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

Abstract. We analyze how fast we can solve general systems of multivariate equations of various low degrees over $\mathbb{F}_2$; this constitutes a generic attack on a hard problem both important in itself and often arising as part of algebraic cryptanalysis. We do not propose new generic techniques, but instead we revisit the standard exhaustive-search approach. We show that it can be made more efficient both asymptotically and practically. We implemented 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 using just an NVIDIA GTX 295 video card in 27 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 shows concretely that some combinatorial problems and cryptanalysis in general could benefit from implementations on GPUs.
Keywords: multivariate polynomials, system-solving, parallelization, Graphic Processing Units (GPUs)

1 Introduction

Solving systems of algebraic equations is a natural mathematical problem that has been given much attention by various research groups including the cryptographic community. The interest of latter in this problem has two sources. On the one hand, since the problem is NP-complete over finite fields (and random instances seem hard), it could be used to design cryptographic primitives. This has lead to the development of multivariate cryptography in the last few decades, with one-way trapdoor functions such as HFE [18], SFLASH [19], and QUARTZ [20], 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 equations. This was attempted without success to break the AES, but succeeded against other block ciphers such as KeeLoq [9], and gave a faster collision attack on 58 rounds of SHA-1 [22].

Since the work of Buchberger [8], Gröbner-basis techniques have been the most prominent tool to tackle this problem, notably after the emergence of faster algorithms such as F_4 [12] or F_5 [13]. Gröbner basis attacks culminated with the break of the first HFE challenge [14].

The cryptographic community independently rediscovered some of the ideas underlying efficient Gröbner-basis algorithms, under the form of the XL algorithm [10] and its variants. Cryptanalysts also introduced their own techniques, such as the conversion to SAT instances [23, 1], which turned out to be efficient if the systems are sparse enough, e.g., having few different variables per term or few terms per equation.

In this paper we take a different path, namely improving the well-understood exhaustive search algorithm. All the known methods have exponential complexity on random systems of n quadratic equations in n unknowns. Gröbner-basis methods are at an advantage on very overdetermined systems (with many more equations than unknowns) and systems with weaknesses, but are shown to be exponential on "generic" enough systems [2, 3]. Experts believe that Gröbner-basis methods will not outperform exhaustive search until $n \geq 220$, even if we assume enough memory and that they work with sparse matrix solvers.

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 underlied the security of a signature scheme [17].

Now, of course, most people would instinctively consider an algebraic attack that reduces a cryptosystem to 64 equations of degree 4 in as many $\mathbb{F}_2$-variables to be successful, and the system to be practically broken. The matter is however not that easily settled because the complexity of a naïve exhaustive-search algorithm would be $2 \cdot \binom{64}{4} \cdot 2^{64} \approx 2^{84}$ logical operations, which would make the attack hardly feasible on current hardware. As we shall show in this paper, this complexity can be improved and be brought to the feasible region. It is also a possible scenario that the break

of a system might come from a large but fixed number of different polynomial systems, in which case solving them as quickly as possible is important.

Our Contribution. Our contribution is twofold. On the theoretical side, we present an exhaustive search algorithm which is both asymptotically and practically faster than existing techniques. Finding all the zeroes of a single degree-d polynomial in n variables requires $d \cdot 2^n$ bit operations. We extend it to find the common zeroes of m quadratic polynomials in $\log_2 n \cdot 2^{n+2}$ bit operations.

On the practical side, we implemented our 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 one of the fastest CPUs currently available, an Intel quad-code Core i7 at 3 GHz. Today, we can solve 48+ quadratic equations in 48 binary variables using just an nVidia GTX 295 video card in 27 minutes. This device is available for about $\$500$. It would be 45 minutes for cubic equations and three hours for quartics. The 64-bit challenge [17] mentioned earlier can thus be broken with 10 such GPUs in 4 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-2.15-5`, often cited as the best Gröbner-basis solver available today, would run out of memory on a 64 GB computational server with 25 $\mathbb{F}_2$-variables in as many cubic equations. Some systems that can be solved on this server (which is equipped with 16 2.2 GHz AMD K8 cores) by `MAGMA` in $\mathbb{F}_2$ before running out of memory are: about 2.5 hours to solve 20 cubic equations in 20 variables, 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 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, 21], 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 describe the two platform on which we implemented the algorithm, and section 7 describes the implementation and performance evaluation results.

2 Generalities

Gray Code. Gray Codes play a crucial role in all the algorithms presented in this paper. In this section we summarize all the results we need. Let us denote by $\nu_2(i)$ the 2-adic valuation of the integer i (*i.e.*, the index of the lowest-significant bit set to 1). To make things clear, $\nu_2(0)$ is undefined, $\nu_2(1) = 0$ and $\nu_2(2) = 1$.

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

Lemma 1. *For* $i \in \mathbb{N}$*:* $\textsc{GrayCode}(i+1) = \textsc{GrayCode}(i) \oplus e_{\nu_2(i+1)}$.

Lemma 2. *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. It should be clear that GRAYCODE is a linear function on V. 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. □

Derivatives. Define the $\mathbb{F}_2$ *derivative* $\frac{\partial f}{\partial i}$ of a polynomial over the 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 informations the enumeration algorithm need 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. $\textsc{Next}(State)$ advance to the next value and update $State$. Three values can be directly read from the state: $State.\mathbf{x}$, $State.\mathbf{y}$ and $State.i$. The invariants below explicit which relation should link them at all times:

i) $State.\mathbf{y} = f(State.\mathbf{x})$
ii) $State.\mathbf{x} = x_0 \oplus (\textsc{GrayCode}(State.i) \ll k_0)$.
iii) $\textsc{Next}(State).i = State.i + 1$.

With such an algorithm, finding all the zeroes of f is achieved with the loop shown in algorithm 1. 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.

3 Known Techniques for Quadratic Polynomials

We briefly discuss the enumeration techniques known to the authors.

Fig. 1. Enumeration, main loop.

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

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 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 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 an almost $w\times$ speed up, where w is the max. 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 happen 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 V such that two consecutive elements differ in only one bit. 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 $2(n+1)$, which is about $n/4$ times less than the naive method.

Fig. 2. The Folklore Differential Enumeration

```
1: function INIT(f, _, _)
2:     i ← 0
3:     x ← 0
4:     y ← f(0)
5:     For all 0 ≤ k ≤ n − 1, initialize c_k and D_k
6: end function
```

(a) Initialisation

```
1: function NEXT(State)
2:     i ← i + 1
3:     k = ν_2(i)
4:     z ← VECTORVECTORPRODUCT (D_k, x) ⊕ c_k
5:     y ← y ⊕ z
6:     x ← x ⊕ e_k
7: end function
```

(b) Update

4 A Faster Recursive Algorithm for any Degree

We build an asymptotically and practically faster enumeration algorithm, summarized in Theorem 1.

Theorem 1. *The zeroes of a 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 bits of memory, after an initialization phase of (presumably 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 fairly easy resolved, as there is almost nothing to do, except ensuring that our definition of an enumeration algorithm is fulfilled. This algorithm is shown in fig. 3.

Fig. 3. Enumeration in the constant case

```
1: function INIT(f, k_0, x_0)
2:     i ← 0
3:     x ← x_0
4:     y ← f(x_0)
5: end function
```

(a) Initialisation

```
1: function NEXT(State)
2:     i ← i + 1
3:     k = ν_2(i)
4:     x ← x ⊕ e_k
5: end function
```

(b) Update

When f is of higher degree, we need a little more effort, but we may use the enumeration algorithm recursively on polynomials of strictly smaller degree. The 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. 1 is $\mathcal{O}\left(d \cdot 2^n + n^{2d}\right)$. When $d = 2$, this is about n times faster than algorithm 2. In fact, NEXT performs lots of useless work, such as maintaining i and $\mathbf{x}$. This could be removed without altering the results. Also, computing the 2-adic valuation, although taking amortized constant time, could be made negligible through unrolling. This is much more apparent on the iterative version given below. In its most optimized form, NEXT essentially perform d bit operations, and since it is in fact only necessary to store $\mathbf{y}$, INIT requires exactly n^d bits of memory. The correctness of this algorithm is proved in annex A.

Practical Instantiation for Quadratic Polynomials While the combination of algorithms 1, 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 the four 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.

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 $f_1, \ldots, f_m$. All the parallel runs would execute the same instruction on different data, which makes it easy to implement on vector or SIMD architectures. At 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.

Fig. 4. Faster Enumeration.

```
1: function INIT(f, k_0, x_0)
2:    i ← 0
3:    x ← x_0
4:    y ← f(x_0)
5:    for i from 0 to 2^n − 1
6:       x'_0 ← x_0 ⊕ GRAYCODE((2^{k+k_0})
7:       Derivative[k] ← INIT(∂f/∂e_{k+k_0}, k + k_0 + 1, x'_0)
8:    end for
9: end function

1: function NEXT(State)
2:    i ← i + 1
3:    k = ν_2(i)
4:    x ← x ⊕ e_{k+k_0}
5:    y ← y ⊕ Derivative[k].y
6:    Derivative[k] ← NEXT(Derivative[k])
7: end function
```

(a) General Setting

```
1:  y ← f(0)
2:  if y = 0 then 0 is a zero of f
3:  z[0] ← c_0
4:  y ← y ⊕ z[0]
5:  for u from 1 to n − 1
6:     if y = 0 thenGRAYCODE(2^u − 1) is a zero of f
7:     z[u] ← D_u[u − 1] ⊕ c_u
8:     y ← y ⊕ z[u]
9:     for v from 0 to 2^u − 2
10:       if y = 0 thenGRAYCODE(2^u + v) is a zero of f
11:       k ← ν_2(2^u + v + 1)
12:       ℓ ← ν'_2(2^u + v + 1)
13:       z[k] ← z[k] ⊕ D_k[ℓ]
14:       y ← y ⊕ z[k]
15:    end for
16: end for
```

(b) Iterative version for quadratic f

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 result. 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.

Early Aborting the Evaluation. A possible strategy is to compute the Z_i recursively: first Z_0, then Z_1, etc. However, while algorithm 5(b) can be used to compute Z_0, it cannot be used to compute Z_1 from Z_0, 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 $Z_{k+1}, \ldots, Z_m$ 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, it must be kept in mind that it is often more efficient to choose k in accordance with the hardware platform (for instance, $k = 32$ if 32-bit registers are available).

Early-abort + Naive Evaluation. We compute Z_{i+1} from Z_i using the early-abort strategy with naive evaluation, for $k \leq i \leq n-1$. 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 [11] 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 $Z_i = \left\{\mathbf{x}_1^i, \ldots \mathbf{x}_{q_i}^i\right\}$ (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.

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 [15] for their throughput and latencies.

6.2 GT2xx series of GPUs from nVidia

We choose to use nVidia's GPUs because they provide a programmer-friendly parallel programming environment called Compute Unified Device Architecture (CUDA), where we program GPUs using the familiar C/C++ programming language with 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 one super function units (SFU) 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 the SFU 4 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. Each MP can achieve one 32-bit read from each shared memory bank, and one read from the constant cache *which can broadcast to every thread*.

Each MP contains a scheduling and dispatching unit that can handle a large number of lightweight threads. However, the decoding unit can only decode one instruction every 4 cycles. 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 ~ 24 cycles, nVidia recommends a min. of 6 warps on each MP, although it is sometimes possible to get acceptable performance with 2-3 warps.

7 Implementations

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

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

Enumeration Kernel: Run the algorithm of figure 5(a) and find all candidate vectors $\mathbf{x}$ which satisfies w equations out of m ($\approx 2^{n-w}$ of them), which are handed over to ...

Candidate Checking: Candidate solutions $\mathbf{x}$ are checked against the other $m-w$ 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
```

(c) cubics, x86 assembly

```
.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
```

(d) cubics, C++ x86 instrinsics

```
      ...
      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);
      ...
```

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 the implementation for quartic systems as an example. Recall that each thread needs to maintain third derivatives, which we may call d_{ijk} for $0 \le i < j < k < K$, 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 the G200b series GPUs: Exceptional adeptness at handling conditional move instructions, generated reliably by `nvcc` with certain constructs, *e.g.*, the CUDA 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.

testing The inner loop for GPUs differs from that of the 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:

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

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 is different according to whether the enumeration is on the CPU or the GPU.

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 $\lesssim 800$ 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 consisting of three 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 (knowing that there are no other solutions within the same small system).

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, 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 in a kernel, and to store all the small systems would exhaust the device memory, 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

There are 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 $\binom{n-s}{d}$ 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, check time is then 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 $\propto \frac{1}{w}\binom{n}{d}$ instructions. But we can get down to $\propto \frac{1}{w}\binom{n-s}{d}$ 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 then average about 7/6 cycles per iteration, and a K10+ takes about 4.5 cycles per iteration per core. Unfortunately we do not have a Cell system to play with, but we estimate a Cell at peak speed would perform very close to a quad-core K10+3.2GHz (the PhenomIIX4 965) in exhaustive searching quadratic systems. Of course, it won't be much of a match for the hundreds of cores on an nVidia G200 series GPU.

Table 1. Performance results

cycles for $n = 32$			cycles for $n = 40$			cycles for $n = 48$			seconds for $n = 48$			platform			
$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	GHz	arch	name	USD
0.58	1.21	1.41	0.57	1.27	1.43	0.57	1.26	1.50	1217	2686	3191	2.2	K10	Phenom9550	120
0.56	0.91	1.31	0.58	0.97	1.30	0.56		1.30	1201		2776	2.2	K10+	Opteron2427	399
0.40	0.65	0.95	0.40	0.70	0.94	0.40	0.70	0.93	780	1364	1819	2.4	C2	Xeon X3220	210
0.40	0.66	0.96	0.41	0.71	0.94	0.41	0.71	0.94	671	1176	1560	2.83	C2+	Core2Quad Q9550	250
0.50	0.66	1.00	0.38	0.65	0.91	0.37	0.62	0.894	761	1279	1856	2.26	Ci7	Xeon E5520	384
2.87	4.66	15.01	2.69	4.62	17.94	2.72	4.82	17.95	41	73	271	1.296	G200	GTX280	
									0	1	3	2.4	C2	(GPU overhead)	

Table 2. percentage

	$n = 32$			$n = 40$			$n = 48$			platform			
	$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	$d = 2$	$d = 3$	$d = 4$	GHz	arch	name	USD
partial	0	0	0	0	0	0	0	0	0	2.83	C2+	Core2Quad Q9550	250
kernel	98.5	96.6	94.2	98.6	96.8	93.6	98.5	96.8	93.7				
check	1.5	3.4	5.7	1.4	3.2	6.4	1.5	3.2	6.3				
partial	13.9	31.9	31.1	0	2.5	0	0	0	0	1.296	G200	GTX280	
kernel	85.3	67.9	68.6	99.6	97.5	99.1	98.9	98.9	99.2				
check	0	0	0	0	0	0	1.1	0	0				

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. Cryptology ePrint Archive, Report 2007/024, 2007. `http://eprint.iacr.org/`.
2. M. Bardet, J.-C. Faugère, and B. Salvy. On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proc. International Conference on Polynomial System Solving (ICPSS)*, pages 71–75, 2004.

3. M. Bardet, J.-C. Faugère, B. Salvy, and B.-Y. Yang. Asymptotic Behaviour of the Degree of Regularity of Semi-Regular Polynomial Systems. In *MEGA'05*, 2005. Eighth International Symposium on Effective Methods in Algebraic Geometry, Porto Conte, Alghero, Sardinia (Italy), May 27th – June 1st.
4. C. Berbain, H. Gilbert, and J. Patarin. QUAD: A practical stream cipher with provable security. In S. Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2006.
5. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. Ecm on graphics cards. In A. Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2009.
6. C. Bouillaguet, J.-C. Faugère, P.-A. Fouque, and L. Perret. Differential-algebraic algorithms for the isomorphism of polynomials problem. Cryptology ePrint Archive, Report 2009/583, 2009. `http://eprint.iacr.org/`.
7. C. Bouillaguet, P.-A. Fouque, A. Joux, and J. Treger. A family of weak keys in hfe (and the corresponding practical key-recovery). Cryptology ePrint Archive, Report 2009/619, 2009. `http://eprint.iacr.org/`.
8. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, 1965.
9. N. Courtois, G. V. Bard, and D. Wagner. Algebraic and slide attacks on keeloq. In K. Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 97–115. Springer, 2008.
10. N. T. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Bart Preneel, ed., Springer, 2000. Extended Version: `http://www.minrank.org/xlfull.pdf`.
11. N. de Bruijn. *Asymptotic methods in analysis. 2nd edition.* Bibliotheca Mathematica. Vol. 4. Groningen: P. Noordhoff Ltd. XII, 200 p. , 1961.
12. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases (F_4). *Journal of Pure and Applied Algebra*, 139:61–88, June 1999.
13. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *International Symposium on Symbolic and Algebraic Computation — ISSAC 2002*, pages 75–83. ACM Press, July 2002.
14. J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equation (HFE) cryptosystems using Gröbner bases. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *LNCS*, pages 44–60. Springer, 2003.
15. 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`.
16. D. Naccache, editor. *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*. Springer, 2001.
17. J. Patarin. Asymmetric cryptography with a hidden monomial. In N. Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 1996.
18. J. Patarin. Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): two new families of asymmetric algorithms. In *Advances in Cryptology — EUROCRYPT 1996*, volume 1070 of *Lecture Notes in Computer Science*, pages 33–48. Ueli Maurer, ed., Springer, 1996. Extended Version: `http://www.minrank.org/hfe.pdf`.
19. J. Patarin, N. Courtois, and L. Goubin. Flash, a fast multivariate signature algorithm. In Naccache [16], pages 298–307.
20. J. Patarin, N. Courtois, and L. Goubin. QUARTZ, 128-Bit Long Digital Signatures. In Naccache [16], pages 282–297.
21. J. Patarin, L. Goubin, and N. Courtois. Improved Algorithms for Isomorphisms of Polynomials. In *EUROCRYPT*, pages 184–200, 1998.
22. M. Sugita, M. Kawazoe, L. Perret, and H. Imai. Algebraic cryptanalysis of 58-round sha-1. In A. Biryukov, editor, *FSE*, volume 4593 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.
23. H. vard Raddum and I. Semaev. New technique for solving sparse equation systems. Cryptology ePrint Archive, Report 2006/475, 2006. `http://eprint.iacr.org/`.

A Correctness Proof of the Algorithm Presented in Section 4

At first glance, it may not seem trivial that the combination of algorithms 1 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. The proof works by induction on the degree of f. If f is a constant polynomial, we hope that the reader will be convinced that the "constant enumeration" algorithm works correctly.

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 three following lemma are devoted to this task.

Lemma 3. *After line 3, we have:* $i + 1 = 2^k + Derivative[k].i \times 2^{k+1}$.

Proof. It is not difficult to see that the ℓ-th value of j such that $\nu_2(j) = k$ is $j = 2^k + \ell \times 2^{k+1}$. Our claim is equivalent to saying that $Derivative[k].i$ counts the number of time where $\nu_2 i + 1 = k$ happened (for a given k) previously since the begining of the main loop follows from the fact that $Derivative[k].i$ counts the number of times NEXT($Derivative[k]$) is called. This last fact is true by induction hypothesis: invariant ii holds for $Derivative[k]$, since the derivative has degree $d - 1$. □

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

$$Derivative[k].\mathbf{x} = x_0 \oplus \textsc{GrayCode}(2^{k+k_0}) \oplus (\textsc{GrayCode}(Derivative[k].i) \ll k + k_0 + 1)$$

Now, at this point we have $\mathbf{x} = x_0 \oplus (\textsc{GrayCode}(i) \ll k_0)$ (we already established this fact). It follows from lemma 1 that:

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

Then, because of our previous claim, we obtain:

$$\mathbf{x} = x_0 \oplus e_{k+k_0} \oplus \Big(\textsc{GrayCode}(2^k + Derivative[k].i \times 2^{k+1}) \ll k_0\Big)$$

Then because of lemma 2 applied to $\mathbf{x}$, we have:

$$\mathbf{x} = \begin{cases} Derivative[k].\mathbf{x} \oplus e_{k+k_0} & \text{if } Derivative[k].i \text{ is even} \\ Derivative[k].\mathbf{x} & \text{if } Derivative[k].i \text{ is odd} \end{cases}$$

□

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:

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

□