A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms

Abdulhadi Shoufan*, Thorsten Wink‡, Gregor Molter†, Sorin Huss† and Falko Strentzke§
*Center for Advanced Security Research Darmstadt CASED, Germany
Email: shoufan@cased.de
†Intergrated Circuits and Systems Labs, TU-Darmstadt, Germany
Email: {molter,huss}@iss.tu-darmstadt.de
‡Embedded Systems and Applications, TU-Darmstadt, Germany
Email: wink@esa.informatik.tu-darmstadt.de
§FlexSecure GmbH, Germany
Email: strenzke@flexsecure.de

Abstract

McEliece scheme represents a code-based public-key cryptosystem. So far, this cryptosystem was not employed because of efficiency questions regarding performance and communication overhead. This paper presents a novel processor architecture as a high-performance platform to execute key generation, encryption and decryption according to this cryptosystem. A prototype of this processor is realized on Virtex-5 FPGA and tested via a software API. A comparison with a similar software solution highlights the performance advantage of the proposed hardware solution.

Keywords: *Cryptography hardware and implementation, Cryptoprocessor, McEliece cryptosystem, Goppa code, FPGA.*

1. Introduction

Current public-key cryptosystems rely on the computational complexity of different mathematical problems such as the factoring of large primes in RSA [1] or the calculation of the discrete logarithm in ECC [2]. These algorithms are assumed to become insecure in the era of quantum computers [3]. Therefore, several solutions for the post-quantum cryptography have been proposed in literature such as hash-based, code-based, lattice-based, and multivariate-quadratic-equation-based cryptosystems, see [4]–[7]. Generally, these solutions suffer from efficiency problems regarding execution time and data and key sizes. To tackle the performance problems in hash-based and multivariate-quadratic-equation based approaches three hardware architectures have been proposed recently, see [8] and [9]. Apart from the solution in [10], which addresses digital signatures, no hardware solutions for code-based encryption/decryption are known, so far. This paper closes this gap by presenting a novel hardware architecture for the McEliece cryptosystem, which was proposed 1978 by R. McEliece [5] as one of the first public key cryptography systems. The proposed architecture, which supports key generation, encryption, and decryption, is implemented on a Virtex-5 platform and tested through a dedicated API.

The remainder of the paper is structured as follows. Section 2 provides an introduction into the McEliece cryptosystem. Section 3 provides the high-level architecture of the proposed cryptoprocessor. The next three sections detail the Key Generator, the Encryptor and the Decryptor as the main components of the cryptoprocessor. Section 7 provides some implementation aspects and the results. Section 8 concludes the paper.

2. McEliece Cryptosystem (MECS)

MECS is a highly complex system which relies on sophisticated Goppa code and composed finite field arithmetic. A thorough treatment of this system goes beyond the focus of this paper. This section provides an algorithmic description of MECS in a way that facilitates understanding the implementation aspects illustrated in following sections. Neither formal notations nor mathematical justifications come to the fore. Interested readers are referred to special literature on coding theory, e.g., [11], and cryptography, e.g., [12].

The relation between MECS and Goppa code [13] is illustrated as follows. Generally, a coding system includes three main functions: the generation of the code characteristics such as the generator and the control matrix, the encoding, and the decoding. Similarly, a public-key cryptosystem includes three operations: the generation of a pair of public key and private key, the encryption and the decryption. MECS uses the Goppa code characteristics as public and private keys, the Goppa encoding for encryption, and the Goppa decoding for decryption.

An essential aspect of code-based cryptography relates to the selection of the code parameters, as these parameters directly affect the security of the cryptosystem. In the case of MECS two parameters are relevant: the dimension of the code subspace m and the maximal number of correctable

$$\mathbf{X} = \underbrace{\begin{bmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{bmatrix}}_{t \times t \text{ matrix}}, \quad \mathbf{Y} = \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_0^{t-1} & \alpha_1^{t-1} & \cdots & \alpha_{n-1}^{t-1} \end{bmatrix}}_{t \times n \text{ matrix}}, \quad \mathbf{Z} = \underbrace{\begin{bmatrix} g(\alpha_0)^{-1} & 0 & \cdots & 0 \\ 0 & g(\alpha_1)^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g(\alpha_{n-1})^{-1} \end{bmatrix}}_{n \times n \text{ matrix}}$$

Figure 1. Auxiliary Matrices for the Control Matrix $\mathbf{H}$

errors t. Bernstein et. al. [14] showed that the selection of $m = 11$ and $t = 27$ ensures a security level of 80-bit under consideration of the strongest attack against MECS known today. For the proposed architecture we use $m = 11$ and $t = 50$, which guarantees considerably higher security level. An overview of all parameters is given in Table 1.

Similarly to other cryptosystems, a plain implementation of MECS is attackable by adaptive chosen-ciphertext attacks (CCA2). Therefore, additional steps are required during encryption and decryption to impede these attacks. A good overview of CCA2-safe conversions can be found in [15].

2.1. Key Generation

Algorithm 1 depicts the key generation in MECS. Based on the domain parameters m and t the code length n and its dimension k are determined and the basic finite field $\mathbb{GF}(2^m)$ is constructed. The first step in the key generation is to generate a monic, irreducible polynomial $g(x) = x^t + g_{t-1}x^{t-1} + \cdots + g_1 x + g_0$, which is denoted as *Goppa polynomial*. All coefficients of $g(x)$ are elements of $\mathbb{GF}(2^m)$. This polynomial is part of the private key which is kept secret.

Based on $g(x)$ and the field $\mathbb{GF}(2^m)$ the control matrix $\mathbf{H}$ is created. This step is performed using three auxiliary matrices $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$, as depicted in Fig. 1. Note that $g(\alpha_i)^{-1}$ indicates to the inverse of $g(\alpha_i)$ in $\mathbb{GF}(2^m)$, where $g(\alpha_i)$ results from the evaluation of $g(x)$ for the element α_i.

Following, the control matrix is permuted using a random matrix P, which is also kept secret as a part of the private key. Although the resulting matrix $\widetilde{\mathbf{H}}$ can now be used as a public key, this is inefficient because of the huge length of this key. The next steps, therefore, aim at producing a shorter public key. First $\widetilde{\mathbf{H}}$ is expanded to the binary form $\mathbf{H}_2$, which is converted into a systematic form $\widetilde{\mathbf{G}}$. Lastly, $\widetilde{\mathbf{G}}$ is transposed into $\mathbf{G}$ and its part $\mathbf{R}^T$ is returned as the public key, which has a length of $550 \cdot 1498$ bit instead of $1498 \cdot 2048$ in the case of $\widetilde{\mathbf{H}}$.

Algorithm 1 MECS Key Generation

Require: McEliece domain parameters m and t.
Let $n = 2^m$ and $k = n - mt$.
Elements of $\mathbb{GF}(2^m) = \{\alpha_0, \alpha_1, \ldots, \alpha_{n-1}\}$.
Ensure: The public key $\mathbf{R}^T$and the private key $(\mathbf{P}, g(x))$.
1: Create a random monic, irreducible polynomial $g(x)$ with $\deg(g) = t$ and $x \in \mathbb{GF}(2^m)$.
2: Create the auxiliary matrices $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$.
3: Calculate the $t \times n$ control matrix $\mathbf{H} = \mathbf{XYZ}$.
4: Create a random $n \times n$ permutation matrix $\mathbf{P}$.
5: Calculate the permutated control matrix $\widetilde{\mathbf{H}} = \mathbf{HP}^T$.
6: Transform the $t \times n$ matrix $\widetilde{\mathbf{H}}$ over $\mathbb{GF}(2^m)$ into a $mt \times n$ matrix $\mathbf{H}_2$ over $\mathbb{GF}(2)$.
7: Bring $\mathbf{H}_2$ into the systematic form $\widetilde{\mathbf{G}} = [\mathbb{I}_{mt}|\mathbf{R}]$.
8: The expanded public key is the $k \times n$ matrix over $\mathbb{GF}(2)$ $\mathbf{G} = \left[\mathbf{R}^T|\mathbb{I}_k\right]$.
9: **return** $\mathbf{R}^T$and $(\mathbf{P}, g(x))$

2.2. Encryption

Algorithm 2 depicts the encryption procedure in MECS. $\mathrm{hash}(x)$ describes a hash-function with output length l. Note that most steps of this algorithm serves the security against CCA2 based on a conversion similar to the Kobara-Imai scheme [15]. Without this conversion the ciphertext would be constructed simply as $z = m\mathbf{G} \oplus e$.

2.3. Decryption

Algorithm 3 presents the decryption process in MECS, which is clearly more complex than encryption. For efficient error correction, the *Patterson Algorithm* [16] is employed, which is included in Algorithm 3 from step 3 to 8. It

Algorithm 2 MECS Encryption

Require: l-bit plaintext m; public key $\mathbf{R}^T$.
Ensure: $(n + 2l)$-bit ciphertext z.
1: Generate a random n-bit error vector e with t bits having the value 1.
2: Expand the public key $\mathbf{R}^T$ to $\mathbf{G} = \left[\mathbf{R}^T|\mathbb{I}_k\right]$.
3: Generate a random $(k - l)$-bit vector r_1 and a random l-bit vector r_2.
4: Create CCA2-safe plaintext $\widetilde{m} = r_1 \parallel \mathrm{hash}(m \parallel r_2)$.
5: Encode $\widetilde{m}$ into $z' = \widetilde{m}\mathbf{G}$.
6: Imprint t errors to z' and add CCA2-safe data extension $z = (z' \oplus e) \parallel (\mathrm{hash}(r_1) \oplus m) \parallel (\mathrm{hash}(e) \oplus r_2)$.
7: **return** z.

Algorithm 3 MECS Decryption

Require: $(n + 2l)$-bit ciphertext z; private key $(\mathbf{P}, g(x))$.
Ensure: l-bit plaintext m if CCA2 test successful; otherwise an error message.

1: Split z to $(\underbrace{z_1}_{n \text{ bits}}, \underbrace{z_2}_{l \text{ bits}}, \underbrace{z_3}_{l \text{ bits}})$.
2: Permute z_1: $z' = z_1\mathbf{P}$.
3: Determine the syndrome polynomial $S_{z'}(x) = z'\mathbf{H}^T(x^{t-1}, \ldots, x, 1)^T$.
4: Invert $S_{z'}^{-1}(x)$.
5: Let $\tau(x) = \sqrt{S_{z'}^{-1}(x) + x}$.
6: Find two polynomials $a(x)$ and $b(x)$, so that $b(x)\tau(x) = a(x) \mod (g(x))$ and $\deg(a) \leqslant \lfloor \frac{t}{2} \rfloor$ hold.
7: Determine the error locator polynomial $\sigma(x) = a^2(x) + xb(x)^2$.
8: Reconstruct the error vector $e' = (\sigma(\alpha_0), \sigma(\alpha_1), \ldots, \sigma(\alpha_{n-1})) \oplus (1, \ldots, 1)$.
9: Permute the error vector $e = e'\mathbf{P}^T$.
10: Reconstruct the CCA2-safe plaintext $m' = z_1 \oplus e$.
11: Split m' to $(\underbrace{r}_{k-l \text{ bits}}, \underbrace{h}_{l \text{ bits}}, \underbrace{m''}_{n-k \text{ bits}})$.
12: Reconstruct plaintext candidate $m = z_2 \oplus \text{hash}(r)$.
13: Determine check value $h' = \text{hash}(m \parallel \text{hash}(e) \oplus z_3)$.
14: **if** $h' \equiv h$ **then**
15: **return** m
16: **end if**
17: **return** "Error"

creates the error locator polynomial $\sigma(x)$. By evaluating this polynomial for all the elements α_i of $\mathbb{GF}(2^m)$, all errors are revealed. If an error occurred at the i-th position of z', $\sigma(\alpha_i)$ is zero. Then, the recreated error vector is reapplied to the CCA2-safe ciphertext, decrypting it to the CCA2-safe plaintext. Afterwards, the CCA2-related padding is checked and removed. If the CCA2-safe padding data is valid, the plaintext is returned. Otherwise an error is signaled. Note that the transpose control matrix $\mathbf{H}^T$ is assumed to be saved after key generation. Alternatively, $\mathbf{H}^T$ can be determined based on $g(x)$, P, and the $\mathbb{GF}(2^m)$ elements.

Table 1 gives an overview of the most important parameters and data sizes of MECS as supported by our cryptoprocessor.

3. High-Level-Architecture

Fig. 2 depicts the overall architecture of the proposed cryptoprocessor, which consists of three main units: Key Generator, Encryptor and Decryptor. The cryptoprocessor operates as a coprocessor in a server environment and communicates with the host over a PCI bridge. The host writes a command into the command register of the cryptoprocessor and the relating data into the In FIFO. The Master Controller decodes this command and activates the unit responsible for its execution. Output data are written into the Out FIFO. The status register contains auxiliary information for exception cases such as the reception of erroneous opcode or a failed CCA2 test. The data path on this level has a word width of 64 bits which is enforced by the local bus on the hardware card. While the command and status registers are accessed as memory-mapped registers, the data transfer into and from FIFOs is accomplished via DMA for performance reasons. The cryptoprocessor executes mainly three commands relating to McEliece cryptosystem. These commands are summarized in Table 2. The data sizes for plain and ciphertexts in this table correspond to the CCA2 version implemented in this work. Note that in a public key cryptosystem for encryption and decryption, the public key generated by the server is never required by this server for further cryptographic operations. Thus, the cryptoprocessor returns the public key $\mathbf{R}^T(\mathrm{S})$ directly after generation. For encryption, in addition to the plaintext the server receives the public key of the client $\mathbf{R}^T(\mathrm{C})$ and forwards it to the cryptoprocessor as a parameter. In the case of decryption, the cryptoprocessor uses the server private key ($g(x)$, $\mathbf{P}$), which is generated and saved on the FPGA, to decrypt a ciphertext encrypted by the client using the server public key.

Note that all data sizes in Table 2 are given in terms of the word width of 64 bit. Recall that $\mathbf{R}^T$ is a 550×1498 binary matrix, which is not a multiple of 64 bit. To save memory and execution time, $\mathbf{R}^T$ is arranged as illustrated in Fig. 3. $\mathbf{R}^T$ is transferred column-wise between host and FPGA. This corresponds to the way this matrix is produced or consumed by the FPGA, as will be seen later. Note that in the case of encryption $\mathbf{R}^T(\mathrm{C})$ is not required to be transferred completely to start encryption. Thus, transfer and encryption proceed in parallel.

Besides its mathematical complexity, the McEliece cryptosystem raises implementation difficulties as it operates on three different fields simultaneously:

1) The Goppa field $\mathbb{GF}((2^{11})^{50})$ with Goppa polynomial $g(x)$ as the generator polynomial.

Parameter	Meaning	Size (Bit)
m	Degree of the extension field	11
t	Number of correctable errors	50
n	Code length $n = 2^m$	2048
k	Code rank $k = n - mt$	1498
l	Hash value of SHA-512	512
m	Plaintext	512
z	Ciphertext	3072
$\mathbf{R}^T$	Public key	1498×550
$\mathbf{P}$	Permutation matrix	2048×11
$g(x)$	Goppa polynomial	51×11
$\mathbf{H}$	Control matrix	$(50 \times 11) \times 2048$

Table 1. Parameters of McEliece

Instruction	Input	Output
Generate Key	*none*	*Server public key* $\mathbf{R}^T(S)$: $(9 \cdot 64) \cdot 1498 = 862,848$ bit
Encrypt	*Plaintext*: $8 \cdot 64 = 512$ bit, *Client public key* $\mathbf{R}^T(C)$: $(9 \cdot 64) \cdot 1498 = 862,848$ bit	*Ciphertext*: $48 \cdot 64 = 3072$ bit
Decrypt	*Ciphertext*: $48 \cdot 64 = 3072$ bit	*Plaintext*: $8 \cdot 64 = 512$ bit

Table 2. Cryptoprocessor Instruction Set

2) $\mathbb{GF}(2^{11})$ with the polynomial $p(x) = x^{11} + x^2 + 1$ specified in IEEE Standard Specification for Public-Key Cryptography [17].
3) Binary field $\mathbb{GF}(2)$.

Because of place reasons, the following sections describe the cryptoprocessor main modules on a high level of abstraction without detailing the underlying implementations of cryptographic arithmetic. For details on that subject we refer to [12].

4. Key Generator

Fig. 4 shows the Key Generator (KG) with three types of blocks: functional units represented as squares, single-port memories (SBRAM) represented as rectangles, and dual-port memories (DBRAM) represented as rectangles with splitting neck in the middle. Regardless of the 64-bit words of the PRNG and OUT memory, the data path of the KG is of width 11 bit. Obviously, that does not mean that all functional units of KG work on $\mathbb{GF}(2^{11})$.

Another important point relates to memory sharing between the Key Generator and the Decryptor. Remember that all the control matrix $\mathbf{H}$, the permutation matrix $\mathbf{P}$ and Goppa polynomial $g(x)$ are required during decryption. Thus the corresponding memories in Fig. 4 have all interfaces with

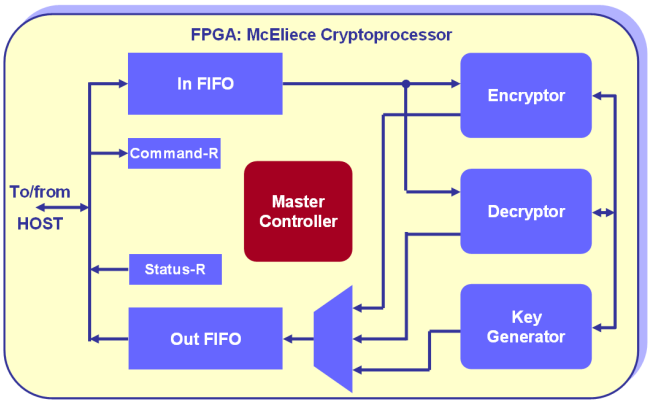

Figure 2. Overall Architecture of Proposed McEliece Cryptoprocessor

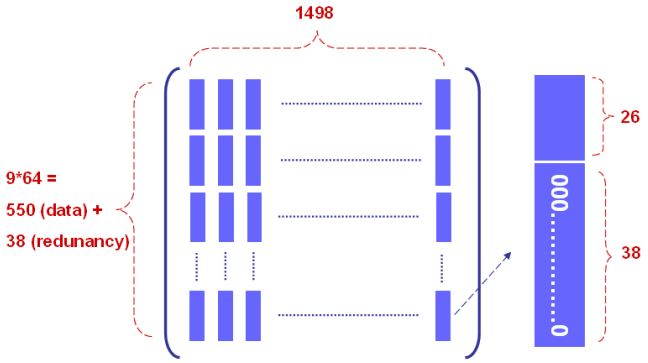

Figure 3. Transfer Format of the Public Key $\mathbf{R}^T$

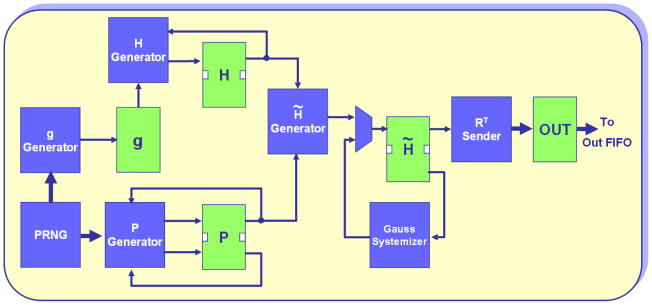

Figure 4. Key Generator

the Decryptor. These interfaces, however, are not depicted in Fig. 4, for simplicity.

4.1. Pseudo Random Number Generator

For all random numbers in the key generation, encryption and decryption a PRNG is used which is based on the hash function SHA-512. Thus the generated random numbers are all of 512-bit width. These numbers, however, are delivered to invoking modules in 64-bit words to avoid timing problems. Initiated by a start seed S_0 a random number $R_i = \text{hash}(S_i)$ and a next seed $S_{i+1} = S_i + R_i + 1$ are determined.

4.2. g Generator and g Memory

This module generates a monic, irreducible polynomial $g(x)$ of order 50 in two steps. First, 50 11-bit random numbers are generated, i.e. a total of 550 bits are required. For this purpose, the PRNG is utilized to provide two 512-bit random numbers. Following, an irreducibility test is started based on the IEEE Standard Specification for Public-Key Cryptography [17]. The time needed to generate $g(x)$ depends on the generated random numbers and is random, therefore. Extensive measurements have shown that 32 iterations are required at an average in order to obtain an

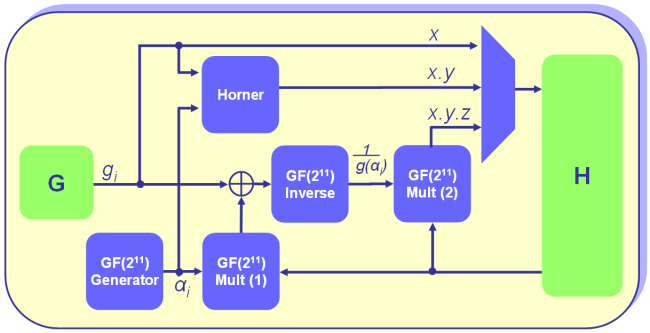

Figure 5. H Generator

irreducible polynomial. Irreducibility test operates on Goppa field. This operation includes both polynomial squaring and the determination of the greatest common divisor gcd. While squaring in Goppa field is relatively straightforward, determining the gcd is highly complex. For this purpose the Extended Euclidean Algorithm (XGCD) is used. To save $g(x)$ a dedicated Block RAM on the Virtex-5 device is used, whereas this selection is not mandatory and justified by the availability of free BRAMs.

4.3. H Generator and H Memory

Generating the control matrix $\mathbf{H}$ is the most time-consuming step in the key generation because of the huge number of operations in Goppa field and in $\mathbb{GF}(2^n)$. This is detailed as follows:

1) Constructing the matrix $\mathbf{Y}$: all the 2048 elements $\alpha_i, i \in \{0, \ldots, 2047\}$ of the field $\mathbb{GF}(2^{11})$ must be generated. Furthermore, for each element α_i all the powers from α_i^2 to α_i^{49} must be calculated.
2) Constructing the matrix $\mathbf{Z}$: 2048 evaluations of the Goppa polynomial and 2048 polynomial inversions in $\mathbb{GF}(2^{11})$ are required.
3) Calculating $\mathbf{XY}$: $25 \cdot 2048 \cdot 50 = 2,560,000$ modular multiplications in $\mathbb{GF}(2^{11})$ are required.
4) Calculating $(\mathbf{XY})\mathbf{Z}$: $50 \cdot 2048 = 102,400$ modular multiplications in $\mathbb{GF}(2^{11})$ are required.

Fig. 5 illustrates the general architecture of the $\mathbf{H}$ Generator, which operates as follows: To determine the product $\mathbf{XY}$ we observe that this multiplication amounts to a partial evaluation of Goppa polynomial. This point is illustrated in Fig. 6.The element $g_1 + g_2\alpha_0 + g_3\alpha_0^2$, for instance, can be seen as the evaluation of the polynomial $g_1 + g_2x + g_3x^2$ for the value α_0. Polynomial evaluation can be performed efficiently using the Horner schemes which can be written as follows for our example: $g_1 + g_2\alpha_0 + g_3\alpha_0^2 = g_1 + \alpha_0(g_2 + \alpha_0 g_3)$. $\mathbf{XY}$ is determined row-wise using a new element of $\mathbb{GF}(2^{11})$ for each row element of $\mathbf{XY}$. The $\mathbb{GF}(2^{11})$ Generator, see Fig. 5, produces the field elements sequentially one each clock cycle. Therefore, no storage of these elements is required. Instead they are generated at runtime when needed. The product $(\mathbf{XY})\mathbf{Z}$ is determined column-wise starting from the lowest row of $\mathbf{XY}$. Consider the example shown in Fig. 7. At the beginning, the third row of $\mathbf{XY}$ is multiplied by the first column of $\mathbf{Z}$. That includes the following steps:

1) $g(\alpha_0)$ is calculated by multiplying $g_1 + g_2\alpha_0 + g_3\alpha_0^2$, which is already available in $\mathbf{H}$, by α_0 and the result is added to g_0. The multiplication is performed in $\mathbb{GF}(2^{11})$ Mult(1) and the addition by the xor, see Fig. 5.
2) $g(\alpha_0)$ is inverted using $\mathbb{GF}(2^{11})$ Inverse.
3) $\frac{1}{g(\alpha_0)}$ is multiplied by $g_1 + g_2a_0 + g_3a_0^2$ using $\mathbb{GF}(2^{11})$ Mult(2).

The memory $\mathbf{H}$ has a word width of 11 bit and a depth of $t \cdot n = 50 \cdot 2048$. Thus, a 17-bit address bus is required. A total of 32 Block RAMs in Virtex-5 are used to save $\mathbf{H}$.

4.4. P Generator and P Memory

The private permutation matrix $\mathbf{P}$ has a size of 2048×2048 bit. To save memory we don't save entire rows or columns. Instead, the position of the 1 in each row is saved. By this means, $\mathbf{P}$ is established as a 2048 11-bit memory, which is realized by one BRAM on Virtex-5. For an efficient generation of $\mathbf{P}$ dual-port BRAM is used. First the BRAM is initialized by the identity matrix through port A. Following, the address of port A is incremented from 0 to 2047. For each of these address values a random value for the address of port B is generated and the content of the cells addressed by A and B are exchanged.

4.5. $\widetilde{\mathbf{H}}$ Generator and $\widetilde{\mathbf{H}}$ Memory

$\widetilde{\mathbf{H}}$ results from $\mathbf{H}$ through permuting the elements of each row according to the permutation matrix $\mathbf{P}$, see Algorithm 1. Thus, $\widetilde{\mathbf{H}}$ Generator mainly performs memory access and control tasks. Understandably, $\widetilde{\mathbf{H}}$ Memory has the same size and organization as $\mathbf{H}$ Memory. In contrast to $\mathbf{H}$ Memory, which keeps the control matrix for decryption purposes, $\widetilde{\mathbf{H}}$ Memory represents a temporary data container for $\widetilde{\mathbf{H}}$, $\mathbf{H}_2$, $\widetilde{\mathbf{G}}$, and $\mathbf{G}$, see Algorithm 1. The public key $\mathbf{R}^T$is transferred to the host after generation and need not to be saved permanently on hardware.

4.6. Gauss Systemizer

In order to reduce the size of the public key, Gauss Systemizer (GS) applies the Gauss-Jordan algorithm to convert $\mathbf{H}_2$ into the systematic form $[\mathbb{I}_{mt}|\mathbf{R}]$, see step 7 in Algorithm 1. Note that step 6 in Algorithm 1 does not cause any expense in our implementation since hardware can interpret any element of $\widetilde{\mathbf{H}}$ as a binary vector, i.e. as

$$\mathbf{XY} = \begin{pmatrix} g_3 & 0 & 0 \\ g_2 & g_3 & 0 \\ g_1 & g_2 & g_3 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ \alpha_0 & \alpha_1 & \alpha_2 \\ \alpha_0^2 & \alpha_1^2 & \alpha_2^2 \end{pmatrix} = \begin{pmatrix} g_3 & g_3 & g_3 \\ g_2 + g_3\alpha_0 & g_2 + g_3\alpha_1 & g_2 + g_3\alpha_2 \\ g_1 + g_2\alpha_0 + g_3\alpha_0^2 & g_1 + g_2\alpha_1 + g_3\alpha_1^2 & g_1 + g_2\alpha_2 + g_3\alpha_2^2 \end{pmatrix}$$

Figure 6. Generation of the matrix $\mathbf{H}$ (Part 1)

$$\mathbf{H} = (\mathbf{XY})\mathbf{Z} \begin{pmatrix} g_3 & g_3 & g_3 \\ g_2 + g_3\alpha_0 & g_2 + g_3\alpha_1 & g_2 + g_3\alpha_2 \\ g_1 + g_2\alpha_0 + g_3\alpha_0^2 & g_1 + g_2\alpha_1 + g_3\alpha_1^2 & g_1 + g_2\alpha_2 + g_3\alpha_2^2 \end{pmatrix} \begin{pmatrix} \frac{1}{g(\alpha_0)} & 0 & 0 \\ 0 & \frac{1}{g(\alpha_1)} & 0 \\ 0 & 0 & \frac{1}{g(\alpha_2)} \end{pmatrix}$$

Figure 7. Generation of the matrix $\mathbf{H}$ (Part 2)

an element of $\mathbf{H}_2$. Therefore, operations performed by GS amount to simple row permutation and XOR addition of two rows. The Gauss Systemizer proceeds in two phases: First, the front part of the matrix is converted into a triangular matrix with 1's on the diagonal. Then, this triangular matrix is transformed into the identity matrix. To perform this task efficiently, we used an approach based on systolic arrays which is similar to the proposal in [18].

Systemizing the public key is the second most time-consuming operation in the Key Generator. A special problem of this operation relates to its infeasibility, when the first 550 columns of $\widetilde{\mathbf{H}}$ are linearly dependent. In this case, a new permutation matrix $\mathbf{P}$ must be generated and $\widetilde{\mathbf{H}}$ must be regenerated. Our measurements showed that four iterations are needed in average in order to complete the matrix systemization successfully.

5. Encryptor

Fig. 8 depicts the architecture of the Encryptor schematically. While the continuous lines relates to the original encryption algorithm, the dotted lines highlight the augmented paths needed in the CCA2 variant as described in Algorithm 2. The following description of the Encryptor relates to the CCA2 variant. Note that only one SHA-512 module is used in the Encryptor. This module is replicated in Fig. 8 for clarity. The PRNG was described in Section 4.1. The e-Generator provides a random 2048-bit vector of weight 50. For this purpose the vector is first initialized with 50 logical 1's at predefined positions and then 2048 permutations are executed in a similar way to the generation of the $\mathbf{P}$ matrix. The dual-port temporary block RAM enables executing different tasks in parallel. The most important part of the Encryptor is the Encoder which performs the actual coding: $z' = \widetilde{m}\mathbf{G}$. This operation, however, is highly efficient as it is realized using 64 2-AND gates and an XOR tree. Two main processes control the Encryptor. The first process is responsible for generating the random numbers r_1, r_2 and e and for the interaction with the hash module. The second process is responsible for the main functionalities. This includes the read of the 512-bit plaintext m and its storage in TEMP BRAM. Additionally, this process reads the public key $\mathbf{R}^T$ column-wise from the In FIFO and uses it directly in the encoder without buffering.

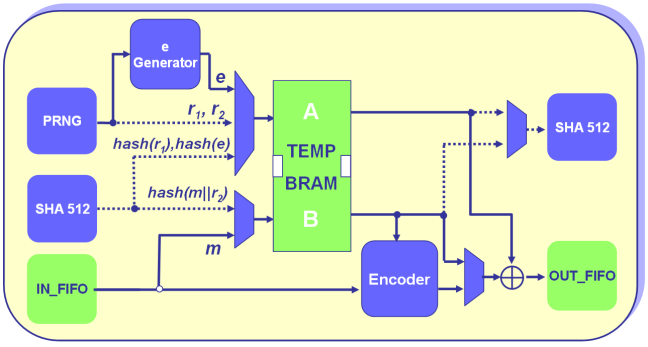

Figure 8. Encoder Architecture

6. Decryptor

Fig. 9 depicts the architecture of the Decryptor schematically. First the ciphertext z is read from the In FIFO and buffered into the temporary memory TEMP BRAM.

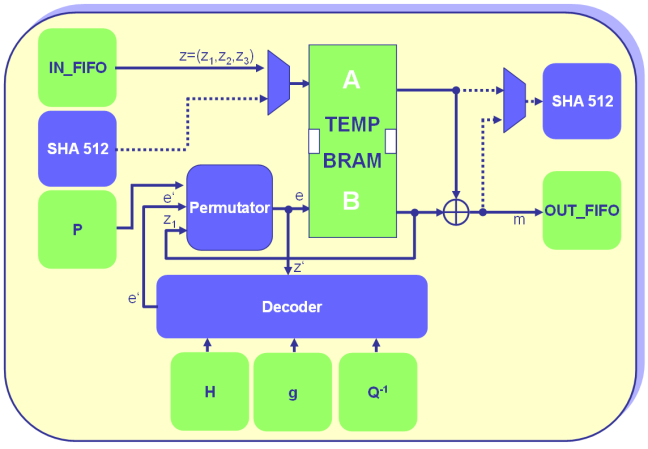

Figure 9. Decryptor Architecture

Next z_1 is permutated using the permutation matrix $\mathbf{P}$. The resulting vector $z^{'}$ is sent to the Decoder which determines the $e^{'}$ by applying the Patterson Algorithm. This begins with multiplying $z^{'}$ by the transposed control matrix $\mathbf{H}^T$ to determine the syndrome $S_{z'}$. This multiplication is realized by AND and XOR functions, as the $z^{'}$ is a binary vector. Note that $S_{z'}$ is a polynomial of order 49 over $\mathbb{GF}(2^{11})$. Following, the syndrome is inverted by extended Euclidean Algorithm XGCD. In a next step the square root of the polynomial $S^{-1}_{z'}(x) + x$ is determined. For this purpose this polynomial is multiplied by a pre-constructed squaring matrix Q^{-1} [12]. This matrix is established during key generation for performance reasons. The root square polynomial $\tau(x)$ is then decomposed into two polynomials $a(x)$ and $b(x)$, where $b(x)\tau(x) = a(x) \mod g(x)$, where $a(x)$ and $b(x)$ have a degree of at most 25. $a(x)$ and $b(x)$ are determined using the XGCD which proceeds until the desired degree of $a(x)$ and $b(x)$ is reached. Then, the error locator polynomial $\sigma(x) = a^2(x) + xb^2(x)$ is determined. This operation is highly efficient in hardware as it amounts mainly to squaring coefficients from $\mathbb{GF}(2^{11})$ which is realized by shift operations. The error vector $e^{'}$ is, then, constructed by evaluating the locator polynomial $\sigma(x)$ for all elements of $\mathbb{GF}(2^{11})$. A bit in $e^{'}$ is set to 1, when the evaluation of $\sigma(x)$ for the corresponding element of $\mathbb{GF}(2^{11})$ resulted in 0. For the evaluation of $\sigma(x)$ the Horner scheme addressed previously is used. After getting $e^{'}$ the Decryptor permutes it. Remember that the Encryptor uses a permuted key, see step 5 of Algorithm 1. Upon obtaining e, some simple calculations and hash operations are perfomed to determine m and check its correctness against adaptive chosen-ciphertext attacks. Only if this check succeeds, m is written into the Out FIFO.

7. Implementation and Results

The presented cryptoprocessor was implemented on the FPGA platform ADM-XRC-5T1 from Alpha Data Inc. [19]. This PCI card is equipped by the device LX110T of the Virtex-5 family from Xilinx Inc. [20]. Table 3 depicts the resource usage on the FPGA for the entire cryptoprocessor in terms of utilized slices and block RAMs. Despite the high usage of FPGA slices, which shows the complexity of the McEliece cryptosystem, this table illustrates the feasibility of the implementation of this system on today's FPGAs. For performance measurement we developed a software API based on the software development kit (SDK) from

	Available	Utilized	%
Slices	17,280	14,537	84
36 Kbit BRAMs	148	75	50

Table 3. Device Utilisation

	FPGA	Software	Acceleration Factor
Key Generation	143 ms	5,500 ms	38
Encryption	0.5 ms	10 ms	20
Decryption	1.4 ms	54 ms	39

Table 4. Performance FPGA Compared to Software

Alpha Data and tested the FPGA implementation thoroughly. Table 4 shows the performance figures for key generation, encryption, and decryption. Note that the timing figures in Table 4 are overal values, which include, besides the actual processing on hardware, the communication with the FPGA card and the data management on the host. For accuracy, each operation is executed 10,000 times and the avarage timing value is determined. This procedure is especially important for the key generation as its execution time depends on a random number of iterations to generate $g(x)$ and to apply Gauss-Jordan systemization successfully. Note, furthermore, that the encryption time of ca. 0.5 ms is restricted by the host system (32 Bit) and the PCI bus (66 MHz) capability to transfer the 1498×550-bit public key. The encryption itself takes on hardware about 0.1 ms. The FPGA was clocked with 163 MHz which was enforced by complexity of the design and the large resource usage on the FPGA. To have an overview of the advantage of this implementation we developed a C++ realization of the McEliece cryptosystem with the same security parameters t = 50 and m = 11, compiled it for Linux using gcc-4.2.4 and run it on an Intel Core Duo T7300, 2 GHz with 2GB RAM. The performance figures of this implementation are also depicted in Table 4. In despite of its first prototypical architecture, the proposed cryptoprocessor enables considerably faster key generation, encryption and decryption than a comparable software implementation. In this regard, not only the feasibility of hardware solutions for sophisticated code-based cryptography is shown, but also their high-performance which is indispensable for their acceptance.

8. Conclusion

A novel hardware architecture for McEliece cryptosystem was presented, which shows the advantage of modern FPGAs to answer performance questions regarding post quantum computer cryptography. In despite of its acceptable features our prototypical cryptoprocessor will undergo an optimization process regarding both resource usage and performance. For instance, in the current version two hash modules are used, one in the PRNG and the other for CCA2. In future, these will be reduced to only one with corresponding rescheduling to avoid performance overhead.

Acknowledgement

This project was funded by the German Federal Office for Information Security (*BSI*).

References

[1] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, 1978.

[2] N. Koblitz, "Elleptic Curve Cryptosystems," *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[3] Peter W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," *Proceedings, 35-th Annual Symposium on Foundation of Computer Science*, 1994.

[4] R. Merkle, "A certified digital signature," *CRYPTO 89: Proceedings on Advances in Cryptology*, 1989.

[5] R. J. McEliece, "A Public Key Cryptosystem Based on Algebraic Coding Theory," *DSN Progress Report*, vol. 42–44, pp. 114–116, 1978.

[6] J. L. A.K. Lenstra and L. Lovasz, "Factoring polynomials with rational coefficients," *Math*, pp. 515–534, 1982.

[7] H. Fell and W. Diffie, "Analysis of apublic key approach based on polynomial substitution," *LNCS on Advances in Cryptology-CRYPTO'85*, 1986.

[8] A. Shoufan, S. A. Huss, O. Kelm, and S. Schipp, "A Novel Rekeying Message Authentication Procedure based on Winternitz OTS and Reconfigurable Hardware Architectures," *ReConFig 2008*, 2008.

[9] S. Balasubramanian et. al., "Fast Multivariate Signature Generation in Hardware: The Case of Rainbow," *19th IEEE Int. Conf. on Application-specific Systems, Architectures and Processors ASAP 2008*, 2008.

[10] J.-C. Beuchat, N. Sendrier, A. Tisserand, and G. Villard, "FPGA Implementation of a recently published signature scheme," *Rapport de recherche RR LIP 2004-14*, 2004.

[11] S. Lin, *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.

[12] F. Rodriguez-Henriques, N. Saqib, A. Perez, and C. Koc, *Cryptographic Algorithms on Reconfigurable Hardware*. Springer, 2006.

[13] V. D. Goppa, "A new class of linear correcting codes," *Problems of Information Transmission*, vol. 6, pp. 207–212, 1970.

[14] D. J. Bernstein, T. Lange, and C. Peters, "Attacking and defending the McEliece cryptosystem," *Post-Quantum Cryptography, LNCS*, vol. 5299, pp. 31–46, 2008.

[15] K. Kobara and H. Imai, "Semantically Secure McEliece Public-Key Cryptosystems-Conversions for McEliece PKC," *Lecture Notes in Computer Science*, pp. 19–35, 2001.

[16] N. Patterson, "Algebraic decoding of Goppa Codes," *IEEE Transactions Information Theory*, vol. 21, pp. 203–207, 1975.

[17] The Institute of Electrical and Electronics Engineers, *IEEE Standard Specifications for Public-Key Cryptography*, 2000.

[18] B. Hochet, P. Quinton, and Y. Robert, "Systolic Gaussian Elimination over $\mathbb{GF}(p)$ with Partial Pivoting," *IEEE Transaction on Computers, Vol 39*, 1989.

[19] "Alpha-Data," http://www.alpha-data.com.

[20] "Xilinx," http://www.xilinx.com.