Small Odd Prime Field Multivariate PKCs

Anna Inn-Tung Chen[1], Ming-Shing Chen[2], Tien-Ren Chen[2], Chen-Mou Cheng[1], Jintai Ding[3], Eric Li-Hsiang Kuo[2], Frost Yu-Shuang Li[1], and Bo-Yin Yang[2]

[1] School of EECS, National Taiwan University, Taipei, Taiwan,
{anna1110,doug,frost}@crypto.tw
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan,
{mschen,trchen,lorderic,by}@crypto.tw
[3] Dept. of Math. Sciences, U. of Cincinnati, Cincinnati, Ohio, USA,
ding@math.uc.edu

Abstract. We show that Multivariate Public Key Cryptosystems (MPKCs) *over fields of small odd prime characteristic*, say 31, can be highly efficient. Indeed, at the same design security of 2^{80} under the best known attacks, odd-char MPKC is generally faster than prior MPKCs over $\mathbb{F}_{2^k}$, which are in turn faster than "traditional" alternatives.
This seemingly counter-intuitive feat is accomplished by exploiting the comparative over-abundance of small integer arithmetic resources in commodity hardware, here embodied by SSE2 or more advanced special multimedia instructions on modern x86-compatible CPUs.
We explain our implementation techniques and design choices in implementing our chosen MPKC instances modulo small a odd prime. The same techniques are also applicable in modern FPGAs which often contains a large number of multipliers.

Keywords: Gröbner basis, multivariate public key cryptosystem, TTS, rainbow, HFE, ℓIC, SSE2, vector instructions

1 Introduction

MPKCs (multivariate public key cryptosystems) [16, 42] are PKCs whose public keys are multivariate polynomials in many variables over a small field $\mathbb{K} = \mathbb{F}_q$.

$$\mathcal{P} : \mathbf{w} = (w_1,\, w_2, \ldots,\, w_n) \in \mathbb{K}^n \mapsto \mathbf{z} = (p_1(\mathbf{w}),\, p_2(\mathbf{w}), \ldots,\, p_m(\mathbf{w})) \in \mathbb{K}^m.$$

Here $p_1, p_2, \ldots$ are polynomials (in practice always quadratic for speed). It is often mentioned that trying to invert $\mathcal{P}$ directly is equivalent to an instance of

Problem $\mathcal{MQ}(q; n, m)$: Solve the system $p_1(\mathbf{x}) = p_2(\mathbf{x}) = \cdots = p_m(\mathbf{x}) = 0$, where each p_i is a quadratic in $\mathbf{x} = (x_1, \ldots, x_n)$. All coefficients and variables are in $\mathbb{K} = \mathbb{F}_q$, the field with q elements.

Which is a well-known difficult problem [27] and it is usually conjectured that a random $\mathcal{MQ}$ instance is hard. Of course, a random $\mathcal{P}$ would not be invertible by the legitimate user either. So in practice, the design involves two affine maps $S : \mathbf{w} \mapsto \mathbf{x} = \mathrm{M}_S\mathbf{w} + \mathbf{c}_S$, and $T : \mathbf{y} \mapsto \mathbf{z} = \mathrm{M}_T\mathbf{y} + \mathbf{c}_T$, and an efficiently inverted map $\mathcal{Q} : \mathbf{x} \mapsto \mathbf{y}$, and let $\mathcal{P} = T \circ \mathcal{Q} \circ S$. The public key consists of the polynomials in $\mathcal{P}$. $\mathcal{P}(0)$ is always taken to be zero. The private key is $\mathrm{M}_s^{-1}, \mathbf{c}_s, \mathrm{M}_T^{-1}, \mathbf{c}_T$, and whatever information that determines the *central map* $\mathcal{Q}$.

1.1 Questions

MPKCs are always touted as (a) potentially surviving of future attacks with quantum computers, and (b) faster than "traditional" competition. In 2003, SFLASH was a finalist for the NESSIE project signatures, recommended for speed.

However, the playing field has been changing rapidly. In the seminal papers more than a decade ago, it is pointed out that eventually it would be the memory latency and bandwidth to become the bottleneck of the performance of a microprocessor [8, 43]. This same trend has been observed by MPKC implementers. When MPKCs were initially proposed two decades ago [32, 38], commodity CPUs compute a 32-bit integer product maybe every 15–20 cycles. When NESSIE called for primitives, x86 CPUs could compute one 64-bit product every 3 (Athlon) to 10 (Pentium 4) cycles. A 64-bit AMD Opteron today has a big pipelined multiplier that outputs one 128-bit integer product every 2 cycles plus other multipliers each capable of outputing one 64-bit integer every cycle.

In stark contrast, cost of multiplying in $\mathbb{F}_{256}$ or $\mathbb{F}_{16}$ have not changed much per cycle. An '80s vintage 8051 microcontroller from Intel multiplies in $\mathbb{F}_{256}$ in about a dozen instruction cycles using three table lookups. The AMD Opteron today has the same latency, although a somewhat higher throughput. Despite the fact that microprocessor manufacturers like Intel are planning carryless multiplication instructions [34] for new architectures to support polynomial multiplications in $\mathbb{F}_2$, the development and deployment of these new instructions still lag several generations behind compared with their integer counterparts.

This striking disparity came about because memory access speed increased at a snail's pace compared to the number of gates available, which had been doubling every 18 months ("Moore's Law") for two decades. Now the width of a typical ALU is 64 bits, vector units are everywhere, and even FPGAs have dozens of multipliers built-in. Commodity hardware has never been more friendly to RSA and ECC – the deck seems stacked considerably against MPKCs.

Furthermore, we now understand multivariates and equation-solving much better. The speed champions TTS/4 and SFLASH were much faster signature schemes than traditional competition using RSA and ECC [1, 13, 44]. However, both these two instances have been broken since [20, 21]. Today TTS/7 and 3IC-p still seem to do fine [10], but the impending doom of SHA-1 [40] will force longer message digests and slower MPKCs, but leaving RSA untouched.

Of course, multivariates still represent a future-proofing effort, however, it is imperative to address the obvious question: **Can we design MPKCs that are more efficient on modern commodity hardware?**

1.2 Our Answers

Algebra tells us that q can be any prime power. However, in almost all proposed multivariates to date, q is a power of two so that addition can be easily accomplished by the logical XOR (exclusive-or) operation.

By letting q be a relatively small odd prime, in most of our tests here 31, instead of a small power of 2, we achieve higher efficiency on current-generation

CPUs. MPKCs using mod-q arithmetic can become a few times faster by using vector instructions that are present in almost every modern CPU.

All x86-64 (AMD64 or Intel64) CPUs today on a PC, either Intel's Core 2 or AMD's K8 (Athlon64) and K10 (Phenom) CPUs, support the SSE2 vector instruction set. SSE2, which can pack eight 16-bit integer operands in its special 128-bit registers and access them together, can thus dispatch 8 simultaneous integer operations per cycle. We hasten to point out that this is never really 8 times the speed since still need to set up, to move data, to fit data appropriately and so on. Especially when the size of the vector is not divisible by 8, there is often some extra cost in space and time. But our conclusion is:

Switching to $\mathbb{F}_q$ for MPKCs instead of $\mathbb{F}_{256}$ or $\mathbb{F}_{16}$ enables us to take advantage of modern hardware. Even considering the need to convert between base-q and binary, schemes over $\mathbb{F}_{31}$ is usually faster than an equivalent scheme over $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ at the same design security.

Note that we are not really discussing the security of MPKCs, which is so complicated that it may take a book and more. We simply aim to show that today, there is no longer as much need to stick to $\mathbb{F}_{2^k}$.

1.3 Previous Work

Proposed MPKCs There has been a great many. Most basic ideas remaining — TTS, Rainbow, ℓIC-p, even oil-and-vinegar [14,15,20,36] can be implemented over small odd-prime fields equally as well as over $\mathbb{F}_{2^k}$. For some (e.g., ℓIC-derivatives) an odd-char version is less convenient, but not impossible.

Note that C^* and HFE in an odd char field was mentioned in [42] and recently brought up again in [18], but not much researched so far.

Prior Attacks over $\mathbb{F}_{2^k}$ Nearly as many attacks and cryptanalytical techniques exist as there are variations in MPKCs; most attacks applies equally to an MPKC implemented over $\mathbb{F}_{2^k}$ or over a small prime field. Some attacks [7,12,24] do better for smaller characteristics, or less well over an odd $\mathbb{F}_q$ than $\mathbb{F}_{2^k}$. Significantly, some direct solving attacks by Gröbner basis methods such as [22, 23] are in these last category. *Thus, MPKCs over odd $\mathbb{F}_q$ are of independent interest.*

Implementation over $\mathbb{F}_{2^k}$ Current state of the art is given in [2, 10]. The former discusses primarily the public map and include special bit-slicing implementations for quadratic maps (i.e., public map) over $\mathbb{F}_{2^k}$. The latter discusses current security criteria, parameter choices and give a few tricks to use for private maps, such as bitslicing to solve linear systems.

To implement "big-field" MPKCs we also need to implement arithmetic and algebraic operations in composite fields. I.e., powers, multiplications, inversions and square roots and so on. Most techniques work similarly in any $\mathbb{F}_q$, except

- Some arithmetic issues differ due to modulo-q considerations [5,30].
- Square roots may require special methods [4] such as Tonelli-Shanks.

– Cantor-Zassenhaus [9] is usually the algorithm of choice for equation-solving in odd-char fields (required for HFE-like schemes), which typically is simpler and faster than Berlekamp [3] that is used for $\mathbb{F}_{2^k}$.

1.4 Future Work

Today's FPGAs have many built-in multipliers and IP (intellectual property) for good integer multipliers are common for ASICs (Application-Specific Integrated Circuits). One example of using the multipliers in FPGAs for PKC can be found in [31]. Hence our results can easily carry over to FPGAs as well as any other specialized hardware with multiple small multipliers. There are also a variety of massively parallel processor architectures on the rise, such as NVIDIA's and AMD/ATI's graphics processors, as well as Intel's upcoming Larrabee [39]. The comparisons herein must of course be re-evaluated with each new instruction set and new silicon, but we believe that the general trend stands on our side.

2 Our Tools

We aim to be as portable as possible, hence we programming everything in `g++`, the `C++` compiler in the GNU Compiler Collection (`gcc`). Since `g++-4.3` can use either Intel's or its own naming conventions for x86 and x86-64 compatible processor-specific intrinsics or inlined assembly, we choose to use the Intel names. For the most part, `icc` (Intel's own `C++` compiler, version 10) gives similar results.

2.1 The SSE2 instruction Set

SSE2 stands for Streaming SIMD Extensions 2, where SIMD in turn stands for Single Instruction Multiple Data. I.e., doing the same action on many operands.

All Intel CPUs since the Pentium 4 and all AMD CPUs since the K8 (Opteron and Athlon 64) supports SSE2. The SSE2 instruction set operates on 16 architectural 128-bit registers called `xmm` registers. Most relevant to us is SSE2's integer operations, which work on `xmm` registers as packed 8-, 16-, 32- or 64-bit operands.

SSE2 integer instructions are not easily to comprehend. The instruction set is arcane and highly non-orthogonal. To summarize, there are the following:

Load/Store: To and from `xmm` registers from memory (both aligned and unaligned) and traditional registers (using the lowest unit in the `xmm` register and zeroing all higher units on a load).

Reorganize Data: A multi-way 16- and 32-bit move called Shuffle, and Packing/Unpacking on vector data of different densities.

Logical: AND, OR, NOT, XOR; Shift (packed operands of 16-, 32- and 64-bits) Left, Right Logical and Right Arithmetic (copies the sign bit); Shift entire `xmm` register right and left (bytewise only).

Arithmetic: Add/subtract on 8-, 16-, 32- and 64-bits (including "saturating" versions); multiply of 16-bit (high and low word returns, signed and unsigned, and fused multiply-adds) and 32-bits unsigned; max/min (signed 16-bit, unsigned 8-bit); unsigned averages (8-/16-bit); sum-of-differences 8-bits.

2.2 Other Vector Instruction Sets

The SSE3 instruction set does not do much for us, but the SSSE3 (Supplementary Streaming SIMD Extensions) adds some instructions that assists with our programming. The biggest drawback to SSE2, as explained above, is the awkwardness with inner products, i.e., no operation between elements inside a 8-wide vector. One instruction that comes in very useful is `PALIGNR` ("packed align right", really a 32-byte shift). To execute "`PALIGNR xmm (`i`), xmm (`j`),` k", We shift `xmm` (j) right by k bytes, and insert the k rightmost bytes of `xmm` (i) in the space vacated by the shift to the left. The result is placed in `xmm` (i).

Another useful intrinsic is `PHADDW` ("packed horizontal add word"). If destination register `xmm` (i) starts out as $(x_0, x_1, \ldots, x_7)$, the source register `xmm` (j) as $(y_0, y_1, \ldots, y_7)$, then after "`PHADDW xmm` (i), `xmm` (j)", `xmm` (i) will hold

$$(x_0 + x_1,\ x_2 + x_3,\ x_4 + x_5,\ x_6 + x_7,\ y_0 + y_1,\ y_2 + y_3,\ y_4 + y_5,\ y_6 + y_7).$$

We can check that if there are eight vectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_7$ seven invocations of `PHADDW` can obtain $\left(\sum_j v_j^{(0)}, \sum_j v_j^{(1)}, \ldots, \sum_j v_j^{(7)}\right)$ arranged in the right order. Note that for both `PALIGNR` and `PHADDW` the source register `xmm` (j) can be replaced by an 16-byte-aligned memory location.

Note: For introduction to optimizing x86-64 see [33]. Neither Intel's SSE4 instructions nor AMD's preemptive strike of what they call SSE5 are necessarily useful to us as most effects of later instructions (e.g., `PHADDW` and `PALIGNR`) can be achieved at a small cost using SSE2 instructions, which are remarkably versatile. One exception is `PSHUFB` that does a 16-byte permute or table lookup.

2.3 A Comparison Between Current CPUs

Intel's Core architecture, with SSSE3 and larger/faster L2 caches, have received rave reviews and are remarkably cost-effective — but especially for floating point. Even the new K10s from AMD don't have SSSE3, but they are nicely optimized for integer programming. Furthermore, one might simply use `MOVDQU` (move dbl. quadword unaligned) to do unaligned loads/stores on AMD where one need `PALIGNR` for Intel, due to the following subtle trap: *One time in four* on Intel CPUs a random unaligned load would fall across a cache line for a penalty of ~ 250 cycles. The SSE3 instruction `LDDQU` (load dbl. quadword unaligned) issues *two loads* to avoid line faults but does not work very well on current Core 2's.

In general, overall Intel's Core 2 CPUs and AMD's K10 CPUs remain close if not evenly-matched by most estimates. But Intel's newer 45nm process C2's excel particularly at vector code, since it is basically Intel's own dominion.

3 General Implementation Techniques mod q

The most important optimization is *avoid unnessary modulo operations.* Careless `C++` programming will lead to frequent taking of remainders mod q. We try to

delay this as much as possible and hence need to track carefully the size of the operands. SSE2 uses 16- or 32-bit operands for most of its integer vector operations. In general, we would like to use 16-bit integers (either signed or unsigned) in our implementation because it makes for more parallel processing.

3.1 Conversion Between Bytes and Blocks in mod q

The division instruction is always particularly time-consuming, so by now everyone knows about doing division by multiplication for large integers [30]:

Proposition 1. *If M satisfies $2^{n+\ell} \le Md \le 2^{n+\ell} + 2^{\ell}$, then*

$$\left\lfloor \frac{X}{d} \right\rfloor = \left\lfloor 2^{-\ell} \left\lfloor \frac{XM}{2^n} \right\rfloor \right\rfloor = \left\lfloor 2^{-\ell} \left(\left\lfloor \frac{X * (M - 2^n)}{2^n} \right\rfloor + X \right) \right\rfloor \tag{1}$$

for $0 \le X < 2^n$, where $\gg$ denotes the right shift operation. Note that the second equality in Eq. 1 is to take care of the common situation where $M > 2^n$.

Most of the time there is a machine instruction returning the "top half of the product of n-bit unsigned integers x and y" which achieves $\lfloor \frac{xy}{2^n} \rfloor$ relatively easily. However, neither `g++` nor Intel `C++` have such intrinsics. For an unsigned integer x in $[0; 2^{64} - 1]$, we might use something like the inline assembly code in Fig. 1 to access the requisite extra-precision arithmetic to achieve the effect of

$$Q = \left\lfloor \frac{1}{32} \left(\left\lfloor \frac{5950562604422436 01\, x}{2^{64}} \right\rfloor + x \right) \right\rfloor = x \text{ div } 31, \quad R = x - 31\, Q.$$

One problem with the mod q is that the conversion between binary data and base-q data can never be exact. Suppose the public map is $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$. For digital signatures, we need to have $q^m > 2^{\ell}$, where ℓ is the length of the hash, so that all hash digests of the appropriate size fit into $\mathbb{F}_q$ blocks; for encryption schemes that pass an ℓ-bit session key, we need $q^n > 2^{\ell}$.

It so happens that quadword (8-byte) unsigned integers in $[0; 2^{64} - 1]$ fit decently into 13 blocks in $\mathbb{F}_{31}$. So to convert a 20-byte hash digest, we might do two 8-byte-to-$(\mathbb{F}_{31})^{13}$ conversions, then convert the remaining 4 bytes into 7 $\mathbb{F}_{31}$ blocks for a total of 33. So to transfer 128-, 192-, and 256-bit AES keys, we need at least 26, 39, and 52 $\mathbb{F}_{31}$ blocks respectively.

Once we have expressed a digital signature, a ciphertext, or even keys in $\mathbb{F}_q$-blocks, conversion to binary can be more wasteful as long as it is injective and convenient. In our testing, we opted for a very simple packing that fits only three $\mathbb{F}_{31}$ blocks in every 16-bit word. Thus, a digital signature scheme whose public key is $(\mathbb{F}_{31})^{53} \to (\mathbb{F}_{31})^{36}$ might take a 20-byte hash, convert into 33 $\mathbb{F}_{31}$ blocks, pad to 36, take the inverse of the public map to get 53 $\mathbb{F}_{31}$ blocks, and then pack to 36 bytes, which will be our signature.

3.2 Multiple Simultaneous mod q Operations as a Vector

We often need to take many remainders modulo q at the same time. Using divide instructions is very time-wasting and this is eminently suitable for parallelization.

SSE2 does have instructions returning the upper word of a 16-bit-by-16-bit product. However, since there are no carrying facilities, it is difficult to guarantee a range of size q, especially for a general q. It is then important to realize that *we do not need always the tightest range*. Minus signs are ok, it is only required that the absolute values are relatively small to avoid costly mods. The following is guaranteed to return a value $y \equiv x \pmod{q}$ such that $|y| \leq q$ for general b-bit word arithmetic:

$$y = x - q \cdot \texttt{IMULHI}b \left(\left\lfloor \frac{2^b}{q} \right\rfloor, \left(x + \left\lfloor \frac{q-1}{2} \right\rfloor \right) \right), \tag{2}$$

where `IMULHI`b returns *the upper half in a signed product of two b-bit words*, for $-2^{b-1} \leq x \leq 2^{b-1} - (q-1)/2$. For specifically $q = 31$ and $b = 16$, we can do better and get a $y \equiv x \pmod{31}, -16 \leq y \leq 15$, for any $-32768 \leq x \leq 32752$:

$$y = x - 31 \cdot \texttt{IMULHI16}\ (2114, x + 15), \tag{3}$$

Here `IMULHI16` is implemented via the intel intrinsic of `__mm_mulhi_epi16`. How to multiply a vector by given q needs to be analyzed for individual architectures.

We implemented a special version for input and output in $\mathbb{F}_{31}$, where we normalize the result to a principal value between 0 and 30 from y above:

$$y' = y - 31\ \&\ (y \ggg 15), \tag{4}$$

where & is the logical `AND` and $\ggg$ arithmetically shifts in the sign bit.

3.3 Multiplying a Matrix M to a Vector v

Core 2 and newer Intel CPUs have SSSE3 and can add horizontally within an `xmm` register. On a Core 2, the matrix can be simply stored as rows. Each row is multiplied componentwise to the vector. Then we use `PHADDW` to add horizontally and arrange the elements at the same time. Surprisingly, the convenience of having `PHADDW` available only makes a $< 10\%$ difference (cf. Sec. 2.2).

Using just SSE2, it is advisable to store M column-wise and treat the matrix-to-vector product as taking a linear combination of the column vectors. Each `short int` in **v** is copied 8 times into every 16-bit field in an `xmm` register using an `__mm_set1` intrinsic, which without SSSE3 takes three data-moving instructions (using shuffles), but still avoids the penalty for accessing the L1 cache. Multiply this register into one column of M, 8 components at a time, and accumulate.

One particular optimization for long rows and small modulus is given below.

3.4 Evaluation of Public Polynomials

Normally we write the public map in the following manner:

$$z_k = \sum_i P_{ik} w_i + \sum_i Q_{ik} w_i^2 + \sum_{i<j} R_{ijk} w_i w_j = \sum_i w_i \left[P_{ik} + Q_{ik} w_i + \sum_{i<j} R_{ijk} w_j \right].$$

But often it is better to compute a vector $\mathbf{c}$ with contents $[(w_i)_i, (w_i w_j)_{i \le j}]^T$, then $\mathbf{z}$ as a product of a $m \times n(n+3)/2$ matrix times and $\mathbf{c}$. Normally we multiply as in Sec. 3.3, with coefficients grouped 8 at a time according to k. But often we may exploit a *packed multiply-add word to double word* instruction that computes $(x_0y_0 + x_1y_1,\ x_2y_2 + x_3y_3,\ x_4y_4 + x_5y_5,\ x_6y_6 + x_7y_7)$ given $(x_0, \ldots, x_7)$ and $(y_0, \ldots, y_7)$. We interleave one `xmm` with two monomials (32-bit load plus a single `__mm_set1` call), load a 4×2 block in another, `PMADDWD`, and *continue in 32-bits until the eventual reduction mod q*. This messier way saves a few mod-q's.

The Special Case of $\mathbb{F}_{31}$ We also pack keys (cf. Sec. 3.1) so that the public key is roughly $mn(n+3)/3$ bytes, which holds $mn(n+3)/2$ $\mathbb{F}_{31}$ entries. For $\mathbb{F}_{31}$, we go to special pains to avoid writing the data to memory and execute the public map on-the-fly as we unpack (to avoid cache effects). It turns out that it does not slow things down too much. Further, we can do the messier 32-bit mod-q reduction (note: no `__mm_mulhi_epi32`!) via shifts as $2^5 = 1 \mod 32$.

3.5 Inverting a Vector of Elements

To invert one element in $\mathbb{F}_q$, we usually use a lookup table. In some cases, we need to invert many $\mathbb{F}_q$ elements at the same time. Looking up 8 elements in a table usually don't take that long, but getting entries out of and into `xmm` registers can be troublesome. Thus, we can elect to use a $(q-2)$-th power ("patched inverse") to get the inverse for a vector of elements. Taking into account the possibility of overflowing, we do this to get a 29-th power in $\mathbb{F}_{31}$ using `short int`

$y = x*x*x \mod 31$; $y = x*y*y \mod 31$; $y = y*y \mod 31$; $y = x*y*y \mod 31$.

3.6 Solving Systems of Linear Equations

Systems of linear equations are involved directly with TTS (and Rainbow), and indirectly in the other schemes through taking inverses. Normally, one runs a Gaussian elimination, which is sped up also by SSE2.

One must notice here that during a Gaussian elimination, one need to do frequent modular reductions, which rather slows things down from the speed that you might otherwise expect. To elaborate a little more, let's say that we have an augmented matrix $[A|\mathbf{b}]$ modulo 31. For ease of doing elementary row operations, we naturally store the matrix row-first. Now suppose we have done elimination on the first column. Each entry in the remaining columns will now

be be of size up to about 1000, or at least up to around 250 if we use signed representations. So, to do the second column of eliminations, we need to reduce that column mod 31 before we can look up the correct coefficients. Given that and the results from implementations in Sec. 3.5, we might as well reduce the entire matrix, especially when the size is not large.

3.7 Choosing $q = 31$

Clearly, we need to avoid having too large q (too many reductions mod q) and too small q (too large an array). The choice of $q = 31$ seems a good compromise, since it also allows us several convenient tower fields and easy packing conversions.

4 Tower Fields for Big-Field Schemes

In a "big-field" or "two-field" scheme, we need to handle $\mathbb{L} = \mathbb{F}_{q^k} \cong \mathbb{F}_q[t]/\left(p(t)\right)$, where p is an irreducible polynomial of degree k. Of course, any irreducible p results in an isomorphic representation of the same field, but often it is of paramount importance to pick a p that makes for easier computations. It would be convenient if $p(t) = t^k - a$ for a small positive a. When $k|(q-1)$ and in a few other cases such a suitable a can be found.

When p is in a convenient form, the map $X \mapsto X^q$ in $\mathbb{L}$, as a precomputable linear map over $\mathbb{K} = \mathbb{F}_q$ becomes nearly trivial, and multiplication/division/inverses become much easier. Some example timings for a tower field in Tab. 5.

4.1 Multiplications

When we have $\mathbb{F}_{q^k} \cong \mathbb{F}_q[t]/(t^k - a)$, we can see that

$$
\begin{aligned}
&\left(x_0 + x_1 t + \cdots + x_{k-1} t^{k-1}\right) \cdot \left(y_0 + y_1 t + \cdots + y_{k-1} t^{k-1}\right) \\
&= t^{k-1}(x_0y_{k-1}+ \; x_1y_{k-2}+ \; x_2y_{k-3}+ \cdots+ \; x_{k-3}y_2+ \quad x_{k-2}y_1+ \quad x_{k-1}y_0) \\
&+ t^{k-2}(x_0y_{k-2}+ \; x_1y_{k-3}+ \; x_2y_{k-4}+ \cdots+ \; x_{k-3}y_1+ \quad x_{k-2}y_0+ \; ax_{k-1}y_{k-1}) \\
&+ t^{k-3}(x_0y_{k-3}+ \; x_1y_{k-4}+ \; x_2y_{k-5}+ \cdots+ \; x_{k-3}y_0+ \; ax_{k-2}y_{k-1}+ \; ax_{k-1}y_{k-2}) \\
&+ \quad \cdots \\
&+ \quad t^2(\quad x_0y_2+ \quad x_1y_1+ \quad x_2y_0+ \ldots+ \; ax_{k-3}y_5+ \quad ax_{k-2}y_4+ \quad ax_{k-1}y_3) \\
&+ \quad t(\quad x_0y_1+ \quad x_1y_0+ \; ax_2y_{k-1}+ \ldots+ \; ax_{k-3}y_4+ \quad ax_{k-2}y_3+ \quad ax_{k-1}y_2) \\
&+ \quad (\quad x_0y_0+ \; ax_1y_{k-1}+ \; ax_2y_{k-2}+ \ldots+ \; ax_{k-3}y_3+ \quad ax_{k-2}y_2+ \quad ax_{k-1}y_1)
\end{aligned}
$$

The straightforward way is to take each x_i, copy it 8 times; and multiply by the correct y_i's using `PMULLW`, then shift the results by the appropriate distances using `PALIGNR` (if SSSE3 is available) and unaligned load/stores or shifts (if not). Each option may be better or worse depending on the architecture and compiler.

For inconvenient cases like when $k = 9$, we need to tune the code somewhat to the occasion. As an example, for $k = 9$, we would multiply the $\mathbf{x}$-vector by y_8 and the $\mathbf{y}$-vector by x_8, leaving the rest in a convenient 8×8 pattern for access.

For very large fields, we can use Karatsuba [35] or more advanced multiplications. E.g., treat $\mathbb{F}_{31^{30}}$ as $\mathbb{F}_{31^{15}}[u]/(u^2-t)$, where $\mathbb{F}_{31^{15}} = \mathbb{F}_{31}[t]/(t^{15}-3)$. Then $(a_1u+a_0)(b_1u+b_0) = [(a_1+a_0)(b_1+b_0)-a_1b_1-a_0b_0]u+[a_1b_1t+a_0b_0]$; similarly, we treat $\mathbb{F}_{31^{54}}$ as $\mathbb{F}_{31^{18}}[u]/(u^3-t)$, where $\mathbb{F}_{31^{18}} = \mathbb{F}_{31}[t]/(t^{18}-3)$. Then

$$(a_2u^2+a_1u+a_0)(b_2u^2+b_1u+b_0) = [(a_2+a_0)(b_2+b_0)-a_2b_2-a_0b_0+a_1b_1]\,u^2$$
$$+\,[(a_1+a_0)(b_1+b_0)-a_1b_1-a_0b_0+ta_2b_2]\,u+[t\,((a_2+a_1)(b_2+b_1)-a_1b_1-a_2b_2)+a_0b_0]\,.$$

4.2 Squaring

Often people say $\mathbf{S} = \mathbf{0.8M}$. Here we simply skip some of the iterations in the loops used for the multiplication, and weigh some of the other iterations double. Due to architectural effects S/M is *anywhere from as high as* 0.92 *to as low as* 0.75 for fields in the teens of $\mathbb{F}_{31}$ blocks, somewhat randomly, which carries from any implementation over up to any tower field built over it.

4.3 Square and Other Roots

To compute square roots, there are many ways [4] today. For field sizes $q = 3 \pmod 4$, it is easy to compute the square root in $\mathbb{F}_q$ via $\sqrt{y} = \pm y^{\frac{q+1}{4}}$. Here we implement the Tonelli-Shanks method for $1 \pmod 4$ field sizes, as working with a fixed field we can include precomputated tables with the program "for free". To recap, let's assume that we want to compute square roots in the field $\mathbb{L}$, where $|\mathbb{L}|-1 = 2^k a$, with a being odd.

0. Compute one of the primitive solutions of $g^{2^k} = 1$ in $\mathbb{L}$. We only need to take a random $x \in \mathbb{L}$ and compute $g = x^a$, and it is almost even money (i.e., x is a non-square) that $g^{2^{k-1}} = -1$, which means we have found a correct g. *Start with a precomputed table of* $(j,\, g^j)$ *for* $0 \le j < 2^k$.
1. We wish to compute an x such that $x^2 = y$. First compute $v = y^{\frac{a-1}{2}}$.
2. Look up in our table of 2^k-th roots $yv^2 = y^a = g^j$. If j is odd then y is a non-square. If j is even, then $x = \pm vyq^{\frac{-j}{2}}$ because $x^2 = y(yv^2g^{-j}) = y$.

Since we implemented mostly mod 31, for $\mathbb{F}_{31^k}$ taking a square root is easy when k is odd and not very hard when k is even. Example: in $\mathbb{F}_{31^9}$, we compute

$i.$ $\texttt{temp1} := (((\texttt{input})^2)^2)^2$, $ii.$ $\texttt{temp2} := (\texttt{temp1})^2 * ((\texttt{temp1})^2)^2$,
$iii.$ $\texttt{temp2} := \left[\texttt{temp2} * ((\texttt{temp2})^2)^2\right]^{31}$, $iv.$ $\texttt{temp2} := \texttt{temp2} * (\texttt{temp2})^{31}$,
$v.$ $\texttt{result} := \texttt{temp1} * \texttt{temp2} * \left((\texttt{temp2})^{31}\right)^{31}$;

which achieves the square root in $\mathbb{F}_{31^9}$, by raising to a power of

$$\tfrac{1}{4}\left(31^9+1\right) = 2^3 \cdot \left[1+2(1+2+2^2+2^3)(31+31^3+31^5+31^7)\right].$$

Similarly, in $\mathbb{F}_{31^{18}}$, the key step in taking a square root is accomplished via

$$
\begin{aligned}
&i. \quad \texttt{temp1} := \texttt{input} * (\texttt{input})^2 * (\texttt{input})^4,\\
&ii. \quad \texttt{temp2} := \texttt{input} * (\texttt{temp1})^2,\\
&iii. \quad \texttt{temp2} := \left[\left(((\texttt{temp1})^{31} * \texttt{temp2})^{31} * \texttt{temp1 * temp2 * input}\right)^{31}\right]^{31},\\
&iv. \quad \texttt{temp2} := \texttt{temp2} * ((((\texttt{temp2})^{31})^{31})^{31})^{31},\\
&v. \quad \texttt{result} := \texttt{temp1} * \texttt{temp2} * (((((((\texttt{temp2})^{31})^{31})^{31})^{31})^{31})^{31})^{31}.
\end{aligned}
$$

This raises to the power of $\frac{1}{128}\left(31^{18}-65\right)$. Note that powers of 31 are "trivial".

We should mention here that taking other (e.g. cubic or fifth) roots is, while dependent on the exact field concerned, analogous to square roots.

4.4 Multiplicative Inverses

There are several ways to do multiplicative inverses in $\mathbb{F}_{q^k}$. The classical one is an extended Euclidean Algorithm; another is to solve a system of linear equations; the last one is to invoke Fermat's little theorem and raise to the power of q^k-2.

While the extended Euclidean Algorithm is a classic, for our specialized tower fields of characteristic 31, it is slower because after the very first division the sparsity of the polynomial is lost. Solving every entry in the inverse as a variable and running a elimination is about 30% better; *even though it is counter-intuitive to compute* $X^{31^{15}-2}$ *to get* $1/X$*, it ends up fastest by a factor of* $2\times$–$3\times$*.*

One final twist: when we compute $\sqrt{X}$ and $1/X$ as high powers at the same time, we can share some exponentiations and saving 10% of the work.

4.5 Equation-Solving in a Large Field

Using an odd-prime base field let us implement Cantor-Zassenhaus, today the standard recommended algorithm for odd characteristics finite fields, for finding all solutions in $\mathbb{L}=\mathbb{F}_{q^k}$ to a univariate degree-d equation $u(X)=0$:

1. Replace $u(X)$ by $\gcd(u(X), X^{q^k}-X)$ so that u splits (factors completely) in $\mathbb{L}$. Most of the work is to compute $X^{q^k} \bmod u(X)$, which can be done by
   (a) Compute and tabulate $X^d \bmod u(X), \ldots, X^{2d-2} \bmod u(X)$.
   (b) Compute $X^q \bmod u(X)$ via square-and-multiply.
   (c) Compute and tabulate $X^{qi} \bmod u(X)$ for $i=2, 3, \ldots, d-1$.
   (d) Compute $X^{q^i} \bmod u(X)$ for $i=2, 3, \ldots, k$.
2. Compute $\gcd\left(v(X)^{(q^k-1)/2}-1, u(X)\right)$ for a random $v(X)$, where $\deg v = \deg u - 1$; half of the time we find a nontrivial factor; repeat till u is factored.

The work is normally cubic in $\mathbb{L}$-multiplications and quintic in $(d, k, \lg q)$ overall.

5 Tests

Some recent implementations of MPKCs over $\mathbb{F}_{2^k}$ are tested in [10]. We choose these more well-known schemes for comparison: TTS/Rainbow signature schemes; 3IC-p and pFLASH signature scheme; HFE encryption scheme. We have implemented the following as comparable schemes: TTS/Rainbow of comparable sizes; 3IC-p (analogue to pFLASH); HFE-based encryption schemes, all prefixed with a 0 block (which means discarding a variable). Results are tabulated in Tab. 2 and 1, with further data in Tab. 3 and 4.

5.1 Rainbow and TTS Families

Rainbow with u stages is characterized as follows [17, 20]:

- The segment structure is given by a sequence $0 < v_1 < v_2 < \cdots < v_{u+1} = n$.
- For $l = 1, \ldots, u+1$, set $S_l := \{1, 2, \ldots, v_l\}$ so that $|S_l| = v_l$ and $S_0 \subset S_1 \subset \cdots \subset S_{u+1} = S$. Denote by $o_l := v_{l+1} - v_l$ and $O_l := S_{l+1} \setminus S_l$ for $l = 1 \cdots u$.
- The central map $\mathcal{Q}$ has component polynomials $y_{v_1+1} = q_{v_1+1}(\mathbf{x})$, $y_{v_1+2} = q_{v_1+2}(\mathbf{x}), \ldots, y_n = q_n(\mathbf{x})$ — *notice unusual indexing* — of the following form

$$y_k = q_k(\mathbf{x}) = \sum_{i=1}^{v_l} \sum_{j=i}^{n} \alpha_{ij}^{(k)} x_i x_j + \sum_{i<v_{l+1}} \beta_i^{(k)} x_i, \text{ if } k \in O_l := \{v_l + 1 \cdots v_{l+1}\}.$$

  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 at all. So given all the y_i with $v_l < i \leq v_{l+1}$, and all the x_j with $j \leq v_l$, we can compute $x_{v_l+1}, \ldots, x_{v_{l+1}}$.
- a Rainbow is said to be a TTS [44] if the coefficients of $\mathcal{Q}$ are sparse.
- To invert $\mathcal{Q}$, determine (usu. at random) $x_1, \ldots x_{v_1}$, i.e., all x_k, $k \in S_1$. From the components of $\mathbf{y}$ that corresponds to the polynomials $p'_{v_1+1}, \ldots p'_{v_2}$, we obtain a set of o_1 equations in the variables x_k, $(k \in O_1)$. We may repeat the process to find all remaining variables.

[10] suggests TTS/Rainbow on $(\mathbb{F}_{2^4}, 24, 20, 20)$ and $(\mathbb{F}_{2^8}, 18, 12, 12)$, both with 2^{80} level design security (see the Appendix); we implement completely analogous TTS/Rainbow at $(\mathbb{F}_{31}, 24, 20, 20)$ which according to [20] should be somewhat stronger, and $(\mathbb{F}_{31}, 16, 16, 8, 16)$ at the same design security as the [10] versions. Note that Rainbow is usually regarded the "normal" or parent scheme, where TTS is faster but less investigated.

5.2 C^*, ℓ-Invertible Cycles (ℓIC) and Minus-p Schemes

The ℓ-invertible cycle [19] can be considered an improved version or extension of Matsumoto-Imai, otherwise known as C^* [38]. Let's review first the latter. "big-field" variants, the central map is really a map in a larger field $\mathbb{L}$, a degree n extension of a finite field $\mathbb{K}$. We have a map $\overline{\mathcal{Q}} : \mathbb{L} \to \mathbb{L}$ that we can invert, and pick a $\mathbb{K}$-linear bijection $\phi : \mathbb{L} \to \mathbb{K}^n$. Then we have the multivariate polynomial

map $\mathcal{Q} = \phi \circ \overline{\mathcal{Q}} \circ \phi^{-1}$, which is presumably quadratic (for efficiency). then, one "hide" this map $\mathcal{Q}$ by composing invertible affine linear maps S and T in $\mathbb{K}^n$. According to Imai, we pick a $\mathbb{K}$ of characteristic 2 for easy computation, and "of course we would like to have just the easiest map, the square". However, squaring is linear in characteristic two, so they use this map instead $\overline{\mathcal{Q}}$

$$\overline{\mathcal{Q}} : \mathbf{x} \longmapsto \mathbf{y} = \mathbf{x}^{1+q^\alpha}, \tag{5}$$

where $\mathbf{x}$ is an element in $\mathbb{L}$, and such that $\gcd(1 + q^\alpha, q^n - 1) = 1$. The last condition ensures that the map $\overline{\mathcal{Q}}$ has an inverse, which is given by

$$\overline{\mathcal{Q}}^{-1}(\mathbf{x}) = \mathbf{x}^h, \tag{6}$$

where $h(1 + q^\alpha) = 1 \bmod (q^n - 1)$. ℓIC also uses an intermediate field $\mathbb{L} = \mathbb{K}^k$. Here we use the simplest $\ell = 3$ and extends C^* by using this map:

$$\mathcal{Q} : (X_1,\, X_2,\, X_3) \in (\mathbb{L}^*)^3 \mapsto (Y_1,\, Y_2,\, Y_3) := (X_1X_2,\, X_2X_3,\, X_3X_1). \tag{7}$$

Most of the analysis of the properties of the 3IC map can be found in [14,19,26] — the 3IC and C^* maps has a lot in common. Typically, we do "minus" on 1/3 of the variables (3IC$^-$) and use a prefix (set one of the variables to zero) [14].

For signature schemes, [14] recommends for 2^{80} level security "pFLASH", which is $C^{*-}\mathrm{p}(2^4, 74, 22, 1)$ and closely related to the original FLASH except that it uses half as wide variables and fix one variable to 0. [10] in addition chooses $3\mathrm{IC}^-\mathrm{p}(2^4, 32, 1)$ as their implementation model.

We simply use the central 3IC over $\mathbb{F}_{31^{18}}$ here. For each projected minus scheme, to sign:

1. Put in random numbers to the "minus" coordinates.
2. Invert the linear transformation T to get $\mathbf{y}$.
3. Invert the central map C^* or 3IC to get $\mathbf{x}$. For 3IC, from (Y_1, Y_2, Y_3) we do
   (a) Compute Y_1Y_2 [1 multiplication].
   (b) Compute $(\#a)^{-1}$ [1 inverse].
   (c) Compute $Y_3(\#b) = X_2^{-2}$ [1 multiplication].
   (d) Compute $(\#c)^{-1} = X_2^2$ and $\pm\sqrt{(\#c)} = X_2^{-1}$ [1 sqrt+inverse].
   (e) Multiply X_2^{-1} from step $(\#d)$ to Y_1, Y_2, X_2^2 from $(\#d)$ [3 multiplications].
4. Invert the final linear transformation S to get $\mathbf{w}$ (usually two answers).
5. Return if either $\mathbf{w}$ has the last component zero, else go to step 1 and repeat.

Note that 3IC in $\mathbb{F}_{31^{18}}$ resembles pFLASH more due to much exponentiation.

5.3 HFE and Others

HFE is a well-known, perhaps the best-known Multivariate encryption method.

At the moment, solving HFE systems directly is considered to be sub-exponential [29], and what is considered to be a "standard" HFE implementations for at least 128-bit key transmission works over $\mathbb{F}_{2^{103}}$ with degree $d = 129$. We know of no

timings below 100M cycles. Recently there is an attempt to get faster HFE with a multivariate construction [7]. Randomly choose a $\mathbb{L}^h \to \mathbb{L}^h$ quadratic map

$$\overline{\mathcal{Q}}(X_1, ..., X_h) = (Q_1(X_1, ..., X_h), \cdots, Q_h(X_1, ..., X_h))$$

where each Q_ℓ for $\ell = 1, \ldots, h$ is a randomly chosen quadratic polynomial:

$$Q_\ell(X_1, \ldots, X_h) = \sum_{1 \le i \le j \le h} \alpha_{ij}^{(\ell)} X_i X_j + \sum_{j=1}^{h} \beta_j^{(\ell)} X_j + \gamma^{(\ell)}. \tag{8}$$

When h is small, this $\overline{\mathcal{Q}}$ can be easily converted into an equation in one of the X_i using Gröbner basis methods at degree no longer than 2^h, this is good since solving univariate equations is cubic in the degee. The problem is that the authors showed that these schemes are basically equivalent to normal HFE, meaning equally insecure.

It has been recently noticed that for odd characteristics, the usual Gröbner basis attacks on HFE does not work as well [18]. Hence, we tried our hands at multivariate HFEs. Since we are not acquainted with the theory, we enforced prefixed zero blocks (the p modifier, blocking structural attacks) at a $31\times$ speed penalty. The results, with $h = 3, 4$, is listed.

6 A Recap and Discussion

We do not claim to be experts at writing vector code. In fact, we probably ran into many of the normal traps for newcomers to SSE2 code. We believe that further efforts will improve the speed tests. However, all our data indicates that even now MPKCs in odd-characteristic fields hold their own against prior MPKCs that is based in $\mathbb{F}_{2^k}$, and even look generally faster. Given the above, and some recent interest into the theory of algebraic attacks on odd-characteristic HFE, we think that Odd-Field MPKCs merit more investigation.

References

1. M.-L. Akkar, N. T. Courtois, R. Duteuil, and L. Goubin. A fast and secure implementation of SFLASH. In *Public Key Cryptography — PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 267–278. Y. Desmedt, ed., Springer, 2002.
2. C. Berbain, O. Billet, and H. Gilbert. Efficient implementations of multivariate quadratic systems. In E. Biham and A. M. Youssef, editors, *Selected Areas in Cryptography*, volume 4356 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2007.
3. E. R. Berlekamp. Factoring polynomials over finite fields. *Bell Systems Technical Journal*, 46:1853–1859, 1967. Republished in: Elwyn R. Berlekamp. "Algebraic Coding Theory". McGraw Hill, 1968.
4. D. J. Bernstein. Faster square roots in annoying finite fields. `http://cr.yp.to/papers.html#sqroot`. draft, 2001, to appear in "High-Speed Cryptography".

Scheme	KeyGen	SecrMap	PublMap
rainbow $(16, 24, 20, 20)$	344M	188k	81k
rainbow $(256, 18, 12, 12)$	109M	135k	117k
rainbow $(31, 16, 16, 8, 16)$	108.2M	99.4k	75.1k
TTS $(16, 24, 20, 20)$	102M	31k	117k
TTS $(256, 18, 12, 12)$	151M	61k	81k
TTS $(31, 24, 20, 20)$	79M	64k	85k
3IC-p $(16, 32, 1)$	143M	456k	788k
3IC-p $(31, 18, 1)$	11.5M	822k	59.0k
pflash	72M	2400k	853k
3HFE-p $(31, 9)$ no sse	9.8M	1308k	36k
4HFE-p $(31, 10)$	32M	2104k	41k
RSA 1024b	108M	2950k	121k
ECC 256b	2.7M	2850k	3464k

Table 1. Cycle Counts in One Core of Core2 65nm

Scheme	KeyGen	SecrMap	PublMap
rainbow $(16, 24, 20, 20)$	396.2M	138.7k	83.9k
rainbow $(256, 18, 12, 12)$	234.6M	297.0k	224.4k
rainbow $(31, 16, 16, 8, 16)$	145.4M	110.7k	117.3k
TTS $(16, 24, 20, 20)$	225.2M	103.8k	84.8k
TTS $(256, 18, 12, 12)$	20.4M	69.1k	224.4k
TTS $(31, 24, 20, 20)$	110M	88k	147k
3IC-p $(16, 32, 1)$	204M	683k	758k
3IC-p $(31, 18, 1)$	18.9M	1.55M	95.9k
pflash	127M	5.01M	914k
4HFE-p $(31, 10)$	33M	2090k	41k
RSA 1024b	150M	2647k	117k
ECC 256b	2.8M	3205k	3837k

Table 2. Cycle Counts in One Core of K8

Scheme	KeyGen	SecrMap	PublMap
rainbow $(16, 24, 20, 20)$	343.8M	136.8k	79.3k
rainbow $(256, 18, 12, 12)$	110.7M	143.9k	121.4k
rainbow $(31, 16, 16, 8, 16)$	98.5M	93.5k	60.3k
TTS $(16, 24, 20, 20)$	175.7M	64.8 k	78.9k
TTS $(256, 18, 12, 12)$	11.5M	35.9k	121.4k
TTS $(31, 24, 20, 20)$	79M	62k	78k
3IC-p $(16, 32, 1)$	143M	452k	788k
3IC-p $(31, 18, 1)$	10.3M	728k	48.6k
pflash	71M	2450k	853k
3HFE-p $(31, 9)$ no sse	9.9 M	1305k	36k
4HFE-p $(31, 10)$	29M	1948k	34k

Table 3. Cycle Counts in One Core of Core2 45nm

Scheme	KeyGen	SecrMap	PublMap
rainbow $(16, 24, 20, 20)$	367M	172k	82k
rainbow $(256, 18, 12, 12)$	169M	214k	152k
rainbow $(31, 16, 16, 8, 16)$	104.3M	95.2k	68.4k
TTS $(16, 24, 20, 20)$	179M	82k	79k
TTS $(256, 18, 12, 12)$	143M	46k	154k
TTS $(31, 24, 20, 20)$	92M	151k	84k
3IC-p $(16, 32, 1)$	144M	600k	735k
3IC-p $(31, 18, 1)$	26.6M	1.41M	63.9k
pflash	87M	3.68M	801k

Table 4. Comparison on One Core of a Phenom (K10)

CPU	Mult.	Square	Inverse	Sqrt	Sqrt+Inv
K8 (Opteron)	397	312	5521	8120	11646
K10(Phenom)	242	222	2984	5153	7170
Core2 (65nm)	234	194	2640	4693	6332
Core2 (45nm)	145	129	1980	3954	5244

Table 5. SSE code cycle counts in $\mathbb{F}_{31^{18}}$, a Tower Field

5. D. J. Bernstein. *Algorithmic number theory*, chapter Fast multiplication and its applications. Cambridge University Press, Joe Buhler, Peter Stevenhagen, eds., 2003. ISBN 978–0521808545.
6. O. Billet and H. Gilbert. Cryptanalysis of rainbow. In *Security and Cryptography for Networks*, volume 4116 of *LNCS*, pages 336–347. Springer, September 2006.
7. O. Billet, J. Patarin, and Y. Seurin. Analysis of intermediate field systems. presented at SCC 2008, Beijing.
8. D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 78–89, 1996.
9. D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(587–592), 1981.
10. A. I.-T. Chen, C.-H. O. Chen, M.-S. Chen, C.-M. Cheng, and B.-Y. Yang. Practical-sized instances of multivariate PKCs: Rainbow, TTS, and ℓIC-derivatives. In J. Buchmann, J. Ding, and T. Hodges, editors, *Post-Quantum Crypto*, volume 5299 of *LNCS*, pages 95–106. Springer, Oct 2008.
11. D. Coppersmith, J. Stern, and S. Vaudenay. The security of the birational permutation signature schemes. *Journal of Cryptology*, 10:207–221, 1997.
12. N. Courtois. Algebraic attacks over $GF(2^{\mathrm{k}})$, application to HFE challenge 2 and SFLASH-v2. In PKC [25], pages 201–217. ISBN 3-540-21018-0.
13. N. Courtois, L. Goubin, and J. Patarin. *SFLASH: Primitive specification (second revised version)*, 2002. `https://www.cosic.esat.kuleuven.be/nessie`, Submissions, Sflash, 11 pages.
14. J. Ding, V. Dubois, B.-Y. Yang, C.-H. O. Chen, and C.-M. Cheng. Could SFLASH be repaired? In L. Aceto, I. Damg**a**rd, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 691–701. Springer, 2008. E-Print 2007/366.

15. J. Ding and J. Gower. Inoculating multivariate schemes against differential attacks. In *PKC*, volume 3958 of *LNCS*. Springer, April 2006. Also available at `http://eprint.iacr.org/2005/255`.
16. J. Ding, J. Gower, and D. Schmidt. *Multivariate Public-Key Cryptosystems*. Advances in Information Security. Springer, 2006. ISBN 0-387-32229-9.
17. J. Ding and D. 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.
18. J. Ding, D. Schmidt, and F. Werner. Algebraic attack on hfe revisited. In *ISC 2008*, Lecture Notes in Computer Science. Springer. to appear.
19. J. Ding, C. Wolf, and B.-Y. Yang. ℓ-invertible cycles for multivariate quadratic public key cryptography. In *PKC*, volume 4450 of *LNCS*, pages 266–281. Springer, April 2007.
20. J. Ding, B.-Y. Yang, C.-H. O. Chen, M.-S. Chen, and C.-M. 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`.
21. V. Dubois, P.-A. Fouque, A. Shamir, and J. Stern. Practical cryptanalysis of SFLASH. In *Advances in Cryptology — CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 1–12. Alfred Menezes, ed., Springer, 2007.
22. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139:61–88, June 1999.
23. J.-C. 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.
24. J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Dan Boneh, ed., Springer, 2003.
25. Feng Bao, Robert H. Deng, and Jianying Zhou (editors). *Public Key Cryptography — PKC 2004*, volume 2947 of *Lecture Notes in Computer Science*. Springer, 2004. ISBN 3-540-21018-0.
26. P.-A. Fouque, G. Macario-Rat, L. Perret, and J. Stern. Total break of the ℓIC-signature scheme. In *Public Key Cryptography*, pages 1–17, 2008.
27. M. R. Garey and D. 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.
28. L. Goubin and N. T. Courtois. Cryptanalysis of the TTM cryptosystem. In *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 44–57. Tatsuaki Okamoto, ed., Springer, 2000.
29. L. Granboulan, A. Joux, and J. Stern. Inverting HFE is quasipolynomial. In C. Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2006.
30. T. Granlund and P. Montgomery. Division by invariant integers using multiplication. In *In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, 1994. `http://www.swox.com/~tege/divcnst-pldi94.pdf`.
31. T. Güneysu and C. Paar. Ultra high performance ecc over nist primes on commercial fpgas. In E. Oswald and P. Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2008.

32. H. Imai and T. Matsumoto. Algebraic methods for constructing asymmetric cryptosystems. In *Algebraic Algorithms and Error-Correcting Codes, 3rd International Conference, AAECC-3, Grenoble, France, July 15-19, 1985, Proceedings*, volume 229 of *Lecture Notes in Computer Science*, pages 108–119. Jacques Calmet, ed., Springer, 1985.
33. Intel Corp. Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/design/processor/manuals/248966.pdf`, Nov. 2007.
34. Intel Corp. Carry-less multiplication and its usage for computing the GCM mode. `http://http://software.intel.com/en-us/articles/carry-less-multiplicati%on-and-its-usage-for-computing-the-gcm-mode`, 2008.
35. A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in Physics-Doklady, **7** (1963), pp. 595–596.
36. A. Kipnis, J. Patarin, and L. Goubin. Unbalanced Oil and Vinegar signature schemes. In *Advances in Cryptology — EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Jacques Stern, ed., Springer, 1999.
37. A. Kipnis and A. Shamir. Cryptanalysis of the oil and vinegar signature scheme. In *Advances in Cryptology — CRYPTO 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 257–266. Hugo Krawczyk, ed., Springer, 1998.
38. T. Matsumoto and H. Imai. Public quadratic polynomial-tuples for efficient signature verification and message-encryption. In *Advances in Cryptology — EUROCRYPT 1988*, volume 330 of *Lecture Notes in Computer Science*, pages 419–545. Christoph G. Günther, ed., Springer, 1988.
39. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(18), August 2008.
40. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full sha-1. In *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Victor Shoup, ed., Springer, 2005.
41. C. Wolf, A. Braeken, and B. Preneel. Efficient cryptanalysis of RSE(2)PKC and RSSE(2)PKC. In *Conference on Security in Communication Networks — SCN 2004*, volume 3352 of *Lecture Notes in Computer Science*, pages 294–309. Springer, Sept. 8–10 2004. Extended version: `http://eprint.iacr.org/2004/237`.
42. C. Wolf and B. Preneel. Taxonomy of public key schemes based on the problem of multivariate quadratic equations. Cryptology ePrint Archive, Report 2005/077, 12$^{\text{th}}$ of May 2005. `http://eprint.iacr.org/2005/077/`, 64 pages.
43. W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.
44. B.-Y. Yang and J.-M. 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.

A Security and Prior Instances of TTS/Rainbow

A.1 Known Attacks and Security Criteria from [10, 20]

1. Rank (or Low Rank, MinRank) attack to find a central equation with least rank [44].

$$C_{\text{low rank}} \approx \left[q^{v_1+1} m(n^2/2 - m^2/6)/\right] \mathfrak{m}.$$

Here the unit $\mathfrak{m}$ is a multiplications in $\mathbb{K}$, and v_1 the number of vinegars in layer 1. This is the "MinRank" attack of [28]. as improved by [6, 44].

2. Dual Rank (or High Rank) attack [11, 28], which finds a variable appearing the fewest number of times in a central equation cross-term [20, 44]:

$$C_{\text{high rank}} \approx \left[q^{o_n - v'} n^3/6\right] \mathfrak{m},$$

where v' counts vinegar variables that never appears until the final segment.

3. Trying for a direct solution. The complexity is roughly as $\mathcal{MQ}(q; m, m)$.
4. Using the Reconciliation Attack [20], the complexity is as $\mathcal{MQ}(q; v_u, m)$.
5. Using the Rainbow Band Separation from [20], the complexity is determined by that of $\mathcal{MQ}(q; n, m+n)$.
6. Against TTS, there is Oil-and-Vinegar Separation [36,37,41], which finds an Oil subspace that is sufficiently large (estimates as corrected in [44]).

$$C_{\text{UOV}} \approx \left[q^{n-2o-1} o^4 + (\text{some residual term bounded by } o^3 q^{m-o}/3)\right] \mathfrak{m}.$$

o is the max. *oil set* size, i.e., there is a set of o central variables which are never multiplied together in the central equations, and no more.

A.2 Rainbow and TTS Instances

It is demonstrated by [20] that using $\mathbb{F}_{2^8}$ there is no way to get to 2^{80} security at a size smaller than vinegar sequence $(18, 12, 12)$. If we look at smaller fields such as $\mathbb{F}_{2^4}$, we can do smaller such as $(24, 20, 20)$. The digest is 160 bits.

TTS of the same size over $\mathbb{F}_{2^8}$ or $\mathbb{F}_{2^4}$ are $2\times$ or more the speed of than a Rainbow instance. They also tend to have much lower memory requirement. The following instances are called TTS/7 ([10]), built with exactly the same rainbow structural parameters as above. We add two instances in $\mathbb{F}_{31}$ one should be more secure and the other equally secure than the [10] instances according to the formulas in [20].

TTS $(2^8, 18, 12, 12)$ $\mathbb{K} = \mathbb{F}_{2^8}$, $n = 42$, $m = 24$. $\mathcal{Q}$ is structured as follows:

$$y_i = x_i + a_{i1} x_{\sigma_i} + a_{i2} x_{\sigma'_i} + \sum_{j=0}^{11} p_{ij} x_{j+18} x_{\pi_i(j)}$$
$$+ p_{i,12} x_{\pi_i(12)} x_{\pi_i(15)} + p_{i,13} x_{\pi_i(13)} x_{\pi_i(16)} + p_{i,14} x_{\pi_i(14)} x_{\pi_i(17)},\ i = 18 \cdots 29$$
[indices $0 \cdots 17$ appears exactly once in each random permutation π_i, and exactly once among the σ, σ' (where six σ'_i slots are empty)];

$$y_i = x_i + a_{i1} x_{\sigma_i} + a_{i2} x_{\sigma'_i} + a_{i3} x_{\sigma''_i} + \sum_{j=0}^{11} x_{j+29}(p_{ij} x_{\pi_i(j)} + p_{i,j+12} x_{\pi_i(j+12)})$$
$$+ p_{i,24} x_{\pi_i(24)} x_{\pi_i(27)} + p_{i,25} x_{\pi_i(25)} x_{\pi_i(28)} + p_{i,26} x_{\pi_i(26)} x_{\pi_i(29)},\ i = 30 \cdots 41$$
[indices $0 \cdots 29$ appears exactly once in each random permutation π_i, and exactly once among the σ, σ', σ'' (where six σ''_i slots are empty)].

TTS $(2^4, 24, 20, 20)$ $\mathbb{K} = \mathbb{F}_{2^4}$, $n = 64$, $m = 40$.

$$y_i = x_i + a_{i1}x_{\sigma_i} + a_{i2}x_{\sigma'_i} + \sum_{j=0}^{19} p_{ij}x_{j+23}x_{\pi_i(j)}$$
$$+ p_{i,20}x_{\pi_i(20)}x_{\pi_i(22)} + p_{i,21}x_{\pi_i(21)}x_{\pi_i(23)},\ i = 24 \cdots 43$$

[indices $0 \cdots 23$ appears exactly once in each random permutation π_i, and exactly once among the σ, σ' (there are only four σ'_i)];

$$y_i = x_i + a_{i1}x_{\sigma_i} + a_{i2}x_{\sigma'_i} + a_{i3}x_{\sigma''_i} + \sum_{j=0}^{19} x_{j+44}(p_{ij}x_{\pi_i(j)} + p_{i,j+20}x_{\pi_i(j+20)})$$
$$+ p_{i,40}x_{\pi_i(40)}x_{\pi_i(42)} + p_{i,41}x_{\pi_i(41)}x_{\pi_i(43)},\ i = 44 \cdots 63$$

[indices $0 \cdots 43$ appears exactly once in each random permutation π_i, and exactly once among the σ, σ', σ'' (there are only four σ''_i)].

TTS $(31, 24, 20, 20)$ $\mathbb{K} = \mathbb{F}_{31}$, $n = 64$, $m = 40$. Exactly as above except for $\mathbb{F}_{31}$.

TTS $(31, 16, 16, 8, 16)$ $\mathbb{K} = \mathbb{F}_{31}$, $n = 56$, $m = 40$. We copy the structure above, except that in the first stage, second, and third stages we use respectively: 1 linear and 1 quadratic cross-term per index, 4 linear and 4 quadratic cross-term per index; 2 linear, 2 quadratic cross-terms per index.

```
unsigned int div31 (unsigned long long int a, short int *b){
  // rdi is a, rsi is b[]
  // r8 will store div31=595056260442243601=ceil(2^69/31)-2^64
  // rdx, rax are scratch; rax will return final result

  asm ("movq $595056260442243601, %r8\n\t"
       "movq %rdi, %rax\n"
       "mulq %r8\n\t"        // rdx:rax is now x * div31
       "addq %rdi, %rdx\n\t" // rdx = floor(x*div31/2^64)+x
       "rcrq %rdx\n\t''      //   last add can overflow to carry!
       "shrq $4, %rdx\n\t"   // rdx = old x div 31
       "movq %rdx, %rax\n\t"
       "shlq $5, %rdx\n\t"   // rdx = rax * 32, might carry!
       "subq %rax, %rdx\n\t" // rdx = rax * 31, even with carry!
       "subq %rdx, %rdi\n\t" // rdi now = x mod 31
       "movw %di, (%rsi,%rcx,2)\n\t"
}
```

Fig. 1. Sample divide-mod 31 code in AT&T syntax x86-64 inline assembly