Fast Exhaustive Search for Quadratic Systems in $\mathbb{F}_2$ on FPGAs

Extended Version

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

[1] Université de Lille, France: `charles.bouillaguet@lifl.fr`
[2] National Taiwan University, Taipei, Taiwan: `doug@crypto.tw`
[3] Technische Universiteit Eindhoven, the Netherlands: `blueprint@crypto.tw`
[4] Academia Sinica, Taipei, Taiwan: `ruben@polycephaly.org, by@crypto.tw`

Abstract. In 2010, Bouillaguet *et al.* proposed an efficient solver for polynomial systems over $\mathbb{F}_2$ that trades memory for speed [BCC+10]. As a result, 48 quadratic equations in 48 variables can be solved on a graphics card (GPU) in 21 minutes. The research question that we would like to answer in this paper is how specifically designed hardware performs on this task. We approach the answer by solving multivariate quadratic systems on reconfigurable hardware, namely Field-Programmable Gate Arrays (FPGAs). We show that, although the algorithm proposed in [BCC+10] has a better asymptotic *time* complexity than traditional enumeration algorithms, it does not have a better asymptotic complexity in terms of silicon *area*. Nevertheless, our FPGA implementation consumes 25 times less energy than the GPU implementation. This is a significant improvement, not to mention that the monetary cost per unit of computational power for FPGAs is generally much cheaper than that of GPUs.

Keywords. multivariate quadratic polynomials, solving systems of equations, exhaustive search, parallelization, Field-Programmable Gate Arrays (FPGAs)

1 Introduction

Solving a system of m nonlinear multivariate polynomial equations in n variables over $\mathbb{F}_q$ is called the MP problem. It is known to be NP-hard even if $q = 2$ and if we restrict ourselves to multivariate quadratic equations (in which case we call the problem MQ). These problems are mathematical problems of natural interest to cryptographers since an NP-hard problem whose random instances seem hard could be used to design cryptographic primitives. Indeed, a seldom challenged standard conjecture is "any probabilistic Turing machine has negligible chance of successfully solving a random MQ instance with a given sub-exponential (in n) complexity when m/n is a constant" [BGP06].

This led to the development of *multivariate public-key cryptography* over the last decades, using one-way trapdoor functions to build cryptosystems such as

HFE [Pat96], SFLASH [CGP02], and QUARTZ [PCG01]. It also led to the study of "provably-secure" stream ciphers like `QUAD` [BGP06].

In *algebraic cryptanalysis*, on the other hand, 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 [CBW08], for a recent example. Fast solving would also be a very useful subroutine in attacks such as [BFJ+09].

Fast Exhaustive Search. When evaluating a quadratic system with n variables over $\mathbb{F}_2$, each variable can be chosen as either 0 or 1. Thus, a straight forward approach is to evaluate each equation for all of the 2^n choices of inputs and to return any input that is evaluated to 0 by every single equation. The 2^n inputs can be enumerated by, e.g., using the binary representation of a counter of n bits where bit i is used as value for x_i. Since there are $\frac{n\cdot(n-1)}{2}$ pairs of variables and since in a generic (random) system each coefficient is 1 with probability $\frac{1}{2}$, each generic equation has about $\frac{n\cdot(n-1)}{2}\cdot\frac{1}{2}$ quadratic terms. Therefore, this approach has an asymptotic time complexity of $\mathrm{O}(2^n\cdot m\cdot\frac{n\cdot(n-1)}{2}\cdot\frac{1}{2})$. Obviously, the second equation only needs to be evaluated in case the first one evaluates to 0 which happens for about 50% of the inputs. The third one only needs to be evaluated if the second one evaluated to 0 and so forth. The expected number of equations that need to be evaluated per iteration is $\sum_{i=1}^{m} 2^{1-i} < 2$. Thus, the overall complexity can be reduced to $\mathrm{O}(2^n\cdot 2\cdot\frac{n\cdot(n-1)}{2}\cdot\frac{1}{2}) = \mathrm{O}(2^{n-1}(n-1)n)$ or more roughly $\mathrm{O}(2^n n^2)$. Observe that the asymptotic time complexity is independent of m, the number of equations in the system, and only depends on n, the number of variables. This straight forward approach will be called *full-evaluation* approach in the remainder of this paper.

The full-evaluation approach requires a small amount of memory. The equation system is known beforehand and can be hard-coded into program code. It requires only n bits to store the current input value plus a small number of registers for the program state and temporary results. Thus, it has an asymptotic memory complexity of $\mathrm{O}(n)$.

However, [BCC+10] suggests that we can trade memory for speed. The full-evaluation approach has the disadvantage that computations are repeated since the input of two consecutive computations is only slightly different. For example, for a counter step from 16 (10000_b) to 17 (10001_b) only the least significant bit and thus the value of x_0 has changed; all the other inputs do not change, the computations not involving x_0 are exactly the same as in the previous step. In other examples, e.g., stepping from 15 (01111_b) to 16 (10000_b) more bits and therefore more variables are affected. Nevertheless, it is not important in which order the inputs are enumerated. The authors of [BCC+10] point out that, by enumerating the inputs in Gray-code order, we can make sure that between two consecutive enumeration steps only exactly one bit and therefore only one variable is changed. Therefore only those parts of an equation need to be recomputed that are affected by the change of that single variable. For $\mathbb{F}_2$ this means that in case variable x_i has changed, we only need to add $\frac{\partial f}{\partial x_i}(x)$

to the previous result. This reduces the computational cost from evaluating a *quadratic* multivariate equation in each enumeration step to evaluating a *linear* multivariate equation.

Furthermore, comparing the inputs of two consecutive evaluations of $\frac{\partial f}{\partial x_i}(x)$ for a particular variable i, one observes that due to the structure of the Gray code only one other variable x_j of the input has changed. That is, the partial derivative of each variable is also evaluated in Gray-code order, and hence the trick can be applied *recursively*. Thus, by storing the result of the previous evaluation of $\frac{\partial f}{\partial x_i}(x)$, we only need to compute the change in regard to that particular variable x_j, i.e., the second derivative $\frac{\partial^2 f}{\partial x_i \partial x_j}(x)$, which is a *constant* value for quadratic equations.

Therefore, as pointed out by [BCC+10], we can trade larger memory for less computation by storing the second derivatives in respect to all pairs of variables in a constant lookup table and by storing the first derivative in respect to each variable in registers. This requires $\frac{n \cdot (n-1)}{2}$ bits for the constant lookup table of the second derivatives and n bits of registers for the first derivatives. The computational cost is reduced to updating the value of one particular first derivative (which requires one table lookup and an `xor`) and to computing the result of the equation (which requires just another `xor`).

The computational cost for each equation is independent from the values of n and m and thus will be considered constant for asymptotic estimations. However, since a state is updated in every iteration, all equations need to be computed (in parallel, e.g., using the bitslicing technique as suggested in [BCC+10]) in every single iteration. Therefore, the asymptotic time complexity for this approach is $\mathrm{O}(2^n \cdot m)$. The asymptotic memory complexity is $\mathrm{O}(m \cdot (\frac{n(n-1)}{2} + n)) = \mathrm{O}(\frac{mn(n+1)}{2})$ or more roughly $\mathrm{O}(n^2 m)$.

Note that both the Gray-code approach and the full-evaluation approach can be combined by using only m_g equations for the Gray-code approach, thus producing 2^{n-m_g} solution *candidates* to be tested by the remaining $m - m_g$ equations using full evaluation.

Lastly, we note that Gröbner-basis methods like XL [CKP+00] and F_5 [Fau02] using sparse linear solvers such as Wiedemann might have better performance than exhaustive search even over $\mathbb{F}_2$. For example, they are claimed to asymptotically outperform exhaustive search when $m = n$ with guessing of $\approx 0.45n$ variables [YCC04; BFS+13]. However, as with all asymptotic results, one must carefully check all explicit and implicit assumptions to see how they hold in practice. When taking into account the true cost of Gröbner-basis methods, e.g., communication involved in running large-memory machines, the cross-over point is expected to be much higher than $n = 200$ as predicted in [BFS+13]. However, even systems in 200 variables are out of reach for todays computing capabilities.

The Research Question. The Gray-code approach implementation described in [BCC+10] for x86 CPUs and GPUs can solve 48 quadratic equations in 48 binary variables using just one NVIDIA GTX 295 graphics card in 21 minutes. The research question that we would like to answer in this paper is *how specifi-*

cally designed hardware would perform on this task. We approach the answer by solving multivariate quadratic systems on reconfigurable hardware (FPGAs).

While the Gray-code approach has a lower asymptotic time complexity than full evaluation and is — given a sufficient amount of memory — the best choice for a software implementation, we show in Sec. 2.1 that both approaches have the same asymptotic *area* complexity. Therefore, for an FPGA implementation the choice of using either Gray code or full evaluation depends on the specific parameters and the target architecture of the implementation. We motivate our choice and describe our implementation for the Xilinx Spartan-6 FPGA in Sec. 2.

A question that has not been discussed in [BCC+10] is the probability of having collisions of solutions during the computation: For a massively parallel implementation on GPUs or FPGAs, it is most efficient to work on a set of input values in a batch. In this case, it is necessary to detect whether more than one input value in a batch is a solution for the equation system. The implementation must guarantee that no solution is silently dropped. We discuss this effect in detail in Sec. 2.3, followed by the discussion of the implementation results and the conclusion of this paper in Sec. 3.

2 Implementation

The target hardware platform of our implementation is a Xilinx FPGA of the Spartan-6 architecture, device xc6slx150, package fgg676, and speed grade -3. The Spartan-6 architecture offers three different types of logic slices: SLICEX, SLICEL, and SLICEM.

The largest amount with about 50% of the slices is of type SLICEX. These slices offer four 6-input lookup tables (LUTs) and eight flip-flops. The LUTs can either be interpreted as logic or as memory: Seen as logic, each LUT-6 is computing the output value of any logical expression in 6 binary variables; seen as memory, each LUT-6 uses the 6 input wires to address a bit in a 64-bit read-only memory. Alternatively, each LUT-6 can be used as two LUT-5 with identical input wires and to independent output wires.

About 25% of the slices are of type SLICEL, additionally offering wide multiplexers and carry logic for large adders. Another roughly 25% of the slices are of type SLICEM, which offer all of the above; in addition, the LUTs of these slices can be used as shift registers or as distributed *read-and-write* memory.

2.1 Full Evaluation or Gray Code?

There is a major difference between programming for general-purpose architectures (from an FPGA perspective, even recent programmable GPUs are considered "general purpose") and implementing algorithms in hardware. For general-purpose architectures, the programmer has to use the resources provided by the specific architecture as efficiently as possible; the architecture has a major influence on the choice of the algorithm. However, FPGAs allow the engineer

	time	memory	comp. logic	area
full evaluation	$\mathrm{O}(2^n n^2)$	$\mathrm{O}(n)$	$\mathrm{O}(n^2 m)$	$\mathrm{O}(n^2 m)$
Gray code	$\mathrm{O}(2^n m)$	$\mathrm{O}(n^2 m)$	$\mathrm{O}(m)$	$\mathrm{O}(n^2 m)$

Table 1: Asymptotic complexities of the two approaches for exhaustive search.

to choose the hardware according to his particular needs, choosing different algorithms allows also to choose different hardware resources. For example, as described above, LUTs can either be used for logic or as memory.

Table 1 summarizers asymptotic time and memory complexities of the full-evaluation approach and the Gray-code approach. Considering a software implementation, for larger systems, the Gray-code approach obviously is the more efficient choice, since it has a significantly lower time complexity and it is rather computational than memory bound. Because the memory complexity is much smaller than the time complexity, the memory demand can be handled easily by most modern architectures for such choices of parameters n and m that can be computed in realistic time.

However, a key measure for the complexity of a hardware design is the *area consumption* of the implementation: A smaller area consumption of a single instance of the implementation allows either to reduce cost or to increase the number of parallel instances and thus to reduce the total runtime. The area can be estimated as the sum of the logic for computation and the logic required for memory: The asymptotic complexity for the computational logic of the full-evaluation approach is about $\mathrm{O}(n^2)$ for each equation line, thus in total $\mathrm{O}(n^2 m)$. The memory complexity is $\mathrm{O}(n)$, so the area complexity is $\mathrm{O}(n + n^2 m) = \mathrm{O}(n^2 m)$. We point out that in contrast to the *time* complexity, the *area* complexity *depends* on m. The asymptotic complexity for the computational logic of the Gray-code approach is $\mathrm{O}(m)$, the memory complexity is $\mathrm{O}(n^2 m)$; the area complexity in total is $\mathrm{O}(n^2 m + m) = \mathrm{O}(n^2 m)$. Therefore, the asymptotic area complexity of the full-evaluation approach is equal to the area complexity of the Gray-code approach. In contrast to a software implementation, it is not obvious from the asymptotic complexities, which approach eventually gives the best performance for specific hardware and specific values of n and m. The question is: which approach is using the resources of an FPGA more efficiently for those parameters. Before we discuss our choice, we describe the overall architecture of a parallel implementation in the following paragraphs.

Parallelization using Accelerators. Exhaustive search for solutions of multivariate systems is embarrassingly parallel — all inputs are independent from each other and can be tested in parallel on as many instances as physically available. Furthermore, resources can be shared during the computation of inputs that have the same value for some of the variables.

Assume that we want to compute 2^i instances in parallel. We simply *clamp* the values of i variables such that $x_{n-i}, \ldots, x_{n-1}$ are constant for each instance, e.g., in case $i = 4$ for instance $5 = 0101_\mathrm{b}$ variable $x_{n-1} = 0$, $x_{n-2} = 1$,

$x_{n-3} = 0$, and $x_{n-4} = 1$. Therefore, the 2^n inputs for computations of a system in n variables can be split into 2^i new systems of 2^{n-i} inputs for $n - i$ variables using precomputation. These 2^i independent systems can either be computed in parallel on 2^i computing devices or sequentially on any smaller number of devices. (Obviously there is a limit on the efficiency of this approach; choosing $i = n$ would result in solving the whole original system during precomputation.) The same procedure of fixing variables can be repeated to cut the workload into parallel instances to exploit parallelism on each computing device.

After fixing variables $x_{n-i}, \ldots, x_{n-1}$, all 2^i instances of one polynomial share the same quadratic terms; all terms involving $x_{n-i}, \ldots, x_{n-1}$ become either linear terms or constant terms. Therefore, the computations of the quadratic terms can be shared: For the Gray-code approach, the second derivatives can be shared between all instances while one set of first derivatives needs to be stored per instance; for full evaluation, the logic for the quadratic terms can be shared while the logic for the linear terms differs between the instances. Sharing resources requires communication between the instances and therefore is particularly suitable for computations on one single device. Given a sufficient amount of instances, the total area consumption is dominated by the instances doing the linear computations rather than by the shared computations on the quadratic part; therefore, the computations on the linear part require the most attention for an efficient implementation.

In the following, we investigate the optimal choices of n and m and the number of instances to exhaust the resources of *one single* FPGA most efficiently. Larger values of n can easily be achieved by running such a design several times or in parallel on several FPGAs. Larger values of m can be achieved by forwarding solution candidates from the FPGA to a host computer. The flexibility in choosing n and m allows to cut the total workload into pieces that take a moderate amount of computation time on a single FPGA. This has the benefit of recovering from hardware failures or power outages without loss of too many computations.

Choosing the Most Efficient Approach. As described above, for fixed parameters n and m we want to run as many parallel instances as possible on the given hardware. Since the quadratic terms are shared by the instances, the optimization goal is to minimize the resource requirements for the computations on the linear terms.

The main disadvantage of the Gray-code approach is that it requires access to *read-and-write memory* to keep track of the first derivatives. The on-chip memory resources, i.e., block memory and distributed memory using slices of type SLICEM, are quite limited. In contrast, the full-evaluation approach “only” requires *logic* that can be implemented using the LUTs of all types of slices.

However, each LUT in a SLICEM can store 64 bits; this is sufficient space for the first derivatives of 64 variables using the Gray-code approach. On the other hand, there are four times more logic-LUTs than RAM-LUTs. Four LUT-6 can cover at most the evaluation of 24 variables. Therefore, the Gray-code approach is using the available input ports more efficiently. This is due to the fact that the inputs for the Gray-code approach are addresses of width $O(\log n)$, whereas full evaluation requires $\mathrm{O}(n)$ inputs for the variables. This also reduces bus widths and buffer sizes for pipelining.

Finally, the Gray-code approach allows to easily reuse a placed and routed design for different equation systems by exchanging the data in the lookup tables. An area-optimized implementation of the full-evaluation approach only requires logic for those terms of an equation that have a non-zero coefficient. To be able to use the same design for different equation systems, one would have to provide logic for *all* terms regardless of their coefficients, thus roughly doubling the required logic compared to the optimal solution. The Xilinx tool chain does not include a tool to exchange the LUT data from a fully placed and routed design, so we implemented our own tool for this purpose.

All in all, the Gray-code approach has several benefits compared to the full-evaluation approach that make it more suitable and more efficient for an FPGA implementation on a Spartan-6. The figures might be different, e.g., for an ASIC implementation or for FPGAs with different LUT sizes. We decided to use the Gray-code approach for the main part of our implementation to produce a number of solution candidates from a subset of the equations. These candidates are then checked for the remaining equations using full evaluation, partly on the FPGA, partly on the host computer.

2.2 Implementation of the Gray-Code Approach

As described in Sec. 1, the Gray-code approach trades larger memory for less computation. Algorithm 1 shows the pseudo code of the Gray-code approach (see the extended version of [BCC+10]). In case of the FPGA implementation, the initialization (Alg. 1, lines 20 to 35) is performed during compile time and hard-coded into the program file.

Instead of evaluating the polynomial, first and second derivatives in respect to each variable are stored in lookup tables d' and d'' (Alg. 1, lines 27 and 32). The second derivatives are constant and thus only require read-only memory. They require a quadratic amount of bits depending on the number of variables n. The first derivatives are computed in each iteration step based on their previous value (Alg. 1, line 16). Therefore, the first derivatives are buffered in a relatively small random access memory with a size linear to n.

The implementation of the Gray-code approach works as follows: Due to the structure of the Gray code, when looking at two consecutive values v_{i-1}, v_i in Gray-code enumeration, the position k_1 of the least-significant non-zero bit in the binary representation of i is the particular bit that is toggled when stepping from v_{i-1} to v_i. Therefore, the first derivative $\frac{\partial f}{\partial x_{k_1}}$ in respect to variable x_{k_1} needs to be considered for the evaluation. Furthermore, since the last time the bit i had toggled, only the bit at the position k_2 of the second least-significant non-zero bit in i has changed. So we need to access $\frac{\partial^2 f}{\partial x_{k_1} \partial x_{k_2}}$ in the static lookup table.

```
 1: function RUN($f, n$)
 2:     $s \leftarrow$ INIT($f, n$);
 3:     while $s.i < 2^n$ do
 4:         NEXT($s$);
 5:         if $s.y = 0$ then
 6:             return s.y;
 7:         end if
 8:     end while
 9: end function
10:
11: function NEXT($s$)
12:     $s.i \leftarrow s.i + 1$;
13:     $k_1 \leftarrow \mathrm{BIT}_1(s.i)$;
14:     $k_2 \leftarrow \mathrm{BIT}_2(s.i)$;
15:     if $k_2$ valid then
16:         $s.d'[k_1] \leftarrow s.d'[k_1] \oplus s.d''[k_1, k_2]$;
17:     end if
18:     $s.y \leftarrow s.y \oplus s.d'[k_1]$;
19: end function
20: function INIT($f = a_{n,n-1}x_nx_{n-1} + a_{n,n-2}x_nx_{n-2} + \cdots + a_{1,0}x_1x_0 + a_nx_n + a_{n-1}x_{n-1} + \cdots + a_0x_0 + a, n$)
21:     state $s$;
22:     $s.i \leftarrow 0$;
23:     $s.x \leftarrow 0$;
24:     $s.y \leftarrow a$;
25:     for all $k, 0 < k < n$, do
26:         for all $j, 0 \leqslant j < k$, do
27:             $s.d''[k, j] \leftarrow a_{k,j}$;
28:         end for
29:     end for
30:     $s.d'[0] \leftarrow a_0$;
31:     for all $k, 1 \leqslant k < n$, do
32:         $s.d'[k] \leftarrow s.d''[k, k-1] \oplus a_k$;
33:     end for
34:     return $s$;
35: end function
```

Alg. 1: Pseudo code for the Gray-code approach (see [BCC+10]). The functions BIT_1 and BIT_2 return the positions of the first and second least-significant non-zero bits respectively.

Fig. 1 shows the structure of the parallel FPGA implementation of the Gray-code approach for 2^i instances. To compute k_1 and k_2 (Alg. 1, lines 13 and 14), we use a module *counter* that is incrementing a counter by 1 in each cycle (cf. Alg. 1, line 12). The counter counts from 0 to 2^{n-i}. To determine its first and second least-significant non-zero bits, we feed the counter value to a module called *gray_tree* that derives the index positions of the first and the second non-zero bit based on a divide-and-conquer approach. The output of the *gray_tree* module are buses k_1 and k_2 of width $\lceil \log_2(n) \rceil$ and two wires $enable_1$ and $enable_2$ (not shown in the figure) indicating whether k_1 and k_2 contain valid information (e.g., for all counter values 2^i the output k_2 is invalid since the binary representation of 2^i has at most one non-zero bit).

Next, we compute the address *addr* of the second derivative in the lookup table from the values k_1 and k_2 as $addr = k_2(k_2 - 1)/2 + k_1$ (cf. Alg. 1, line 16). The computation is implemented fully pipelined to guarantee short data paths and a high frequency at runtime. The modules *counter* and *gray_tree* and the computation of the address for the lookup in the table are the same for all instances and all equations and therefore are required only once.

Now, the address is forwarded to the logic for the first equation eq_0. Here, the buses *addr* and k_1 are buffered and in the next cycle forwarded to the lookup table of equation eq_1 and so on. The address is fed to the constant memory that returns the value of the second derivative d''_0. We implement the constant memory using LUTs. The address of an element in the lookup table is split into segment and offset: The 6 least significant bits address an offset into a particular

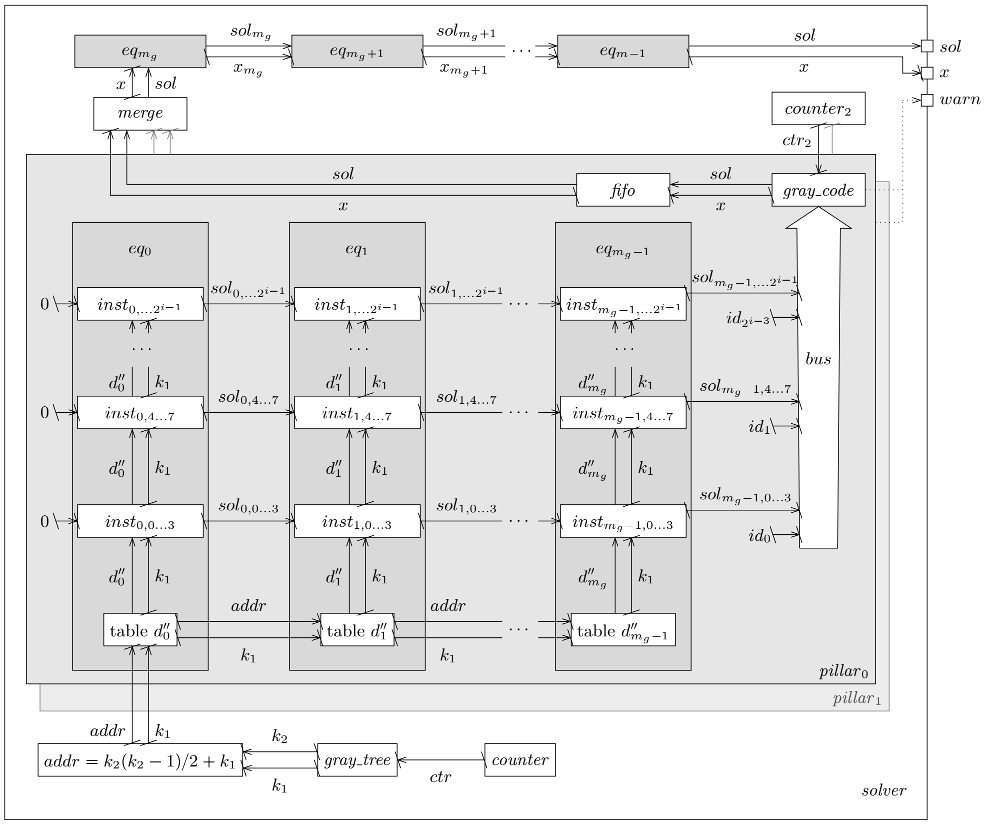

Fig. 1: Structure of the overall architecture.

LUT that holds the data for the address; the according LUT is selected by the remaining bits.

After value d''_0 of the second derivative of eq_0 has been read from the lookup table, it is forwarded together with k_1 to the first instance $inst_{0,0}$ of eq_0. Here, d''_0 and k_1 are buffered to be forwarded to the next instance of eq_0 in the next cycle and so on.

In instance $inst_{0,0}$, the value of the first derivative in respect to x_{k_1} is updated and its `xor` with the previous result y is computed. For the random access memory storing the first derivatives we are using distributed memory implemented by slices of type SLICEM. Figure 2 shows a schematic of a Gray-code instance $inst_{j,k}$. Storing the first derivative requires one single LUT-6 for up to 64 variables. Storing the result y of each iteration step requires a one-bit storage; we use a flip-flop for this purpose. The logic for updating the first derivative requires three inputs: d'', the first derivative d', and $enable_2$ to distinguish whether d'' is valid (see Alg. 1, line 15 and 16). The logic for updating y requires two inputs, the new first derivative d' and the previous value of y (Alg. 1, line 18). We combine both computations in one single LUT-6 by using one LUT-6 as two

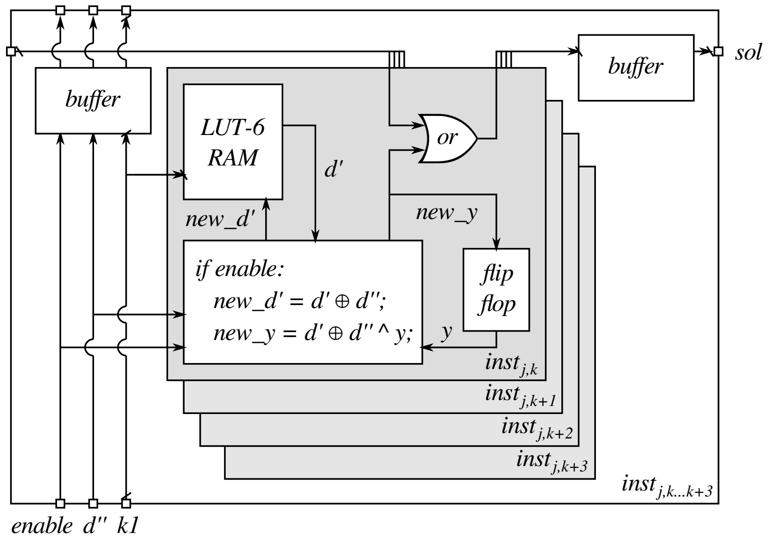

Fig. 2: Schematic of a Gray-code instance group.

LUT-5, giving four inputs d'', d', $enable_2$, and y and receiving two outputs for the new first derivative and for the new y. Furthermore, we compute the `or` with the solutions of the previous equations as $sol_{j,k} = sol_{j-1,k} \vee y$ using another LUT. Finally, the inputs d'', $enable_2$, and k_1 as well as the output are buffered using flip-flops.

Each SLICEM has four LUTs that can be addressed as memory. However, they can only be written to if they all share the same address wires as input. Therefore, we combine four instances $inst_{j,k...k+3}$ of an equation j in one SLICEM using the same address as input. As a side effect, this reduces the number of buffers that are required for the now four-fold shared inputs. All in all, for up to 64 variables, a group of four instances for one equation requires 4 slices, one of them being a SLICEM.

Finally the buffered results $sol_{j,k...k+3}$ are forwarded to $inst_{j+1,k...k+3}$ of eq_{j+1} in the next cycle. After the result of $inst_{j+1,k}$ has been computed as described before, the cumulated result $sol_{j+1,k} = sol_{j,k} \vee y$ is computed and forwarded to instance $inst_{j+2,k}$ and so on.

Eventually, the result $sol_{m_g-1,k}$ is put on a bus together with an ID that defines the value of the clamped variables. If more than one instance finds a solution candidate in the same enumeration step, there might be a collision on the bus. We describe these collisions and our counter measures in detail in Sec. 2.3.

In each cycle, the solution candidates together with their ID are forwarded from bus segment bus_i to bus_{i+1} until they eventually are leaving the bus after the last segment has been reached.

We are using the remaining resources of the FPGA to compute the actual solutions of the equation system. The computations using the Gray-code approach drastically reduce the search space from 2^n to 2^{n-m_g}. Therefore, we only need single instances of the remaining equations to check the candidates we receive from the Gray-code part. Since the inputs are quasi-random, we use full evaluation to check the candidates. If the system has more equations than we can fit on the FPGA, the remaining solution candidates are eventually forwarded to the host for final processing.

In order to check a solution candidate on the FPGA, we need to compute the actual Gray code for the input first. Since the design is fully pipelined, the value of each solution candidate from the Gray-code part is uniquely defined by the cycle when it appears. Therefore, we use a second counter ($counter_2$) that runs in sync, delayed by the pipeline length, to the original counter ($counter$). We compute the according Gray code from the value ctr_2 of this counter as $x = ctr_2 \oplus (ctr_2 >> 1)$. This value is appended to the ID and $x_{m_g-1} = (id, x)$ is forwarded into a fifo queue.

To give some flexibility when fitting the design to the physical resources on the FPGA, our design allows the instances to be split into several *pillars*, each with their own bus, *gray_code* module and fifo queue. The data paths are merged in module *merge* by selecting one solution candidate per cycles from the fifo queues in a round-robin fashion.

The solution candidate is forwarded to a module eq_{m_g} which simply evaluates equation m_g for the given input. The implementation of the evaluation is explained in detail below. the terms of the each equation to LUTs. The result of eq_{m_g} is `or`-ed to sol_{m_g-1} and forwarded to eq_{m_g+1} together with a buffered copy of x_{m_g-1} and so on.

Eventually, a vector x, its solution sol and the warning signal $warn$ are returned by the module $solver$. In case sol is equal to zero, i.e., all equations evaluated to zero for input x, the vector x is sent to the host.

The whole computation is fully pipelined and allows us to compute one evaluation in every cycle (after a warm-up latency of several cycles).

Full Evaluation. The full-evaluation approach requires logic to evaluate all equations for any given input. Furthermore, there must be logic to check whether all equations evaluate to zero for a particular input. We implemented each equation as a separate module in Verilog. The input to the module is a bus x of n wires, one wire for each variable. The output is a single wire sol for the result of the evaluation.

Given the input wires, the equation can be easily evaluated in Verilog (as in any other hardware description language (HDL)) by assigning a logical expression of the input wires to the output wire, e.g., the quadratic equation $x_5x_4 + x_3x_1 + x_2x_0 + x_1x_0 + x_5 + x_3 + x_0 + 1$ can be evaluated as:

```
assign sol = (x[5] & x[4]) ^ (x[3] & x[1]) ^ (x[2] & x[0])
      ^ (x[1] & x[0]) ^ x[5] ^ x[3] ^ x[0] ^ 1'b1;
```

However, for large equations of several hundred terms this may result in long data paths and thus in a low clock frequency. Furthermore, long expressions make the work of the FPGA tool chain more difficult and increase processing time during synthesis. We encountered very long compile times even for equations with a moderate number of variables. Therefore, we are generating optimized Verilog code automatically for a given equation by explicitly transferring the logic of an equation into LUTs.

Our target architecture is a Spartan-6 FPGA, which has LUTs with 6 input ports; each of these LUT-6 can cover 6 variables and thus at most $\binom{6}{2} = 15$ quadratic terms. We do not need to worry about the linear and constant terms; assuming that each variable appears in at least one quadratic term, linear and constant terms can be packed "for free" into the LUTs computing the quadratic terms. For example the equation above requires only one single LUT with the inputs x_0 to x_5.

Mapping the quadratic terms to LUTs breaks down to the Set Cover Problem: Given a universe U and a set S of subsets of U, find a minimal cover $C \subseteq S$ of U such that the union of C is U and such that $|C|$ is minimal. The elements in the universe U we want to cover are the quadratic terms of the equation. The sets in S are given by picking for all choices of 6 variables the terms of the equation that are pairwise combinations of these variables. The Set Cover Problem is NP-complete; therefore we implemented a Greedy algorithm to obtain a reasonable approximation of the optimal solution in a moderate amount of time.

After splitting the equation into sub-equations of up to 6 variables, these intermediate results are added up in a tree fashion. Our approach performs the mapping of the equation to LUTs much faster than the FPGA tool chain and gives slightly better results (i.e., a smaller number of LUTs). Furthermore, this code-generation approach allows us to fully pipeline the evaluation of the equation at each level of the tree.

It would be possible to reduce the number of LUTs further by sharing the result of common subexpressions between several equations. However, the assignment of LUT is complicated and we require the full evaluation only to check solution candidates provided by the Gray-code approach; the percentage of full evaluation on the overall design is very small. Therefore, there is no need to put too much effort in fully optimizing this part of the design.

2.3 Collisions, or Overabundance of Candidate Solutions

A parallelized brute-force enumerative solution of a system of equations is akin to a map-reduce process, wherein $V = 2^n$ input vectors (n being the number of variables) are passed to many small instances that each screen a portion of the inputs against a subset of k equations. Solution candidates which pass this stage move to a stage where they are checked against the remaining equations.

Every input processed by one of the processing instances may become a candidate solution with probability 2^{-k}. This is an individually very unlikely event. Logically, the checking stage requires only a small fraction of the throughput of the screening stage, occupying a correspondingly smaller amount of resources. How to collect the candidate solutions from a large number of screening instances and channel them to the checker becomes a problem.

When the screening instances are execution threads or processor cores on standard CPUs, there will be relatively few of them, each with dedicated multiple kB of high-speed static RAM as well as processor state, control logic and read-write ports on the memory bus. Thus it is viable for each instance (core or thread) to process its own pile of candidate solutions.

When performing the same computation on a GPU or fully-pipelined FPGA, the same resources mentioned above — SRAM, control logic, and memory bandwidth — are scarce. The programmer has to partition inputs into small pools. Some buffering then enables all candidate solutions in this pool to be recovered up to a given number, past which an error is returned and most of the pool must be re-checked. Re-checking may be delegated to a CPU, as in [BCC+10, Sec. 7.2], which is a highly efficient GPU implementation with a buffer depth of only 2, and it was experimentally shown that the costs of re-checking does not overwhelm the cost of the initial screening.

Expecting Collisions. Let us assume that each of 2^n candidate vectors is checked against k equations in pools of size $P = 2^s$. [BCC+10] using a GPU such as the NVIDIA GTX 295 has $(n, k, s) = (48, 32, 11)$. A reasonable setup on a Spartan-6 FPGA might have $(n, k, s) = (48, 28, 9)$ or $(n, k, s) = (48, 14, 10)$.

A back-of-the-envelope calculations would go as follows: There are approximately $V/2^k = 2^{n-k}$ candidate solutions, randomly spread among $V/P = 2^{n-s}$ pools. The birthday paradox says that we may reasonably expect one or more collisions from x balls in y bins as soon as $x \gtrsim \sqrt{2y}$, therefore we should expect a small but non-zero number of "collisions", pools that have more than one solution.

To articulate the above differently, each test vector has probability 2^{-k} to pass screening, and the event for each vector may be considered independent. Thus, the probability to have two or more solutions among a pool of P is given by the sum of all coefficients of the quadratic and higher terms in the expansion of $(1 + (x-1)/2^k)^P$. The quadratic term represent the probability of having a collision of two values, the cubic term the probability of three values, and so on. The quadratic coefficient can be expected to be the largest and contribute to most of the sum. The expected number of collisions among all inputs is V/P times this sum, which is roughly

$$(V/P)\left[x^2\right]\left((1-2^{-k}) + 2^{-k}x\right)^P = (1-2^{-k})^{P-2}\frac{(P-1)}{2^{2k-n+1}} \approx 2^{s+n-2k-1}.$$

The last approximation holds when $(1-2^{-k})^{P-2} \approx \exp\left(2^{-(k-s)}\right) \approx 1$ and $P \gg 1$.

We can judge the quality of this approximation by the ratio between the quadratic and cubic term coefficients, which is $(P-2)2^{-k}/3 \lesssim 2^{-(k+1-s)}$. In other words, if $k-s > 3$, the number of expected collisions is roughly $2^{n+s-(2k+1)}$ with an error bar of 5% or less. Similarly, the expected number of c-collisions (with at least c solutions among the same pool) is

$$(V/P)\left[x^c\right]\left((1-2^{-k}) + 2^{-k}x\right)^P \approx 2^{n-ck+(c-1)s}/c!.$$

2.4 Choosing Parameters

The parallel implementation described in Sec. 2.2 has two crucial parameters: the number of instances 2^i and the number of Gray-code equations m_g. This section describes how to choose these parameters to fit the capabilities of the target FPGA.

In case of the Spartan-6 xc6slx150-fgg676-3 FPGA, the slices are physically located in a quite regular, rectangular grid of 128 columns and 192 rows. The grid has some large gaps on top and in the bottom as well as several large vertical and a few small horizontal gaps. By picking a subset of slices in the center of the FPGA we obtain a regular grid structure of 116 columns and 144 rows. Each row has 29 groups of 4 slices: one SLICEM, one SLICEL and two SLICEX. Such a group has enough resources for four Gray-code instances each. Therefore, the whole region can be used for up to $29 \cdot 4 \cdot 144 = 16704$ Gray-code instances. The area below the slices for the Gray-code instances is used for the modules *counter* and *gray_tree*, for the computation of the address, and for the second-derivative tables. The area above the instances contains enough logic for evaluating the remaining equations using full evaluation and for the logic necessary for FPGA-to-host communication.

Using 128 rows, we could fit $128 \cdot 4 = 512$ instances of 28 equations of the Gray-code approach onto the FPGA — one equation per column, four instances per row — while guaranteeing short signal paths, leaving space of four slice columns for the bus, and giving more space for full evaluation on the top. With 28 equations in 2^9 instances, collisions of two solutions during one cycle are very rare and easy to handle by the host CPU. The obvious optimization to double the number of instances (and halve the runtime), however, introduces additional complications: Even if we can fit 14 equations into the Gray-code approach, Sec. 2.3 shows that one collision appears every 2^{10} cycles on average. We can no longer use the simple approach of re-checking all blocks with collisions on the CPU, we have to handle collisions on the FPGA. We describe in the following how to achieve 2^{10} instances for up to 14 (actually only 12) equations.

Handling of Collisions. Due to the physical layout of the FPGA and in order to save space for input-buffers, our implementation groups four instances with the same inputs together into an instance group. Instead of resolving a collision within an instance group right away, we forward a word of four bits, one for each instance, to the bus and cope with those collision later.

Whenever there is a collision at a instance group j, i.e., there is already a solution candidate on the bus in segment bus_j, the candidate of group j is postponed giving precedence to the candidate on the bus. However, the actual input giving this solution candidate is not stored in the Gray-code instances but is later derived from the cycle in which the solution was found. Therefore, delaying the solution distorts the computation of the input value. Computing the input value immediately at each bus segment would require a lot of logic and would increase the bus width to n. Instead, we count how many cycles a candidate is postponed. We use 4 bits to encode this information. Therefore, we

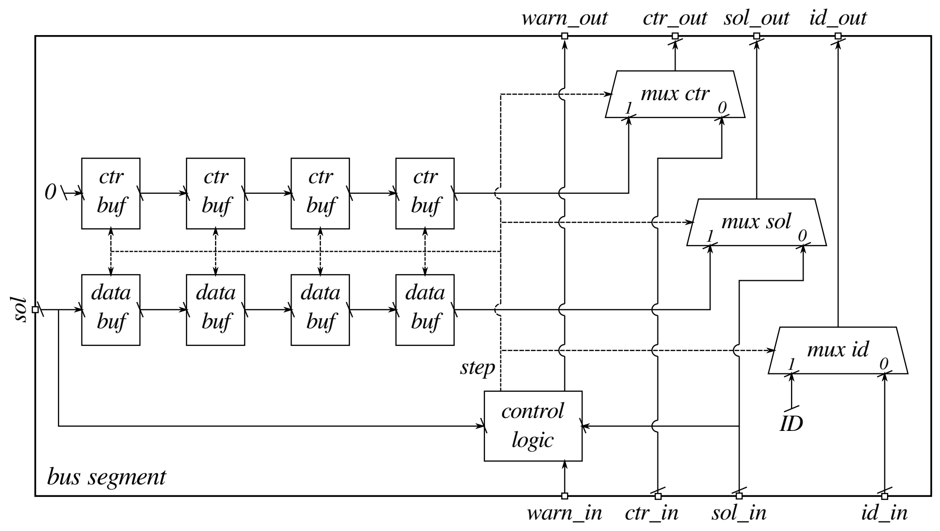

Fig. 3: Schematic of a bus segment.

can cope with a push-back of at most 14 cycles, encoding 15 and more cycles of bush-back as 1111_b; this value is treated as an error in the following logic and is reported to the host. Since the delay has a very limited maximum number of cycles, we can not use classical bus congestion techniques like exponential backoff; we must ensure that candidates are pushed onto the bus as soon as possible. This leads to high congestion in particular at the end of the bus.

Due to the push-back, our collision pool has become temporal as well as spatial. That is, it might happen that another solution candidate is produced by the same instance group before the previous one is handed to the bus. Therefore, we provide four buffer slots for each instance group to handle the rare cases where candidates are pushed back for several cycles while further candidates come up. If there are more candidates than there are buffer slots available, a warning signal is fired up and the involved input values are recomputed by the host.

All in all, the bus is transporting $i + 7$ signals for 2^i instances; $i - 2$ signals for the instance-group ID of the solution candidate, 4 signals for the push-back counter, 4 signals for the four outputs of a group of instances, and 1 warning signal.

Figure 3 shows a schematic of a bus segment. The solutions from an instance group of eq_{m_g} are sent in from the left using signal *sol*; the inputs from the previous bus segment are shown in the bottom. Whenever there is no signal on the bus, i.e., *sol_in* is all high, the control logic sets the signal *step* to high and a buffered result is pushed onto the bus; further delayed results are forwarded to the next buffer. If an available result can not be sent to the bus because there is already data on the bus, the step signal is set to low and each cycle counter in the counter buffers is incremented by one.

The logic for each bus segment covering a group of 4 instances requires 5 slices (the area of 2.5 instances) including buffers, counters, and multiplexers.

Therefore, even though 29 instances would fit into one row on the FPGA, with two buses and two pillars of instances we can only fit instances for 12 equations, but we do achieve the desired $2^{10} = 1024$ instances.

At the end of the buses, two FIFOs buffer the solution candidates so that the two data streams can be joined safely to forward a single data stream to the following logic for further handling of solution candidates (see Fig. 1). Here also the occasional collisions of solutions are resolved that might occur in an instance group of four instances as described above. Since the Gray-code part is using 2^{10} instances and 12 equations, there is one solution candidate on average every $2^{12-10} = 4$ cycles going into full evaluation.

With each bus averaging 1/8 new entries and being capable of dispatching 1 entry every cycle, the buses should not suffer from too much congestion (confirmed by simulations and tests). With a push-back of maximally 14 cycles, an unhandleable super-collision should only happen if 15 candidates appear within 15 consecutive 4-instance groups each with probability 2^{-10}, *all within 15 cycles*. The back-of-the-envelope probability evaluation like in Sec. 2.3 gives us $\binom{225}{15}\left(2^{-10}\right)^{15} \approx 6.4 \times 10^{-21}$. Just to be very sure, such super-collisions are still detected and passed to the host CPU, which re-checks the affected inputs. We can see that the CPUs on even a 256-FPGA Rivyera have sufficient computation power to recheck 1 in 5 million blocks. In all our tests and simulations, we have detected no super-collisions, which confirms that our push-back buffer and counter sizes are sufficient to prevent too many unhandled collisions that needs to be passed back to CPU.

We are able to fit logic for full evaluation of at least 42 more equations on the chip, giving 54 equations in the FPGA in total. This reduces the amount of outgoing solution candidates from the FPGA to the host computer to a marginal amount. Therefore, the host computer is able to serve a large amount of FPGAs even for a large total amount of equations in the system.

3 Performance Results and Concluding Remarks

We tested our implementation on a "RIVYERA S6-LX150 FPGA Cluster" from SciEngines. The RIVYERA has a 19-inch chassis of 4U height with an off-the-shelf host PC that controls 16 to 128 Spartan-6 LX150 FPGAs (xc6slx150-fgg676-3); our RIVYERA has 16 FPGAs. The FPGAs are mounted on extension cards of 8 FPGAs each with an extra FPGA exclusively for the communication with the host via PCIe.

Area Consumption. The Spartan-6 LX150 FPGA has 23,038 slices. In total, our logic occupies 18,613 slices (80.79%) of the FPGA. We are using 63.44% of the LUTs and 44.47% of the registers.

The logic for the Gray-code evaluation occupies the largest area with 15,281 slices (67.43%). Only 253 of those slices are used for the second-derivative tables, the counter, and address calculation. The bus occupies 2,740 slices, the remaining 12,288 slices are used for the 1,024 instances of 12 equations.

The logic for full evaluation of the remaining 42 equations, the fifo queues, and the remaining logic requires 1,702 slices (7.39%). Each equation in 54 variables requires 88 LUTs for computational logic, thus about 22 slices. All these slices are located in an area above the Gray-code logic. More than 50% of the slices in this area are still available, leaving space to evaluate more equations using full evaluation if required.

The logic for communication with the host using SciEngine's API requires 1,377 slices (5.98%).

Performance Evaluation. The GPU implementation of [BCC+10] from 2010 uses a GTX 295 graphics card. We also tried to run their CUDA program on a GTX 780 graphics card which is state-of-the-art in 2013. However, the computation took slightly more time on the GTX 780 than on the GTX 295, although the GTX 780 should be more than three times faster than the GTX 295: the GTX 780 has 2304 ALUs running at 863MHz while the GTX 295 has 480 ALUs running at 1242MHz. We suspect that the relative decrease of SRAM compared to the number of ALUs and the new instruction scheduling of the new generation of NVIDIA GPUs is responsible for the tremendous performance gap. To get full performance on the GTX 780 a thorough adaption and hardware-specific optimization of the algorithm would be required; the claim of NVIDIA that CUDA kernels can just be recompiled to profit from new hardware generations does not apply. Since most of the computing power of the GTX 780 is wasted for the currently available GPU implementation, we will continue the discussion based on the outdated GTX 295 graphics card.

Our Spartan-6 FPGA design runs at 200MHz. The design is fully pipelined and evaluates 2^{10} input values in each clock cycle. Thus, we can find all solutions of a system of 48 variables and 48 equations by evaluating all possible 2^{48} input values in $2^{48-10}/200\text{MHz} = 23\text{min}$ with a single FPGA. The GTX 295 graphics card computes all solutions of the system in 21min. Therefore, a Spartan-6 FPGA performs about the same as the [BCC+10] GPU implementation.

However, total runtime is not the only factor that affects the overall cost of the computation; power consumption is another important factor. We measured both the power consumptions of the Spartan-6 FPGA and the GTX 295 during computation: Our RIVYERA requires 305W on average during the computation using all 16 FPGAs. The host computer with all FPGA cards removed requires 165W. Therefore, a single FPGA requires $(305\text{W}-165\text{W})/16 = 8.8\text{W}$ on average, including communication overhead. We measured the power consumption of the GTX 295 in the same way: During computation on the GTX 295, the whole machine required 357W on average. Without the graphics card, the GPU-host computer requires 122W. Therefore, the GTX 295 requires 235W on average during computation. For a system of 48 variables, a single Spartan-6 FPGA requires $8.8\text{W} \cdot 23\text{min} = 3.4\text{Wh}$ for the whole computation. The GPU requires $235\text{W} \cdot 21\text{min} = 82.3\text{Wh}$. Therefore, the Spartan-6 FPGA requires about 25 times less energy than the GTX 295 graphics card. More recent graphics cards are more power efficient than the GTX 295 but still an adapted version of the algorithm would require more energy than a Spartan-6 FPGA.

		time	energy	energy cost	
				Germany	USA
48 variables	Spartan-6	23 min	3.4Wh	–	–
	GTX 295	21 min	82.3Wh	–	–
64 variables	Spartan-6	1,042 days	216kWh	€56	US$28
	GTX 295	956 days	5,390kWh	€1,401	US$701
80 variables	Spartan-6	187,182 years	14.4GWh	€3.7 mil.	US$1.9 mil.
	GTX 295	171,603 years	353.3GWh	€91.8 mil.	US$45.9 mil.

Table 2: Comparison of the runtime and cost for systems in 48, 64, and 80 variables.

For a system of 64 variables, the very same FPGA design needs about 2^{64-10}/200MHz = 1042 days and therefore about 216kWh. For this system, the GPU requires about 965 days and roughly 5,390kWh. A single kWh costs, e.g., about €0.26 in Germany* and about US$0.13 in the USA**. Therefore, solving a system of 64 variables with an FPGA costs about 216kWh·€0.26/kWh = €56 in Germany and 216kWh·US$0.13/kWh = US$28 in the US. Solving the same system using a GTX 295 GPU costs €1,401 or US$701. Table 2 shows an overview for systems in 48, 64, and 80 variables.

80-bit Security. We want to point out that it is actually feasible to solve a system in 80 variables in a reasonable time: using $2^{80-64} = 2^{16} = 65,536$ FPGAs in parallel, such a system could be solved in 1042 days. Building such a large system is feasible; e.g., the Tianhe-2 supercomputer has 80,000 CPUs.

Each RIVYERA has up to 128 FPGAs; therefore, this computation would require 512 RIVYERAs. The list price for one RIVYERA is €70,000, about US$90,000. Therefore, solving a system in 80 variables in 2.85 years costs at most US$48 million, including the electricity bill of US$2.2 million for a continuous supply of 660kW. For comparison, the budget for the Tianhe-2 supercomputer was 2.4 billion Yuan (US$390 million), *not* including the electricity bill for its peak power consumption of 17.8MW. Therefore, 80-bit security coming from solving 80-variable systems over $\mathbb{F}_2$ is, as more cryptographers gradually acknowledge, no longer secure against institutional attackers and today's computing technology.

References

[BCC+10] C. Bouillaguet, H.-C. Chen, C.-M. Cheng, T. Chou, R. Niederhagen, A. Shamir, and B.-Y. Yang. "Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$". In: *Cryptographic Hardware and Embedded Systems – CHES 2010.* Ed. by S. Mangard and F.-X. Standaert. Vol. 6225. Lecture Notes in Computer Science. Extended Version: `http://www.lifl.fr/~bouillag/pub.html`. Springer, 2010, pp. 203–218.

*average in 2012 according to the Agentur für Erneuerbare Energien

**average in 2012 according to the Bureau of Labor Statistics

[BFJ+09] C. Bouillaguet, P.-A. Fouque, A. Joux, and J. Treger. “A Family of Weak Keys in HFE (and the Corresponding Practical Key-Recovery)”. IACR Cryptology ePrint Archive, Report 2009/619. `http://eprint.iacr.org/2009/619`. 2009.

[BFS+13] M. Bardet, J.-C. Faugère, B. Salvy, and P.-J. Spaenlehauer. “On the Complexity of Solving Quadratic Boolean Systems”. In: *Journal of Complexity* 29.1 (Feb. 2013), pp. 53–75.

[BGP06] C. Berbain, H. Gilbert, and J. Patarin. “QUAD: A Practical Stream Cipher with Provable Security”. In: *Advances in Cryptology — EUROCRYPT 2006*. Ed. by S. Vaudenay. Vol. 4004. Lecture Notes in Computer Science. Springer, 2006, pp. 109–128.

[CBW08] N. Courtois, G. V. Bard, and D. Wagner. “Algebraic and Slide Attacks on KeeLoq”. In: *Fast Software Encryption — FSE 2008*. Ed. by K. Nyberg. Vol. 5086. Lecture Notes in Computer Science. Springer, 2008, pp. 97–115.

[CGP02] N. Courtois, L. Goubin, and J. Patarin. “SFLASH, A Fast Asymmetric Signature Scheme for Low-Cost Smartcards: Primitive Specification”. Second Revised Version, `https://www.cosic.esat.kuleuven.be/nessie/tweaks.html`. 2002.

[CKP+00] 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*. Ed. by B. Preneel. Vol. 1807. Lecture Notes in Computer Science. Extended Version: `http://www.minrank.org/xlfull.pdf`. Springer, 2000, pp. 392–407.

[Fau02] 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*. ACM Press, July 2002, pp. 75–83.

[Pat96] J. Patarin. “Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms”. In: *Advances in Cryptology — EUROCRYPT 1996*. Ed. by U. Maurer. Vol. 1070. Lecture Notes in Computer Science. Extended Version: `http://www.minrank.org/hfe.pdf`. Springer, 1996, pp. 33–48.

[PCG01] J. Patarin, N. Courtois, and L. Goubin. “QUARTZ, 128-Bit Long Digital Signatures”. In: *Topics in Cryptology — CT-RSA 2001*. Ed. by D. Naccache. Vol. 2020. Lecture Notes in Computer Science. Extended Version: `http://www.minrank.org/quartz/`. Springer, 2001, pp. 282–297.

[YCC04] B.-Y. Yang, J.-M. Chen, and N. Courtois. “On Asymptotic Security Estimates in XL and Gröbner Bases-Related Algebraic Cryptanalysis”. In: *Information and Communications Security — ICICS 2004*. Ed. by J. Lopez, S. Qing, and E. Okamoto. Vol. 3269. Lecture Notes in Computer Science. Springer, Oct. 2004, pp. 401–413.