SSE Implementation of Multivariate PKCs on Modern x86 CPUs

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 Lee[1], and Bo-Yin Yang[2]

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

Abstract. Multivariate Public Key Cryptosystems (MPKCs) are often touted as future-proofing against Quantum Computers. It also has been known for efficiency compared to "traditional" alternatives. However, this advantage seems to erode with the increase of arithmetic resources in modern CPUs and improved algorithms, especially with respect to Elliptic Curve Cryptography (ECC). In this paper, we show that *hardware advances do not just favor ECC.* Modern commodity CPUs also have many small integer arithmetic/logic resources, embodied by SSE2 or other vector instruction sets, that are useful for MPKCs. In particular, Intel's SSSE3 instructions can speed up both public and private maps over prior software implementations of Rainbow-type systems up to 4×. Furthermore, *MPKCs over fields of relatively small odd prime characteristics* can exploit SSE2 instructions, supported by most modern 64-bit Intel and AMD CPUs. For example, Rainbow over $\mathbb{F}_{31}$ can be up to 2× faster than prior implementations of similarly-sized systems over $\mathbb{F}_{16}$. Here a key advance is in using Wiedemann (as opposed to Gauss) solvers to invert the small linear systems in the central maps. We explain the techniques and design choices in implementing our chosen MPKC instances over 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 small multipliers, usable by odd-field MPKCs.

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

1 Introduction

Multivariate public-key cryptosystems (MPKCs) [35, 13] is a genre of PKCs whose public keys represent multivariate polynomials over a small field $\mathbb{K} = \mathbb{F}_q$:

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

C. Clavier and K. Gaj (Eds.): CHES 2009, LNCS 5747, pp. 33–48, 2009.

Polynomials $p_1, p_2, \ldots$ have (almost always) been quadratic since MPKCs came to public notice [30]. Since this is public-key cryptography, we can let $\mathcal{P}(\mathbf{0}) = \mathbf{0}$.

Of course, a random $\mathcal{P}$ would not be invertible by the legitimate user, so almost always $\mathcal{P} = T \circ \mathcal{Q} \circ S$ with 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" quadratic map $\mathcal{Q} : \mathbf{x} \mapsto \mathbf{y}$. The public key then comprise 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 information to determine 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 affected by the evolution of computer architecture.* Without going into any theory, we will discuss the implemention of MPKCs on today's commodity CPUs. *We will conclude that modern single-instruction-multiple-data (SIMD) units also make great cryptographic hardware for MPKCs, making them stay competitive speed-wise.*

1.1 History and Questions

Conventional wisdom used to be: "MPKCs replace arithmetic operations on large units (e.g., 1024+-bit integers in RSA, or 160+-bit integers in ECC) by faster operations on many small units." But the latter means many more memory accesses. People came to realize that eventually the memory latency and bandwidth would become the bottleneck of the performance of a microprocessor [7, 36].

The playing field is obviously changing. When MPKCs were initially proposed [25,30], 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 produce one 128-bit integer product every 2 cycles. ECC implementers quickly exploited these advances.

In stark contrast, a MOSTech 6502 CPU or an 8051 microcontroller from Intel multiplies in $\mathbb{F}_{256}$ in a dozen instruction cycles (using three table look-ups) — not too far removed from the latency of multiplying in $\mathbb{F}_{256}$ in modern x86.

This striking disparity came about because the number of gates available has been doubling every 18 to 24 months ("Moore's Law") for the last few decades. Compared to that, memory access speed increased at a snail's pace. Now the width of a typical arithmetic/logic unit is 64 bits, vector units are everywhere, and even FPGAs have hundreds of multipliers built in. On commodity hardware, the deck has never seemed so stacked against MPKCs or more friendly to RSA and ECC. Indeed, ECC over $\mathbb{F}_{2^k}$, the only "traditional" cryptosystem that has been seemingly left behind by advances in chip architectures, will get a new special struction from Intel soon — the new carryless multiplication [27].

Furthermore, we now understand attacks on MPKCs much better. In 2004, traditional signature schemes using RSA or ECC are much slower than TTS/4 and SFLASH [1, 10, 37], but the latter have both been broken [17, 18]. Although TTS/7 and 3IC-p seem ok today [8], the impending doom of SHA-1 [33] will force longer message digests and thus slower MPKCs while leaving RSA 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 do 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 x86 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 x86 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. Virtually all x86 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 a SIMD style.
   - Using MPKCs with a small odd prime base field $\mathbb{F}_q$ (say $\mathbb{F}_{31}$, as opposed to the usual $\mathbb{F}_{256}$ or $\mathbb{F}_{16}$) enables us to take advantage of vector hardware. Even with an overhead of conversion between bases, schemes over $\mathbb{F}_{31}$ is usually faster than an equivalent scheme over $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ without SSSE3.
   - MPKCs over $\mathbb{F}_q$ can still be faster than ECC or RSA. While q can be any prime power, it pays to tune to a small set of carefully chosen instances. In most of our implementations, we specialize to $q = 31$.
2. Certain CPUs have simultaneous look-ups from a small, 16-byte table:
   - all current Intel Core and Atom CPUs with SSSE3 instruction `PSHUFB`;
   - all future AMD CPUs, with SSE5 `PPERM` instruction (superset of `PSHUFB`);
   - IBM POWER derivatives — with AltiVec/VMX instruction `PERMUTE`.

   Scalar-to-vector multiply in $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$ can get around a 10× speed-up; MPKCs like TTS and Rainbow get a 4× factor or higher speed-up.

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. When solving a small and dense matrix equation in $\mathbb{F}_{31}$, iterative methods like Wiedemann may still beat straight Gaussian elimination.
2. $X \mapsto X^{q-2}$ may be a fast way to component-wise invert a vector over $\mathbb{F}_q^*$.
3. For big-field MPKCs, some fields (e.g., $\mathbb{F}_{31^{15}}$) admit *very fast arithmetic representations* — in such fields, inversion is again 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` (*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 [6], 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 [13] and numerous research papers in the literature.

2 Background on MPKCs

In this section, we summarize the MPKC instances that we will investigate. Using the notation in Sec. 1, we only need to describe the central map $\mathcal{Q}$ (M_S and M_T are square and invertible matrices, usu. resp. of dim $= n$ and m, respectively. To execute a private map, we replace the "minus" components if needed, invert T, invert $\mathcal{Q}$, invert S, and if needed verify a prefix/perturbation.

Most small-field MPKCs — TTS, Rainbow, oil-and-vinegar [11, 12, 17, 29] seem to behave the same over small odd prime fields and over $\mathbb{F}_{2^k}$. Big-field MPKCs in odd-characteristic were mentioned in [35], but not much researched until recently. In some cases 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. Some attacks are known or conjectured to be easier over binary fields than over small odd prime fields [5, 19, 9, 15], but never vice versa.

2.1 Rainbow and TTS Families of Digital Signatures

Rainbow($\mathbb{F}_q, o_1, \ldots, o_\ell$) is characterized as follows as a u-stage UOV [14, 17].

- 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 $v_l < i \leq v_{l+1}$, and all the x_j with $j \leq v_l$, 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. 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 [8, 17]. 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 as ($\mathbb{F}_{31}, 16, 16, 8, 16$). Note that in today's terminology, TTS is simply a Rainbow with sparse coefficients, which is faster but less understood.

2.2 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 $\mathbb{K}$ with $(\mathbb{F}_q)^n$ via an implicit bijective map $\phi : \mathbb{L} \to (\mathbb{F}_q)^n$ [34]. 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 [22], 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.

The following multi-variable HFE appeared in [5]. 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 higher 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.

It was recently conjectured that for odd characteristic, Gröbner basis attacks on HFE does not work as well [15]. Hence we try to implement multivariate HFEs over $\mathbb{F}_q$ for an odd q. We will be conservative here and enforce one prefixed zero block to block structural attacks at a q-time speed penalty.

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

C^* is the original Matsumoto-Imai scheme [30], 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 (ℓIC) can be considered as an improved extension of C^* [16]. Here we use the simple case where $\ell = 3$. In 3IC we also use 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 a lot in common [16,20,11]. 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 [11]. This is written as $C^{*-}\mathrm{p}(q, n, \alpha, r, s)$ or 3IC-p(q, n, r, s). Ding et al. recommend $C^{*-}\mathrm{p}(2^4, 74, 22, 1)$, also known as the "pFLASH" [11].

To invert 3IC-p over a field like $\mathbb{F}_{31^{18}}$, from (Y_1, Y_2, Y_3) we do the following.

1. Compute $A = Y_1Y_2$ [1 multiplication].
2. Compute $B = A^{-1}$ [1 inverse].
3. Compute $C = Y_3B = X_2^{-2}$ [1 multiplication].
4. Compute $D = C^{-1} = X_2^2$ and $\pm\sqrt{C} = X_2^{-1}$ [1 sqrt+inverse].
5. Multiply X_2^{-1} to Y_1, Y_2, and D [3 multiplications].

We note that for odd q, square roots are non-unique and slow.

3 Background on x86 Vector Instruction Set Extensions

The use of vector instructions to speed up MPKCs is known since the seminal Matsumoto-Imai works, in which bit slicing is suggested for MPKCs over $\mathbb{F}_2$ as

a form of SIMD [30]. Berbain et al. 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 [2]. Chen et al. extended this further to Gaussian elimination in $\mathbb{F}_{16}$, to be used for TTS [8].

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 [31]. This fell into obscurity after TTM was cryptanalyzed [21].

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 convention, 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, 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 *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 (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 to assist with our vector programming.

`PALIGNR` ("packed align right"): "`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. E.g., consider `PHADDW`. 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: $(x_0 + x_1, x_2 + x_3, x_4 + x_5, x_6 + x_7, y_0 + y_1, y_2 + y_3, y_4 + y_5, y_6 + y_7)$. From 8 vectors $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_7$, after seven invocations of `PHADDW`, we 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.

`PSHUFB` From a source $(x_0, x_1, \ldots, x_{15})$, and destination register $(y_0, y_1, \ldots, y_{15})$, the result at position i is $x_{y_i \bmod 32}$. Here x_{16} through x_{31} are taken to be 0.

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

The source register `xmm` (j) can usually be replaced by a 16-byte-aligned memory region. The interested reader is referred to Intel's manual for further information on optimizing for the x86-64 architecture [26]. To our best knowledge SSE4 do not improve the matter greatly for us, 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 look-ups 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 two bytes (in $\mathbb{F}_{256}$) or nybbles (in $\mathbb{F}_{16}$), each representing a field element, then $i * j$ will be the byte or nybble representing their product in the finite field.

$\mathbb{F}_{16}$, $\mathbf{v}$ 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), $\mathbf{v}$".

$\mathbb{F}_{16}$, $\mathbf{v}$ 2-packed per byte or $\mathbb{F}_{16}$, a 2-packed: Similar with shifts and ORs.

$\mathbb{F}_{256}$: Use two 256×128-bit tables, for products of any byte-value by bytes `[0x00, 0x10, ..., 0xF0]`, and `[0x00, 0x01, ..., 0x0F]`. One `AND`, one shift, 2 `PSHUFB`s, and one `OR` dispatches 16 multiplications.

Solving a Matrix Equation: We can speed up Gaussian elimination a lot on fast row operations. Note: Both SSSE3 and bit-slicing require column-first matrices for matrix-vector multiplication and evaluating MPKCs' public maps.

Evaluating public maps: We can do $z_k = \sum_i w_i \left[P_{ik} + Q_{ik} w_i + \sum_{i<j} R_{ijk} w_j\right]$. But on modern processors it is better to compute $\mathbf{c} := [(w_i)_i, (w_i w_j)_{i \le j}]^T$, then $\mathbf{z}$ as a product of a $m \times n(n+3)/2$ matrix (public key) and $\mathbf{c}$.

In theory, it is good to bit-slice in $\mathbb{F}_{16}$ when multiplying a scalar to a vector that is a multiple of 64 in length. Our tests show bit-slicing 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 bit-slice using `xmm` registers. It comes out to around 70 cycles with bit-slicing, against 60 cycles using `PSHUFB`. This demonstrate the usefulness of SSSE3 since these should be optimal cases for bit-slicing.

4 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 one 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

$\mathbb{F}_q$ operations for odd prime q uses many modulo-q. We almost always replace slow division instructions with multiplication as follows.

Proposition 1 ([23]). *If* $2^{n+\ell} \le Md \le 2^{n+\ell} + 2^\ell$ *for* $2^{\ell-1} < d < 2^\ell$, *then* $\left\lfloor \frac{X}{d} \right\rfloor = \left\lfloor 2^{-\ell} \left\lfloor \frac{XM}{2^n} \right\rfloor \right\rfloor = \left\lfloor 2^{-\ell} \left(\left\lfloor \frac{X(M-2^n)}{2^n} \right\rfloor + X \right) \right\rfloor$ *for* $0 \le X < 2^n$.

An instruction giving "top n bits of product of n-bit integers x, y" achieves $\lfloor \frac{xy}{2^n} \rfloor$ and thus can be used to implement division by multiplication. E.g.,when we take $n = 64$, $\ell = 5$, and $d = 31$, $Q = \left\lfloor \frac{1}{32} \left(\left\lfloor \frac{595056260442243601\, x}{2^{64}} \right\rfloor + x \right) \right\rfloor = x \text{ div } 31$, $R = x - 31\, Q$, for an unsigned integer $x < 2^{64}$. Note often $M > 2^n$ as here.

Inverting one element in $\mathbb{F}_q$ is usually via a look-up table. Often we need to invert simultaneously many $\mathbb{F}_q$ elements. As described later, we vectorize most arithmetic operations using SSE2 and hence need to store the operands in `xmm` registers. Getting the operands between `xmm` and general-purpose registers for table look-up is very troublesome. Instead, we can use a $(q-2)$-th power ("patched inverse") to invert a vector. For example, the following raises to the 29-th to find 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 $16 < q \le 31$ 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 operand sizes. 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 vectorized integer add, subtract, and multiply instructions provided by SSE2, we can easily execute multiple integer arithmetic operations simultaneously. A problem is how to implement vectorized modulo operations (cf. Sec. 4.2). While SSE2 does provide instructions returning the upper word of a 16-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.

- If `IMULHI`b returns "the upper half in a signed product of two b-bit words", $y = x - q \cdot \texttt{IMULHI}b\left(\left\lfloor \frac{2^b}{q} \right\rfloor, \left(x + \left\lfloor \frac{q-1}{2} \right\rfloor\right)\right)$ will return a value $y \equiv x \pmod{q}$ such that $|y| \leq q$ for b-bit word arithmetic, where $-2^{b-1} \leq x \leq 2^{b-1} - (q-1)/2$.
- For $q = 31$ and $b = 16$, we do better finding $y \equiv x \pmod{31}, -16 \leq y \leq 15$, 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 $\mathbb{F}_{31}$, the principal value between 0 and 30 is $y' = y - 31 \,\&\, (y \ggg 15)$, where & is the logical `AND`, and $\ggg$ arithmetically shifts in the sign bit.
- When SSSE3 is available, *rounding* with `PMULHRSW` is faster.

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 row-major. Each row is multiplied component-wise to the vector $\mathbf{v}$. Then `PHADDW` can add horizontally and arrange the elements at the same time. Surprisingly, this convenience 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 $(x_0, \ldots, x_7)$ 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 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 others through taking inverses. Normally, one runs a Gaussian elimination, where elementary row operations can be sped up by SSE2.

However, during a Gaussian elimination, one needs frequent modular reductions, which rather slows things down from the otherwise expected speed. Say we have an augmented matrix $[A|\mathbf{b}]$ modulo 31 in row-major order. Let us do elimination on the first column. Each entry in the remaining columns will now be of size up to about 1000 (31^2), or 250 if representatives are between ± 16.

To eliminate on the second column, we must reduce that column mod 31 before looking up the correct multipliers. 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.

We can switch to an iterative method like Wiedemann or Lanczos. To solve by Wiedemann an $n \times n$ system $A\mathbf{x} = \mathbf{b}$, one computes $\mathbf{z}A^i\mathbf{b}$ for $i = 1 \ldots 2n$ for some given $\mathbf{z}$. Then one computes the minimal polynomial from these elements in $\mathbb{F}_q$ using the Berlekamp-Massey algorithm.

It looks very counter-intuitive, as a Gaussian elimination does around $n^3/3$ field multiplications but Wiedemann takes $2n^3$ for a dense matrix for the matrix-vector products, plus extra memory/time to store the partial results and run Berlekamp-Massey. Yet in each iteration, we only need to reduce a single vector, not a whole matrix. That is the key observation and the tests show that Wiedemann is significantly faster for convenient sizes and odd q. Also, Wiedemann outperforms Lanczos because the latter fails too often.

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. It is particularly efficient if $p(t) = t^k - a$ for a small positive a, which is possible $k|(q-1)$ and in a few other cases. With a convenient p, the map $X \mapsto X^q$ in $\mathbb{L}$, becomes an easy precomputable linear map over $\mathbb{K} = \mathbb{F}_q$. Multiplication, division, and inversion all become much easier. See some example timing for such a tower field in Tab. 1.

Table 1. Cycle counts for various $\mathbb{F}_{31^{18}}$ arithmetic operations using SSE2

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

5.1 Multiplication and the S:M (Square:Multiply) Ratio

When $\mathbb{F}_{q^k} \cong \mathbb{F}_q[t]/(t^k - a)$, a straightforward way to multiply is to copy each x_i 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 some cases we need to tune the code. E.g., for $\mathbb{F}_{31^9}$, we multiply the **x**-vector by y_8 and the **y**-vector by x_8 with a convenient 8×8 pattern remaining.

For very large fields, we can use Karatsuba [28] or other more advanced multiplication algorithms. E.g. $\mathbb{F}_{31^{30}} := \mathbb{F}_{31^{15}}[u]/(u^2 - t)\mathbb{F}_{31^{15}} = \mathbb{F}_{31}[t]/(t^{15} - 3)$. 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

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

For ECC, often the rule-of-thumb is "**S=0.8M**". Here the **S:M** ratio *ranges from 0.75 to 0.92* for fields in the teens of $\mathbb{F}_{31}$-blocks depending on architecture.

5.2 Square and Other Roots

Today there are many ways to compute square roots in a finite field [3]. For field sizes $q = 4k + 3$, 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 $4k + 1$ field sizes, as working with a fixed field we can include pre-computed 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 pre-computed 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, via fast 31^k-th powers, in $\mathbb{F}_{31^9}$ we take square roots by raising to the $\frac{1}{4}\left(31^9 + 1\right)$-th power

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. $\texttt{result} := \texttt{temp1} * \texttt{temp2} * \left((\texttt{temp2})^{31}\right)^{31}$;

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 an 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 to 3.*

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 in an Odd-Characteristic Field $\mathbb{L} = \mathbb{F}_{q^k}$

Cantor-Zassenhaus solves a univariate degree-d equation $u(X) = 0$ as follows. The work is normally cubic in $\mathbb{L}$-multiplications and quintic in $(d, k, \lg q)$ overall.

1. Replace $u(X)$ by $\gcd(u(X), X^{q^k} - X)$ so that u factors completely in $\mathbb{L}$.
   (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 $i = 2, 3, \ldots, k$, then $X^{q^k} \bmod u(X)$.
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.

6 Experiment Results

Clearly, we need to avoid too large q (too many reductions mod q) and too small q (too large arrays). 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.

Some recent implementations of MPKCs over $\mathbb{F}_{2^k}$ are tested by Chen et al. [8] We choose the following well-known schemes for comparison: HFE (an encryption scheme); pFLASH, 3IC-p, and Rainbow/TTS (all signature schemes). We summarize the characteristics and performances, measured using SUPERCOP-20090408 [4] 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 pFLASH, which is over $\mathbb{F}_{16}$. The table is divided into two regions: top for encryption schemes and bottom for signature schemes, with the traditional competitors (1024-bit RSA and 160-bit ECC) listed first. 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 fields on various x86 microarchitectures. As in Tab. 1, the C2 microarchitecture

Table 2. Current MPKCs vs. traditional competitors on an Intel C2Q Q9550

Scheme	Result	PubKey	PriKey	KeyGen	PubMap	PriMap
RSA (1024 bits)	128 B	128 B	1024 B	27.2 ms	26.9 μs	806.1 μs
4HFE-p (31,10)	68 B	23 KB	8 KB	4.1 ms	6.8 μs	659.7 μs
3HFE-p (31,9)	67 B	7 KB	5 KB	0.8 ms	2.3 μs	60.5 μs
RSA (1024 bits)	128 B	128 B	1024 B	26.4 ms	22.4 μs	813.5 μs
ECDSA (160 bits)	40 B	40 B	60 B	0.3 ms	409.2 μs	357.8 μs
C^*-p (pFLASH)	37 B	72 KB	5 KB	28.7 ms	97.9 μs	473.6 μs
3IC-p (31,18,1)	36 B	35 KB	12 KB	4.2 ms	11.7 μs	256.2 μs
Rainbow (31,24,20,20)	43 B	57 KB	150 KB	120.4 ms	17.7 μs	70.6 μs
TTS (31,24,20,20)	43 B	57 KB	16 KB	13.7 ms	18.4 μs	14.2 μs

Table 3. MPKC private map timings in kilocycles on various x86 microarchitectures

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)	141	61	65	104	82
TTS (256,18,12,12)	104	31	36	69	46

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 believe 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 [24]. 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, such as NVIDIA, AMD/ATI, and

Intel [32] graphics processors coming. 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.

Acknowledgements. CC thanks the National Science Council of Taiwan for support under grants NSC 97-2628-E-001-010- and Taiwan Information Security Center (grant 98-2219-E-011-001), BY for grant NSC 96-2221-E-001-031-MY3.

References

1. Akkar, M.-L., Courtois, N.T., Duteuil, R., Goubin, L.: A fast and secure implementation of SFLASH. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 267–278. Springer, Heidelberg (2002)
2. Berbain, C., Billet, O., Gilbert, H.: Efficient implementations of multivariate quadratic systems. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 174–187. Springer, Heidelberg (2007)
3. Bernstein, D.J.: Faster square roots in annoying finite fields. In: High-Speed Cryptography (2001) (to appear), `http://cr.yp.to/papers.html#sqroot`
4. Bernstein, D.J.: SUPERCOP: System for unified performance evaluation related to cryptographic operations and primitives (April 2009), `http://bench.cr.yp.to/supercop.html`
5. Billet, O., Patarin, J., Seurin, Y.: Analysis of intermediate field systems. Presented at SCC 2008, Beijing (2008)
6. Bogdanov, A., Eisenbarth, T., Rupp, A., Wolf, C.: Time-area optimized public-key engines: MQ-cryptosystems as replacement for elliptic curves? In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 45–61. Springer, Heidelberg (2008)
7. Burger, D., Goodman, J.R., Kägi, A.: Memory bandwidth limitations of future microprocessors. In: Proceedings of the 23rd annual international symposium on Computer architecture, pp. 78–89 (1996)
8. Chen, A.I.-T., Chen, C.-H.O., Chen, M.-S., Cheng, C.-M., Yang, B.-Y.: Practical-sized instances of multivariate pkcs: Rainbow, TTS, and ℓIC-derivatives. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 95–108. Springer, Heidelberg (2008)
9. Courtois, N.: Algebraic attacks over $GF(2^k)$, application to HFE challenge 2 and SFLASH-v2. In: Bao, F., Deng, R., Zhou, J. (eds.) PKC 2004. LNCS, vol. 2947, pp. 201–217. Springer, Heidelberg (2004)
10. Courtois, N., Goubin, L., Patarin, J.: SFLASH: Primitive specification (second revised version), Submissions, Sflash, 11 pages (2002), `https://www.cosic.esat.kuleuven.be/nessie`
11. Ding, J., Dubois, V., Yang, B.-Y., Chen, C.-H.O., Cheng, C.-M.: Could SFLASH be repaired? In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 691–701. Springer, Heidelberg (2008)
12. Ding, J., Gower, J.: Inoculating multivariate schemes against differential attacks. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T.G. (eds.) PKC 2006. LNCS, vol. 3958, pp. 290–301. Springer, Heidelberg (2006), `http://eprint.iacr.org/2005/255`

13. Ding, J., Gower, J., Schmidt, D.: Multivariate Public-Key Cryptosystems. In: Advances in Information Security. Springer, Heidelberg (2006) ISBN 0-387-32229-9
14. Ding, J., Schmidt, D.: Rainbow, a new multivariable polynomial signature scheme. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 164–175. Springer, Heidelberg (2005)
15. Ding, J., Schmidt, D., Werner, F.: Algebraic attack on hfe revisited. In: Wu, T.-C., Lei, C.-L., Rijmen, V., Lee, D.-T. (eds.) ISC 2008. LNCS, vol. 5222, pp. 215–227. Springer, Heidelberg (2008)
16. Ding, J., Wolf, C., Yang, B.-Y.: ℓ-invertible cycles for multivariate quadratic public key cryptography. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 266–281. Springer, Heidelberg (2007)
17. Ding, J., Yang, B.-Y., Chen, C.-H.O., Chen, M.-S., Cheng, C.-M.: New differential-algebraic attacks and reparametrization of rainbow. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 242–257. Springer, Heidelberg (2008), http://eprint.iacr.org/2008/108
18. Dubois, V., Fouque, P.-A., Shamir, A., Stern, J.: Practical cryptanalysis of SFLASH. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 1–12. Springer, Heidelberg (2007)
19. Faugère, J.-C., Joux, A.: Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 44–60. Springer, Heidelberg (2003)
20. Fouque, P.-A., Macario-Rat, G., Perret, L., Stern, J.: Total break of the ℓIC- signature scheme. In: Cramer, R. (ed.) PKC 2008. LNCS, vol. 4939, pp. 1–17. Springer, Heidelberg (2008)
21. Goubin, L., Courtois, N.T.: Cryptanalysis of the TTM cryptosystem. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 44–57. Springer, Heidelberg (2000)
22. Granboulan, L., Joux, A., Stern, J.: Inverting HFE is quasipolynomial. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 345–356. Springer, Heidelberg (2006)
23. Granlund, T., Montgomery, P.: Division by invariant integers using multiplication. In: Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 61–72 (1994), http://www.swox.com/~tege/divcnst-pldi94.pdf
24. Güneysu, T., Paar, C.: Ultra high performance ecc over nist primes on commercial fpgas. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 62–78. Springer, Heidelberg (2008)
25. Imai, H., Matsumoto, T.: Algebraic methods for constructing asymmetric cryptosystems. In: Calmet, J. (ed.) AAECC 1985. LNCS, vol. 229, pp. 108–119. Springer, Heidelberg (1986)
26. Intel Corp. Intel 64 and IA-32 architectures optimization reference manual (November 2007), http://www.intel.com/design/processor/manuals/248966.pdf
27. Intel Corp. Carryless multiplication and its usage for computing the GCM mode. (2008), http://software.intel.com/en-us/articles/carry/less-multiplication-and-%its-usage-for-computing-the-gcm-mode
28. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Doklady Akad. Nauk SSSR 145, 293–294 (1962); Translation in Physics-Doklady. 7, 595–596 (1963)
29. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced Oil and Vinegar signature schemes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 206–222. Springer, Heidelberg (1999)

30. Matsumoto, T., Imai, H.: Public quadratic polynomial-tuples for efficient signature verification and message-encryption. In: Günther, C.G. (ed.) EUROCRYPT 1988. LNCS, vol. 330, pp. 419–545. Springer, Heidelberg (1988)
31. Moh, T.: A public key system with signature and master key function. Communications in Algebra 27(5), 2207–2222 (1999), Electronic version, http://citeseer/moh99public.html
32. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. ACM Transactions on Graphics 27(18) (August 2008)
33. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full sha-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
34. Wolf, C.: Multivariate Quadratic Polynomials in Public Key Cryptography. PhD thesis, Katholieke Universiteit Leuven (2005), http://eprint.iacr.org/2005/393
35. Wolf, C., Preneel, B.: Taxonomy of public key schemes based on the problem of multivariate quadratic equations. Cryptology ePrint Archive, Report 2005/077, 64 pages, May 12 (2005),http://eprint.iacr.org/2005/077/
36. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. Computer Architecture News 23(1), 20–24 (1995)
37. Yang, B.-Y., Chen, J.-M.: Building secure tame-like multivariate public-key cryptosystems: The new TTS. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 518–531. Springer, Heidelberg (2005)