Masking Large Keys in Hardware: A Masked Implementation of McEliece

Cong Chen[1](✉), Thomas Eisenbarth[1], Ingo von Maurich[2], and Rainer Steinwandt[3]

[1] Worcester Polytechnic Institute, Worcester, MA, USA
{cchen3,teisenbarth}@wpi.edu
[2] Ruhr-Universität Bochum, Bochum, Germany
ingo.vonmaurich@rub.de
[3] Florida Atlantic University, Boca Raton, USA
rsteinwa@fau.edu

Abstract. Instantiations of the McEliece cryptosystem which are considered computationally secure even in a post-quantum era still require hardening against side channel attacks for practical applications. Recently, the first differential power analysis attack on a McEliece cryptosystem successfully recovered the full secret key of a state-of-the-art FPGA implementation of QC-MDPC McEliece. In this work we show how to apply masking countermeasures to the scheme and present the first masked FPGA implementation that includes these countermeasures. We validate the side channel resistance of our design by practical DPA attacks and statistical tests for leakage detection.

Keywords: Threshold implementation · McEliece cryptosystem · QC-MDPC codes · FPGA

1 Motivation

Prominent services provided by public-key cryptography include signatures and key encapsulation, and their security is vital for various applications. In addition to classical cryptanalysis, quantum computers pose a potential threat to currently deployed asymmetric solutions, as most of these have to assume the hardness of computational problems which are known to be feasible with large-scale quantum computers [18]. Given these threats, it is worthwhile to explore alternative public-key encryption schemes that rely on problems which are believed to be hard even for quantum computers, which might become reality sooner than the sensitivity of currently encrypted data expires [5]. The McEliece cryptosystem [12] is among the promising candidates, as it has withstood more than 35 years of cryptanalysis. To that end, efficient and secure implementations of McEliece should be available even nowadays. The QC-MDPC variant of the McEliece scheme proposed in [13] is a promising efficient alternative to prevailing schemes, while maintaining reasonable key sizes. The first implementations

O. Dunkelman and L. Keliher (Eds.): SAC 2015, LNCS 9566, pp. 293–309, 2016.
DOI: 10.1007/978-3-319-31301-6_18

of QC-MDPC McEliece were presented in [10], and an efficient and small hardware engine of the scheme was presented in [19]. However, embedded crypto cores usually require protection against the threat of physical attacks when used in practice. Otherwise, side channel attacks can recover the secret key quite efficiently, as shown in [6].

Our Contribution. In this work we present a masked hardware implementation of QC-MDPC McEliece. Our masked design builds on the lightweight design presented in [19]. We present several novel approaches of dealing with side channel leakage. First, our design implements a hybrid masking approach, to mask the key and critical states, such as the syndrome and other intermediate states. The masking consists of Threshold Implementation (TI) based Boolean masking for bit operations and arithmetic masking for needed counters. Next, we present a solution for efficiently masking long bit vectors, as needed to protect the McEliece keys. This optimization is achieved by generating a mask on-the-fly using a LFSR-derived PRG. Through integration of PRG elements, the amount of external randomness needed by the engine is considerably reduced when compared to other TI-based implementations. Our design is fully implemented and analyzed for remaining side channel leakage. In particular, we validate that the DPA attack of [6] is no longer feasible. We further show that there are also no other remaining first-order leakages nor other horizontal leakages as exploited in [6].

After introducing necessary background in Sect. 2, we present the masked McEliece engine in Sects. 3 and 4. Performance results are presented in Sect. 5. A thorough leakage analysis is presented in Sect. 6.

2 Background

In the following we introduce moderate-density parity-check (MDPC) codes and their quasi-cyclic (QC) variant with a focus on decoding since we aim to protect the secret key. Afterwards we summarize how McEliece is instantiated with QC-MDPC codes as proposed in [13]. As our work extends an FPGA implementation of QC-MDPC McEliece that is unprotected against side channel attacks [19], we give a short overview of the existing implementation and summarize relevant works on the masking technique of threshold implementations.

2.1 Moderate-Density Parity-Check Codes

MDPC codes belong to the family of binary linear $[n, k]$ error-correcting codes, where n is the length, k the dimension, and $r = n - k$ the co-dimension of a code C. Binary linear error-correcting codes are equivalently described either by their generator G or by their parity-check matrix H. The rows of generator matrix $G \in \mathbb{F}_2^{k \times n}$ form a basis of C while $H \in \mathbb{F}_2^{r \times n}$ describes the code as the kernel $C = \{c \in \mathbb{F}_2^n | Hc^T = 0^\perp\}$ where $0^\perp$ represents an all-zero column vector. The syndrome of any vector $x \in \mathbb{F}_2^n$ is defined as $s = Hx^T \in \mathbb{F}_2^r$. Hence, the code C is comprised of all vectors $x \in \mathbb{F}_2^n$ whose syndrome is zero for a particular parity-check matrix H. MDPC codes are defined by only allowing a *moderate* Hamming weight $w = \mathrm{O}(\sqrt{n \log(n)})$ for each row of the parity-check matrix. By an (n, r, w)-MDPC code we refer to a binary linear $[n, k]$ code with such a constant row weight w.

A code C is called quasi-cyclic (QC) if for some positive integer $n_0 > 0$ the code is closed under cyclic shifts of its codewords by n_0 positions. Furthermore, it is possible to choose the generator and parity-check matrix to consist of $p \times p$ circulant blocks if $n = n_0 \cdot p$ for some positive integer p. This allows to completely describe the generator and parity-check matrices by their first row. If an (n, r, w)-MDPC code is quasi-cyclic with $n = n_0 \cdot r$, we refer to it as an (n, r, w)-QC-MDPC code.

Several t-error-correcting decoders have been proposed for (QC-)MDPC codes [1,8,10,11,13,21]. The implementation that we base our work on implements the optimized decoder presented in [10], which in turn is an extended version of the bit-flipping decoder of [8]. Decoding a ciphertext $x \in \mathbb{F}_2^n$, is achieved by:

1. Computing the syndrome $s = Hx^T$.
2. Computing the number of unsatisfied parity checks $\#_{\text{upc}}$ for every ciphertext bit.
3. If $\#_{\text{upc}}$ exceeds a precomputed threshold b, invert the corresponding ciphertext bit and add the corresponding column of the parity-check matrix to the syndrome.
4. In case $s = 0^{\perp}$, decoding was successful, otherwise repeat Steps 2/3.
5. Abort after a defined maximum of iterations with a decoding error.

2.2 McEliece Public Key Encryption with QC-MDPC Codes

The McEliece cryptosystem was introduced using binary Goppa codes [12]. Instantiating McEliece with t-error-correcting (QC-)MDPC codes was proposed in [13], mainly to significantly reduce the size of the keys while still maintaining reasonable security arguments. The proposed parameters for an 80-bit security level are $n_0 = 2, n = 9602, r = 4801, w = 90, t = 84$, which results in a much more practical public key size of 4801 bit and a secret key size of 9602 bit compared to binary Goppa codes which require around 64 kByte for public keys at the same security level.

The main idea of the McEliece cryptosystem is to encode a plaintext into a codeword using the generator matrix of a code selected by the receiver and to add a randomly generated error vector of weight t to the codeword which can only be removed by the intended receiver. We summarize QC-MDPC McEliece in the following by introducing key-generation, encryption and decryption.

Key-Generation. The parity-check matrix H is the secret key in QC-MDPC McEliece. As the code is quasi-cyclic, the parity-check matrix consists of n_0 concatenated $r \times r$ blocks $H = (H_0 \,|\, \dots \,|\, H_{n_0-1})$. We denote the first row of each of these blocks by $h_0, \dots, h_{n_0-1} \in \mathbb{F}_2^r$. The public key in QC-MDPC McEliece is the corresponding generator matrix G, which is computed from H in standard form as $G = [I_k \,|\, Q]$ by concatenation of the identity matrix $I_k \in \mathbb{F}_2^{k\times k}$ with

$$Q = \begin{pmatrix} (H_{n_0-1}^{-1} \cdot H_0)^T \\ (H_{n_0-1}^{-1} \cdot H_1)^T \\ \cdots \\ (H_{n_0-1}^{-1} \cdot H_{n_0-2})^T \end{pmatrix}.$$

The key generation starts by randomly selecting first row candidates $h_0, \ldots, h_{n_0-1} \in_R \mathbb{F}_2^r$ such that the overall row weight (wt) sums up to $w = \sum_{i=0}^{n_0-1} \mathrm{wt}(h_i)$. Since we intend to generate a code which is quasi-cyclic, the n_0 blocks of the parity-check matrix are generated from the first rows by cyclic shifts. The resulting parity-check matrix belongs to an (n, r, w)-QC-MDPC code with $n = n_0 \cdot r$. If the last block H_{n_0-1} is non-singular, i. e., if $H_{n_0-1}^{-1}$ exists, the public key is computed as $G = [I_k \,|\, Q]$. Otherwise new candidates for h_{n_0-1} are generated until a non-singular H_{n_0-1} is found.

Encryption. A plaintext $m \in \mathbb{F}_2^k$ is encrypted by encoding it into a codeword using the recipient's public key G and by adding a random error vector $e \in \mathbb{F}_2^n$ of weight $\mathrm{wt}(e) \leq t$ to it. Hence, the ciphertext is computed as $x = (m \cdot G \oplus e) \in \mathbb{F}_2^n$.

Decryption. Given a ciphertext $x \in \mathbb{F}_2^n$, the intended recipient removes the error vector e from x using the secret code description H and a QC-MDPC decoding algorithm Ψ_H yielding mG. Since $G = [I_k \,|\, Q]$, the first k positions of mG are equal to the k-bit plaintext.

2.3 FPGA Implementation of QC-MDPC McEliece

Our work extends on the lightweight implementation of McEliece based on QC-MDPC code for reconfigurable devices by [19]. Their resource requirements are 64 slices and 1 block RAM (BRAM) to implement encryption and 159 slices and 3 BRAMs to implement decryption on a Xilinx Spartan-6 XC6SLX4 FPGA. The goal of our work is to protect the secret key. Hence, only the decryption engine is discussed in this paper. From a high-level point of view, decryption works as follows: at first the syndrome of the ciphertext is computed. Then for each ciphertext bit the number of unsatisfied parity-checks $\#_{\mathrm{upc}}$ are counted and if they exceed a defined threshold, the ciphertext bit is inverted. When a ciphertext bit is inverted, the corresponding row of the parity-check matrix is added to the syndrome. The DPA presented in [6] shows that the described architecture is vulnerable to an efficient horizontal key recovery attack, since neither the key nor internal states are masked.

2.4 Threshold Implementation

Threshold implementation (TI) is a masking-based technique to prevent first order and higher order side channel leakage. Since its introduction in [15], many symmetric cryptosystems have been implemented in TI [2–4,14,17]. More importantly, most of these works have performed thorough leakage analysis and have shown that TI actually prevents the promised order leakage (if carefully implemented). Even higher-order leakage, while not prevented, usually comes at a highly increased cost of needed observations. TI performs Boolean secret sharing on all sensitive variables. Computations on the shares are then performed in a way that ensures *correctness*, maintains *uniformity* of the shares, and ensures *non-completeness* of the computation, that is, each sub-operation can only be performed on a strict sub-set of the inputs. A detailed description of TI is available in [15].

We choose TI for McEliece because TI is fairly straightforward to apply and to implement, yet it is effective. Furthermore, large parts of McEliece are linear, and hence cheap to mask using TI. The decoder part, while not linear, is also fairly efficient to mask using TI, as shown in Sect. 4. At the same time, our implementation avoids several of the disadvantages of TI: Unlike [16], we convert our addition to arithmetic masking once the values get larger, yielding a much more efficient addition engine than one solely relying on TI. By including the pseudorandom mask generation in the crypto core, we significantly cut both the required memory space usually unavoidably introduced by TI as well as the required overhead of random bits consumed by TI engines. Note that the TI-AES engines presented in [3,14] consume about 8000 bits of randomness per encryption, while our engine only consumes 160 bits per decryption.

3 Masking QC-MDPC McEliece

An effective way to counteract side channel analysis is to employ masking. Masking schemes aim to randomize sensitive intermediate states such that the leakage is independent of processed secrets. In QC-MDPC McEliece, the key bits and the syndrome are sensitive values that need to be protected and therefore they must be masked whenever they are manipulated. Similarly, since the decoding operation processes the sensitive syndrome, leakage of the decoder needs to be masked as well.

3.1 Masked Syndrome Computation

As described in Sect. 2.1, the decoding algorithm begins with the syndrome computation $s = Hx^T$. Both the parity-check matrix H and the syndrome s are sensitive values and can cause side channel leakage. However, since the syndrome computation is a linear operation, masking this operation is simple and efficient. Intuitively, H can be split into two shares, H_m and M such that $H = H_m \oplus M$, by Boolean masking. The mask matrix M is created in correspondence to H, by first generating uniformly distributed random masks for h_i, $m_0, \ldots, m_{n_0-1} \in \mathbb{F}_2^r$ of the n_0 blocks, which then comprise the first row of mask matrix M. Each bit in the m_i is uniformly set to 0 or 1. Next, the remaining rows of the mask matrix M are obtained by quasi-cyclic shifts of the first row, according to the construction of H. The masked syndrome s_m and the syndrome mask m_s can be computed independently as $s_m = H_m x^T$ and $m_s = Mx^T$. The syndrome s is available as the combination of the two shares $s = s_m \oplus m_s$.

Algorithm 1. Masked Error Correction Decoder

```
Input: $H_m$, $M_1$, $M_2$, $s_m$, $m_{s_1}$, $m_{s_2}$, $x$, $B = b_0, ..., b_{max-1}$, max
Output: Error free codeword $x$ or DecodingFailure
 1: for $i = 0$ to max−1 do
 2:     for every ciphertext bit $x_j$ do
 3:         $\#_{\text{upc}} = \mathsf{SecHW}(\mathsf{SecAND}(s_m, m_{s_1}, m_{s_2}, H_{m,j}, M_{1,j}, M_{2,j}))$
 4:         $d = (\#_{\text{upc}} > b_i)$ , $d \in \{0, 1\}$
 5:         $x = x \oplus (d \cdot 1_j)$                                  ▷ Flip the $j$th bit of $x$
 6:         $s_m = s_m \oplus (d \cdot H_{m,j} \oplus \bar{d} \cdot M_{2,j})$          ▷ Update syndrome
 7:         $m_{s_1} = m_{s_1} \oplus M_{1,j}$                              ▷ Update masks
 8:         $m_{s_2} = m_{s_2} \oplus M_{2,j} \oplus (\bar{d} \cdot M_{1,j})$
 9:     end for
10:     if $\mathsf{SecHW}(s_m, m_{s_1}, m_{s_2}) == 0$ then            ▷ Check for remaining errors
11:         return $x$
12:     end if      ▷ For constant run time, this if-statement can be moved after the
        for-loop
13: end for
14: return DecodingFailure
```

3.2 Masked Decoder

After syndrome computation, the error correction decoder computes the number of unsatisfied parity check equations between the sensitive syndrome and one row of the sensitive parity check matrix. By comparing that number with a predefined threshold (usually denoted b), the decoder decides whether to flip the corresponding bit in the ciphertext. Masking the actual decoding steps is more complex, since both inputs, namely the syndrome and the parity check matrix, as well as the control flow of the decoder can leak sensitive information and thus need to be protected. Unlike the syndrome computation, the decoder performs a binary AND and a Hamming weight computation on sensitive data. Both operations are non-linear and thus need more elaborate protection than just a straightforward Boolean masking. In the following we explain how these operations can be implemented. Algorithm 1 describes the masked version of the decoder. Note that the algorithm has been formulated with a constant execution flow to better represent the intended hardware implementation. Further note that the algorithm and its FPGA implementation exhibit a constant timing behavior (except the number of decoder iterations) and that all key-related variables are masked. The number of decoder iterations can be set to maximum by simply moving the if-statement out of the loop. For the chosen 9602/4801 parameter set, max would be set to 5, increasing the average run time roughly by a factor 2 (cf. [21]).

In Algorithm 1, we make use of two special functions. Function SecAND computes the bitwise AND operation between syndrome s and secret key H in a secure way without leaking any sensitive information. The other function SecHW computes the Hamming Weight of a given vector. Both functions are explained in detail in the following. An all-zero vector with the jth bit equal to 1 is indicated by 1_j.

Secure AND Computation. One important step when decoding a QC-MDPC code is to compute the unsatisfied parity-check equations which starts with a non-linear bitwise AND operation between the syndrome and one row of the secret key matrix. Our function SecAND performs a bitwise AND operation between two bit vectors, namely $s \wedge h$. Since the AND is a non-linear operation, simple two-share Boolean masking is not applicable. Instead, we follow the concept of Threshold Implementation as described in Sect. 2.4. We adopt the bitwise AND operation from [15], which provides first-order security when applied to *three* Boolean shares. This means that the two-share representations of the two inputs, i. e., the syndrome and parity check matrix, need to be extended to a three-share representation.

To achieve a three-share representation of both syndrome and parity check matrix, the masking is expanded in the following way: After syndrome computation as explained in Sect. 3.1, the syndrome is represented as $s_m \oplus m_s$ and the secret key is represented as $H_{m,j} \oplus M_j$. Next, the syndrome representation is extended as $s_m \oplus m_{s_1} \oplus m_{s_2}$ and the key as $H_{m,j} \oplus M_{1,j} \oplus M_{2,j}$. Here, m_{s_2} and $M_{2,j}$ are two new uniformly distributed random mask vectors and m_{s_1} is derived as $m_{s_1} = m_s \oplus m_{s_2}$ and $M_{1,j} = M_j \oplus M_{2,j}$. The following equations show how to achieve a TI version of $s \wedge h$ that satisfies correctness and non-completeness, but not uniformity.

$$\begin{aligned} s \wedge h &= (s_m \oplus m_{s_1} \oplus m_{s_2}) \wedge (H_{m,j} \oplus M_{1,j} \oplus M_{2,j}) \\ &= (s_m \wedge H_{m,j}) \oplus (s_m \wedge M_{1,j}) \oplus (H_{m,j} \wedge m_{s_1}) \oplus \\ &\quad (m_{s_1} \wedge M_{1,j}) \oplus (m_{s_1} \wedge M_{2,j}) \oplus (M_{1,j} \wedge m_{s_2}) \oplus \\ &\quad (m_{s_2} \wedge M_{2,j}) \oplus (m_{s_2} \wedge H_{m,j}) \oplus (M_{2,j} \wedge s_m) \end{aligned} \tag{1}$$

As pointed out in [15], in order to fulfill uniformity, one can introduce additional uniform random masks to mask each share. By introducing two more uniformly random vectors r_1 and r_2, the three output shares can be computed as follows. Let sh denote the result of the TI version of the AND operation. Using the equations above, sh can be split into three shares sh_i, which are now uniformly distributed thanks to the r_i and are given as:

$$\begin{aligned} sh_1 &= (s_m \wedge H_{m,j}) \oplus (s_m \wedge M_{1,j}) \oplus (H_{m,j} \wedge m_{s_1}) \oplus r_1 \\ sh_2 &= (m_{s_1} \wedge M_{1,j}) \oplus (m_{s_1} \wedge M_{2,j}) \oplus (M_{1,j} \wedge m_{s_2}) \oplus r_2 \\ sh_3 &= (m_{s_2} \wedge M_{2,j}) \oplus (m_{s_2} \wedge H_{m,j}) \oplus (M_{2,j} \wedge s_m) \oplus r_1 \oplus r_2 \end{aligned} \tag{2}$$

Secure Hamming Weight Computation. In the unprotected FPGA implementation of [19], the Hamming weight computation of sh is performed by looking up the weight of small chunks of sh from a precomputed table and then accumulating those weights to get the Hamming weight of sh. However, the weight of a chunk is always present in plain and the computation of it can result in side channel leakage that will lead to the recovery of the Hamming weight.

Even though the knowledge of the weight does not necessarily recover the chunk value, it still yields information about sh and thus the secret key h.

For a side-channel secure implementation, both the input and the output of a Hamming weight computation for each chunk must be masked. Since the weight of all chunks needs to be accumulated, it is preferable to use Arithmetic masking instead of Boolean masking. For example, the Hamming weight of sh can be calculated using the following equation:

$$\mathrm{wt}(sh) = \sum_{i=1}^{|sh|} sh_{1,i} \oplus sh_{2,i} \oplus sh_{3,i} \tag{3}$$

where subscript i refers to the i-th bit of each share and $|sh|$ is the length of sh in bits. Using a secure conversion function from Boolean masking to Arithmetic masking [7], each Boolean mask tuple $(sh_{1,i}, sh_{2,i}, sh_{3,i})$ can be converted to an Arithmetic mask pair $(A_{1,i}, A_{2,i})$ such that $sh_{1,i} \oplus sh_{2,i} \oplus sh_{3,i} = A_{1,i} + A_{2,i}$. Then, the Hamming weight of sh can be computed as:

$$\mathrm{wt}(sh) = \sum_{i=1}^{|sh|} A_{1,i} + A_{2,i} = \sum_{i=1}^{|sh|} A_{1,i} + \sum_{i=1}^{|sh|} A_{2,i} \tag{4}$$

According to Eq. (4), we only accumulate $A_1 = \sum_{i=1}^{|sh|} A_{1,i}$ and $A_2 = \sum_{i=1}^{|sh|} A_{2,i}$, respectively, and sum them up in the end to obtain the total Hamming weight $\mathrm{wt}(sh) = A_1 + A_2$.

Secure Syndrome Checking. In order to test whether decoding of the input vector was successful, the syndrome has to be tested for zero. If the Hamming weight of the syndrome is zero, then all bits of the syndrome must be zero. Otherwise, there must be some bits set as 1 and the number of set bits equals the Hamming weight of the syndrome. Note that we perform SecHW operation over the three shares of syndrome s in order to prevent the leakage.

4 Implementing a Masked QC-MDPC McEliece

This section presents more details of the masked FPGA implementation of QC-MDPC McEliece decryption based on the unprotected one in [19]. We follow the structure of the original design, including the same security parameters, but replace vulnerable logic circuits with masked circuits.

4.1 Overview of the Masked Implementation

Each time before the decryption is started, both the ciphertext and the masked secret keys h_{0m}, h_{1m} are written into the BRAMs of the decryption engine. As shown in Fig. 1, one BRAM stores the $2 \cdot 4801$-bit ciphertext, the second BRAM stores the $2 \cdot 4801$-bit masked secret key and third BRAM stores the 4801-bit masked syndrome and the 4801-bit syndrome mask. Note that the secret keys are

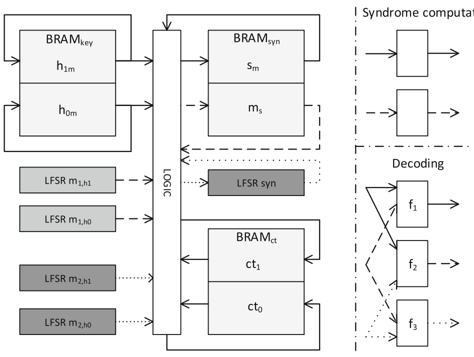


Fig. 1. Abstract block diagram of the masked QC-MDPC McEliece decryption implementation.

masked before being transferred to the crypto core. The seeds for the internal PRG are transferred with the masked key. Each BRAM is dual-ported, offers 18/36 kBit, and allows to read/write two 32-bit values at different addresses in one clock cycle.

Computations are performed in the same order as in [19]: To compute the masked syndrome s_m, set bits in the ciphertext x select rows of the masked parity-check matrix blocks that are accumulated. In parallel, the syndrome mask m_s is computed in the same manner. Rotating the two parts of the secret key is implemented in parallel, as in the unprotected implementation. Efficient rotation is realized using the READ_FIRST mode of Xilinx's BRAMs which allows to read the content of a 32-bit memory cell and then to overwrite it with a new value, all within one clock cycle.

An abstraction of this implementation is depicted in Fig. 1. The three block RAMs are used to store the masked keys (h_{0m} and h_{1m}), the shared syndrome (s_m and m_s) and the ciphertext (ct_0 and ct_1). The LFSR blocks are used to generate the missing masks on-the-fly. The logic blocks for the two phases of the McEliece decryption are shown on the left side of Fig. 1.

4.2 Masking Syndrome Computation

The syndrome computation is a linear operation and requires only two shares for sensitive variables. Once the decryption starts, 32-bit blocks of the masked secret keys h_{0m}, h_{1m} are read from the secret key BRAM at each clock cycle and are XORed with the 32-bit block of s_m read from the syndrome BRAM depending on whether the corresponding ciphertext bits are 1. Then the result will be written back into the syndrome BRAM at the next clock cycle and at the same time the rotated 32-bit blocks of the masked keys will be written back into the secret key BRAM. Meanwhile, we need to keep track of the syndrome mask m_s. Since syndrome computation is a linear operation, we can similarly add up the secret key masks synchronously to generate the syndrome mask.

In our secure engine, we use two 32-bit leap forward LFSRs to generate random 32-bit secret key masks each clock cycle which are XORed with the 32-bit block of m_s read from the syndrome BRAM depending on the ciphertext.

Cyclic Rotating LFSRs. Our 32-bit leap forward LFSRs not only generate a 32-bit random mask at each clock cycle but also rotate synchronously with the key. For example, the LFSR for h_{0m} first needs to generate the 4801-bit mask m_{h_0} in the following sequence: $m_{h_0}[0:31], m_{h_0}[32:63], \ldots, m_{h_0}[4767:4799], m_{h_0}[4800]$. This is done in 150 clock cycles. In the next round, the secret key is rotated by one bit as $h_{0m} \ggg 1$ and hence the mask sequence should be: $m_{h_0}[4800:30], m_{h_0}[31:62], \ldots, m_{h_0}[4766:4798], m_{h_0}[4799]$. After 4801 rounds of rotation, the LFSR ends up with its initial state. In order to construct a cyclic rotating PRG with a period of 4801 bits, we combine a common 32-bit leap forward LFSR with additional memory and circuits, based on the observation that the next state of the LFSR either completely relies on the current state or actually sews two ends of the sequence together, e.g., $m_{h_0}[4800:30]$. As shown in Fig. 2, five 32-bit registers are employed instead of just one. The combinational logic circuit computes the next 32-bit random mask given the input stored in IntStateReg. The following steps describe the functionality of our LFSR:

1. Initially, the 32-bit seed *seed* $[0:31]$ of the sequence is stored in register IvReg and the first 32 bits of the sequence, e.g., $m_{h_0}[0:31]$ are stored in the other registers.
2. During the rotation, the combinational logic circuits output the new 32-bit result and feed it back. If the new result is part of the 4801-bit sequence, then it will go through the Mux, overwriting the current state registers IntStateReg and ExtStateReg at the next clock cycle.
3. If the new result contains bits that are not part of the sequence, then those bits will be replaced. For example, when $m_{h_0}[4767:4799]$ is in IntStateReg, the new result will be $m_{h_0}[4800:4831]$ in which only bit $m_{h_0}[4800]$ is in the mask sequence and $m_{h_0}[4801:4831]$ will be dropped. The Mux gate will only let $m_{h_0}[4800]$ go through together with $m_{h_0}[0:30]$ stored in ExtBit$_{0\_31}$ and the concatenation $m_{h_0}[4800:30]$ will overwrite register ExtStateReg.
4. $m_{h_0}[4800:30]$ will not be written into register IntStateReg because given $m_{h_0}[4800:30]$ as input, the combinational logic circuit will not output the next valid state $m_{h_0}[31:62]$. Therefore, we concatenate part of the seed in IvReg and part of the first 32-bits in IntBit$_{0\_31}$, e.g., $\{seed[31], m_{h_0}[0:30]\}$ and overwrite IntStateReg. Then, the new output will be $m_{h_0}[31:62]$. The concatenation is implemented as a cyclic bit rotation as shown in Fig. 2. After 32 rotations, the seed is rotated to IntBit$_{0\_31}$ and the first 32-bit $m_{h_0}[0:31]$ is rotated to IvReg. Hence, they will be swapped back in the next clock cycle.

To sum up, ExtStateReg always contains the valid 32-bit mask while IntStateReg always contains 32-bit input that results in the next valid state. The rotated secret key is generated in 150 clock cycles. After 4801×150 clock cycles, the LFSR returns to its initial state and idles.

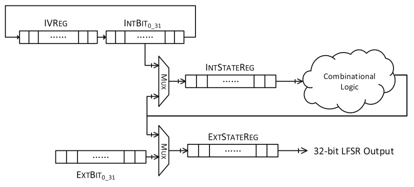


Fig. 2. The structure of the cyclic rotating LFSR that is used to generate the masks on-the-fly.

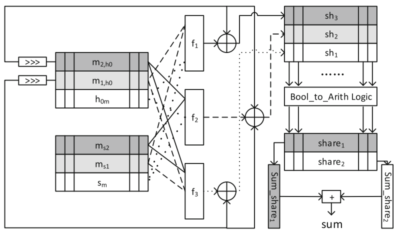


Fig. 3. Layout of our pipelined QC-MDPC McEliece decoder for the first part of the secret key, h_0.

4.3 Masking the Decoder

As mentioned in Sect. 3, the masked secret keys and the syndrome are extended to three shares. Hence, more LFSRs are instantiated to generate the additional shares as shown in Fig. 1. Two LFSRs generate the third shares of h_0 and h_1, another LFSR generates the third share of the syndrome.

We use h_0 as example to describe the decoder, since h_1 is processed in parallel using identical logic circuits. We split h_0 into three shares: h_{0m} stored in the BRAM and m_{1,h_0} and m_{2,h_0} generated by two LFSRs. The syndrome is split into s_m and m_{s_1} which are stored in BRAM and m_{s_2} which is generated by an LFSR. After decoding is started, each 32-bit share is read or generated at each clock cycle and then SecAND and SecHW are performed. This is implemented using a pipelined approach as shown in Fig. 3.

The left part of Fig. 3 illustrates the bitwise SecAND operation using Eq. (2). The 32-bit shares are fed into shared functions f_1, f_2, f_3, and the outputs are three 32-bit shares of the result. As mentioned before, two additional random vectors r_1, r_2 are required to mask the outputs in order to achieve uniformity. Our design uses only two fresh random bits b_1, b_2 together with the shifted input shares as the random vectors because the neighboring bits are independent of each other. That is $r_1 = \{b_1, m_{1,h_0}[0:30]\}$ and $r_2 = \{b_2, m_{2,h_0}[0:30]\}$. Both $m_{1,h_0}[31]$ and $m_{2,h_0}[31]$ are shifted out and are used as b_1 and b_2 in the next clock cycle. The right part shows the structure of SecHW. To compute the Hamming weight of the unmasked result $sh_1 \oplus sh_2 \oplus sh_3$ without leaking side channel information, a parallel counting algorithm is applied to accumulate the weight of each bit position of the word. We use 32×2 6-bit Arithmetic masked counters[1] and each bit in the word $sh_1 \oplus sh_2 \oplus sh_3$ will be added into the corresponding counter during each clock cycle. More specifically, the three shares of each bit of sh are converted and added into the two Arithmetic masked counters. After 150 clock cycles, we sum the overall Arithmetic masked Hamming weight. To convert and accumulate the masked weights, we employ the secure conversion method developed in [7].

[1] Note that the Hamming weight of $s \wedge H$ is bounded to the weight of h_i, i. e., $\text{wt}(s \wedge h_i) \leq w/2 = 45$, i. e., 6-bit registers are always sufficient.

5 Implementation Results

The masked design is implemented in VHDL and is synthesized for Xilinx Virtex-5 XC5VLX50 FPGA which holds the crypto engine in the side channel evaluation board SASEBO-GII. The implementation results are listed in Table 1 in comparison with the unprotected implementation of [19]. In terms of Flip-Flops (FFs) and Look-Up Tables (LUTs), the masked implementation uses 8 times as many resources as the unprotected implementation. The increase is mainly due to the masked Hamming weight computation which requires many registers to store the Hamming weights of small chunks. Moreover, the leap forward LFSR also utilizes many Flip-Flops and has to be instantiated five times in our design. The number of occupied BRAMs remains constant, only the occupied memory within the syndrome BRAM increases by a factor of 2 in the masked implementation because the syndrome masks are also stored in this BRAM. The performance of the masked design is compromised for security and the maximum clock frequency is reduced by a factor of 4.3. This is mainly because the addition of 32 6-bit weight registers in SecHW is done in one clock cycle resulting a long critical path and in turn a low clock frequency. Shortening the critical path can be an interesting goal in future work. Note that the number of clock cycles remains the same as for the unprotected implementation, unless the early termination of the decoder is disabled, in which case the average run time doubles compared to [19] (assuming that the maximum number of iterations is set to 5 similarly to [20], with early termination enabled the decoder requires 2.4 iterations on average as was shown in [10,21]). The resulting mean overhead of our implementation is 4, which is in line with other masked implementations[2]. The TI AES engine in [14] introduces an area overhead of a factor 4 as well, but that implementation does not include the pseudorandom generators needed to generate the 48 bits of randomness consumed per cycle, while ours does.

Table 1. Resources usage comparison between the unprotected and masked implementations on Xilinx Virtex-5 XC5VLX50 FPGAs.

Implementation	FFs	LUTs	Slices	BRAMs	Frequency
Unprotected [19]	412	568	148	3	318 MHz
Masked	3045	4672	1549	3	73 MHz
Overhead factor	7.4x	8.2x	10.5x	1x	4.3x

6 Leakage Analysis

Next we analyze the implementation for remaining leakage. We first apply the DPA presented in [6] on the protected implementation. Next we use the leakage detection methodology developed in [9] to detect any other potentially

[2] When computing the geometric mean of the overhead of the three hardware components (LUTs, FFs, and BRAMs), the resulting area overhead is actually 3.9.

exploitable leakages. The evaluated implementation is placed on the Xilinx Virtex-5 XC5VLX50 FPGA of the SASEBO-GII board. The power measurements are acquired using a Tektronix DSO 5104 oscilloscope. The board was clocked at 3 MHz and the sampling rate was set to 100 M samples per second. In order to quantify the resilience of our masked implementation to power analysis attacks, we collected $10,000$ measurements using the same ciphertext but two different sets of secret keys. The first set is actually $5,000$ repetitions of a fixed key while the second set contains $5,000$ random keys. The two sets of keys are fed into the decryption engine alternatingly.

6.1 Differential Power Analysis

A Differential Power Analysis on the FPGA implementation of QC-MDPC McEliece of [19] was presented in [6]. The attack exploits the leakage caused by the key rotation in the syndrome computation phase. The 4801-bit keys h_0 and h_1 rotate in parallel for 4801 rounds, each round lasts for 150 clock cycles. Thus during one decryption, each key bit is rotated into the one bit carry register 150 times which results in a strong leakage. By averaging the 150 leakage samples for each key bit, one can generate the 4801-sample differential trace which contains features caused by the set key bits and then one can recover the value of the key bits by interpreting the features. For the unprotected implementation, the secret key can be completely recovered using the average differential trace of only 10 measurements. For more details about the key recovery we refer to [6]. In contrast to the unprotected implementation, no features are present in the differential trace of the fixed secret key (red line) even with 500 times more traces, as shown in Fig. 4. Hence, the key bit value cannot be recovered. The peaks in the trace are not the features caused by set key bits because in the differential trace of the random secret keys where the key bits are randomly set as 1 the same peaks appear. Thus, they cannot be used as features to recover secret key bits as done in [6]. The two differential traces almost overlap, showing that the leakage is indistinguishable between fixed key and random key when using a masked implementation.

6.2 Leakage Detection

We employ Welch's T-test suite to quantify the leakage indistinguishability between two sets of secret keys. Welch's T-test is a statistical hypothesis test used to decide whether the means of two distributions are the same. T-statistic t can be computed as:

$$t = \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{{s_X}^2}{N_X} + \frac{{s_Y}^2}{N_Y}}} \tag{5}$$

where $\overline{X}, \overline{Y}$ are the sample means of random variables X, Y, s_X, s_Y are the sample variances and N_X, N_Y are the sample sizes. The pass fail criteria is

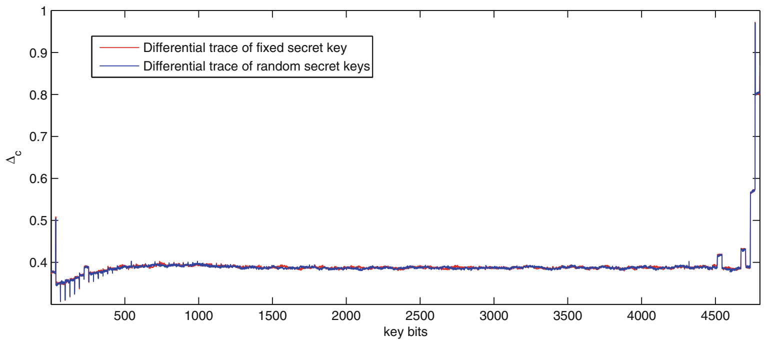


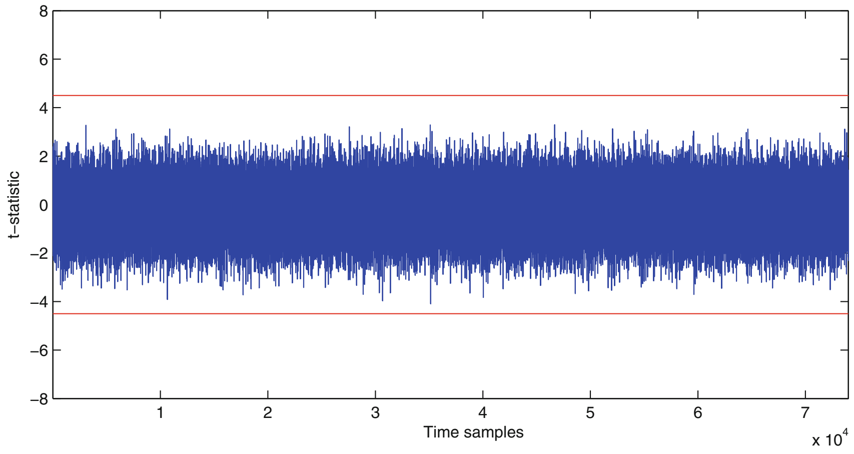
Fig. 4. Comparison between two differential traces of two sets of secret keys.

defined as [−4.5, 4.5] as developed in [9]. In our case, we obtained two groups of leakage samples, one for the fixed key set and the other for the random key set. Each group has 5,000 power traces as well as 5,000 derived differential traces. We first performed the T-test using the original power traces and Fig. 5.1 shows the t-statistics along the whole decryption. The t-statistics are within the range of [−4.5, 4.5] which implies a confidence of more than 99.999 % for the null hypothesis showing that the two sample groups are indistinguishable.

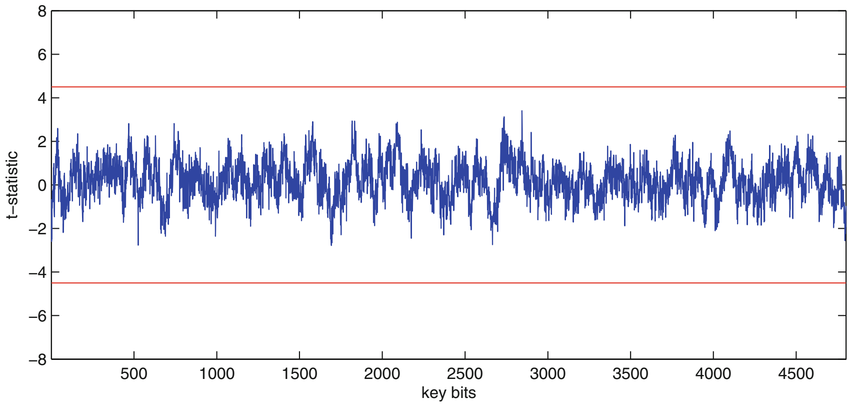
To assess the vulnerability to first-order horizontal attacks, we also performed a T-test on the derived differential traces. The results are shown in Fig. 5.2. Similarly, the t-statistics are also within the predefined range and it validates the indistinguishability between the two sets of secret keys. Hence, it can be concluded that the design does not contain any remaining first-order leakage of the key.

6.3 Masking the Ciphertext?

The decoder corrects errors in the ciphertext x, eventually yielding the plaintext derived value $m \cdot G$ and thereby implicitly the error vector e. Similarly, the values d and $\#_{\mathrm{upc}}$ assigned in line 3 and 4 of Algorithm 1 are not masked and can potentially reveal the error locations, hence e. In either case, the equivalent leakage of information of e or $m \cdot G$ is possible. In our implementation, we chose not to mask x and its intermediate state, nor d and $\#_{\mathrm{upc}}$. This choice is justifiable for two reasons. First, both e or $m \cdot G$ are key-independent and will not reveal information about the secret key. Furthermore, e or $m \cdot G$ are ciphertext dependent, that is, any information that can be revealed will be only valid for the specific encrypted message. Hence, if such information is to be discovered, it must be recovered using SPA-like approaches. More explicitly, the only possible attack is a message recovery attack, and that requires SPA techniques, as, e.g., applied in [20]. Nevertheless, d and $\#_{\mathrm{upc}}$ are variables that have dependence on both the ciphertext and the key, just as the number of decoding iterations that might be revealed by a non-constant time implementation, i. e., if the decoding algorithm



5.1: Results of original traces



5.2: Results of differential traces

Fig. 5. T-test between the two groups of *original* power traces (5.1) and *differential* power traces (5.2) corresponding to the two sets of secret keys. Both cases indicate the absence of leakage for the given number of traces.

tests the syndrome for zero after each decoding iteration and exits when this condition is reached. However, up to now there is no evidence suggesting that their information can be used to perform key recovery attacks. We leave this as an open question for future research.

7 Conclusion

This work presents the first masked implementation of a McEliece cryptosystem. While masking the syndrome computation is straightforward and comes at a low overhead, the decoding algorithm requires more involved masking techniques.

Through on-the-fly mask generation, the area overhead is limited to a factor of approximately 4. While the maximum clock frequency of the engine decreases, the number of clock cycles for the syndrome computation and each decoder run is unaffected by the countermeasures. The effectiveness of the applied masking has been analyzed by leakage detection methods and by showing that previous attacks do not succeed anymore. Exploring if any information about the secret key can be derived from the number of decoding iterations leaves an interesting challenge for future work.

Acknowledgments. This work is supported by the National Science Foundation under grant CNS-1261399 and grant CNS-1314770. IvM was supported by the European Union H2020 PQCrypto project (grant no. 645622) and the German Research Foundation (DFG). RS is supported by NATO's Public Diplomacy Division in the framework of "Science for Peace", Project MD.SFPP 984520.

References

1. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.: On the inherent intractability of certain coding problems (corresp.). IEEE Trans. Inf. Theor. **24**(3), 384–386 (1978)
2. Bilgin, B., Daemen, J., Nikov, V., Nikova, S., Rijmen, V., Van Assche, G.: Efficient and first-order DPA resistant implementations of Keccak. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 187–199. Springer, Heidelberg (2014)
3. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: A more efficient AES threshold implementation. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT. LNCS, vol. 8469, pp. 267–284. Springer, Heidelberg (2014)
4. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014, Part II. LNCS, vol. 8874, pp. 326–343. Springer, Heidelberg (2014)
5. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In: 36th IEEE Symposium on Security and Privacy (2015)
6. Chen, C., Eisenbarth, T., von Maurich, I., Steinwandt, R.: Differential power analysis of a McEliece cryptosystem. In: Malkin, T., et al. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 538–556. Springer, Heidelberg (2015). doi:10.1007/978-3-319-28166-7_26. Preprint http://eprint.iacr.org/2014/534
7. Coron, J.-S., Großschädl, J., Vadnala, P.K.: Secure conversion between boolean and arithmetic masking of any order. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 188–205. Springer, Heidelberg (2014)
8. Gallager, R.: Low-density Parity-check Codes. IRE Trans. Inf. Theor. **8**(1), 21–28 (1962)
9. Gilbert Goodwill, B.J., Jaffe, J., Rohatgi, P., et al.: A testing methodology for side-channel resistance validation. In: NIST Non-invasive Attack Testing Workshop (2011)
10. Heyse, S., von Maurich, I., Güneysu, T.: Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices. In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 273–292. Springer, Heidelberg (2013)

11. Huffman, W.C., Pless, V.: Fundamentals of Error-Correcting Codes. cambridge University Press, Cambridge (2010)
12. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. Deep Space Netw. Prog. Rep. **44**, 114–116 (1978)
13. Misoczki, R., Tillich, J.-P., Sendrier, N., Barreto, P.: MDPC-McEliece: new McEliece variants from moderate density parity-check codes. In: Proceedings of the IEEE International Symposium on Information Theory (ISIT), pp. 2069–2073. IEEE (2013)
14. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: a very compact and a threshold implementation of AES. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011)
15. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 529–545. Springer, Heidelberg (2006)
16. Schneider, T., Moradi, A., Güneysu, T.: Arithmetic addition over boolean masking. In: Malkin, T., et al. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 559–578. Springer, Heidelberg (2015). doi:10.1007/978-3-319-28166-7_27
17. Shahverdi, A., Taha, M., Eisenbarth, T.: Silent SIMON: a threshold implementation under 100 slices. In: Proceedings of IEEE Symposium on Hardware Oriented Security and Trust (HOST) (2015)
18. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
19. von Maurich, I., Güneysu, T.: Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices. In: Design, Automation and Test in Europe - DATE, pp. 1–6. IEEE (2014)
20. von Maurich, I., Güneysu, T.: Towards side-channel resistant implementations of QC-MDPC McEliece encryption on constrained devices. In: Mosca, M. (ed.) PQCrypto 2014. LNCS, vol. 8772, pp. 266–282. Springer, Heidelberg (2014)
21. von Maurich, I., Oder, T., Güneysu, T.: Implementing QC-MDPC McEliece encryption. ACM Trans. Embed. Comput. Syst. **14**(3), 44:1–44:27 (2015)