Implementing Joux-Vitse's Crossbred Algorithm for Solving $\mathcal{MQ}$ Systems over $\mathbb{F}_2$ on GPUs

Ruben Niederhagen[1], Kai-Chun Ning[2], and Bo-Yin Yang[3]

[1] Fraunhofer SIT, Darmstadt, Germany
ruben@polycephaly.org
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
kaichun.ning@gmail.com
[3] IIS and CITI, Academia Sinica, Taipei, Taiwan
by@crypto.tw

Abstract. The hardness of solving multivariate quadratic ($\mathcal{MQ}$) systems is the underlying problem for multivariate-based schemes in the field of post-quantum cryptography. The concrete, practical hardness of this problem needs to be measured by state-of-the-art algorithms and high-performance implementations. We describe, implement, and evaluate an adaption of the Crossbred algorithm by Joux and Vitse [11] for solving $\mathcal{MQ}$ systems over $\mathbb{F}_2$. Our adapted algorithm is highly parallelizable and is suitable for solving $\mathcal{MQ}$ systems on GPU architectures. Our implementation is able to solve an $\mathcal{MQ}$ system of 134 equations in 67 variables in 98.39 hours using one single commercial Nvidia GTX 980 graphics card, while the original Joux-Vitse algorithm requires 6200 CPU-hours for the same problem size. We used our implementation to solve all the Fukuoka Type-I MQ challenges for $n \in \{55, \ldots, 74\}$. Based on our implementation, we estimate that the expected computation time for solving an $\mathcal{MQ}$ system of 80 equations in 84 variables is about one year using a cluster of 3647 GTX 980 graphics cards. These parameters have been proposed for 80-bit security by, e.g., Sakumoto, Shirai, and Hiwatari in 2011 [15].

Keywords: Post-quantum cryptography, multivariate quadratic systems, parallel implementation, GPU.

1 Introduction

With the advent of quantum computing, an adversary can efficiently break universally adopted public key cryptographic schemes, e.g. RSA and elliptic curve cryptography, with a sufficiently large quantum computer [16,17]. In order to mitigate this imminent threat, cryptographic schemes that are resistant against quantum computers have drawn great attention from academia. These schemes are collectively referred to as post-quantum cryptography (PQC).

One potential candidate for PQC is *multivariate cryptography*. Multivariate cryptography relies on the difficulty of solving a system of m polynomial equations in n variables over a finite field. The complexity of solving a multivariate

Permanent ID of this document: f9066f7294db4f2b5fbdb3e791fed78e.
Date: 2017.11.24.

polynomial system ($\mathcal{MP}$ problem) or a multivariate quadratic system ($\mathcal{MQ}$ problem) where coefficients of the monomials are independently and uniformly distributed (i.e. random) is well-known to be NP-hard. An arbitrary $\mathcal{MP}$ system can be transformed into an equivalent $\mathcal{MQ}$ system by substituting monomials of degree larger than two with new variables and introducing extra equations to the system. Furthermore, a polynomial system over any extension field $\mathbb{F}_{2^n}$ can be reduced into an equivalent system over $\mathbb{F}_2$ using Weil descent.

Since the early 1980s, various asymmetric multivariate encryption schemes (e.g., [14,5,18]) based on Hidden Field Equations (HFE) [10] as well as signature schemes (e.g., [13,12,6]) have been proposed. Besides these asymmetric schemes, some symmetric encryption schemes, e.g., the stream cipher QUAD [1], have been proposed and analyzed [20].

When introducing a trapdoor into an $\mathcal{MQ}$ system for the use in public key cryptography, the resulting system is not truly random and typically exhibits a hidden structure that often can be exploited in its cryptanalysis. However, we do not focus on the cryptanalysis of any particular cryptographic scheme by exploiting some hidden structure. Our goal is to investigate the concrete, practical hardness of the underlying problem of solving random $\mathcal{MQ}$ systems over $\mathbb{F}_2$ by providing an efficient, parallel implementation of the state-of-the-art algorithm.

This paper is structured as follows: In Section 2, we introduce the Crossbred algorithm by Joux and Vitse and our adaption to this algorithm. In Section 3, we describe our implementation of the adapted algorithm for a cluster of GPUs. In Section 4 we describe how to choose the parameters for our implementation, given a specific $\mathcal{MQ}$ system size, and in Section 5, we provide an evaluation of our implementation.

We are going to provide the source code of our implementation online.

2 Joux-Vitse's Crossbred Algorithm

There are several approaches for solving $\mathcal{MP}$ systems, e.g., Faugère's F4 and F5 algorithms [7,8] based on the computation of Gröbner-bases and a family of algorithms based on extended linearization (XL) [19]. For $\mathcal{MQ}$ systems over $\mathbb{F}_2$, brute-force enumeration over the search space is an efficient alternative [3], which was the approach used by the previous record holder [4] of Fukuoka MQ Type-I and Type-IV challenges[4]. The record on Type-I challenges is now held by an implementation of the Crossbred algorithm by Joux and Vitse [11].

The basic idea of the XL algorithm is to extend the original $\mathcal{MQ}$ system by multiplying it by all monomials up to a certain degree $D-2$ and by treating monomials in the resulting degree-D system as linear variables. Solving this linear system gives a solution for the original $\mathcal{MQ}$ system with high probability, if D is chosen large enough.

Brute-force enumeration works by simply enumerating all possible assignments of the variables and by checking the correctness of each assignment with

[4] https://www.mqchallenge.org/

the original $\mathcal{MQ}$ system. This approach can be optimized by enumerating the possible assignments in Gray-code order such that there is only one single variable with a different assignment in each enumeration step. This allows to compute the new evaluation result based on the change in regard to the previous evaluation result, which requires storage and recursive update of partial derivatives up to the total degree of the system [4].

2.1 The Crossbred Algorithm

The basic idea of Joux and Vitse's Crossbred algorithm is to extend the original $\mathcal{MQ}$ system to a system with a degree D lower than the degree required for XL and to derive a sub-system the has at most degree-d in the first k variables. This sub-system is then solved by iterating over the remaining $n-k$ variables and solving the resulting degree-d system in k variables in each iteration. For $d = 1$, this requires to only solve a *linear* system in k variables for each assignment of $n-k$ variables.

For example, by fixing the last two variables x_3 and x_4, the sub-system

$$\mathcal{S} = \begin{cases} x_1x_4 + x_2x_3 + x_1 + x_3 + x_4 = 0 \\ x_1x_3 + x_3x_4 + x_2 + 1 = 0 \\ x_2x_3 + x_2x_4 + x_3x_4 + x_1 + x_4 = 0 \end{cases}$$

becomes a linear system in x_1 and x_2. Clearly, the resulting linear system can be directly solved with Gaussian elimination, with which solutions to the system $\mathcal{S}$ can be derived efficiently.

For a monomial $x^\alpha = x_1^{\alpha_1}x_2^{\alpha_2}\ldots x_k^{\alpha_k}x_{k+1}^{\alpha_{k+1}}\ldots x_n^{\alpha_n}$, the total degree of the first k variables is denoted as $\deg_k x^\alpha = \sum_{i=1}^{k}\alpha_i$. Given an $\mathcal{MQ}$ system $\mathcal{F}$, the Crossbred algorithm first computes a degree-D Macaulay matrix with respect to a monomial order $>_{\deg_k}$where monomials are sorted according to $\deg_k$ in descending order. Subsequently the algorithm extracts at least k equations where the monomials of $\deg_k$ larger than one (which are non-linear in $x_1, \ldots, x_k$) are eliminated and only keeps monomials of $\deg_k \leq 1$ (which are linear in $x_1, \ldots, x_k$). These equations give a sub-system that can be transformed into a linear system in the first k variables by fixing the remaining $n-k$ variables. After one such sub-system $\mathcal{S}$ is obtained, Crossbred performs exhaustive search by fixing the last $n-k$ variables and testing whether or not the resulting linear system $\mathcal{S}'$ is solvable. If so, solutions to $\mathcal{S}'$ are checked with the original $\mathcal{MQ}$ system $\mathcal{F}$. The algorithm terminates if a solution is found, otherwise it fixes $n-k$ variables in $\mathcal{S}$ with another set of values and continues the exhaustive search procedure.

To obtain a linear system $\mathcal{S}'$ from the extracted sub-system $\mathcal{S}$, the Crossbred algorithm uses a recursive algorithm called FastEvaluate to fix $n-k$ variables in $\mathcal{S}$. The basic idea of this algorithm is to split each polynomial into two groups of monomials. An arbitrary polynomial p can be written as $p = p_0 + x_ip_1$, where x_ip_1 are monomials that involve a specific variable x_i while p_0 are monomials that do not. It is clear from this form that p_0 is exactly the result of fixing

Algorithm 1 The Original Crossbred Algorithm

```
procedure CROSSBRED
    Input:
        an $\mathcal{MQ}$ system of $m$ equations in $n$ variables $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$
        Macaulay degree: $D$
        number of variables to keep: $k$
        number of variables to fix during $\mathcal{MQ}$ external hybridization: $p$

    for each $(x_{n-p+1}, \ldots, x_n)$ in $\{0,1\}^p$ do
        1. Fix the last $p$ variables in $\mathcal{F}$ to obtain an $\mathcal{MQ}$ system $\mathcal{F}'$.
        2. Compute the degree-$D$ Macaulay matrix $\mathrm{Mac}_D^k$
           where monomials are sorted by $\deg_k$ based on $\mathcal{F}'$.
        3. Extract $r$ linearly independent equations $\mathcal{S} = \{s_1, s_2, \ldots, s_r\}$ from $\mathrm{Mac}_D^k$
           where monomials of $\deg_k > 1$ have been eliminated.

        Call FastEvaluate($\mathcal{S}, k, n-p$) and
        for each output linear system $\mathcal{S}'$ do
            4. Test if $\mathcal{S}'$ is solvable. If so, extract solutions and verify them with $\mathcal{F}$.
            5. Continue if no solution is found.
               Otherwise output the solution and terminate.
        end for
    end for
end procedure
```

$x_i = 0$ in p and $p_0 + p_1$ is the result of fixing $x_i = 1$ in p. This idea can be applied recursively to fix $n - k$ variables.

One can further fix some variables in the original $\mathcal{MQ}$ system before computing Macaulay matrices, which is referred to as *external hybridation* by the authors [11]; here, we use the term *external hybridization*. The authors of the Crossbred algorithm consider external hybridization merely as a method to distribute the workload between computers and do not expect it to be asymptotically useful [11]. Nevertheless, this technique can be helpful to increase the number of variables that can be kept for linearization, which reduces the runtime of the algorithm significantly.

2.2 Adapting the Crossbred Algorithm for Parallel Implementation

The FastEvaluate algorithm proposed by Joux and Vitse has the disadvantage that computing the subsets p_0 and p_1 on higher levels of the recursion is relatively expensive. We propose to use Gray-code enumeration [3] instead of FastEvaluate, which requires only $\mathcal{O}(2^{n-k} \cdot D \cdot k)$ machine instructions on the cost of $\mathcal{O}(\sum_{i=0}^{D} \binom{n-k}{i} \cdot k)$ memory.

Gray-code enumeration was proposed to efficiently evaluate a polynomial function $f(x_1, x_2, \ldots, x_n)$ in all points $(x_1, x_2, \ldots, x_n) \in \mathbb{F}_2^n$. To obtain the result of evaluating f on the next point $\boldsymbol{a} \in \mathbb{F}_2^n$ from the current result $f(\boldsymbol{a'})$ where only the i^{th} coordinates of $\boldsymbol{a}$ and $\boldsymbol{a'}$ differ, $\mathcal{O}(1)$ machine instructions are executed to combine $f(\boldsymbol{a'})$ with the result of evaluating the first order partial derivative $\frac{\partial f}{\partial x_i}$

on $\boldsymbol{a'}$ [3]. In particular, $f(\boldsymbol{a}) = f(\boldsymbol{a'}) + \frac{\partial f}{\partial x_i}(\boldsymbol{a'})$. This technique can be applied recursively to evaluate $\frac{\partial f}{\partial x_i}(\boldsymbol{a'})$ and its higher order partial derivatives until the partial derivative reduces to a constant. Therefore, if f is of degree D, $\mathcal{O}(D)$ operations are required to compute $f(\boldsymbol{a})$.

The same technique can also be applied to evaluate a function f whose output is a *linear function* in k variables instead of a constant over $\mathbb{F}_2$ by simply splitting the polynomial into a sum of $k+1$ sub-polynomials, one for each of the k variables and one for a constant term. For example, the polynomial

$$\begin{aligned} f = {} & x_1x_4x_5x_6 + x_1x_4x_5x_7 + x_4x_5x_6x_7 + x_1x_4x_5 + x_3x_4x_7 + x_3x_5 \\ & + x_2x_4x_6 + x_4x_6x_7 + x_1x_4 + x_1x_5 + x_5x_7 + x_6x_7 + x_1 + x_2 + x_4 + 1 \end{aligned}$$

which is linear in x_1, x_2, x_3 can be split into the 4 polynomials

$$\begin{aligned} f_1 &= x_1(x_4x_5x_6 + x_4x_5x_7 + x_4x_5 + x_4 + x_5 + 1), \\ f_2 &= x_2(x_4x_6 + 1), \\ f_3 &= x_3(x_4x_7 + x_5), \\ f_4 &= x_4x_5x_6x_7 + x_4x_6x_7 + x_5x_7 + x_6x_7 + x_4 + 1, \end{aligned}$$

such that $f = f_1 + f_2 + f_3 + f_4$. Now, f can be evaluated by applying Gray-code enumeration to $f_1, f_2, f_3, \text{and} f_4$ individually.

Since the result of evaluating f or any of its partial derivatives on a point $\boldsymbol{a} \in \mathbb{F}_2^4$ is a linear function that can be represented by four $\mathbb{F}_2$ elements (three variables and the constant term) and the last order partial derivatives reduce to constants, evaluating $f(\boldsymbol{a})$ takes at most $3 \cdot (3+1)+1$ `xor`-operations and another $4 \cdot 2$ operations for computing the indices of the coordinates that changed during enumeration. In general, for a polynomial function f of degree D whose output is a linear function in k variables, evaluating f requires $\mathcal{O}(D \cdot k)$ operations.

Since a machine instruction operates on machine words, which for example have size 64 for 64-bit architectures or 32 on GPUs, multiple polynomials can be evaluated with Gray-code enumeration simultaneously. Therefore, the algorithm described above can be applied to fix $n - k$ variables in an extracted sub-system $\mathcal{S}$ of m equations in n variables using $\mathcal{O}(D \cdot k)$ instructions, as long as m is not larger than the machine word size.

Gray-code enumeration can be easily parallelized: To run the enumeration with 2^t threads in parallel, first fix t variables in the sub-system $\mathcal{S}$ with all t-tuples in $\{0, 1\}^t$ to create 2^t smaller sub-systems in $n - t$ variables. With this approach, although the sub-systems are distinct from each other, their last order partial derivatives with respect to the $n - t - k$ variables that must be fixed are identical.

3 Implementation

Our target platform for the implementation is a hybrid cluster of workstations equipped with GPUs. Therefore, we have two processor architectures to our disposal: AMD64 CPUs and Nvidia GPUs (Kepler and Maxwell microarchitecture).

The Gray-code enumeration part of the Crossbred algorithm is particularly easy to parallelize and therefore suitable for GPU deployment. Thus, we use the CPUs to generate and process the Macaulay matrix and the GPUs for Gray-code enumeration and linear system solving.

3.1 Macaulay Matrix Computations

The first step in Joux-Vitse's Crossbred algorithm is to extend the original $\mathcal{MQ}$ system to a Macaulay matrix of degree D. (Our implementation works for $D = 3$ and $D = 4$.) The columns are ordered such that the monomials with $\deg_k > 1$ are in the front. Then, the left kernel of the Macaulay matrix is computed. Finally, a sub-system linear in $x_1, \ldots, x_k$ is extracted for Gray-code enumeration.

Since the Macaulay matrix is very sparse, a sparse-system solver like the block Lanczos algorithm or the block Wiedemann algorithm could be used. However, the Macaulay matrix exhibits a special structure: Since the Macaulay matrix is generated from the original system by multiplying the polynomials with all monomials up to a certain degree, the resulting matrix is close to being diagonal. Therefore, we decided to exploit this special structure in a specifically adapted implementation of Gaussian elimination.

The first step is to compute the reduced echelon form of the original input system. This is a very small computation and requires a negligible amount of time. Then, we compute the Macaulay matrix $\mathcal{M}$ such that the columns are in the required order. We store $\mathcal{M}$ in a sparse representation. Then we search for rows in the Macaulay matrix that have an increasing number of leading zeros and swap them into place: Find a row that has no leading zeros and swap it to the top, find a row that has one leading zero and swap it to the second row, and so on. Due to the structure of the Macaulay matrix, usually about two thirds of the rows of the upper-triangular form of $\mathcal{M}$ can be obtained just by swapping in suitable rows. Now, only the remaining one third of the upper-triangular form of $\mathcal{M}$ needs to be computed. Observe that up to this point, $\mathcal{M}$ can be stored in a sparse format and no costly row reductions needed to be performed.

In order to compute the remaining rows of the upper-triangular form of $\mathcal{M}$, one must perform row reduction. Therefore, we switch over to a dense representation by first performing row reduction on rows that have not found their final position during row-swapping with those that did. In this manner, we drop those rows and columns that already have been pivoted by row-swapping and obtain a dense, reduced matrix $\mathcal{RM}$. On this matrix, we perform classical Gaussian elimination in order to compute the desired sub-system that is linear in $x_1, \ldots, x_k$.

After $\mathcal{RM}$ is computed, it can be copied to the GPU if the off-chip memory if large enough to accommodate it. Subsequently a sub-system can be extracted with Gaussian elimination on the GPU and copied back to the system main memory. On the other hand, if the size of $\mathcal{RM}$ is too large or if the overall workload is pipelined between CPU and GPU, Gaussian elimination is simply performed on the CPU. We parallelized the CPU implementation using the *POSIX Thread API* to distribute the workload over all CPU cores. We observed during experiments that our GPU implementation on a Nvidia GTX 980 graphics card outperforms our CPU version on a AMD FX-8350 4GHz processor by a factor of 9 in most cases.

Since the size of registers on a GPU is 32 bits and both Gray-code enumeration and linear system solving require the input system to be stored in column-wise format, only 32 linearly independent equations need to be extracted from the reduced Macaulay matrix $\mathcal{RM}$ for the sub-system $\mathcal{S}$.

3.2 Fixing Variables in the Sub-system

We implemented the Gray-code enumeration algorithm for fixing $n-k$ variables in the degree-D sub-system $\mathcal{S}$ to enumerate linear systems in k variables for the GPU architecture. The data structures used by Gray-code enumeration are allocated from the off-chip global memory. We simply distribute the workload over 2^t threads by fixing t variables in $\mathcal{S}$ to obtain individual and independent smaller sub-systems $\mathcal{S}_i, 1 \leq i \leq 2^t$ for each thread. Since the last partial derivatives are constants and remain the same for all 2^t smaller sub-systems as noted in Section 2.2, they can be shared by all threads. Since they are constant, we store them in read-only constant memory.

The GPU threads in a warp begin enumeration with the same starting point and consequently they will access partial derivatives in the same order in each iteration. Therefore, the data structures for one warp can be interleaved to obtain optimal memory throughput. In addition, because of the cyclic nature of Gray-code enumeration, the last-level derivatives stored in constant memory are likely to be cached in the constant memory cache. Since the data of the 32 equations in the sub-system is stored in column-wise format, in total $\binom{n-k-t}{D}$ 32-bit integers are required for storing the constant last-level derivatives.

As described in Section 2.2, the evaluation of a k-linear polynomial is split into the evaluation of $k+1$ polynomials. Therefore, we store the data for the non-constant partial derivatives for the 32 threads in one warp in basic units of $32(k+1)$ words interleaved in memory. Since for each of the $k+1$ polynomials $n-k-t$ variables have to be fixed during enumeration, storing results of evaluating the non-constant partial derivatives of $\mathcal{S}_i$ requires $\sum_{j=1}^{D-1}\binom{n-k-t}{j}$ such basic memory units for one warp. Together with the result of evaluating $\mathcal{S}_i$ at the current point (which requires one basic unit as well) a warp requires $\sum_{j=0}^{D-1}\binom{n-k-t}{j}$ basic units. Therefore, in total Gray-code enumeration requires $(2^{t-5} \cdot \sum_{j=0}^{D-1}\binom{n-k-t}{j}) \cdot 32 \cdot (k+1)$ words of size 32-bit in global memory and $\binom{n-k-t}{D}$ words of size 32-bit in constant memory.

3.3 Testing the Solvability of a Linear System

After a linear system has been computed during an iteration step of Gray-code enumeration, the system needs to be checked for solvability. The straight-forward approach for testing the solvability of a linear system is to simply solve it with Gauss-Jordan elimination.

In the standard Gauss-Jordan elimination algorithm, once a pivot row for the i^{th} pivot element is located, it is moved to its final position by swapping with the i^{th} row. However, we are storing the linear system in column order, so row swapping is expensive. Therefore, we avoid row-swapping by maintaining a mask that tracks which rows are in their final position.

After as many rows as the number of variables k in the linear system $\mathcal{S}_i$ have been marked as final, the algorithms stops. The remaining unmarked rows are redundant equations and their first k coefficients which represent the variables $x_1, x_2, \ldots, x_k$ are guaranteed to be zero. Therefore, testing the solvability of $\mathcal{S}_i$ is as simple as checking if the constant term of any of the redundant equations is non-zero.

Clearly, if the system is solvable, a solution can be extracted from the last column based on the first k columns. In particular, the position of 1 in the i^{th} column points to the value for x_i in the last column. Note that before extracting a solution, one has to test whether or not the system is underdetermined. To achieve this, one can simply verify that none of the first k columns is completely zero since one such column implies a missing pivot element. This verification can be done simultaneously while extracting a solution and does not require extra computation.

We avoid storing data for linear system solving in global memory by storing the entire data in registers. In order to make sure that the compiler maps data to registers, we do not use an array data structure to store the data. Instead, we use a Python script to generate unrolled code with distinct variables for all data. However, the consequence of generating CUDA code at compilation time is that the program has to be re-compiled for each choice of k. This takes roughly 6 seconds on an AMD FX-8350 4GHz processor, which is negligible.

3.4 Probability of False Positives

There are three possible outcomes of solving the linear system: there can be no, one, or more than one solution. The expected outcome is that there is no solution in which case we proceed to the next Gray-code iteration step. Ideally, we find one single solution only once — which then is also a solution for the original quadratic system. However, there is a small probability that a solution for the subsystem $\mathcal{S}$ is not a solution for the original system, i.e., it is a false positive. Finally, there is also a chance for finding more than one solution which requires further processing.

Suppose we have a random linear system of m equations in $\mathbb{F}_2$ of n variables. We would like to estimate the probability that this system has at least one solution. Let A be the augmented matrix of this system $(m \times (n+1))$.

Assume that during Gaussian elimination the upper-left corner is a pivot. This means that there is at least one 1 in the first column ($2^m - 1$ possibilities). The first row with a leading 1 gets swapped to the top, and the rest of the first column is eliminated. There are n other entries in the first row (2^n possibilities). The remaining $(m-1) \times n$ sub-matrix remain uniformly random.

We can continue this reasoning conclude that if the Gaussian elimination have pivots in columns $a_1 < a_2 < \cdots < a_\ell \leq n+1$ exactly

$$2^{\sum_{j=1}^{\ell}(n+1-a_j)} \left(\prod_{j=1}^{\ell} (2^{m+1-j} - 1) \right)$$

times. Thus, when $m > n$, we can tell that the largest block of consistent systems have pivots $a_1 = 1$, $a_2 = 2, \ldots, a_n = n$, and these number

$$2^{n(n+1)/2} \left(\prod_{j=1}^{n} (2^{m+1-j} - 1) \right) < 2^{n(n+1)/2} \left(\prod_{j=1}^{n} (2^{m+1-j}) \right) = 2^{n(m+1)}.$$

There are $2^{m(n+1)}$ possible matrices so probability of a full-rank consistent system is bounded by $2^{-(m-n)}$. More precisely, for large $m = n$, the probability of full-rank consistency is $\frac{1}{2} \cdot \frac{3}{4} \cdot \frac{7}{8} \cdots \left(1 - \frac{1}{2^n}\right) \gtrsim p_0 = \prod_{j=1}^{\infty} \left(1 - \frac{1}{2^j}\right) \approx 0.288788$.

In general a full-rank consistent systems occurs with probability roughly

$$2^{-(m-n)} \prod_{j=1}^{n} (1 - 2^{m-n+j}) \gtrsim p_{m-n} := p_0 \cdot 2^{-(m-n)} / \left(\prod_{j=1}^{m-n} (1 - 2^{-j}) \right).$$

The second largest block of systems (missing a pivot in column n) is $\frac{1}{2(2^{(m+1-n)}-1)}$ times less likely. Systems missing a pivot in column $(n-j)$ are a further factor of $1/2^{j-1}$ less likely. Thus, probability of consistent systems with $(n-1)$ pivots is $\approx \frac{p_{m-n}}{2(2^{(m+1-n)}-1)} \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^{n-1}}\right) \approx \frac{p_{m-n}}{(2^{(m+1-n)}-1)}$.

The largest block missing two pivots (in columns n and $n-1$) is a factor $\frac{1}{2^3(2^{(m+1-n)}-1)(2^{(m+2-n)}-1)}$ smaller than full-rank. Each time we move the first missing pivot left there is a factor of $1/2$. Each time we move the second (rightmost) missing pivot left there is a factor of $1/4$. Summing over $2^{-i}4^{-j}$ gets a factor of $8/3$, so we end up having probability of missing 2 pivots close to

$$\approx \frac{p_{m-n}}{3(2^{(m+1-n)}-1)(2^{(m+2-n)}-1)}$$

Continuing this argument, we note that the largest term missing k pivots is smaller by a factor of $2^{k(k+1)} \left(\prod_{j=1}^{k} (2^{m-n+k} - 1) \right)$. Summing over all matrices missing k pivots, we get a factor of $\lesssim \frac{2}{1}\frac{4}{3} \cdots \frac{2^k}{2^k-1}$. So the totality of all matrices missing k pivots is $\approx p_{m-n} / \left(\prod_{j=1}^{k} (2^{m-n+k} - 1) \right) \left(\prod_{j=1}^{k} (2^k - 1) \right)$.

Finally, we have the "too long, didn't read" answer: The probability of a set of consistent equations for large m and n approaches

$$\left(\frac{p_0}{2^{m-n} \left(\prod_{j=1}^{m-n} (1 - 2^{-j}) \right)} \right) \left[\sum_{k=0}^{\infty} \left(\frac{1}{\left(\prod_{j=1}^{k} (2^{m-n+k} - 1) \right) \left(\prod_{j=1}^{k} (2^k - 1) \right)} \right) \right].$$

If we only take two terms, it becomes roughly

$$\left(\frac{p_0}{2^{m-n}\left(\prod_{j=1}^{m-n+1}(1-2^{-j})\right)}\right) \to 2^{-(m-n)} \text{ for large } m-n.$$

This is consistent with intuition. For example, to have no more than 1 consistent system in 1000, we need $m - n \geq 10$. For two further examples, we note that $p_1 = p_0 = 0.288788$. The probability when $m - n = 1$ of a set of consistent equations is approximately

$$p_1 \cdot \left[\sum_{k=0}^{\infty}\left(\frac{1}{\left(\prod_{j=1}^{k}(2^{k+1}-1)\right)\left(\prod_{j=1}^{k}(2^{k}-1)\right)}\right)\right] = 0.389678.$$

When $m = n$, the probability of a set of consistent equations is approximately

$$p_0 \sum_{k=0}^{\infty} \frac{1}{\left(\prod_{j=1}^{k}(2^k-1)\right)^2} = 0.610322,$$

which is exactly the complement of the previous result!

3.5 Verification of Solution Candidates

When a single solution candidate is found, it needs to be verified with the original $\mathcal{MQ}$ system. Ideally, one would copy the solution candidate from the GPU off-chip memory back to the main memory and verify it on the CPU immediately. In practice, this is not efficient because checking each solution candidate right away on the CPU interrupts the workflow of the GPU. Therefore, an alternative approach is to store all solution candidates in a buffer and only copy them back to the main memory after the GPU kernel finishes. One caveat of this approach is that a sufficiently large buffer must be allocated on the off-chip memory, which may have little capacity left after allocating memory blocks for the data structures used in Gray-code enumeration. If the number of solution candidates is larger than the size of the buffer, some candidates must be dropped.

To avoid this pitfall, we copy some polynomials from the original $\mathcal{MQ}$ system to the GPU which serve as a filter. Evaluating a random polynomial over $\mathbb{F}_2$ at a random input results in zero with probability 0.5. Therefore, using i polynomials reduces the number of candidates by a factor of 2^{-i} (for $i \ll n$). Only solution candidates that pass the filter will then be verified with the rest of the equations in the original $\mathcal{MQ}$ system by the CPU.

We are using 32 polynomials that are stored in column-wise format. In this manner, to apply the filter a thread needs to evaluate $\frac{n(n-1)}{2} + n + 1$ monomials in the polynomials with the solution candidate. Therefore, this takes at most $\mathcal{O}(n^2)$ machine instructions. However, filtering only needs to be applied very rarely and its execution time was completely hidden during our experiments.

If more than one solution is found, more effort is required in order not to miss the solution. The probability of having more than one solution is very small for well chosen implementation parameters (see Section 3.4). Therefore, our implementation simply reports when it encounters this case and moves on to the next iteration step. During all our experiments, this case never occurred.

3.6 Pipelining

When external hybridization is applied, i.e, p variables are fixed in the original $\mathcal{MQ}$ system, one has to extract a sub-system and subsequently perform Gray-code enumeration at most 2^p times. Since we perform Gray-code enumeration on the GPU, which operates independently from the CPU, we are able pipeline the two stages. In other words, while performing Gray-code enumeration on the GPU, a sub-system for the next Gray-code enumeration can be computed in parallel on the CPU. In this manner, as long as extracting a sub-system takes at most as much time as Gray-code enumeration, which can be controlled by the choice of p, only the runtime of extracting the first sub-system will manifest.

4 Choice of Parameters

There are several parameters to choose before the Crossbred algorithm can be executed on a CPU/GPU cluster. First, we need to know how many variables k we can keep for linearization. This depends on the Macaulay degree D and the number of variables p fixed in external hybridization. Finally we need to decide how many variables to fix before deploying the workload to the GPUs and how many GPU threads to launch in parallel.

4.1 Number of Variables to Keep

We want to set the parameter k as high as possible in order to reduce the search space for Gray-code enumeration: For every extra variable that can be kept, the search space is halved. As described in the original Crossbred algorithm [11], the maximum of k depends on the Macaulay degree D as well as the number of variables n and the number of equations m in the original $\mathcal{MQ}$ system. The number of linearly independent equations that can be extracted from a Macaulay matrix can be computed as the difference of the number of independent rows $N_{\text{indep\_row}}$ in the Macaulay matrix and the number of monomials N_{nl} which are non-linear in $x_1, \dots, x_k$. This number must be no less than k; otherwise, there will not be enough equations in the sub-system to obtain a unique solution. The maximum value of k for $\mathcal{MQ}$ systems with $n = m$ and $m = 2n$, based on Macaulay degree $D = 3$ and 4, can be computed as

$$N_{\text{indep\_row}} = \begin{cases} m \cdot (n+1), \text{ when } D = 3, \\ m \cdot (\binom{n}{2} + n + 1) - (\binom{m}{2} + m), \text{ when } D = 4, \end{cases}$$

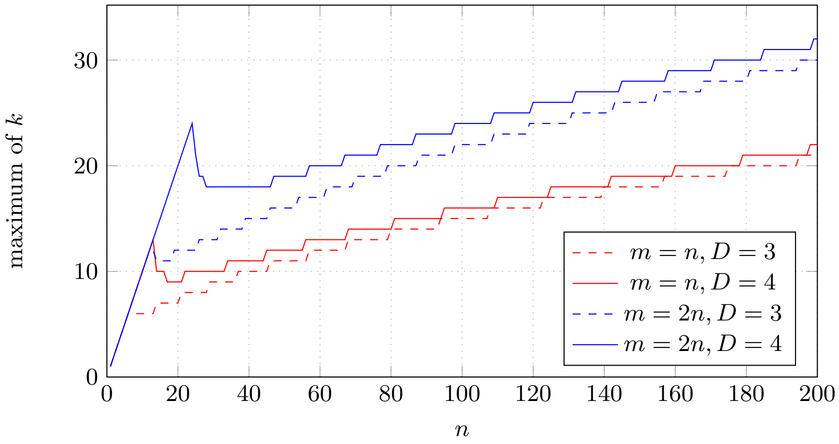

Fig. 1: Maximum number of variables k that can be kept depending on n and m.

$$N_{\text{nl}} = \sum_{i=2}^{D} \sum_{j=2}^{i} \binom{k}{j} \cdot \binom{n-k}{i-j},$$

$$N_{\text{indep\_row}} - N_{\text{nl}} \overset{!}{\geq} k.$$

Figure 1 shows a graph for the number of variables we can keep in relation to the system size for $n < 200$. Clearly, with degree-4 Macaulay matrices one can keep more variables than with degree-3 Macaulay matrices for large enough n. However, for some determined systems, e.g. $n = 140$, using a degree-4 Macaulay matrix does not allow us to keep more variables than when using a degree-3 matrix. In addition, the gap between the two curves for overdetermined systems becomes narrower as n grows. Therefore, similar to determined systems, the effectiveness of degree-4 matrices is expected to become marginal at which point degree-5 Macaulay matrices are required if one wishes to keep considerably more variables than when using degree-3 matrices.

Note that k grows linearly in the beginning of each curve, where the degree of regularity of the $\mathcal{MQ}$ system is smaller than or equal to the Macaulay degree. In this case, a Gröbner basis can be extracted directly from the Macaulay matrix, which immediately yields a solution to the system.

4.2 Macaulay Degree

As discussed in [11], since the Macaulay matrix is used to induce cancellation of the monomials where any of the variables $x_1, x_2, \ldots, x_k$ has a degree larger than one, the degree of the Macaulay matrix must be no less than the degree of regularity of a random system of m equations in k variables. In addition, the Macaulay degree is a key factor that determines the maximum value of k. One should therefore choose a Macaulay degree that is larger than the degree of regularity requirement and that can provide a sufficient number of linearly independent equations for the intended value of k.

One caveat of choosing the Macaulay degree is that the memory requirement must be smaller than the available system memory. Since both the number of rows and columns of a Macaulay matrix grow considerably when the degree increases, one might have to choose a smaller Macaulay degree and subsequently a smaller k in case the available memory is insufficient.

Our implementation supports both degree-3 and degree-4 Macaulay matrices. Degree-3 Macaulay matrices are useful for small toy examples, while degree-4 Macaulay matrices are sufficient for the largest problem sizes that we target.

4.3 Number of Variables to Fix during External Hybridization

Section 4.1 gives the formula for computing the maximum value of k for a given system. Since the parameter n in the formula is the number of variables in the $\mathcal{MQ}$ system, one can achieve a higher k by fixing some p variables with external hybridization. In this manner, the number of variables in the system drops by p but the number of equations remains the same. Therefore, the number of variables that can be kept may be higher.

For example, an $\mathcal{MQ}$ system of 148 equations in 74 variables allows to keep $k = 21$ variables with a degree-4 Macaulay matrix. By fixing $p = 4$ variables, it becomes a system in 70 variables, which allows to keep one more variable, i.e., $k = 22$. In this manner, the search space of Gray-code enumeration is split into $2^4 \times 2^{74-4-22}$ instead of $1 \times 2^{74-21}$, which reduces the total number of iterations for Gray-code enumeration by half. On the other hand, 2^p sub-systems of Macaulay matrices need to be computed — so there is a limit on the effectiveness of applying external hybridization.

4.4 Number of Variables to Fix before Exhaustive Search

In addition to fixing variables by external hybridization, one can further fix some variables in the extracted sub-system *before* entering the exhaustive search stage. By fixing b variables the sub-system beforehand, one can divide the workload evenly into 2^b smaller sub-systems which require less resources for applying exhaustive search. Clearly, since the main purpose of fixing these b variables in the sub-system is to fine-tune the resource requirement, the choice of b should be adjusted based on the hardware architecture and the remaining parameters.

4.5 Number of GPU Threads

Typically, more threads than available cores are launched on a GPU in order to hide memory and instruction latencies. Therefore, there should be a certain threshold for the number of threads after which increasing the number of threads on the GPU does not have an influence on the performance anymore. To find this threshold, we performed a series of experiments by running our GPU kernel

Number of Threads	Memory per Thread	Total Memory	Constant Memory	Search Space per Thread	Runtime (seconds)
2^{9}	15.41 kB	7.70 MB	5.32 kB	2^{21}	20.97
2^{10}	14.01 kB	14.01 MB	4.56 kB	2^{20}	11.93
2^{11}	12.68 kB	25.37 MB	3.87 kB	2^{19}	7.16
2^{12}	11.42 kB	45.69 MB	3.26 kB	2^{18}	4.89
2^{13}	10.22 kB	81.81 MB	2.72 kB	2^{17}	5.15
2^{14}	9.10 kB	145.56 MB	2240 kB	2^{16}	5.15
2^{15}	8.04 kB	257.12 MB	1820 kB	2^{15}	5.04
2^{16}	7.04 kB	450.50 MB	1456 kB	2^{14}	4.95
2^{17}	6.11 kB	782.00 MB	1144 kB	2^{13}	4.94
2^{18}	5.25 kB	1343.00 MB	880 kB	2^{12}	4.79
2^{19}	4.45 kB	2278.00 MB	0.66 kB	2^{11}	4.86
2^{20}	3.72 kB	3808.00 MB	0.48 kB	2^{10}	4.86

Table 1: Effect of changing the number of GPU threads.

on a randomly generated $\mathcal{MQ}$ system of 92 equations in 46 variables with different numbers of 2^t threads. We performed the experiments on a Nvidia Quadro M1000M GPU using the following settings:

- GPU: Nvidia Quadro M1000M, 4GB off-chip memory, 512 CUDA cores
- Macaulay degree: $D = 3$
- external hybridization: $p = 0$
- Number of variables to fix before enumeration: $b = 0$
- Number of variables to keep: $k = 16$

The results are given in Table 1. As expected, the runtime basically remains constant for $t \geq 12$. For $t < 12$, the degree of parallelism is not sufficient and the latencies manifest.

When $t = 9$, there are $2^9 = 512$ GPU threads deployed, which is exactly the number of CUDA cores available on this particular GPU. In this case, the workload is evenly distributed to all the CUDA cores. Nevertheless, the degree of parallelism is far from enough because executing one single thread per CUDA core is not enough to hide latencies. For example, when the thread loads data from the global memory, which requires hundreds of cycles to access, there is no other thread that can take over the execution resources. Therefore, the CUDA core has no choice but to stall.

Starting from $t = 10$, there are several threads per CUDA core and some latencies can be hidden. The performance gradually improves until $t = 12$, where the degree of parallelism reaches a point where deploying more threads does not improve the ability of the GPU to hide latencies anymore. Therefore, for these experimental settings the threshold where the optimal performance of our implementation can be achieved is 2^{12}.

Note that as explained in Section 3.2, for each doubling in the number of GPU threads the amount of global memory required for a single GPU thread reduces slightly but the total amount of memory required for the GPU kernel increases nearly twofold. However, since the last-level derivatives stored in constant memory are shared by all threads, constant memory requirement decreases as t increases.

5 Evaluation

We evaluated the performance of our implementation on the *Saber clusters* [2]. Saber is located at Eindhoven University of Technology and Saber2 at University of Illinois at Chicago. The clusters consist of mostly homogeneous workstations. Out of all the nodes in these two clusters, we used 27 cluster nodes, each equipped with two Nvidia graphics cards. Twelve out of those 27 nodes have two GTX 780 graphics cards while the remaining 15 nodes have two GTX 980 cards. Each node has 32GB RAM and one AMD FX-8350 4GHz processor, which has four physical CPU *modules* (similar to a physical core in an Intel CPU) shared by eight logical threads (similar to Intel's hyper threading), 16KB L1 data cache per thread, 2MB L2 cache per module, and 8MB L3 cache shared by the whole CPU. We used CUDA version 7.5 and compiled our implementation with the back-end compiler bundled with CUDA, which is GCC version 4.8.

5.1 Overdetermined Systems — Fukuoka $\mathcal{MQ}$ Challenge

We solved some of the *Fukuoka* $\mathcal{MQ}$ *challenges* using our implementation. These challenges were created in 2015 in order to help determining appropriate parameters for public-key cryptographic schemes based on $\mathcal{MQ}$ systems. In particular, we chose Type-I challenges generated with seed 4 because they consist of $\mathcal{MQ}$ systems in n variables and $m = 2n$ equations over $\mathbb{F}_2$.

The experimental results of solving Type-I challenges for $n \in \{55, \dots, 67\}$ using *one single* GTX 980 graphics card are given in Table 2. The workflow of the algorithm, i.e., how the search space is split and enumerated, is listed in the 3rd column of the table. For parameters $p > 0$ and $b > 0$, external hybridization and Gray-code enumeration might need to be repeated at most 2^p and 2^b times respectively. The numbers inside the parentheses in the 4th and 5th column specify how many repetitions were required during the experiments. For all these small experiments we used the GPU instead of the CPU to extract sub-systems except for the last two experiments marked with an asterisk, because the reduced Macaulay matrix was too large to fit into the 4GB off-chip memory of the GTX 980 graphics card. As shown in Table 2, solving an $\mathcal{MQ}$ system of 134 equations in 67 variables requires at most 354231.11 seconds which equals to 98.39 hours on a single GPU, including the computation time of extracting sub-systems.

For larger Type-I challenges with $n \in \{68, \dots, 74\}$ we used 27 nodes in the Saber and Saber2 clusters by distributing the 2^p smaller $\mathcal{MQ}$ systems obtained from external hybridization evenly over the nodes. The results are given in Table 3, which basically has the same format and notation as Table 2. In these larger experiments, sub-systems were extracted from degree-4 Macaulay matrices with the CPU because the GPU off-chip memory cannot accommodate the size of the reduced Macaulay matrices. However, these parameters allowed us to pipeline the extraction of sub-systems on the CPU and the exhaustive search stage on the GPU. Therefore, the computation time of the former can be completely hidden

n	Parameters (D, p, k, b, t)	Search Space $2^p \times 2^b \times 2^{n-p-k-b}$	Extracting Sub-systems (seconds)	Exhaustive Search (seconds)	Total Runtime (seconds)	Maximum Runtime (seconds)
55	(4, 0, 19, 0, 14)	$1 \times 1 \times 2^{36}$	387.80	318.25	706.20	706.20
56	(4, 1, 20, 0, 14)	$2^1 \times 1 \times 2^{35}$	491.60 (1)	169.94 (1)	658.67	1317.34
57	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{37}$	606.75	650.90	1258.73	1258.73
58	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{38}$	670.26	1311.97	1982.74	1982.74
59	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{39}$	741.62	2619.00	3361.77	3361.77
60	(4, 0, 20, 0, 14)	$1 \times 1 \times 2^{40}$	782.12	5211.05	5994.41	5994.41
61	(4, 0, 20, 1, 14)	$1 \times 2^1 \times 2^{40}$	872.34	5204.18 (1)	6077.13	11280.34
62	(4, 0, 20, 2, 14)	$1 \times 2^2 \times 2^{40}$	920.24	10485.95 (2)	11407.64	21892.14
63	(4, 4, 21, 0, 14)	$2^4 \times 1 \times 2^{38}$	9406.21 (11)	14827.94 (11)	24234.15	35250.72
64	(4, 3, 21, 1, 13)	$2^3 \times 2^1 \times 2^{39}$	1991.48 (2)	10469.58 (4)	12456.97	49844.24
65	(4, 3, 21, 2, 14)	$2^3 \times 2^2 \times 2^{39}$	1046.62 (1)	10517.21 (4)	11565.10	92510.64
66*	(4, 1, 21, 5, 13)	$2^1 \times 2^5 \times 2^{39}$	16268.10 (2)	133896.93 (51)	151867.70	184295.62
67*	(4, 0, 21, 7, 13)	$1 \times 2^7 \times 2^{39}$	10298.95	198835.78 (74)	209172.34	354231.11

Table 2: Solving overdetermined systems with a single GTX 980 graphics card.

n	Parameters (D, p, k, b, t)	Search Space $2^p \times 2^b \times 2^{n-p-k-b}$	Extracting Sub-systems (seconds)	Total Runtime (seconds)	Maximum Runtime (GPU-hours)
68	(4, 6, 21, 2, 13)	$2^6 \times 2^2 \times 2^{39}$	9799.15	12802.11	214.45
69	(4, 8, 22, 0, 13)	$2^8 \times 1 \times 2^{39}$	11238.49	56697.70	229.10
70	(4, 7, 22, 2, 13)	$2^7 \times 2^2 \times 2^{39}$	14367.71	44223.81	452.65
71	(4, 8, 22, 2, 13)	$2^8 \times 2^2 \times 2^{39}$	14392.00	87415.91	947.20
72	(4, 9, 22, 2, 13)	$2^9 \times 2^2 \times 2^{39}$	13912.39	144145.58	1867.44
73	(4, 8, 22, 4, 13)	$2^8 \times 2^4 \times 2^{39}$	18055.07	159585.32	3700.87
74	(4, 10, 22, 3, 13)	$2^{10} \times 2^3 \times 2^{39}$	15163.72	118323.38	8236.05

Table 3: Solving overdetermined systems using 27 nodes of the Saber clusters.

except in the first run. Some of the cluster nodes we used have GTX 780 graphics cards with only 3GB of off-chip memory while GTX 980 graphics cards have 4GB. Therefore, we adjusted the parameters t and b according to the memory size of the GTX 780. Consequently, the 4GB off-chip memory on the GTX 980 was not fully utilized but there was no noticable impact on performance.

Impact of *k*. The experiments show that despite the number of variables increasing by one for each experiment, whenever the maximum value of the parameter k increases (either with or without external hybridization), the runtime almost stays the same. For example, for the overdetermined $\mathcal{MQ}$ system $\mathcal{F}_{68}, n = 68$ by keeping $k = 21$ variables, there are 47 variables left in $\mathcal{F}_{68}$ to enumerate (see Table 3). For the overdetermined $\mathcal{MQ}$ system $\mathcal{F}_{69}, n = 69$, the maximum value of k can be increased by one (with external hybridization), so $k = 22$ variables can be kept. Therefore, there are also 47 variables to enumerate for $\mathcal{F}_{69}$. Hence, for both systems the total maximum number of iterations that need to be performed during Gray-code enumeration is the same. However, since for $\mathcal{F}_{69}$ linear systems in 22 variables instead of 21 have to be computed, the cost of each iteration of Gray-code enumeration for $\mathcal{F}_{69}$ is slightly larger than for $\mathcal{F}_{68}$. Thus, the maximum runtime for $n = 69$ is slightly larger than for $n = 68$.

Comparison. Previous records of solving Type-I challenges were held by Fast Exhaustive Search [4] and Crossbred [11]. The Fast Exhaustive Search implementation is able to perform full enumeration over the search space for an $\mathcal{MQ}$ system in 64 variables in 956 days [4]. Therefore, it solves a $\mathcal{MQ}$ system of 134 equations in 67 variables in at most $956 \cdot 2^{67-64} = 7648$ FPGA-days. The original Crossbred implementation reduces the search space for this system by a factor of 2^{-22} and requires at most 258 CPU-days to solve the challenge. As shown in Table 3, our implementation requires at most 354231.11 seconds, i.e., 4.1 GPU-days.

For solving the largest Fukuoka Type-I $\mathcal{MQ}$ challenge of 148 equations in 74 variables, the Fast Exhaustive Search implementation requires at most 2924 FPGA-years while the Crossbred implementation needs at most 41 CPU-years. Our implementation takes at most 8236.05 GPU-hours, which is less than one GPU-year.

The GPU implementation of the Fast Exhaustive Search from [3] requires 21 minutes to solve an $\mathcal{MQ}$ system with $n = 48$ variables on a GTX 295 graphics card. The GTX 295 graphics card has 480 CUDA cores and runs at 1242 MHz. Therefore, the interpolated maximum runtime for that implementation on a GTX 980 with 2048 CUDA cores running at 1278.50 MHz for $n = 74$ is roughly

$$2^{74-48} \cdot 21 \text{ minutes} \cdot \frac{1242}{1278} \cdot \frac{480}{2048} \approx 5,345,000 \text{ GPU-hours.}$$

This is about 650 times more than our implementation.

Estimated Security for $\boldsymbol{n = 74, m = 2n}$. As mentioned above, a GTX 980 graphics card consists of 2048 CUDA cores operating at 1278.50 MHz. Based on profiling information, our implementation achieves 37.06% GPU utilization. Therefore, we estimate the security strength of this particular $\mathcal{MQ}$ system, defined as the number of operations required to solve the system, as

$$8236.05 \cdot 2048 \cdot 1278.50 \cdot 10^6 \cdot 3600 \cdot 0.3706 \approx 2^{64.64}$$

Thus, an $\mathcal{MQ}$ system where $n = 74, m = 2n$, only provides about 64 bits of security.

5.2 Determined Systems

Determined systems with $n = m$ are not included in the Fukuoka $\mathcal{MQ}$ challenges. Therefore, we performed experiments for such systems using randomly generated, solvable systems. We solved those systems with one single GTX 980 graphics card on a node in the Saber2 cluster. The experimental results are given in Table 4, whose format and notation is the same as Table 2.

For determined systems, the number of variables that can be kept is much smaller than for overdetermined systems due to that fact that fewer equations are available. However, the linear systems that are enumerated during Gray-code enumeration consist of fewer variables. Therefore, the cost of each iteration is also lower. As Table 4 shows, solving a determined $\mathcal{MQ}$ system in n variables is roughly as difficult as solving an overdetermined $\mathcal{MQ}$ system where $m' = 2n', n' = n + 7 \sim n + 8$. Nevertheless, Figure 1 shows that the gap between the

n	Parameters $(D,\ p,\ k,\ b,\ t)$	Search Space $2^p \times 2^b \times 2^{n-p-k-b}$	Extracting Sub-systems (seconds)	Exhaustive Search (seconds)	Total Runtime (seconds)	Maximum Runtime (seconds)
46	(4, 0, 12, 0, 16)	$1 \times 1 \times 2^{34}$	33.90	47.63	82.12	82.12
47	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{35}$	36.31	96.12	132.92	132.92
48	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{36}$	39.76	190.59	230.88	230.88
49	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{37}$	42.98	380.91	424.48	424.48
50	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{38}$	46.86	754.86	802.34	802.34
51	(4, 0, 12, 0, 15)	$1 \times 1 \times 2^{39}$	50.74	1542.07	1593.46	1593.46
52	(4, 0, 12, 0, 14)	$1 \times 1 \times 2^{40}$	53.59	3049.07	3103.21	3103.21
53	(4, 0, 12, 1, 14)	$1 \times 2^1 \times 2^{40}$	57.05	6249.61 (2)	6307.22	6307.22
54	(4, 0, 12, 2, 14)	$1 \times 2^2 \times 2^{40}$	60.86	3141.67 (1)	3205.11	12635.54
55	(4, 1, 13, 1, 14)	$2^1 \times 2^1 \times 2^{40}$	95.30 (1)	3322.54 (1)	3418.48	13480.76
56	(4, 0, 13, 3, 14)	$1 \times 2^3 \times 2^{40}$	118.85	6600.55 (2)	6720.12	26521.05
57	(4, 0, 13, 4, 14)	$1 \times 2^4 \times 2^{40}$	121.54	46053.43 (14)	46175.72	52754.03
58	(4, 0, 13, 5, 14)	$1 \times 2^5 \times 2^{40}$	133.97	105432.90 (32)	105567.66	105567.66
59	(4, 0, 13, 6, 14)	$1 \times 2^6 \times 2^{40}$	144.13	197303.32 (60)	197448.24	210601.00

Table 4: Solving determined systems with a single GTX 980 graphics card.

number of variables that can be kept for determined and overdetermined systems gradually becomes larger as n grows. Therefore, this observation only applies to the systems in Table 4 but not to larger determined systems, e.g. $n = 172$.

Comparison. As mentioned above, the GPU implementation of the Fast Exhaustive Search algorithm from [3] requires 21 minutes to solve an $\mathcal{MQ}$ system with $n = 48$ variables on a GTX 295 graphics card using 480 CUDA cores operating at 1242 MHz. Therefore, the interpolated maximum runtime for solving an $\mathcal{MQ}$ system where $n = m = 59$ with that implementation on a GTX 980 with 2048 CUDA cores running at 1278.50 MHz is

$$2^{59-48} \cdot 21 \text{ minutes} \cdot \frac{1242}{1278} \cdot \frac{480}{2048} \approx 9796 \text{ minutes}.$$

Our implementation requires at most 210601 seconds, i.e., about 3510 minutes. Therefore, our implementation is only about 2.8 times faster with these parameters than the old implementation. This shows that the Crossbred algorithm is less efficient for small k and therefore more suitable for larger systems and for overdetermined systems.

“80-bit Security”. Sakumoto, Shirai, and Hiwatari propose an $\mathcal{MQ}$-based public-key identification schemes and “80-bit secure” parameters $n = 84$, $m = 80$ in [15]. When using our implementation for solving such systems, the best configuration is $(D, p, k, b, t) = (4, 27, 16, 1, 14)$, because this choice gives the largest k for the smallest p such that the computation on one Macaulay matrix does not take more time than the corresponding computations on the GPU. Therefore, the runtime of extracting the sub-systems can be completely hidden by pipelining CPU and GPU computations. Extracting a sub-system with these parameters takes 985.86 seconds and each GPU kernel launch takes on average 4338.59 seconds. The maximum runtime for solving the $\mathcal{MQ}$ system is therefore

$$4338.59 \text{ seconds} \cdot 2^{(27+1)} = 36930.21 \text{ GPU-years}.$$

However, since the probability of obtaining a solution for a determined system is approximately $1 - \frac{1}{e} \approx 0.63$ [9] and the runtime r for exploring a sub-space of size 2^{80-16} with the parameters above is

$$r = 4338.59 \text{ seconds} \cdot 2^{(23+1)} = 2308.13 \text{ GPU-years},$$

the expected runtime of solving an $\mathcal{MQ}$ system where $n = 84$, $m = 80$ is only

$$r \cdot \left(1 - \frac{1}{e}\right) \cdot \sum_{i=1}^{\infty} i \, \frac{1}{e^{i-1}} = r \cdot \frac{e}{e-1} \approx 3646.85 \text{ GPU-years}.$$

This shows that solving the underlying $\mathcal{MQ}$ systems of that public-key identification scheme using the security parameters of [15] is feasible on average within one year using 3647 GTX 980 graphics cards at the cost of electricity and about \$2 million US dollars for hardware, assuming a price of \$550 US dollars per GTX 980 graphics card[5]. Following the calculation in Section 5.1, the expected number of operations required for solving such a system therefore is

$$3646.85 \cdot 2048 \cdot 1278.50 \cdot 10^6 \cdot 365 \cdot 24 \cdot 3600 \cdot 0.3706 \approx 2^{76.55},$$

i.e., these parameters are roughly “76-bit secure”.

Acknowledgments

We would like to thank Dan Berstein and Tanja Lange for fruitful discussions and for granting us access to their Saber GPU clusters at Eindhoven University of Technology and the Univeristy of Illinois at Chicago.

References

1. Berbain C., Gilbert H., Patarin J.: QUAD: A Practical Stream Cipher with Provable Security. In: Vaudenay S. (ed.) Advances in Cryptology — EUROCRYPT 2006. Lecture Notes in Computer Science, vol. 4004, pp. 109–128. Springer (2006)
2. Bernstein D.J.: The Saber Cluster. URL: https://blog.cr.yp.to/20140602-saber.html
3. Bouillaguet, C., Chen, H.C., Cheng, C.M., Chou, T., Niederhagen, R., Shamir, A., Yang, B.Y.: Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$. In: Mangard, S., Standaert, F.X. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2010. LNCS, vol. 6225, pp. 203—218. Springer Berlin Heidelberg (2010)
4. Bouillaguet C., Cheng C.M., Chou T., Niederhagen R., Yang B.Y.: Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs. In: Lange T., Lauter K., Lisoněk P. (ed.) Selected Areas in Cryptography — SAC 2013. Lecture Notes in Computer Science, vol. 8282, pp. 205–222. Springer (2014)
5. Clough C., Baena J., Ding J., Yang B.Y., Chen M.: Square, a New Multivariate Encryption Scheme. In: Fischlin M. (ed.) Topics in Cryptology — CT-RSA 2009. Lecture Notes in Computer Science, vol. 5473, pp. 252–264. Springer (2009)

[5] https://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review

6. Ding J., Schmidt D.: Rainbow, a New Multivariable Polynomial Signature Scheme. In: Ioannidis J., Keromytis A., Yung M. (ed.) Applied Cryptography and Network Security — ACNS 2005. Lecture Notes in Computer Science, vol. 3531, pp. 164–175. Springer (2005)
7. Faugère J.C.: A new efficient algorithm for computing Gröbner bases (F_4). Journal of Pure and Applied Algebra 139, 61–88 (1999)
8. Faugère J.C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In: International Symposium on Symbolic and Algebraic Computation — ISSAC 2002. pp. 75–83. ACM Press (2002)
9. Fusco G., Bach E.: Phase Transition of Multivariate Polynomial Systems. Mathematical Structures in Computer Science 19, 9–23 (2009)
10. J., P.: Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms. In: U., M. (ed.) Advances in Cryptology — EUROCRYPT 1996. Lecture Notes in Computer Science, vol. 1070, pp. 33–48. Springer (1996)
11. Joux A., Vitse V.: A Crossbred Algorithm for Solving Boolean Polynomial Systems. IACR Cryptology ePrint Archive (2017), uRL: https://eprint.iacr.org/2017/372.pdf
12. Kipnis A., Patarin J., Goubin L.: Unbalanced Oil and Vinegar Signature Schemes. In: Stern J. (ed.) Advances in Cryptology — EUROCRYPT 1999. Lecture Notes in Computer Science, vol. 1592, pp. 206–222. Springer (1999)
13. Patarin J., Courtois N., Goubin L.: QUARTZ, 128-Bit Long Digital Signatures. In: Naccache D. (ed.) Topics in Cryptology — CT-RSA 2001. Lecture Notes in Computer Science, vol. 2020, pp. 282–297. Springer (2001)
14. Porras J., Baena J., Ding J.: ZHFE, a New Multivariate Public Key Encryption Scheme. In: Mosca M. (ed.) Post-Quantum Cryptography — PQCrypto 2014. Lecture Notes in Computer Science, vol. 8772, pp. 229–245. Springer, Cham (2014)
15. Sakumoto K., Shirai T., Hiwatari H.: Public-Key Identification Schemes Based on Multivariate Quadratic Polynomials. In: Rogaway P. (ed.) Advances in Cryptology — CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 706–723. Springer (2011)
16. Shor, P.W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: Foundations of Computer Science. pp. 124—134. IEEE (1994)
17. Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. SIAM Review 41(2), 303—332 (1999)
18. Szepieniec A., Ding J., Preneel B.: Extension Field Cancellation: A New Central Trapdoor for Multivariate Quadratic Systems. In: Takagi T. (ed.) Post-Quantum Cryptography. Lecture Notes in Computer Science, vol. 9606, pp. 182–196. Springer, Cham (2016)
19. Yang B.-Y., Chen J.-M.: All in the XL Family: Theory and Practice. In: Park C., Chee S. (ed.) Information Security and Cryptology — ICISC 2004. Lecture Notes in Computer Science, vol. 3506, pp. 67–86. Springer (2005)
20. Yang B.Y., Chen O.C.H., Bernstein D.J., Chen J.M.: Analysis of QUAD. In: Biryukov A. (ed.) Fast Software Encryption. FSE 2007. Lecture Notes in Computer Science, vol. 4593, pp. 290–308. Springer (2007)