Implementing 128-bit Secure MPKC Signatures

Ming-Shing Chen[1], Wen-Ding Li[2], Bo-Yuan Peng[2], Bo-Yin Yang[2], and Chen-Mou Cheng[1]

[1]Department of Electrical Engineering, National Taiwan University, Taiwan , mschen@crypto.tw, doug@crypto.tw
[2]Institute of Information Science, Academia Sinica, Taiwan , kvlxu3@gmail.com, bypeng@crypto.tw, by@crypto.tw

June 28, 2017

Abstract

Multivariate Public Key Cryptosystems (MPKCs) are often touted as future-proofing against Quantum Computers. In 2009, it was shown that hardware advances do not favor just "traditional" alternatives such as ECC and RSA, but also makes MPKCs faster and keeps them competitive at 80-bit security when properly implemented. These techniques became outdated due to emergence of new instruction sets and higher requirements on security.

In this paper, we review how MPKC signatures changes from 2009 including new parameters (from a newer security level at 128-bit), crypto-safe implementations, and the impact of new AVX2and AESNI instructions. We also present new techniques on evaluating multivariate polynomials, multiplications of large finite fields by additive Fast Fourier Transforms, and constant time linear solvers.

1 Introduction

1.1 The Requirements on Post-Quantum Security

Since Shor's algorithm [Sho97] was invented, it is clear that traditional public key cryptography(PKCs) based on discrete logarithm and RSA assumptions are going to be solved in polynomial time once large quantum computers are built. PKCs that retain sufficient security levels when quantum computers have arrived are said to be post-quantum. Such cryptosystems are also sometimes called Postquantum Cryptosystems or PQCs. There are four or five main classes of PQCs one of which comprise Multivariate public-key cryptosystems (MPKCs) [CJL+16].

1.2 MPKCs and its Security

MPKCs are PKCs whose public keys represent multivariate polynomials over a small finite field(GF) $\mathbb{K} = \mathbb{F}_q$:

$$\mathcal{P} : \mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{K}^n \mapsto \mathbf{z} = (p_1(\mathbf{w}), \ldots, p_m(\mathbf{w})) \in \mathbb{K}^m.$$

Polynomials $p_1, p_2, \ldots$ have (almost always) been quadratic. In public-key cryptography, we can let $\mathcal{P}(\mathbf{0}) = \mathbf{0}$.

We need to discuss the security of MPKCs in order to set the parameters needed for the required security level(s). Public key of MPKCs are instances of solving multivariate quadratic equations, or instances. One can break all MPKCs if one is able to efficiently solve $\mathcal{MQ}$ problems.

1.2.1 Class $\mathcal{MQ}(q, n, m)$ and the $\mathcal{MQ}$ Problem

For given q, n, m, the class $\mathcal{MQ}(q, n, m)$ consists of all systems of m quadratic polynomials in $\mathbb{F}_q$ with n variables. To choose a random system $\mathbf{S}$ from $\mathcal{MQ}(q, n, m)$, we write each polynomial $P_k(\mathbf{x})$ as $\sum_{1 \le i \le j \le n} a_{ijk} x_i x_j + \sum_{1 \le i \le n} b_{ik} x_i + c_k$, where every a_{ijk}, b_{ik}, c_k is chosen uniformly in $\mathbb{F}_q$.

Solving $\mathbf{S}(\mathbf{x}) = \mathbf{b}$ for any $\mathcal{MQ}$ system S is then known as the "multivariate quadratic" problem. It is an NP-complete problem [GJ79]. However, it is not easy to base a proof on worst-case hardness. Often the premise used is the hereto unchallenged average-case $\mathcal{MQ}$ hardness assumption [BGP06, LLY08]:

Assumption $\mathcal{MQ}$

Given any k and prime power q, for parameters n, m satisfying $m/n = c + o(1)$, no probabilistic algorithm in subexponential(n)-time can solve $\mathbf{S}(\mathbf{x}) = \mathbf{b}$ with a non-neglible probability $\varepsilon > 0$, if the systems $\mathbf{S}$ are drawn from $\mathcal{MQ}(q, n, m)$, and a vector $\mathbf{b} = (b_1, b_2, \ldots, b_m)$ drawn from $\mathbf{S}(U_n)$, where U_n is the uniform distribution over $(\mathbb{F}_q)^n$.

1.2.2 Hardness of generic $\mathcal{MQ}$

The complexity of solving a random instance out of $\mathcal{MQ}(n, m, q)$ is estimated using Gröbner basis methods, often XL with sparse matrices [CKPS00,YCBC07] or F5 [Fau02, BFSY05]. We simply use prior estimates for complexity of solving $\mathcal{MQ}$.

1.2.3 Effect of Quantum Computers on $\mathcal{MQ}$ signatures

Since we discuss MPKC as post-quantum, we must consider a direct quantum computer attack using Grover's algorithm [Gro96], which is considered in [WS16]. The summary of this attack is that a system of $\mathcal{MQ}$ equations with n-bits of inputs can be solved in $2^{\frac{n}{2}+1} n^3$ quantum operating steps ("gates").

Note that this is not usually a problem because a signature scheme usually requires $2b$-bit wide hashes for b-bit security, so usually a 128-bit secure digital signature scheme has 256 bits of input anyway. If we assume that a quantum step ("gate") can run at the speed as a CPU cycle (a very very aggressive assumption about quantum computers), solving a quadratic system with 210 bits of input and output takes an equivalent of 2^{128} cycles.

1.2.4 Extended Isomorphism of Polynomials (EIP)

Notice MPKCs cannot be random $\mathcal{MQ}$ polynomials, because the legitimate user would be equally unable to invert $\mathcal{P}$. Usually the public map of an MPKC have structure in the "bipolar form": $\mathcal{P} = T \circ \mathcal{Q} \circ S$ where T and S are affine,

$$\mathcal{P} : \mathbf{w} \in \mathbb{K}^n \overset{S}{\mapsto} \mathrm{M}_S \mathbf{w} + \mathbf{c}_S := \mathbf{x} \overset{\mathcal{Q}}{\mapsto} \mathbf{y} \overset{T}{\mapsto} \mathrm{M}_T \mathbf{y} + \mathbf{c}_T := \mathbf{z} \in \mathbb{K}^m.$$

The field $\mathbb{K} = \mathbb{F}_q$ is often called the base field. The requirement for the quadratic *central map* $\mathcal{Q}$ is that it is easy to "invert" $\mathcal{Q}$ but not $\mathcal{P}$. That is, given $y \in \mathbb{K}^m$, it is easy to compute x such that $\mathcal{Q}(x) = y$. but find a x such that $\mathcal{P}(x) = y$ is hard. The structure is hidden away by S and T. Given this, the MPKC may be attacked via what is called structural attacks.

EIP and "Structural Attacks" Given a class $\mathcal{C}$ of quadratic maps $\mathbb{K}^n \mapsto \mathbb{K}^m$ and a quadratic map $\mathcal{P} : \mathbb{K}^n \mapsto \mathbb{K}^m$, an associated EIP instance means to find S and T such that $\mathcal{P} = T \circ \mathcal{Q} \circ S$, where $\mathcal{Q} \in \mathcal{C}$. Defeating a bipolar-form MPKC through solving an EIP is known as a "structural" or Key-Recovery attack.

Note that solving an EIP problem is very ad hoc, depending very much on what $\mathcal{Q}$ is like, and again we do not go into the theoretical details but uses known EIP results in this paper.

1.2.5 Non-bipolar $\mathcal{M}Q$ and Proofs of Knowledge

There are $\mathcal{M}Q$ public-key schemes which are based only on the security of hash functions and the $\mathcal{M}Q$ problem only, such as [CHR+16] (and the older [SSH11]). These are based on proofs of knowledge rather than the traditional MQ paradigm. The key steps of both the public and secret operations involves only repeated $\mathcal{M}Q$ evaluations. The cost is running time and the length of the signature (many kilobytes).

1.3 The Implementation of MPKCs

1.3.1 The challenge of Cryptographic Implementations

In practice, a security system can be broken due to its implementation instead of the cryptography, e.g., the cache-timing attack to AES [BM06]. We would like reasonable implementations which retain as much as possible side channel resilience. This means that the secret data should be independent of memory access and table indices. In other words, time constancy is always a basic requirment when processing secret data. We want such implementations for generic 32-bit architectures (many of today's micro-controllers) and for the diverse instruction set in mainstream CPUs.

MPKCs were usually advertised for speed, which still needs to be reviewed according to today's security requirements. In 2009 MPKCs were shown [CCC+09] to be easily a match for RSA and ECC at the 80-bit security level. It seems the basic security requirements has shifted to 128-bit, which can be seen from the call of new post-quantum cryptographic schemes from NIST [oST16]. We have to see whether MPKC signature schemes still remain viable in the age of 128-bit security.

1.3.2 Revisiting the Implementations of MPKCs

We can partition implementation of MPKCs into smaller components most of which are procedures in basic linear algebra. The efficiency of MPKCs usually relies on the implementations of these components:

- The evaluation of quadratic polynomials is a key component in implementing multivariate cryptography and had been studied in [BBG07, CCC$^+$09, CHR$^+$16]. In general, these works studied the most efficient instructions in the target platforms for evaluating using the minimum number of instructions.

- Arithmetic in finite fields is a basic topic in computer science and closely related to the implementation of MPKCs for fields up to 512 bits. For these large fields of characteristic 2, the polynomial-multiplication instruction(`PCLMULQDQ`) is a perfectly fit for the requirements and had been used as primary choice for building field multiplications, e.g., in [PCY$^+$15].

  To perform the multiplications for the platform without `PCLMULQDQ`, some implementations build a multiplication from vector instructions using Karatsuba or similar algorithms, e.g,., in [CCC$^+$09, PCY$^+$15]. In 2014, Bernstein and Chou [BC14] presented the multiplications by applying additive FFT [GM10] and bit-slicing when implementing for generic platforms without SIMD instruction sets. Motivated by [BC14], we present a multiplication for general SIMD platforms using a new additive FFT [LCH14] in this paper.

- The secret maps of some MPKCs are mainly root-finding of high degree univariate polynomials using Berlekamp's algorithm [PCY$^+$15]. Since the success of signing is dependent on the existence a root in some MPKCs, we can parallelize the signing process with different randomness for increasing the probability of a successful signing. We achieve the parallelization of big GF arithmetic using SIMD in this paper.

- Solving linear equations is also a key component in some MPKC schemes [DS05]. In 2014 [BC14] demonstrated a constant time Gauss eliminations for $\mathbb{F}_2$. We extend the method to $\mathbb{F}_{16}$ and $\mathbb{F}_{31}$ in this paper. The key is to remove branching on zero pivots and instead use conditional move.

Throughout this paper, we will revisit these key components of MPKCs while taking into consideration side-channel resilience and a 128-bit security level.

2 Backgrounds on MPKC Signatures

2.1 Recap of MPKC Signatures

An Multivariate Public Key Cryptosystem has a public map $\mathcal{P} = T \circ \mathcal{Q} \circ S$ where T and S are affine,

$$\mathcal{P} : \mathbf{w} \in \mathbb{K}^n \overset{S}{\mapsto} \mathrm{M}_S\mathbf{w} + \mathbf{c}_S := \mathbf{x} \overset{\mathcal{Q}}{\mapsto} \mathbf{y} \overset{T}{\mapsto} \mathrm{M}_T\mathbf{y} + \mathbf{c}_T := \mathbf{z} \in \mathbb{K}^m.$$

The field $\mathbb{K} = \mathbb{F}_q$ is often called the base field. The quadratic *central map* $\mathcal{Q}$ (but not $\mathcal{P}$) must by easy to "invert" . The structure of $\mathcal{Q}$ is hidden away by S

and T and the various MPKCs are characterized by the struction of their $\mathcal{Q}$'s. When evaluating the private map, the legitimate user inverts T, $\mathcal{Q}$, and S in that order.

It is almost universally accepted that it is difficult to design multivariate encryption schemes. Most such schemes are either already been broken or have much larger sizes than signature schemes. We enumerate the main MPKC signatures considered secure today and modify their parameters for 128-bit security in this section. We will discuss the implementation of these schemes in the later sections.

According to whether $\mathcal{Q}$ involves a mapping in a much larger field $\mathbb{L} \supset \mathbb{K}$, the scheme is called "big" or "small" field respectively. The size of big field $\mathbb{L}$ is usually somewhere from 2^{64} to 2^{512} and multiplications in $\mathbb{L}$ are usually the time consuming steps of the secret map in big field schemes.

2.2 Rainbow/TTS

Rainbow [DS05] is the stereotypical "small field" MPKC, where we work on the same $\mathbb{F}_{16}$, $\mathbb{F}_{31}$, or $\mathbb{F}_{256}$ throughout. Although TTS [DYC+08] had been proposed earlier, it can be considered as Rainbow with a sparse $\mathcal{Q}$ in today's terminology. The definitive analysis of security for Rainbow/TTS and the formulation of current instances can be found in the 2008 paper [DYC+08]. 80-bit secure parameters are chosen in [PBB10].

2.2.1 The central map in Rainbow/TTS

Rainbow($\mathbb{F}_q, v_1, o_1, \dots, o_u$) is characterized as follows as a u-stage UOV [DS05, DYC+08].

- The segment structure is given by a sequence $0 < v_1 < v_2 < \cdots < v_{u+1} = n$. For $l = 1, \dots, u+1$, set labels for "vinegar" variables as $V_l := \{1, 2, \dots, v_l\}$ so that $|V_l| = v_l$ and $V_0 \subset V_1 \subset \cdots \subset V_{u+1} = V$. Denote sets of "oil" variables by $o_l := v_{l+1} - v_l$ and $O_l := V_{l+1} \setminus V_l$ for $l = 1 \cdots u$.
- The central map $\mathcal{Q}$ comprises m structurized quadratic equations $\mathbf{y} = (y_{v_1+1}, \dots, y_n) = (q_{v_1+1}(\mathbf{x}), \dots, q_n(\mathbf{x}))$, where
$$y_k = q_k(\mathbf{x}) = \sum_{i=1}^{v_l} \sum_{j=i}^{v_{l+1}} \alpha_{ij}^{(k)} x_i x_j + \sum_{i<v_{l+1}} \beta_i^{(k)} x_i \ ,$$
for $k \in O_l := \{v_l + 1 \cdots v_{l+1}\}$.
- *Note that in every q_k, where $k \in O_l$, there is no cross-term $x_i x_j$ where both i and j are in O_l.* So given all the y_i with $v_l < i \le v_{l+1}$, and all the x_j with $j \le v_l$, we can easily compute $x_{v_l+1}, \dots, x_{v_{l+1}}$.

2.2.2 Signatures in Rainbow/TTS

To sign a message, the signer calculate the hash digest $\mathbf{z}$ of message and inverts $\mathcal{P}$ with the secret key T, S, and $\mathcal{Q}$ by
$$\mathbf{z} \in \mathbb{K}^m \overset{T^{-1}}{\longmapsto} \mathbf{y} \overset{\mathcal{Q}^{-1}}{\longmapsto} \mathbf{x} \overset{S^{-1}}{\longmapsto} \mathbf{w} \in \mathbb{K}^n \ ,$$

where $\mathbf{w}$ is the signature. The key step here is inverting the central map $\mathcal{Q}$. While inverting $\mathcal{Q}$ with given $\mathbf{y}$, the signer randomly guesses vinegar variables $\bar{\mathbf{x}} = (x_1, \dots x_{v_1})$ and solve $(x_{v_1+1}, \dots, x_{v_1+o_1})$ by

$$\begin{aligned} y_{v_1+1} &= \bar{\alpha}^{(v_1+1)}_{v_1+1} x_{v_1+1} + \cdots + \bar{\alpha}^{(v_1+1)}_{v_1+o_1} x_{v_1+o_1} + \bar{\beta}^{(v_1+1)}_{V_1} \\ &\vdots \\ y_{v_1+o_1} &= \bar{\alpha}^{(v_1+o_1)}_{v_1+1} x_{v_1+1} + \cdots + \bar{\alpha}^{(v_1+o_1)}_{v_1+o_1} x_{v_1+o_1} + \bar{\beta}^{(v_1+o_1)}_{V_1} \quad . \end{aligned} \tag{1}$$

Here $(\bar{\beta}^{(v_1+1)}_{V_1}, \dots, \bar{\beta}^{(v_1+o_1)}_{V_1})$ is an evaluation of secret-quadratic equations with secret values $\bar{\mathbf{x}}$ and the matrix

$$\begin{bmatrix} \bar{\alpha}^{(k)}_i & \cdots & \bar{\alpha}^{(k)}_{i'} \\ & \ddots & \\ \bar{\alpha}^{(k')}_i & & \bar{\alpha}^{(k')}_{i'} \end{bmatrix} , \text{ where } i, i' \text{ and } k, k' \in O_1 \ ,$$

denoted by `matVO`($\bar{\mathbf{x}}$), is evaluated as linear forms in $\bar{\mathbf{x}}$. All x_i where $i \in O_l$ is solved with a linear solver and there are total u linear systems to be solved. The signer may have to repeat the process if any `matVO`($\bar{\mathbf{x}}$) is a singular matrix. Hence, the main computation cost of signing is solving linear equations and computing the matrices `matVO`($\bar{\mathbf{x}}$) from vinegar variables $\bar{\mathbf{x}}$.

2.2.3 Parameters of Modern Rainbow/TTS

In current Rainbow/TTS, u is always 2, with parameters (v, o, o), and at b-bit security $q^o \gtrsim 2^b$ (rank attacks [YC05]). The number of variables and equations are $(n, m) = (v{+}2o, 2o)$. We require $2^b \lesssim \min(C_{FXL}(m, m), C_{FXL}(n, m{+}n{-}1))$ [DYC+08]. Ding et al. [DYC+08, CCC+08] suggest for 80-bit design security Rainbow/TTS with parameters $(\mathbb{F}_{2^4}, 24, 20, 20)$ and $(\mathbb{F}_{2^8}, 18, 12, 12)$. We modify the parameters for modern security requirements in Table 1.

Table 1: Parameters of Rainbow.

security	parameters	$\mathbb{F}_{16}$	$\mathbb{F}_{31}$	$\mathbb{F}_{256}$
128 bits	(v_1, o_1, o_2)	32,32,32	28,28,28	28,20,20
	$n \to m$	$96 \to 64$	$84 \to 56$	$68 \to 40$
192 bits	(v_1, o_1, o_2)	48,48,48	53,40,40	52,32,32
	$n \to m$	$144 \to 96$	$133 \to 80$	$116 \to 64$
256 bits	(v_1, o_1, o_2)	64,64,64	74,56,56	73,48,48
	$n \to m$	$192 \to 128$	$186 \to 112$	$169 \to 96$

2.3 pFLASH

C^*-p or pFLASH [DDY+08] was a 2008 prefix modification of the earlier SFLASH by Patarin [PCG01a].

2.3.1 The central map $\mathcal{Q}$ in pFLASH

pFLASH($\mathbb{K} = \mathbb{F}_q, n - \pi, m$) is a large field scheme. We identify $\mathbb{L}$, a degree-n extension of the base field $\mathbb{K} = \mathbb{F}_q$ with $(\mathbb{F}_q)^n$ via an implicit bijective map $\phi : (\mathbb{F}_q)^n \in \mathbb{K}^n \to \mathbb{F}_{q^n} \in \mathbb{L}$. Thus we consider $\mathbf{x} \in \mathbb{F}_q^n$. In this view, the central map $\mathcal{Q}$:

$$\mathbf{x} \in \mathbb{K}^n \overset{\phi}{\mapsto} X \in \mathbb{L} \xmapsto{\mathcal{Q}} Y = X^{q^\alpha+1} \xmapsto{\phi^{-1}} \mathbf{y} \ ,$$

which is quadratic in the components of $\mathbf{x}$, because $X \mapsto X^{q^\alpha}$ is linear in (the components of) $\mathbf{x}$. We need $\gcd(q^n - 1, q^\alpha + 1) = 1$, so there exists an h such that $h \cdot (q^\alpha + 1) = 1 + g \cdot (q^n - 1)$ and thus

$$\mathcal{Q}^{-1} : Y \mapsto Y^h = X^{1+g\cdot(q^n-1)} = X \ . \tag{2}$$

2.3.2 The signing process

For generating a signature, two modifications are designed in pFLASH for improving the security. The first special feature is that pFLASH is "prefixed" meaning the first π (almost always $= 1$) components of the input variables $\mathbf{w}$ are fixed to be zero. No coefficients relating to them are released with the public key because they are not needed. The other is a "minus" scheme where $a = n - m$ equations are not released. In other words, the user first computes $\mathcal{P}' = T' \circ \mathcal{Q} \circ S$ with invertible S and T, then remove all coefficients of the first variable (or more, if $\pi > 1$), and the last $n-m$ equations to find a $\mathcal{P}$ with $n-m$ equations in $n - \pi$ variables. The secret key still contains the entirety of S and T.

To sign, the user finds the (padded) hash $\mathbf{z}$, pad it randomly in the last $n-m$ positions to form $\mathbf{z}'$, invert T to get $\mathbf{y}$, compute $\mathbf{x} = \mathbf{y}^h$, then $\mathbf{w} = S^{-1}(\mathbf{x})$. This is considered to be a valid signature if and only if the initial component(s) are zero, otherwise we repeat the process with different randomness.

The computational cost of pFLASH is mostly in the part of raising Y with different randomness to a power $Y^h = X$. What this involve are repeated squarings, raising to a power of q, and multiplications in the big field. The last is by far the main computational bottleneck.

2.3.3 Parameters of pFLASH

We show the modified the parameters pFLASH from [CSTY15] in Tab. 2. The initial parameters are ($\mathbb{F}_{16}$,62 − 1,40) which is designed for lightweight devices at the 80-bit security in [CSTY15]. We include this parameter set in our implementations because it is still topical.

Table 2: Parameters of pFLASH.

security	80 bits	128 bits	256 bits
$(\mathbb{F}_q, n - \pi, m)$	$\mathbb{F}_{16}$,62-1,40	$\mathbb{F}_{16}$,96-1,64	$\mathbb{F}_{16}$,192-1,128
pub map: $n \to m$	$61 \to 40$	$95 \to 64$	$191 \to 128$

2.4 HFEv- and QUARTZ/GUI

HFE or Hidden Field Equations [Pat96] is also a "big-field" variant of MPKC. It was the most venerable of these schemes (having been proposed nearly 20 years ago in the last millennium) and in slightly modified form became QUARTZ and Gui [PCG01b, PCY+15].

2.4.1 The central map of HFEv-

As in other big-field schemes, we identify $\mathbb{L}$, a degree-ℓ extension of the base field $\mathbb{K}$ with $(\mathbb{F}_q)^\ell$ and a bijective map $\phi : (\mathbb{F}_q)^\ell \in \mathbb{K}^\ell \mapsto \mathbb{F}_{q^\ell} \in \mathbb{L}$. With a pre-defined degree d, the central map of HFE$(\mathbb{F}_{q^\ell}, d)$ is defined: $Y = \sum_{0\le i,j}^{q^i+q^j\le d} \alpha_{ij} X^{q^i+q^j} + \sum_{0\le i}^{q^i\le d} \beta_i X^{q^i} + \gamma$, which is quadratic in $\mathbf{x}$ and invertible via the Berlekamp algorithm in asymptotic complexity $O(d^{1.815}\log q^\ell)$ [KS98] with X and Y as elements of $\mathbb{F}_{q^\ell}$. Solving HFE via public equations directly is considered to be sub-exponential($O(\ell^{\log d})$ for $q = 2$, quasi-polynomial) [GJS06].

To increase the security, we may add v vinegar variables and define HFEv$(\mathbb{F}_{q^\ell}, d, v)$ as follows

$$\begin{aligned}\mathcal{Q}(X,\bar{\mathbf{x}}) := & \sum_{0\le i,j}^{q^i+q^j\le d} \alpha_{ij} X^{q^i+q^j} \\ & + \sum_{0\le i}^{q^i\le d} \beta_i(\bar{\mathbf{x}}) X^{q^i} + \gamma(\bar{\mathbf{x}})\ ,\end{aligned} \tag{3}$$

where $\bar{\mathbf{x}} = (x_{\ell+1}, \dots, x_{\ell+v}) \in \mathbb{F}_q^v$ are vinegar variables. There are extra injections from $(x_{\ell+1}, \dots, x_{\ell+v}) \in \mathbb{K}^v$ into $\mathbb{L}^v$. $\beta_i(\bar{\mathbf{x}}) \in \mathbb{L}$ and $\gamma(\bar{\mathbf{x}}) \in \mathbb{L}$ are linear and quadratic respectively in $\bar{\mathbf{x}}$ (and thus in $\mathbf{x}$).

Now we have a quadratic central map of $\mathbf{x} = (x_1, \dots, x_{\ell+v})$ to $\mathbf{y}$. This is efficiently invertible by guessing $(x_{\ell+1}, \dots, x_{\ell+v})$, substituting $\bar{\mathbf{x}}$, then solve the resulting equation for $\mathbf{x}$ by Berlekamp algorithm. Finally, we can add a minus variation just like in pFLASH, by releasing only $\ell - a$ of the equations. Now we have HFEv-$(\mathbb{F}_{q^\ell}, d, v, a)$ with $n = \ell + v$, $m = \ell - a$.

2.4.2 The Patarin variation and QUARTZ/GUI

QUARTZ is HFEv-$(\mathbb{F}_{2^{103}}, 129, 4, 3)$ with $(n, m) = (107, 100)$, yet QUARTZ is a 128-bit signature and uses SHA-1. The key is that in QUARTZ/GUI, the public map is used k times and the result chained together as in Alg. 1–2.

One final important detail about the Patarin variation. The central operation in the signing process of HFEv- is the Berlekamp algorithm, which about e^{-1} of the time returns "no solution". In QUARTZ/GUI when we take $\gcd(X^{q^\ell} - X, \mathcal{Q}(X, \bar{\mathbf{x}}))$ at the beginning and the result isn't degree one (exactly one solution), we forego the rest of the process and restart from picking new padding. In QUARTZ this opens the possibility of there being no solutions. Since $a + v$ in GUI is fairly large, the possibility of there being no solutions is negligible.

Algorithm 1 Signature Generation Process of Gui

```
Require: Gui private key ($\mathcal{S}$, $\mathcal{F}$, $\mathcal{T}$) message $\mathbf{d}$, repetition factor $k$
Ensure: signature $\sigma \in \mathbb{F}_2^{(\ell-a)+k(a+v)}$
 1: $\mathbf{h} \leftarrow$ SHA-256($\mathbf{d}$)
 2: $S_0 \leftarrow \mathbf{0} = 0^{\ell-a}$ ($S_i$ are $\ell - a$ bits, $X_i$ are $a + v$ bits).
 3: for $i = 1$ to $k$ do
 4:     $D_i \leftarrow$ first $\ell - a$ bits of $\mathbf{h}$
 5:     $(S_i, X_i) \leftarrow$ HFEv$-^{-1}(D_i \oplus S_{i-1})$
 6:     $\mathbf{h} \leftarrow$ SHA-256($\mathbf{h}$)
 7: end for
 8: $\sigma \leftarrow (S_k||X_k||\ldots||X_1)$
 9: return $\sigma$
```

Algorithm 2 Signature Verification Process of Gui

```
Require: Gui public key $\mathcal{P}$, message $\mathbf{d}$, repetition factor $k$, signature $\sigma \in$
    $\mathbb{F}_2^{(\ell-a)+k(a+v)}$
Ensure: TRUE or FALSE
 1: $\mathbf{h} \leftarrow$ SHA-256($\mathbf{d}$)
 2: $(S_k, X_k, \ldots, X_1) \leftarrow \sigma$ ($S_i$ are $\ell - a$ bits, $X_i$ $a + v$ bits).
 3: for $i = 1$ to $k$ do
 4:     $D_i \leftarrow$ first $\ell - a$ bits of $\mathbf{h}$
 5:     $\mathbf{h} \leftarrow$ SHA-256($\mathbf{h}$)
 6: end for
 7: for $i = k - 1$ to $0$ do
 8:     $S_i \leftarrow \mathcal{P}(S_{i+1}||X_{i+1}) \oplus D_{i+1}$
 9: end for
10: if $S_0 = \mathbf{0}$ then
11:     return TRUE
12: else
13:     return FALSE
14: end if
```

2.4.3 The parameters of GUI

The main results about the security of HFEv- (and hence QUARTZ/GUI) is that the effective rank for MinRank [KS99] is $r+a+v$, where $r = (\lfloor \log_q^d -1 \rfloor + 1)$ is the rank of the HFE polynomial. An upper bound for the degeneration degree of a Gröbner Basis attack against HFEv- systems is given by [DY13]

$$\mathrm{d_{reg}} \leq \begin{cases} \frac{(q-1)\cdot(r-1+a+v)}{2} + 2 & q \text{ even and } r+a \text{ odd} \\ \frac{(q-1)\cdot(r+a+v)}{2} + 2 & \text{otherwise} \end{cases},$$

and we need to evaluate the complexity of the F5 algorithm [Fau02] at this degree to be at least 2^b for b-bit design security.

Parameters for our 128-bit HFEv- variants are given in Tab. 3. Note that these are both for 256 bit hashes and signatures, repeated 3 times a la QUARTZ/GUI.

Table 3: Parameters of HFEv-/GUI (256bits signatures and hashes).

security	parameter	$\mathbb{F}_2$	$\mathbb{F}_4$
128 bits	$(\mathbb{F}_{q^\ell}, d, v, a, k)$	($\mathbb{F}_{2^{240}}$,9,16,16,3)	($\mathbb{F}_{4^{120}}$,17,8,8,2)
	$n \to m$	$256 \to 224$	$128 \to 112$

2.5 Hidden Medium Field Equations (HmFE) and HmFEv-

HmFEv- where the "m" stands for "medium" is a variant on HFEv- which uses a smaller "big field" but uses more than one hidden equations.

The idea of hidden medium field equations, or multivariate HFE, first appeared in [BPS08].

$$\begin{aligned} p^{(1)}(X_1,\ldots,X_k) &= \sum_{i=1}^{k}\sum_{j=i}^{k} p_{ij}^{(1)} \cdot X_iX_j + \sum_{i=1}^{k} p_i^{(1)} \cdot X_i + p_0^{(1)} \\ p^{(2)}(X_1,\ldots,X_k) &= \sum_{i=1}^{k}\sum_{j=i}^{k} p_{ij}^{(2)} \cdot X_iX_j + \sum_{i=1}^{k} p_i^{(2)} \cdot X_i + p_0^{(2)} \\ &\vdots \\ p^{(k)}(X_1,\ldots,X_k) &= \sum_{i=1}^{k}\sum_{j=i}^{k} p_{ij}^{(k)} \cdot X_iX_j + \sum_{i=1}^{k} p_i^{(k)} \cdot X_i + p_0^{(k)} \end{aligned} \tag{4}$$

With k equations and k unknowns we can eliminate the unknowns to reach exactly one equation of degree at most (and usually equal to) 2^k. This elimination process is a Gröbner basis computation and can be pre-scripted. For $k = 2$ and $k = 3$, the solution process is relatively simple. For $k \geq 4$ it starts to be more work than the eventual univariate equation.

An unmodified HmFE is not secure, just like straight HFE is not secure. The reason is that the rank of HmFE is equation to k, it is susceptible to the same MinRank attacks as HFE. Just like we have HFEv-, we can do HmFEv- to combat this problem. The rank is bounded by $k(1+v)+a$ and is conjectured to be reasonably taken as $k+a+v$. Here the number of variables and equations are equal to $n = k\ell + v$, $m = k\ell - a$. For completeness, the HmFEv central map is, for $h = 1 \dots k$:

$$\begin{aligned} p^{(h)}(X_1, \dots, X_k, \bar{\mathbf{x}}) = & \sum_{i=1}^{k} \sum_{j=i}^{k} p_{ij}^{(h)} \cdot X_i X_j \\ & + \sum_{i=1}^{k} p_i^{(h)}(\bar{\mathbf{x}}) \cdot X_i + p_0^{(h)}(\bar{\mathbf{x}}) \ . \end{aligned}$$

Unlike HFEv-, there is no structure of Patarin variation as in GUI/QUARTZ. We use the parameter of HmFEv- chosen from recent work [PCD+17] in the Tab. 4.

Table 4: Parameters of HmFEv-.

security(bits)	parameter	$\mathbb{F}_{256}$
128	$(\mathbb{F}_{q^\ell}, k, v)$	$(\mathbb{F}_{256^{15}}, 3, 16)$
	$n \to m$	$61 \to 45$

2.6 Implementation Tools: Useful SIMD instructions

Advanced Vector Extensions 2 (AVX2) instruction set is Intel's new SIMD(single instruction multiple data) instruction set in mainstream processors for manipulating integer commands. In the SIMD instruction set, one register can be treated as a group of 8-bit, 16-bit, 32-bit, or 64-bit data and the instruction effects paralleled on multiple data. The size of group is dependent on the size of the machine register. In contrast to previous 128-bit `xmm` registers in SSE instruction sets, the size of registers in AVX2 extends to 256-bit `ymm` registers, which affords us 32-way parallelism for 8-bit data.

Beside the common SIMD for arithmetic, logic, or data manupulations, there are some key instructions heavily used in our $\mathcal{MQ}$ implementations:

`PSHUFB` in SSE takes a source considered as a lookup table of 16 bytes, $(x_0, x_1, \dots, x_{15})$, and does a simultaneous lookup using the other operand register $(y_0, y_1, \dots, y_{15})$ as 16 indices, where the result at position i is $x_{y_i \bmod 16}$ except negative indices result in 0. The AVX2 instruction `VPSHUFB` performs `PSHUFB` 2 times in one instruction.

`VPMADDUBSW` requires two vectors of 32 8-bit numbers $(x_0, \dots, x_{31})$ and $(y_0, \dots, y_{31})$ and then computes vector of 16 16-bit words $(x_0y_0 + x_1y_1,\ x_2y_2 + x_3y_3,\ \dots, x_{30}y_{30} + x_{31}y_{31})$. This instruction is very useful for implementing efficient arithmetic in small prime field.

`PMULHRSW` performs the multiplication of 16-bit binary fixed point fractions, rounded and signed. This instruction is useful for taking the remainder of 16-bit integers modulo a small prime.

`PCLMULQDQ` performs the multiplication of two polynomials of degree 63 over $\mathbb{F}_2(\mathbb{F}_2[x])$ and result in a degree 127 polynomials over $\mathbb{F}_2$.

Note that PCLMULQDQ is part of AES instruction set (AES-NI) and is absent in many Intel Core-i3 processors. The other three instructions are available in all Intel-compatible processors manufactured today since a few years ago. Most larger ARMs have a corresponding vector instruction to PSHUFB called TBL. 64-bit ARM has a corresponding instruction to PCLMULQDQ, but 32-bit ARMs don't, and some small 32-bit ARM microprocessors don't have vector instructions at all.

3 Evaluating of Quadratics and the Public Map

The evaluation of $\mathcal{M}Q$ is an important component in MPKC signatures and corresponds to the verification of signature or the public map directly. We don't require constant-time evaluations in the public map since the computation is publicly executable. The evaluation of $\mathcal{M}Q$ also appears in the secret map of some MPKC-signature schemes, e.g., generating Eq. 1 in Rainbow. In this case time constancy is required when evaluating $\mathcal{M}Q$.

In this section, we review arithmetic in various GFs underlying the $\mathcal{M}Q$ equations and followed by the evaluation of $\mathcal{M}Q$ with respect constant-time and non-constant-time cases.

3.1 GF Arithmetic in a small field

A Finite field, or Galois Field (GF), is an algebraic structure, a field containing a finite number of elements. It plays an important role in the areas of math and computer science. To perform the arithmetic in GF, the rule of thumb is always choose equivalent native instruction if it is supported on the platform. However, there are only few GFs with the multiplications correspond to native hardware instructions in mainstream CPUs, so the efficient software implementation of GF arithmetic is a topic of great interest in computer engineering.

3.1.1 Small prime field such as $\mathbb{F}_{31}$

Hardware parallel add and multiply instructions (mostly VPMADDUBSW, see previous section) are used. However another key to efficient $\mathbb{F}_{31}$ arithmetic is handling reductions modulo 31. Since $31 = 2^5 - 1$, a lazy reduction instead of full reduction can be done for $\mathbb{F}_{31}$ by shift 5 bits right and adding. The aforementioned VPMULHRSW helps carrying out Barrett reduction. Having said that, in general we need to avoid reductions as much as possible and keep the operations as packed as possible.

3.1.2 $\mathbb{F}_2$ and $\mathbb{F}_4$

The $\mathbb{F}_2$ is probable the only GF with fully HW support, which the multiplications and additions correspond to AND and XOR respectively. However, there are usually 32-bit or larger machine words in nowadays CPU instead of one "Bit" for $\mathbb{F}_2$, the main issue in implementing systems over $\mathbb{F}_2$ is to utilize the full width of the machine word. In the case of $\mathbb{F}_4$, we believe that the best way to do multiplication is usually to use bit operations. For this, the 2-bits in one $\mathbb{F}_4$ is often stored in separate registers, or "bitsliced". A multiplication costs 4 AND

and 1 `XOR` for un-reduced 3-bits results and 2 more `XOR` for reducing to 2-bits form of $\mathbb{F}_4$.

3.1.3 The case of $\mathbb{F}_{16}$

Use `VPSHUFB/TBL` for multiplication tables is the general strategy of multiplications in $\mathbb{F}_{16}$. While multiply a bunch of $\mathbf{a} \in \mathbb{F}_{16}$ stored in a SIMD register with a scalar $b \in \mathbb{F}_{16}$, we load the table of results of multiplication with b and follow one `(V)PSHUFB` for the result $\mathbf{a} \cdot b$. However, the address of table is a side-channel leakage which reveals the value of b to a cache-time attack [BM06].

When time-constancy is needed, the straightforward method is to use again `VPSHUFB`, but for logarithm and exponential tables, and store in log-form if warranted. That is, we compute $a \cdot b = \exp(\log_g a + \log_g b)$, and due to the characteristic of (V)PSHUFB, setting $\log 0 = -42$ is sufficient to make this operation time-constant even if we multiply three elements. We shall see a different method below when working on an constant-time $\mathcal{MQ}$ evaluation for $\mathbb{F}_{16}$.

3.1.4 The case of $\mathbb{F}_{256}$

The GF of 256 elements occupies exact one byte in storage and have been extensively studied, e.g. [CCC+09,CCY13], since it is the basic building elements of numerous applications in the area of cryptography. Multiplications in $\mathbb{F}_{256}$ can be implemented as 2 table lookup instructions in the mainstream Intel SIMD instruction set. However, this is not time-constant.

For time-constant multiplications, we adopt the *tower field* representation of $\mathbb{F}_{256}$ which formulating an element in $\mathbb{F}_{256}$ as degree-1 polynomial over $\mathbb{F}_{16}$. The sequence of tower fields from which we build $\mathbb{F}_{256}$ is the following:

$$\begin{aligned}
\mathbb{F}_4 &:= \mathbb{F}_2[e_1]/(e_1^2 + e_1 + 1), \\
\mathbb{F}_{16} &:= \mathbb{F}_4[e_2]/(e_2^2 + e_2 + e_1), \\
\mathbb{F}_{256} &:= \mathbb{F}_{16}[e_3]/(e_3^2 + e_3 + e_2 e_1) \ .
\end{aligned}$$

In the rest of this paper, we adopt the following correspondence: our basis is $(1, e_1, e_2, e_1e_2, e_3, e_1e_3, e_2e_3, e_1e_2e_3)$. The element encoded as `0x2` is e_1, `0x4` is e_2, `0x8` is e_1e_2, `0x10` is e_3 etc., and numbers up to `0xff` are their combinations, for example `0x1d` $= e_3 + e_1e_2 + e_2 + 1$. In this representation, we can build constant-time multiplications over $\mathbb{F}_{256}$ from the techniques of $\mathbb{F}_{16}$. A time-constant $\mathbb{F}_{256}$ multiplication costs about 3 $\mathbb{F}_{16}$ multplications for multiplying 2 degree-1 polynomials over $\mathbb{F}_{16}$ with the Karatsuba method and one extra table lookup instruction for reducing the degree-2 term.

3.2 The evaluation of $\mathcal{MQ}$

Note on lack of special structures in $\mathcal{MQ}$

For the evaluation of quadratic equations (which is the public map of MPKCs), there is no real method to reduce the required computations since we expect to be evaluating a set of random-looking equations unless special patterns were designed into the equations (which only happens in certain unusual variant schemes which does not concern us here). Since the amount of required computations is the same across all platforms, the main focus in evaluating $\mathcal{MQ}$

is to reduce the required number of cycles via choosing the correct instruction sequences over various platforms to achieve the required computations. Most of the time, this equates to using the fewest instructions.

3.2.1 Matrix-Vector and Scalar-Vector product

Usually a multivariate quadratic system $\mathcal{P}$ is stored as a column-major matrix with the columns being all monomials up to degree 2 and the rows being the equations. The evaluation of $\mathcal{P}$ can roughly be divided in two parts: the generation of all monomials, and computation of the resulting polynomials for known monomials. Generating the quadratic monomials given the variables requires $n \cdot (n+1)/2$ multiplications. The second part requires $m \cdot (n + n \cdot \frac{n+1}{2})$ multiplications to multiply the coefficients of $\mathcal{P}$ with the quadratic monomials and almost exactly the same number additions to accumulate results. The second part is clearly more computationally intensive.

The computation proceeds by accumulating the product of a column vector with a prepared monomial as showed in Fig. 1. This is exactly a matrix-vector production. So we can thus keep all results in the registers in this representation.

$$\boxed{\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ \vdots \end{array}} = \boxed{\begin{array}{c} c_{11} \\ c_{21} \\ c_{31} \\ \vdots \end{array}} \cdot x_1 + \boxed{\begin{array}{c} c_{12} \\ c_{22} \\ c_{32} \\ \vdots \end{array}} \cdot x_2 + \dots$$

Figure 1: An example of parallel evaluation of polynomials. The results $x_1 \cdot (c_{11}, c_{21}, c_{31}, \dots)$, $x_2 \cdot (c_{12}, c_{22}, c_{32}, \dots)$, etc. are accumulated to $(y_1, y_2, y_3, \dots)$.

The computational complexity of evaluation is clearly proportional to the number of monomials multiplied with coefficients of polynomials. In general this is equal to the number of coefficients which is $\frac{1}{2}n \cdot (n+3) \cdot m$ multiplication in total. We cannot optimize the computations by the value according to the computed monomials (zero) if they are secret data.

There are 2 alternative methods for dealing with quadratic terms. First is to generate all quadratic monomials and then multiply them to all coefficients. To generate all monomials, we arrange the variables in registers and follow by multiplying them by each variable. We need to shift the results to pack them together. This requires careful handling and is not always straightforward.

The second method generates the quadratic terms through multiplications by variables (twice). In a degree-reverse-lex order for the monomials of polynomial, the quadratic terms is ordered as $c_{11}x_1x_1 + (c_{12}x_1 + c_{22}x_2)x_2 + (c_{13}x_1 + c_{23}x_2 + c_{33}x_3)x_3 + \cdots$. One can accumulate all the linear terms in one parentheses and follows with a multiplication with second variable. There are $n \cdot m$ extra multiplications caused by this method. One can choose the method of calculation of quadratic terms with the value of n and m for a lower cost of computation, except when doing the constant-term evaluation of $\mathcal{M}Q$ in $\mathbb{F}_{16}$ (see below) where one has to choose the second method.

3.2.2 Optimization from the viewpoint of streaming data

The evaluation of $\mathcal{P}$ is actually depending on how fast one can accumulate all data of $\mathcal{P}$. No matter what instructions we choose to perform the calculations, the inevitable fact is the we have to load all data of $\mathcal{P}$. The optimization process is how to minimize the number of cycles (usually meaning instructions) between 2 load instructions of coefficients of $\mathcal{P}$. We discuss the various cases of evaluation of $\mathcal{M}Q$ according to the underlying GF. The results of our implementations are reported in Tab. 5.

3.2.3 $\mathcal{M}Q$ over $\mathbb{F}_2$ and $\mathbb{F}_4$

The main operations in the deepest loop should contain only the accumulation between load instruction of polynomials since `AND` and `XOR` are native HW instructions for arithmetic in these field. We have to prepare the input data achieve this situation. For vertical evaluation of $\mathcal{M}Q$, we broadcast every bit of $\mathbb{F}_2$ to the full SIMD register and store them in stack with their order of variable. While accumulating the results, we can load the variables by their corresponding positions and the instructions remained are only on `AND` and one `XOR`. In the Tab. 5, we can see the effect of non-constant acceleration came from skipping some coefficients of equations from multiplying 0.

3.2.4 $\mathcal{M}Q$ over $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$

For truly public-key operations, the multiplications over $\mathbb{F}_{16}$ can be done by simply (1) loading the multiplication tables(`multab`) by the value of the multiplier and (2) performing a `VPSHUFB` for 32 results simultaneously. The multiplications over $\mathbb{F}_{256}$ can also be performed with the same technique via 2 `VPSHUFB` instructions since one lookup deals 4 bits. Other tricks are multiplying by two $\mathbb{F}_{16}$ elements to a vector of $\mathbb{F}_{16}$ elements with one `VPSHUFB` since `VPSHUFB` can actually be seen as 2 independent `PSHUFB` instructions. This method of multiplication can easily transform to time-constant version by replacing `multab` to log/exp tables as in Sec. 3.1.3.

However, since the log/exp strategy costs many operations on addition, reduction of sum, and looking up in the exponential table, we should try to use a `multab` strategy in evaluating $\mathcal{M}Q$ (since it costs only one `VPSHUFB`), in order to increase efficiency, even under the constant-time requirement.

Constant-Time $\mathcal{M}Q$ evaluation in $\mathbb{F}_{16}$ We have to avoid loading `multab` according to a secret index for preventing cache-time attack. To this, we "generate" the desired `multab` instead of "load" by secret value. More precisely, suppose we are evaluating $\mathcal{P}$ with a vector $\mathbf{w} = (w_1, w_2, \ldots, w_n) \in \mathbb{F}_{16}^n$, we can have a time-constant evaluation if we already have the `multab` of $\mathbf{w}$, which is $(w_1 \cdot \texttt{0x0}, \ldots, w_1 \cdot \texttt{0xf}), \ldots, (w_n \cdot \texttt{0x0}, \ldots, w_n \cdot \texttt{0xf})$, in the registers. [1]

[1]Note that here and in the following, if there is a natural basis ($b_0 = 1, b_1, \ldots$) in a binary field $\mathbb{F}_q$, for convenience we represent b_j as 2^j. So b_1 is 2, $1 + b_1$ as 3, ..., $1 + b_1 + b_2 + b_3$ is

In other words, we transform the memory access indexed by a secret value to sequential access by the index of variables to prevent revealing of side-channel information.

We show the generation of `multab` for elements of $\mathbf{w}$ in Fig. 2. To generate the desired `multab` on-the-fly using the 16x16 `multab` for $\mathbb{F}_{16}$, we first multiply $\mathbf{w}$ by `0x0`, then `0x1`, then the rest of the elements of $\mathbb{F}_{16}$. Now we have 16 registers in which are the products of $\mathbf{w}$ and all 16 elements of $\mathbb{F}_{16}$. By collecting the first bytes, second bytes ... etc. of these, we get our desired new `multab`.

$$\begin{array}{c} \mathbf{w} \cdot \mathtt{0x0} \\ \mathbf{w} \cdot \mathtt{0x1} \\ \vdots \\ \mathbf{w} \cdot \mathtt{0xf} \end{array} \longmapsto \boxed{\begin{array}{c} w_1 \cdot \mathtt{0x0} \\ w_1 \cdot \mathtt{0x1} \\ \vdots \\ w_1 \cdot \mathtt{0xf} \end{array}}, \boxed{\begin{array}{c} w_2 \cdot \mathtt{0x0} \\ w_2 \cdot \mathtt{0x1} \\ \vdots \\ w_2 \cdot \mathtt{0xf} \end{array}}, \ldots \boxed{\begin{array}{c} w_{15} \cdot \mathtt{0x0} \\ w_{15} \cdot \mathtt{0x1} \\ \vdots \\ w_{15} \cdot \mathtt{0xf} \end{array}}$$

Figure 2: Generating `multab` for $\mathbf{w} = (w_1, w_2, \ldots w_{16})$. After $\mathbf{w} \cdot \mathtt{0x0}$, $\mathbf{w} \cdot \mathtt{0x1}$, $\ldots$, $\mathbf{w} \cdot \mathtt{0xf}$ are calculated, each row stores the results of multiplications and the columns are the `multab` corresponding to $w_1, w_2, \ldots, w_{15}$. The `multab` of w_1, w_2, $\ldots$,w_{15} can be generated by collecting data in columns.

A further matrix-transposition-like operation is needed to generate the desired `multab`, since the initial byte from each register forms our first new table, corresponding to w_1, the second byte from each register is the table of multiplication by w_2, etc. *The obvious way to do this is by shuffle instructions, but this matrix transposition operation is actually very fast on newer Intel processors simply by moving bytes into memory, due to some hardware-related scatter-gather magic in the L1 Cache.* One desired table cost 16 `PSHUFB` to generate and we can generate 16 or 32 tables simultaneously according to SIMD environment. The amortized cost for generating one `multab` is 1 `PSHUFB` plus some data movements.

As a result, the constant-time evaluation of $\mathcal{MQ}$ over $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ is then only slightly lower than the non-constant time version since the extra cost is low, with only n tables to be generated before the evaluation begin. In Tab. 5, we can see only about 5% difference between constant-time and general evaluations.

3.2.5 $\mathcal{MQ}$ over $\mathbb{F}_{31}$

The matrix-like coefficients of $\mathcal{P}$ are stored as 8-bit values because we heavily rely on the AVX2 instruction `VPMADDUBSW`. In one instruction, this computes two 8-bit SIMD multiplications and a 16-bit SIMD addition(see Sec. 2.6). This requires a slight variation on the representation of $\mathcal{P}$ described above: we put coefficients in a column major matrix with each 16-bit element corresponding to **two** adjacent monomials. All these operations are time-constant.

`0xF` for elements of $\mathbb{F}_{16}$, and continuing for larger fields; this is analogous to how the AES field representation of $\mathbb{F}_{2^8}$ is called `0x11B` because its irreducible polynomial is $x^8+x^4+x^3+x+1$.

Because VPMADDUBSW takes both a signed and an unsigned operand, one of the matrix and the monomial vector must be stored as signed bytes and one as unsigned bytes. Since $64 \cdot 31 \cdot 15 = 29760 < 2^{15}$, we can handle two YMM register full of monomials before performing reductions on each individual accumulator. This is different from [CCC+09] because they were still using SSE2 and PMADDWD, which produces a 32-bit result and makes the bookkeeping easier.

Field elements during computation are expressed as signed 16-bit values. If $m = 64$, we require 1024 bits of storage for each vector, precisely fitting four 256-bit SIMD (YMM) registers. If $m = 32$, two registers.

To efficiently compute all polynomials for a given set of monomials, we keep all required data in registers and try to avoid register spilling throughout the computation, as much as possible.

Table 5: Benchmarks on evaluations of quadratic polynomials on Intel XEON E3-1245 v3 @ 3.40GHz with AVX2 instruction set, in CPU cycles.

system	size	constant time	general
	k byte	k cycles	k cycles
$\mathbb{F}_2, n = 256, m = 256$	1020	92.8	51.5
$\mathbb{F}_4, n = 128, m = 128$	258	32.3	25.6
$\mathbb{F}_{16}, n = 64, m = 64$	65	9.6	9.1
$\mathbb{F}_{31}, n = 64, m = 64$	130 [a]	8.7	8.7
$\mathbb{F}_{256}, n = 64, m = 64$	130	16.2	15.6

[a] Each element over $\mathbb{F}_{31}$ is stored in one byte.

4 Main components in the secret map

In this section, we discuss the key components in various MPKC signatures.

4.1 Solving Linear Equations

Solving linear equations (1) takes up much of the time in the signing process of Rainbow/TTS as seen in Sec. 2.2.2. In [CCC+09] this was done using Wiedemann over $\mathbb{F}_{31}$ and reported to be faster than Gauss Elimination due to not needing to as many reductions modulo 31. However, since there are no reduction issues for the binary GF arithmetic (see Sec. 3.1) and the asymptotic complexity is actually lower for Gauss Eliminations, we decided to implement the constant-time solver with a simpler Gauss Elimination in this paper.

We use constant-time Gauss Elimination in the signing process of Rainbow. Constant-time Gauss Elimination originally presented in [BCS13] for $\mathbb{F}_2$ matrices and we extend the method to other GFs. The problem of eliminations is that the pivot may be zero and one has to swap rows with zero pivots with other rows, which reveals side-channel information. To test pivots against zero and switch rows in constant time, we can use the current pivot as a predicate for conditional moves and switch with every possible row which can possibly con-

Table 6: Benchamrks on solving linear systems with Gauss elimination on Intel XEON E3-1245 v3 @ 3.40GHz, in CPU cycles.

system	plain elimination	constant version
$\mathbb{F}_{16}, h = 32, w = 32$	6,610	9,539
$\mathbb{F}_{31}, h = 28, w = 28$	7,889	10,227
$\mathbb{F}_{256}, h = 20, w = 20$	4,702	9,901

tain non-zero leading terms. This constant-time Gaussian elimination is slower as reported in Tab. 6, but is still an $O(n^3)$ operation.

4.2 GF Arithmetic – large fields

The arithmetic over big GFs is the most important component in big-field MP-KCs. In this section, we discuss the multiplication over $\mathbb{F}_{2^n}$ which is almost equivalent to the multiplication in $\mathbb{F}_2[x]$. We divide our discussion into to two parts by the existence of `PCLMULQDQ` in the platform.

4.2.1 Platforms with `PCLMULQDQ`

In the platform with `PCLMULQDQ`, the obvious thing to do is use the monomial representation over $\mathbb{F}_2$ to implement $\mathbb{F}_{2^k}$. When it comes to fields of sizes of cryptographic interest, choosing the representation for the fastest operations depends very much on the underlying hardware for implementation. We show the representations in this paper for `PCLMULQDQ` in Tab. 7.

Table 7: The field representations for `PCLMULQDQ` instruction.

$\mathbb{F}_{2^{384}} :=$	$\mathbb{F}_2[x]/x^{384} + x^8 + x^7 + x^6 + x^4 + x^3 + x^2 + x + 1$
$\mathbb{F}_{2^{256}} :=$	$\mathbb{F}_2[x]/x^{256} + x^{10} + x^5 + x^2 + 1$
$\mathbb{F}_{2^{240}} :=$	$\mathbb{F}_2[x]/x^{240} + x^8 + x^5 + x^3 + 1$
$\mathbb{F}_{2^{128}} :=$	$\mathbb{F}_2[x]/x^{128} + x^7 + x^2 + x + 1$
$\mathbb{F}_{2^{120}} :=$	$\mathbb{F}_2[x]/x^{120} + x^4 + x^3 + x + 1$

The multiplication in large GFs are implemented as polynomial multiplication in $\mathbb{F}_2[x]$ and followed by a reduction, i.e., taking the remainder modulo the polynomial defining the field extension. For the details of multiplying with `PCLMULQDQ`, the data is split in 64-bit limbs. In general we are working on the polynomial multiplication of 2 to 6 limbs. The multiplication in $\mathbb{F}_2[x]$ was accomplished by recursive 2- or 3-way Karatsuba's multiplication. For reducing the results of polynomial multiplication to its original length, this operation is also accomplished by `PCLMULQDQ`. We choose the generating polynomial of field with low-degree second term so the polynomials for reduction won't exceed x^{63}. For example of $\mathbb{F}_{2^{240}}$, we modified the polynomial of $x^{240} + x^8 + x^5 + x^3 + 1$ to $x^{256} + x^{24} + x^{21} + x^{19} + x^{16}$ so the polynomial $x^{24} + x^{21} + x^{19} + x^{16}$ fit into 64-bit range. The reduction is performed by reducing partial polynomial with degree over x^{384}, x^{320}, x^{256}, and x^{240} iteratively.

4.2.2 Big GF multiplications without `PCLMULQDQ`

For processors with SIMD table lookup instructions but without `PCLMULQDQ`– most Core-i3 CPUs don't have this instruction, and most ARMv7 with Neon also fits this description, we build the desired big GF from $\mathbb{F}_{256}[x]$ (polynomials over $\mathbb{F}_{256}$). The arithmetic of $\mathbb{F}_{256}$ is described in Sec. 3.1.4.

For general 32-bit platforms, such as the ARM Cortex M series, or other ARMs without Neon, the base field $\mathbb{F}_{256}$ is built by bit-slicing, i.e., storing the 8 bits of $\mathbb{F}_{256}$ across 8 registers. Starting out with $\mathbb{F}_2[x]$, the multiplications over $\mathbb{F}_{256}$ is built as three rounds of Karatsuba multiplications, which is the lowest bit operations count for $\mathbb{F}_{256}$ in [BC14].

The Constructions of GF and its Multiplication The $\mathbb{F}_{2^k}$ in this paper can also be extended from $\mathbb{F}_{256}$. Here are the field extensions we used in this work:

$$\begin{aligned}\mathbb{F}_{2^{64}} &:= \mathbb{F}_{256^8} := \mathbb{F}_{256}[x]/(x^8 + x^3 + x + \texttt{0x10})\\ \mathbb{F}_{2^{128}} &:= \mathbb{F}_{256^{16}} := \mathbb{F}_{256}[x]/(x^{16} + x^5 + x^3 + \texttt{0x10})\\ \mathbb{F}_{2^{256}} &:= \mathbb{F}_{256^{32}} := \mathbb{F}_{256}[x]/(x^{32} + \texttt{0x10} \cdot x^3 + x + 1)\ .\end{aligned}$$

The multiplication of these GF comprise the polynomial multiplications in $\mathbb{F}_{256}[x]$ and a reduction (modulo the irreducible polynomial defining the field). Since the reduction is performed by some multiplication with constant over $\mathbb{F}_{256}$, it can be easily accomplished with the SIMD method described in Sec. 3.1.4. We discuss the polynomial multiplication in $\mathbb{F}_{256}[x]$ in the following sections.

FFT Polynomial Multiplications over $\mathbb{F}_{256}$ It is well known that polynomial multiplications can be accomplished by FFT algorithm [CLRS09]. To multiply two degree-$(n-1)$ polynomials $a(x), b(x) \in \mathbb{F}_{256}[x]$ with FFT algorithm, one can

1. (`FFT`) evaluate $a(x)$ and $b(x)$ at $2n$ points by a FFT algorithm,
2. (`pointMul`) multiply the evaluated values pairwise together, and
3. (`ivsFFT`) interpolate back into a polynomial of degree $\leq 2n-1$ by the inverse FFT algorithm.

However, it was not easy to build a suitable FFT for GF of characteristic two($\mathbb{F}_{2^k}$), since there is not always an applicable $w \in \mathbb{F}_{2^k}$ such that $w^m = 1$ for a large range of m. In 2014, Bernstein and Chou [BC14] showed the additive[2] FFT [GM10] provides an efficient polynomial multiplications in the circumstance of $\mathbb{F}_{2^k}$.

For better efficiency, we implement a variation of the Gao-Mateer additive FFT, which is a generalization of Gao-Mateer FFT proposed by Lin, Chung, and Han(LCH) [LCH14], in this paper. Using the LCH's additive FFT, we first carry out a sequence of additions for converting the polynomial to a polynomial basis, presented by Cantor [Can89], in $\Theta(n \log n \log\log n)$ operations (see Fig. 3) and

[2] Following the terminology of Gao-Mateer, "additive" FFT means the evaluation points are not a multiplicative subgroup generated by $w = e^{2\pi i/2^k}$ but in a vector space comprising GF or its subset.

follow up with a $\Theta(n \log n)$ butterfly network much like the standard FFT (in Fig. 5). We call the first stage of additions "basis conversion" which corresponds to "bit reversal" exchanges between the coefficients in regular Decimation-in-Time and in-place FFT.

Note that the butterfly network in the forward transform typically splits into two smaller butterfly networks, fed with the same input but with different offsets and multipliers, just like multiplicative FFT's. Furthermore, we discover that when using a tower construction in the additive FFT, all the multipliers in the butterflies have regular and simple forms. There are only some *small* [3] constants in the multipliers and the calculation in a butterfly can be accomplished with less instructions for multiplying these constants. It turns out the general(constant-time) multiplications in $\mathbb{F}_{256}$ are only performed in the pairwise multiplications (step (2)). Details of the additive FFT can be found in [GM10, LCH14, BC14]

Truncated Additive Fourier Transform For multiplying polynomials containing terms is not power of two, we can also use a ***truncated FFT*** [Mat11, Har09] for omitting some computations. These previous research focused on the remaining evaluated points which becomes straightforward in the LCH FFT since the butterfly network is quiet regular(see Fig. 6).

We simply omit the calculation related to higher degree in the `ivsFFT` since we can expect the zero values after `ivsFFT` from the degree of input polynomials. If a portion of the coefficients is zero in the polynomial, then

- it is easy to simplify the `FFT` by omitting the zero in higher degree of inputs and the outputs related to "larger" evaluated points;
- also easy to simplify (cf. Fig. 3) the basis conversion stage, which only involves adding from higher degree to lower degree coefficients, both going forwards and backwards;
- and not very obvious but still true that the inverse butterflies can be simplified, knowing that a portion (in Fig. 6–8 exactly one quarter) of the polynomial coefficients are zero.

This turns out to be the case due to the multipliers in the final butterfly stages of the `ivsFFT` being particularly simple.

We extend the method in Fig. 8 to polynomials of 96 terms for implementing $\mathbb{F}_{2^{384}}$ which is represented as

$$\mathbb{F}_{2^{384}} := \mathbb{F}_{256}[x]/x^{48} + x^3 + \texttt{0x10} \cdot x^2 + \texttt{0x4} \cdot x + 1 \ .$$

The details of truncated `ivsFFT` are similar to Fig. 8 since we omit exactly one quarter of original `ivsFFT` results in both case. Aside from completely omitting the computations of the last-quarter coefficients, we still have to specialize the last two layers of butterflies. (Since there are no interaction between fourth-quarter coefficients and others before last two layers of butterflies, only two layers have to be specialized.)

[3]Small here means that the encoding of the element as a hexadecimal number is small.

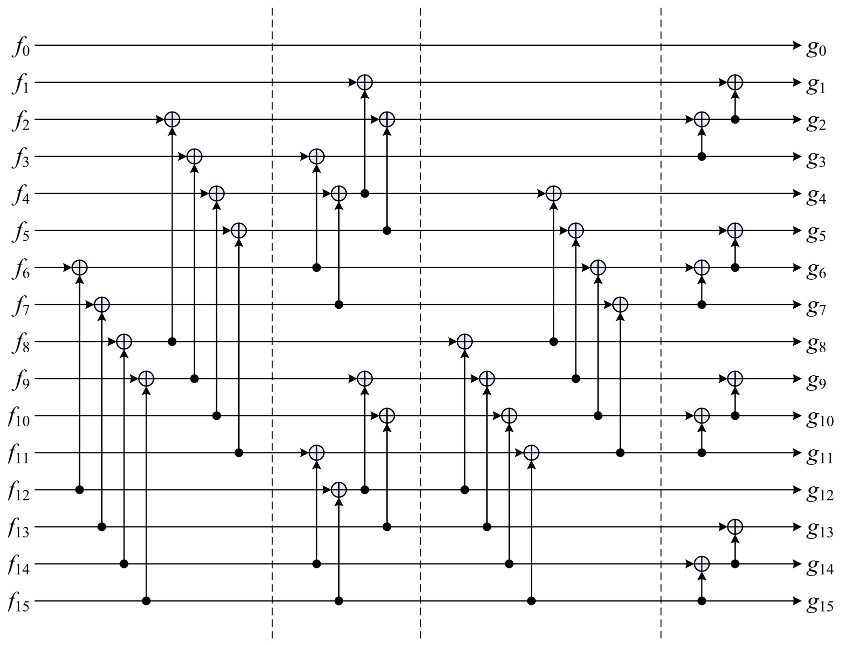

Figure 3: The forward basis conversion, 16 coefficients

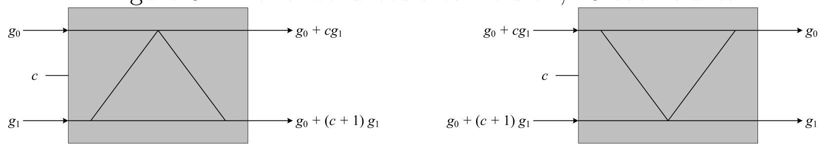

Figure 4: The forward/inverse butterfly units.

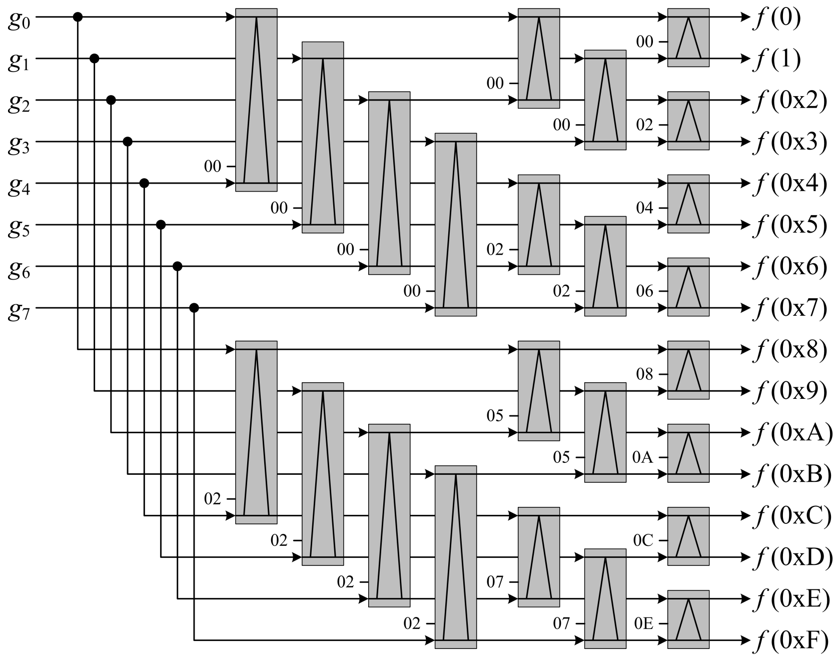

Figure 5: Forward butterfly network for degree-7 polynomials in $\mathbb{F}_{256}[x]$.

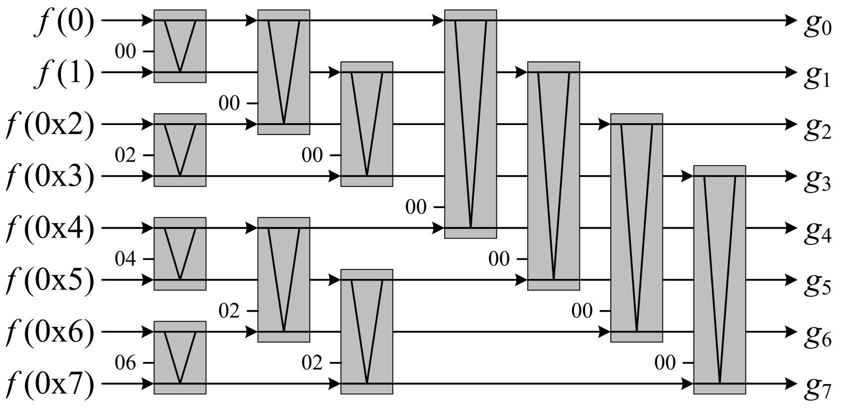

Figure 6: Inverse butterfly network for degree-7 polynomials in $\mathbb{F}_{256}[x]$.

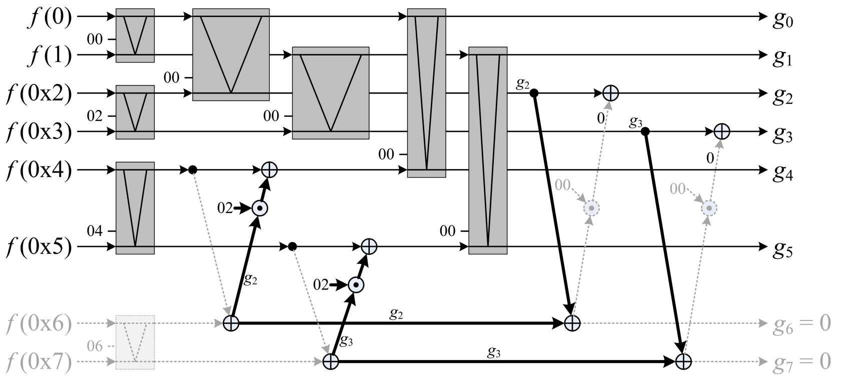

Figure 7: Same inverse butterfly network **with two known zeroes**.

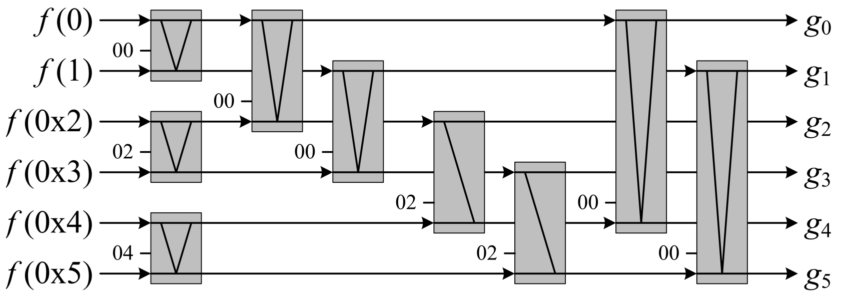

Figure 8: Inverse butterfly network for degree-5 polynomials in $\mathbb{F}_{256}[x]$.

Alternative method for multiplications in $\mathbb{F}_{2^{384}}$ For field whose size is not equal or just below a power of two, we can also choose to extend from different base fields besides truncated FFT. For example, we may use the polynomial multiplication of $\mathbb{F}_{256^3}[x]$ to implement the $\mathbb{F}_{2^{384}} := \mathbb{F}_{(256^3)^{16}}$:

$$\mathbb{F}_{2^{24}} := \mathbb{F}_{256^3} := \mathbb{F}_{256}[x]/x^3 + \texttt{0x2}$$
$$\mathbb{F}_{2^{384}} := \mathbb{F}_{(2^{24})^{16}} = \mathbb{F}_{2^{24}}[x]/x^{16} + \texttt{0x2}x^3 + x + \texttt{0x10} \ .$$

We would then use a Karatsuba algorithm with 3 terms in the `pointMul` stage for multiplications in $\mathbb{F}_{256^3}$, which cost 6 multiplications over $\mathbb{F}_{256}$ for one multiplication over $\mathbb{F}_{2^{24}}$. Note that the multiplications in this stage is constant-time multiplications(see Sec. 3.1.4) which cost higher than general multiplications in `FFT` and `ivsFFT` stages. In both representation, the hight of `FFT` are 96 over $\mathbb{F}_{256}$. There are $\log 32 = 5$ layers butterflies in $\mathbb{F}_{2^{24}}$ `FFT` but $\lceil \log 96 \rceil = 7$ layers in $\mathbb{F}_{256}$. The detailed cost of these 2 representations can be found in Tab. 8. Although the count of multiplications for degree-15 $\mathbb{F}_{2^{24}}[x]$ is less than degree-48 $\mathbb{F}_{256}[x]$ in Tab. 8, the implementation of $\mathbb{F}_{256}[x]$ multiplications is actually 6% faster than $\mathbb{F}_{2^{24}}[x]$ in our experiment.

Table 8: Cost of polynomial multiplications for degree-15 $\mathbb{F}_{2^{24}}[x]$ and degree-48 $\mathbb{F}_{256}[x]$, in number of multiplications over $\mathbb{F}_{256}$

	32 terms $\mathbb{F}_{2^{24}}$	96 terms $\mathbb{F}_{256}$
`FFT`	$98 \cdot 3$	450
`pointMul` [a]	$32 \cdot 6$	96
`ivsFFT`	$49 \cdot 3$	241
total	633	787

[a] The cost of constant-time multiplication in `pointMul` is actually higher the multiplications in `FFT` and `ivsFFT`.

Although the multiplications cost similarly for these different representations, the difference in arithmetic are also effected by the construction of GF and discussed in Sec. 4.2.3.

Benchmarks on GF multiplications We shows the benchmarks of our implementations on GF multiplications over various instruction set in Tab. 9. Besides `PCLMULQDQ`, all GFs are represented as $\mathbb{F}_{256}[x]$ and implemented in the SIMD style which many copies of GF multiplications are performed simultaneously. The multiplications over $\mathbb{F}_{256}$ are implemented with SSE instructions as in Sec. 3.1. We also use bit-slice implementations as in [BC14] for platforms without SIMD instructions. For comparing the effect of FFT-related multiplications, we also list the results for $\mathbb{F}_{256^{16}}$ implemented by school-book multiplications.

The results show `PCLMULQDQ` outperform all other implementations. For example, in the case of $\mathbb{F}_{2^{384}}$, the amortized cost of SSE-FFT implementations are 5.4 times slower than `PCLMULQDQ` version. The results also showed there was a huge gap between FFT and school book implementations.

Table 9: Benchamrks on multiplications of big GFs on Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz, in CPU cycles.

	$\mathbb{F}_{2^{128}}$	$\mathbb{F}_{2^{256}}$	$\mathbb{F}_{2^{384}}$
`PCLMULQDQ`	25	44	76
`PSHUFB`,FFT	1,462/16	3,679/16	6,582/16
Bit-slice(32-bit),FFT	12,232/32	31,249/32	50,827/32
school book,SSE	519	1,080	2,087

4.2.3 Power of big-GF

In the signing process of pFLASH, we have to raise an element X in big-GF to a high power h in Eq. (2). The raising process occurs even in calculation the multiplicative inverse by little-Fermat's law like process for time constancy. It is traditionally done by a square-and-multiply process. Since the power is not a secret value in these scenarios, what we concerns here is only the issue of efficiency.

We generate the pattern of power by a divide-and-conquer process. For example, if we want to raise $a \in \mathbb{F}_{2^{128}}$ to $a^{\texttt{0xFFFF}4}$. we sequentially generate $a^{\texttt{0x3}}$, $a^{\texttt{0xF}}$, $a^{\texttt{0xFF}}$, and $a^{\texttt{0xFFFF}}$ by few squares and one multiplication.

The other method to accelerate the process is to bunch some squares into a linear map. This process is linked to the field representation. For example, if we construct $\mathbb{F}_{2^{128}} := \mathbb{F}_2[x]/x^{128} + x^7 + x^2 + x + 1$, all "raising to the 2^j-th powers" are linear maps (in the vector space $\mathbb{F}_2^{128}$). Assuming we know 16 squares takes more than a linear map by experimentally, we would implement raising to the 2^{16}-th power with a linear map instead of squaring 16 times. Alternatively, if we build the field as $\mathbb{F}_{2^{128}} := \mathbb{F}_{256}[x]/x^{16} + x^5 + x^3 + \texttt{0x10}$, only raising to the 256^j-th powers can be linear maps (in the vector space $\mathbb{F}_{256}^{16}$). We express raising to any given power by as a sequence of squares, multiplies and linear maps interleaved, depending on benchmarking results.

4.2.4 Conversion between field representations

We require a method of changing field representations for the compatibility between different field representations. The change of field representation is simply done as multiplying a pre-defined matrix with the data treated as a vector. The matrix product can be computed by the famous method of four Russians [AH74]. However, while multiplying with secret values, this requires a constant-time multiplication which is often done with conditional move instruction. In this work, we broadcast single bit to full register and followed by `AND` and `XOR` for accumulation when working in SSE or AVX instruction set.

5 The Implementations and Benchmarks

In this section, we give comparisons of benchmarks among various signing schemes, including different MPKCs and some widely used schemes (though

[4] Note the hexadecimal number here is simply for conveniently reading the number in binary, not for representing elements in GF.

not post-quantum ones). Almost all the schemes in the comparisons are parametered at a 128-bit security level, besides the RSA-2048 is in the 112-bit security level. Tab. 10 lists the specific parameters for all schemes under comparisons.

Table 10: Specific parameters for 128-bit MPKCs and other signing schemes.

schemes	pub.key kbyte	sec.key kbyte	digest bit	signature bit
Rainbow(16,32,32,32)	145.5	100.2	256	384
Rainbow(31,28,28,28)	236.5	156.9	256	448
Rainbow(256,28,20,20)	94.3	62.9	320	544
PFLASH(16,96-1,64)	142.5	9.1	256	384
GUI(2,240,9,16,16,3)	899.5	21.2	256	320
GUI(4,120,17,8,8,2)	225.8	9.6	256	288
HmFEv(256,15,3,16)	83.1	14.2	240	488
MQDSS-31-64 [a]	0.072	0.064	256	327616
ECDSA(NIST P256)	0.064	0.096	256	512
Ed25519	0.032	0.064	256	512
RSA-2048 [b]	0.256	2.048	2048	2048
RSA-3072	0.384	3.072	3072	3072

[a] [CHR+16]
[b] 112-bit security.

5.1 The benchmarks

We list the results of benchmarking in Tab. 11. Our implementations of MPKCs[5] were tested in the following environment:

- CPU: Intel XEON E3-1245 v3 (haswell) @ 3.40GHz, turbo boost disabled.
- memory: 32 GB ECC.
- OS: ubutnu 1604, Linux version 4.4.0-78-generic.
- gcc: 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1 16.04.4).

All other benchmarks are tested under the same Intel haswell architecture. We focused on optimizing the signing and verifying processes which are the most commonly used functions in signature systems.

The results show all MPKCs are indeed very efficient for verifying signatures(public map) in general. For generating signatures, we can observe the Rainbow over $\mathbb{F}_{256}$ are the most efficient among all schemes in comparisons. All small field MPKCs(Rainbows) are comparable with Ed25519 [BDL+11], whch is the most efficient pre-quantum signature in our comparisons. Although MPKCs over big field are slower than small field schemes, they are still comparable with commonly used RSAs at similar security level.

[5] The software for MPKC experiments can be downloaded from `https://github.com/fast-crypto-lab/mpkc-128bit` .

Table 11: Benchmark of 128-bit MPKCs on Intel Haswell Archiecture.

schemes	gen-key() M cycles	sign() k cycles	verify() k cycles
Rainbow(16,32,32,32)	1,359.7	68.1	22.8
Rainbow(31,28,28,28)	93.4	77.4	70.8
Rainbow(256,28,20,20)	328.9	47.8	18.3
PFLASH(16,96-1,64)	78.8	226.0	22.6
GUI(2,240,9,16,16,3)	484.2	4,445.4	197.6
GUI(4,120,17,8,8,2)	213.2	7,992.8	342.5
HmFEv(256,15,3,16)	201.7	1,497.8	15.7
MQDSS-31-64 [a]	1.827	8,510.6	5,752.6
ECDSA(NIST P256) [b]	0.286	377.1	901.5
Ed25519 [b]	0.066	61.0	185.1
RSA-2048 [b]	233.7	5,240.2	66.4
RSA-3072 [b]	844.4	15,400.9	119.3

[a] MQDSS [CHR$^+$16] is benchmarked on Intel Core i7-4770K (haswell) at 3.5GHz.

[b] [BL16] benchmarked ECC and RSA on Intel Xeon E3-1275 v3 (haswell) at 3.5GHz.

5.2 Alternative implementations for big-field MPKCs

For big-field MPKCs in the platforms without `PCLMULQDQ`, we show the benchmarks of our implementation in Tab. 12. The biggest difference between Tab. 11 and Tab. 12 is in the signing process. The arithmetic over big fields are accomplished by additive FFT, described in Sec. 4.2.2, in Tab. 12. The other restriction is that we have only 128-bit registers in SSE platforms in Tab. 12 but 256-bit AVX registers in Tab. 11.

Table 12: Benchmark of 128-bit big-field MPKCs on SSE instruction sets.

schemes	gen-key() M cycles	sign() k cycles	verify() k cycles
PFLASH(16,96-1,64)	3,264	763.4	29.9
GUI(2,240,9,16,16,3)	2,095	146,164.1	241.0
GUI(4,120,17,8,8,2)	406	133,157.9	346.4

6 Summary

We analyze the main components of MPKC signatures including evaluating $\mathcal{MQ}$ equations, multiplications over big finite fields, and solving linear equations. We present techniques for implementing these main components in x86 platforms using AVX2 instructions with side-channel resilience.

Beside reviewing MPKC signatures at 128-bit security level, we demonstrate the following techniques for implementing underlying components of signatures.

1. We use SIMD table lookup and log/exp tables for preventing cache-time attacks.
2. For the private evaluation of $\mathcal{MQ}$ over $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$, we generate instead of load the multiplication table with the value of multiplier and thus obtain a constant-time evaluation of $\mathcal{MQ}$ nearly as fast as a public evaluation.
3. We demonstrate how to evaluate multiplications in $\mathbb{F}_{2^m}$ where m is not a power of two, for example $\mathbb{F}_{2^{384}}$, using FFT techniques. The main ideas include building a tower field from an unusual base such as 2^{24}, or a truncated FFT algorithm. The FFT techniques accelerates the multiplications in big GF for the platforms without instructions to multiply large binary polynomials (`PCLMULQDQ`).
4. We demonstrate side-channel resilient elimination over $\mathbb{F}_{16}$ and $\mathbb{F}_{31}$ for solving systems of linear equations.

From the benchmarks, we conclude that MPKC signatures remain competitive speedwise under crypto-safe requirements in current mainstream instruction sets.

References

[AH74] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[BBG07] Côme Berbain, Oliver Billet, and Henri Gilbert. Efficient implementations of multivariate quadratic systems. In Eli Biham and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2007.

[BC14] Daniel J. Bernstein and Tung Chou. Faster binary-field multiplication and faster binary-field macs. In Antoine Joux and Amr M. Youssef, editors, *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, volume 8781 of *Lecture Notes in Computer Science*, pages 92–111. Springer, 2014.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastian Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2013. Document ID: e801a97c500b3ac879d77bcecf054ce5, `http://cryptojedi.org/papers/#mcbits`.

[BD08] Johannes Buchmann and Jintai Ding, editors. *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*. Springer, 2008.

[BDL$^+$11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[BFSY05] M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. In P. Gianni, editor, *MEGA 2005 Sardinia (Italy)*, 2005.

[BGP06] Côme Berbain, Henri Gilbert, and Jacques Patarin. QUAD: A practical stream cipher with provable security. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2006.

[BL16] Daniel J. Bernstein and Tanja Lange. eBACS: Ecrypt benchmarking of cryptographic systems. `http://bench.cr.yp.to`, July 2016. Accessed May 10, 2017.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2006.

[BPS08] Olivier Billet, Jacques Patarin, and Yannick Seurin. Analysis of intermediate field systems, 2008. talk at the First International Conference on Symbolic Computation and Cryptography (SCC 2008), Beijing.

[Can89] David G. Cantor. On arithmetical algorithms over finite fields. *J. Comb. Theory Ser. A*, 50(2):285–300, March 1989.

[CCC$^+$08] Anna Inn-Tung Chen, Chia-Hsin Owen Chen, Ming-Shing Chen, Chen-Mou Cheng, and Bo-Yin Yang. Practical-sized instances of multivariate PKCs: Rainbow, TTS, and ℓIC-derivatives. In Buchmann and Ding [BD08], pages 95–108.

[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, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2009.

[CCY13] Ming-Shing Chen, Chen-Mou Cheng, and Bo-Yin Yang. Raidq: A software-friendly, multiple-parity raid. In *USENIX HotStorage*, 2013.

[CHR$^+$16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass *MQ* -based identification to *MQ* -based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd*

International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II, volume 10032 of *Lecture Notes in Computer Science*, pages 135–165, 2016.

[CJL$^+$16] L. Chen, S. Jordan, Y.K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. `https://doi.org/10.6028/NIST.IR.8105`, 2016.

[CKPS00] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Bart Preneel, ed., Springer, 2000. Extended Version: `http://www.minrank.org/xlfull.pdf`.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[CSTY15] Ming-Shing Chen, Daniel Smith-Tone, and Bo-Yin Yang. pFLASH - secure asymmetric signatures on smart cards. `http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session3-smith-tone-paper.pdf`, 2015. NIST Lightweight Cryptography Workshop 2015.

[DDY$^+$08] Jintai Ding, Vivien Dubois, Bo-Yin Yang, Chia-Hsin Owen Chen, and Chen-Mou Cheng. Could SFLASH be repaired? In Luca Aceto, Ivan Damgard, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 691–701. Springer, 2008. E-Print 2007/366.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *Conference on Applied Cryptography and Network Security — ACNS 2005*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2005.

[DY13] Jintai Ding and Bo-Yin Yang. Degree of regularity for hfev and hfev-. In Philippe Gaborit, editor, *PQCrypto*, volume 7932 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2013.

[DYC$^+$08] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen, and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of rainbow. In *Applied Cryptography and Network Security*, volume 5037 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2008. cf. `http://eprint.iacr.org/2008/108`.

[Fau02] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*, pages 75–83. ACM Press, July 2002.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979. ISBN 0-7167-1044-7 or 0-7167-1045-5.

[GJS06] Louis Granboulan, Antoine Joux, and Jacques Stern. Inverting HFE is quasipolynomial. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2006.

[GM10] Shuhong Gao and Todd D. Mateer. Additive fast fourier transforms over finite fields. *IEEE Trans. Information Theory*, 56(12):6265–6272, 2010.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In Gary L. Miller, editor, *STOC*, pages 212–219. ACM, 1996.

[Har09] David Harvey. A cache-friendly truncated fft. *Theoretical Computer Science*, 410(27):2649 – 2658, 2009.

[KS98] Erich Kaltofen and Victor Shoup. Subquadratic-time factoring of polynomials over finite fields. *Math. Comput.*, 67(223):1179–1197, 1998.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem. In *Advances in Cryptology — CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 19–30. Michael Wiener, ed., Springer, 1999. `http://www.minrank.org/hfesubreg.ps` or `http://citeseer.nj.nec.com/kipnis99cryptanalysis.html`.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. Novel polynomial basis and its application to reed-solomon erasure codes. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 316–325. IEEE Computer Society, 2014.

[LLY08] Feng-Hao Michael Liu, Chi-Jen Lu, and Bo-Yin Yang. Secure PRNGs from specialized polynomial maps over any GF(q). In Buchmann and Ding [BD08], pages 95–106.

[Mat11] T. D. Mateer. Truncated fast fourier transform algorithms. In *2011 Digital Signal Processing and Signal Processing Education Meeting (DSP/SPE)*, pages 78–83, Jan 2011.

[oST16] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, 2016. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf`.

[Pat96] Jacques Patarin. Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of asymmetric algorithms. In *Advances in Cryptology — EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Ueli Maurer, ed., Springer, 1996. Extended Version: `http://www.minrank.org/hfe.pdf`.

[PBB10] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. Selecting parameters for the rainbow signature scheme. In Nicolas Sendrier, editor, *PQCrypto*, volume 6061 of *Lecture Notes in Computer Science*, pages 218–240. Springer, 2010.

[PCD$^+$17] Albrecht Petzold, Ming-Shing Chen, Jintai Ding, , and Bo-Yin Yang. Mhfev - an efficient multivariate signature scheme. In Tanja Lange and Tsuyoshi Takagi, editors, *Lecture note on computer science*, page To appear. PQCrypto, Springer, July 2017.

[PCG01a] Jacques Patarin, Nicolas Courtois, and Louis Goubin. Flash, a fast multivariate signature algorithm. In C. Naccache, editor, *Progress in cryptology, CT-RSA*, volume 2020 of *LNCS*, pages 298–307. Springer, 2001.

[PCG01b] Jacques Patarin, Nicolas Courtois, and Louis Goubin. QUARTZ, 128-bit long digital signatures http://www.minrank.org/quartz/. In C. Naccache, editor, *Progress in cryptology, CT-RSA*, volume 2020 of *LNCS*, pages 282–297. Springer, 2001.

[PCY$^+$15] Albrecht Petzoldt, Ming-Shing Chen, Bo-Yin Yang, Chengdong Tao, and Jintai Ding. Design principles for hfev- based multivariate signature schemes. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 311–334. Springer, 2015.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997.

[SSH11] Koichi Sakumoto, Taizo Shirai, and Harunaga Hiwatari. Public-key identification schemes based on multivariate quadratic polynomials. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 706–723. Springer, 2011.

[WS16] Bas Westerbaan and Peter Schwabe. Solving binary $\mathcal{MQ}$ with grover's algorithm. In Claude Carlet, Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Advanced Cryptography Engineering*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2016. Document ID: 40eb0e1841618b99ae343ffa073d6c1e, `http://cryptojedi.org/papers/#mqgrover`.

[YC05] Bo-Yin Yang and Jiun-Ming Chen. Building secure tame-like multivariate public-key cryptosystems: The new TTS. In *ACISP 2005*, volume 3574 of *Lecture Notes in Computer Science*, pages 518–531. Springer, July 2005.

[YCBC07] Bo-Yin Yang, Owen Chia-Hsin Chen, Daniel J. Bernstein, and Jiun-Ming Chen. Analysis of `QUAD`. In Alex Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 290–307. Springer, 2007.