Attacking Code-Based Cryptosystems with Information Set Decoding using Special-Purpose Hardware

Stefan Heyse, Ralf Zimmermann, and Christof Paar

Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany

Abstract. In this work, we describe the first implementation of an information set decoding (ISD) attack against code-based cryptosystems like McEliece or Niederreiter using special-purpose hardware. We show that in contrast to other ISD attacks due to Lee and Brickel [7], Leon [8], Stern [15] and recently [9] (May *et al.*) and [2] (Becket *et al.*), reconfigurable hardware requires a different implementation and optimization approach: Proposed time-memory trade-off techniques are not possible in the desired parameter sets. We thus derive new parameter sets from all steps involved in the ISD attack, taking a near cycle-accurate runtime estimation as well as the communication overhead into account.
Finally, we present the implementation of a hardware/software co-design – based on the Stern's attack –, evaluate it against the challenges from the Wild-McEliece website[5], discuss its shortcomings and possible enhancements.
Keywords: Special-purpose hardware, Implementation, McEliece, Niederreiter, Challenge, Information Set Decoding

1 Introduction

The majority of the currently deployed asymmetric cryptosystems work on the basis of either the discrete logarithm or the integer factorization problem as the underlying mathematical problem. Shor's Algorithm [14] in combinination with upcoming advances in quantum computing pose a severe threat to these primitives.

The McEliece cryptosystem – introduced by McEliece in 1978 [10] – is one of the alternative code-based cryptosystems unaffected by the known weaknesses against quantum computers. Like most other systems, its key size needs to be doubled to withstand Grovers algorithms [6,12]. The same holds for Niederreiter's variant [11], proposed in 1986. The best know attacks on these promising code-based cryptosystems are decoding-attacks based on information set decoding (ISD) [13,7,8,15,9,2].

So far, all proposed ISD-variants and the single public implementation we are aware of [3] optimize the parameters for CPU-based software implementations. As code-based systems mature over time, it is important to know if and how

these attacks scale when using not only CPUs, but incorporating also dedicated hardware accelerators. This allows a more realistic estimation of the true attacking costs and attack efficiency than the analysis of an algorithm's asymptotic behaviour.

The base field of most proposed code-based systems is $GF(2)$, which is favourable for hardware implementations. The authors of [4] published a wide range of challenges [5] – including binary codes, which we target in this work with a hardware attack.

Contribution of this work: In this paper, we describe the first hardware accelerated ISD-attack using special-purpose hardware. Starting with Stern's variant [15], analyze the possibilities and restrictions of dedicated hardware, present a way of mapping collision search techniques to hardware and derive parameter sets for binary codes. We also present a nearly cycle-accurate runtime estimation targeting different FPGA families for a wide range of parameter sets from [5] and discuss the drawbacks of the attack and possible ways to build upon these results.

Outline: In Section 2, we give the necessary background regarding code-based cryptosystems and describe the basic ISD-variants. We explain the different optimization strategies and hardware restrictions in Section 3. Then, we present our implementation of the hardware optimized attack in Section 4 and finish with a discussion of the results and conclusions in Sections 5 and 6.

2 Background

In this section, we briefly discuss the background required for the remainder of this work. We start with a very short introduction into code-base cryptography including McEliece, Niederreiter and Information Set Decoding, followed by a short overview on reprogrammable hardware.

2.1 Code-based Cryptography

Definition 1. *Let $\mathbb{F}_q$ denote a finite field of q elements and $\mathbb{F}_q^n$ a vector space of n tuples over $\mathbb{F}_q$. An* [n,k]-linear code $\mathcal{C}$ *is a k-dimensional vector subspace of $\mathbb{F}_q^n$. The vectors $(a_1, a_2, \ldots, a_{q^k}) \in \mathcal{C}$ are called* codewords *of $\mathcal{C}$.*

Definition 2. *The* Hamming distance $d(x, y)$ *between two vectors $x, y \in \mathbb{F}_q^n$ is defined to be the number of positions at which corresponding symbols x_i, y_i, $\forall 1 \leq i \leq n$ are different. The* Hamming weight $wt(x)$ *of a vector $x \in \mathbb{F}_q^n$ is defined as Hamming distance $d(x, 0)$ between x and the zero-vector.*

Definition 3. *A matrix $G \in \mathbb{F}_q^{k \times n}$ is called* generator matrix *for an [n,k]-code $\mathcal{C}$ if its rows form a basis for $\mathcal{C}$ such that $\mathcal{C} = \{x \cdot G \mid x \in \mathbb{F}_q^k\}$. In general there are many generator matrices for a code. An* information set *of $\mathcal{C}$ is a set of coordinates corresponding to any k linearly independent columns of G while the remaining $n - k$ columns of G form the* redundancy set *of $\mathcal{C}$.*

If G is of the form $[I_k|Q]$, where I_k is the $k \times k$ identity matrix, then the first k columns of G form an information set for $\mathcal{C}$. Such a generator matrix G is said to be in standard (systematic) form.

Definition 4. *For any [n,k]-code $\mathcal{C}$ there exists a matrix $H \in \mathbb{F}_q^{n-k \times n}$ with $(n-k)$ independent rows such that $\mathcal{C} = \{y \in \mathbb{F}_q^n \mid H \cdot y^T = 0\}$. Such a matrix H is called* parity-check matrix *for $\mathcal{C}$. In general, there are several possible parity-check matrices for $\mathcal{C}$.*

McEliece The secret key of the McEliece cryptosystem consists of a linear code $\mathcal{C}$ over $\mathbb{F}_q$ of length n and dimension k capable of correction w errors. A *generator matrix* G, an $n \times n$ permutation P and an invertible $k \times k$ matrix S are randomly generated and form the secret key. The public key consists of the $k \times n$ matrix $\hat{G} = SGP$ and the error weight w. A message m of length k is encrypted as $y = m\hat{G}+e$, where e has Hamming weight w. The decryption works by computing $yP^{-1} = mSG+eP^{-1}$ and using a decoding algorithm for $\mathcal{C}$ to find mS and finally m.

Niederreiter The secret key of the Niederreiter cryptosystem consists of a linear code $\mathcal{C}$ over $\mathbb{F}_q$ of length n and dimension k capable of correction w errors. A *parity check matrix* H, an $n \times n$ permutation P and an invertible $n-k \times n-k$ matrix S are randomly generated and form the secret key. The public key is the $n \times n-k$ matrix $\hat{H} = SHP$ and the error weight w. To encrypt, the message m of length n and Hamming weight w is encrypted as $y = \hat{H}m^{\mathrm{T}}$. To decrypt, compute $S^{-1}y = HPm^{\mathrm{T}}$ and use a decoding algorithm for $\mathcal{C}$ to find Pm^{T} and finally m.

Information Set Decoding Attacks based on information set decoding were introduced by Prange in [13]. They are the best known algorithms, which do not rely on any specific structure in the code, which is the case for code-based cryptography, i. e., an attacker deals with a random-looking code without a known structure. In its simplest form, an attacker tries to find a subset of generator matrix columns that is error-free and where the submatrix composed by this subset is invertible. The message can then be recovered by multiplying the codeword by the inverse of this submatrix. Several improvements of the attack were published, including [7] (Lee and Brickel), [8] (Leon), [15] (Stern) and recently [9] (May *et al.*) and [2] (Becket *et al.*).

The latest and – to the best of our knowledge – only publicly available implementation is [3]. The authors present an improved attack based on Stern's variant that broke the originally proposed parameters (a binary (1024,524) Goppa code with 50 errors added) of the McEliece system. The attack ran for 1400 days on a single 2.4 GHz Core2 Quad CPU or 7 days on a cluster of 200 CPUs.

We now give a short introduction into the classical ISD-variants based on [12]. Given a word $y = c + e$ with $c \in \mathcal{C}$, the basic idea is to find a word e with Hamming weight of $e \leq w$. The ISD-algorithms differ in the assumption on the distribution of 1s in e. If a given matrix G does not successfully find a solution,

Algorithm 1 Information set decoding for parameter p

Input: $k \times n$ matrix G, Integer w
Output: a non-zero codeword c of weight $\leq w$

1: **repeat**
2: pick a $n \times n$ permutation P.
3: compute $G' = UGP = (ID|R)$ (w.l.o.g we assume the first k positions from an information set).
4: compute all the sums s of $\leq p$ rows of G'
5: **until** Hamming weight of $s \leq w$
6: **return** s

	n			
Brute force	w			
	k		$n-k$	
Lee-Brickel	p		$w-p$	
		l	$n-k-l$	
Leon	p	0	$w-p$	
Stern	$p/2$	$p/2$	0	$w-p$

Table 1. Weight profile of the codewords searched for by the different algorithms. Numbers in boxes are Hamming weight of the tuples. Derived from [12].

the matrix is randomized, swapping columns and converting the result back into reduced row-echelon form by Gauss-Jordan elimination. As each of these column swaps also transforms the positions of the error vector e, there is a chance that it now matches the assumed distribution. The trade-off is between the success probability of one iteration of Algorithm 1 (or, in other words, the number of required randomizations) and the cost of a single iteration of this algorithm. Stern's algorithm is special as it allows a collision search in the two $p/2$ sized windows by a birthday attack technique.

The latest improvements from [9] and [2] extend this technique, but are out of scope of this work because they introduce large tables highly unsuitable for hardware implementations. For the sake of completeness, Table 1 on page 4 in [9] shows the time and memory complexities of the different ISD-variants.

2.2 Reprogrammable Hardware

Compared to general-purpose CPUs (and also GPUs as a many-core architecture), application-specific integrated circuits (ASICs) – chips designed for exactly

one task – are much more efficient in terms of area- and power consumption. There are none of the architectural limitations like fixed register-width or data-busses: you have full control over the design and data paths, e. g., if you need to store matrix columns of 139 bits, you may operate natively on them. The full power and potential of ASICs comes at a price: once produces, the chip can be used for one tasks only, e. g., reusing it for 141-bit columns is not possible.

An effort to balance the two approaches leads to reconfigurable hardware, i. e., Field-Programmable Gate Arrays (FPGAs), allowing rapid hardware prototyping at the cost of a reconfiguration overhead. These chips provide a large number of lookup-tables (LUTs) and storage elements (FF) and combine them with dedicated hardware cores, e. g., fast dual-port memory or digital signal processing (DSP) cores. The designer builds upon these resources and creates a chip with an application-specific architecture, but can reprogram it on demand.

3 Modified Hardware Attack

In this section, we will describe the modified algorithms for the hardware-based attack. We will highlight the main differences to a pure software attack, the limitations posed by the hardware and the solutions to circumvent these restrictions. We will end with the parameters generated for selected challenges.

3.1 From Software to Hardware

As this is the first hardware implementation of the attacks, we need to figure out the best basis and tweak the parameters for the underlying hardware platforms. It is important to keep in mind that we are mostly restricted by the memory consumption of the matrices and that this is a hard limitation on FPGAs. Thus, we cannot precompute collision tables of several gigabytes to speed up the attack.

We evaluated the choices of parameters of the attacks for hardware suitability. As starting point, we chose Stern's ISD variant without the requirement of splitting the p-sized windows into two equal sized halves. The main problem we identified in this process were the l-bit collision-search proposed in [15] and the different choices for splitting p into p_1 and p_2 to gain the most from this search. To take advantage of the birthday-like attack strategy, while at the same time reducing memory consumption to a hardware friendly level, we developed a hash-table like memory structure called collision memory ($CMEM$). Please note that this construction fixes $p_1 = 1$ and thus $p_2 = p - 1$.

Before we explain the different hardware modules required for an ISD-attack, we need to define the parts of the matrix we use in each step. Figure 1 shows the full matrix including the identity part and the notation we use: The last k_2 columns of the matrix of $n - k$ bits each form the submatrix HK_2, where the enumerator computes all sums of $p_2 = p - 1$ columns. In the middle, k_1 columns form HK_1 of $n - k - l$ bits each. $CMEM$ contains all information about the integer representation of the remaining lower l bits of these k_1 columns.

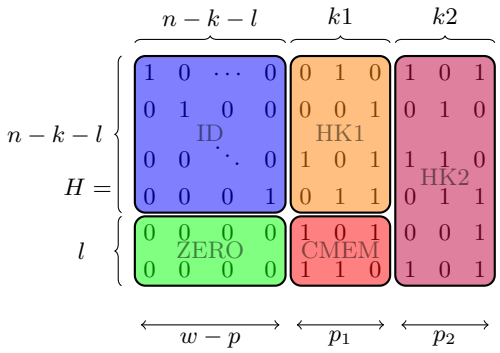

Fig. 1. Splitting of the public key into memory segments. Values under the arrows denote the assumed Hamming weight distribution of the error e

3.2 Enumerator

The most expensive step in the attacks is the computation of the $\binom{k_2}{p_2}$ sums of p_2 columns each. As $n-k$ bits per column usually do not match the register sizes of CPUs [3], we may need even more operations for each addition to update all registers involved. To reduce the costs, only the sum of the lower l bits is computed. In case of a collision with the p_1-sums from the first part of the columns, more bits are used for the sum and checked for the final Hamming weight, where usually early-abort techniques reduce the number of times the full check is done.

Please note that – as long as we store the full matrix on the FPGA – we can perform the full $n-k$ bit addition of two columns in one clock cycle in hardware, regardless of the parameters. This allows us to perform the full iteration on the FPGA without further post-computations, e. g., to sum up remaining bits.

This has another advantage: instead of computing the sums from scratch for each intermediate step, we can modify the previous sum (of p_2 columns) by utilizing a gray-code approach: we add one new column and remove one old column in one step. That way, we keep the number of p_2 columns in the sum constant and minimize the effort - given that this enumeration process is fast enough.

3.3 Collision search

As we outlined before, collision search is tricky in hardware. The approach of a large precomputed memory is not possible within the restricted device. We use a $CMEM$ construction, consisting of of $2^l \times (\lceil \log_2(k_1) \rceil + 1)$ bits, which prepares the relevant information for fast access in hardware: for a given l-bit integer, we can find out (a) if at least one of the k_1 columns contains this bit sequence in the last l positions, (b) how many matches exist and (c) the position in the memory of these columns – all in one clock cycle.

In order to remove additional wait cycles and minimize the memory consumption, we generate the part denoted as $CMEM$ in Figure 1 in two steps during

the matrix generation. First, we sort the k_1 columns according to the integer representation of the last l bits. Please note that the cost for the column swaps are negligible, as the matrix is stored in column representation. Afterwards, we generate the 2^l elements of the new structure: for index i, the MSB of $CMEM[i]$ is set only if the integer was present in the k_1 columns. The remaining l bits contain the position of its first occurrence.

Example: In the following example, we use $l = 3, k_1 = 6$. Each line represents a step in the generation process: (1) contains the integer representation of the last $l = 3$ bits of the $k_1 = 6$ columns, while (2) consists of the sorted column list and (2) of the (larger) memory content of $CMEM$.

```
1 [  0,  1,  0,  4,  3,  6 ]
2 [  0,  0,  1,  3,  4,  6 ]
3 [ 1|0, 1|2, 0|3, 1|3, 1|4, 0|5, 1|5, 0|6 ]
```

When checking for a collision with i, we simply check the MSB of $CMEM[i]$. As we are able to use two ports simultaneously, we can directly derive the number of collisions from the subtraction of $CMEM[i+1] - CMEM[i]$ and only need one multiplexer for the special case $i = k_1 - 1$. The base address is provided by the last l bits of $CMEM[i]$.

3.4 Determining Hamming Weight

For all collisions found by the collision search, a column from HK_1 is added to the current sum computed from the HK_2 columns and the Hamming weight of the result is checked against $w - p$.

The Hamming weight check in hardware needs to be a fully pipelined adder-tree, automatically generated for the target FPGA: the size of the internal look-up tables are used as a parameter during this process. More recent FPGAs with 6-input LUTs can benefit from this.

4 Implementation

In this section, we will present our hardware-implementation of the modified attack and start from a algorithmic description of the attack before we describe the software and hardware parts in more detail.

The hardware design was carefully build to work on different types of FPGAs – in this case the Xilinx Spartan-3, Spartan-6 and Virtex-6 Familiy – and integrate well into the RIVYERA FPGA cluster. Algorithm 2 describes the combination of the FPGA and the host-CPU for pre- and post-processing: the iteration on the FPGAs is computed in parallel to the generation step (CPU) and the CPU may utilize multiple parallel cores for the matrix randomization.

Algorithm 2 Modified HW/SW algorithm

Input: Challenge Parameters and the optimal attack configuration
Challenge Parameters: n, k, w, public key matrix, ciphertext
Attack Parameters: FPGA bitstream, #FPGAs, #cores, p, l, k_1
Output: Valid solution to the challenge

```
Program all available FPGAs with the provided bitstream
repeat
    for all cores do
        Randomize matrix
        Generate collision memory
        Store HK_1, HK_2, CMEM in datastream
        Store permutation
    end for
    Evaluate FPGA success flag of previous iteration
    if success then
        Read columns of successful FPGA
    else
        Burst-Transfer datastream to FPGAs
        FPGAs: compute iteration on all datasets in parallel
    end if
until success flag is set
Recover solution on challenge
```

4.1 Software Part

As mentioned in Section 3, the complete randomization step is done in software. After the challenge file and actual attack parameter are read in, as many data sets as cores available are generated. The data sets are generated using the OpenMP library in parallel. Each thread takes the original public key matrix and processes as described in Algorithm 3.

Algorithm 3 Randomization Step

Input: Public key matrix, r=#columns to swap
Output: Randomized matrix in reduced row echelon form

```
while less than r columns swapped do
    Choose a column i from the identity part at random
    Choose a random column j from the redundant part, but ensure that the bit at
    position (i, j) is one.
    Swap columns i and j
    Eliminate by optimized Gauss-Jordan
end while
Construct the collision memory(CMEM)
Store HK_1, HK_2 and CMEM in memory.
```

As the FPGA expects the data in columns, the matrix is also organized in columns in memory. Thus, pointer swaps reflect the column swaps. The Gauss-Jordan elimination is optimized taking advantage of the following facts: Only one column in the identity part has changed and the pivot bit in this column is one by definition, therefore only this column is important during elimination. Thus, only the $k+l$ rightmost bits of each row (which are in the redundant part) must be added to other rows, as the leftmost $n-k-l$ bits (except the pivot column) remain unchanged.

The performed column swaps during randomization and $CMEM$ construction are stored in a separate memory. This is necessary to recover the actual matrix the successful FPGA core was working on, as the randomized matrices are not stored. Once an FPGA sends back the $p1 = 1$ column from $CMEM$ and the $p2$ columns from the enumerator, the low weight word is recomputed locally after applying all previous permutations to the original matrix, followed by a Gauss-Jordan elimination. In a final step, the remaining $w-p$ 1s in the plaintext are recovered.

4.2 Hardware Part

As it is not possible to generate an optimized design inherently suitable for all matrices, the ISD attack requires a flexible hardware design, where we traded some hand-optimizations for a more generic design. This allows us to generate custom configurations for every parameter set with a close to optimal configuration in terms of area utilization and the number of parallel cores. These parameters are included into the source code as a configuration package and define constants used throughout the design. Thus, we can adjust the parameters very easily and automatically create valid bitstreams for the challenges.

The basic layout is the same for all FPGAs types. We use a fast interface to read incoming data, distribute it to multiple ISD-cores and initialize the local memory cores. After this initialization, all ISD-cores compute the iteration steps in parallel.

The iteration step consists of three major parts: the gray-code enumeration, the collision search and the Hamming weight computation.

Algorithm 4 describes the iteration process of each core on the FPGA. First, the different memories are initialized from the transferred data. Afterwards, the columns from the enumeration step provide the intermediate sum, which is used in the collision check step. If a collision is found on the lower l bits, the corresponding column from HK_1 is added to the sum and the Hamming weight is computed.

Enumeration Step For the enumeration process, we implemented a generic, optimized, constant-weight gray-code enumerator as described in Section 3.2. It starts with the initial state of $[0, 1, \ldots, p_2 - 1]$ and keeps track of the columns used to build the current column-sum. Aside from the internal state necessary to recover the solutions, it provides the memory core with two addresses to

Algorithm 4 Iteration Step in Hardware

Input: Memory content for $HK_1, HK_2, CMEM$, Parameters $n, k, l, w, p_2, k_1, k_2$
Output: On success: 1 column index from HK_1, p_2 column indices from HK_2

1: Initialize HK_1: $(k_1 \times (n-k-l))$-bit memory (BRAM)
2: Initialize HK_2: $(k_2 \times (n-k)$-bit memory (BRAM)
3: Initialize $CMEM$: $(2^l \times (\lceil \log_2 k_1 \rceil + 1))$-bit memory (BRAM or LUT)
4: **while** (`not` enumeration_done) **and** (`not` successful) **do**
5: Enumerate columns in HK_2 and update `sum`
6: **for** all collisions of `sum` (last l-bit) in $CMEM$ **do**
7: Update `sum` (upper part) with column from HK_1
8: **if** HammingWeight(sum) $= w$ **then**
9: Set success flag and column indices
10: Set done flag and terminate
11: **end if**
12: **end for**
13: **end while**

modify the sum. With this setup, we can compute a new valid sum of p_2 columns in exactly one clock cycle. The timing is independent of the parameters, even though the area consumption is determined by the p_2 registers of $\log_2 k_2$ bits. The enumerator is automatically adjusted to these parameters and always provides the optimal implementation for the given FPGA and Challenge.

Collision Search After the enumerator provides a sum of p_2 columns from HK_2, we check the lower l bits for collisions with $CMEM$ for valid candidates. Due to the memory restrictions on FPGAs, we keep the parameter l smaller than in software-oriented attacks. If storage in distributed memory (in constrast to a BRAM memory core) requires only small area, we automatically evaluate if an additional core may be placed when using LUTs instead of BRAMs and configure the design accordingly.

The additional logic surrounding the memory triggers the Hamming weight check in case a match was found and provides the column addresses to access HK_1.

Hamming Weight Computation The final part of the implementation is the computation of the Hamming weight. To speed up the process at a minimal delay, we split the resulting $(n-k-l)$-bit word into an adder-tree of depth $log_2(n-k-l)-1$ and compute the Hamming weight of the different parts in parallel. These intermediate results are merged afterwards with a delay equal to the depth of the tree. The circuit is automatically generated from the parameters and uses multiple registers as pipeline steps, i. e., we can start a new Hamming weight computation in each clock cycle.

4.3 Pipeline and Routing

To maximize the effect of the hardware attack, the design is build as a fully pipelined implementation: All modules work independently and store the intermediate values in registers.

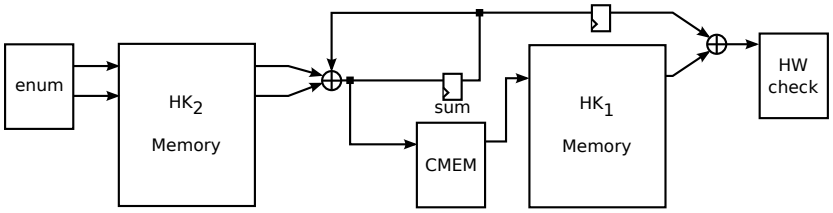

Fig. 2. Overview of the different modules inside one iteration core.

Figure 2 illustrates this pipeline structure. Every memory provides an implicit stage and the HW check is automatically pipelined. In addition, the figure shows that the single most important resource for the attack is the on-chip memory.

Due to the large amount of free area in terms of logic, i. e., not memory hardcores, the routing is usually unproblematic. In theory, we could also use parts of the free logic resources as memory and add to the dedicated memory cores. This complicates the automated generation process and does not guarantee a successful build for all parameters. Thus, we did not utilize these resources and used them to relax the routing process.

5 Results

In this section, we present the results of our analysis. The hardware results are based on Xilinx ISE Foundation 14 for synthesis and place and route. We compiled the software part using GCC 4.1.2[1] and the OpenMP library for multithreading and ran the tests on the i7 CPU integrated in the RIVYERA cluster.

5.1 Runtime Analysis

Based on the partition of the public key matrix (see Figure 1) and the distribution of errors necessary for a successful attack, the number of expected iterations is

$$\#it = \left\lceil \frac{\binom{n}{w}}{\binom{k_1}{p_1} \times \binom{k_2}{p_2} \times \binom{n-k-l}{w-p}} \right\rceil .$$

[1] Please note that the version is due to the LTS system and mentioned only for completeness. While better compiler optimizations may increase software speed, the speed-up for the overall hardware attack is negligible.

As the hardware layout is very straight-forward and is fully pipelined, we can determine the amount of cycles per iteration as

$$\#c = c_{enum} + c_{pipe} + c_{popcount} + c_{collision}$$

with

$$\begin{aligned} c_{enum} &= \binom{k_2}{p_2} \\ c_{pipe} &= 4 \\ c_{popcount} &= \log_2(n-k-l) - 1 \\ c_{collision} &= \frac{c_{enum}}{2^l} \times \frac{1}{\#mcols} \end{aligned}$$

All operations of the iteration are computed in exactly one clock cycle. After a constant pipeline delay, every clock cycle generates an iteration result. Thus, we have an almost equal running time for all iterations with one exception: The only part, which may vary from iteration to iteration is the collision search. If we find more than one candidate using $CMEM$, we need to process them before continuing with the next enumeration step. Thus, we need to add the number of multiple column candidates to the total number of clock cycles.

We can estimate the expected number of collisions inside $CMEM$ - which is the amount of multiple column candidates to test - as

$$\#mcols = k_1 \times (1 - (1 - \frac{1}{2^l})^{k_1 - 1}).$$

5.2 Optimal Parameters

We will now derive the optimal parameter sets for selected challenges taken from [5] and provide the expected number of iterations on different FPGA families: the Xilinx Spartan-3, Spartan-6 and Virtex-6. The first two are integrated into the RIVYERA framework, which features 128 Spartan-3 5000 (RIVYERA-S3) and 64 Spartan-6 LX150 (RIVYERA-S6) FPGAs, respectively. During the tests with the RIVYERA framework we noticed that the transfer time of the randomized data exceeds the generation time.

To measure the impact of the transfer speed on the overall performance, we added a single Virtex-6 LX240T evaluation board offering PCIe interface including DMA transfer. The PCIe engine based on [16,1,17,18] is, depending on the data block size, capable of transferring at 0.014Mbps, 181Mbps, 792Mbps, 1412Mbps, 2791Mbps for block sizes of 128 byte, 100 Kbyte, 500 Kbyte, 1 Mbyte, 4 Mbyte, respectively. [2]

[2] As only a single device was available and a completely different interface must be used, the actual attack is not performed using these device.

We use a Sage script to generate the optimal parameters for all challenge and provide the script and the output online[3]. Table 4 in the Appendix contains the results for the selected challenges. Given the bottleneck of the data transfer time, the script optimizes the parameters l, p and k_1 in such a way that the iteration step requires approximately as much time as transferring the data for all cores. The number of cores per FPGA depends on the challenge and the available memory and takes the area and memory consumption of the data transfer interface into account.

As the challenges from [5] are sorted according to there public key size, we selected four challenges as examples. These are the binary field challenges with public key sizes of 5Kbyte, 20KByte, 62Kbyte and 168Kbyte. The last two correspond roughly to 80 and 128 bit symmetric security, respectively[3]. All solved challenges will be send to the authors of [5] and hopefully published on their site after verification. The related parameters of the challenges C_1 to C_4 are given in Table 2 and the implementation related data in Table 3.

	C_1	C_2	C_3	C_4
n	414	848	1572	2752
k	270	558	1132	2104
w	16	29	40	54

Table 2. Parameters of C_1 to C_4

	RIV-S3	RIV-S6	V6 LX240T
clk (MHz)	75	125	250
data rate (Mbps)	240	640	up to 2791

Table 3. HW settings of C_1 to C_4

5.3 Discussion

We also implemented the complete algorithm in software to generate testvectors and to compare the runtime of the FPGA version against the CPU implementation – for small challenges – on a CPU cluster. As the algorithm operates on the full columns, the software version was extremely slow compared to both the FPGA implementation and other software implementations. Usually, only small parts of the columns (fitting into native register sizes) are added up before the collision search. Afterwards – for the candidates found in the previous step – more the sum on more register-sized parts is updates and the Hamming weight is checked, making additional use of early-abort techniques to increase the speed as well. This makes a comparison of the algorithm difficult, as neither the parameters nor the assumptions on the distribution target asymptotic behaviour.

The FPGA implementation is very fast on small challenges. Please note that one hardware iteration includes the iteration step for all cores on all FPGAs in parallel, as the parameters take the full transfer time into account. Nevertheless, for larger challenges, the implementation performs less well: the memory

[3] `http://fs.crypto.rub.de/isd`

requirements for the matrices then reduce the number of parallel cores drastically and thus remove the advantage of the dedicated hardware. This makes a software attack with a large amount of memory the better choice, as it also has the advantage of larger collision tables.

To circumvent these problems, we can also implement trade-offs in hardware as described for software implementations. To increase the number of parallel cores, we can store smaller parts of the columns, which fit the BRAM cores better and utilize the early-abort techniques. The drawback is that this approach further increases the I/O communication, as a post-processing step per iteration is necessary to check all candidates off-chip. As the communication was the bottleneck in our implementation, we did not implement this approach.

A different approach and a way to minimize the I/O communication up the process might be to generate the randomization on-chip. While the column swaps are easy to implement in one clock cycle, we need more algorithms on the device: We need both a pseudo-random number generator to identify the columns to swap and also a dedicated Gauss-Jordan elimination and also add control logic to the design so that we may reuse them by sequentially updating the cores. In addition, this approach will enforce the storage of the full matrix on the FPGA.

These restrictions and drawbacks lead to another interesting platform for ISD attacks: recent GPUs combine a large amount of parallel cores at high clock frequency and large memory. Even though the memory structure imposes restrictions, an optimized GPU implementation may prove superior to both CPUs and FPGAs. This is especially true when attacking non-binary codes, which are not optimal for FPGAs.

6 Conclusions

We presented the first hardware implementation of ISD-attacks on binary Wild McEliece challenges. Our results show that it is possible to create optimized hardware, mapping the ideas from previously available software approaches into the hardware domain and derived hardware-optimized parameters. We verified the results first in simulation and ran an unoptimized version on the FPGA cluster.

While software attacks benefit from the huge amount of available memory, CPUs are not inherently suited for the underlying operations, e. g., as the columns exceed the register sizes or as the precomputed lookup tables exceed the CPU cache. Nevertheless, a lot of effort was already invested into improvements of these software attacks.

We showed that the strength of a fully pipelined hardware implementation - the computation of all operations including memory access per iteration in exactly one clock cycle - does not lead to the expected massive parallelism, e. g., as hardware clusters have done in case of DES, and does not weaken the security of code-based cryptography dramatically: the benefit is restricted not only by

the data bus latency but - far more importantly - by the memory requirements of the attacks.

These results should be considered as a proof-of-concept and the basis for upcoming hardware/software attacks trying different implementation approaches and evaluating other algorithmic choices. We discussed the benefits and drawbacks of potential techniques for on-chip implementation of the ISD-attacks and stressed the need of an optimized GPU implementation for a better security analysis.

References

1. J. J. Ayer. *Using the Memory Endpoint Test Driver (MET) with the Programmed Input/Output Example Design for PCI Express Endpoint Cores*. Xilinx, xapp1022 v2.0 edition, November 2009.
2. A. Becker, A. Joux, A. May, and A. Meurer. Decoding Random Binary Linear Codes in $2^n/20$: How $1 + 1 = 0$ Improves Information Set Decoding. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 520–536. Springer, 2012.
3. D. J. Bernstein, T. Lange, and C. Peters. Attacking and Defending the McEliece Cryptosystem. In J. Buchmann and J. Ding, editors, *PQCrypto*, volume 5299 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2008.
4. D. J. Bernstein, T. Lange, and C. Peters. Wild McEliece. In *Proceedings of the 17th international conference on Selected areas in cryptography*, SAC'10, pages 143–158, Berlin, Heidelberg, 2011. Springer-Verlag.
5. D. J. Bernstein, T. Lange, and C. Peters. Cryptanalytic challenges for wild McEliece. `http://pqcrypto.org/wild-challenges.html`, June 2013.
6. S. Hallgren and U. Vollmer. Quantum Computing. In *Post-Quantum Cryptography*, pages 15–34, 2008.
7. P. J. Lee and E. F. Brickell. An Observation on the Security of McEliece's Public-Key Cryptosystem. In C. G. Günther, editor, *EUROCRYPT*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280. Springer, 1988.
8. J. S. Leon. A Probabilistic Algorithm for Computing Minimum Weights of Large Error-correcting Codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, 1988.
9. A. May, A. Meurer, and E. Thomae. Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$. In D. H. Lee and X. Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2011.
10. R. J. McEliece. A Public-key Cryptosystem Based on Algebraic Coding Theory. Technical report, Jet Propulsion Lab Deep Space Network Progress report, 1978.
11. H. Niederreiter. Knapsack-type Cryptosystems and Algebraic Coding Theory. *Problems Control Inform. Theory/Problemy Upravlen. Teor. Inform.*, 15(2):159–166, 1986.
12. R. Overbeck and N. Sendrier. Code-based Cryptogrpahy. In *Post-Quantum Cryptography*, pages 95–145, 2008.
13. E. Prange. The Use of Information Sets in Decoding Cyclic Codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962.
14. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, Oct. 1997.

15. J. Stern. A Method for Finding Codewords of Small Weight. In G. D. Cohen and J. Wolfmann, editors, *Coding Theory and Applications*, volume 388 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 1988.
16. J. Wiltgen and J. Ayer. *Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions*. Xilinx, xapp1052 edition, September 2010.
17. Xilinx. *Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Virtex-6, Virtex-5, Spartan-6 and Spartan-3 FPGA Families Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI*, 2010.
18. Xilinx. *Virtex-6 FPGA Integrated Block for PCI Express*, ug517 v5.0 edition, April 2010.

A Appendix

		C_1	C_2	C_3	C_4
Rivyera-S3	cores/FPGA	12	5	2	1
	p	5	4	4	4
	l	7	7	9	11
	k_1	113	127	511	1424
	k_2	164	438	630	691
	#cycles / iterations (log_2)[1]	24.79	23.73	25.31	25.71
	#expected iterations (log_2)	10.58	29.53	55.76	94.32
Rivyera-S6	cores/FPGA	32	15	7	2
	p	5	4	4	4
	l	7	7	9	11
	k_1	126	127	502	1525
	k_2	151	438	639	590
	#cycles / iterations (log_2)[1]	24.31	23.73	25.37	25.02
	#expected iterations (log_2)	10.9	29.53	55.72	94.90
Virtex-6[2]	cores/FPGA	43	21	14	6
	p	3	3	3	3
	l	6	8	10	11
	k_1	63	204	642	1578
	k_2	213	362	500	537
	#cycles / iterations (log_2)[1]	17.72	16.55	18.58	19.50
	#expected iterations (log_2)	13.82	33.40	59.95	94.96

Table 4. Optimal Parameter Set for selected Challenges

[1] Please note that the amount of cycles is the total cycle count to perform $\#cores \times \#FPGAs$ iterations, as they start after receiving data and finish all iterations within the transfer time frame of the other FPGAs.

[2] As the data transfer rate is significantly higher for the Virtex-6 device, the sage script does not optimize correctly as it neglects the - in this case - relevant preprocessing time in software and assumes zero delay.