On Roots Factorization for PQC Algorithms

Alexander Maximov

Ericsson Research, Lund, Sweden
alexander.maximov@ericsson.com

Abstract. In this paper we consider several methods for an efficient extraction of roots of a polynomial over large finite fields. The problem of computing such roots is often the performance bottleneck for some multivariate quantum-immune cryptosystems, such as HFEv-based Quartz, Gui, etc. We also discuss a number of techniques for fast computation of traces as part of the factorization process. These optimization methods could significantly improve the performance of cryptosystems where roots factorization is a part thereof.

Keywords: NIST · PQC · HFE · Gui · Quartz · Roots · Factorization · Traces

1 Introduction

The development of multivariate quantum-immune cryptography is progressing rapidly, and some cryptographic primitives, such as Gui, have been proposed to NIST as part of the Post-Quantum Cryptography (PQC) Standardisation effort [NIS20a, NIS20b].

The class of Multivariate Quadratic (MQ) public keys cryptosystems with Hidden Field Equations (HFE) based schemes, such as Rainbow, Quartz, Gui, etc [PCG01, DPCY14, ZT15, PCY$^+$15], require at their core the computation of roots of a polynomial over a large (presumably binary) field. This part of computation in a scheme often appears the performance bottleneck for such cryptosystems. For example, as a consequence, the Quartz scheme can only perform 1 signature per about 10 seconds. Hence, approaches to improve methods for computing roots of polynomials over large finite fields may have a positive impact on several PQC schemes. Of course, improved methods may also have other applications such as computing an isomorphism between two finite fields, etc.

In this paper we consider factorization algorithms in order to find roots of a polynomial over a large finite field; also we consider various techniques to compute traces as this is the critical part of the factorization core. We show that the steps of a polynomial factorization over a large finite field can be implemented by using a linear function $\Gamma_k(\xi) = \xi^{q^{2^k}}$ in an extension field of characteristic q. Particularly, this opens-up for multiple implementation choices, which are no longer limited to only the polynomial modular composition as proposed by Shoup et. al. in [KS97, vzGS92a] for trace computation and factorization purposes.

The paper is organized as follows. In Section 2 we introduce notations and review basic approaches for roots factorization algorithms. In Section 3 and Section 4 we consider methods for fast computation of traces. Finally, in Section 5 we give some software performance results for Quartz and a number of versions of Gui with various scheme configuration parameters, and conclude the paper.

2 Preliminaries

2.1 Notations

For the sake of notation, let us consider a monic primitive polynomial $f(x) = x^n + c_{n-1}x^{n-1} ... + c_0$ over $GF(q)$ of degree n, i.e., over some prime field with characteristic q, and let this polynomial be the generating polynomial of the extension field $\mathbb{F}_{q^n} = GF(q)[x]/f(x)$. The number of elements in $\mathbb{F}_{q^n}$ is

$$N = q^n.$$

Consider a polynomial $F(Z) = f_D Z^D + ... f_1 Z + f_0$ of degree D over the base field $\mathbb{F}_{q^n}$, i.e., the coefficients f_i can be represented as polynomials modulo $f(x)$ of degree at most $n-1$ and represent elements of the base field $\mathbb{F}_{q^n}$. For the purpose of this paper we assume that n is large (e.g., $n \approx 100 \ldots 400$).

2.2 Roots factorization algorithms

A crucial task in the mentioned PQC schemes is the factorization of the polynomial $F(Z)$, and extraction from that root values $R_1, R_2, \ldots$ such that $F(R_i) = 0$ in $\mathbb{F}_{q^n}$. Many factorization algorithms of polynomials usually have the following three steps:

- Square-free factorization
- Distinct-degree factorization
- Equal-degree factorization

with an exception of Berlekamp's algorithm that combines the last two steps.

For the purpose of finding roots of the equation $F(Z) = 0$, we need to perform only the following steps:

- Extract the product $R(Z)$ of all monic 1-degree factors of $F(Z)$;
- Perform equal-degree factorization.

where $R(Z) = (Z - R_1)(Z - R_2)...(Z - R_r)$ is the monic polynomial of degree r that is the product of $(Z - R_i)$s for all r roots R_i of $F(Z)$ such that $F(R_i) = 0$. The number of roots is then denoted as r. When the degree r of $R(Z)$ is 0 then $F(Z)$ has no roots in the base field. When $r = 1$ then $F(Z)$ has exactly one root and it is the first coefficient of $R(Z)$. For $r > 1$ we need to perform an equal-degree factorization.

Some PQC schemes, such as the original Quartz scheme, require to sample for $F(Z)$ that have only an unique root. In those schemes we can thus ignore the second step.

In the subsequent sections we go through each of the above two steps.

2.3 Step 1: Product of all roots $R(Z)$ given $F(Z)$

The polynomial $R(Z)$ is the product of all 1-degree monic factors of $F(Z)$ and it can be obtained by computing

$$R(Z) = \gcd(F(Z);\ Z^N - Z) = \gcd(F(Z);\ Z^{q^n} - Z \mod F(Z)),$$

since $Z^N - Z$ is the product of all 1-degree monic polynomials over the extension field $\mathbb{F}_{q^n}$.

2.4 Step 2: Equal-degree factorization of $R(Z)$

In this section we assume that $R(Z)$ is a monic polynomial of degree $r > 1$, for which we seek its factorization.

One approach to factor out the roots from $R(Z)$ is to use Berlekamp's algorithm, [Ber67]. However, the complexity of this algorithm appears to be $O(r^3 + r^2q^n)$ operations in $\mathbb{F}_{q^n}$, [vzGP01], which is large and not applicable in our case.

Alternatively, factorization algorithms over large base fields $\mathbb{F}_{q^n}$ utilize another technique. Because it is the product of all 1-degree monic factors of $F(Z)$, the given $R(Z)$ forms an extension ring $\mathbb{R}^r_{q^n}$:

$$\mathbb{R}^r_{q^n} = \mathbb{F}_{q^n}[Z]/R(Z).$$

Thereafter, if we take a random element $Y \in \mathbb{R}^r_{q^n}$ then

$$Y^N - Y = 0 \mod R(Z).$$

Odd characteristic. For *odd characteristic* the splitting of $R(Z)$ into two smaller factors can be done using the Cantor-Zassenhaus [CZ81] algorithm, since:

$$\gcd(R;\ Y^N - Y) = \gcd(R;\ Y \cdot (Y^{(N-1)/2} - 1) \cdot (Y^{(N-1)/2} + 1)),$$

and, therefore, the computation of

$$\gcd(R;\ Y^{(N-1)/2} - 1)$$

gives a non-trivial factor of $R(Z)$ with probability $\approx 1 - (1/2)^{r-1}$.

Even characteristic. The above method, however, does not work for fields with *even characteristic* i.e., $q = 2$. A different technique is then applied. The square-free factorization of $R(Z)$ over a large *binary* field with characteristic 2 can be done with, e.g., a modified Berlekamp's algorithm given in [Ber70].

Let us define:

$$W(Z) = \sum_{i=0}^{n-1} Y^{2^i} \text{over } \mathbb{F}_{2^n},$$

then $Y^{2^n} - Y = W \cdot (W + 1)$, which means that

$$\gcd(R;\ Y^{2^n} - Y) = \gcd(R;\ W) \cdot \gcd(R;\ W - 1).$$

Each root will fall into W or $(W-1)$'s "bin" with probability $1/2$, so the overall probability to get a non-trivial factor will be $1 - (1/2)^{r-1}$, see Theorem 8.12 in [Ric09] for details.

2.5 Using traces for factorization

As it will become clear, traces are very useful in solving the factorization problem. Consider the following trace for some element ξ defined as the function

$$\mathtt{Tr}(\xi, n) = \sum_{i=0}^{n-1} \xi^{q^i} \quad \text{over } K, \tag{1}$$

where K is some field (e.g., $K = \mathbb{F}_{q^n}$). Then we get that

$$\mathtt{Tr}(\xi, n)^q - \mathtt{Tr}(\xi, n) = (\xi^{q^n} - \xi) \text{ over } \mathbb{F}_{q^n},$$

since $(a + b)^q = a^q + b^q$ in a field with characteristic q. Hence, we also get that $\mathtt{Tr}(\xi, n)$ divides $\xi^{q^n} - \xi$, i.e.,

$$\xi^{q^n} - \xi = \mathtt{Tr}(\xi, n) \cdot (\mathtt{Tr}(\xi, n)^{q-1} - 1).$$

Firstly, in any field of characteristic q we can get the product of roots $R(Z)$ via the computation of traces as follows:

$$R(Z) = \gcd(F(Z);\ (Z^{q^n} - Z)) = \gcd(F(Z);\ \texttt{Tr}(Z, n)^q - \texttt{Tr}(Z, n) \mod F(Z)).$$

Secondly, in fields with even characteristic, $q = 2$, the polynomial $R(Z)$ can be split into two smaller subfactors by computing

$$\gcd(R(Z);\ W(Z)) = \gcd(R(Z);\ \texttt{Tr}(Y(Z), n) \mod R(Z)),$$

with the success probability

$$1 - \left(\frac{1}{2}\right)^{r-1},$$

where r is the degree of $R(Z)$. Note that in the first splitting we can get the trace value "for free" by doing

$$\gcd(R(Z);\ \texttt{Tr}(Z, n) \mod F(Z)),$$

where $\texttt{Tr}(Z, n) \mod F(Z)$ has already been computed in the first step.

Thirdly, in any field with odd field characteristic, $q \neq 2$, the polynomial $R(Z)$ can be split into subfactors in a similar way as we do for the even characteristic:

$$\gcd(R(Z);\ \texttt{Tr}(Y(Z), n) \mod R(Z)),$$

for a random $Y(Z) \in \mathbb{R}^r_{q^n}$, but the probability that a single root will fall into the "bin" $\texttt{Tr}(Y(Z), n)$ is $1/q$, and the other "bin" is $(q-1)/q$, which leads to the probability of successful splitting of $R(Z)$ to be

$$1 - \left(\frac{1}{q}\right)^r - \left(1 - \frac{1}{q}\right)^r = \left[(q^r - 1) - (q-1)^r\right]/q^r.$$

For example, for $r = 2$ roots and field characteristic $q = 3$ the success of such a splitting will be $4/9 \approx 0.444$, but for larger $q = 7$ the probability drops to $12/49 \approx 0.245$.

2.6 Trace computation

The critical part of roots factorization appears to be the part of computing traces of $\xi(Z)$ modulo a given random polynomial $F(Z)$ or $R(Z)$. However, the base field $\mathbb{F}_{q^n}$ is usually known in advance and fixed.

The naive approach is to follow Equation 1 so that: we start with the given ξ, then compute ξ^q and sum, then power again to ξ^{q^2} and sum, and so on until we get $\xi^{q^{n-1}}$. Thus we need $n-1$ modular powerings to q. For the even characteristic case $q = 2$ we thus need $n-1$ squarings.

In [vzGS92b] V. Shoup and Von zur Gathen proposed a nice algorithm to compute a trace using a polynomial convolution technique. Thus, computing $\texttt{Tr}(\xi, n)$ can be done in $\log(n)$ steps instead of the classical $n-1$ modular squarings (or powerings to q). However, the basic step there is the computation of convolution of elements in the base field, which is quite expensive and at best it can be done with n modular multiplications in the field $\mathbb{F}_{q^n}$.

In case of Gui, where $D \geq 5$ (the small value of D comes from the proposal in [PCY$^+$15]) and $q = 2$ are small, but $n \approx 100 \ldots 400$ is large, the complexity of the polynomial convolution becomes too expensive in time.

3 Fast computation of traces

3.1 Preliminaries

Let us have a field $\mathbb{F}_{q^n}$ with characteristic q with the generating primitive polynomial $f(x)$ over $\mathrm{GF}(q)$ of degree n. Also, let us have a polynomial $R(Z)$ of degree r over $\mathbb{F}_{q^n}$, which then forms an extension ring $\mathbb{R}^r_{q^n} = \mathbb{F}_{q^n}[Z]/R(Z)$.

For any element $\xi(Z) \in \mathbb{R}^r_{q^n}$ we would like to compute the trace

$$\mathtt{Tr}(\xi, n) = \sum_{i=0}^{n-1} \xi^{q^i} \mod R(Z).$$

Normally, this would require n polynomial powerings to q modulo the polynomial $R(Z)$, but for a more efficient computation we would like to introduce the following function

$$\Gamma_k(\xi) = \xi^{q^{2^k}} \mod R(Z), \text{ for } \forall k = 0, 1, 2, ...$$

and then we derive the needed trace via that function, instead. Later on we will show how that function may be implemented in an efficient way with complexity $O(\log(n))$.

Some properties of Γ function include the following equations:

$$\begin{aligned} \Gamma_k(a+b) &= \Gamma_k(a) + \Gamma_k(b), \\ \Gamma_{k+1}(\xi) &= \Gamma_k(\Gamma_k(\xi)). \end{aligned}$$

3.2 Partial traces

Let us define the kth partial trace as follows:

$$\mathtt{Tr}_k(\xi, m) = \sum_{i=0}^{m-1} \xi^{q^{i\cdot 2^k}} \mod R(Z).$$

Hence, the full trace $\mathtt{Tr}(\xi, n)$ is the partial trace with $k = 0, m = n$. We then get the following recursive formulae:

$$\begin{aligned} \Gamma_{k-1}(\mathtt{Tr}_k(\xi, m)) + \mathtt{Tr}_k(\xi, m) &= \left(\sum_{i=0}^{m-1} \xi^{q^{i\cdot 2^k}}\right)^{q^{2^{k-1}}} + \sum_{i=0}^{m-1} \xi^{q^{i\cdot 2^k}} \\ = \sum_{i=0}^{m-1} \left[\left(\xi^{q^{i\cdot 2^k}}\right)^{q^{2^{k-1}}} + \xi^{q^{i\cdot 2^k}}\right] &= \sum_{i=0}^{m-1} \left[\xi^{q^{(2i+1)\cdot 2^{k-1}}} + \xi^{q^{2i\cdot 2^{k-1}}}\right] \\ &= \sum_{i=0}^{2m-1} \xi^{q^{i\cdot 2^{k-1}}} = \mathtt{Tr}_{k-1}(\xi, 2m). \end{aligned} \tag{2}$$

Thus, $\mathtt{Tr}_k(\xi, m)$ divides $\mathtt{Tr}_{k-1}(\xi, 2m)$ since

$$\mathtt{Tr}_{k-1}(\xi, 2m) = \mathtt{Tr}_k(\xi, m) \cdot (\mathtt{Tr}_k(\xi, m)^{q^{2^{k-1}}-1} + 1).$$

The computation of the partial trace may be done as follows. Let us introduce the following sums

$$S_t(\xi) = \sum_{i=0}^{2^t-1} \xi^{q^{i\cdot 2^k}},$$

which can be computed recursively as

$$S_{t+1} = S_t + S_t^{q^{2^{t+k}}} = S_t + \Gamma_{t+k}(S_t),$$

where $S_0 = \xi$. Then the partial trace can be expressed recursively via S and Γ as

$$\mathtt{Tr}_k(\xi, m) = S_t + \Gamma_{t+k}(\mathtt{Tr}_k(\xi, m - 2^t)), \text{ where } t = \lfloor \log_2(m) \rfloor.$$

This means that computing the partial trace would require $\lfloor \log_2(m) \rfloor + H(m)$ applications of Γ_t functions, where $H(m)$ is the Hamming weight of m. I.e., the complexity to compute the partial trace is then $O(\log(m))$ with Γ_t being the basic operation. Summarizing the above, the algorithm for computing the partial trace for any m and k is then as follows.

```
PartialTrace($\xi$, $m$, $k$) -> T=Tr$_k(\xi, m)$
set $S = \xi$ and $T = 0$
for(int t=0; m; m>>=1, ++t)
{  if(m&1)
        $T = S + \Gamma_{t+k}(T)$
   $S = S + \Gamma_{t+k}(S)$
}
return $T$;
```

3.3 Computation of all partial traces and the full trace

The summary algorithm that computes all partial traces $T_1 ... T_k$ and the full trace T_0 can be written this way.

```
AllPartialTraces($\xi$, $n$, $k : 2^k | n$) -> $\{T_0, ..., T_k\} : T_{i+1} | T_i, \forall i \in [0..k-1]$
compute $T_k$=PartialTrace($\xi, n/2^k, k$)
for(int t=k-1; t>=0; --t)
   compute $T_t = T_{t+1} + \Gamma_t(T_{t+1})$
```

The algorithm has complexity $O(\log(n))$ with exactly $\lfloor \log_2(n) \rfloor + H(n)$ applications of Γ_t function, for $t = 0, ..., \lfloor \log_2(n) \rfloor$.

3.4 Computation of ξ^{q^t}

One method is to use the trace:

$$\xi^{q^t} = \mathtt{Tr}(\xi, t) \cdot (\mathtt{Tr}(\xi, t)^{q-1} - 1),$$

or, we can use the $\Gamma(\cdot)$ function:

```
PowerOnly($\xi$, t) -> $Q = \xi^{q^t}$
set $Q = \xi$
for(i=0; t; t>>=1, ++i)
    if(t&1)
        $Q = \Gamma_i(Q)$
return $Q$.
```

This function may be useful for the first step of the factorization algorithm where we need to compute $Z^{q^n} - Z$, and that can be done with $H(n)$ applications of the Γ_k function, instead of $H(n) + \lfloor \log_2(n) \rfloor$ applications in case of computing all partial traces.

3.5 Application of partial traces for factorization

In the first step of factorization we compute $R(Z) = \gcd(F(Z);\ Z^{q^n} - Z)$ that can be done via partial traces as shown in the previous sections. Those partial traces can be used for the first splittings of $R(Z)$, without having to compute other traces modulo $R(Z)$ as the first step of the splitting.

In the second step, in order to split $R(Z)$ into subfactors, we pick a random $Y(Z)$ mod $R(Z)$ and then compute its trace modulo $R(Z)$, then take gcd with $R(Z)$. With the algorithm `AllPartialTraces` we can, instead of a single trace, compute all partial traces $T_0...T_k$, so that we get

$$\gcd(R(Z);\ Z^{q^n} - Z) = \gcd(R(Z);\ T_0 \cdot \Delta_0(Z)).$$

If T_0 is not trivial ($T_0 \neq 1$) then $F_0(Z) = \gcd(R(Z);\ T_0(Z))$ gives a non trivial subfactor $F_0(Z)$. Then we try to factor F_0 further on by using the partial traces in the order of $F_i = \gcd(F_{i-1}(Z);\ T_i(Z))$, since $T_i | T_{i-1}$, and stop when F_i is trivial or all partial traces are used. If some subfactors still need to be factored, then we repeat the computation of new partial traces with a new random $Y(Z)$ modulo the remaining subfactor polynomial.

4 Preparation and computing $\Gamma_k \mod R(Z)$

We have shown that the overall time complexity of roots factorization relies on the ability to compute the Γ_k function fast. That function becomes the most critical one for the overall performance of PQC algorithms that require a roots factorization step.

During the factorization we will have to deal with a random polynomial $R(Z)$ or $F(Z)$ that will serve as the modulus in which we then compute $\Gamma_k(\xi) = \xi^{q^{2^k}}$, for $k = 0, \ldots, \lfloor \log_2(n) \rfloor$.

4.1 Further splitting of the trace computation problem

For a polynomial $\xi(Z)$ of degree $r-1$, taken modulo $R(Z)$, the application of $\Gamma_k(\xi)$ can be written as

$$\Gamma_k(\xi(Z)) = \left(\sum_{i=0}^{r-1} \xi_i \cdot Z^i\right)^{q^{2^k}} = \sum_{i=0}^{r-1} \xi_i^{q^{2^k}} \cdot \left(Z^{q^{2^k}}\right)^i.$$

If we now define

$$\gamma_k(\xi)(Z) = \sum_{i=0}^{r-1} \xi_i^{q^{2^k}} \cdot Z^i,$$

then we can rewrite the above expression as

$$\Gamma_k(\xi(Z)) = \sum_{i=0}^{r-1} \xi_i^{q^{2^k}} \cdot (\Gamma_k(Z))^i = \gamma_k(\xi)(Z) \circ \Gamma_k(Z) = \gamma_k(\xi)(\Gamma_k(Z)).$$

That is, the evaluation of Γ_k is now split into an evaluation of γ_k coefficient-wise of ξ, and an evaluation of $\Gamma_k(Z)$.

4.2 Evaluation of $\gamma_k(\xi) \in \mathbb{F}_{q^n}$

$\gamma_k(\xi) \in \mathbb{F}_{q^n}$ is a linear function in the field $\mathbb{F}_{q^n}$. The base field is usually known in advance and fixed, which opens-up a variety of different options to implement the computation of $\gamma_k(\xi)$. Note also that we will only use γ_k functions for the maximum value of k such that $2^k \leq n$. I.e., one needs to develop only $0 \leq k \leq \lfloor \log_2(n) \rfloor$ of such functions.

Option 1. For $q = 2$ one can utilize, on an Intel CPU, the `PCLMULQDQ` instruction to speed up the computation of $\xi_i^{2^{2^k}}$ in the base field. That requires at most $2^k \leq n$ field multiplications, but it is a lot faster than the classical and naive binary-wise multiplication, in a general case.

Option 2. On the other hand, that operation is linear and since in most cases the base field is fixed, one could precompute linear matrices to derive that power directly. That means we will have a $n \times n$ matrix of elements in $GF(q)$. For $q = 2$ it is a binary matrix, and with efficient implementations and, perhaps, efficient lookup tables, the computation may be done with $\approx n/8$ XORs of rows of length n bits. In our experience, this method seem to be the fastest for binary fields.

When $q \neq 2$, the classical matrix multiplication approach would take n^2 operations over $GF(q)$, which is comparable to n field multiplications.

Option 3. Polynomial composition in the base field $\mathbb{F}_{q^n}$ may still be applied. That requires around n multiplications in $\mathbb{F}_{q^n}$.

There could be a heuristic selector of the fastest method in the code, based on the input parameters n, q, k.

4.3 Evaluation of $\Gamma_k(\xi(Z))$ via polynomial modular composition

Assume we have precomputed $\Gamma_k(Z)$ for all k, then $\Gamma_k(\xi(Z)) = \gamma_k(\xi)(Z) \circ \Gamma_k(Z)$ is the polynomial composition applied after $\gamma_k(\xi)$ is computed.

Option 1. I.e., let $\gamma_k(\xi)(Z) = a_{r-1}Z^{r-1} + \ldots + a_1 \cdot Z + a_0$ then the evaluation can be done with Horner's method as

$$\gamma_k(\xi)(Z) \circ \Gamma_k(Z) = (\ldots((a_{r-1} \cdot \Gamma_k(Z) + a_{r-2}) \cdot \Gamma_k(Z) + a_{r-3}) \ldots) \cdot \Gamma_k(Z) + a_0,$$

which requires r polynomial multiplications modulo $R(Z)$.

Option 2. There is a more efficient method for fields with characteristic $q = 2$, see [ERS11]. Combine even and odd coefficients of $\gamma_k(\xi)$ as follows:

$$A_{\mathtt{even}}(Z) = a_0 + a_2 \cdot Z + a_4 \cdot Z^2 + \ldots$$
$$A_{\mathtt{odd}}(Z) = a_1 + a_3 \cdot Z + a_5 \cdot Z^2 + \ldots$$

then the evaluation becomes

$$\gamma_k(\xi)(\Gamma_k(Z)) = A_{\mathtt{even}}(\Gamma_k(Z)^2) + \Gamma_k(Z) \cdot A_{\mathtt{odd}}(\Gamma_k(Z)^2),$$

where both $A_{\mathtt{even}}$ and $A_{\mathtt{odd}}$ reuse the same powers of the input $\Gamma_k(Z)^i$, and thus the evaluation of both can be done with $\approx (1 + r/2)$ modular polynomial multiplications $R(Z)$, plus 1 squaring.

Continuing this way the evaluation is then can be done with $O(\sqrt{r})$ modular polynomial multiplications [ERS11].

4.4 Evaluation of $\Gamma_k(\xi(Z))$ via reduction matrix multiplication

For this method to work we need a number of precomputed matrices M_k of dimension $r \times r$ with elements from the base field $\mathbb{F}_{q^n}$ of the following form:

$$M_k[i] = \Gamma_k(Z)^i \mod R(Z).$$

I.e., the ith row is the reduced polynomial of $\Gamma_k(Z)^i \mod R(Z)$. Let us call a matrix M_k for *Reduction Matrix.* This way, the evaluation of $\Gamma_k(\xi(Z))$ is a vector-by-matrix multiplication as follows:

$$\Gamma_k(\xi(Z)) = [\gamma_k(\xi)] \times M_k,$$

where $[\gamma_k(\xi)]$ is the vector of r coefficients from $\gamma_k(\xi)$.

4.5 Precomputation of $\Gamma_k(Z)$

The precomputation of $\Gamma_k(Z)$ can be done sequentially by any of the above evaluation methods as follows

$$\Gamma_k(Z) = \Gamma_{k-1}(\Gamma_{k-1}(Z)),$$

i.e., when $\Gamma_{k-1}(Z)$ is already computed then $\Gamma_k(Z)$ can be derived from Γ_{k-1} by using any of the evaluation methods described above, and we start with $\Gamma_0(Z) = Z^q$.

4.6 Precomputation of reduction matrices M_k

In case we choose the second evaluation method, we then need to precompute the matrices M_k. For the sake of notation, for a vector V and a matrix M whose elements are from the base field $\mathbb{F}_{q^n}$, we define the notation:

$$V^{\langle t \rangle} \text{ and } M^{\langle t \rangle},$$

meaning that every element of V and M is raised to the power t individually. Thus, the following may be seen as an almost equivalent notation

$$\gamma_k(V) \Rightarrow V^{\left\langle q^{2^k} \right\rangle},$$

and similar for $\gamma_k(M)$, with only the difference that γ_k is actually a polynomial with coefficients from the vector $V^{\left\langle q^{2^k} \right\rangle}$.

4.6.1 Method A: Extracting from the sequence of Z^i

The ith row of M_k corresponds to the coefficients of the polynomial

$$Z^{i \cdot q^{2^k}} \mod R(Z).$$

We start to pre-compute the matrix M_0. It can be done quite easily by producing the following sequence of polynomials:

$$\begin{aligned}
&P_0(Z) = 1 \\
&P_1(Z) = Z \cdot P_0(Z) \mod R(Z) \\
&P_2(Z) = Z \cdot P_1(Z) \mod R(Z) \\
&P_3(Z) = Z \cdot P_2(Z) \mod R(Z) \\
&\quad ...
\end{aligned}$$

Note that $P_j(Z) = Z \cdot P_{j-1}(Z)$ is basically a shift by 1 element, and xor with $R(Z)$ multiplied by a base field element. Note also that each P_j is a vector of r elements from the base field. The rows of the first matrix M_0 are then every qth P, i.e.,

$$M_0 = [P_0, P_q, P_{2q}, ..., P_{(r-1)q}]^T.$$

This way, if we want to raise ξ to the power of q then we simply raise each coefficient of ξ to the power q, then multiply the resulting vector with the reduction matrix M_0.

The construction of M_0 relies on the construnction of qr rows of Ps, where in each step we make r multiplications in the base field, except for the first r rows as they are trivial. The complexity to precompute the first M_0, when the base field is binary $q = 2$, is then $r(r-1)$ base field multiplications.

Construction of the next M_k. The construction of the next matrix M_1 can be done in a similar way as for M_0, but just we take every q^2th vector of P. In general the construction of M_k is then to pick every q^{2^k}th vector of P.

It is easy to see that when k grows, then the distance between two Ps that will constitute two consecutive rows of M_k becomes very large, and, in fact, grows exponentially. However, there is the second method to construct M_k from the previous M_{k-1}.

4.6.2 Method B: Deriving M_k from M_{k-1} by a matrix multiplication

Recall that M_{k-1} holds the reduced value of $Z^{i \cdot q^{2^{k-1}}}$, for all $0 \leq i < r$. Then the ith row of M_k can be computed as:

$$\begin{aligned} M_k[i] &= \left(Z^i \cdot M_{k-1}\right)^{\left\langle q^{2^{k-1}} \right\rangle} \cdot M_{k-1} && (3) \\ &= M_{k-1}[i]^{\left\langle q^{2^{k-1}} \right\rangle} \cdot M_{k-1}. && (4) \end{aligned}$$

I.e., we take the ith row of M_{k-1} which corresponds to $Z^{i \cdot q^{2^{k-1}}}$, then we raise each coefficient to the power $q^{2^{k-1}}$ and apply M_{k-1} to get the final reduction of $(Z^{i \cdot q^{2^{k-1}}})^{q^{2^{k-1}}} = Z^{i \cdot q^{2^k}}$, which is then the ith row of M_k. Therefore, the full matrix M_k can be computed from M_{k-1} through the following expression:

$$M_k = M_{k-1}^{\left\langle q^{2^{k-1}} \right\rangle} \cdot M_{k-1},$$

where the matrix multiplication can be done with around $O(r^{2.38})$ multiplications (see, e.g., [CW90]) in the base field $\mathbb{F}_{q^n}$.

4.6.3 Method C: Mixed techniques

The above Method B may be performed by a mixed technique, in order to reduce the complexity of a matrix multiplication. The assumption is that the matrix M_{k-1} is already precomputed and we want to construct the next matrix M_k.

Idea 1. The row $M_k[0]$ is the constant polynomial and it is equal to 1.

Idea 2. The row $M_k[i]$ can be copied from $M_{k-1}[i \cdot q^{2^{k-1}}]$ for all i such that $i \cdot q^{2^{k-1}} < r$. This is due to the fact that

$$M_k[i] = \left(Z^{q^{2^k}}\right)^i = \left(Z^{q^{2^{k-1}}}\right)^{i \cdot q^{2^{k-1}}} = M_{k-1}[i \cdot q^{2^{k-1}}].$$

Idea 3. Rows divisible by q, i.e. $M_k[qi]$ can be computed via the reduction matrix M_0 due to:

$$M_k[qi] = M_k[i]^q \mod R(Z) = M_k[i]^{\left\langle q^{2^0} \right\rangle} \cdot M_0.$$

E.g., in the field of characteristic q powering the polynomial into q can be done efficiently point-wise, while the reduction matrix M_0 is then applied.

Idea 4a. Other rows may be computed by a polynomial modular multiplication as

$$M_k[i] = M_k[i-1](Z) \cdot M_k[1](Z) \mod R(Z).$$

Idea 4b. ...or by a vector-by-matrix multiplication as

$$M_k[i] = M_{k-1}[i]^{\left\langle q^{2^{k-1}} \right\rangle} \cdot M_{k-1}.$$

Idea 4c. ...or by the modular polynomial convolution as

$$M_k[i] = \gamma_{k-1}(M_{k-1}[i])(Z) \circ M_{k-1}[1](Z) \mod R(Z).$$

4.6.4 Heuristic switch between the methods

We start precomputing the power matrices with Method A. In the beginning of Method A we analyze if this method is still faster than the other ones, and when the heuristic finds that the other Method becomes faster then it performs a switch to that other Method, and all further power matrices are then computed with Method B or C.

4.7 Not all M_k ($\Gamma_k(Z)$) have to be computed

It could be worth to consider the case when we precompute not all matrices (or $\Gamma_k(Z)$, resp), but a little bit less, for example for only $k = 0 \ldots \lfloor \log_2(n) \rfloor - 2$. One possible reason could be that precomputing the matrix M_k takes a little more time than evaluation, and it may be worth skipping 1 or 2 last matrices and perform a bit more of evaluation steps, instead. The optimal number of matrices to skip depends on many parameters and the platform capabilities for efficient implementation techniques, but it may be estimated by simulations.

In this case, the evaluation of $\Gamma_k(\xi)$ given the maximum available function $\Gamma_{s<k}(\cdot)$ with precomputed tables up to s can be done by applying $\Gamma_s(\cdot)$ sequentially 2^{k-s} times as follows:

$$\Gamma_k(\xi) = \underbrace{\Gamma_s(\ldots \Gamma_s(\xi) \ldots)}_{2^{k-s} \text{ times}}.$$

5 Benchmarks and Conclusions

We have implemented all the tricks presented in this paper, and tested various HFEv-based signature schemes on a user-grade laptop with Intel Core i7-8650U CPU @ 1.90GHz with Turbo Boost up to 4.20GHz, utilizing SIMD instructions (e.g., PCLMULQDQ and others). Due to a high misalignment in the way how various scientific papers derive the metric "cycles per operation", we decided to present the results in terms of "operations per second", where possible. Although that metric is not the best and depends on the evaluation platform (e.g., CPU), we believe it still gives some indication of what can be achieved in terms of the signing speed in this class of PQC algorithms.

In the original paper on Quartz [PCG01] the authors mentioned that the time to sign a single message takes around 30 seconds, while tested on Pentium III 500 MHz. The performance was later improved to 1 signature per ∼10 seconds [CCC+09] using SIMD instructions. In our implementation with all the tricks described in this paper the signing speed reaches 30 signatures per 1 second.

Also, it is required in Quartz that the central equation $F_v(Y) = X$ should have a unique root. However, if we allow multiple roots, then the number of roots factorizations per a signature decreases; as the penalty, the "equal-degree" factorization step is then required to be implemented. Our experiments show that allowing multiple roots actually makes the signing process faster overall. The performance comparison table for Quartz is given in Table 1.

Table 1: Performance for Quartz

Profile	Secur.	Parameters as in [PCY+15]						Sign	PubKey	Roots	Signs	Verif.
	level						unique	size	size	factoriz.	per	per
	(bits)	n	D	K	v	r	root	(bits)	(Kbits)	per sign.	second	second
Original paper [PCG01] (Pentium III 500MHz)												
Quartz	80	103	129	4	4	3	Yes	128	578	–	0.03	8334
Our implementations (Intel Core i7-8650U @1.9/4.2GHz)												
Quartz	80	103	129	4	4	3	Yes	128	577	10.9	30.7	42276
Quartz-r	80	103	129	4	4	3	No	128	577	6.4	51.6	42706

In [PCY+15] the authors proposed a HFEv-based signature scheme Gui, where the degree D is very small, $D \in \{5, 9, 17\}$, while the other parameters like n, v, r, can be increased in order to reach the security level. This appeared to be crucial for the speed of HFEv-based schemes, since the complexity to extract a root is $O(D^3 + n \cdot D^2)$. There were four versions of Gui proposed with the requirement for a unique root. In the mentioned paper, only cycles/signature are given, and it is not easy to convert that into real signing time. However, the authors state that the speed of Gui is ∼100-150 times faster than Quartz, from which we can assume the performance of Gui is around 15 signatures per second, given the speed of Quartz being 1 signature per 10 seconds, implemented with SIMD as mentioned above.

In our experiments we give our performance numbers for Gui, as well as we give the results when multiple roots are allowed. The results are given in Table 2.

Table 2: Performance for original versions of Gui

Profile	Secur.	Parameters as in [PCY+15]						Sign	PubKey	Roots	Signs	Verif.
	level						unique	size	size	factoriz.	per	per
	(bits)	n	D	K	v	r	root	(bits)	(Kbits)	per sign.	second	second
Our implementations (Intel Core i7-8650U @1.9/4.2GHz)												
Gui-94	80	94	17	4	4	4	Yes	122	436	10.9	1227.2	56672
Gui-95	80	95	9	3	5	5	Yes	120	454	8.2	7528.0	70178
Gui-96	80	96	5	3	6	6	Yes	126	472	8.0	25725.2	66113
Gui-127	123	127	9	4	6	4	Yes	163	1096	10.9	5312.8	36551
Gui-94r	80	94	17	4	4	4	No	122	436	6.3	2114.3	57599
Gui-95r	80	95	9	3	5	5	No	120	454	4.7	12365.6	70561
Gui-96r	80	96	5	3	6	6	No	126	472	4.7	35941.1	64471
Gui-127r	123	127	9	4	6	4	No	163	1096	6.3	8431.4	36883

Versions of Gui submitted to NIST [PCYD18] also perform very slow, as we can see it from Table 3; in our implementations the speed is around 2 times faster, even though the reference speed was measured on a server-class Intel Xeon CPU. Moreover, if we allow multiple roots, then we can achieve ∼3-20 times speed up.

Table 3: Performance for versions of Gui as submitted to NIST

Profile	Secur.	Parameters as in [PCY+15]						Sign	PubKey	Roots	Signs	Verif.
	level						unique	size	size	factoriz.	per	per
	(bits)	n	D	K	v	r	root	(bits)	(Kbits)	per sign.	second	second
Intel Xeon E3-1225 v5 (Skylake) @3.3GHz with `PCLMULQDQ` [PCYD18]												
Gui-184	128	184	33	2	16	16	Yes	232	3376	—	96.2	19607
Gui-312	192	312	129	2	20	24	Yes	376	15920	—	1.88	5524
Gui-448	256	448	513	2	28	32	Yes	536	47227	—	0.038	976
Our implementations (Intel Core i7-8650U @1.9/4.2GHz)												
Gui-184	128	184	33	2	16	16	Yes	232	3376	5.4	163.0	30849
Gui-312	192	312	129	2	20	24	Yes	376	15920	5.5	5.7	6030
Gui-448	256	448	513	2	28	32	Yes	536	47227	6.6	0.4	2273
Gui-184r	128	184	33	2	16	16	No	232	3376	3.2	262.0	32495
Gui-312r	192	312	129	2	20	24	No	376	15920	3.0	10.0	6066
Gui-448r	256	448	513	2	28	32	No	536	47227	3.3	0.8	2329

Conclusions. Overall, it now becomes clear that the way to speed up an HFEv-based signature scheme is to use low values for D and K, and to allow multiple roots in the central equation $F_v(Y) = X$. We tried to collect a number of alternative parameters for Gui that we think might give us the intended security level whilst having a superior signing performance. The results are given in Table 4.

In all our implementations we used Montgomery representation for elements in the base field $\mathbb{F}_{2^n}$. However, when n becomes larger, it is then worth considering "optimal primitive polynomials" given in [MS17] for which modular arithmetics become faster than

Table 4: Performance for alternative versions of Gui with $D = 5$ and multiple roots allowed

Profile	Secur.	Parameters as in [PCY+15]						Sign	PubKey	Roots	Signs	Verif.
	level						unique	size	size	factoriz.	per	per
	(bits)	n	D	K	v	r	root	(bits)	(Kbits)	per sign.	second	second
Our implementations (Intel Core i7-8650U @1.9/4.2GHz)												
Gui80A	80	112	5	4	6	2	No	142	772	6.3	25778.9	44629
Gui80B	80	120	5	3	6	2	No	142	944	4.7	30333.1	53124
Gui112A	112	160	5	3	8	6	No	196	2186	4.7	16954.3	38268
Gui112B	112	176	5	3	8	6	No	212	2893	4.7	14501.2	28385
Gui128A	128	184	5	4	14	2	No	246	3585	6.3	11134.8	22914
Gui128B	128	190	5	3	2	13	No	222	3279	4.7	13009.0	26303
Gui128C	128	224	5	3	2	13	No	256	5412	4.7	10642.4	17985
Gui192A	192	252	5	6	4	28	No	416	7368	9.4	4602.0	7138
Gui192B	192	288	5	3	16	12	No	360	12795	4.7	6329.8	10322
Gui192C	192	304	5	3	16	12	No	376	14997	4.6	5864.6	8989
Gui256A	256	352	5	5	16	26	No	536	22134	8.2	1556.1	3156
Gui256B	256	368	5	4	16	26	No	510	25280	6.3	1984.9	3891
Gui256C	256	380	5	3	4	38	No	468	25280	4.7	2467.6	5186

that in the Montgomery representation, even with the PCLMULQDQ instruction. As an example, the primitive polynomial $f(x) = x^{457} + x^{16} + 1$ $(n = 457)$ over $GF(2)$ has a very low reduction penalty and performs 4x times faster than the reduction in the Montgomery representation. Other interesting examples are $f(x) = x^{375} + x^{16} + 1$ and $f(x) = x^{511} + x^{15} + 1$, which are also very fast. For $n \geq 512$, the most interesting polynomials would be: $f(x) = x^{521} + x^{32} + 1$, $f(x) = x^{543} + x^{16} + 1$, $f(x) = x^{577} + x^{25} + 1$, $f(x) = x^{609} + x^{128} + 1$, $f(x) = x^{721} + x^{9} + 1$, $f(x) = x^{865} + x + 1$, etc. The last example gives around 8x speed up of the modular reduction vs. similar arithmetics in the Montgomery representation.

Acknowledgements

We thank our colleague, Ben Smeets, for valuable comments to this manuscript. We also thank Helena Sjöberg for providing the ranges of Gui parameters which were used to select alternative Gui instances in Table 4.

References

[Ber67] Elwyn R. Berlekamp. Factoring polynomials over finite fields. *The Bell System Technical Journal*, 46(8):1853–1859, Oct 1967.

[Ber70] Elwyn R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24:713–735, 1970.

[CCC+09] 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 Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 33–48, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. https://www.iacr.org/archive/ches2009/57470031/57470031.pdf.

[CW90] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, March 1990. http://dx.doi.org/10.1016/S0747-7171(08)80013-2.

[CZ81] David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587–592, April 1981.

[DPCY14] Jintai Ding, Albrecht Petzoldt, Ming-Shing Chen, and Bo-Yin Yang. Gui : Revisiting multivariate digital signature schemes based on HFEv-. 2014. https://pdfs.semanticscholar.org/8575/eb87e81ed01302ad6ce4a57d24c67080ede6.pdf.

[ERS11] Michele Elia, Joachim Rosenthal, and Davide Schipani. Polynomial evaluation over finite fields: new algorithms and complexity bounds. *CoRR*, abs/1102.4772, 2011. http://arxiv.org/abs/1102.4772.

[KS97] Erich Kaltofen and Victor Shoup. Fast polynomial factorization over high algebraic extensions of finite fields. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, ISSAC '97, pages 184–188, New York, NY, USA, 1997. ACM. http://www4.ncsu.edu/~kaltofen/bibliography/97/KaSh97.pdf.

[MS17] Alexander Maximov and Helena Sjöberg. On fast multiplication in binary finite fields and optimal primitive polynomials over GF(2). Cryptology ePrint Archive, Report 2017/889, 2017. https://eprint.iacr.org/2017/889.

[NIS20a] NIST. Post-quantum cryptography standardization, Visited 2020. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[NIS20b] NIST. Post-Quantum Cryptography Standardization, round 2 submissions, Visited 2020. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[PCG01] Jacques Patarin, Nicolas Courtois, and Louis Goubin. QUARTZ, 128-bit long digital signatures, 2001. http://www.goubin.fr/papers/rsa2001b.pdf.

[PCY$^+$15] Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, Chengdong Tao, and Jintai Ding. Design principles for HFEv- based multivariate signature schemes. In *Proceedings, Part I, of the 21st International Conference on Advances in Cryptology – ASIACRYPT 2015 - Volume 9452*, pages 311–334, New York, NY, USA, 2015. Springer-Verlag New York, Inc. https://www.iis.sinica.edu.tw/papers/byyang/19342-F.pdf.

[PCYD18] Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, and Jintai Ding. PPT slides for Gui at NIST, 2018. https://csrc.nist.gov/CSRC/media/Presentations/Gui/images-media/Gui-April2018.pdf.

[Ric09] Chelsea Richards. Algorithms for factoring square-free polynomials, 2009. http://www.cecm.sfu.ca/CAG/theses/chelsea.pdf.

[vzGP01] Joachim von zur Gathen and Daniel Panario. Factoring polynomials over finite fields: A survey. *Symbolic Computation*, 31:3–17, 2001. https://people.csail.mit.edu/dmoshkov/courses/codes/poly-factorization.pdf.

[vzGS92a] Joachim von zur Gathen and Victor Shoup. Computing Frobenius maps and factoring polynomials. *Computational Complexity*, 2:187–224, 1992. http://www.shoup.net/papers/frobenius.pdf.

[vzGS92b] Joachim von zur Gathen and Victor Shoup. Computing Frobenius maps and factoring polynomials. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 97–105, New York, NY, USA, 1992. ACM. http://doi.acm.org/10.1145/129712.129722.

[ZT15] Wenbin Zhang and Chik How Tan. MI-T-HFE, A new multivariate signature scheme. In *Proceedings of the 15th IMA International Conference on Cryptography and Coding - Volume 9496*, IMACC 2015, pages 43–56, New York, NY, USA, 2015. Springer-Verlag New York, Inc. https://doi.org/10.1007/978-3-319-27239-9_3.

A Complexities

Complications to derive the complexity. One of the difficulties to derive the computational complexity comes from different ways to perform modular reduction in $\mathbb{F}_{q^n}$. Indeed, if we perform the reduction in Montgomery space then we achieve one complexity with a certain number of multiplications; on the other hand, if we take a primitive generating polynomial of the form, e.g., $f(x) = x^n + x^c + x^b + x^a + 1$, then the reduction may be done in a number of shifts and xors, with zero multiplications. How to assign the complexity in the second case, in terms of which basic functions?

Assume the instruction `PCLMULQDQ` or equivalent is available, which can perform a 64×64 bit multiplication in $\mathbb{F}_{2^n}$. Which method is faster - to use `PCLMULQDQ` or shifts-and-xors for modular multiplication, and how to compare these complexities?

Another problem comes from the choice of data representation. For example, in fields of characteristic 2 we can represent elements of the field $\mathbb{F}_{2^n}$ as a vector of 64-bit limbs. More precisely as $1 + \lfloor n/64 \rfloor$ limbs. This way, a xor of $a(x)$ and $b(x)$ in $\mathbb{F}_{2^n}$ can be done in $1 + \lfloor n/64 \rfloor$ xors. Or, it may be performed as a single 512-bit xor by using CPU's intrinsic instructions, such as `_mm512_xor_si512()` in AVX-512.

Basic field arithmetics in $\mathbb{F}_{q^n}$. Let us assume that a single modular multiplication of two elements in $\mathbb{F}_{q^n}$ can be performed with complexity

$$C_{\texttt{mul}} = \lambda_{\texttt{mul}} \cdot n \log(n),$$

where $\lambda_{\texttt{mul}}$ is a constant that depends on field representation, data structures, usage of intrinsics, and efficient implementation. Raising an element to the power q can be done more efficiently than a usual multiplication, so the complexity is

$$C_{\texttt{pwrq}} = \lambda_{\texttt{pwrq}} \cdot n.$$

Basic polynomial arithmetics over $\mathbb{F}_{q^n}$. Here we again assume that $R(Z)$ is a monic polynomial over $\mathbb{F}_{q^n}$ of degree r, which is the modulus polynomial for further arithmetics.

Multiplication of $A(Z)$ and $B(Z)$ each of degree at most $r-1$ can be done in $O(r \log(r))$ multiplications in the base field. The following reduction needs $(r-1)$ steps, each consisting of r multiplications in the base field. Thus, the complexity of modular polynomial multiplication can be derived as

$$C_{\texttt{pol.mul}} = (r \log(r) + r(r-1)) \cdot C_{\texttt{mul}}.$$

Raising a polynomial to the power of q and taking the result by modulo $R(Z)$ can be done with the complexity

$$C_{\texttt{pol.pwrq}} = (r + r(r-1)) \cdot C_{\texttt{sqrq}} = r^2 \cdot C_{\texttt{pwrq}}.$$

Evaluation of $\gamma_k(\xi)$ over $\mathbb{F}_{q^n}$. Here we have several options.

Option 1 with the sequence of raising each element of ξ to the power q requires

$$C_{\texttt{gamma.opt1}}(k) = r2^k \cdot C_{\texttt{pwrq}} = \lambda_{\texttt{pwrq}} \cdot rn2^k.$$

Option 2 with pre-computed lookup tables. Let's assume the tables will have q^8 entries (in case of characteristic 2 it means one byte) each of which then gives a partial sum of length n values in $\mathbb{F}_q$. Then we get

$$C_{\texttt{gamma.opt2}} = r\lfloor n/8 \rfloor.$$

where the basic operation is a xor of elements of $\mathbb{F}_{q^n}$.

The combined complexity is then

$$C_\gamma(k) = \begin{cases} \lambda_{\texttt{pwrq}} \cdot rn2^k, & \text{if } \lambda_{\texttt{pwrq}} \cdot 2^k < 1/8 \\ r\lfloor n/8 \rfloor, & \text{otherwise} \end{cases}.$$

Evaluation of $\Gamma_k(\xi)$ via polynomial modular composition. We evaluated the algorithm using Horner's method and Option 2 from our previous sections.

Option 1 with Horner's method, requires $r-2$ polynomial multiplications modulo $R(Z)$, thus

$$C_{\Gamma.\texttt{eval.comp.opt1}} = (r-2) \cdot C_{\texttt{pol.mul}} = \lambda_{\texttt{mul}} \cdot r(r-2)(\log(r) + r - 1) \cdot n\log(n).$$

or, in $O()$ notation, it will be

$$O(r^3 \cdot n\log(n)).$$

Option 2 is faster for characteristic 2 as it gives the complexity

$$C_{\Gamma.\texttt{eval.comp.opt2}} = \sqrt{r}\ln(r) \cdot C_{\texttt{pol.mul}} = \lambda_{\texttt{mul}} \cdot r\sqrt{r}\ln(r) \cdot (\log(r) + r - 1) \cdot n\log(n),$$

which leads to

$$O(r^2\sqrt{r}\log(r) \cdot n\log(n)).$$

Evaluation of $\Gamma_k(\xi)$ via reduction matrices. With a given reduction matrix M_k the evaluation of Γ can be done as a vector-by-matrix multiplication, excluding the rows with indexes i such that $iq^{2^k} < r$. For simplicity we will only exclude the first row as it is always 1.

$$C_{\Gamma.\texttt{eval.redmatr}} = r(r-1) \cdot C_{\texttt{mul}} = \lambda_{\texttt{mul}} \cdot r(r-1)n\log(n).$$

So, in $O(\cdot)$ notation, we get

$$O(r^2 \cdot n\log(n)),$$

which seem to be faster than using the polynomial composition technique.

Complexity of the algorithms `PowerOnly` and `PartialTrace`. Let us now have a closer look at the computational complexity of the discussed algorithms. The combined complexity of applying $\Gamma_k(\xi)$ is then

$$C_\Gamma = C_\gamma + C_{\Gamma.\texttt{eval.}} \approx r\lfloor n/8 \rfloor + \lambda_{\texttt{mul}} \cdot r(r-1) \cdot n\log(n).$$

The complexity of the algorithms `PowerOnly` and `PartialTrace` can be derived as

$$C_{\texttt{alg.power.only}}(t) = H(t) \cdot C_\Gamma,$$

and

$$C_{\texttt{alg.part.trace}}(m) = (\lfloor \log_2(m) \rfloor - 1 + H(m)) \cdot C_\Gamma,$$

where $H(\cdot)$ is the Hamming weight of the input integer.

Classical way to compute $\xi(Z)^{q^n}$, for comparison purpose. Given a polynomial $\xi(Z)$ we would need to raise it to the power q n times. Obviously, the computation of the trace requires a similar task, with the difference that in the latter we also sum all intermediate results. The complexity is the following:

$$C_{\texttt{pol.std.pwrq.n}} = n \cdot C_{\texttt{pol.sqrq}} = nr^2 \cdot C_{\texttt{pwrq}} = \lambda_{\texttt{pwrq}} \cdot (rn)^2.$$

So, the algorithms `PowerOnly` and `PartialTrace` have complexity

$$O(r^2 \cdot n \log^2(n)).$$

while the classical method has complexity

$$O(r^2 \cdot n^2)$$

However, the speed up is even larger if we consider also complexity constants, implementation tricks, etc. If we, otherwise, use the technique with polynomial composition then in $O()$ notation we get the complexity

$$(\lfloor \log_2(n) \rfloor - 1 + H(n)) \cdot (r\lfloor n/8 \rfloor + \sqrt{r}\ln(r) \cdot C_{\texttt{pol.mul}}) \to O(r^2\sqrt{r}\log(r) \cdot n\log^2(n)),$$

which seems to be worse than the complexity when we use reduction matrices. For getting the complete picture, we should also calculate the complexity to precompute the matrices M_k. This we do in the following subsection.

Precomputation of $\Gamma_k(Z)$. When we do a precomputation of $\log(n)$ polynomials of $\Gamma_k(Z)$ for $k = 0..\lfloor \log(n) \rfloor$ by sequential evaluation as $\Gamma_k(Z) = \Gamma_{k-1}(\Gamma_{k-1}(Z))$. Then the complexity for all precomputation will be

$$C_{Z.\texttt{prec.eval}} = \log(n) \cdot (C_\gamma + C_{\Gamma.\texttt{eval.comp.opt2}}),$$

which results in

$$O(r^2\sqrt{r}\log(r) \cdot n\log^2(n)).$$

Precomputation of reduction matrices M_k.

Method A. Computation of the next $P_j(Z)$ from $P_{j-1}(Z)$ takes in general $rC_{\texttt{mul}}$ steps, with the exception of the first $i = \lfloor \log_q(r) \rfloor$ steps for which $q^{2^i} < r$ – those Ps are just copied from $P_1(Z) = Z$ since $P_i(Z) = Z^{q^{2^i}}$ has degree $< r$. Therefore, to compute all matrices $M_0, M_1, \ldots, M_k$, the complexity is then

$$C_{M.\texttt{all.methA}}(k) = (rq^{2^k} - \lfloor \log_q(r) \rfloor) \cdot rC_{\texttt{mul}}.$$

The complexity to compute the next matrix M_k is

$$\begin{aligned} C_{M.\texttt{one.methA}}(k) &= C_{M.\texttt{all.methA}}(k) - C_{M.\texttt{all.methA}}(k-1) \\ &= r^2(q^{2^k} - q^{2^{k-1}}) \cdot C_{\texttt{mul}} = \lambda_{\texttt{mul}} \cdot r^2(q^{2^k} - q^{2^{k-1}}) \cdot n\log(n). \end{aligned}$$

Method B. In the second method we compute $M_k = M_{k-1}^{\left\langle q^{2^{k-1}} \right\rangle} \cdot M_{k-1}$. That can be done with complexity (for matrix multiplication we can use the complexity, e.g., from [CW90])

$$C_{M.\texttt{one.methB}}(k) = rC_\gamma(k) + r^{2.38}C_{\texttt{mul}} = r^2\lfloor n/8 \rfloor + r^{2.38}\lambda_{\texttt{mul}} n\log(n).$$

Method C. The first $s = \lceil r/q^{2^{k-1}} \rceil$ rows can be copied from the previous matrix M_{k-1} (Ideas 1 and 2). The other $\lfloor \log_q(r-s) \rfloor$ rows can be computed by using M_0 (Idea

3) with the complexity $rC_{\mathtt{pwrq}} + r^2 C_{\mathtt{mul}}$. The remaining rows may be computed by a usual evaluation method. We can upper bound the complexity as follows:

$$C_{M.\mathtt{one.methC}}(k) \le r^2 \cdot (r - \lceil r/q^{2^{k-1}} \rceil) \cdot (\lfloor n/8 \rfloor + \lambda_{\mathtt{mul}} n \log(n)),$$

which is then

$$O(r^3 n \log(n)).$$

Combining everything. To compute all M_k's would require $O(r^{2.38} n \log^2(n))$ time in the worst case, which has the term $r^{2.38}$ larger than in the classical complexity r^2. However, the real implementation with all the explained tricks gives a huge speed up even for large degree r of input polynomials. It would be more fair to introduce the coefficient $\lambda_M < 1$ in the complexity that will reflect other tricks while computing M_ks. I.e., for a fairly small r or for r much smaller than n we get a good speed up in computations.

Finally, we switch from the method A to B or C heuristically, when complexity becomes smaller in the other method.

Overall conclusion on complexity. The standard algorithm to compute a trace would cost about $O(r^2 n^2)$ of time. The proposed implementation would cost in total

$$O(r^2 \sqrt{r} \log(r) \cdot n \log^2(n)),$$

in case we use polynomial composition for both the precomputation phase and evaluation of $\Gamma_k(\xi)$; and the complexity is

$$O((\underbrace{\lambda_M r^{2.38}}_{\mathtt{precompute}} + \underbrace{r^2}_{\mathtt{evaluate}}) \cdot n \log^2(n)),$$

if we precompute reduction matrices M_k; in practice, and in the context of this paper, the term $r^{2.38}$ has a low constant $\lambda_M < 1$.