

GeMSS: A Great Multivariate Short Signature

Principal submitter

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

- A. Casanova, CS
- J.-C. Faugère, CryptoNext Security, INRIA and Sorbonne University
- G. Macario-Rat, Orange
- J. Patarin, University of Versailles
- L. Perret, CryptoNext Security, 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: ×. See also printed version of “Statement by Each Submitter”.

1 Changes for the Third round (October 1st, 2020)

GeMSS has been selected by NIST for the third round of the Post-Quantum Cryptography standardization process. The report on round-2 candidates [11] acknowledges the confidence on the security of GeMSS and summarised its distinguishing practical features : large public-key, smallest signature size among all candidates, signing process slower than several round-2 candidates and fast verification process. As mentioned [11], these practical features make GeMSS a good candidate for “*applications in which offline signing and no transmission of the public key are acceptable and expected*”. On the other hand [11], the large public-key makes the use of GeMSS for low-end devices difficult. It was also suggested in [11], that there is possibly rooms for improvements in the choice of the parameters.

In view of theses comments, we summarize below the modifications done on GeMSS since April 15, 2020 (Section B).

1. We have improved the quality of the software implementation. Namely:
 - We have been able to further improved the speed the main steps of GeMSS for all the parameters. In particular, this is to the use of new algorithmic ideas. In Section 2.1, we describe a faster public-key generation, via the evaluation-interpolation algorithm. This method is also particularly adapted to low-end devices.
 - We have reinforced the security of the software implementation. Now, we use a constant-time gcd (Section 2.2) during the root finding. Moreover, we propose a new method to perform roots finding in constant-time (Section 2.3). Compared to the state-of-the-art, the new method only induces a small slow-down in comparison to a non constant-time algorithm (approximately 10% on average).
 - In Section 2, we summarize the impact of the various modifications of the efficiency of GeMSS. For instance, that round-3 version of GeMSS128 has a keypair generation more than six times faster than the best round-1 implementation, a signing process more than two times faster than the round-1 version and a verification process more than 1.5 times than the round-1 version.
2. We updated the security analysis by including a new third-party analysis [2] of the security of GeMSS. The authors of [2] proposed an improved method for solving MinRank; a problem underlying the security of GeMSS and others round-2 candidates (Rainbow, ROLLO and RQC). [2] leads to complexity bounds smaller than the claimed security for Rainbow, ROLLO and RQC. This is not the case for GeMSS, whose claimed security level was not impacted by [2] for any parameter (Section 2.6).
3. We propose new parameters sets to improve arithmetic in \mathbb{F}_{2^n} on low-end devices (Section 2.5). The values of n are chosen to accelerate multiplications with parallel multiplications in \mathbb{F}_{2^4} and \mathbb{F}_{2^8} . The latter are not implemented for the moment.
4. Following a remark of NIST, we analysed more closely the security of Feistel-Patarin construction. We conclude that it is possible to improve GeMSS by decrementing the number of iterations in Feistel-Patarin construction (Section 2.7). This change requires to estimate the gap between the cost of the public-key evaluation compared to a SHA-3 evaluation. As a result, we formally propose a new variation of colors (i.e., that is new parameters) : WhiteGeMSS, CyanGeMSS and MagentaGeMSS.

2 Evolution of the Fastest Version of MQsoft during the NIST PQC Standardization Process

We show in Table 1 (respectively Table 2 and 3) the evolution of the performance of GeMSS128 (respectively RedGeMSS128 and BlueGeMSS256). The improvements leading to such speed-ups are detailed in the following sections.

operation	NIST round 1	NIST round 2	NIST round 2 (V2)	NIST round 3
keypair generation	118 MC	$\times 3.07$	$\times 3.05$	$\times 6.03$
signing process	1270 MC	$\times 1.69$	$\times 2.39$	$\times 2.09$
verifying process	0.166 MC	$\times 2.03$	$\times 1.57$	$\times 1.57$

Table 1: Performance of GeMSS128 at the first round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the first round. For each round, we use the corresponding MQsoft version with a Skylake processor (LaptopS). MC stands for Mega Cycles. The results have three significant digits. For example, $\times 3.05$ means a performance of $118/3.05 = 38.7$ MC with the NIST round 3 implementation of MQsoft, for the keypair generation.

operation	NIST round 2	NIST round 2 (V2)	NIST round 3
keypair generation	39.2 MC	$\times 0.992$	$\times 2.41$
signing process	2.79 MC	$\times 1.20$	$\times 1.36$
verifying process	0.109 MC	$\times 0.772$	$\times 0.774$

Table 2: Performance of RedGeMSS128 at the second round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the second round. For each round, we use the corresponding MQsoft version with a Skylake processor (LaptopS). MC stands for Mega Cycles. The results have three significant digits. For example, $\times 2.41$ means a performance of $39.2/2.41 = 16.3$ MC with the NIST round 3 implementation of MQsoft, for the keypair generation.

operation	NIST round 2	NIST round 2 (V2)	NIST round 3
keypair generation	529 MC	$\times 0.998$	$\times 3.47$
signing process	545 MC	$\times 1.37$	$\times 2.20$
verifying process	0.583 MC	$\times 0.852$	$\times 0.857$

Table 3: Performance of BlueGeMSS256 at the second round of the NIST PQC standardization process, followed by the speed-up between the other rounds and the second round. For each round, we use the corresponding MQsoft version with a Skylake processor (LaptopS). MC stands for Mega Cycles. The results have three significant digits. For example, $\times 3.47$ means a performance of $529/3.47 = 152$ MC with the NIST round 3 implementation of MQsoft, for the keypair generation.

2.1 Keypair Generation by Evaluation-Interpolation

As mentioned in the seminal paper of Matsumoto-Imai [9], the public-key polynomials $\mathbf{p} = (p_1, \dots, p_m) \in \mathbb{F}_2[x_1, \dots, x_{n+v}]^m$ can be generated by evaluation-interpolation : \mathbf{p} is evaluated in N distinct evaluation points in \mathbb{F}_2^{n+v} , then a multivariate interpolation allows to find the coefficients of p . Since \mathbf{p} is not yet known, each evaluation of \mathbf{p} in $\mathbf{a} \in \mathbb{F}_2^{n+v}$ is computed from the secret-key as follows:

$$p(\mathbf{a}) = (\pi \circ \mathcal{T} \circ \mathcal{F} \circ \mathcal{S})(a) = \mathcal{F}(a \cdot \mathbf{S}) \cdot \tilde{\mathbf{T}} \in \mathbb{F}_2^m, \quad (1)$$

where $\tilde{\mathbf{T}}$ is \mathbf{T} without its Δ last columns. This method can be easily simplified with a smart choice of the evaluation points. In [9], the authors consider points having a Hamming weight less or equal to two. Let \mathbf{e}_i be the i -th row of the $n+v \times n+v$ identity matrix in \mathbb{F}_2 , and let p_i be as follows:

$$p_i = \sum_{j=1}^{n+v} \sum_{k=1}^{i-1} c_{i,j,k} x_j x_k + \sum_{j=1}^{n+v} c_{i,j} x_j + c \in \mathbb{F}_2[x_1, \dots, x_{n+v}],$$

with $p_{i,j}, p_i, c \in \mathbb{F}_2^m, \forall i, j, 1 \leq j < i \leq n+v$. Then, we have:

1. $p_i(\mathbf{0}_{n+v}) = c$,
2. $p_i(\mathbf{e}_i) = c + c_i$, for all $i, 1 \leq i \leq n+v$,
3. $p_i(\mathbf{e}_i + \mathbf{e}_j) = c + c_i + c_j + c_{i,j}$, for all $i, j, 1 \leq j < i \leq n+v$.

We deduce easily the coefficients of the public-key. This method is adapted for low-end devices, since quadratic terms of the public-key can be computed one by one.

2.2 Constant-Time gcd

In [3], the authors introduce a constant-time gcd. The latter is based on the Euclid-Stevin relationship [13]. Given F and H in $\mathbb{F}_{2^n}[X]$ of degrees $d_f \geq d_h$ respectively, we compute $F = d_h F - d_f H X^{d_f-d_h}$ until $d_f < d_h$. Let F_1 be the last computed value of F . Thus, we have $\gcd(F, H) = \gcd(H, F_1)$. The idea of [3] is to use this relationship in constant-time. This implies to compute $d_f + d_h$ Euclid-Stevin relationship, by swapping F and H in constant-time when $d_f < d_h$. In practice, the constant-time version of the Euclid-Stevin algorithm is a little bit more tricky and we refer to [3] for more details.

2.3 Constant-time Root Finding

We propose to modify the GeMSS implementation to obtain a constant-time signing process. The root finding algorithm is currently implemented with a constant-time Frobenius map and a constant-time gcd, but the root finding of split polynomials has a time that depends of the degree (which is the number of roots). In Quartz [12] and Gui (round-1 candidate), the strategy is to select one root only if the latter is unique. However, this method generates a theoretical slow-down of 1.72 [8], compared to the selection of roots when they exist.

So, we propose to extend this strategy, by introducing constant-time solver for degree two and degree three split polynomials in $\mathbb{F}_{2^n}[X]$. By selecting roots only if there are one or two, the theoretical slow-down drops to 1.15. Then, an additional slow-down is due to the fact that degree one polynomial must be solved with the same time than the degree two. This process can also be performed by selecting roots only if there are one, two or three. In this case, the theoretical slow-down drops to 1.03. In return, the cost to solve degree three in constant-time is larger than degree two, which can give worst performance than the previous method. Therefore, the best strategy between allowed one or two roots, or one, two and three roots, should be chosen in function of the available implementation and security parameters.

Solver of degree two and degree three split polynomials can be implemented with linear algebra, respectively by using that $X^2 + X + A$ and $X^4 + sX^2 + pX = X \cdot (X^3 + sX + p)$ are \mathbb{F}_2 -linear. For degree two split polynomials, we can also use the half-trace when n is odd.

2.4 Performance Results

We present the performance of our new implementation, which is mainly impacted by the keypair generation by evaluation-interpolation (Section 2.1) and by the constant-time gcd (Section 2.2). The conditions of experiments are identical to these of Section B.2.

2.4.1 Reference Implementation

In Table 4, we summarize timings of the reference implementation. This code has not been modified, excepted about the use of the half-trace (Section 2.3).

scheme	$(\lambda, D, n, \Delta, v, \text{nb_ite})$	key gen. (MC)	sign (MC)	verify (KC)
GeMSS128	(128, 513, 174, 12, 12, 4)	140	2420	211
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	108	473	236
RedGeMSS128	(128, 17, 177, 15, 15, 4)	89.2	49.4	242
GeMSS192	(192, 513, 265, 22, 20, 4)	600	6310	591
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	506	1290	603
RedGeMSS192	(192, 17, 266, 23, 25, 4)	413	121	596
GeMSS256	(256, 513, 354, 30, 33, 4)	1660	10600	1140
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	1500	2060	1180
RedGeMSS256	(256, 17, 358, 34, 35, 4)	1290	200	1170
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	138	1810	163
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	117	383	172
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	92.4	37	170
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	620	4940	464
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	529	929	468
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	433	86	464
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	1740	8020	956
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	1540	1700	990
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	1350	157	985

Table 4: Performance of the reference implementation. We use a Skylake processor (LaptopS). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits.

2.4.2 Optimized (Haswell) Implementation

In Table 5, we summarize timings of the optimized implementation. The use of the evaluation-interpolation method for generating the public-key is rather efficient for small degrees as 17. For large degrees as 129 and 513, we use the old keypair generation. To do this, we set to zero the C macro INTERPOLATE_PK_HFE in the file `sign_keypairHFE.c`.

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 / $\times 1.0$	1340 / $\times 0.93$	163 / $\times 1.0$
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	52.1 / $\times 1.0$	195 / $\times 1.01$	168 / $\times 1.01$
RedGeMSS128	(128, 17, 177, 15, 15, 4)	40.5 / $\times 1.29$	4.93 / $\times 1.16$	179 / $\times 1.0$
GeMSS192	(192, 513, 265, 22, 20, 4)	270 / $\times 1.0$	3550 / $\times 0.94$	455 / $\times 1.01$
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	268 / $\times 1.0$	474 / $\times 1.01$	468 / $\times 1.0$
RedGeMSS192	(192, 17, 266, 23, 25, 4)	228 / $\times 1.16$	12.8 / $\times 1.07$	477 / $\times 0.99$
GeMSS256	(256, 513, 354, 30, 33, 4)	816 / $\times 1.0$	5670 / $\times 0.95$	979 / $\times 0.99$
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	810 / $\times 1.0$	736 / $\times 1.0$	990 / $\times 1.0$
RedGeMSS256	(256, 17, 358, 34, 35, 4)	793 / $\times 1.02$	20.4 / $\times 1.08$	1010 / $\times 1.01$
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	52.4	997	118
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	53.5	157	125
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	41.3	4.03	129
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	281	2640	357
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	281	370	364
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	231	9.93	368
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	844	4400	819
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	846	553	833
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	770	16.4	845

Table 5: Performance of the optimized implementation, followed by the speed-up between the new and the previous implementation (Table 13). We use a Haswell processor (ServerH). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits. For example, $228 / \times 1.16$ means a performance of 228 KC with the new code, and a performance of $228 \times 1.16 = 264$ KC with the old code.

2.4.3 Additional (Skylake) Implementation

In Table 6, we summarize obtained speed-ups compared to these of April 2020 (Section B.2.3). The use of the evaluation-interpolation method for generating the public-key is rather efficient for degrees as 17 and 129. For larger degrees as 513, we use the old keypair generation. The use of the constant-time Euclid-Stevin algorithm improves the signing process for small degrees but not for large degrees. Indeed, this strategy uses $D - 1$ multiplications in \mathbb{F}_{2^n} instead of an inversion in $\mathbb{F}_{2^n}^\times$ for a classical Euclidean algorithm. For small degrees, the cost of $D - 1$ multiplications is less than this of one inverse, whereas we have the opposite behavior for large degrees.

scheme	$(\lambda, D, n, \Delta, v, \text{nb_ite})$	key gen. (MC)	sign (MC)	verify (KC)
GeMSS128	(128, 513, 174, 12, 12, 4)	51.9 / $\times 1.01$	1080 / $\times 0.96$	163 / $\times 1.01$
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	51.5 / $\times 1.04$	154 / $\times 1.07$	174 / $\times 1.01$
RedGeMSS128	(128, 17, 177, 15, 15, 4)	41.1 / $\times 1.32$	4.37 / $\times 1.2$	183 / $\times 1.01$
GeMSS192	(192, 513, 265, 22, 20, 4)	274 / $\times 1.01$	3170 / $\times 0.94$	495 / $\times 1.01$
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	262 / $\times 1.06$	445 / $\times 1.01$	509 / $\times 1.01$
RedGeMSS192	(192, 17, 266, 23, 25, 4)	221 / $\times 1.26$	12 / $\times 1.1$	514 / $\times 1.01$
GeMSS256	(256, 513, 354, 30, 33, 4)	915 / $\times 1.0$	5300 / $\times 0.93$	1120 / $\times 1.01$
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	856 / $\times 1.08$	658 / $\times 0.99$	1130 / $\times 1.01$
RedGeMSS256	(256, 17, 358, 34, 35, 4)	765 / $\times 1.2$	19.5 / $\times 1.1$	1140 / $\times 1.03$
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	52.9	815	112
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	54.4	119	116
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	41.9	3.51	125
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	287	2380	388
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	289	339	396
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	223	9.38	401
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	960	3910	914
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	963	529	911
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	750	15.6	936

Table 6: Performance of the additional implementation, followed by the speed-up between the new and the previous implementation (Table 14). We use a Skylake processor (LaptopS). MC (resp. KC) stands for Mega (resp. Kilo) Cycles. The results have three significant digits. For example, $41.1 / \times 1.32$ means a performance of 41.1 KC with the new code, and a performance of $41.1 \times 1.32 = 54.3$ KC with the old code.

2.5 New Family of Parameters for Low-End Devices

The multiplication in \mathbb{F}_{2^n} is a crucial operation for the performance of GeMSS. On low-end devices, this operation is naturally very expensive.

So, we design new parameters especially for these devices. It is well-known that multiplications in \mathbb{F}_{2^4} and \mathbb{F}_{2^8} can be efficiently computed in parallel, with vector instructions and logarithm tables (as in the Rainbow submission). A solution to improve performance of the multiplication in \mathbb{F}_{2^n} is to choose n multiple of four or eight. This allows to use an isomorphism between \mathbb{F}_{2^n} and $\mathbb{F}_{16^{\frac{n}{4}}}$ or $\mathbb{F}_{256^{\frac{n}{8}}}$. We base our new parameters on these of GeMSS, by modifying slightly the balance between minus and vinegar. We obtain Table 7.

The proposed parameters are a proposal to improve the performance of GeMSS on low-end devices. We do not have implementations allowing to estimate obtained speed-ups (but the implementation supports these parameters).

level	nb_ite	m	D	$\Delta + v$	$n = 0 \bmod 4, \Delta, v$	$n = 0 \bmod 8, \Delta, v$
I	4	162	513	24	172, 10, 14	176, 14, 10
			129	27	176, 14, 13	
			17	30	176, 14, 16	
III		243	513	42	264, 21, 21	
			129	45	264, 21, 24	
			17	48	268, 25, 23	264, 21, 27
V		324	513	63	356, 32, 31	352, 28, 35
			129	66	356, 32, 34	360, 36, 30
			17	69	360, 36, 33	

Table 7: Modification of GeMSS for low-end devices.

2.6 Third Party Analysis

In [2], the authors proposed a new technique for solving MinRank. This attack improved all previous **MinRank** attacks against HFEv--based schemes. However, they do not impact the security of GeMSS. For security level I (respectively III and V) of GeMSS, this attack requires 2^{158} (respectively 2^{224} and 2^{304}) bit operations. The complexity is similar (but slightly larger) for BlueGeMSS and RedGeMSS.

2.7 WhiteGeMSS, CyanGeMSS and MagentaGeMSS : A New Family of Parameters

The report [11] pointed that “*It is possible that there may yet be additional trade-offs to further improve performance. In particular, the consideration of the number of bit operations involved in a hash collision attack may warrant a reevaluation of the number of iterations required in the Feistel-Patarin transformation*”.

In this part, we show that it is indeed possible to decrease the number of iterations in the Feistel-Patarin scheme. To support this fact, we performed a detailed analysis of generic attacks against the Feistel-Patarin construction (and slightly improved the state-of-the-art). As a consequence, we conclude that decreasing the number of iterations (from 4 to 3) only requires to slightly increase the number of equations. This improvement is possible thanks to the fact that evaluating a system of algebraic equations is strictly slower than evaluating SHA-3. All in all, this has a very modest impact on the size of the public-key but improves 1) the efficiency of GeMSS and 2) also decreases the size of the signature.

2.7.1 General Idea

Following [12, 4], the number of iterations nb_ite in GeMSS was chosen such that

$$2^m \frac{\text{nb_ite}}{\text{nb_ite}+1} \geq 2^\lambda, \quad (2)$$

with $\lambda \in \{128, 192, 256\}$ being the security parameter.

This inequality is derived by assuming that the cost of evaluating the public-key polynomials is the same than evaluating a hash function. Let g be the \log_2 of the ratio between the number of gates

required to evaluate the public-key polynomials and the number of gates required to evaluate the hash function. Then, we can prove that (2) should be more precisely:

$$2^m \frac{\text{nb_ite}}{\text{nb_ite}+1} \geq 2^{\lambda-g}, \quad (3)$$

This is the basic fact allowing to decrease the number of iterations whilst almost keeping the same m . In the next part, we will approximate the value of g and detail the analysis leading to (3).

2.7.2 Detailed Analysis

The principle of the generic attack against Feistel-Patarin [4] is as follows. Let $G : \mathbb{F}_2^{n+v} \rightarrow \mathbb{F}_2^m$ be a trapdoor function and $H_1 : \{0, 1\}^* \rightarrow \mathbb{F}_2^m$ be a hash function returning an element of \mathbb{F}_2^m . Assume that a signature $\text{sm} \in \mathbb{F}_2^{n+v}$ is obtained as the inverse of G evaluated in the hash of a message $d \in \{0, 1\}^*$, *i.e.*

$$\text{sm} = G^{-1}(H_1(d)). \quad (4)$$

As described in [4], the attack requires to generate from random inputs, an inversion table of ℓ evaluations of G . Then, this table is used to try to sign random messages, *i.e.* to compute Equation (4). The table contains ℓ elements. So, the probability to invert G , for a random input, is $\frac{\ell}{2^m}$. Thus, $\frac{2^m}{\ell}$ attempts are required to forge a valid signature with very high probability. The cost of the generic attack is ℓ evaluations of G , as well as $\frac{2^m}{\ell}$ computations of H_1 . Its memory costs is ℓm bits.

Now, assume that the Feistel-Patarin construction is used with $\text{nb_ite} > 1$. The probability to invert nb_ite times G becomes $\left(\frac{\ell}{2^m}\right)^{\text{nb_ite}}$. The cost of the generic attack is still ℓ evaluations, and the number of computations is at least $\left(\frac{2^m}{\ell}\right)^{\text{nb_ite}}$. The memory cost is unchanged.

So, the cost of the generic attack on GeMSS is lower bounded by

$$\min_{\ell} \left(\ell \cdot C_G + \left(\frac{2^m}{\ell} \right)^{\text{nb_ite}} \cdot C_{H_1} \right), \quad (5)$$

where C_G is the cost to evaluate G and C_{H_1} is the cost to compute H_1 .

In [4], the author considered the value of ℓ which balances the number of evaluations and hash, *i.e.* $\ell = \left(\frac{2^m}{\ell}\right)^{\text{nb_ite}} = 2^{\frac{\text{nb_ite}}{\text{nb_ite}+1}m}$. This choice of ℓ is optimal when $C_G = C_{H_1}$. But in practice, the evaluation of the public-key is more expensive than evaluating SHA-3, *i.e.* $C_G > C_{H_1}$. Thus, we can take this fact in consideration to decrease the number of iterations of GeMSS. To do that, the fundamental point is to evaluate the gap between C_G and C_{H_1} .

In Table 8, we summarize some experiments to estimate the ratio C_G/C_{H_1} . We compare our best and state-of-the-art (variable-time) evaluation function (in AVX2) to the best implementations of SHA-3 from XKCP (*i.e.* the Haswell implementation). We consider the hash of $\frac{\lambda}{2}$ -bit data when SHA3- λ is used. We use SHA3-256 (respectively SHA3-384 and SHA3-512) for level I (respectively III and V).

The minimal required ratio required to reach the given security level (first column) is obtained from Equation (5) by taking $C_{H_1} = 2^{18}$ and the value of ℓ minimizing C_G . The experimental (exp.) ratio corresponds to the ratio of the running time of our public-key evaluation and a SHA-3. Sequential

means that C_{H_1} corresponds to the running time to compute one **SHA-3** hash (in **AVX2**), whereas the parallel version considers the cost to compute four **SHA-3** hash (in **AVX2**), divided by four to obtain the cost of one **SHA-3** hash. All the values has been obtained for 3 iterations.

level	m	$C_G/2^{18}$ (5)	exp. C_G/C_{H_1} (sequential SHA-3)	exp. C_G/C_{H_1} (parallel SHA-3)
I	162	12	10.87	26.79
I	163	6	≥ 10.87	≥ 26.79
I	164	3	≥ 10.87	≥ 26.79
I	165	1.5	≥ 10.87	≥ 26.79
I	166	0.75	≥ 10.87	≥ 26.79
III	243	242	24.47	60.60
III	244	121	≥ 24.47	≥ 60.60
III	245	60.5	≥ 24.47	≥ 60.60
III	246	30.25	≥ 24.47	≥ 60.60
III	247	15.125	≥ 24.47	≥ 60.60
III	248	7.0625	≥ 24.47	≥ 60.60
V	324	12288	55.20	135.15
V	332	48	> 55.20	> 135.15
V	333	24	> 55.20	> 135.15
V	334	12	> 55.20	> 135.15

Table 8: Minimal required ratio between the cost to evaluate a boolean system of m equations in m variables and **SHA-3**. For example, 12 means that **SHA-3** should be at least 12 times faster than evaluating a square system of m polynomials and equations to reach the first security level. We give the experimental ratio on a Skylake processor (LaptopS) using **AVX2** instructions set. We consider the sequential and parallel versions of **SHA-3** from the Extended Keccak Code Package (XKCP), both using **AVX2**.

From Table 8, we see that slightly increasing the number of equations allows to reduce the number of iterations to 3. The results are summarized in Table 9. We assume that $C_{H_1} = 2^{18}$, and we give the minimum required ratio C_G/C_{H_1} to reach security level I, III and V.

m	nb_ite	C_G/C_{H_1}	Optimal ℓ	Generic attack	Memory (bits)
163	3	6	2^{122}	2^{143}	$2^{129.35}$
247		15.11	$2^{184.70}$	$2^{207.00}$	$2^{192.65}$
333		24	2^{249}	2^{272}	$2^{257.38}$

Table 9: Lower bound on the complexity to find a collision with the generic attack , i.e. Equation (5). Here, we consider $C_{H_1} = 2^{18}$, ℓ calls to \mathbf{p} , $(\frac{2^m}{\ell})^{\text{nb_ite}}$ calls to the hash function, and a memory cost of ℓm bits.

2.7.3 New Parameters

We can then propose a new family of parameters using this analysis: **WhiteGeMSS**, **CyanGeMSS** and **MagentaGeMSS** (Table 10), corresponding respectively to **GeMSS**, **BlueGeMSS** and **RedGeMSS**.

By construction, these new parameters will have a signature and verification processes 33% faster than corresponding round-2 parameters. The signature sizes are also smaller. On the other hand, the public-key is slightly bigger, but less than 6% compared to the original schemes.

scheme	$(\lambda, D, n, \Delta, v, \text{nb_ite})$	equations	variables	$ pk $ (KB)	$ sk $ (bits)	sign (bits)
WhiteGeMSS128	(128, 513, 175, 12, 12, 3)	163	187	358.17	128	235
WhiteGeMSS192	(192, 513, 268, 21, 21, 3)	247	289	1293.85	192	373
WhiteGeMSS256	(256, 513, 364, 31, 29, 3)	333	393	3222.69	256	513
CyanGeMSS128	(128, 129, 177, 14, 13, 3)	163	190	369.72	128	244
CyanGeMSS192	(192, 129, 270, 23, 22, 3)	247	292	1320.80	192	382
CyanGeMSS256	(256, 129, 364, 31, 32, 3)	333	396	3272.02	256	522
MagentaGeMSS128	(128, 17, 178, 15, 15, 3)	163	193	381.46	128	253
MagentaGeMSS192	(192, 17, 271, 24, 24, 3)	247	295	1348.03	192	391
MagentaGeMSS256	(256, 17, 366, 33, 33, 3)	333	399	3321.72	256	531

Table 10: Summary of the parameters of WhiteGeMSS, CyanGeMSS and MagentaGeMSS.

Acknowledgement. We would like to thank the NIST PQC team, and in particular Daniel Smith-Tone, for pointing to us the improvement detailed in this section.

Appendix

A Changes for the second round (April 1st, 2019)

The goal of this part is to summarize the modifications done on GeMSS for the second round.

A.1 Large set of parameters

The parameters proposed for GeMSS in the first round were very conservative in term of security. In [10], 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 GeMSS 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 GeMSS proposed in the first round, and two new more aggressive parameters (BlueGeMSS and RedGeMSS).
 - RedGeMSS128 is 269 times faster than GeMSS128.
 - BlueGeMSS128 is 7.08 times faster than GeMSS128.
- We design a family of possible values that depends on only one parameter n . We call this family FGeMSS(n) (Section 9.4).

A.2 Further details on known attacks

The paper [5] was published at about the same time than the deadline for the first round. In this revision, we further details [5] 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 [5]. 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.

A.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 GeMSS. We have a paper accepted at CHES on this topic [7, 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 GeMSS, 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 [6]).

Our new submitted implementations are more secure against timing attacks.

- We have removed NTL¹ 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" [7] 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/>

B 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 B.1). For the public-key, we have improved the implementation to reach the theoretical size. We introduce a so-called "hybrid representation" [7], which allows to keep an efficient evaluation (Section B.3). The changes in our implementation are enumerated in Section B.4. In particular, the old KATs files are no longer valid since we have changed the keys format.

B.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 11, 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 B.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	$(\lambda, D, n, \Delta, v, \text{nb_ite})$	$ pk $ (KB)	$ sk $ (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 11: Memory cost. 1 KB is 1000 bytes.

B.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` library². The reference implementation does not exploit these instructions sets. The used machines (LaptopS, ServerH) have not changed and are described in the specification.

B.2.1 Reference implementation

For the second round, we had removed the use of NTL in the optimized and additional implementations (Section A.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 GeMSS. 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 12. 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)
GeMSS128	(128, 513, 174, 12, 12, 4)	145 / $\times 13$	2730 / $\times 2.5$	211 / $\times 140$
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	118 / $\times 13$	530 / $\times 1.46$	228 / $\times 130$
RedGeMSS128	(128, 17, 177, 15, 15, 4)	91.1 / $\times 13$	52 / $\times 0.34$	239 / $\times 110$
GeMSS192	(192, 513, 265, 22, 20, 4)	619 / $\times 13$	6510 / $\times 2.3$	585 / $\times 150$
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	520 / $\times 13$	1290 / $\times 0.99$	592 / $\times 150$
RedGeMSS192	(192, 17, 266, 23, 25, 4)	423 / $\times 14$	126 / $\times 0.22$	627 / $\times 120$
GeMSS256	(256, 513, 354, 30, 33, 4)	1660 / $\times 12$	10500 / $\times 2.4$	1160 / $\times 150$
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	1510 / $\times 13$	2080 / $\times 0.79$	1190 / $\times 150$
RedGeMSS256	(256, 17, 358, 34, 35, 4)	1310 / $\times 14$	203 / $\times 0.18$	1190 / $\times 120$

Table 12: 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, $145 / \times 13$ 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/>

B.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 / $\times 1.01$	1240 / $\times 0.98$	163 / $\times 0.92$
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	52.1 / $\times 1.02$	198 / $\times 1.02$	170 / $\times 0.93$
RedGeMSS128	(128, 17, 177, 15, 15, 4)	52.4 / $\times 1.06$	5.72 / $\times 0.97$	178 / $\times 0.91$
GeMSS192	(192, 513, 265, 22, 20, 4)	270 / $\times 1.01$	3320 / $\times 1.08$	459 / $\times 0.96$
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	268 / $\times 1.07$	481 / $\times 1.09$	468 / $\times 0.94$
RedGeMSS192	(192, 17, 266, 23, 25, 4)	264 / $\times 1.03$	13.7 / $\times 1.01$	474 / $\times 0.96$
GeMSS256	(256, 513, 354, 30, 33, 4)	814 / $\times 1.04$	5380 / $\times 1.32$	973 / $\times 0.97$
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	810 / $\times 1.08$	733 / $\times 1.43$	989 / $\times 0.97$
RedGeMSS256	(256, 17, 358, 34, 35, 4)	805 / $\times 1.07$	22.1 / $\times 1.17$	1010 / $\times 0.97$

Table 13: 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, $163 / \times 0.92$ means a performance of 163 KC with the new code, and a performance of $163 \times 0.92 = 150$ KC with the old code.

B.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)
GeMSS128	(128, 513, 174, 12, 12, 4)	52.6 / $\times 0.97$	1040 / $\times 0.9$	164 / $\times 0.89$
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	53.8 / $\times 0.97$	164 / $\times 0.97$	176 / $\times 0.88$
RedGeMSS128	(128, 17, 177, 15, 15, 4)	54.3 / $\times 0.98$	5.24 / $\times 0.88$	185 / $\times 0.86$
GeMSS192	(192, 513, 265, 22, 20, 4)	275 / $\times 0.96$	2960 / $\times 0.98$	501 / $\times 0.87$
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	278 / $\times 0.96$	448 / $\times 0.96$	512 / $\times 0.86$
RedGeMSS192	(192, 17, 266, 23, 25, 4)	277 / $\times 0.96$	13.1 / $\times 0.9$	518 / $\times 0.87$
GeMSS256	(256, 513, 354, 30, 33, 4)	916 / $\times 0.95$	4940 / $\times 0.98$	1120 / $\times 0.91$
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	923 / $\times 0.96$	653 / $\times 1.06$	1140 / $\times 0.89$
RedGeMSS256	(256, 17, 358, 34, 35, 4)	921 / $\times 0.97$	21.4 / $\times 0.86$	1170 / $\times 0.9$

Table 14: 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, $164 / \times 0.89$ means a performance of 164 KC with the new code, and a performance of $164 \times 0.89 = 146$ KC with the old code.

B.2.4 MQsoft

We give here the times with the latest version of **MQsoft** [7] 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)
GeMSS128	(128, 513, 174, 12, 12, 4)	38.7 / $\times 0.99$	531 / $\times 1.41$	106 / $\times 0.77$
BlueGeMSS128	(128, 129, 175, 13, 14, 4)	39.2 / $\times 1.00$	81.3 / $\times 1.3$	136 / $\times 0.82$
RedGeMSS128	(128, 17, 177, 15, 15, 4)	39.5 / $\times 0.99$	2.33 / $\times 1.2$	141 / $\times 0.77$
GeMSS192	(192, 513, 265, 22, 20, 4)	175 / $\times 1.00$	1800 / $\times 1.29$	304 / $\times 0.79$
BlueGeMSS192	(192, 129, 265, 22, 23, 4)	174 / $\times 0.99$	252 / $\times 1.31$	325 / $\times 0.78$
RedGeMSS192	(192, 17, 266, 23, 25, 4)	173 / $\times 0.99$	5.97 / $\times 1.4$	334 / $\times 0.76$
GeMSS256	(256, 513, 354, 30, 33, 4)	530 / $\times 1.00$	3020 / $\times 1.21$	678 / $\times 0.83$
BlueGeMSS256	(256, 129, 358, 34, 32, 4)	530 / $\times 1.00$	399 / $\times 1.37$	684 / $\times 0.85$
RedGeMSS256	(256, 17, 358, 34, 35, 4)	534 / $\times 0.98$	9.82 / $\times 1.31$	704 / $\times 0.84$

Table 15: 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, $106 / \times 0.77$ means a performance of 106 KC with the new code, and a performance of $106 \times 0.77 = 82$ KC with the old code.

B.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" [7]. 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.

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

Figure 1: Example of hybrid representation 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 $\text{nb_ite} > 1$, 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 alignment 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.

B.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] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray A. Perlner, Daniel Smith-Tone, Jean-Pierre Tillich, and Javier A. Verbel. Algebraic attacks for solving the rank decoding and minrank problems without gröbner basis. *CoRR*, abs/2002.08322, 2020.
- [3] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] Giordano Fusco and Eric Bach. *Phase Transition of Multivariate Polynomial Systems*, pages 632–645. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [9] Tsutomu Matsumoto and Hideki Imai. Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. In *EUROCRYPT*, volume 330 of *Lecture Notes in Computer Science*, pages 419–453. Springer, 1988.
- [10] NIST. Status report on the first round of the nist post-quantum cryptography standardization process.
- [11] NIST. Status report on the second round of the nist post-quantum cryptography standardization process.
- [12] Jacques Patarin, Nicolas Courtois, and Louis Goubin. Quartz, 128-bit long digital signatures. In David Naccache, editor, *Topics in Cryptology - CT-RSA 2001, The Cryptographer’s Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2001.

[13] Simon Stevin. *L'arithmétique*. Imprimerie de Christophe Plantin, 1585.