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

Charles Bouillaguet[1], Hsieh-Chung Kevin Chen[2], Chen-Mou Cheng[3],
Tony (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. Modern graphic cards allows our technique to run more than 10 times faster than the most powerful CPU available. Today, we can solve 48+ quadratic equations in 48 binary variables on a NVIDIA GTX 295 video card (USD 500) 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 power of GPUs in solving many types of combinatorial and cryptanalytic problems.
Keywords: multivariate polynomials, system-solving, parallelization, Graphic Processing Units (GPUs)

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 given much attention by various research groups including the cryptographic community. The interest of the latter in this problem has two sources. On the one hand, since the problem is NP-complete (and random instances seem hard), it could be used to design cryptographic primitives. This led to the development of multivariate cryptography in the last few decades, using one-way trapdoor functions such as HFE, SFLASH, and QUARTZ [10, 18, 19], as well as stream ciphers such as QUAD [4]. On the other hand, it often seems appealing to try to break a cryptographic primitive by expressing the secret to be found as the solution to a system of multivariate polynomial equations. This failed to break the AES block cipher, but succeeded against other block ciphers such as KeeLoq [9], and yielded a faster collision attack on 58 rounds of SHA-1 [22].

Since pioneering work by Buchberger [8], 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 $\mathbf{F_5}$ [13, 14], which broke the first HFE challenge [15]. The cryptographic community independently rediscovered some of the ideas underlying efficient Gröbner-basis algorithms as of the XL algorithm [11] and its variants. They also introduced techniques to deal with special cases, particularly that of sparse systems [1, 21].

In this paper we take a different path, namely improving the standard and 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, 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 [2,23] 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 $\text{poly}(n) \cdot T^2$ bit-operations if we assume the existence of sufficiently large 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, assume that 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 $m = n = 200$, $D_0 = 25$, the value of T exceeds 2^{102}, and thus we may deduce that Gröbner-basis methods would never 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 the theoretical and practical side, 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 [17]. 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* the 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 implemented our new algorithms on x86 CPUs and on nVidia GPUs. While our CPU implementation is fairly optimized using SIMD 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 the 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 video 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 [17] mentioned above 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.

In contrast, the implementation of F_4 in `MAGMA-2.16`, often cited as the best Gröbner-basis solver available today, requires more than 64 GB of memory to solve just 25 cubic equations in as many $\mathbb{F}_2$-variables. When it does not run out of memory, it requires 2.5 hours to solve 20 cubic equations in 20 variables on one Opteron core at 2.2 GHz, or half an hour for 45 quadratic equations with 30 variables, or 7 minutes for 60 quadratic equations with 30 variables. Each of the above are solved in less than a second using negligible memory via enumeration on the same CPU.

Implications. The new exhaustive search algorithm can be used as a black-box in cryptanalysis that need to solve quadratic equations. This include for instance several algorithms for the Isomorphism of Polynomials problem [6, 20], and the attacks that rely on such algorithms such as [7].

We also show with a concrete example that (relatively simple) computations requiring 2^{64} operations can be more and more easily be carried out in practice with readily available hardware and a modest budget. Lastly, we highlight the fact that GPUs can be 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. Some known or useful results on Gray Codes and Derivative of multivariate polynomials are shown in section 2, where a formal framework of exhaustive search algorithms is also given. 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 all the algorithms presented 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. $\text{INIT}(f, x_0, k_0)$ 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 $x_0 \oplus (\text{GRAYCODE}(i) \ll k_0)$, for increasing values of i. $\text{NEXT}(State)$ advance to the next value and updates $State$. Three values can be directly read from the state: $State.\mathbf{x}$, $State.\mathbf{y}$ and $State.i$. The invariants shown in fig. 2(a) explicitly define the relationships which link them at all times. Finding all the zeroes of f is then achieved with the loop shown in figure 2(b).

Fig. 1. A Framework for Enumeration Algorithms.

i) $State.\mathbf{y} = f(State.\mathbf{x})$
ii) $State.\mathbf{x} = x_0 \oplus (\text{GRAYCODE}(State.i) \ll k_0)$.
iii) $\text{NEXT}(State).i = State.i + 1$.

(a) Invariants of an enumeration algorithm

```
procedure ZEROES(f)
  State ← INIT(f, 0, 0)
  for i from 0 to 2^n − 1
    if State.y = 0 then State.x is a zero of f
    NEXT(State)
  end for
end procedure
```

(b) Main loop.

3 Known Techniques for Quadratic Polynomials

We briefly discuss the enumeration techniques known to the authors.

Naive 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 section 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 (*cf.* lemma 1). This leads to the algorithm shown in fig. 2. We believe this technique to be folklore, and in any case it appears more or less explicitly in the existing litterature. 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.

Fig. 2. The Folklore Differential Enumeration

function INIT(f, _, _)
  $i \leftarrow 0$
  $\mathbf{x} \leftarrow 0$
  $\mathbf{y} \leftarrow f(0)$
  For all $0 \leq k \leq n-1$, initialize c_k and D_k
end function

(a) Initialisation

function NEXT($State$)
  $i \leftarrow i+1$
  $k = b_1(i)$
  $\mathbf{z} \leftarrow$ VECTORVECTORPRODUCT $(D_k, \mathbf{x})$
  $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}$
  $\mathbf{x} \leftarrow \mathbf{x} \oplus e_k$
end function

(b) Update

4 A Faster Recursive Algorithm for any Degree

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

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 $\mathcal{O}\left(n^{2d}\right)$.*

Construction of the Recursive Enumeration Algorithm. We will construct an enumeration algorithm in two stages. First, if f is of degree 0, then the problem can be trivially solved, as there is almost nothing to do, except to ensure that our definition of an enumeration algorithm is fulfilled. This algorithm is shown in fig. 3.

Fig. 3. Enumeration in the constant case

function INIT(f, k_0, x_0)
  $i \leftarrow 0$
  $\mathbf{x} \leftarrow x_0$
  $\mathbf{y} \leftarrow f(x_0)$
end function

(a) Initialisation

function NEXT($State$)
  $i \leftarrow i + 1$
  $k = b_1(i)$
  $\mathbf{x} \leftarrow \mathbf{x} \oplus e_k$
end function

(b) Update

When f is of higher degree, we need a little more effort. The main idea is that in the folklore differential algorithm of fig. 2, 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 fig. 5(a).

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 of fig. 2(b) 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 2. The correctness of this algorithm is proved in annex A.

Removing Useless Computation. In fact, NEXT performs lots of useless work, such as maintaining $\mathbf{x}$. Less obviously, we can also avoid maintaining i.

Lemma 2. *After line 3 of* NEXT *in fig. 5(a), we have: $i = 2^k + Derivative[k].i \times 2^{k+1}$.*

Proof. It is not difficult to see that the ℓ-th value of j such that $b_1(j) = k$ is $2^k + \ell \times 2^{k+1}$. The statement of the lemma is equivalent to saying that $Derivative[k].i$ counts the number of time where $b_1(i) = k$ happened since the begining of the main loop (not counting the current value of i). This simply follows from the fact that $Derivative[k].i$ counts the number of times NEXT($Derivative[k]$) has been called. □

Next, we argue that:

$$b_2\left(2^k + j \cdot 2^{k+1}\right) = \begin{cases} -1 & \text{if } j = 0 \\ (k+1) + b_1(j) & \text{otherwise} \end{cases} \tag{4}$$

Thus, after line 3 of NEXT, we have that $b_1(Derivative[k].i) = b_2(i) + (k+1)$. In fact, $b_2(i)$ then gives precisely the index of the bit of $Derivative[k].\mathbf{x}$ that gets flipped. The same reasoning carries over to $Derivative[k].Derivative[k'].i$, (using $b_3(i)$) and so on.

Thus, it is possible to avoid storing the i values, except in the main loop, by evaluating b_ℓ on the index of the main loop. These computations, although taking amortized constant time, could be made negligible by unrolling. NEXT then essentially perform d bit operations, and since it is in fact only necessary to store $\mathbf{y}$. INIT creates an array of n^d bits of constants (the degree-d derivatives), and allocates n^{d-1} bits of internal state (corresponding the derivatives of smaller degree).

An Iterative Version. While the combination of algorithms 2(b), and 5(a) gives a correct and complete algorithm, its recursive formulation is not the easiest way of obtaining an efficient implementation. Therefore, we explicitly unrolled recursive calls, and packed all the sub-algorithms into a simpler one, algorithm 5(b). We also removed all the useless computations (for instance, the i and the $\mathbf{x}$ fields of each State in fact do not need to be maintained). The c_i and D_i notations are those of section 2. The critical section of this code is the inner loop that starts at line 10. It performs two XORs and one comparison. The cost of computing the 2-adic valuation can be made negligible by partially unrolling this critical loop.

Fig. 4. Faster Enumeration.

1: **function** INIT(f, k_0, x_0)
2: $\quad i \leftarrow 0$
3: $\quad \mathbf{x} \leftarrow x_0$
4: $\quad \mathbf{y} \leftarrow f(x_0)$
5: $\quad$ **for** i **from** 0 **to** $2^n - 1$
6: $\quad\quad x'_0 \leftarrow x_0 \oplus \text{GRAYCODE}\left(2^{k+k_0}\right)$
7: $\quad\quad Derivative[k] \leftarrow \text{INIT}\left(\dfrac{\partial f}{\partial k + k_0}, k + k_0 + 1, x'_0\right)$
8: $\quad$ **end for**
9: **end function**

1: **function** NEXT($State$)
2: $\quad i \leftarrow i + 1$
3: $\quad k = b_1(i)$
4: $\quad \mathbf{x} \leftarrow \mathbf{x} \oplus e_{k+k_0}$
5: $\quad \mathbf{y} \leftarrow \mathbf{y} \oplus Derivative[k].\mathbf{y}$
6: $\quad$ NEXT($Derivative[k]$)
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: $\quad$ **if** $\mathbf{y} = 0$ **then** GRAYCODE($2^u - 1$) is a zero of f
7: $\quad \mathbf{z}[u] \leftarrow D_u[u-1] \oplus c_u$
8: $\quad \mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}[u]$
9: $\quad$ **for** v **from** 0 **to** $2^u - 2$
10: $\quad\quad$ **if** $\mathbf{y} = 0$ **then** GRAYCODE($2^u + v$) is a zero of f
11: $\quad\quad k \leftarrow b_1(2^u + v + 1)$
12: $\quad\quad \ell \leftarrow b_2(2^u + v + 1)$
13: $\quad\quad \mathbf{z}[k] \leftarrow \mathbf{z}[k] \oplus D_k[\ell]$
14: $\quad\quad \mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}[k]$
15: $\quad$ **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 5(b) could be run in parallel on $f_1, \dots, f_m$. All the parallel runs would execute the same instruction on different data, which makes it easy 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 polynomials in parallel using algorithm 5(b) 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.*

The remaining of this section is devoted to establish this theorem. Let us introduce a useful notation. Given an ordered set U, we denote the common zeroes of $f_1, \dots, f_m$ belonging to U by $Z([f_1, \dots, f_m], U)$. 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.

Early Aborting the Evaluation. A possible strategy is to compute the Z_i recursively: first Z_1, then Z_2, etc. However, while algorithm 5(b) can be used to compute Z_1, it cannot be used to compute Z_2 from Z_1, because it intrinsically enumerate all $(\mathbb{F}_2)^n$. In practice, the best results are in fact obtained by computing Z_k, for some well-chosen value of k, using k parallel runs of algorithm 5(b), and then computing $Z_{k+1}, \dots, Z_m$ one-by-one. Computing Z_k requires $2k2^n$ bit op. It then remains to compute Z_m from Z_k, and to find the best possible value of k. Note that if $m > n$, then we can focus on the first n equations, as they should have a constant number of solutions, which can in turn be checked against the remaining equations efficiently. If $m < n$, then we can specialize $m - n$ variables, and solve the m equations in m variables for any possible values of the specialized variables. All-in-all, the interesting case is when $m = n$. Also note that often k should be chosen to fit the hardware platform (*e.g.*, $k = 32$ if we can only address registers in 32-bit or longer chunks).

Early-abort + Naive Evaluation. We compute Z_{i+1} from Z_i using the early-abort strategy with naive evaluation, for $k \le i \le n-1$. It is clear that the expected cardinality of Z_i is 2^{n-i}. Computing Z_{i+1} then takes $n(n+3)2^{n-i-1}$ bit ops. The expected cost of computing Z is then approximately $n(n+3)2^{n-k}$ bit operations. Minimizing the global cost means solving the equation $2k \cdot 2^n = n(n+3) \cdot 2^{n-k}$. Expressing the solution in terms of the Laurent W function, and using known asymptotic results [12] when $n \to \infty$ gives:

$$k = 2\log_2 n - \log_2 \log_2 n + o(\log \log n)$$

and the complexity of the whole procedure is then about $8 \log_2 n \cdot 2^n$. In general, for degree-d systems, the same reasoning would get $4d \cdot \log_2 n \cdot 2^n$.

Early-Abort + Differential Folklore. We can efficiently evaluate Z_{i+1} from Z_i using an easy consequence of equation (1): given $f(\mathbf{x})$, computing $f(\mathbf{x} + \Delta)$ takes $2|\Delta| \cdot n$ bit operations, where $|\Delta|$ denote the hamming weight of Δ. Let us write $Z_i = \left\{\mathbf{x}_1^i, \dots \mathbf{x}_{q_i}^i\right\}$ (the elements are ordered using the usual lexicographic order), and $\Delta_j^i = \mathbf{x}_{j+1}^i \oplus \mathbf{x}_j^i$.

Computing Z_{i+1} therefore requires approximately $2n \cdot \sum_{j=1}^{q_i-1} |\Delta_j^i|$ bit operations. This quantity is upper-bounded by $2n \cdot \sum_{j=1}^{q_i-1} \left\lceil \log_2 \Delta_j^i \right\rceil$. Now, Δ_j^i follows a geometric distribution of parameter 2^{-i}, and thus has expectation 2^i. Computing Z_{i+1} therefore requires in average $2n \cdot i \cdot 2^{n-i}$ bit op. Finally, computing Z from Z_k requires on average $2n \cdot 2^n \cdot \sum_{i=k}^{n-1} i \cdot 2^{-i} \leq 4n \cdot (k+1) \cdot 2^{n-k}$ bit operations, hence an approximately optimal value of k would then satisfy $2k \cdot 2^n = 4(k+1) \cdot n \cdot 2^{n-k}$ which is approximately $k = 1 + \log_2 n$. The complexity of the whole procedure is then $\log_2 n \cdot 2^{n+2}$. However, implementing this technique efficiently looks like a lot of work for at best a $2\times$ gain.

Practical Considerations. Choosing the "optimal" value of k is not only of theoretical interest, but may have a practical significance if a very *ad hoc* circuit were to be designed from scratch. Even in the software implementations we are concerned with, it provides a guideline. However, when implemented in software on processors with registers, the logical operation width of the hardware becomes a determinant argument in the actual choice of k. If operations are always performed on ω-bit registers, then it is likely that the most meaningful choice of k is precisely ω. In all our implementations, we used the "Early-abort + Naive Evaluation" strategy with $k = 32$. This enables us to make use of the full register width, while keeping the "naive evaluation" time negligible. However, this means that the enumeration process must store 32 times more data in fast memory, compared to the evaluation of only one polynomial.

6 Brief Description of Platforms

6.1 Vector Units on x86-64

The most prevalent SIMD (single instruction, multiple data) instruction set today is SSE2, available in all current Wintel-compatible CPUs today. SSE2 instructions operate on 16 architectural `xmm` registers, each of which is 128-bit wide. There are floating point operations that we don't use, and integer operations treating `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 Ops (`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 reader needs to refer to Intel and AMD's manuals for the operation of the instructions, and to references such as [16] for their 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. In CUDA (Compute Unified Device Architecture), 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 int or float instruction (including add/subtract, multiply, bitwise and/or/xor, and fused multiply-add), and that of an SFU 2 float multiplications or 1 special operation. The arithmetic units have 20+-stage pipelines.

Main memory is slow and a major bottleneck. The read bandwidth to main (device) memory on the card from the GPU is only one 32-bit read per cycle per MP and has a latency of > 200 cycles. To ameliorate this problem, the MP is equipped with a register file of 64kB (16,384 registers, max. of 128 a thread), a 16-bank shared memory of 16kB, and a 8kB cache for read-only access to a declared "constant region" of at most 64kB. Every cycle, each MP can achieve one 32-bit read from each shared memory bank, and 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-size", 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 threads 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 will describe here the parts of our code, the approximate cost structure, our design choices and justifications. Our implementation code always consists of three stages:

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

Enumeration Kernel: Run the algorithm of Fig. 5(a) 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 are seen in Fig. 5.

Fig. 5. Unrolled Inner Loop Snippets to Brute-Force Degree 2/3 $\mathbb{F}_2$-Systems

(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 in Fig. 5 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 any indexing calculations and all 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 branching and memory accesses are inhibitively expensive.

7.2 GPU Enumeration Kernel

register usage Fast memory is precious on GPU and register usage critical for CUDA programmers. Our algorithms' memory use 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. *E.g.*, the code in Tab. 6(b): According to our experimental results, the repetitive 4-line code segments average less than three SP (stream-processor) cycles. However, after applying `decuda` to our program, we found that each such code segment corresponds to at least 4 instructions including 2 XORs and 2 conditional moves [as marked in Fig. 6(a)]. One possible 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.

Note that the annotated segment in Tab. 6(b) correspond to instructions far apart because *an nVidia GPU does opportunistic dispatching but is nevertheless a purely in-order architecture,* so properly scheduling must interleave instructions from different parts of the code.

Fig. 6. CUDA and Cubin code fragments of Degree-2 GPU Implementation

```
...
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

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 in which have all zero constant terms is automatically satisfied by a 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 when he examines the data.

Candidate Checking 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 might become more than an order of magnitude faster. We do not (yet) do so presently, because the check code is but 6-10% of the entire runtime, and the compilation process may take more time than the checking code. However, we should 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 we can compute the time complexity given the architecture, the frequency, and n. However, an Intel core can dispatch *three* XMM (SSE2) logical instructions to AMD's *two* and handle branch prediction and caching better, leading to a marked performance difference.

The Cell The platform received a lot of attention recently. In particular, the Sony Playstation 3 running Linux, is said to be very cost-effective for parallel processing in various kinds of cryptanalytic work. We will briefly discuss how well can a PS3 do in theory. The model that received much press exposure has available to the user 6 synergetic processing elements (SPEs), each of which can do one 128-bit wide logical operation per 3.2GHz cycle in its main pipeline, with a secondary pipeline to handle address calculation, loads and the like.

Since the Cell is fairly memory-poor, we expect to use the Cell like a GPU, and project that it will take also seven 128-bit operations in its inner loop for quadratics, including the two XORs, one compare for equality in each limb, and four more to test and extract the potential solution. Given that a Cell would then average about 7/6 cycles per iteration, and a K10+ takes about 4.5 cycles per iteration per core, we may estimate a Cell at peak speed to perform very close to a quad-core K10+3.2GHz (PhenomII$\times$4 965) in exhaustive searching quadratic systems. While unfortunately we do not have a Cell system to test, the above discussion should also show that a PS3 should not be able to match the hundreds of cores on an nVidia G200 series GPU.

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. 7. As we can see, all except two architectures have inner loop cycle counts that are increasing roughly linearly with the degree. These are the AMD K10 and nVidia G200 architectures, which is in line with the lack of caches on the nVidia GPUs and fact that K10 has the least cache among the CPU architectures.

References

1. G. V. Bard, N. T. Courtois, and C. Jefferson. 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`.

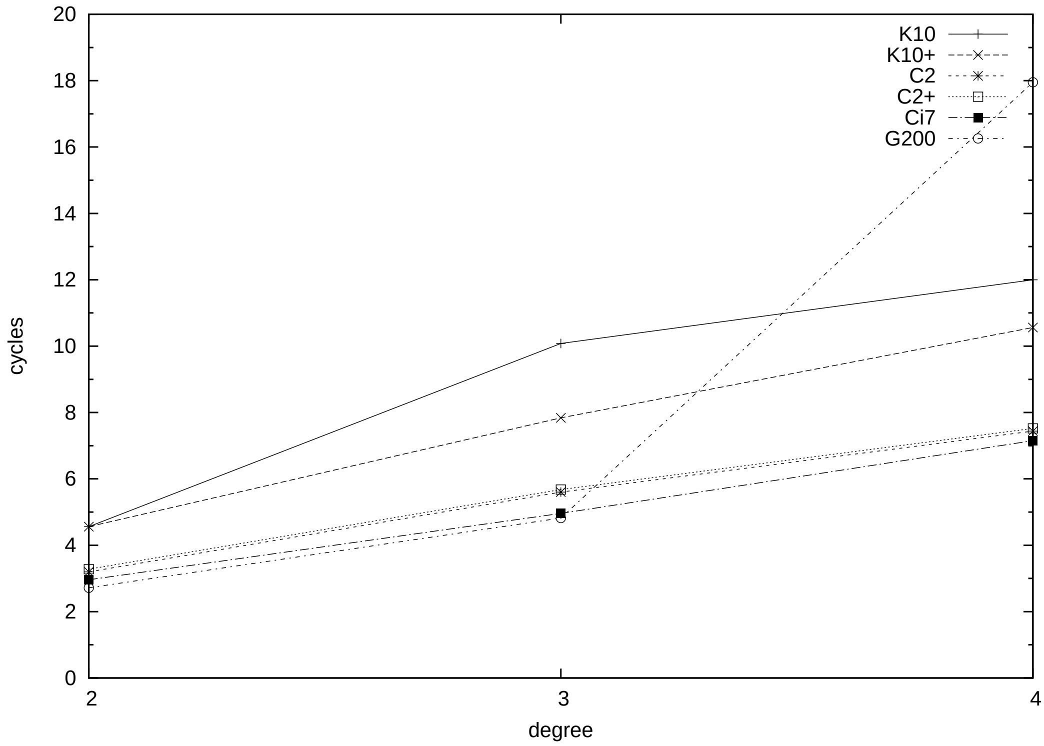

Fig. 7. Cycles per candidate tested for degree 2,3 and 4 polynomials.

Table 1. 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

Table 2. Performance results for $n = 48$

minutes			testing platform				#cores	# threads
$d = 2$	$d = 3$	$d = 4$	GHz	arch	name	USD	available	launched
1217	2686	3191	2.2	K10	Phenom 9550	120	4	1
1157	1992	2685	2.3	K10+	Opteron2376	184	4	1
142	240	336	2.3	K10+	Opteron2376×2	368	8	8
780	1364	1819	2.4	C2	Xeon X3220	210	4	1
671	1176	1560	2.83	C2+	Core2 Q9550	225	4	1
761	1279	1856	2.26	Ci7	Xeon E5520	385	4	1
139	213	327	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	n/a

Table 3. Budget to solve $n = 64$ in one month.

USD	increase	Platform
113'316	8.08	Opteron2376
78'720	5.61	Xeon E5520
60'270	4.30	Xeon X3220
55'575	3.96	Core2 Q9550
53'640	3.82	Phenom 9550
15'500	1.10	GTX 295
14'030	1.00	GTX 280

2. M. Bardet, J.-C. Faugère, and B. Salvy. 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. M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic expansion of the degree of regularity for semi-regular systems of equations. *Proc. MEGA 2005*, 2005.
4. C. Berbain, H. Gilbert, and J. Patarin. QUAD: A practical stream cipher with provable security. Eurocrypt 2006, LNCS 4004, pp. 109–128.
5. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. Eurocrypt 2009, LNCS 5479, pp. 483–501.
6. C. Bouillaguet, J.-C. Faugère, P.-A. Fouque, and L. Perret. Differential-algebraic algorithms for the isomorphism of polynomials problem. `http://eprint.iacr.org/2009/583`
7. C. Bouillaguet, P.-A. Fouque, A. Joux, and J. Treger. A family of weak keys in HFE (and the corresponding practical key-recovery). `http://eprint.iacr.org/2009/619`.
8. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Innsbruck, 1965.
9. N. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on Keeloq. FSE 2008, LNCS 5086, pp. 97–115.
10. N. Courtois, L. Goubin, and J. Patarin. *SFLASH: Primitive specification (second revised version)*, 2002. `https://www.cosic.esat.kuleuven.be/nessie`
11. N. T. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. Eurocrypt 2000, LNCS 1807, pp. 392–407. Extended ver.: `http://www.minrank.org/xlfull.pdf`.
12. N. de Bruijn. *Asymptotic methods in analysis. 2nd edition.* Bibliotheca Mathematica. Vol. 4. Groningen: P. Noordhoff Ltd. XII, 200 p. , 1961.
13. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *J. of Pure and Applied Algebra*, **139**(1999), pp. 61–88.
14. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). ACM ISSAC 2002, pp. 75–83.
15. J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. CRYPTO 2003, LNCS 2729, pp. 44–60.
16. A. Fog. *Instruction Tables*. Copenhagen University, College of Engineering, Feb 2010. Lists of Instruction Latencies, Throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs, `http://www.agner.org/optimize/instruction_tables.pdf`.
17. J. Patarin. Asymmetric cryptography with a hidden monomial. Crypto 1996, LNCS 1109, pp. 45–60.
18. J. Patarin. Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of asymmetric algorithms. Eurocrypt 1996, LNCS 1070, pp. 33–48. Extended ver.: `http://www.minrank.org/hfe.pdf`.
19. J. Patarin, N. Courtois, and L. Goubin. QUARTZ, 128-bit long digital signatures http://www.minrank.org/quartz/. CT-RSA 2001, LNCS 2020, pp. 282–297.

20. J. Patarin, L. Goubin, and N. Courtois. Improved algorithms for Isomorphisms of Polynomials. Eurocrypt 1998, LNCS 1403, pp. 184–200. Extended ver.: `http://www.minrank.org/ip6long.ps`.
21. H. Raddum. MRHS equation systems. SAC 2007, LNCS 4876, pp. 232–245.
22. M. Sugita, M. Kawazoe, L. Perret, and H. Imai. Algebraic cryptanalysis of 58-round SHA-1. FSE 2007, LNCS 4593, pp. 349–365.
23. B.-Y. Yang and J.-M. Chen. Theoretical analysis of XL over small fields. ACISP 2004, LNCS 3108, pp. 277–288.

A Correctness Proof of the Algorithm Presented in Section 4

At first glance, it may not seem trivial that the combination of algorithms 2(b) and 5(a) results in a method for finding all the zeroes of f. In this section, we justify why it is indeed the case. We first state a lemma about Gray Codes.

Lemma 3. *For $j \in \mathbb{N}$:*

$$\textsc{GrayCode}(2^k + j \cdot 2^{k+1}) = \begin{cases} \textsc{GrayCode}(2^k) \oplus (\textsc{GrayCode}(j) \ll (k+1)) & \text{if } j \text{ is even} \\ \textsc{GrayCode}(2^k) \oplus (\textsc{GrayCode}(j) \ll (k+1)) \oplus e_k & \text{if } j \text{ is odd} \end{cases}$$

Proof. It should be clear that $2^k + j \cdot 2^{k+1}$ and $2^k \oplus j \cdot 2^{k+1}$ in fact denote the same number. Also, GrayCode is a linear function on $(\mathbb{F}_2)^n$. Thus it remains to establish that $\textsc{GrayCode}(j \cdot 2^{k+1}) = \textsc{GrayCode}(j) \ll k+1$ (resp. $e_k \oplus (\textsc{GrayCode}(j) \ll k+1)$) when j is even (resp. odd). Again, $j \cdot 2^{k+1} = j \ll (k+1)$, and by definition we have:

$$\textsc{GrayCode}(j \cdot 2^{k+1}) = \textsc{GrayCode}(j \ll (k+1)) = (j \ll (k+1)) \oplus ((j \ll (k+1)) \gg 1)$$

Now, we have :

$$(j \ll k+1) \gg 1 = \begin{cases} (j \gg 1) \ll k+1 & \text{when } j \text{ is even} \\ ((j \gg 1) \ll k+1) \oplus e_k & \text{when } j \text{ is odd} \end{cases}$$

and the result follows. □

We prove by induction on the degree of f that the Init and Next functions described in fig. 5(a) maintain and preserve the three invariants of enumeration algorithms, described in section 2. The base case is when f is a constant polynomial (*i.e.*, of degree zero). we hope that the reader will be convinced that the "constant enumeration" algorithm of fig. 3 works correctly .

In higher degree, it is not difficult to check that the three invariants are enforced at the end of Init. Let us now assume that f has degree $d \geq 1$. Let us assume that we are in the middle of the main loop, and that the invariants defining our enumeration algorithm hold at the begining of Next(). Our objective is to show that they still hold at the end, and that the state has been updated correctly. Let us then focus on the Next() part of algorithm 5(a). Invariant iii is easily seen to be enforced by line 2, while invariant ii follows from line 4, and from lemma 1. The non-trivial part is to show that invariant i holds. The two following lemma are devoted to this task.

Lemma 4. *After line 3, we have:* $Derivative[k].\mathbf{x} = \mathbf{x}$ *or* $Derivative[k].\mathbf{x} = \mathbf{x} + e_{k+k_0}$.

Proof. By assuming that invariant ii holds for the current state at the entry of NEXT, we have $\mathbf{x} = x_0 \oplus (\text{GRAYCODE}(i) \ll k_0)$. Because after line 3 k is set to $b_1(i+1)$, it follows from lemma 1 that:

$$\mathbf{x} = x_0 \oplus e_{k+k_0} \oplus (\text{GRAYCODE}(i+1) \ll k_0)$$

Then, because of lemma 2, we obtain:

$$\mathbf{x} = x_0 \oplus e_{k+k_0} \oplus \left(\text{GRAYCODE}\left(2^k + Derivative[k].i \times 2^{k+1}\right) \ll k_0\right)$$

Additionally, by induction hypothesis on $Derivative[k]$, invariant ii grants:

$$Derivative[k].\mathbf{x} = x_0 \oplus \text{GRAYCODE}\left(2^{k+k_0}\right) \oplus \Big(\text{GRAYCODE}\left(Derivative[k].i\right) \ll k+k_0+1\Big)$$

Then because of lemma 3 applied to $\mathbf{x}$, we have:

$$\mathbf{x} = \begin{cases} Derivative[k].\mathbf{x} \oplus \Big(\text{GRAYCODE}\left(2^k\right) \ll k_0\Big) \oplus \text{GRAYCODE}\left(2^{k+k_0}\right) \oplus e_{k+k_0} & \text{if } Deri \\ Derivative[k].\mathbf{x} \oplus \Big(\text{GRAYCODE}\left(2^k\right) \ll k_0\Big) \oplus \text{GRAYCODE}\left(2^{k+k_0}\right) & \text{if } Deri \end{cases}$$

Now, if $k > 0$ then $\text{GRAYCODE}\left(2^k\right) \ll k_0 = \text{GRAYCODE}\left(2^{k+k_0}\right)$, and the lemma is established. In the case where $k = 0$, we have in fact:

$$\mathbf{x} = \begin{cases} Derivative[0].\mathbf{x} \oplus \text{GRAYCODE}\left(2^{k_0}\right) & \text{if } Derivative[0].i \text{ is even} \\ Derivative[0].\mathbf{x} \oplus \text{GRAYCODE}\left(2^{k_0}\right) \oplus e_{k_0} & \text{if } Derivative[0].i \text{ is odd} \end{cases}$$

and the lemma is established again. □

Lemma 5. *Let* $\mathbf{x}'$ *and* $\mathbf{y}'$ *denote the values of* $\mathbf{x}$ *and* $\mathbf{y}$ *after line 4. Then we have* $\mathbf{y}' = f(\mathbf{x}')$.

Proof. By induction hypothesis on $Derivative[k]$, invariant i and the previous lemma grant us that $Derivative[k].\mathbf{y} = \frac{\partial f}{\partial k+k_0}(\mathbf{x})$ or $Derivative[k].\mathbf{y} = \frac{\partial f}{\partial k+k_0}(\mathbf{x} + e_{k+k_0})$. However, since for all i, $\frac{\partial f}{\partial i}(x + e_i) = \frac{\partial f}{\partial i}(x)$, then we have in all cases:

$$Derivative[k].\mathbf{y} = \frac{\partial f}{\partial k + k_0}(\mathbf{x})$$

So, this yields (using lemma 1):

$$\mathbf{y}' = f(\mathbf{x}) + \frac{\partial f}{\partial k + k_0}(\mathbf{x}) = f(\mathbf{x} + e_{k+k_0}) = f(\mathbf{x}')$$

□