Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$

Charles Bouillaguet[1], Hsieh-Chung Chen[2], Chen-Mou Cheng[3], Tung Chou[3], Ruben Niederhagen[3,4], Adi Shamir[1,5], and Bo-Yin Yang[2]

[1] Ecole Normale Supérieure, Paris, France
charles.bouillaguet@ens.fr
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan
{kc,by}@crypto.tw
[3] National Taiwan University, Taipei, Taiwan
{doug,blueprint}@crypto.tw
[4] Technische Universiteit Eindhoven, The Netherlands
ruben@polycephaly.org
[5] Weizmann Institute of Science, Israel
adi.shamir@weizmann.ac.il

Abstract. We analyze how fast we can solve general systems of multivariate equations of various low degrees over $\mathbb{F}_2$; this is a well known hard problem which is important both in itself and as part of many types of algebraic cryptanalysis. Compared to the standard exhaustive search technique, our improved approach is more efficient both asymptotically and practically. We implemented several optimized versions of our techniques on CPUs and GPUs. Our technique runs more than 10 times faster on modern graphic cards than on the most powerful CPU available. Today, we can solve 48+ quadratic equations in 48 binary variables on a 500-dollar NVIDIA GTX 295 graphics card in 21 minutes. With this level of performance, solving systems of equations supposed to ensure a security level of 64 bits turns out to be feasible in practice with a modest budget. This is a clear demonstration of the computational power of GPUs in solving many types of combinatorial and cryptanalytic problems.

Keywords: multivariate polynomials, solving systems of equations, exhaustive search, parallelization, Graphic Processing Units (GPUs).

Extended Version of this paper is at `eprint.iacr.org/2010/313`.

1 Introduction

Solving a system of m nonlinear polynomial equations in n variables over $\mathbb{F}_q$ is a natural mathematical problem that has been investigated by various research communities. The cryptographers are among the interested parties since an NP-complete problem whose random instances seem hard could be used to design cryptographic primitives, as witness the development of multivariate cryptography in the last few decades, using one-way trapdoor functions such as HFE, SFLASH, and QUARTZ [12,20,21], as well as stream ciphers such as `QUAD` [4]. Conversely, in "algebraic cryptanalysis" one distills from a cryptographic primitive a system of multivariate polynomial equations with the secret among the variables. This does not break AES as first advertised, but does break KeeLoq [11], for a recent example, and find a faster collision on 58-round SHA-1 [24].

Since the pioneering work by Buchberger [9], Gröbner-basis techniques have been the most prominent tool for this problem, especially after the emergence of faster algorithms such as $\mathbf{F_4}$ or $\mathbf{F_5}$ [15,16], which broke the first HFE challenge [17]. The cryptographic community independently rediscovered some of the ideas underlying efficient Gröbner-basis algorithms as of the XL algorithm [13] and its variants. They also introduced techniques to deal with special cases, particularly that of sparse systems [1,23].

In this paper we take a different path, namely improving the standard and seemingly well-understood exhaustive search algorithm. When the system consists of n randomly chosen quadratic equations in n variables, all the known solution techniques have exponential complexity. In particular, Gröbner-basis methods have an advantage on very overdetermined systems (with many more equations than unknowns) and systems with certain algebraic "weaknesses", but were shown to be exponential on "generic" enough systems in [2,3]. In addition, the computation of a Gröbner basis is often a memory-bound process; since memory is more expensive than time at the scale of interest, such sophisticated techniques can be inferior in practice when compared to simple testing of all the possible solutions, which uses almost no memory.

For "generic" quadratic systems, experts believe [2,25] that Gröbner basis methods will go up to degree D_0, which is the minimum possible D where the coefficient of t^D in $(1+t)^n(1+t^2)^{-m}$ goes negative, and then require the solution of a system of linear equations with $T \gtrsim \binom{n}{D_0-1}$ variables. This will take at least $\mathrm{poly}(n) \cdot T^2$ bit-operations, assuming we can afford a sufficiently large amount of memory and that we can solve such a linear system of equations with non-negligible probability in $O(N^{2+o(1)})$ time for N variables. For example, if we assume we can operate a Wiedemann solver on a $T \times T$ submatrix of the extended Macaulay matrix of the original system, then the polynomial is $3n(n-1)/2$. When $m = n = 200$, $D_0 = 25$, making the value of T exceeds 2^{102}; even taking into consideration guessing before solving [6,26], we can still easily conclude that Gröbner-basis methods would not outperform exhaustive search in the practically interesting range of $m = n \leq 200$.

The questions we address are therefore: how far can we go, on both theoretical and practical sides, by pushing exhaustive search further? Is it possible to design more efficient exhaustive search algorithms? Can we get better performance using different hardware such as GPUs? Is it possible to solve *in practice*, with a modest budget, a system of 64 equations in 64 unknowns over $\mathbb{F}_2$? Less than 15 years ago, this was considered so difficult that it even underlied the security of a particular signature scheme [19]. Intuitively, some people may consider an algebraic attack that reduces a cryptosystem to 64 equations of degree 4 in 64 $\mathbb{F}_2$-variables to be a successful practical attack. However, the matter is not that easily settled because the complexity of a naïve exhaustive search algorithm would actually be *much higher* than 2^{64}: simply testing all the solutions in a naïve way results in $2 \cdot \binom{64}{4} \cdot 2^{64} \approx 2^{84}$ logical operations, which would make the attack hardly feasible even on today's best available hardware.

Our Contribution. Our contribution is twofold. On the theoretical side, we present a new type of exhaustive search algorithm which is both asymptotically and practically faster than existing techniques. In particular, we show that finding *all* zeroes of a single degree-d polynomial in n variables requires just $d \cdot 2^n$ bit operations. We then extend this technique and show how to find all the common zeroes of m random quadratic polynomials in $\log_2 n \cdot 2^{n+2}$ bit operations, which is only slightly higher. Surprisingly, this complexity is *independent of the number of equations* m.

On the practical side, we have implemented our new algorithms on x86 CPUs and on NVIDIA GPUs. While our CPU implementation is fairly optimized using vector instructions, our GPU implementation running on one single NVIDIA GTX 295 graphics card runs up to 13 times faster than the CPU implementation using all four cores of an Intel quad-code Core i7 at 3 GHz, one of the fastest CPUs currently available. Today, we can solve 48+ quadratic equations in 48 binary variables using just an NVIDIA GTX 295 graphics card in 21 minutes. This device is available for about \$500. It would be 36 minutes for cubic equations and two hours for quartics. The 64-bit signature challenge [19] can thus be broken with 10 such cards in 3 months, using a budget of \$5000. Even taking into account Moore's law, this is still quite an achievement.

Table 1. Performance results for $n = 48$ and projected budgets for solving $n = 64$ in one month

Time (minutes)			Testing platform				#cores	est. cost
$d = 2$	$d = 3$	$d = 4$	GHz	Arch.	Name	USD	(#used)	(USD)
1217	2686	3191	2.2	K10	Phenom 9550	120	4(1)	54,000
1157	1992	2685	2.3	K10+	Opteron 2376	184	4(1)	113,316
142	240	336	2.3	K10+	Opteron 2376×2	368	8(8)	
780	1364	1819	2.4	C2	Xeon X3220	210	4(1)	60,720
671	1176	1560	2.83	C2+	Core2 Q9550	225	4(1)	55,575
179	294	390	2.83	C2+	Core2 Q9550	225	4(4)	
761	1279	1856	2.26	Ci7	Xeon E5520	385	4(1)	78,720
95	154	225	2.26	Ci7	Xeon E5520×2	770	8(8)	
41	73	271	1.3	G200	GTX 280	n/a	240	n/a
21	36	126	1.25	G200	GTX 295	500	480	15,500

In contrast, the implementation of F_4 in `MAGMA-2.16`, often cited as the best Gröbner-basis solver *commercially* available today, will completely use up 64 GB of RAM in solving just 25 cubic equations in as many $\mathbb{F}_2$-variables. We have also tested it with overdefined systems, for which Gröbner-basis algorithms are known to work better. While it does not run out of memory, the results are not satisfying: 2.5 hours to solve 20 cubic equations in 20 variables, half an hour for 45 quadratic equations in 30 variables, and 7 minutes for 60 quadratic equations in 30 variables on one 2.2-GHz Opteron core. Some very recent improvements on Gröbner-basis solvers have reported speed-up over MAGMA $\mathbf{F_4}$ of two- to five-fold [10]. However, even with such significant improvements, Gröbner-basis solvers do not seem to be able to compete with exhaustive search algorithms in this range, as each of the above is solved in a split second using negligible amount of memory on the same CPU by the latter.

Implications. The new exhaustive search algorithm can be used as a black box in cryptanalysis that needs to solve quadratic equations. This includes, for instance, several algorithms for the Isomorphism of Polynomials problem [7,22], as well as attacks that rely on such algorithms, e.g., [8].

We also show with a concrete example that (relatively simple) computations requiring 2^{64} operations can be easily carried out in practice with readily available hardware and a modest budget. Lastly, we highlight the fact that GPUs have been used successfully by the cryptographic community to obtain very efficient implementations of combinatorial algorithms or cryptanalytic attacks, in addition to the more numeric-flavored cryptanalysis algorithm demonstrated by the implementation of the ECM factorization algorithm on GPUs [5].

Organization of the Paper. Section 2 establishes a formal framework of exhaustive search algorithms including useful results on Gray Codes and derivatives of multivariate polynomials. Known exhaustive search algorithms are reviewed in Section 3. Our algorithm to find the zeroes of a single polynomial of any degree is given in Section 4, and it is extended to find the common zeroes of a collection of polynomials in Section 5. Section 6 describes the two platforms on which we implemented the algorithm, and Section 7 describes the implementation and performance evaluation results.

2 Generalities

In this paper, we will mostly be working over the finite vector space $(\mathbb{F}_2)^n$. The canonical basis is denoted by $(e_0, \ldots, e_{n-1})$. We use $\oplus$ to denote addition in $(\mathbb{F}_2)^n$, and $+$ to denote integer addition. We use $i \ll k$ (resp. $i \gg k$) to denote binary left-shift (resp. right shift) of the integer i by k bits.

Gray Code. Gray Codes play a crucial role in this paper. Let us denote by $b_k(i)$ the index of the k-th lowest-significant bit set to 1, or -1 if the hamming weight of i is less than k. For example, $b_k(0) = -1, b_1(1) = 0, b_1(2) = 1$ and $b_2(3) = 1$.

Definition 1. $\textsc{GrayCode}(i) = i \oplus (i \gg 1)$.

Lemma 1. *For* $i \in \mathbb{N}$*:* $\textsc{GrayCode}(i + 1) = \textsc{GrayCode}(i) \oplus e_{b_1(i+1)}$.

Derivatives. Define the $\mathbb{F}_2$ *derivative* $\frac{\partial f}{\partial i}$ of a polynomial with respect to its i-th variable as $\frac{\partial f}{\partial i} : \mathbf{x} \mapsto f(\mathbf{x} + e_i) + f(\mathbf{x})$. Then for any vector $\mathbf{x}$, we have:

$$f(\mathbf{x} + e_i) = f(\mathbf{x}) + \frac{\partial f}{\partial i}(\mathbf{x}) \tag{1}$$

If f is of total degree d, then $\frac{\partial f}{\partial i}$ is a polynomial of degree $d - 1$. In particular, if f is quadratic, then $\frac{\partial f}{\partial i}$ is an affine function. In this case, it is easy to isolate the constant part (which is a constant in $\mathbb{F}_2$) : $c_i = \frac{\partial f}{\partial i}(0) = f(e_i) + f(0)$. Then, the function $\mathbf{x} \mapsto \frac{\partial f}{\partial i}(\mathbf{x}) + c_i$ is by definition a linear form and can be represented by a vector $D_i \in (\mathbb{F}_2)^n$. More precisely, we have $D_i[j] = f(e_i + e_j) + f(e_i) + f(e_j) + f(0)$. Then equation (1) becomes:

$$f(\mathbf{x} + e_i) = f(\mathbf{x}) + D_i \cdot \mathbf{x} + c_i \tag{2}$$

Enumeration Algorithms. We are interested in *enumeration algorithms*, *i.e.*, algorithms that evaluate a polynomial f over all the points of $(\mathbb{F}_2)^n$ to find its zeroes. Such an enumeration algorithm is composed of two functions: INIT and NEXT. $\textsc{Init}(f, x_0, k_0)$ returns a *State* containing all the information the enumeration algorithm needs for the remaining operations. The resulting State is configured for the evaluation of f over $x_0 \oplus (\textsc{GrayCode}(i) \ll k_0)$, for increasing values of i. $\textsc{Next}(\mathit{State})$ advance to the next value and updates State. Three values can be directly read from the state: $\mathit{State}.\mathbf{x}$, $\mathit{State}.\mathbf{y}$ and $\mathit{State}.i$. These are linked at all times by $\mathit{State}.\mathbf{y} = f(\mathit{State}.\mathbf{x})$, $\mathit{State}.\mathbf{x} = x_0 \oplus (\textsc{GrayCode}(\mathit{State}.i) \ll k_0)$, $\textsc{Next}(\mathit{State}).i = \mathit{State}.i + 1$. Finding all the zeroes of f is then achieved with the loop shown below.

```
1: procedure ZEROES($f$)
2:   $State \leftarrow \textsc{Init}(f, 0, 0)$
3:   for $i$ from $0$ to $2^n - 1$
4:     if $State.\mathbf{y} = 0$ then $State.\mathbf{x}$ is a zero of $f$
5:     NEXT($State$)
6:   end for
7: end procedure
```

3 Known Techniques for Quadratic Polynomials

We briefly discuss the enumeration techniques known to the authors.

Naïve Evaluation. The simplest way to implement an enumeration algorithm is to evaluate the polynomial f from scratch at each point of $(\mathbb{F}_2)^n$. This requires two AND per quadratic monomial, and (almost) as many XORs. Since the evaluation takes place many times for the same f with different values of the variables, we will usually assume that the polynomial can be *hard-coded*, *i.e.*, that it is not necessary to include the terms for which $a_{ijk} = 0$. Each call to NEXT would then require at most $n(n+1)$ bit operations, half-AND and half-XOR (not counting the cost of enumerating $(\mathbb{F}_2)^n$, *i.e.*, incrementing a counter). This can be improved a bit, by factoring out the monomials:

$$f(\mathbf{x}) = \sum_{i=0}^{n-1} x_i \cdot \left(\sum_{j=i}^{n-1} a_{ij} \cdot x_j \right) + c \tag{3}$$

The bit-operation count falls down to $n(n+3)/2$, and in general for degree-d polynomials to a sum dominated by $\binom{n}{d}$. This method is simple but not without its advantages, chiefly (a) insensitivity to the order in which the points of $(\mathbb{F}_2)^n$ are enumerated, and (b) we can bit-slice and get a speed up of nearly ω, where ω is the maximum width of the CPU logical instructions.

The Folklore Differential Technique. It was pointed out in Sec. 2 that once $f(\mathbf{x})$ is known, computing $f(\mathbf{x} + e_i)$ amounts to compute $\frac{\partial f}{\partial i}(\mathbf{x})$, and this derivative happens to be a linear function which can be efficiently evaluated by computing a vector-vector product and a few scalar additions. This strongly suggests to evaluate f on $(\mathbb{F}_2)^n$

using a *Gray Code*, *i.e.*, an ordering of the elements of $(\mathbb{F}_2)^n$ such that two consecutive elements differ in only one bit. This leads to the algorithm shown below.

```
1: function INIT($f$, _, _)
2:     $i \leftarrow 0$
3:     $\mathbf{x} \leftarrow 0$
4:     $\mathbf{y} \leftarrow f(0)$
5:     For all $0 \le k \le n-1$,
           initialize $c_k$ and $D_k$
6: end function
```

(a) Initialize

```
1: function NEXT($State$)
2:     $i \leftarrow i + 1$
3:     $k = b_1(i)$
4:     $\mathbf{z} \leftarrow$ VECTORVECTORPROD$(D_k, \mathbf{x}) \oplus c_k$
5:     $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}$
6:     $\mathbf{x} \leftarrow \mathbf{x} \oplus e_k$
7: end function
```

(b) Update

We believe this technique to be folklore, and in any case it appears more or less explicitly in the existing literature. Each call to NEXT requires n ANDs, as well as $n+2$ XORs, which makes a total bit operation count of $2(n+1)$. This is about $n/4$ times less than the naive method. Note that when we describe an enumeration algorithm, the variables that appear inside NEXT are in fact implicit functions of $State$. The dependency has been removed to lighten the notational burden.

4 A Faster Recursive Algorithm for Any Degree

We now describe one of the main contributions of this paper, a new algorithm which is both asymptotically and practically faster than standard exhaustive search in enumerating the solutions of one polynomial equation, as summarized by Theorem 1 below.

Theorem 1. *All the zeroes of a single multivariate polynomial f in n variables of degree d can be found in essentially $d \cdot 2^n$ bit operations (plus a negligible overhead), using n^{d-1} bits of read-write memory, and accessing n^d bits of constants, after an initialization phase of negligible complexity $\mathcal{O}\left(n^{2d}\right)$.*

The proof is given in the full version.

Construction of the Recursive Enumeration Algorithm. We will construct an enumeration algorithm in two stages. First, if f is of degree 0, then we only need to "enumerate" through all vectors by updating with $\mathbf{x} \leftarrow \mathbf{x} \oplus e_{b_1(i)}$ at the i-th step.

When f is of higher degree, we need a little more effort. The main idea is that in the folklore differential algorithm of Sec. 3, the computation of $\mathbf{z}$ essentially amounts to evaluate $\frac{\partial f}{\partial k}$ on something that looks like a Gray Code. We may then use the enumeration algorithm recursively on $\frac{\partial f}{\partial k}$, since it is a polynomial of strictly smaller degree. The resulting algorithm is shown below.

It is not difficult to see that the complexity of NEXT is $\mathcal{O}(d)$, where d is the degree of f. The temporal complexity of INIT is n^d times the time of evaluating f, which is itself upper-bounded by n^d and its spatial complexity is also of order $\mathcal{O}\left(n^d\right)$. This means that the complexity of the algorithm is $\mathcal{O}\left(d \cdot 2^n + n^{2d}\right)$. When $d = 2$, this is about n times faster than the algorithm described in Sec. 3.

```
1: function INIT($f$, $k_0, x_0$)
2:     $i \leftarrow 0$
3:     $\mathbf{x} \leftarrow x_0$
4:     $\mathbf{y} \leftarrow f(x_0)$
5:     for $i$ from 0 to $2^n - 1$
6:         $x_0' \leftarrow x_0 \oplus \text{GRAYCODE}\left(2^{k+k_0}\right)$
7:         $D[k] \leftarrow \text{INIT}\left(\frac{\partial f}{\partial k + k_0}, k + k_0 + 1, x_0'\right)$
8:     end for
9: end function

1: function NEXT($State$)
2:     $i \leftarrow i + 1$
3:     $k = b_1(i)$
4:     $\mathbf{x} \leftarrow \mathbf{x} \oplus e_{k+k_0}$
5:     $\mathbf{y} \leftarrow \mathbf{y} \oplus D[k].\mathbf{y}$
6:     NEXT($D[k]$)
7: end function
```

(a) General Setting

```
1: $\mathbf{y} \leftarrow f(0)$
2: if $\mathbf{y} = 0$ then 0 is a zero of $f$
3: $\mathbf{z}[0] \leftarrow c_0$
4: $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}[0]$
5: for $u$ from 1 to $n - 1$
6:     if $\mathbf{y} = 0$ then GRAYCODE($2^u - 1$) is a zero
7:     $\mathbf{z}[u] \leftarrow D_u[u-1] \oplus c_u$
8:     $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}[u]$
9:     for $v$ from 0 to $2^u - 2$
10:        if $\mathbf{y} = 0$ then GRAYCODE($2^u + v$) is a zero
11:        $k \leftarrow b_1(2^u + v + 1)$
12:        $\ell \leftarrow b_2(2^u + v + 1)$
13:        $\mathbf{z}[k] \leftarrow \mathbf{z}[k] \oplus D_k[\ell]$
14:        $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}[k]$
15:    end for
16: end for
```

(b) Unrolled version for quadratic f

5 Common Zeroes of Several Multivariate Polynomials

We will use several time the following simple idea: all the techniques we discussed above perform a sequence of operations that is independent of the coefficients of the polynomials. Therefore, m instances of (say) algorithm in Sec. 4 could be run in parallel on $f_1, \ldots, f_m$. All the parallel runs would execute the same instruction on different data, making it efficient to implement on vector or SIMD architectures. In each iteration of the main loop, it is easy to check if *all* the polynomials vanished on the current point of $(\mathbb{F}_2)^n$. Evaluating all the m (quadratic) polynomials in parallel using algorithm in Sec. 4 would take $2m2^n$ bit operations. The point of this section is that it is possible to do much better than this.

Note that for the sake of simplicity, we limit our discussion to the case of quadratic polynomials (this case being the most relevant in practice). Our objective is now to show the following result.

Theorem 2. *The common zeroes of m (random) quadratic polynomials in n variables can be found after having performed in expectation $\log_2 n \cdot 2^{n+2}$ bit operations.*

We sketch a proof (a complete one given in the extended version) to the theorem. Let us introduce a useful notation. Given an ordered set U, we denote the common zeroes of $f_1, \ldots, f_m$ belonging to U by $Z([f_1, \ldots, f_m], U)$. Let us also denote $Z_0 = (\mathbb{F}_2)^n$, and $Z_i = Z([f_i], Z_{i-1})$. It should be clear that $Z = Z_m$ is the set of common zeroes of the polynomials, and therefore the object we wish to obtain.

The key idea is to compute Z_k using k parallel runs of algorithm in Sec. 4, and then computing $Z_{k+1}, \ldots, Z_m$ one by one. Computing Z_k requires $2k2^n$ bit operations. It then remains to compute Z_m from Z_k, and to find the best possible value of k. If we use the naïve evaluation strategy with early abort to compute Z_m from Z_k, then it can be shown that the best value of k is $k = 2\log_2 n - \log_2\log_2 n + o(\log\log n)$, yielding a total complexity of about $8\log_2 n \cdot 2^n$. In general, for degree-d systems, the same reasoning would lead to a total complexity of about $4d \cdot \log_2 n \cdot 2^n$. In practice, it makes more sense to choose k to be the word width on a microprocessor in order to use the hardware in the most efficient way.

6 A Brief Description of the Hardware Platforms

6.1 Vector Units on x86-64

The most prevalent SIMD (single instruction, multiple data) instruction set today is SSE2, available on all current Intel-compatible CPUs. SSE2 instructions operate on 16 architectural `xmm` registers, each of which is 128-bit wide. We use integer operations, which treat `xmm` registers as vectors of 8-, 16-, 32- or 64-bit operands.

The highly non-orthogonal SSE instruction set includes Loads and Stores (to/from `xmm` registers, memory — both aligned and unaligned, and traditional registers), Packing/Unpacking/Shuffling, Logical Operations (`AND, OR, NOT, XOR`, Shifts Left, Right Logical and Arithmetic — bit-wise on units and byte-wise on the entire `xmm` register), and Arithmetic (add, substract, multiply, max-min) with some or all of the arithmetic widths. The interested reader is referred to Intel and AMD's manuals for details on these instructions, and to references such as [18] for throughput and latencies.

6.2 G2xx-Series Graphics Processing Units from NVIDIA

We choose NVIDIA's G2xx GPUs as they have the least hostile GPU parallel programming environment called CUDA (Compute Unified Device Architecture). In CUDA, we program in the familiar C/C++ programming language plus a small set of GPU extensions.

An NVIDIA GPU contains anywhere from 2–30 streaming multiprocessors (MPs). There are 8 ALUs (streaming processors or SPs in market-speak) and two super function units (SFUs) on each MP. A top-end "GTX 295" card has two GPUs, each with 30 MPs, hence the claimed "480 cores". The theoretical throughput of each SP per cycle is one 32-bit integer or floating-point instruction (including add/subtract, multiply, bitwise AND/OR/XOR, and fused multiply-add), and that of an SFU 2 floating-point multiplications or 1 special operation. The arithmetic units have 20+-stage pipelines.

Main memory is slow and forms a major bottleneck in many applications. The read bandwidth from memory on the card to the GPU is only one 32-bit read per cycle per MP and has a latency of $>$ 200 cycles. To ease this problem, the MP has a register file of 64 KB (16,384 registers, max. of 128 per thread), a 16-bank shared memory of 16 KB, and an 8-KB cache for read-only access to a declared "constant region" of at most 64 KB. Every cycle, each MP can achieve one read from the constant cache, *which can broadcast to many thread at once*.

Each MP contains a scheduling and dispatching unit that can handle a large number of lightweight threads. However, the decoding unit can only decode once every 4 cycles. *This is typically 1 instruction, but certain common instructions are "half-sized", so two such instructions can be issued together if independent.* Since there are 8 SPs in an MP, CUDA programming is always on a Single Program Multiple Data basis, where a "warp" of threads (32) should be executing the same instruction. If there is a branch which is taken by some thread in a warp but not others, we are said to have a "divergent" warp; from then on only part of the threads will execute until all threads in that warp are executing the same instruction again. Further, as the latency of a typical instruction is about 24 cycles, NVIDIA recommends a minimum of 6 warps on each MP, although it is sometimes possible to get acceptable performance with 4 warps.

7 Implementations

We describe the structure of our code, the approximate cost structure, our design choices and justify what we did. Our implementation code always consists of three stages:

Partial Evaluation: We substitute all possible values for s variables $(x_{n-s}, \ldots, x_{n-1})$ out of n, thus splitting the original system into 2^s smaller systems, of w equations each in the remaining $(n - s)$ variables $(x_0, \ldots, x_{n-s-1})$, and output them in a form that is suitable for ...

Enumeration Kernel: Run the algorithm of Sec. 4 to find all candidate vectors $\mathbf{x}$ satisfying w equations out of m ($\approx 2^{n-w}$ of them), which are handed over to ...

Candidate Checking: Checking possible solutions $\mathbf{x}$ in remaining $m - w$ equations.

7.1 CPU Enumeration Kernel

Typical code fragments from the unrolled inner loops can be seen below:

(a) quadratics, C++ x86 instrinsics

```
...
diff0 ^= deg2_block[ 1 ];
res ^= diff0;
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^155);
...
```

(b) quadratics, x86 assembly

```
.L746:
  movq       976(%rsp), %rax   //
  pxor       (%rax), %xmm2     // d_y ^= C_yz
  pxor       %xmm2, %xmm1      // res ^= d_y
  pxor       %xmm0, %xmm0      //
  pcmpeqw    %xmm1, %xmm0      // cmp words for eq
  pmovmskb   %xmm0, %eax       // movemask
  testw      %ax, %ax          // set flag for branch
  jne        .L1266            // if needed, check and
.L747:                         // comes back here
```

```
.L1624:
  movq     2616(%rsp), %rax   // load C_yza
  movdqa   2976(%rsp), %xmm0  // load d_yz
  pxor     (%rax), %xmm0      // d_yz ^= C_yza
  movdqa   %xmm0, 2976(%rsp)  // save d_yz
  pxor     8176(%rsp), %xmm0  // d_y ^= d_yz
  pxor     %xmm0, %xmm1       // res ^= d_y
  movdqa   %xmm0, 8176(%rsp)  // save d_y
  pxor     %xmm0, %xmm0       //
  pcmpeqw  %xmm1, %xmm0       // cmp words for eq
  pmovmskb %xmm0, %eax
  testw    %ax, %ax           // ...
  jne      .L2246             // branch to check
.L1625:                       // and comes back
```

(c) cubics, x86 assembly

```
...
diff[0] ^= deg3_ptr1[0];
diff[325] ^= diff[0];
res ^= diff[325];
Mask = _mm_cmpeq_epi16(res, zero);
mask = _mm_movemask_epi8(Mask);
if(mask) check(mask, idx, x^2);
...
```

(d) cubics, C++ x86 instrinsics

testing. All zeroes in one byte, word, or dword in a XMM register can be tested cheaply on x86-64. We hence wrote code to test 16 or 32 equations at a time. Strangely enough, even though the code above is for 16 bits, the code for checking 32/8 bits at the same time is nearly identical, the only difference being that we would subtitute the intrinsics `_mm_cmpeq_epi32/8` for `_mm_cmpeq_epi16` (leading to the SSE2 instruction `pcmpeqd/b` instead of `pcmpeqw`). Whenever one of the words (or double words or bytes, if using another testing width) is non-zero, the program branches away and queues the candidate solution for checking.

unrolling. One common aspect of our CPU and GPU code is deep unrolling by upwards of $1024\times$ to avoid the expensive bit-position indexing. To illustrate with quartics as an example, instead of having to compute the positions of the four rightmost non-zero bits in every integer, we only need to compute the first four rightmost non-zero bits in bit 10 or above, then fill in a few blanks. This avoids most of the indexing calculations and all the calculations involving the most commonly used differentials.

We wrote similar Python scripts to generate unrolled loops in C and CUDA code. Unrolling is even more critical with GPU, since divergent branching and memory accesses are prohibitively expensive.

7.2 GPU Enumeration Kernel

register usage. Fast memory is precious on GPU and register usage critical for CUDA programmers. Our algorithms' memory complexity grows exponentially with the degree d, which is a serious problem when implementing the algorithm for cubic and quartic systems, compounded by the immaturity of NVIDIA's `nvcc` compiler which tends to allocate more registers than we expected.

Take quartic systems as an example. Recall that each thread needs to maintain third derivatives, which we may call d_{ijk} for $0 \leq i < j < k < K$, where K is the number of variables in each small system. For $K = 10$, there are 120 d_{ijk}'s and we cannot waste all our registers on them, especially as all differentials are not equal — d_{ijk} is accessed with probability $2^{-(k+1)}$.

Our strategy for register use is simple: Pick a suitable bound u, and among third differentials d_{ijk} (and first and second differentials d_i and d_{ij}), put the most frequently used — i.e., all indices less than u — in registers, and the rest in device memory (which can be read every 8 instructions without choking). We can then control the number of registers used and find the best u empirically.

fast conditional move. We discovered during implementation an undocumented feature of CUDA for G2xx series GPUs, namely that `nvcc` reliably generates conditional (predicated) move instructions, dispatched with exceptional adeptness.

```
...
xor.b32 $r19, $r19, c0[0x000c]       // d_y^=d_yz
xor.b32 $p1|$r20, $r17, $r20
mov.b32 $r3, $r1
mov.b32 $r1, s[$ofs1+0x0038]
xor.b32 $r4, $r4, c0[0x0010]
xor.b32 $p0|$r20, $r19, $r20         // res^=d_y
@$p1.eq mov.b32 $r3, $r1
@$p1.eq mov.b32 $r1, s[$ofs1+0x003c]
xor.b32 $r19, $r19, c0[0x0000]
xor.b32 $p1|$r20, $r4, $r20
@$p0.eq mov.b32 $r3, $r1             // cmov
@$p0.eq mov.b32 $r1, s[$ofs1+0x0040] // cmov
...
```

(a) `decuda` result from cubin

```
 ...
diff0 ^= deg2_block[ 3 ];   // d_y^=d_yz
res ^= diff0;               // res^=d_y
if( res == 0 ) y = z;       // cmov
if( res == 0 ) z = code233; // cmov
diff1 ^= deg2_block[ 4 ];
res ^= diff1;
if( res == 0 ) y = z;
if( res == 0 ) z = code234;
diff0 ^= deg2_block[ 0 ];
res ^= diff0;
if( res == 0 ) y = z;
if( res == 0 ) z = code235;
 ...
```

(b) CUDA code for a inner loop fragment

Consider, for example, the code displayed above right. According to our experimental results, the repetitive 4-line code segments average less than three SP (streamprocessor) cycles. However, `decuda` output of our program shows that each such code segment corresponds to at least 4 instructions including 2 XORs and 2 conditional moves [as marked in above left]. The only explanation is that conditional moves can be dispatched by the SFUs (Special Function Units) so that the total throughput can exceed 1 instruction per SP cycle. Further note that the annotated segment on the right corresponds to actual instructions far apart because *an NVIDIA GPU does opportunistic dispatching but is nevertheless a purely in-order architecture,* so proper scheduling must interleave instructions from different parts of the code.

testing. The inner loop for GPUs differs from CPUs due to the fast conditional moves.

Here we evaluate 32 equations at a time using Gray code. The result is used to set a flag if it happens to be all zeroes. We can now conditional move of the index based on the flag to a register variable `z`, and at the end of the loop write `z` out to global memory.

However, how can we tell if there are too many (here, *two*) candidate solutions in one small subsystem? Our answer to that is to use a buffer register variable `y` and a second conditional move using the same flag. At the end of the thread, (`y`, `z`) is written out to a specific location in device memory and sent back to the CPU.

Now subsystems which have all zero constant terms are automatically satisfied by the vector of zeroes. Hence we note them down during the partial evaluation phase include the zeros with the list of candidate solutions to be checked, and never have to worry about for all-zero candidate solution. The CPU reads the two doublewords corresponding to `y` and `z` for each thread, and:

1. `z==0` means no candidate solutions,
2. `z!=0` but `y==0` means exactly one candidate solution, and
3. `y!=0` means 2+ candidate solutions (necessitating a re-check).

7.3 Checking Candidates

Checking candidate solutions is always done on CPU because the programming involves branching and hence is difficult on a GPU even with that available. However, the checking code for CPU enumeration and GPU enumeration is different.

CPU. With the CPU, the check code receives a list of candidate solutions. Today the maximum machine operation is 128-bit wide. Therefore we should collect solutions into groups of 128 possible solutions. We would rearrange 128 inputs of n bits such that they appear as n `__int128`'s, then evaluate one polynomial for 128 results in parallel using 128-bit wide ANDs and XORs. After we finish all candidates for one equation, go through the results and discard candidates that are no longer possible. Repeat the result for the second and any further equations (cf. Sec. 3).

We need to transpose a bit-matrix to achieve the effect of a block of w inputs n-bit long each, to n machine-words of w-bit long. This looks costly, however, there is an SSE2 instruction `PMOVMSKB` (packed-move-mask-bytes) that packs the top bit of each byte in an XMM register into a 16-bit general-purpose register *with 1 cycle throughput*. We combine this with simultaneous shifts of bytes in an XMM and can, for example, on a K10+ transpose a 128-batch of 32-bit vectors (0.5kB total) into 32 `__int128`'s in about 800 cycles, or an overhead of 6.25 cycles per 32-bit vector. In general the transposition cost is at most a few cycles per byte of data, negligible for large systems.

GPU. As explained above, for the GPU we receive a list with 3 kinds of entries:

1. The knowledge that there are two or more candidate solutions within that same small system, with only the position of the last one in the Gray code order recorded.
2. A candidate solution (and no other within the same small system).
3. Marks to subsystems that have all zero constant terms.

For Case 1, we take the same small system that was passed into the GPU and run the Enumerative Kernel subroutine in the CPU code and find all possible small systems.

Since most of the time, there are exactly two candidate solutions, we expected the Gray code enumeration to go two-thirds of the way through the subsystem. Merge remaining candidate solutions with those of Case 2+3, collate for checking in a larger subsystem if needed, and pass off to the same routine as used in the CPU above. Not unexpectedly, the runtime is dominated by the thread-check case, since those does millions of cycles for two candidate solutions (most of the time).

7.4 Partial Evaluation

The algorithm for Partial Evaluation is for the most part the same Gray Code algorithm as used in the Enumeration Kernel. Also the highest degree coefficients remain constant, need no evaluation and and can be shared across the entire Enumeration Kernel stage. As has been mentioned in the GPU description, these will be stored in the *constant memory*, which is reasonably cached on NVIDIA CUDA cards. The other coefficients can be computed by Gray code enumeration, so for example for quadratics we have $(n - s) + 2$ XOR per w bit-operations and per substitution. In all, the cost of the Partial Evaluation stage for w' equations is $\sim 2^s \frac{w'}{8} \left(\binom{n-s}{d-1} + (\text{smaller terms}) \right)$ byte memory writes. The only difference in the code to the Enumerative Kernel is we write out the result (smaller systems) to a buffer, and *check for a zero constant term only* (to find all-zero candidate solutions).

Peculiarities of GPUS. Many warps of threads are required for GPUs to run at full speed, hence we must split a kernel into many threads, the initial state of each small system being provided by Partial Evaluation. In fact, for larger systems on GPUs, we do two stages of partial evaluation because

1. there is a limit to how many threads can be spawned, and how many small systems the device memory can hold, which bounds how small we can split; *but*
2. increasing s decreases the fast memory pressure; and
3. a small systems reporting two or more candidate solutions is costly, yet we can't run a batch check on a small system with only one candidate solution — hence, an intermediate partial evaluation so we can batch check with fewer variables.

7.5 More Test Data and Discussion

Some minor points which the reader might find useful in understanding the test data, a full set of which will appear in the extended version.

Candidate Checking. **The check code is now 6–10% of the runtime.** In theory (cf. Sec. 3) evaluation should start with a script which hard-wires a system of equations into C and compiling to an excutable, eliminating half of the terms, and leading to $\binom{n-s}{d}$ SSE2 (half XORs and half ANDs) operations to check one equation for $w = 128$ inputs. The check code can potentially become more than an order of magnitude faster. We do not (yet) do so presently, because compiling may take more time than the checking code. However, we may want to go this route for even larger systems, as the overhead from testing for zero bits, re-collating the results, and wasting due to the number of candidate solutions is not divisible by w would all go down proportionally.

Without hard-wiring, the running time of the candidate check is dominated by loading coefficients. E.g., for quartics with 44 variables, 14 pre-evaluated, K10+ and Ci7 averages 4300 and 3300 cycles respectively per candidate. With each candidate averaging 2 equations of $\binom{44-14}{4}$ terms each, the 128-wide inner loop averages about 10 and 7.7 cycles respectively per term to accomplish 1 PXOR and 1 PAND.

Partial Evaluation. We point out that Partial Evaluation also reduces the complexity of the Checking phase. The simplified description in Sec. 5 implies the cost of checking each candidate solution to be $\approx \frac{1}{w}\binom{n}{d}$ instructions. But we can get down to $\approx \frac{1}{w}\binom{n-s}{d}$ instructions by partially evaluating $w' > w$ equations and storing the result for checking. For example, when solving a quartic system with $n = 48$, $m = 64$, the best CPU results are $s = 18$, and we cut the complexity of the checking phase by factor of at least $4\times$ even if it was not the theoretical $7\times$ (i.e., $\binom{n}{d}/\binom{n-s}{d}$) due to overheads.

The Probability of Thread-Checking for GPUs. If we have n variables, pre-evaluate s, and check w equations via Gray Code, then the probability of a subsystem with 2^{n-s} vectors including at least two candidates $\approx \binom{2^{n-s}}{2}(1-2^{-w})^{2^{n-s}}(2^{-w})^2 \approx 1/2^{2(s+w-n)+1}$, provided that $n < s+w$. As an example, for $n = 48$, $s = 22$, $w = 32$, the thread-recheck probability is about 1 in 2^{13}, and we must re-check about 2^9 threads using Gray Code. This pushes up the optimal s for GPUs.

Architecture and Differences. All our tests with a huge variety of machines and video cards show that the kernel time in cycles per attempt is almost a constant of the architecture, and the speed-up in multi-cores is almost completely linear for almost all modern hardware. So we can compute the time complexity given the architecture, the frequency, the number of cores, and n. The marked cycle count difference between Intel and AMD cores is explained by Intel dispatching *three* XMM (SSE2) logical instructions to AMD's *two* per cycle and handling branch prediction and caching better.

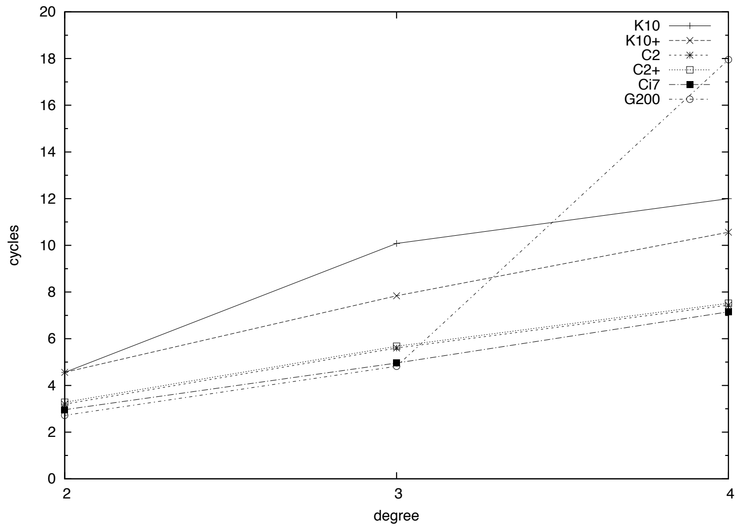

Fig. 1. Cycles per candidate tested for degree 2,3 and 4 polynomials

As the Degree d increases. We plot how many cycles is taken by the inner loop (which is 8 vectors per core for CPUs and 1 vector per SP for GPUs) on different architectures in Fig. 1. As we can see, all except two architectures have inner loop cycle counts that are increasing roughly linearly with the degree. The exceptions are the AMD K10 and NVIDIA G200 architectures, which is in line with fast memory pressure on the NVIDIA GPUs and fact that K10 has the least cache among the CPU architectures.

More Tuning. We can conduct a Gaussian elimination among the m equations and such that $m/2$ selected terms in $m/2$ of the equations are all zero. We can of course make this the most commonly used coefficients (i.e., $c_{01}, c_{02}, c_{12}, \ldots$ for the quadratic case). The corresponding XOR instructions can be removed from the code by our code generator. This is not yet automated and we have to test everything by hand. However, this clearly leads to significant savings. On GPUs, we have a speed up of 21% on quadratic cases, 18% for cubics, and 4% for quadratics. [The last is again due to the memory problems.]

Table 2. Efficiency comparison: cycles per candidate tested on one core

$n = 32$			$n = 40$			$n = 48$			Testing platform			
$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	GHz	Arch.	Name	USD
0.58	1.21	1.41	0.57	1.27	1.43	0.57	1.26	1.50	2.2	K10	Phenom9550	120
0.57	0.91	1.32	0.57	0.98	1.31	0.57	0.98	1.32	2.3	K10+	Opteron2376	184
0.40	0.65	0.95	0.40	0.70	0.94	0.40	0.70	0.93	2.4	C2	Xeon X3220	210
0.40	0.66	0.96	0.41	0.71	0.94	0.41	0.71	0.94	2.83	C2+	Core2 Q9550	225
0.50	0.66	1.00	0.38	0.65	0.91	0.37	0.62	0.89	2.26	Ci7	Xeon E5520	385
2.87	4.66	15.01	2.69	4.62	17.94	2.72	4.82	17.95	1.296	G200	GTX280	n/a
2.93	4.90	14.76	2.70	4.62	15.54	2.69	4.57	15.97	1.242	G200	GTX295	500

Notes and Acknowledgements

C. Bouillaguet thanks Jean Vuillemin for helpful discussions. The Taiwanese authors thank Ming-Shing Chen for assistance with programming and fruitful discussion, Taiwan's National Science Council for partial sponsorship under grants NSC96-2221-E-001-031-MY3, 98-2915-I-001-041, and 98-2219-E-011-001 (Taiwan Information Security Center), and Academia Sinica for the Career Development Award. Questions and esp. corrections about the extended version should be addressed to `by@crypto.tw`.

References

1. Bard, G.V., Courtois, N.T., Jefferson, C.: Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers, `http://eprint.iacr.org/2007/024`
2. Bardet, M., Faugère, J.-C., Salvy, B.: On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In: Proc. Int'l Conference on Polynomial System Solving, pp. 71–74 (2004) INRIA report RR-5049

3. Bardet, M., Faugère, J.-C., Salvy, B., Yang, B.-Y.: Asymptotic expansion of the degree of regularity for semi-regular systems of equations. In: Proc. MEGA 2005 (2005)
4. Berbain, C., Gilbert, H., Patarin, J.: QUAD: A practical stream cipher with provable security. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 109–128. Springer, Heidelberg (2006)
5. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on graphics cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2010)
6. Bettale, L., Faugére, J.-C., Perret, L.: Hybrid approach for solving multivariate systems over finite fields. J. Math. Crypto. 3(3), 177–197 (2009)
7. Bouillaguet, C., Faugére, J.-C., Fouque, P.-A., Perret, L.: Differential-algebraic algorithms for the isomorphism of polynomials problem, `http://eprint.iacr.org/2009/583`
8. Bouillaguet, C., Fouque, P.-A., Joux, A., Treger, J.: A family of weak keys in HFE (and the corresponding practical key-recovery), `http://eprint.iacr.org/2009/619`
9. Buchberger, B.: Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal. PhD thesis, Innsbruck (1965)
10. Buchmann, J., Cabarcas, D., Ding, J., Mohamed, M.S.E.: Flexible Partial Enlargement to Accelerate Gröbner Basis Computation over $\mathbb{F}_{\not\neq}$. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT 2010. LNCS, vol. 6055, pp. 69–81. Springer, Heidelberg (2010)
11. Courtois, N., Bard, G.V., Wagner, D.: Algebraic and slide attacks on Keeloq. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 97–115. Springer, Heidelberg (2008)
12. Courtois, N., Goubin, L., Patarin, J.: SFLASH: Primitive specification (second revised version) (2002), `https://www.cosic.esat.kuleuven.be/nessie`
13. Courtois, N.T., Klimov, A., Patarin, J., Shamir, A.: Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 392–407. Springer, Heidelberg (2000), Extended ver., `http://www.minrank.org/xlfull.pdf`
14. de Bruijn, N.: Asymptotic methods in analysis. 2nd edition. Bibliotheca Mathematica. Vol. 4., 200 p. P. Noordhoff Ltd. XII, Groningen (1961)
15. Faugère, J.-C.: A new efficient algorithm for computing Gröbner bases (F_4). J. of Pure and Applied Algebra 139, 61–88 (1999)
16. Faugère, J.-C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In: ACM ISSAC 2002, pp. 75–83 (2002)
17. 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)
18. Fog, A.: Instruction Tables. Copenhagen University, College of Engineering, Lists of Instruction Latencies, Throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs (February 2010),
`http://www.agner.org/optimize/instruction_tables.pdf`
19. Patarin, J.: Asymmetric cryptography with a hidden monomial. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 45–60. Springer, Heidelberg (1996)
20. Patarin, J.: Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of asymmetric algorithms. In: Maurer, U.M. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 33–48. Springer, Heidelberg (1996), Extended ver.:
`http://www.minrank.org/hfe.pdf`
21. Patarin, J., Courtois, N., Goubin, L.: QUARTZ, 128-bit long digital signatures. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 282–297. Springer, Heidelberg (2001), `http://www.minrank.org/quartz/`

22. Patarin, J., Goubin, L., Courtois, N.: Improved algorithms for Isomorphisms of Polynomials. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 184–200. Springer, Heidelberg (1998); Extended ver.: http://www.minrank.org/ip6long.ps
23. Raddum, H.: MRHS equation systems. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 232–245. Springer, Heidelberg (2007)
24. Sugita, M., Kawazoe, M., Perret, L., Imai, H.: Algebraic cryptanalysis of 58-round SHA-1. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 349–365. Springer, Heidelberg (2007)
25. Yang, B.-Y., Chen, J.-M.: Theoretical analysis of XL over small fields. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 277–288. Springer, Heidelberg (2004)
26. Yang, B.-Y., Chen, J.-M., Courtois, N.: On Asymptotic Security Estimates in XL and Gröbner Bases-Related Algebraic Cryptanalysis. In: López, J., Qing, S., Okamoto, E. (eds.) ICICS 2004. LNCS, vol. 3269, pp. 401–413. Springer, Heidelberg (2004)