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

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

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

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

## 1 Introduction

Solving a system of $m$ nonlinear polynomial equations in $n$ variables over $\mathbb{F}_q$ is a natural mathematical problem that has been investigated by various research communities. The cryptographers are among the interested parties since an NP-complete problem whose random instances seem hard could be used to design cryptographic primitives, as witness the development of multivariate cryptography in the last few decades, using one-way trapdoor functions such as HFE, SFLASH, and QUARTZ [13, 22, 23], as well as stream ciphers such as QUAD [5].

Conversely, in "algebraic cryptanalysis" one distills from a cryptographic primitive a system of multivariate polynomial equations with the secret among the variables. This does not break AES as first advertised, but does break KeeLoq [12], for a recent example, and find a faster collision on 58-round SHA-1 [26].

Since the pioneering work by Buchberger [10], 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 $\mathbf{F_5}$ [16, 17], which broke the first HFE challenge [18]. The cryptographic community independently rediscovered some of the ideas underlying efficient Gröbner-basis algorithms as of the XL algorithm [14] and its variants. They also introduced techniques to deal with special cases, particularly that of sparse systems [1,25].

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

For "generic" quadratic systems, experts believe [2, 27] that Gröbner basis methods will go up to degree $D_0$, which is the minimum possible $D$ where the coefficient of $t^D$ in $(1+t)^n(1+t^2)^{-m}$ goes negative, and then require the solution of a system of linear equations with $T \gtrsim \binom{n}{D_0-1}$ variables. This will take at least $\mathrm{poly}(n) \cdot T^2$ bit-operations, assuming we can afford a sufficiently large amount of memory and that we can solve such a linear system of equations with non-negligible probability in $O(N^{2+o(1)})$ time for $N$ variables. For example, if we assume we can operate a Wiedemann solver on a $T \times T$ submatrix of the extended Macaulay matrix of the original system, then the polynomial is $3n(n-1)/2$. When $m = n = 200$, $D_0 = 25$, making the value of $T$ exceeds $2^{102}$; even taking into consideration guessing before solving [7, 28], we can still easily conclude that Gröbner-basis methods would not outperform exhaustive search in the practically interesting range of $m = n \leq 200$.

The questions we address are therefore: how far can we go, on both theoretical and practical sides, by pushing exhaustive search further? Is it possible to design more efficient exhaustive search algorithms? Can we get better performance using different hardware such as GPUs? Is it possible to solve *in practice*, with a modest budget, a system of 64 equations in 64 unknowns over $\mathbb{F}_2$? Less than 15 years ago, this was considered so difficult that it even underlied the security of a particular signature scheme [21]. Intuitively, some people may consider an algebraic attack that reduces a cryptosystem to 64 equations of degree 4 in 64 $\mathbb{F}_2$-variables to be a successful practical attack. However, the matter is not that easily settled because the complexity of a naïve exhaustive search algorithm would actually be *much higher* than $2^{64}$: simply testing all the solutions in a naïve way results in $2 \cdot \binom{64}{4} \cdot 2^{64} \approx 2^{84}$ logical operations, which would make the attack hardly feasible even on today's best available hardware.

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

On the practical side, we have implemented our new algorithms on x86 CPUs and on NVIDIA GPUs. While our CPU implementation is fairly optimized using vector instructions, our GPU implementation running on one single NVIDIA GTX 295 graphics card runs up to 13 times faster than the CPU implementation using all four cores of an Intel quad-code Core i7 at 3 GHz, one of the fastest CPUs currently available. Today, we can solve 48+ quadratic equations in 48 binary variables using just an NVIDIA GTX 295 graphics card in 21 minutes. This device is available for about $500. It would be 36 minutes for cubic equations and two hours for quartics. The 64-bit signature challenge [21] 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 $\mathrm{F}_4$ in `MAGMA-2.16`, often cited as the best Gröbner-basis solver *commercially* available today, will completely use up 64 GB of RAM in solving just 25 cubic equations in as many $\mathbb{F}_2$-variables. We have also tested it with overdefined systems, for which Gröbner-basis algorithms are known to work better. While it does not run out of memory, the results are not satisfying: 2.5 hours to solve 20 cubic equations in 20 variables, half an hour for 45 quadratic equations in 30 variables, and 7 minutes for 60 quadratic equations in 30 variables on one 2.2-GHz Opteron core. Some very recent improvements on Gröbner-basis solvers have reported speed-up over MAGMA $\mathbf{F_4}$ of two- to five-fold [11]. However, even with such significant improvements, Gröbner-basis solvers do not seem to be able to compete with exhaustive search algorithms in this range, as each of the above is solved in a split second using negligible amount of memory on the same CPU by the latter.

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

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

**Implications.** The new exhaustive search algorithm can be used as a black box in cryptanalysis that needs to solve quadratic equations. This includes, for instance, several algorithms for the Isomorphism of Polynomials problem [8, 24], as well as attacks that rely on such algorithms, e.g., [9].

We also show with a concrete example that (relatively simple) computations requiring $2^{64}$ operations can be easily carried out in practice with readily available hardware and a modest budget. Lastly, we highlight the fact that GPUs have been used successfully by the cryptographic community to obtain very efficient implementations of combinatorial algorithms or cryptanalytic attacks, in addition to the more numeric-flavored cryptanalysis algorithm demonstrated by the implementation of the ECM factorization algorithm on GPUs [6].

**Organization of the Paper.** Section 2 establishes a formal framework of exhaustive search algorithms including useful results on Gray Codes and derivatives of multivariate polynomials. Known exhaustive search algorithms are reviewed in Section 3. Our algorithm to find the zeroes of a single polynomial of any degree is given in Section 4, and it is extended to find the common zeroes of a collection of polynomials in Section 5. Section 6 describes the two platforms on which we implemented the algorithm, and Section 8 describes the implementation and performance evaluation results.

## 2 Generalities

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

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

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

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

**Lemma 2.** *For* $j \in \mathbb{N}$*:*

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

*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. □

**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:

$$f(\mathbf{x} \oplus e_i) = f(\mathbf{x}) \oplus \frac{\partial f}{\partial i}(\mathbf{x}) \quad (1)$$

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 $\mathbb{F}_2$) : $c_i = \frac{\partial f}{\partial i}(0) = f(e_i) \oplus f(0)$. Then, the function $\mathbf{x} \mapsto \frac{\partial f}{\partial i}(\mathbf{x}) \oplus c_i$ is by definition a linear form and can be represented by a vector $D_i \in (\mathbb{F}_2)^n$. More precisely, we have $D_i[j] = f(e_i \oplus e_j) \oplus f(e_i) \oplus f(e_j) \oplus f(0)$. Then equation (1) becomes:

$$f(\mathbf{x} \oplus e_i) = f(\mathbf{x}) \oplus D_i \cdot \mathbf{x} \oplus c_i \quad (2)$$

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

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

Finding all the zeroes of $f$ is then achieved with the loop shown in fig. 1.

## 3 Known Techniques for Quadratic Polynomials

We briefly discuss the enumeration techniques known to the authors.

```
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:       NEXT(State)
6:    end for
7: end procedure
```

Fig. 1: Main loop common to all enumeration algorithms.

**Naive Evaluation.** The simplest way to implement an enumeration algorithm is to evaluate the polynomial $f$ from scratch at each point of $(\mathbb{F}_2)^n$. If $f$ is of degree $d$, this requires $(d-1)$ AND per monomial, and nearly one XOR per monomial. 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*, and that multiplication of a monomial by its coefficient come for free. Each call to NEXT would then require at most $d \cdot \binom{n}{d}$ bit operations, $1/d$ of which being XORs and the rest being ANDs (not counting the cost of enumerating $(\mathbb{F}_2)^n$, *i.e.*, incrementing a counter). This can be improved a bit, using what is essentially a multivariate Hörner evaluation technique. If $f$ is quadratic, it can be written:

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

If $f$ is cubic, it can be written:

$$f(\mathbf{x}) = c \oplus \sum_{i=0}^{n-1} \mathbf{x}_i \cdot \left( c_j \oplus \sum_{j=i+1}^{n-1} \mathbf{x}_j \cdot \left( c_{ij} \oplus \sum_{k=j+1}^{n-1} a_{ijk} \cdot \mathbf{x}_k \right)\right)$$

And so on and so forth. The required numbers of operations in this representation is given by:

$$N_{AND} = \sum_{k=1}^{d-1} \binom{n}{k} \qquad N_{XOR} = \sum_{k=1}^{d} \binom{n}{k}$$

This method is 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 $\omega$, where $\omega$ 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 $f(\mathbf{x} \oplus e_i)$ amounts to compute $\frac{\partial f}{\partial i}(\mathbf{x})$. If $f$ is quadratic, and in this case only, 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 (see 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 literature. Each call to NEXT requires $n$ ANDs, as well as $n+2$ XORs, which makes a total bit operation count of $2(n+1)$. This is about $n/4$ times less than the naive method applied to a quadratic $f$. 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.

| |
|---|
| 1: **function** INIT($f, k_0, \mathbf{x}_0$) |
| 2: $i \leftarrow 0$ |
| 3: $\mathbf{x} \leftarrow \mathbf{x}_0$ |
| 4: $\mathbf{y} \leftarrow f(\mathbf{x}_0)$ |
| 5: **For all** $0 \leq k \leq n-1$, initialize $c_k$ and $D_k$ |
| 6: **end function** |

| |
|---|
| 1: **function** NEXT($State$) |
| 2: $i \leftarrow i+1$ |
| 3: $k = b_1(i)$ |
| 4: $\mathbf{z} \leftarrow$ VECTORVECTORPRODUCT $(D_k, \mathbf{x}) \oplus c_k$ |
| 5: $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}$ |
| 6: $\mathbf{x} \leftarrow \mathbf{x} \oplus e_{k+k_0}$ |
| 7: **end function** |

Fig. 2: The Folklore differential algorithm.

## 4 A Faster Recursive Algorithm for any Degree

We now describe one of the main contributions of this paper, a new algorithm which is both asymptotically and practically faster than other known exhaustive search techniques in evaluating a polynomial of any degree on all the points of $(\mathbb{F}_2)^n$.

### 4.1 Intuition

In the folklore differential algorithm of fig. 2, the dominating part is the scalar product computed in line 4 of NEXT. It would be great if it were possible to exploit the fact that $\mathbf{x}$ is only slightly changed between to calls to NEXT. The problem is that $k$ (defined on line 3) is never the same in two consecutive iterations. Now assume we modify the function this way:

| |
|---|
| 1: **function** NEXT($State$) |
| 2: $i \leftarrow i+1$ |
| 3: $k = b_1(i)$ |
| 4: $\mathbf{z}[k] \leftarrow$ VECTORVECTORPRODUCT $(D_k, \mathbf{x}) \oplus c_k$ |
| 5: $\mathbf{x}[k] \leftarrow \mathbf{x}$ |
| 6: $\mathbf{y} \leftarrow \mathbf{y} \oplus \mathbf{z}[k]$ |
| 8: $\mathbf{x} \leftarrow \mathbf{x} \oplus e_k$ |
| 9: **end function** |

Then, on line 4, the *previous* value of $\mathbf{z}[k]$, when it exists, is still available, and this value is the scalar product of $D_k$ with $\mathbf{x}[k]$ (which is the previous value of $\mathbf{x}$ for the same value of $k$). Thus, the new value of $\mathbf{z}[k]$ is going to be $\mathbf{z}[k] \oplus D_k \cdot (\mathbf{x} \oplus \mathbf{x}[k])$. The key observation is proposition 1 below, as its consequence is that the computation of the scalar product can be done in constant time, with two ANDs and one XOR.

**Proposition 1.** *At the beginning of the function,* $\mathbf{x}^\top \oplus \mathbf{x}[k]^\top$ *has a hamming weight upper-bounded by two.*

*Proof.* Indeed, $\mathbf{x}[k_0]^\top$ is only accessed and modified when $b_1\left(i^\top + 1\right) = k_0$, for any given $k_0$. The integers $u$ such that $b_1(u) = k_0$ are precisely the integers written $u = 2^{k_0} + j \cdot 2^{k_0+1}$, for $j \geq 0$. Then, if we consider the values of the variables at the beginning of the function, by invariant $ii$, we have for some $j$:

$$\mathbf{x}^\top = \text{GRAYCODE}\left(2^k + (j+1) \cdot 2^{k+1}\right)$$
$$\mathbf{x}[k]^\top = \text{GRAYCODE}\left(2^k + j \cdot 2^{k+1}\right)$$

Thus, it follows from lemma 2 that just before line 1 is executed, we have:

$$\begin{aligned}\mathbf{x}^\top \oplus \mathbf{x}[k]^\top &= e_k \oplus (\text{GRAYCODE}(j) \ll (k+1)) \oplus (\text{GRAYCODE}(j+1) \ll (k+1)) \\ &= e_k \oplus ((\text{GRAYCODE}(j) \oplus \text{GRAYCODE}(j+1)) \ll (k+1))\end{aligned}$$

and by lemma 1,

$$\mathbf{x}^\top \oplus \mathbf{x}[k]^\top = e_k \oplus e_{k+1+b_1(j+1)} \tag{4}$$

□

By looking closely at the proof of proposition 1, we can write an *optimized differential algorithm*. However, before that, a few details still need to be addressed.

- The first time that $b_1(i) = k$, then $\mathbf{z}[k]$ is not defined. In this case, we in fact know that $i = 2^k$. Therefore, special care must be taken to initialize $\mathbf{z}[k]$ when $b_2(i) = -1$, which is equivalent to saying that the hamming weight of $i$ is less than two. In that case, by invariant ii, we have:

$$\mathbf{x} = \begin{cases} e_0 & \text{if } i = 1 \\ e_k \oplus e_{k-1} & \text{if } i = 2^k \text{ and } k > 0 \end{cases}$$

- Also note that with the notation $k_1 = b_1(i)$ and $k_2 = b_2(i)$, then if $b_2(i) \neq -1$, equation (4) becomes:

$$\mathbf{x} \oplus \mathbf{x}[k] = e_{k_1} \oplus e_{k_2}$$

  And thus,

$$\textsc{VectorVectorProduct}(D_{k_1}, \mathbf{x} \oplus \mathbf{x}[k_1]) = D_{k_1}[k_1] \oplus D_{k_1}[k_2]$$

  This last formula can be further simplified by observing that $D_{k_1}[k_1] = 0$.

All these considerations lead to the algorithm shown in fig. 3. Note that the conditional statement could be removed by unrolling the loop carefully. The critical part of the algorithm is therefore an extremely reduced section of the code, that performs two XORs, increment a counter, and evaluate $b_1$ as well as $b_2$. The cost of maintaining $i$, $k_1$ and $k_2$ can again be reduced greatly by unrolling the loop.

```
1:  function INIT(f, k_0, x_0)
2:     i ← 0
3:     x ← x_0
4:     y ← f(x_0)
5:     For all 0 ≤ k ≤ n − 1,
6:        initialize c_k and D_k
7:     End for
8:     z[0] ← c_0
9:     For all 1 ≤ k ≤ n − 1,
10:       z[k] ← D_k[k − 1] ⊕ c_k
11:    End for
12: end function
```

```
1:  function NEXT(State)
2:     i ← i + 1
3:     k_1 = b_1(i)
4:     k_2 = b_2(i)
5:     if k_2 ≠ −1 then
6:        z[k_1] ← z[k_1] ⊕ D_{k_1}[k_2]
7:     end if
8:     y ← y ⊕ z[k_1]
9:     x ← x ⊕ e_{k_0+k_1}
10: end function
```

Fig. 3: An optimized differential enumeration algorithm for quadratic forms.

### 4.2 Recursive Generalization to Any Degree.

It is in fact possible to generalize the improvement of the folklore differential algorithm that lead to the optimized differential algorithm in the quadratic case. The core idea is that in this algorithm, a given derivative is evaluated on the consecutive points of something that looks very much like a Gray code. This suggest using the technique recursively.

To make this thing explicit, we introduce a new State for each of the derivatives of $f$ used in the enumeration of $f$. Instead of storing $\mathbf{x}[k]$ and $\mathbf{z}[k]$, we will access $Derivative[k].\mathbf{y}$ and $Derivative[k].\mathbf{y}$. Also, $Derivative[k].i$ will count the number of times $b_1(k)$ happened. We now reformulate our optimized algorithm in this framework. However, know, the $x_0$ and $k_0$ parameters appearing in invariant $ii$ will play a more important role.

Terminal case when $f$ is of degree 0

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

```
1: function NEXT(State)
2:    i ← i + 1
3:    k = b_1(i)
4:    x ← x ⊕ e_{k+k_0}
5: end function
```

Recursive case when $\deg f > 0$.

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

```
1: function NEXT(State)
2:    i ← i + 1
3:    k = b_1(i)
4:    x ← x ⊕ e_{k+k_0}
5:    y ← y ⊕ Derivative[k].y
6:    NEXT(Derivative[k])
7: end function
```

Fig. 4: The recursive differential for all degrees.

**Correctness.** At first glance, it may not seem trivial that the combination of algorithms 1 and 4 results in a method for finding all the zeroes of $f$.

We prove by induction on the degree of $f$ that the INIT and NEXT functions described in fig. 4 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 "base case" of the algorithm shown in fig. 1 correctly enumerates the values of a constant polynomial...

In the recursive case where $f$ is not constant, 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 beginning 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 4. 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. We will always denote by $x^\top$ (resp. $x^\perp$) the value that the $x$ variable had at the begining of the execution of the function (resp. at the end).

**Lemma 3.** *After $k$ is updated on line 3 of* NEXT*, we have:*

$$i^\top + 1 = 2^k + Derivative[k].i \times 2^{k+1}.$$

*Proof.* It is not difficult to see that the $\ell$-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 beginning of the main loop (not counting $i^\top + 1$). This simply follows from the fact that $Derivative[k].i$ counts the number of times NEXT $(Derivative[k])$ has been called. □

**Lemma 4.** *Let $\Pi : (\mathbb{F}_2)^n \to (\mathbb{F}_2)^n$ denote the projection that sets the $(k + k_0)$-th coordinate to zero. After $k$ is updated on line 3, and before $\mathbf{x}$ is updated on line 4, we have:*

$$\Pi\left(\mathbf{x}^\top \oplus Derivative[k].\mathbf{x}\right) = 0$$

*Proof.* By assuming that invariant $ii$ holds for the current state at the entry of NEXT, we have:

$$\mathbf{x}^\top = x_0 \oplus \left(\text{GRAYCODE}\left(i^\top\right) \ll k_0\right).$$

Because after line 3 $k$ is set to $b_1\left(i^\top + 1\right)$, it follows from lemma 1 that:

$$\begin{aligned}\mathbf{x}^\top &= x_0 \oplus \left(\left(\text{GRAYCODE}\left(i^\top + 1\right) \oplus e_k\right) \ll k_0\right) \\ &= x_0 \oplus e_{k+k_0} \oplus \left(\text{GRAYCODE}\left(i^\top + 1\right) \ll k_0\right)\end{aligned}$$

Then, because lemma 3 grants us: $i^\top + 1 = 2^k + Derivative[k].i \times 2^{k+1}$, this becomes:

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

Applying lemma 2 gives:

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

We now distinguish two cases.

- Either $k > 0$, and since $\frac{\partial f}{\partial k + k_0}$ is of strictly smaller degree than $f$, then by induction hypothesis on $Derivative[k]$, invariant $ii$ grants:

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

  And thus:

$$\begin{aligned}\Pi\left(\mathbf{x}^\top \oplus Derivative[k].\mathbf{x}\right) &= \Pi\left(\left(\text{GRAYCODE}\left(2^k\right) \ll k_0\right) \oplus e_{k+k_0-1}\right) \\ &= \Pi\left(\left(e_{k+k_0} \oplus e_{k+k_0-1}\right) \oplus e_{k+k_0-1}\right) \\ &= 0\end{aligned}$$

- Or $k = 0$, and by induction hypothesis invariant $ii$ yields:

$$Derivative[0].\mathbf{x} = x_0 \oplus \left(\text{GRAYCODE}\left(Derivative[0].i\right) \ll k_0 + 1\right)$$

  Next,

$$\begin{aligned}\Pi\left(\mathbf{x}^\top \oplus Derivative[0].\mathbf{x}\right) &= \Pi\left(\text{GRAYCODE}\left(2^k\right) \ll k_0\right) \\ &= \Pi\left(e_{k_0}\right) \\ &= 0\end{aligned}$$

□

**Lemma 5.** *We have $\mathbf{y}^{\perp} = f(\mathbf{x}^{\perp})$. In other terms, invariant i is preserved.*

*Proof.* By induction hypothesis on $Derivative[k]$, invariant $i$ :

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

However, because we are in characteristic two, we have: $\frac{\partial f}{\partial k+k_0} = \frac{\partial f}{\partial k+k_0} \circ \Pi$, and lemma 4 in fact grants us:

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

So, this yields (using lemma 1):

$$\begin{aligned} \mathbf{y}^{\perp} &= \mathbf{y}^{\top} \oplus \frac{\partial f}{\partial k + k_0}\left(\mathbf{x}^{\top}\right) \\ &= f\left(\mathbf{x}^{\top}\right) \oplus f\left(\mathbf{x}^{\top}\right) \oplus f\left(\mathbf{x}^{\top} + e_{k+k_0}\right) \\ &= f\left(\mathbf{x}^{\perp}\right) \end{aligned}$$

□

### 4.3 Time and Space Complexity Considerations

It should be clear from the description of NEXT that it has complexity $\mathcal{O}(d)$. Therefore, the complexity of enumerating all the values of $f$ on $(\mathbb{F}_2)^n$ can be done with complexity $\mathcal{O}(d \cdot 2^n)$. What is the space requirement of the algorithm? The answer to this question is twofold: there is an *internal state* that gets modified by the algorithm, and that correspond to the $\mathbf{y}$ field of all the non-constant derivatives. There is also an array of *constants*, which is only read from the memory, and that correspond to the $\mathbf{y}$ field of degree-$d$ derivatives.

INIT stores one bit per degree-$d$ derivative $\partial f / \partial i_1 \partial i_2 \dots \partial i_d$, with $1 \leq i_1 < i_2 < \cdots < i_d \leq n$. The number of such tuples $(i_1, i_2, \dots, i_d)$ is known to be $\binom{n}{d-1}$. This yields the following result:

**Proposition 2.** *The algorithm allocates* $\sum_{i=0}^{d-1} \binom{n}{i}$ *bits of internal state and* $\binom{n}{d}$ *bits of constants*

### 4.4 An iterative Version

In section 4.2, we gave a recursive algorithm that works for all degree, which is a generalized version of the iterative algorithm described only in the quadratic case in section 4.1. Indeed, one could check that unrolling the algorithm of fig. 4 with a quadratic $f$ gives back the algorithm of fig. 3. We now move on to write an iterative version of the general recursive algorithm. This iterative version allows more optimization, such as the removal of extra useless work, and a more careful parallel scheduling.

But first, the function NEXT$_2$ shown in fig. 5 does exactly the same thing as NEXT, but in a slightly different way. Instead of calling NEXT at the end, it calls it at the beginning, except the first time a given value of $k$ is reached, to avoid calling it an extra time at the begining.

We can therefore work on NEXT$_2$. A first remark is that maintaining $\mathbf{x}$ is required by the invariants, but is otherwise useless for the actual computation. A first step is to completely remove $\mathbf{x}$ from the algorithm. Less obviously, we can also avoid maintaining $i$. To see that, we first need to state an equivalent of lemma 3 adapted to NEXT$_2$, the proof of which is left to the reader.

```
1: function NEXT_2(State)
2:     i ← i + 1
3:     k = b_1(i)
4:     if i ≠ 2^k then
5:         NEXT_2 (Derivative [k])
6:     end if
7:     x ← x ⊕ e_{k+k_0}
8:     y ← y ⊕ Derivative[k].y
9: end function
```

Fig. 5: An equivalent version of NEXT.

**Lemma 6.** *After $k$ is updated on line 3 of* NEXT$_2$*, we have:*

$$i^\top + 1 = 2^k + (Derivative[k].i + 1) \times 2^{k+1}.$$

It is an easy consequence of lemma 6 that in NEXT$_2$, after $k$ is updated on line 3, we have for any $j$:

$$b_j(Derivative[k].i + 1) = b_{j+1}(i^\top + 1).$$

Thus, it is possible to avoid storing the $i$ values, except in the main loop, and to re-generate online by evaluating $b_j$ on the index of the main loop. These computations, although taking amortized constant time, can be made negligible by unrolling. To ease notation, we introduce the following shorthand:

$$D[k_1, k_2, \ldots, k_\ell] \stackrel{\Delta}{=} State.Derivative[k_1].Derivative[k_2] \ldots .Derivative[k_\ell].\mathbf{y}$$

With this notation, the algorithm of fig. 6 is just an unrolled version of the recursive algorithm of fig. 4 in which all the useless operations have been removed.

```
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 GRAYCODE(i) is a zero of f
5:          k_1 = b_1(i + 1)
6:          k_2 = b_2(i + 1)
7:          ...
8:          k_d = b_d(i + 1)
9:          if k_d > −1 then D[k_1, ..., k_{d−1}] ← D[k_1, ..., k_{d−1}] ⊕ D[k_1, ..., k_{d−1}, k_d]
10:         ...
11:         if k_3 > −1 then D[k_1, k_2] ← D[k_1, k_2] ⊕ D[k_1, k_2, k_3]
12:         if k_2 > −1 then D[k_1] ← D[k_1] ⊕ D[k_1, k_2]
13:         y ← y ⊕ D[k_1]
14:     end for
15: end procedure
```

Fig. 6: Iterative algorithm for all degrees.

## 5 Enumeration of Several Multivariate Polynomials Simultaneously

In the previous section, we discussed how to enumerate one single polynomial. We now move on to the enumeration of several polynomial simultaneously.

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) the algorithm of fig. 6 could be run in parallel on $f_1, \ldots, f_m$. All the parallel runs would execute the same instruction on different data, making it efficient to implement on vector or SIMD architectures. In each iteration of the main loop, it is easy to check if *all* the polynomials vanished on the current point of $(\mathbb{F}_2)^n$. Evaluating all the $m$ polynomials in parallel using the algorithm of fig. 6 would require roughly $m \cdot d \cdot 2^n$ bit operations. The point of this section is that it is possible to do much better than this.

Let us first 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 $Z_i = Z\left([f_i], Z_{i-1}\right)$. 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.

### 5.1 General Technique: Splitting the Problem

A possible strategy is to compute the $Z_i$ recursively: first $Z_1$, then $Z_2$, etc. However, while the algorithms of section 4 can be used to compute $Z_1$, they cannot be used to compute $Z_2$ from $Z_1$, because they 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 the algorithm of fig. 6, and then computing $Z_m$ using a *secondary algorithm*. Computing $Z_k$ requires $d \cdot k \cdot 2^n$ bit operations. It then remains to compute $Z_m$ from $Z_k$, and to find the best possible value of $k$.

Note that if $m > n$, we can focus on the first $n$ equations, since a system of $n$ randomly chosen multivariate polynomial equations in $n$ variables of constant degree $d$ is expected to 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 it makes sense to choose $k$ according to the targeted hardware platform (*e.g.*, $k = 32$ if only 32-bit registers are available), it is an interesting theoretical problem choose $k$ in order to minimize the global number of bit operations.

We now move on to discuss several secondary algorithms to compute $Z_m$ from $Z_k$, and discuss their relative merits.

### 5.2 Naive Secondary Evaluation

We compute $Z_{i+1}$ from $Z_i$ using naive evaluation, for $k \leq i \leq n-1$. It is clear that the expected cardinality of $Z_i$ for random polynomial equations is $2^{n-i}$. We will assume for the sake of simplicity that evaluating a degree-$d$ polynomial requires $\binom{n}{d}$, following the reasoning in section 3. Computing $Z_{i+1}$ then takes about $\binom{n}{d} \cdot 2^{n-i}$ bit ops. The expected cost of computing $Z$ is then approximately:

$$\sum_{i=k}^{n} \binom{n}{d} \cdot 2^{n-i} \approx \binom{n}{d} \cdot 2^{n-k+1} \text{ bit operations.}$$

Minimizing the global cost means solving the equation:

$$k \cdot d \cdot 2^n = \binom{n}{d} \cdot 2^{n-k+1}.$$

which is easily seen to be equivalent to:

$$(k \cdot \ln 2) \cdot \exp(k \cdot \ln 2) = 2 \cdot \binom{n}{d} \cdot \frac{\ln 2}{d}$$

Now, the Lambert $W$ function is such that $W(x) \cdot \exp(W(x)) = x$. Thus, the solution of our equation is:

$$k = W\left(\binom{n}{d} \cdot \frac{2 \cdot \ln 2}{d}\right) / \ln 2$$

Using the known fact [15] that when $x$ goes to infinity:

$$W(x) = \ln x - \ln \ln x + o(\ln \ln x)$$

we find that when $n \to \infty$:

$$k = 1 + \log_2 \left(\binom{n}{d} \cdot \frac{1}{d}\right) + \mathcal{O}\left(\ln \ln n\right)$$

The full cost of the algorithm is then approximately $d^2 \cdot \log_2 n \cdot 2^{n+1}$ bit operations..

### 5.3 Differential Secondary Evaluation

We only describe the quadratic case, but this could be extended to higher degrees. We can efficiently evaluate $Z_{i+1}$ from $Z_i$ using an easy consequence of equation (1): given $f(\mathbf{x})$, computing $f(\mathbf{x} + \varDelta)$ takes $2|\varDelta| \cdot n$ bit operations, where $|\varDelta|$ denote the hamming weight of $\varDelta$, by computing $|\varDelta|$ vector-vector products with the derivatives. Let us order the elements of $Z_i$ by writing: $Z_i = \left\{\mathbf{x}^i_1, \dots \mathbf{x}^i_{q_i}\right\}$ (the elements are ordered using the usual lexicographic order), and $\varDelta^i_j = \mathbf{x}^i_{j+1} \oplus \mathbf{x}^i_j$.

Computing $Z_{i+1}$ therefore requires approximately:

$$2n \cdot \sum_{j=1}^{q_i - 1} |\varDelta^i_j| \text{ bit operations.}$$

Now, let us consider the $\varDelta^i_j$ as integer number between 0 and $2^n - 1$. The $x^i_{j+1}$ are the zeroes of a set of $i$ random polynomials, and under the assumption that each point of $(\mathbb{F}_2)^n$ has one chance over $2^i$ to be such a zero, then the difference $\varDelta^i_j$ between two such consecutive zeros follows a geometric distribution of parameter $2^{-i}$, and thus has expectation $2^i$. The hamming weight $|\varDelta^i_j|$ is upper-bounded by $\lceil \log_2 \varDelta^i_j \rceil$ (considered as an integer), and therefore $|\varDelta^i_j|$ has expectation less than $i$.

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:

$$2n \cdot \sum_{i=k}^{n-1} i \cdot 2^{n-i} \leq 4n \cdot (k+1) \cdot 2^{n-k} \text{ bit operations}$$

An approximately optimal value of $k$ would then satisfy

$$2k \cdot 2^n = 4n \cdot (k+1) \cdot 2^{n-k}$$

which is approximately $k = 1 + \log_2 n$. The complexity of the whole procedure is then $4 \log_2 n \cdot 2^n$. However, implementing this technique efficiently looks like a lot of work for at best a $2\times$ gain.

# 6 A Brief Description of the Hardware Platforms

## 6.1 Vector Units on x86-64

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

The highly non-orthogonal SSE instruction set includes Loads and Stores (to/from `xmm` registers, memory — both aligned and unaligned, and traditional registers), Packing/Unpacking/Shuffling, Logical Operations (`AND, OR, NOT, XOR`, Shifts Left, Right Logical and Arithmetic — bit-wise on units and byte-wise on the entire `xmm` register), and Arithmetic (add, substract, multiply, max-min) with some or all of the arithmetic widths. The interested reader is referred to Intel and AMD's manuals for details on these instructions, and to references such as [19] for throughput and latencies.

## 6.2 G2xx-series Graphics Processing Units from NVIDIA

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

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

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

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

# 7 Parallelization and Memory Bandwith Issues

The critical loop of the algorithm is very short, since it performs only $d$ logical operations. However, it accesses the memory $d+1$ times, which suggests that the memory bandwith will be the actual performance bottleneck. We address this issue in two complementary ways. First we argue that the algorithm is *cache-oblivious* [20], *i.e.*, that is uses the cache efficiently regardless of its size. Then we argue that on massively concurrent architectures such as GPUs, then any word read from the memory can be broadcast to all the concurrently running threads almost systematically.

### 7.1 Spatial and Temporal Proximity on Iterative Architectures

We will study the behavior of the algorithm in the *Ideal Cache Model*. This model consists of a computer with a two-level memory hierarchy consisting of an ideal (data) cache of $Z$ words and an arbitrarily large main memory. The cache is partitioned into cache lines, each consisting of L consecutive words that are always moved together between cache and main memory. The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a *cache hit* occurs, and the word is delivered to the processor. Otherwise, a *cache miss* occurs, and the line is fetched into the cache. The ideal cache is *fully associative*: cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line strategy of replacing the cache line whose next access is farthest in the future, and thus it exploits temporal locality perfectly. An algorithm with an input of size $n$ is measured in the ideal-cache model in terms of the usual number of operation performed by the processor, but also in terms of its *Cache Complexity* $Q(n, Z, L)$ – the number of cache misses it incurs as a function of $Z$ and $L$. We now move on to evaluate the cache complexity of the enumeration algorithm, as show in fig. 6. We will assume that the "memory cells" accessed by the algorithm have the same size as the a word in the cache (if this were not the case, it would only incur a constant multiplicative loss, and we are mostly interested in an asymptotic result).

The memory words accessed in the algorithm belong to arrays of various dimension, and are accessed with indices of variable length. It should be clear from the description of the algorithm that for all $k \leq d$, the memory location of index $[i_1, \dots, i_k]$ is accessed at step $s$ if $b_j(s) = i_j$, for all $j \leq k$. This memory access pattern is in fact very regular. We say that a memory word is accessed with period $T$ if, when it is accessed at iteration $i$, it is also accessed at iteration $i + T$, but not in-between.

**Lemma 7.** *For all $k \leq d$, the memory location of index $[i_1, \dots, i_k]$ is accessed with period $2^{i_k+1}$.*

*Proof.* We associate with an index $[i_1, \dots, i_k]$ the set $\Omega_{i_1,\dots,i_k}$ of integers $n$ such that $b_1(n) = i_1, \dots, b_k(n) = i_k$. The problem amounts to show that the difference between two consecutive elements of this set is $2^{i_k+1}$. But it is easily seen that if $n \in \Omega_{i_1,\dots,i_k}$, then $n + j \cdot 2^{i_k+1} \in \Omega_{i_1,\dots,i_k}$ for any positive integer $j$. This follows from the fact that $b_j(n) = b_j(n + 2^\ell)$ if $\ell > j$, and establishes the result. □

It should be clear that all the memory location accessed with period exactly $T$ are accessed in the first $T$ iteration of the main loop. Moreover, they are accessed in a certain order. For instance, memory words with period 8 are accessed in this order in the first 8 iterations: $[2], [0, 2], [1, 2]$. By definition of the period, this access pattern is reproduced without modifications in the next $T$ iterations. Thus, memory words with period $T$ are accessed in a *cyclic* fashion.

The algorithm easily defines a total order relation on the memory locations it accesses: $x \leq y$ if and only if the first access to $x$ takes places *before* the first access to $y$. Let us assume that the actual memory addresses are compatible with this order relation. Then, more frequently accessed words are stored with the lowest addresses, and words with the same access frequency are stored contiguously in memory. There are $\sum_{i=0}^{\min(d-1,k)} \binom{k}{i}$ memory locations that are accessed with period $2^{k+1}$.

This being said, we will focus on the case where all the $\sum_{i=0}^{d} \binom{n}{i}$ memory words accessed by the algorithm do not fit into the cache, to avoid studying the trivial case. Let us define the *critical period* $2^{T_c+1}$ to be the biggest integer such that all the memory words accessed with period $2^{T_c+1}$ fit in the cache:

$$\sum_{k=0}^{T_c} \sum_{i=0}^{\min(d-1,k)} \binom{k}{i} \leq Z - 1$$

Under the (mild) assumption that the cache contains $Z \geq 2^d$ words, and thus that $T_c$ is greater than $d$, this condition becomes:

$$2^d - 1 + \sum_{k=d}^{T_c} \sum_{i=0}^{d-1} \binom{k}{i} \leq Z - 1$$

This this is the summation in a rectangle inside Pascal's triangle, then by applying Pascal's rule recursively, we may simplify this expression, and find that it is equivalent to:

$$\sum_{i=0}^{d} \binom{T_c+1}{i} \leq Z$$

$T_c$ can be easily expressed as a function of $Z$ when $d$ is small:

$$d = 2 \rightarrow T_c + 1 = \frac{\sqrt{8Z-7}-1}{2}$$

$$d = 3 \rightarrow T_c + 1 = \frac{\left(3 \cdot (Z-1) + \sqrt{Z^2 - 2Z + 368/243}\right)^{\frac{2}{3}} - 15}{\left(3 \cdot (Z-1) + \sqrt{Z^2 - 2Z + 368/243}\right)^{\frac{1}{3}}}$$

The important point is that all memory words with period $2^{T_c+1}$ fit in the cache and *do not leave it*. This fact is almost true by definition of $T_c$: the optimal off-line cache strategy will not evict a cache line that will be accessed in $T$ steps if it can evict a cache line that will only be accessed in $2T$ steps. And there will always be a cache line not containing a word accessed with period $2^{T_c+1}$ or less. This being said, we can state our result:

**Proposition 3.** *Under the assumption that $T_c \geq 2d$, the following two inequalities hold:*

*i)* $Q(n,d,Z,L) \leq 2^{n-2-T_c} \cdot (d+1) \cdot \binom{T_c+1}{d-1}$

*ii)* $Q(n,d,Z,L) \leq 2^{n-2-T_c} \cdot \frac{d \cdot (d+1)}{T_c + 2 - d} \cdot Z$

*Proof.* The second inequality is a nearly-direct consequence of the first one, and of the definition of $T_c$. Let us thus focus on the first one.

By definition of $T_c$, the algorithm may make a cache miss every time it accesses a memory location whose index contain a coordinate bigger than $T_c$. Such memory words have period greater than $2^{T_c+2}$, so each of them is accessed at most $2^{n-T_c-2}$ times. Multiplying this by the number of such memory words yields the total number of cache misses:

$$\begin{aligned} Q(n,d,Z,L) &= \sum_{k=T_c+1}^{n} 2^{n-1-k} \cdot \sum_{i=0}^{d-1} \binom{k}{i} \\ &\leq 2^{n-1} \sum_{i=0}^{d-1} \sum_{k=T_c+1}^{+\infty} 2^{-k} \cdot \binom{k}{i} \end{aligned}$$

It is well-known that $\sum_k \binom{k}{i} x^k = x^i/(1-x)^{i+1}$. Thus, we find with $x = 1/2$:

$$\sum_{k=0}^{+\infty} 2^{-k} \cdot \binom{k}{i} = 2$$

We can therefore rewrite:

$$Q(n,d,Z,L) \leq 2^{n-1} \cdot \sum_{i=0}^{d-1} \left(2 - \sum_{k=0}^{T_c} 2^{-k} \cdot \binom{k}{i}\right)$$

Now we claim:

$$2 - \sum_{k=0}^{T_c} 2^{-k} \cdot \binom{k}{i} = 2^{-T_c} \cdot \sum_{j=0}^{i} \binom{T_c + 1}{j}$$

It is not particularly difficult to establish this by induction on $i$, by using Pascal's rule. Going back to the expression of $Q(n, d, Z, L)$, we find:

$$\begin{aligned} Q(n, d, Z, L) &\leq 2^{n-T_c-1} \cdot \sum_{i=0}^{d-1} \sum_{j=0}^{i} \binom{T_c + 1}{j} \\ &\leq 2^{n-1-T_c} \cdot \sum_{j=0}^{d-1} (d - j) \cdot \binom{T_c + 1}{j} \end{aligned}$$

And since $\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}$, we obtain:

$$Q(n, d, Z, L) \leq 2^{n-1-T_c} \cdot \left( d \cdot \sum_{j=0}^{d-1} \binom{T_c + 1}{j} - (T_c + 1) \cdot \sum_{j=1}^{d-1} \binom{T_c}{j - 1} \right)$$

We next claim that by applying Pascal's rule recursively, we obtain:

$$2 \cdot \sum_{j=1}^{d-1} \binom{T_c}{j - 1} = \binom{T_c}{d - 2} + \sum_{j=0}^{d-2} \binom{T_c + 1}{j}$$

Substituting this into $Q(n, d, Z, L)$ yields:

$$\begin{aligned} Q(n, d, Z, L) &\leq 2^{n-1-T_c} \cdot \left( d \cdot \sum_{j=0}^{d-1} \binom{T_c + 1}{j} - \frac{T_c + 1}{2} \cdot \left[ \binom{T_c}{d - 2} + \sum_{j=0}^{d-2} \binom{T_c + 1}{j} \right] \right) \\ &\leq 2^{n-1-T_c} \cdot \left( d \cdot \binom{T_c + 1}{d - 1} + \frac{2d - T_c - 1}{2} \cdot \sum_{j=0}^{d-2} \binom{T_c + 1}{j} - \frac{T_c + 1}{2} \cdot \binom{T_c}{d - 2} \right) \\ &\leq 2^{n-1-T_c} \cdot \left( \frac{(d + 1)(T_c + 1)}{2(d - 1)} \cdot \binom{T_c}{d - 2} + \frac{2d - T_c - 1}{2} \cdot \sum_{j=0}^{d-2} \binom{T_c + 1}{j} \right) \\ &\leq 2^{n-1-T_c} \cdot \left( \frac{d + 1}{2} \cdot \binom{T_c + 1}{d - 1} + \frac{2d - T_c - 1}{2} \cdot \sum_{j=0}^{d-2} \binom{T_c + 1}{j} \right) \end{aligned}$$

And under the (again mild) assumption that $T_c \geq 2d$, we find:

$$Q(n, d, Z, L) \leq 2^{n-2-T_c} \cdot (d + 1) \cdot \binom{T_c + 1}{d - 1}$$

□

Let us consider a polynomial in 64 variables. If we assume an incredibly small cache of $Z = 2^{10}$ bits and that our polynomial is of degree 2, then $T_c = 44$ and the enumeration will make about $2^{25}$ cache misses, for a running time of at least $2^{65}$. If we assume that our polynomial is of degree 4, and that the cache is $2^{14}$-bit large, then $T_c = 24$, and there will be $2^{52}$ cache misses, for more than $2^{66}$ memory accesses.

### 7.2 Constrained Small Memory Chips on Concurrent Architectures

The problem is formulated in very different terms on some parallel architectures, such as GPUs, or the Cell, in which the available "fast" memory is fairly restricted, and main memory is relatively slow.

Parallelizing the process is very easy, as it simply comes down to partition the search space into the number of available cores. An interesting side effect is that when done properly, this partition reduces the amount of data that needs to be transfered from the main memory. We will now assume that we have 32-bit registers, and we will use 32 parallel copies of the algorithm of fig. 6, to enumerate 32 polynomials simultaneously.

For instance, the loop of fig. 4 can be split in independent chunks, as illustrated by fig. 7. An additional benefit is that processing one such chunk only require access to a fraction of the memory used by the full enumeration. In fact, $b_1(i+1)$ is greater or equal than $L$ if the $L$ rightmost bits of $(i+1)$ are zero, or, in other terms, if $(i+1)$ is a multiple of $2^L$. This suggests to split the iteration in chunks of size $2^L$. Enumerating a chunk of size $2^L$ amounts to enumerate a polynomial in $L$ variables (it requires the same amount of internal state, and it makes the same number of calls to NEXT). Let us now consider the $k$-th chunk:

$$C_k = \left\{ i \in \mathbb{N} \mid k \cdot 2^L < i \leq (k+1) \cdot 2^L \right\}$$

```
1: procedure ParallelZeroes(f, L, T)
2:   for b from 0 to 2^{n-L-T} - 1 do
2:     parallel-for t from 0 to 2^T - 1 do
3:       State[t] ← Init (f, 0, GrayCode ((t + b · 2^T) · 2^L))
2:       for i from 0 to 2^L - 1 do
4:         if State[t].y = 0 then State[t].x is a zero of f
5:         Next(State[t])
6:     end for
7:   end-parallel for
6:     end for
8: end procedure
```

Fig. 7: Parallel enumeration, assuming one processing unit capable of running $2^T$ threads. It should be possible to improve it using the enumeration algorithm itself for initialization.

Let $\psi_L(i)$ denote the integer $(i \bmod 2^L)$. We will call $\psi_L(i)$ the *local part* of $i$ when $i \in C_k$, and we will denote it by $\psi(i)$, when not ambiguous. So, what can we say about $b_j(i)$, when $i \in C_k$? We define the subset $\Omega_{k,j}$ of $C_k$ to be such that if $i \in \Omega_{k,j}$, then $b_j(i)$ only depends on the local part $\psi(i)$ of $i$. Very clearly, we in fact have:

$$\Omega_{k,j} = \left\{ i \in C_k \mid \textsc{HammingWeight}(\psi(i)) \geq j \right\}$$

And the three following points are immediate to estalbish.

**Lemma 8.** *For any $k$ and $j$, we have the following properties:*

*i)* *If $j_1 < j_2$ then $\Omega_{k,j_2} \subseteq \Omega_{k,j_1}$*

*ii)* $|\Omega_{k,j}| = 2^L - \sum_{\ell=0}^{j-1} \binom{L}{\ell}$

*iii)* *If $i \in \Omega_{k,j}$, then $b_j(i) < L$.*

Intuitively, lemma 8 tells us that on a chunk of size $2^L$, the $b_j$ that we will compute will be smaller than $L$ except on $\mathcal{O}\left(L^{d-1}\right)$ points, and will only depend on the local part of the index. This has two interesting consequences:

1. Instead of having to store and maintain an internal state of $\sum_{i=0}^{d-1} \binom{n}{i} = \mathcal{O}\left(n^{d-1}\right)$ words, it is sufficient to deal with an internal state of $\sum_{i=0}^{d-1} \binom{L}{i} = \mathcal{O}\left(L^{d-1}\right)$ words.
2. If we were capable of processing all the chunks synchronously, the constant fetched from memory in line 9 of fig. 6 could be used by *all* the chunks *at the same time*, except on $\mathcal{O}\left(L^{d-1}\right)$ points. This means that *most of the time*, we can broadcast a single value to as many threads as possible, and we can amortize the lattency of the memory over the huge number of chunks processed in parallel.

Now that we controlled what happened *inside* $\Omega_{k,d}$, we may take a look at what happens *outside*. Generally speaking, if $\psi(i)$ has hamming weight $h$, and $j > h$, then $b_j(i) = L + b_{j-h}(k)$. Thus, if only a subset of all the chunks can be processed concurrently, it would make sense to treat simultaneously chunks for which $k$ has similar least-significant bits. If a processing unit can handle at most $2^T$ threads simultaneously, then the maximum sharing of values fetched from main memory is achieved by scheduling the $2^T$ Chunks sharing the $T$ most significant bit of $k$ on it.

What level of broadcast should we expect in this situation, namely when $2^t$ threads process chunks of size $2^L$ synchronously? To fully understand what is going on, let us define $H_{k,h} = \Omega_{k,h} - \Omega_{k,h+1}$. It is easily seen that $H_{k,h}$ describes the subset of $C_k$ formed of words of hamming weight exactly $h$, and thus $|H_{h,k}| = \binom{L}{h}$. Now, in all the chunks processed in parallel, the $c = n - L - t$ least significant bits of $k$ remain fixed to $\psi_c(k)$, therefore we will call this value the "fixed part of $k$". We already argued that if $i \in H_{h,k}$, then $b_j(i) = L + b_{j-h}(k)$, and the $2^T$ threads will fetch the same memory location if and only if $b_{j-h}(k)$ only depends on the fixed part of $k$, or, in other terms, if $\psi_c(k)$ has hamming weight at least $j - h$.

An easy consequence of the previous considerations is that if $\psi_c(k)$ has hamming weight at least $d$, then *all* the memory fetches issued on the $2^L$ steps can be broadcast to *all* the $2^T$ threads. If $\psi_c(k)$ has hamming weight $d-1$, then *all but one* memory fetches can be broadcast.

This raises the following question: assume we enumerate the $2^n$ points on a processing unit handling $2^T$ concurrent threads, each thread processing a chunk of size $2^L$ (we of course assume $n \geq L+T$). How many times we will witness non-broadcast memory accesses? Let us denote this number by $N_{NB}(d, n, T, L)$.

**Proposition 4.**

$$N_{NB}(d, n, T, L) = \sum_{i=0}^{d-1} \binom{n-T}{i}.$$

*Proof.* If our processing unit handles a batch of $2^T$ chunks of length $2^L$, then $2^c = 2^{n-T-L}$ batches will have to be processed sequentially, with all the possible values of $\psi_c(k)$.

When processing a given batch, non-broadcast memory accesses may happen when $\psi_c(k)$ has hamming weight less than $d$. Let us then assume that $\psi_c(k)$ has hamming weight $d - \ell$. There are $\binom{n-T-L}{d-\ell}$ such batches in the whole enumeration. Then non-broadcast may happen inside such a batch for $i \in C_k - \Omega_{k,\ell}$, and we know by lemma 8 that this set has size $\sum_{u=0}^{\ell-1} \binom{L}{u}$. This gives:

$$N_{NB}(d, n, T, L) = \sum_{\ell=1}^{d} \binom{n-T-L}{d-\ell} \cdot \sum_{u=0}^{\ell-1} \binom{L}{u}$$

And with a change of indices, we obtain:

$$N_{NB}(d, n, T, L) = \sum_{i=0}^{d-1} \binom{n-T-L}{i} \cdot \sum_{u=0}^{d-1-i} \binom{L}{u} \tag{5}$$

We now claim that $N_{NB}(d, n, T, L)$ is in fact independent of $L$. Indeed, setting $L = 0$ in (5) yields the expected result. Therefore, we now move on to prove that: $N_{NB}(d, n, T, L) = N_{NB}(d, n, T, L + 1)$, as this would allow to conclude by a trivial induction on $L$.

Let us introduce $\Delta(d) = N_{NB}(d + 1, n, T, L + 1) - N_{NB}(d + 1, n, T, L)$. We have:

$$\Delta(d) = \sum_{i=0}^{d} \sum_{u=0}^{d-i} \left[ \underbrace{\binom{n-T-L-1}{i} \cdot \binom{L+1}{u} - \binom{n-T-L}{i} \cdot \binom{L}{u}}_{X_{i,u}} \right]$$

It is easily seen that:

$$\Delta(d) = \Delta(d-1) + \sum_{i=0}^{d} X_{i,d-i}$$

We now demonstrate by induction on $d$ that $\Delta(d) = 0$. This is trivial if $d = 0$. By induction hypothesis, we find:

$$\Delta(d) = \sum_{i=0}^{d} X_{i,d-i}$$

Now, we introduce the notations:

$$\Delta_1 = \sum_{i=0}^{d} \binom{n-T-L-1}{i} \cdot \binom{L+1}{d-i}$$

$$\Delta_2 = \sum_{i=0}^{d} \binom{n-T-L}{i} \cdot \binom{L}{d-i}$$

We clearly have $\Delta(d) = \Delta_1 - \Delta_2$. Applying Vandermonde's indentity (see for instance [4, Identity 132]) yields:

$$\Delta_1 = \Delta_2 = \binom{n-T}{d}$$

And the result is established. □

Fig. 8 shows how the algorithm can be run with 4 threads and obtain the number of non-broadcasts advertised by the proposition.

## 8 Implementations

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

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

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

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

| thread 0 | | | | thread 1 | | | | thread 2 | | | | thread 3 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | $\psi_L(i)$ | $b_1(i)$ | $b_2(i)$ | $k$ | $\psi_L(i)$ | $b_1(i)$ | $b_2(i)$ | $k$ | $\psi_L(i)$ | $b_1(i)$ | $b_2(i)$ | $k$ | $\psi_L(i)$ | $b_1(i)$ | $b_2(i)$ | non-broadcast ? |
| 0 | 1 | 0 | -1 | 100 | 1 | 0 | 5 | 1000 | 1 | 0 | 6 | 1100 | 1 | 0 | 5 | ✓ |
| 0 | 10 | 1 | -1 | 100 | 10 | 1 | 5 | 1000 | 10 | 1 | 6 | 1100 | 10 | 1 | 5 | ✓ |
| 0 | 11 | 0 | 1 | 100 | 11 | 0 | 1 | 1000 | 11 | 0 | 1 | 1100 | 11 | 0 | 1 | |
| 0 | 100 | 2 | -1 | 100 | 100 | 2 | 5 | 1000 | 100 | 2 | 6 | 1100 | 100 | 2 | 5 | ✓ |
| 0 | 101 | 0 | 2 | 100 | 101 | 0 | 2 | 1000 | 101 | 0 | 2 | 1100 | 101 | 0 | 2 | |
| 0 | 110 | 1 | 2 | 100 | 110 | 1 | 2 | 1000 | 110 | 1 | 2 | 1100 | 110 | 1 | 2 | |
| 0 | 111 | 0 | 1 | 100 | 111 | 0 | 1 | 1000 | 111 | 0 | 1 | 1100 | 111 | 0 | 1 | |
| 0 | 0 | 3 | -1 | 100 | 0 | 3 | 5 | 1000 | 0 | 3 | 6 | 1100 | 0 | 3 | 5 | ✓ |
| 1 | 1 | 0 | 3 | 101 | 1 | 0 | 3 | 1001 | 1 | 0 | 3 | 1101 | 1 | 0 | 3 | |
| 1 | 10 | 1 | 3 | 101 | 10 | 1 | 3 | 1001 | 10 | 1 | 3 | 1101 | 10 | 1 | 3 | |
| 1 | 11 | 0 | 1 | 101 | 11 | 0 | 1 | 1001 | 11 | 0 | 1 | 1101 | 11 | 0 | 1 | |
| 1 | 100 | 2 | 3 | 101 | 100 | 2 | 3 | 1001 | 100 | 2 | 3 | 1101 | 100 | 2 | 3 | |
| 1 | 101 | 0 | 2 | 101 | 101 | 0 | 2 | 1001 | 101 | 0 | 2 | 1101 | 101 | 0 | 2 | |
| 1 | 110 | 1 | 2 | 101 | 110 | 1 | 2 | 1001 | 110 | 1 | 2 | 1101 | 110 | 1 | 2 | |
| 1 | 111 | 0 | 1 | 101 | 111 | 0 | 1 | 1001 | 111 | 0 | 1 | 1101 | 111 | 0 | 1 | |
| 1 | 0 | 4 | -1 | 101 | 0 | 4 | 5 | 1001 | 0 | 4 | 6 | 1101 | 0 | 4 | 5 | ✓ |
| 10 | 1 | 0 | 4 | 110 | 1 | 0 | 4 | 1010 | 1 | 0 | 4 | 1110 | 1 | 0 | 4 | |
| 10 | 10 | 1 | 4 | 110 | 10 | 1 | 4 | 1010 | 10 | 1 | 4 | 1110 | 10 | 1 | 4 | |
| 10 | 11 | 0 | 1 | 110 | 11 | 0 | 1 | 1010 | 11 | 0 | 1 | 1110 | 11 | 0 | 1 | |
| 10 | 100 | 2 | 4 | 110 | 100 | 2 | 4 | 1010 | 100 | 2 | 4 | 1110 | 100 | 2 | 4 | |
| 10 | 101 | 0 | 2 | 110 | 101 | 0 | 2 | 1010 | 101 | 0 | 2 | 1110 | 101 | 0 | 2 | |
| 10 | 110 | 1 | 2 | 110 | 110 | 1 | 2 | 1010 | 110 | 1 | 2 | 1110 | 110 | 1 | 2 | |
| 10 | 111 | 0 | 1 | 110 | 111 | 0 | 1 | 1010 | 111 | 0 | 1 | 1110 | 111 | 0 | 1 | |
| 10 | 0 | 3 | 4 | 110 | 0 | 3 | 4 | 1010 | 0 | 3 | 4 | 1110 | 0 | 3 | 4 | |
| 11 | 1 | 0 | 3 | 111 | 1 | 0 | 3 | 1011 | 1 | 0 | 3 | 1111 | 1 | 0 | 3 | |
| 11 | 10 | 1 | 3 | 111 | 10 | 1 | 3 | 1011 | 10 | 1 | 3 | 1111 | 10 | 1 | 3 | |
| 11 | 11 | 0 | 1 | 111 | 11 | 0 | 1 | 1011 | 11 | 0 | 1 | 1111 | 11 | 0 | 1 | |
| 11 | 100 | 2 | 3 | 111 | 100 | 2 | 3 | 1011 | 100 | 2 | 3 | 1111 | 100 | 2 | 3 | |
| 11 | 101 | 0 | 2 | 111 | 101 | 0 | 2 | 1011 | 101 | 0 | 2 | 1111 | 101 | 0 | 2 | |
| 11 | 110 | 1 | 2 | 111 | 110 | 1 | 2 | 1011 | 110 | 1 | 2 | 1111 | 110 | 1 | 2 | |
| 11 | 111 | 0 | 1 | 111 | 111 | 0 | 1 | 1011 | 111 | 0 | 1 | 1111 | 111 | 0 | 1 | |
| 11 | 0 | 5 | -1 | 111 | 0 | 6 | -1 | 1011 | 0 | 5 | 6 | 1111 | 0 | 7 | -1 | ✓ |

Fig. 8: Enumeration with $n = 7$, in chunks of $2^3$ elements with $4$ batches of $4$ concurrent threads. "Non-local" means that a constant of index greater than 3 is accessed, while "Non-broadcast" means that the 4 threads do not access the same memory location. In conformance with lemma 4, there are $1 + 7 - 2 = 6$ non-broadcast memory accesses.

### 8.1 CPU Enumeration Kernel

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

(a) quadratics, C++ x86 instrinsics

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

(b) quadratics, x86 assembly

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

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

(c) cubics, x86 assembly

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

(d) cubics, C++ x86 instrinsics

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

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

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

### 8.2 GPU Enumeration Kernel

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

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

Our strategy for register use is simple: Pick a suitable bound $u$, and among third differentials $d_{ijk}$ (and first and second differentials $d_i$ and $d_{ij}$), put the most frequently used — i.e., all indices less than $u$

— in registers, and the rest in device memory (which can be read every 8 instructions without choking). We can then control the number of registers used and find the best $u$ empirically.

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

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

(a) `decuda` result from cubin

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

(b) CUDA code for a inner loop fragment

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

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

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

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

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

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

### 8.3 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).

### 8.4 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.

### 8.5 More Test Data and Discussion

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

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

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

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

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

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

*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. 9. As we can see, all except two architectures have inner loop cycle counts that are increasing roughly linearly with the degree. The exceptions are the AMD K10 and NVIDIA G200 architectures, which is in line with fast memory pressure on the NVIDIA GPUs and fact that K10 has the least cache among the CPU architectures.

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

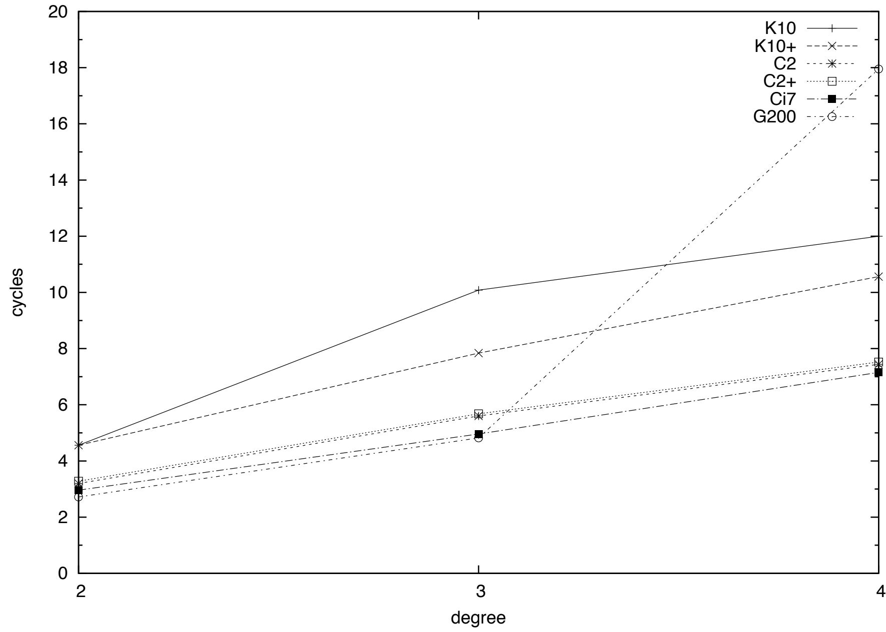


Fig. 9: Cycles per candidate tested for degree 2,3 and 4 polynomials.

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

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

**Notes and Acknowledgements**

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

## References

1. 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 *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004. Previously 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. In P. Gianni, editor, *MEGA 2005 Sardinia (Italy)*, 2005.
4. A. T. Benjamin and J. Quinn. *Proofs that Really Count: The Art of Combinatorial Proof (Dolciani Mathematical Expositions)*. The Mathematical Association of America, August 2003.
5. 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.
6. 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. `http://eprint.iacr.org/2008/480/`.
7. L. Bettale, J.-C. Faugère, and L. Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, 3:177–197, 2009.
8. 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/`.
9. 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/`.
10. B. Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. PhD thesis, Innsbruck, 1965.
11. J. Buchmann, D. Cabarcas, J. Ding, and M. S. E. Mohamed. Flexible partial enlargement to accelerate gröbner basis computation over $_2$. In D. J. Bernstein and T. Lange, editors, *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 69–81. Springer, 2010.
12. 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.
13. N. Courtois, L. Goubin, and J. Patarin. *SFLASH: Primitive specification (second revised version)*, 2002. `https://www.cosic.esat.kuleuven.be/nessie`, Submissions, Sflash, 11 pages.
14. 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`.
15. N. de Bruijn. *Asymptotic methods in analysis. 2nd edition.* Bibliotheca Mathematica. Vol. 4. Groningen: P. Noordhoff Ltd. XII, 200 p. , 1961.
16. J.-C. Faugère. A new efficient algorithm for computing Gröbner bases ($F_4$). *Journal of Pure and Applied Algebra*, 139:61–88, June 1999.
17. 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.
18. J.-C. Faugère and A. Joux. Algebraic cryptanalysis of Hidden Field Equations (HFE) using Gröbner bases. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Dan Boneh, ed., Springer, 2003.

19. 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`.
20. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.
21. J. Patarin. Asymmetric cryptography with a hidden monomial. In *Advances in Cryptology — CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 45–60. Neal Koblitz, ed., Springer, 1996.
22. 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`.
23. J. Patarin, N. Courtois, and L. Goubin. QUARTZ, 128-Bit Long Digital Signatures. In D. Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2001.
24. J. Patarin, L. Goubin, and N. Courtois. Improved algorithms for Isomorphisms of Polynomials. In *Advances in Cryptology — EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 184–200. Kaisa Nyberg, ed., Springer, 1998. Extended Version: `http://www.minrank.org/ip6long.ps`.
25. H. Raddum. Mrhs equation systems. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2007.
26. 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.
27. B.-Y. Yang and J.-M. Chen. Theoretical analysis of XL over small fields. In *ACISP 2004*, volume 3108 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 2004.
28. B.-Y. Yang, J.-M. Chen, and N. Courtois. On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis. In *ICICS 2004*, volume 3269 of *Lecture Notes in Computer Science*, pages 401–413. Springer, Oct. 2004.