SSE Implementation of Multivariate PKCs on Modern x86 CPUs

Anna Inn-Tung Chen[†], Ming-Shing Chen[‡], Tien-Ren Chen[‡], Chen-Mou Cheng[†], Jintai Ding[*], Eric Li-Hsiang Kuo[‡], Frost Yu-Shuang Lee[†], Bo-Yin Yang[‡]

[†]National Taiwan University, Taipei, Taiwan, `{anna1110,frost,doug}@crypto.tw`
[‡]Academia Sinica, Taipei, Taiwan, `{mschen,trchen1103,lorderic,by}@crypto.tw`
[*]University of Cincinnati, Cincinnati, Ohio, USA, `ding@math.uc.edu`

May 20, 2009

Abstract

Multivariate Public Key Cryptosystems (MPKCs) are often touted as future-proofing the advent of the Quantum Computer. It also has been known for efficiency compared to "traditional" alternatives. However, this advantage seems to be eroding with the increase of arithmetic resources in modern CPUs and improved algorithms, especially with respect to ECC. We show that the same hardware advances do not necessarily just favor ECC. The same modern commodity CPUs also have an overabundance of small integer arithmetic/logic resources, embodied by SSE2 or other vector instruction set extensions, that are also useful for MPKCs. On CPUs supporting Intel's SSSE3 instructions, we achieve a 4× speed-up over prior implementations of Rainbow-type systems (such as the ones implemented in hardware by Bogdanov et al. at CHES 2008) in both public and private map operations. Furthermore, if we want to implement MPKCs for all general purpose 64-bit CPUs from Intel and AMD, we can switch to MPKC *over fields of relatively small odd prime characteristics*. For example, by taking advantage of SSE2 instructions, Rainbow over $\mathbb{F}_{31}$ can be up to 2× faster than prior implementations of same-sized systems over $\mathbb{F}_{16}$. A key advance is in implementing Wiedemann instead of Gaussian system solvers. We explain the techniques and design choices in implementing our chosen MPKC instances, over representative fields such as $\mathbb{F}_{31}$, $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$. We believe that our results can easily carry over to modern FPGAs, which often contain a large number of multipliers in the form of DSP slices, offering superior computational power to odd-field MPKCs.

Keywords: multivariate public key cryptosystem (MPKC), TTS, rainbow, ℓIC, vector instructions, SSE2, SSSE3, Wiedemann.

1 Introduction

Multivariate public-key cryptosystems (MPKCs) [12, 36] form a genre of PKCs dating from 1988 [31], whose public keys represent 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).

Of course, a random $\mathcal{P}$ would not be invertible by the legitimate user, so in practice, the design almost always 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 invertible" map $\mathcal{Q} : \mathbf{x} \mapsto \mathbf{y}$, such that $\mathcal{P} = T \circ \mathcal{Q} \circ S$ with $\mathcal{P}(0)$ always taken to be zero. Under this design, the public key consists of the polynomials in $\mathcal{P}$, while the private key is $\mathrm{M}_s^{-1}, \mathbf{c}_s, \mathrm{M}_T^{-1}, \mathbf{c}_T$, plus whatever information that determines the *central map* $\mathcal{Q}$.

MPKCs have been touted as (a) potentially surviving future attacks using quantum computers, and (b) faster than "traditional" competitors — in 2003, SFLASH was a finalist for the NESSIE project signatures, recommended for embedded use. *We seek to evaluate whether (b) is still true, and how things has been changing with the evolution of computer architecture. Without going into the theory behind any cryptosystem, we will discuss the implemention of MPKCs on today's commodity CPUs. We conclude that modern single-instruction-multiple-data (SIMD) units also make great cryptographic hardware for MPKCs, which stays ahead speed-wise.*

1.1 History and Questions

It was commonly accepted that MPKCs are faster since arithmetic operations on large units (e.g., 1024+-bit integers in RSA, or 256-bit integers in ECC) are replaced by operations on many small units. However, the latter means many more accesses to memory. Seminal papers more than a decade ago [6, 37] pointed out that eventually the memory latency and bandwidth would become the bottleneck of the performance of a microprocessor.

The playing field is obviously changing. When MPKCs were initially proposed two decades ago [26, 31], commodity CPUs computed a 32-bit integer product maybe every 15–20 cycles. When NESSIE called for primitives in 2000, x86 CPUs could compute one 64-bit product every 3 (Athlon) to 10 (Pentium 4) cycles. The big, pipelined multiplier in an AMD Opteron today can output one 128-bit integer product every 2 cycles. ECC implementers quickly exploited these advances.

In stark contrast, per-cycle timings of multiplication in $\mathbb{F}_{256}$ or $\mathbb{F}_{16}$ seem to have changed much less. In the '80s, a MOS Technology's 6502 CPU or an 8051 microcontroller from Intel multiplies in $\mathbb{F}_{256}$ in around a dozen instruction cycles using three table look-ups. Modern Intel Core and AMD Athlon/Phenom CPUs takes a comparable number of cycles today to complete that.

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 to 24 months ("Moore's Law") for the last two decades. Now the width of a typical arithmetic/logical unit is 64 bits, vector units are everywhere, and even FPGAs have dozens to hundreds of multipliers built in. It may seem that commodity hardware has never been more friendly to RSA and ECC — the deck seems stacked considerably against MPKCs. Indeed, Intel even has a new carryless multiplication instruction [28] to be deployed soon, which will support ECC over $\mathbb{F}_{2^k}$, the only "traditional" cryptosystem that has been seemingly left behind by advances in chip architectures.

Furthermore, we now understand MPKCs and equation solving much better. In 2004, TTS/4 and SFLASH were much faster signature schemes than traditional competitors using RSA or ECC [1, 9, 38]. However, both these two instances have been broken since [16, 17]. Today TTS/7 and 3IC-p still seem to do fine [7], but the impending doom of SHA-1 [34] will force longer message digests and thus slower MPKCs while leaving RSA mostly untouched.

The obvious question is, then: *Can all the extras on modern commodity CPUs be put to use with MPKCs as well? If so, how does MPKCs compare to traditional PKCs today, and how is that likely going to change for the future?*

1.2 Our Answers and Contributions

We will show that advances in chip building also benefit MPKCs. First, vector instructions available on many modern CPUs can provide significant speed-ups for MPKCs over binary fields. Secondly, we can derive an advantage for MPKCs by using as the base field $\mathbb{F}_q$ for q equal to a small odd prime such as 31 on most of today's CPUs. This may sound somewhat counter-intuitive, since for binary fields addition can be easily accomplished by the logical exclusive-or (XOR) operation, while for odd prime fields, costly reductions modulo q are unavoidable. Our reasoning and counter arguments are detailed as follows.

1. Certain CPUs can do simultaneous look-ups from a small table (usually 16 bytes). Architectures supporting such an instruction include:

- all current Intel x86 CPUs of the Core and Atom lines with the Intel SSSE3 instruction `PSHUFB`. *The Atom CPUs are specially designed for low-power and embedded applications;*
- all future AMD CPUs, which supports the SSE5 instruction `PPERM`, which can do everything `PSHUFB` does;
- many Power derivatives — PowerPC G4/G5, Power6, Xenon (in XBOX 360) and Cell (in PS3) CPUs — having the AltiVec or VMX instruction `PERMUTE`, essentially identical to `PSHUFB`.

Scalar-to-vector multiplication in $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ can thus be sped up roughly tenfold. Using this, MPKC schemes such as TTS and Rainbow get a speed-up by a factor of $4\times$ or higher.

2. Many CPUs today do not support the simultaneous table look-ups but instead have some vector instructions, e.g., virtually all desktop CPUs today support SSE2 instructions, which can pack eight 16-bit integer operands in its 128-bit `xmm` registers and hence dispatch eight simultaneous integer operations per cycle in an SIMD style.
   - Using as the base field of MPKCs $\mathbb{F}_q$ with a small odd prime q instead of $\mathbb{F}_{256}$ or $\mathbb{F}_{16}$ enables us to take advantage of SSE2 and equivalent hardware. Even considering the overhead of conversion between base q and binary, schemes over $\mathbb{F}_{31}$ is usually faster than an equivalent scheme over $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ (that does not use SSSE3).
   - MPKCs over $\mathbb{F}_q$ are still substantially faster than ECC or RSA. While algebra tells us that q can be any prime power, it pays to specialize to a small set of carefully chosen instances. In most of our implementations, we will be using $q = 31$, which experiments show to be particularly computationally efficient.

In this work, we will demonstrate that the advances in chip architecture *do not* leave MPKCs behind while improving traditional alternatives. Furthermore, we list a set of *counter-intuitive* techniques we have discovered during the course of implementing finite field arithmetic using vector instructions.

1. Sometimes it is still much better when solving a small and dense matrix equation to use iterative methods like Wiedemann instead of straight Gaussian elimination.
2. To invert a vector in $\mathbb{F}_q^*$ element by element, it can be better to raise to the $(q-2)$-th power.
3. For big-field MPKCs, fields of certain sizes such as $\mathbb{F}_{31^{15}}$ are much better than other comparable sizes because arithmetic operations are *much faster*. For such fields, computing the multiplicative inverse is again fastest by raising to a high power.
4. It is important to manage numerical ranges to avoid overflow, for which certain instructions are unexpectedly useful. For example, the `PMADDWD` or *packed multiply-add word to double word* instruction, which computes from two 8-long vectors of 16-bit signed words $(x_0, \ldots, x_7)$ and $(y_0, \ldots, y_7)$ the 4-long vector of 32-bit signed words $(x_0y_0 + x_1y_1, x_2y_2 + x_3y_3, x_4y_4 + x_5y_5, x_6y_6 + x_7y_7)$, avoids many carries when evaluating a matrix-vector product $\pmod q$.

Finally, we reiterate that like most implementation works such as the one by Bogdanov et al [5], we only discuss implementation issues and do not concern ourselves with the security of MPKCs in this paper. Those readers interested in the security and design of MPKCs are instead referred to the MPKC book [12] and numerous research papers in the literature.

2 Background on MPKCs

In this section, we summarize the MPKC instances that we will be investigating. We will use the notation in Sec. 1, so we only need to describe the central map $\mathcal{Q}$ since M_S and M_T are always square and invertible matrices, usually of dimension n and m, respectively. To execute a private map, we replace the "minus" components if necessary, invert T, invert central map $\mathcal{Q}$, invert S, and check that the prefix or perturbation is good.

2.1 Notes on Odd-field MPKCs

Some MPKCs — TTS, Rainbow, even oil-and-vinegar [10,11,16,30] — can be implemented verbatim over small odd prime fields just like over $\mathbb{F}_{2^k}$. For the remaining ones, e.g., ℓIC-derivatives, an odd-characteristic version is inconvenient but not impossible. Most attacks on and their respective defenses of MPKCs are fundamentally independent of the base field. What is even better is that some attacks are known or conjectured to be easier over binary fields than over small odd prime fields [4,8,14,20], but never vice versa. For example, the powerful Gröbner basis methods [18,19] are in this last category.

C^* and HFE in an odd-characteristic field were mentioned by Wolf and Preneel [36], but not much researched until recently.

2.2 Rainbow and TTS Families of Digital Signatures

Rainbow$(\mathbb{F}_q, o_1, \ldots, o_\ell)$ is characterized as follows [13,16] as u stages of UOV:

- 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 components $y_{v_1+1} = q_{v_1+1}(\mathbf{x})$, $y_{v_1+2} = q_{v_1+2}(\mathbf{x}), \ldots, y_n = q_n(\mathbf{x})$, where $y_k = q_k(\mathbf{x}) = \sum_{i=1}^{v_l} \sum_{j=i}^{n} \alpha_{ij}^{(k)} x_i x_j + \sum_{i<v_{l+1}} \beta_i^{(k)} x_i$, 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.* 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 $x_{v_l+1}, \ldots, x_{v_{l+1}}$. So given $\mathbf{y}$, we guess $x_1, \ldots x_{v_1}$, recursively solve for all x_i's to invert $\mathcal{Q}$, and repeat if needed.

Ding et al. [7,16] suggest Rainbow/TTS with parameters $(\mathbb{F}_{2^4}, 24, 20, 20)$ and $(\mathbb{F}_{2^8}, 18, 12, 12)$ for 2^{80} design security. According to their criteria, the former instance should not be more secure than Rainbow/TTS at $(\mathbb{F}_{31}, 24, 20, 20)$ and roughly the same with $(\mathbb{F}_{31}, 16, 16, 8, 16)$. Note that in today's terminology, TTS is simply a Rainbow with sparse coefficients, which is faster but less understood.

2.3 Hidden Field Equation (HFE) Encryption Schemes

HFE is a "big-field" variant of MPKC. We identify $\mathbb{L}$, a degree n extension of the base field $\mathbb{K}$ [35] with $(\mathbb{F}_q)^n$ via an implicit bijective map $\phi : \mathbb{L} \to (\mathbb{F}_q)^n$. With $\mathbf{y} = \sum_{0 \le i,j < \rho} a_{ij} \mathbf{x}^{q^i + q^j} + \sum_{0 \le i < \rho} b_i \mathbf{x}^{q^i} + c$, we have a quadratic $\mathcal{Q}$, invertible via the Berlekamp algorithm with $\mathbf{x}, \mathbf{y}$ as elements of $(\mathbb{F}_q)^n$.

Solving HFE directly is considered to be sub-exponential [23], and a "standard" HFE implementation for 2^{80} security works over $\mathbb{F}_{2^{103}}$ with degree $d = 129$. We know of no timings below 100 million cycles on a modern processor like a Core 2. Modifiers like vinegar or minus cost extra.

A recent attempt to make HFE faster uses the following multi-variable construction [4]. First, 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 $Q_\ell = Q_\ell(X_1, \ldots, X_h) = \sum_{1 \le i \le j \le h} \alpha_{ij}^{(\ell)} X_i X_j + \sum_{j=1}^{h} \beta_j^{(\ell)} X_j + \gamma^{(\ell)}$ is also a randomly chosen quadratic for $\ell = 1, \ldots, h$. 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, which is good since solving univariate equations is cubic in the degree. The problem is that the authors also showed that these schemes are equivalent to the normal HFE and hence are equally (in-)secure.

Given the recent conjecture [14] that for odd characteristic, the usual Gröbner basis attacks on HFE does not work as well. We will try our hands at multivariate HFEs over $\mathbb{F}_q$ for an odd q. To be conservative, we will enforce one prefixed zero block (the prefix or p modifier, to block structural attacks) at a q times speed penalty.

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

C^* is the original Matsumoto-Imai scheme [31], also a big-field variant of MPKC. We identify a larger field $\mathbb{L}$ with $\mathbb{K}^n$ with a $\mathbb{K}$-linear bijection $\phi : \mathbb{L} \to \mathbb{K}^n$. The central map $\mathcal{Q}$ is essentially $\overline{\mathcal{Q}} : \mathbf{x} \longmapsto \mathbf{y} = \mathbf{x}^{1+q^\alpha}$, where $\mathbb{K} = \mathbb{F}_q$. This is invertible if $\gcd(1 + q^\alpha, q^n - 1) = 1$.

The ℓ-invertible cycle [15] can be considered as an improved extension of C^*. We use the simple $\ell = 3$. In 3IC we also uses an intermediate field $\mathbb{L} = \mathbb{K}^k$, where $k = n/3$. The central map is $\mathcal{Q} : (X_1, X_2, X_3) \in (\mathbb{L}^*)^3 \mapsto (Y_1, Y_2, Y_3) := (X_1X_2, X_2X_3, X_3X_1)$. 3IC and C^* maps have lots in common [10, 15, 21]. To sign, we do "minus" on r variables and use s prefixes (set one or more of the variables to zero), to defend against all known attacks against C^* schemes [10]. This is written as $C^{*-}\mathrm{p}(q, n, \alpha, r, s)$ or 3IC-p(q, n, r, s). Ding et al. recommend $C^{*-}\mathrm{p}(2^4, 74, 22, 1)$ or "pFLASH" [10].

The following is a sketch of how to invert 3IC-p over a field like $\mathbb{F}_{31^{18}}$, from (Y_1, Y_2, Y_3) we do:

1. Compute Y_1Y_2 [1 multiplication].
2. Compute $(\#a)^{-1}$ [1 inverse].
3. Compute $Y_3(\#b) = X_2^{-2}$ [1 multiplication].
4. Compute $(\#c)^{-1} = X_2^2$ and $\pm\sqrt{(\#c)} = X_2^{-1}$ [1 sqrt+inverse].
5. Multiply X_2^{-1} from step $(\#d)$ to Y_1, Y_2, X_2^2 from $(\#d)$ [3 multiplications].

Note: for odd q, square roots are non-unique and slow, and $\alpha = 0$ (square function) is a valid C^*.

3 Background on x86 Vector Instruction Set Extensions

The seminal [31] of Matsumoto-Imai notes that bit slicing is useful for MPKCs over $\mathbb{F}_2$ as a form of SIMD. Berbain et al. [2] pointed out that bit slicing can be extended appropriately for $\mathbb{F}_{16}$ to evaluate public maps of MPKCs, as well as to run the `QUAD` stream cipher. Chen et al. extended this further to Gaussian elimination in $\mathbb{F}_{16}$, to be used for TTS [7].

To our best knowledge, the only mention of more advanced vector instructions in the MPKC literature is T. Moh's suggestion to use AltiVec instructions (only available then in the PowerPC G4) in his TTM cryptosystem [32]. This fell into obscurity after TTM was cryptanalyzed [22].

In this section, we describe one of the most widely deployed vector instruction sets, namely, the x86 SIMD extensions. The assembly language mnemonics and code in this section are given according Intel's naming conventions, which is supported by both `gcc` and Intel's own compiler `icc`. We have verified that the two compilers give similar performance results for the most part.

3.1 Integer instructions in 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. It is supported by all Intel CPUs since the Pentium 4, all AMD CPUs since the K8 (Opteron and Athlon 64), as well as the VIA C7/Nano CPUs. The SSE2 instructions operate on 16 architectural 128-bit registers, called the `xmm` registers. Most relevant to us are SSE2's integer operations, which treat `xmm` registers as vectors of 8-, 16-, 32- or 64-bit operands (called *packed* operands in Intel's terminology).

The SSE2 instruction set is 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 an `xmm` register and zeroing the others on a load).

Reorganize Data: Various permutations of 16- and 32-bit packed operands (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 byte-wise only.

Arithmetic: Add/Subtract on 8-, 16-, 32- and 64-bits; Multiply of 16-bit (high and low word returns, signed and unsigned, and fused multiply-adds) and 32-bit unsigned; Max/Min (signed 16-bit, unsigned 8-bit); Unsigned Averages (8/16-bit); Sum-of-differences on 8-bits.

3.2 SSSE3 (Supplementary SSE3) instructions

SSSE3 adds a few very useful instructions that assist with our vector programming.

`PALIGNR` ("packed align right", really a byte-wise shift): "`PALIGNR xmm` (i), `xmm` (j), k" shifts `xmm` (j) right by k bytes, and insert the k rightmost bytes of `xmm` (i) in the space vacated by the shift, with the result placed in `xmm` (i). Can be used to rotate an `xmm` register by bytes.

`PHADDx,PHSUBx` H means horizontal. Take as an example, the `PHADDW` instruction. If destination register `xmm` (i) starts out as $(x_0, x_1, \dots, x_7)$, the source register `xmm` (j) as $(y_0, y_1, \dots, y_7)$, then after "`PHADDW xmm` (i), `xmm` (j)", `xmm` (i) will hold

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

If there are eight vectors $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_7$, then after seven invocations of `PHADDW`, we can obtain $\left(\sum_j v_j^{(0)}, \sum_j v_j^{(1)}, \dots, \sum_j v_j^{(7)}\right)$ arranged in the right order.

`PSHUFB` If the source register is $(x_0, x_1, \dots, x_{15})$, and the destination register is $(y_0, y_1, \dots, y_{15})$, the result at position i is given by $x_{y_i \& 31}$. Note that we assume $x_{16} = \cdots = x_{31} := 0$.

`PMULHRSW` gives the *rounded* higher word of the product of two signed words in each of 8 positions. [SSE2 only has `PMULHW`, which gives the higher word in the product.]

The source register `xmm` (j) of each instruction above can be replaced by a 16-byte-aligned memory region. The interested reader is referred to Intel's own reference manual for further information on optimizing for the x86-64 architecture [27]. To our best knowledge SSE4 and SSE5 do not improve the matter greatly for us (except for SSE5's version of `PSHUFB`), so we skip their descriptions here.

3.3 Speeding Up In $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ via `PSHUFB`

`PSHUFB` enables us to do 16 simultaneous lookups at the same time in a table of 16. The basic way it helps with $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ arithmetic is by speeding up multiplication of a vector $\mathbf{v}$ by a scalar a.

We will use the following notation: if i, j are 2 bytes or nybbles each representing a value in $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$, then $i \cdot j$ will be the byte or nybble representing their product in the finite field.

$\mathbb{F}_{16}$, v is unpacked, 1 entry per byte: Make a table `TT` of 16 entries, each 128 bits, where the i-th entry contains $i * j$ in byte j. Load `TT[`a`]` into `xmm` (i), and do "`PSHUFB xmm` (i), $\mathbf{v}$".

$\mathbb{F}_{16}$, v is packed, 2 entry per byte: Same as above, but shift $\mathbf{v}$ bytewise right 4 to get one mask, then `AND` $\mathbf{v}$ with 16 `0xF0`'s to get the other mask. After `PSHUFB`'s, shift and combine with `OR`.

$\mathbb{F}_{16}$, a **is packed with 2 entries:** Table can be made 256 entries wide, capable of multiplying *two* scalars to one packed or unpacked vector at the same time.

$\mathbb{F}_{256}$: We need two 256-entry tables, for products of any byte by the values represented by the bytes `[0x00, 0x10, ..., 0xF0]`, and `[0x00, 0x01, ..., 0x0F]` respectively. Again, one `AND`, one shift, two `PSHUFB`'s, and one `OR` will dispatch 16 multiplications.

Both SSSE3 and bitslicing then accomplishes matrix-to-vector multiplication in the same way, and even the evaluation of MPKCs' public maps in the same way, requiring column-first matrices.

Evaluating public maps: We normally do $z_k = \sum_i w_i \left[P_{ik} + Q_{ik}w_i + \sum_{i<j} R_{ijk}w_j\right]$. But often it is better to compute a vector $\mathbf{c}$ with contents $[(w_i)_i,\ (w_iw_j)_{i\le j}]^T$, then $\mathbf{z}$ as a product of a $m \times n(n+3)/2$ matrix times and $\mathbf{c}$. That's how one would store the public coefficients.

In theory, it is good to bitslice in $\mathbb{F}_{16}$ when multiplying a scalar to a vector that is a multiple of 64 in length. Our tests show bitslicing a $\mathbb{F}_{16}$ scalar-to-64-long-vector to take a tiny bit less than 60 cycles on a core of a newer (45nm) Core 2 CPU. The corresponding `PSHUFB` code takes close to 48 cycles. For 128-long vectors, we can still bitslice using `xmm` registers. It comes out to around 70 cycles with bitslicing, against 60 cycles using `PSHUFB`. This demonstrate the usefulness of SSSE3 since these should be optimal cases for bitslicing.

Elimination or Solving a Matrix Equation: We can speed up Gaussian Elimination greatly because of fast row operations. [Here one should be wary of `gcc-4.2.0` sometimes emitting bad code. We once concluded that Gaussian for $\mathbb{F}_{16}$ was slower than Wiedemann.]

4 Code Components: Arithmetic in Odd Prime Field $\mathbb{F}_q$

4.1 Data Conversion between $\mathbb{F}_2$ and $\mathbb{F}_q$

The first problem with MPKCs over odd prime fields is the conversion between binary and base-q data. 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 $q^n > 2^\ell$.

Quadword (8-byte) unsigned integers in $[0, 2^{64} - 1]$ fit decently into 13 blocks in $\mathbb{F}_{31}$. So to transfer 128-, 192-, and 256-bit AES keys, we need at least 26, 39, and 52 $\mathbb{F}_{31}$ blocks, respectively.

Packing $\mathbb{F}_q$-blocks into binary can be more "wasteful" in the sense that it can use more bits than necessary, as long as the map is injective and convenient to compute. For example, we have opted for a very simple packing strategy in which every three $\mathbb{F}_{31}$ blocks are fit in a 16-bit word.

4.2 Basic Arithmetic Operations and Inversion mod q

In $\mathbb{F}_q$, the addition, subtraction, and multiplication operations are simply integer arithmetic operations mod q. The division instruction is always particularly time-consuming, so we replace it with multiplication using the following proposition [24]:

Proposition 1. *If M satisfies $2^{n+\ell} \le Md \le 2^{n+\ell} + 2^\ell$ for $2^{\ell-1} < d < 2^\ell$, then $\lfloor \frac{X}{d} \rfloor = \lfloor 2^{-\ell} \lfloor \frac{XM}{2^n} \rfloor \rfloor = \left\lfloor 2^{-\ell} \left(\left\lfloor \frac{X(M-2^n)}{2^n} \right\rfloor + X \right) \right\rfloor$ for $0 \le X < 2^n$. Note that the second equality, since often $M > 2^n$.*

Most of the time there is an arithmetic instruction that returns the "top n bits of the product of n-bit unsigned integers x and y", which achieves $\lfloor \frac{xy}{2^n} \rfloor$ and thus can be used to implement division by multiplication. For example we may take $n = 64$, $\ell = 5$, and $d = 31$: $Q = \lfloor \frac{1}{32} \left(\lfloor \frac{595056260442243601\,x}{2^{64}} \rfloor + x \right) \rfloor = x \text{ div } 31$, $\quad R = x - 31\,Q$, for an unsigned integer $x < 2^{64}$.

To invert one element in $\mathbb{F}_q$, we usually use a look-up table. In some cases, we need to invert many $\mathbb{F}_q$ elements at the same time. As will be described later, we vectorize most arithmetic operations using SSE2 and hence need to store the operands in `xmm` registers. Although eight table look-ups usually do not take long, getting the operands out of and into `xmm` registers can be troublesome. Instead, we can use a $(q-2)$-th power ("patched inverse") to get the inverse for a vector of elements. For example, after taking into account the possibility of overflow, we do the following to get a 29-th power to compute multiplicative inverses in $\mathbb{F}_{31}$ using 16-bit integers (`short int`):

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

Finally, if SSSE3 is available, inversion in a $\mathbb{F}_q$ for $q < 16$ is possible using one `PSHUFB`, and for $31 \geq q > 16$ using two `PSHUFB`'s and some masking.

Overall, the most important optimization is *avoiding unnecessary modulo operations* by delaying them as much as possible. To achieve this goal, we need to carefully track the size of the operands. SSE2 uses fixed 16- or 32-bit operands for most of its integer vector operations. In general, the use of 16-bit operands, either signed or unsigned, gives the best trade-off between modulo reduction frequency (wider operands allow for less frequent modulo operations) and parallelism (narrower operands allow more vector elements packed in an `xmm` register).

4.3 Vectorizing mod q using SSE2

Using the vectorized integer addition, subtraction, and multiplication provided by SSE2, we can easily execute multiple integer arithmetic operations simultaneously. Now the question is how we implement vectorized modulo operations described in Sec. 4.2. While SSE2 does provide instructions returning the upper word of a 16-bit-by-16-bit product, there are no facilities for carries, and hence it is difficult to guarantee a range of size q for a general q. It is then important to realize that *we do not always need the tightest range.* Minus signs are okay, as long as the absolute values are relatively small to avoid non-trivial modulo operations.

- $y = x - q \cdot \texttt{IMULHI}b \left(\left\lfloor \frac{2^b}{q} \right\rfloor, \left(x + \left\lfloor \frac{q-1}{2} \right\rfloor \right) \right)$, is guaranteed to return a value $y \equiv x \pmod q$ such that $|y| \leq q$ for general b-bit word arithmetic, 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 $-32768 \leq x \leq 32752$ by: $y = x - 31 \cdot \texttt{IMULHI16}\ (2114, x + 15)$. Here `IMULHI16` is implemented via the Intel intrinsic of `__mm_mulhi_epi16`.
- For I/O in $\mathbb{F}_{31}$, from y above we get a principal value between 0 and 30: $y' = y - 31\ \&\ (y \ggg 15)$, where & is the logical `AND` and $\ggg$ arithmetically shifts in the sign bit.
- There is a slightly faster version using *rounding*, when SSSE3 (with `PMULHRSW`) is available.

4.4 Matrix-vector Multiplication and Polynomial Evaluation

Core 2 and newer Intel CPUs have SSSE3 and can add horizontally within an `xmm` register, c.f., Sec. 3.2. Specifically, the matrix M can be stored in the row-major order. Each row is multiplied component-wise to the vector $\mathbf{v}$. We then use `PHADDW` to add horizontally and arrange the elements at the same time. Surprisingly, the convenience of having `PHADDW` available only makes at most a 10% difference for $q = 31$.

If we are restricted to using just SSE2, then it is advisable to store M in the column-major order and treat the matrix-to-vector product as taking a linear combination of the column vectors. For $q = 31$, each 16-bit component in $\mathbf{v}$ is copied eight times into every 16-bit word in an `xmm` register using an `__mm_set1` intrinsic, which takes three data-moving (shuffle) instructions, but still avoids the penalty for accessing the L1 cache. Finally we multiply this register into one column of M, eight components at a time, and accumulate.

Public maps are evaluated as in Sec. 3.3, except that We may further exploit PMADDWD as mentioned in Sec. 1.2, which 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 $(x_0,\dots,x_7)$ and $(y_0,\dots,y_7)$. 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 way we are able to save a few mod-q operations.

The Special Case of $\mathbb{F}_{31}$. We also pack keys (c.f., Sec. 4.1) 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 avoid writing the data to memory and execute the public map on the fly as we unpack to avoid cache contamination. It turns out that it does not slow things down too much. Further, we can do the messier 32-bit mod-q reduction *without* __mm_mulhi_epi32 via shifts as $2^5 = 1 \mod 32$.

4.5 Solving Systems of Linear Equations

Solving systems of linear equations are involved directly with TTS and Rainbow, as well as indirectly in the other schemes through taking inverses. Normally, one runs a Gaussian elimination, in which the elementary row operations can be sped up by SSE2.

One must notice here that during a Gaussian elimination, one needs to do frequent modular reductions, which rather slows things down from the otherwise expected speed. To elaborate, say we have an augmented matrix $[A|\mathbf{b}]$ modulo 31. For ease of doing elementary row operations, we naturally store the matrix in the row-major order. 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 (31^2), or at least up to around 250 if signed representation is used. So, to do the second column of eliminations, we need to reduce that column mod 31 before we can look up the correct coefficients. Note that reducing a single column by table look-up is no less expensive than reducing the entire matrix when the latter is not too large due to the overhead associated with moving data in and out of the xmm registers, so we end up reducing the entire matrix many times.

What can we do then? Well, we can switch to an iterative method like Wiedemann or Lanczos. To solve by Wiedemann a $n\times n$ system $A\mathbf{x}=\mathbf{b}$, one computes $\mathbf{z}A^i\mathbf{b}$ for $i=1\dots 2n$ for some given $\mathbf{z}$. Then computes the minimal polynomial from these elements in $\mathbb{F}_q$, using the Berlekamp-Massey algorithm. Assuming that this is the minimal polynomial of A, we have a solution.

It looks very counter-intuitive, since a Gaussian elimination does around $n^3/3$ multiplications in $\mathbb{F}_q$ but Wiedemann takes $2n^3$ for a dense matrix for the matrix-vector products alone, then need extra memory/time to store the partial results and run Berlekamp-Massey. Yet, each iteration we only need to reduce a single vector, not a matrix. That is the key observation and the tests show that Wiedemann (Lanczos too often fails) is significantly faster for convenient sizes and odd q.

4.6 Recommendation: $q = 31$

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 the best compromise, since it also allows us several convenient tower fields and easy packing conversions (close to $2^5 = 32$). This is verified empirically.

5 Arithmetic in $\mathbb{F}_{q^k}$

In a "big-field" or "two-field" variant of MPKC, 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. 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 efficient 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/inversion become much easier, which is evident from the example timings for a tower field in Tab. 1.

Microarchitecture	MULT	SQUARE	INV	SQRT	INV+SQRT
C2 (65nm)	234	194	2640	4693	6332
C2+ (45nm)	145	129	1980	3954	5244
K8 (Athlon 64)	397	312	5521	8120	11646
K10 (Phenom)	242	222	2984	5153	7170

Table 1: Clock cycle counts for various arithmetic operations in $\mathbb{F}_{31^{18}}$ implemented using SSE2

5.1 Multiplication and Squaring

When we have $\mathbb{F}_{q^k} \cong \mathbb{F}_q[t]/(t^k - a)$,The straightforward way to multiply is to take each x_i, copy it eight times, multiply by the correct y_i's using `PMULLW`, and then shift the result by the appropriate distances using `PALIGNR` (if SSSE3 is available) or unaligned load/stores/shifts (otherwise), 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 [29] or other more advanced multiplication algorithms. For example, we can 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 $(a_1 u + a_0)(b_1 u + b_0) = [(a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0]u + [a_1 b_1 t + a_0 b_0]$. 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

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

We often say "squaring is 0.8 times of a multiplication". Here we simply skip some of the iterations in the loops used for multiplication and weigh some of the other iterations double. Due to architectural differences, the ratio of the costs of squaring and multiplication measured *anywhere from as high as* 0.92 *to as low as* 0.75 for fields in the teens of $\mathbb{F}_{31}$ blocks.

5.2 Square and Other Roots

Today there are many ways to compute square roots in a finite field [3]. 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 (mod 4) field sizes, as working with a fixed field we can include precomputated tables with the program "for free". To recap, 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 a primitive solution to $g^{2^k} = 1$ 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. For example, in $\mathbb{F}_{31^9}$, we compute

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

for the square root in $\mathbb{F}_{31^9}$ by raising the $\frac{1}{4}\left(31^9 + 1\right)$-th power, since powers of 31 are easy.

Scheme	Result	PriKey	PubKey	KeyGen	PriMap	PubMap
RSA (1024 bits)		148 B	128 B	38.1 ms	1032.1 μs	24.8 μs
ECC (256 bits)		96 B	64 B	1.0 ms	1006.0 μs	1222.7 μs
4HFE-p (31,10)	25 B	2.2 KB	20 KB	11.3 ms	787.3 μs	4.4 μs
3HFE-p (31,9)	17 B	1.0 KB	6 KB	3.5 ms	81.0 μs	1.5 μs
C^*-p (pFLASH)	292 b	5.5 KB	72 KB	30.6 ms	850.7 μs	310.3 μs
3IC-p (31,18,1)	34 B	2.7 KB	22 KB	4.1 ms	290.2 μs	20.8 μs
Rainbow (31,16,16,8,16)	35 B	16.0 KB	30 KB	22.8 ms	17.9 μs	8.3 μs
TTS (31,24,20,20)	40 B	6.2 KB	34 KB	11.0 ms	13.3 μs	11.7 μs

Table 2: Current MPKCs and their traditional competitors compared on an Intel Core 2 Quad Q9550 processor

5.3 Multiplicative Inverse

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 $q^k - 2$.

For our specialized tower fields of characteristic 31, the extended Euclidean Algorithm is slower because after one 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* $2\times$–$3\times$.

Finally, we note that when we compute $\sqrt{X}$ and $1/X$ as high powers at the same time, we can share some exponentiation and save 10% of the work.

5.4 Equation Solving

We implement Cantor-Zassenhaus, today's standard algorithm to solve equations in odd-characteristic finite fields, to find 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 (i.e., 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.

6 Experiment Results

Some recent implementations of MPKCs over $\mathbb{F}_{2^k}$ are tested by Chen et al. [7]. We choose the following well-known schemes for comparison: HFE (an encryption scheme); 3IC-p, pFLASH, and Rainbow/TTS (all signature schemes). We summarize the characteristics and performances, measured on an Intel Core 2 Quad Q9550 processor running at 2.833 GHz, of these MPKCs and their traditional competitors in Tab. 2. The current MPKCs are over odd-characteristic fields except for C^* (pFLASH), which is over $\mathbb{F}_{16}$. The first two rows are their traditional competitors,

Scheme	Atom	C2	C2+	K8	K10
4HFE-p (31,10)	4732	2703	2231	8059	2890
3HFE-p (31,9)	528	272	230	838	259
C^*-p (pFLASH)	7895	2400	2450	5010	3680
3IC-p (31,18,1)	2110	822	728	1550	1410
3IC-p (16,32,1)	1002	456	452	683	600
Rainbow (31,16,16,8,16)	191	62	51	101	120
Rainbow (16,24,24,20)	147	61	48	160	170
Rainbow (256,18,12,12)	65	27	22	296	211
TTS (31,24,20,20)	78	38	38	65	72
TTS (16,24,20,20)		61	65	104	82
TTS (256,18,12,12)		31	36	69	46

Table 3: Speeds of private maps of MPKCs in number of kilo cycles on various x86 microarchitectures

the 1024-bit RSA and 256-bit ECC. The results clearly indicate that MPKCs can take advantage of the latest x86 vector instructions and hold their speeds against RSA and ECC.

Tab. 3 shows the speeds of the private maps of the MPKCs over binary vs. odd-characteristic fields on various x86 microarchitectures. As in Tab. 1, the C2 microarchitecture refers to the 65 nm Intel Core 2, C2+ the 45 nm Intel Core 2, K8 the AMD Athlon 64, and K10 the AMD Phenom processors. The results clearly indicate that even now MPKCs in odd-characteristic fields hold their own against prior MPKCs that are based in $\mathbb{F}_{2^k}$, if not generally faster, on various x86 microarchitectures.

7 Concluding Remarks

Given the results in Sec. 6 and the recent interest into the theory of algebraic attacks on odd-characteristic HFE, we think that odd-field MPKCs merit more investigation. Furthermore, today's FPGAs have many built-in multipliers and intellectual properties (IPs), as good integer multipliers are common for application-specific integrated circuits (ASICs). One excellent example of using the multipliers in FPGAs for PKCs is the work of Güneysu and Paar [25]. We believe our results can easily carry over to FPGAs as well as any other specialized hardware with a reasonable number of 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 [33]. The comparisons herein must of course be re-evaluated with each new instruction set and new silicon implementation, but we believe that the general trend stands on our side.

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] 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".

[4] O. Billet, J. Patarin, and Y. Seurin. Analysis of intermediate field systems. presented at SCC 2008, Beijing.

[5] A. Bogdanov, T. Eisenbarth, A. Rupp, and C. Wolf. Time-area optimized public-key engines: MQ-cryptosystems as replacement for elliptic curves? In *Proceedings of the 10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008)*, pages 45–61, Washington, DC, USA, August 2008.

[6] 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.

[7] 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 and J. Ding, editors, *PQCrypto*, volume 5299 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2008.

[8] N. Courtois. Algebraic attacks over $GF(2^k)$, application to HFE challenge 2 and SFLASH-v2. In *Public Key Cryptography — PKC 2004*, volume 2947 of *Lecture Notes in Computer Science*, pages 201–217. Feng Bao, Robert H. Deng, and Jianying Zhou (editors), Springer, 2004. ISBN 3-540-21018-0.

[9] 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.

[10] J. Ding, V. Dubois, B.-Y. Yang, C.-H. O. Chen, and C.-M. Cheng. Could SFLASH be repaired? In L. Aceto, I. Damgard, 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.

[11] 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`.

[12] J. Ding, J. Gower, and D. Schmidt. *Multivariate Public-Key Cryptosystems*. Advances in Information Security. Springer, 2006. ISBN 0-387-32229-9.

[13] 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.

[14] J. Ding, D. Schmidt, and F. Werner. Algebraic attack on hfe revisited. In *ISC 2008*, Lecture Notes in Computer Science. Springer. to appear.

[15] 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.

[16] 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`.

[17] 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.

[18] J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139:61–88, June 1999.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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`.

[25] 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.

[26] 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.

[27] Intel Corp. Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/design/processor/manuals/248966.pdf`, Nov. 2007.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] T. Moh. A public key system with signature and master key function. *Communications in Algebra*, 27(5):2207–2222, 1999. Electronic version: `http://citeseer/moh99public.html`.

[33] 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.

[34] 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.

[35] C. Wolf. *Multivariate Quadratic Polynomials in Public Key Cryptography.* PhD thesis, Katholieke Universiteit Leuven, 2005. `http://eprint.iacr.org/2005/393`.

[36] C. Wolf and B. Preneel. Taxonomy of public key schemes based on the problem of multivariate quadratic equations. Cryptology ePrint Archive, Report 2005/077, 12th of May 2005. `http://eprint.iacr.org/2005/077/`, 64 pages.

[37] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[38] 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.