_2^2 + p_{2,3}^{(1)}x_2x_3 + p_{3,3}^{(1)}x_3^2 \\
c^{(2)} + p_{1,1}^{(2)}x_1^2 + p_{1,2}^{(2)}x_1x_2 + p_{1,3}^{(2)}x_1x_3 + p_{2,2}^{(2)}x_2^2 + p_{2,3}^{(2)}x_2x_3 + p_{3,3}^{(2)}x_3^2 \\
c^{(3)} + p_{1,1}^{(3)}x_1^2 + p_{1,2}^{(3)}x_1x_2 + p_{1,3}^{(3)}x_1x_3 + p_{2,2}^{(3)}x_2^2 + p_{2,3}^{(3)}x_2x_3 + p_{3,3}^{(3)}x_3^2 \\
c^{(4)} + p_{1,1}^{(4)}x_1^2 + p_{1,2}^{(4)}x_1x_2 + p_{1,3}^{(4)}x_1x_3 + p_{2,2}^{(4)}x_2^2 + p_{2,3}^{(4)}x_2x_3 + p_{3,3}^{(4)}x_3^2 \\
c^{(5)} + p_{1,1}^{(5)}x_1^2 + p_{1,2}^{(5)}x_1x_2 + p_{1,3}^{(5)}x_1x_3 + p_{2,2}^{(5)}x_2^2 + p_{2,3}^{(5)}x_2x_3 + p_{3,3}^{(5)}x_3^2 \\
c^{(6)} + p_{1,1}^{(6)}x_1^2 + p_{1,2}^{(6)}x_1x_2 + p_{1,3}^{(6)}x_1x_3 + p_{2,2}^{(6)}x_2^2 + p_{2,3}^{(6)}x_2x_3 + p_{3,3}^{(6)}x_3^2 \\
c^{(7)} + p_{1,1}^{(7)}x_1^2 + p_{1,2}^{(7)}x_1x_2 + p_{1,3}^{(7)}x_1x_3 + p_{2,2}^{(7)}x_2^2 + p_{2,3}^{(7)}x_2x_3 + p_{3,3}^{(7)}x_3^2 \\
c^{(8)} + p_{1,1}^{(8)}x_1^2 + p_{1,2}^{(8)}x_1x_2 + p_{1,3}^{(8)}x_1x_3 + p_{2,2}^{(8)}x_2^2 + p_{2,3}^{(8)}x_2x_3 + p_{3,3}^{(8)}x_3^2 \\
\boxed{c^{(9)} + p_{1,1}^{(9)}x_1^2 + p_{1,2}^{(9)}x_1x_2 + p_{2,2}^{(9)}x_2^2 + p_{1,3}^{(9)}x_1x_3 + p_{2,3}^{(9)}x_2x_3 + p_{3,3}^{(9)}x_3^2} \\
\boxed{c^{(10)} + p_{1,1}^{(10)}x_1^2 + p_{1,2}^{(10)}x_1x_2 + p_{2,2}^{(10)}x_2^2 + p_{1,3}^{(10)}x_1x_3 + p_{2,3}^{(10)}x_2x_3 + p_{3,3}^{(10)}x_3^2}
\end{array}$$

Figure 1: Example of hybrid represention of a multivariate quadratic system with 10 equations and 3 variables. Each row corresponds to one equation, and the $c^{(k)}$ and $p_{i,j}^{(k)}$ are in $\mathbb{F}_2$.

Our aim is to decrease the cost to unpack the bits of the public-key during the verifying process. With our format, a big part of the public-key uses the monomial representation. At the beginning of the second round, this representation was used to store the m equations (instead of $8k$ equations). So, the evaluation of the $8k$ first equations is performed as efficiently as before. They do not require to be unpacked. This implies that only the r last equations generate an additional cost, which is slight ($r \leq 7$ is small compared to $8k$). These equations can be evaluated packed, but when nb_ite > 1, to unpack them permits to accelerate the evaluation (which is repeated nb_ite times).

Implementation details

An important point in our implementation is the memory alignment. All used data has to be aligned on bytes. This permits to have more simple and more efficient implementations. In the previous implementation, we used a zero padding when necessary. However, this implied that the theoretical size was not reached.

Firstly, the $8k$ first equations are stored without loss. Since for each monomial, $8k$ coefficients in $\mathbb{F}_2$ are packed, we obtain that k bytes are required to store them. So, we do not require padding to align data on bytes. The monomials are stored in the graded lexicographic order (as on Figure 1). Secondly, the r last equations are stored in the graded reverse lexicographic order (as on Figure 1). Each equation requires to store $N = \frac{(n+v)(n+v+1)}{2}$ elements of $\mathbb{F}_2$. The alignement of the equations requires to use a zero padding when N is not multiple of 8. In this case, the padding size is $N_p = 8 - (N \bmod 8)$ bits. We solve this problem by using the $(r-1)N_p$ last bits of the last equation to fill the paddings of the $(r-1)$ other equations. In particular, we take these last bits by pack of N_p, and the ℓ-th pack is used to fill the padding of the $(8k+\ell)$-th equation. For example, on Figure 1, the 9-th equation contains 7 coefficients. So, with our process, we would remove $p_{3,3}^{(10)}$ from the 10-th equation to store it just after $p_{3,3}^{(9)}$. Thus, the 9-th equation would be aligned on 8 bits.

2.4 Improved implementation

Here is the list of updates in the implementation.

- The KATs files have been updated. The old KATs files are no longer valid.
- The multiplication and squaring in $\mathbb{F}_2[x]$ are now adapted for the Haswell processors.
- A seed expander has been added to generate the secret-key.
- The output of the linear transformation by $\mathbf{T}$ is modified to obtain a public-key stored on bytes instead of 64-bit words, during the keypair generation.
- The functions to pack, unpack and evaluate the public-key, when the latter is stored on bytes, have been added.
- The equality and comparison tests are performed in constant-time when secret data is used.
- Constant-time sorts have been added.
- The Frobenius trace and map functions are performed in constant-time.
- All Intel intrinsics are used only in `simd_intel.h`.
- The documentation of the functions has been extended and made clearer.
- The names have been made more explicit.
- Useless functions have been removed to decrease the code size.
- Reference implementations of functions have been added.

- The reference implementation has been simplified. Our new reference implementation is based on the optimized implementation, but by using the reference implementations added previously. Unlike the old implementation which used `NTL` with `C++`, this new implementation is in `C` and `NTL` is not required anymore. Since the reference implementation of the multiplication is $\mathbb{F}_2[x]$ is slow, the use of the `gf2x` library is set to 1 (`ENABLED_GF2X` macro from `arch.h`). This accelerates the performances, but can also be disabled.

References

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

[2] Jintai Ding, Ray A. Perlner, Albrecht Petzoldt, and Daniel Smith-Tone. Improved cryptanalysis of hfev- via projection. In *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, pages 375–395, 2018.

[3] Jean-Charles Faugère, Ludovic Perret, and Jocelyn Ryckeghem. DualModeMS: A Dual Mode for Multivariate-based Signature. Research report, UPMC - Paris 6 Sorbonne Universités ; INRIA Paris ; CNRS, December 2017.

[4] Jean-Charles Faugère, Ludovic Perret, and Jocelyn Ryckeghem. Software toolkit for hfe-based multivariate schemes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):257–304, 2019.

[5] NIST. Status report on the first round of the nist post-quantum cryptography standardization process.