MicroEliece: McEliece for Embedded Devices

Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar

Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
{eisenbarth,gueneysu,heyse,cpaar}@crypto.rub.de

Abstract. Most advanced security systems rely on public-key schemes based either on the factorization or the discrete logarithm problem. Since both problems are known to be closely related, a major breakthrough in cryptanalysis tackling one of those problems could render a large set of cryptosystems completely useless. The McEliece public-key scheme is based on the alternative security assumption that decoding unknown linear binary codes is NP-complete. In this work, we investigate the efficient implementation of the McEliece scheme on embedded systems what was – up to date – considered a challenge due to the required storage of its large keys. To the best of our knowledge, this is the first time that the McEliece encryption scheme is implemented on a low-cost 8-bit AVR microprocessor and a Xilinx Spartan-3AN FPGA.

1 Introduction

The advanced properties of public-key cryptosystems are required for many cryptographic issues, such as key establishment between parties and digital signatures. In this context, RSA, ElGamal, and later ECC have evolved as most popular choices and build the foundation for virtually all practical security protocols and implementations with requirements for public-key cryptography. However, these cryptosystems rely on two primitive security assumptions, namely the factoring problem (FP) and the discrete logarithm problem (DLP), which are also known to be closely related. With a significant breakthrough in cryptanalysis or a major improvement of the best known attacks on these problems (i.e., the *Number Field Sieve* or *Index Calculus*), a large number of recently employed cryptosystems may turn out to be insecure overnight. Already the existence of a quantum computer that can provide computations on a few thousand qubits would render FP and DLP-based cryptography useless. Though quantum computers of that dimension have not been reported to be built yet, we already want to encourage a larger *diversification* of cryptographic primitives in future public-key systems. However, to be accepted as real alternatives to conventional systems like RSA and ECC, such security primitives need to support efficient implementations with a comparable level of security on recent computing platforms. For example, one promising alternative are public-key schemes based on Multivariate Quadratic (MQ) polynomials for which hardware implementations were proposed on CHES 2008 [11].

In this work, we demonstrate the efficient implementation of another public-key cryptosystem proposed by Robert J. McEliece in 1978 that is based on coding theory [22]. The McEliece cryptosystem incorporates a linear error-correcting code (namely a Goppa code) which is hidden as a general linear code. For Goppa codes, fast decoding algorithms exist when the code is known, but decoding codewords without knowledge of the coding scheme is proven NP-complete [5]. Contrary to DLP and FP-based systems, this makes this scheme also suitable for post-quantum era since it will remain unbroken when appropriately chosen security parameters are used [8].

The vast majority[1] of today's computing platforms are embedded systems. Only a few years ago, most of these devices could only provide a few hundred bytes of RAM and ROM which was a tight restriction for application (and security) designers. Thus, the McEliece scheme was regarded impracticable on such small and embedded systems due to the large size of the private and public keys. But nowadays, recent families of microcontrollers provide several hundreds of bytes of Flash-ROM. Moreover, recent off-the-shelf hardware such as FPGAs also contain dedicated memory blocks and Flash memories that support on-chip storage of up to a few megabits of data. In particular, these memories can be used to store the keys of the McEliece cryptosystem.

In this work, we present first implementations of the McEliece cryptosystem on a popular 8-bit AVR microcontroller, namely the ATxMega192, and a Xilinx Spartan-3AN 1400 FPGA which are both suitable for many embedded system applications. To the best of our knowledge, no implementations for the McEliece scheme have been proposed targeting embedded platforms. Fundamental operations for McEliece are based on encoding and decoding binary linear codes in binary extension fields that, in particular, can be implemented very efficiently in dedicated hardware. Unlike FP and DLP-based cryptosystems, operations on binary codes do not require computationally expensive multi-precision integer arithmetic what is beneficial for small computing platforms.

This paper is structured as follows: we start with a brief introduction to McEliece encryption and shortly explain necessary operations on Goppa codes. In Section 4, we discuss requirements and strategies to implement McEliece on memory-constrained embedded devices. Section 5 and Section 6 describe our actual implementations for an AVR 8-bit microprocessor and a Xilinx Spartan-3AN FPGA. Finally, we present our results for these platforms in Section 7.

2 Previous Work

Although invented already more than 30 years ago, the McEliece encryption scheme has never gained much attention due to its large keys and thus has not been implemented in many products. The most recent implementation of the McEliece scheme is due to Biswas and Sendrier [10] and presented a slightly modified version for PCs that achieves about 83 bit security (taken the attack in

[1] Already in 2002, 98% of 32-bit microprocessors in world-wide production were integrated in embedded platforms.

Algorithm 1. McEliece Message Encryption

```
Input: $m, K_{pub} = (\hat{G}, t)$
Output: Ciphertext $c$
1: Encode the message $m$ as a binary string of length $k$
2: $c‘ \leftarrow m \cdot \hat{G}$
3: Generate a random $n$-bit error vector $z$ containing at most $t$ ones
4: $c = c‘ + z$
5: return $c$
```

Algorithm 2. McEliece Message Decryption

```
Input: $c, K_{sec} = (P^{-1}, G, S^{-1})$
Output: Plaintext $m$
1: $\hat{c} \leftarrow c \cdot P^{-1}$
2: Use a decoding algorithm for the code $C$ to decode $\hat{c}$ to $\hat{m} = m \cdot S$
3: $m \leftarrow \hat{m} \cdot S^{-1}$
4: return $m$
```

[8] into account). Comparing their implementation to other public key schemes, it turns out that McEliece encryption can even be faster than that of RSA and NTRU [7]. In addition to that, only few further McEliece software implementations have been published up to now and they were all designed for 32 bit architectures [25,26]. The more recent implementation [26] is available only as uncommented C-source code and was nevertheless used for the open-source P2P software Freenet and Entropy [15].

Hardware implementations of the original McEliece cryptosystem do not exist, except for a proof-of-concept McEliece-based signature scheme that was designed for a Xilinx Virtex-E FPGA [9]. Hence, we here present the first FPGA-based hardware and 8-bit software implementation of the McEliece public-key encryption scheme up to date.

3 Background on the McEliece Cryptosystem

The McEliece scheme is a public key cryptosystem based on linear error-correcting codes. The secret key is the generator matrix G of an error-correcting code with dimension k, length n and error correcting capability t. To create a public key, McEliece defined a random $k \times k$-dimensional scrambling matrix S and $n \times n$-dimensional permutation matrix P disguising the structure of the code by computing the product $\hat{G} = S \times G \times P$. Using the public key $K_{pub} = (\hat{G}, t)$ and private key $K_{sec} = (P^{-1}, G, S^{-1})$, encryption and decryption algorithms can be given by Algorithm 1 and Algorithm 2, respectively.

Note that Algorithm 1 only consists of a simple matrix multiplication with the input message and then distributes t random errors on the resulting code word. Thus, the generation of random error vectors requires an appropriate random number generator to be available on the target platform.

Decoding the ciphertext c for decryption as shown in Algorithm 2 is the most time-consuming process and requires several more complex operations in binary extension fields. In Section 3.1 we briefly introduce the required steps for decoding codewords that we need to implement on embedded systems.

As mentioned in the introduction, the main caveat against the McEliece cryptosystem is the significant size of the public and private key. The choice of even a minimal set of security parameters ($m = 10, n = 1024, t = 38, k \geq 644$) according to [23] already translates to a size of 80.5 kByte for the public key and at least 53 kByte for the private key (without any optimizations). However, this setup only provides the comparable security of a 60 bit symmetric cipher. For appropriate 80 bit security, even larger keys, for example the parameters $m = 11, n = 2048, t = 27, k \geq 1751$, are required (more details in Section 3.2).

Many optimizations (cf. Section 4.2) of the original McEliece scheme focus on size reduction of the public key, since the public-key has to be distributed. Hence, a size reduction of K_{pub} is directly beneficial for *all* parties. However, the situation is different when implementing McEliece on embedded platforms: note that the private key must be kept secret at all times and thus should be stored in a protected location on the device (that may be used in a potentially untrustworthy environment). An effective approach for secret key protection is the use of secure on-chip key memories that would require (with appropriate security features such as prohibited memory readback) invasive attacks on the chip to reveal the key. However, secure storage of key bits usually prove costly in hardware so that effective strategies are required to reduce the size of the private key to keep costs low. Addressing this issue, we demonstrate for the first time how to use on-the-fly generation of the large scrambling matrix S^{-1} for the McEliece instead of storing it in memory as in previous implementations. More details on the reduction of the key size are given in Section 4.2.

3.1 Classical Goppa Codes

Theorem 1. *Let $G(z)$ be an irreducible polynomial of degree t over $GF(2^m)$. Then the set*

$$\Gamma(G(z), GF(2^m)) = \{(c_\alpha)_{\alpha \in GF(2^m)} \in \{0,1\}^n \mid \sum_{\alpha \in GF(2^m)} \frac{c_\alpha}{z-\alpha} \equiv 0\} \qquad (1)$$

defines a binary Goppa code C of length $n = 2^m$, dimension $k \geq n - mt$ and minimum distance $d \geq 2t+1$. The set of the c_α is called the support of the code. A fast decoding algorithm exists with a runtime of $n \cdot t$.

For each irreducible polynomial $G(z)$ over $GF(2^m)$ of degree t exists a binary Goppa code of length $n = 2^m$ and dimension $k = n - mt$. This code is capable of correcting up to t errors [4] and can be described as a $k \times n$ generator matrix G such that $C = \{mG : m \in F_2^k\}$.

To encode a message m into a codeword c, represent the message m as a binary string of length k and multiply it with the $k \times n$ matrix G.

However, decoding such a codeword $\underline{r}$ on the receiver's side with a (possibly) additive error vector $\underline{e}$ is far more complex. For decoding, we use Patterson's algorithm [24] with improvements from [29].

Since $\underline{r} = \underline{c} + \underline{e} \equiv \underline{e} \mod G(z)$ holds, the syndrome $Syn(z)$ of a received codeword can be obtained from Equation (1) by

$$Syn(z) = \sum_{\alpha \in GF(2^m)} \frac{r_\alpha}{z-\alpha} \equiv \sum_{\alpha \in GF(2^m)} \frac{e_\alpha}{z-\alpha} \mod G(z) \tag{2}$$

To finally recover $\underline{e}$, we need to solve the key equation $\sigma(z) \cdot Syn(z) \equiv \omega(z) \mod G(z)$, where $\sigma(z)$ denotes a corresponding error-locator polynomial and $\omega(z)$ denotes an error-weight polynomial. Note that it can be shown that $\omega(z) = \sigma(z)'$ is the formal derivative of the error-locator and by splitting $\sigma(z)$ into even and odd polynomial parts $\sigma(z) = a(z)^2 + z \cdot b(z)^2$, we finally determine the following equation which needs to be solved to determine error positions:

$$Syn(z)(a(z)^2 + z \cdot b(z)^2) \equiv b(z)^2 \mod G(z) \tag{3}$$

To solve Equation (3) for a given codeword $\underline{r}$, the following steps have to be performed:

1. From the received codeword $\underline{r}$ compute the syndrome $Syn(z)$ according to Equation (2). This can also be done using simple table-lookups.
2. Compute an inverse polynomial $T(z)$ with $T(z) \cdot Syn(z) \equiv 1 \mod G(z)$ (or provide a corresponding table). It follows that $(T(z) + z)b(z)^2 \equiv a(z)^2 \mod G(z)$.
3. There is a simple case if $T(z) = z \Rightarrow a(z) = 0$ s.t. $b(z)^2 \equiv z \cdot b(z)^2 \cdot Syn(z) \mod G(z) \Rightarrow 1 \equiv z \cdot Syn(z) \mod G(z)$ what directly leads to $\sigma(z) = z$. Contrary, if $T(z) \neq z$, compute a square root $R(z)$ for the given polynomial $R(z)^2 \equiv T(z) + z \mod G(z)$. Based on a observation by Huber [19] we can then determine solutions $a(z), b(z)$ satisfying

$$a(z) = b(z) \cdot R(z) \mod G(z). \tag{4}$$

Algorithm 3. Decoding Goppa Codes

```
Input: Received codeword $r$ with up to $t$ errors, inverse generator matrix $iG$
Output: Recovered message $\hat{m}$
 1: Compute syndrome $Syn(z)$ for codeword $r$
 2: $T(z) \leftarrow Syn(z)^{-1} \mod G(z)$
 3: if $T(z) = z$ then
 4:     $\sigma(z) \leftarrow z$
 5: else
 6:     $R(z) \leftarrow \sqrt{T(z) + z}$
 7:     Compute $a(z)$ and $b(z)$ with $a(z) \equiv b(z) \cdot R(z) \mod G(z)$
 8:     $\sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
 9: end if
10: Determine roots of $\sigma(z)$ and correct errors in $r$ which results in $\hat{r}$
11: $\hat{m} \leftarrow \hat{r} \cdot iG$ {Map $r_{cor}$ to $\hat{m}$}
12: return $\hat{m}$
```

Finally, we use the identified $a(z), b(z)$ to construct the error-locator polynomial $\sigma(z) = a(z)^2 + z \cdot b(z)^2$.

4. The roots of $\sigma(z)$ denote the positions of error bits. If $\sigma(\alpha_i) \equiv 0 \mod G(z)$ with α_i being the corresponding bit of a generator in $GF(2^{11})$, there was an error in the position i in the received codeword that can be corrected by bit-flipping.

This decoding process, as required in Step 2 of Algorithm 2 for message decryption, is finally summarized in Algorithm 3.

3.2 Security Parameters

All security parameters for cryptosystems are chosen in a way to provide sufficient protection against the best known attack (whereas the notion of "sufficient" is determined by the requirements of an application). A recent paper [8] by Bernstein *et al.* presents a state-of-the-art attack of McEliece making use of a list decoding algorithm [6] for binary Goppa codes.

This attack reduces the binary work factor to break the original McEliece scheme with a $(1024, 524)$ Goppa code and $t = 50$ to $2^{60.55}$ bit operations. According to [8], Table 1 summarizes the security parameters for specific security levels.

4 Design Criteria for Embedded Systems

In this section, we discuss our assumptions, requirements and restrictions which are required when implementing the original McEliece cryptosystem on small, embedded systems. Target platforms for our investigation are 8-bit AVR microprocessors as well as low-cost Xilinx Spartan-3AN FPGAs. Some devices of these platforms come with large integrated Flash-RAMs (e.g., 192 kByte and 2,112 kByte for an AVR ATxMega192 and Spartan-3AN XC3S1400AN, respectively).

4.1 Requirements and Assumptions

For many embedded systems such as prepaid phones or micropayment systems, the short life cycle or comparably low value of the enclosed product often does

Table 1. Security of the McEliece scheme

Security Level	**Parameters** (n, k, t), errors added	**Size** K_{pub} in KBits	**Size** K_{sec} $(G(z), P, S)$ in KBits
Short-term (60 bit)	$(1024, 644, 38), 38$	644	$(0.38, 10, 405)$
Mid-term (80 bit)	$(2048, 1751, 27), 27$	$3,502$	$(0.30, 22, 2994)$
Long-term (256 bit)	$(6624, 5129, 115), 117$	$33,178$	$(1.47, 104, 25690)$

not demand for very long-term security, Hence, mid-term security parameters for public-key cryptosystems providing a comparable security to 64-80 key bits of symmetric ciphers are often regarded sufficient (and help reducing system costs). Hence, our implementations are designed for security parameters that correspond to an 80 bit key size of a symmetric cipher. A second important design requirement is the processing and storage of the private key solely *on-chip* so that all secrets are optimally never used outside the device. With appropriate countermeasures to prevent data extraction from on-chip memories, an attacker can then recover the private key only by sophisticated invasive attacks. For this purpose, AVR μCs provide a lock-bit feature to enable write and read/write protection of the Flash memory [2]. Similar mechanisms are also available for Spartan-3AN FPGAs preventing configuration and Flash readback from chip internals, e.g., using JTAG or ICAP interfaces [27]. Note that larger security parameters of the McEliece scheme are still likely to conflict with this requirement due to the limited amount of permanent on-chip memories of today's embedded platforms.

Analyzing McEliece encryption and decryption algorithms (cf. Section 3.1), the following arithmetic components are required supporting computations in $GF(2^m)$: a multiplier, a squaring unit, calculation of square roots, and an inverter. Furthermore, a binary matrix multiplier for encryption and a permutation element for step 2 in Algorithm 1 are needed. Many arithmetic operations in McEliece can be replaced by table lookups to significantly accelerate computations at the cost of additional memory. For both implementations in this work, our primary goal is area and memory efficiency to fit the large keys and required lookup-tables into the limited on-chip memories of our embedded target platforms.

The susceptibility of the McEliece cryptosystem to side channel attacks has not extensively been studied, yet. However, embedded systems can always be subject to passive attacks such as timing analysis [20] and power/EM analysis [21]. In [28], a successful timing attack on the Patterson algorithm was demonstrated. The attack does not recover the key, but reveals the error vector z and hence allows for efficient decryption of the message c. Our implementations are not susceptible to this attack due to unconditional instruction execution, e.g., our implementation will not terminate after a certain number of errors have been corrected. Differential EM/power attacks and timing attacks are impeded by the permutation and scrambling operations (P and S) obfuscating all internal states, and finally, the large key size. Yet template-like attacks [12] might be feasible if no further protection is applied.

4.2 Reducing Memory Requirements

To make McEliece-based cryptosystems more practical (i.e., to reduce the key sizes), there is an ongoing research to replace the code with one that can be represented in a more compact way.

Using a naïve approach in which the support of the code is the set of all elements in $GF(2^m)$ in lexicographical order and both matrices S, P are totally random, the public key $\hat{G} = S \times G \times P$ becomes a random $n \times k$ matrix. However, since P is a sparse permutation matrix with only a single 1 in each row and column, it is more efficient to store only the positions of the ones, resulting in an array with $n \cdot m$ bits.

Another trick to reduce the public key size is to convert $\hat{G}$ to systematic form $\{I_k \mid Q\}$, where I_k is the $k \times k$ identity matrix. Then, only the $(k \times (n-k))$ matrix Q is published [14].

In the last step of code decoding (Algorithm 3), the k message bits out of the n (corrected) ciphertext bits need to be extracted. Usually, this is done by a mapping matrix iG with $G \times iG = I_k$. But if G is in systematic form, then this step can be omitted, since the first k bits of the corrected ciphertext corresponds to the message bits. Unfortunately, G and $\hat{G}$ cannot both be systematic at the same time, since then $\hat{G} = \{I_k \mid \hat{Q}\} = S \times \{I_k \mid Q\} \times P$ and S would be the identity matrix which is inappropriate for use as the secret key.

For reduction of the secret key size, we chose to generate the large scrambling matrix S^{-1} on-the-fly using a cryptographic pseudo random number generator (CPRNG) and a seed. During key generation, it must be ensured that the seed does not generate a singular matrix S^{-1}. Depending on the target platform and available cryptographic accelerators, there are different options to implement such a CPRNG (e.g. AES in counter mode or a hash-based PRNG) on embedded platforms. However, the secrecy of S^{-1} is not required for hiding the secret polynomial $G(z)$ [14].

5 Implementation on AVR Microprocessors

In this section, we discuss our implementation of the McEliece cryptosystem for 8-bit AVR microcontrollers, a popular family of 8-bit RISC microcontrollers (μC) used in embedded systems. The Atmel AVR processors operate at clock frequencies of up to 32 MHz, provide few kBytes of SRAM, up to hundreds of kBytes of Flash program memory, and additional EEPROM or mask ROM. For our design, we chose an ATxMega192A1 μC due to its 16 kBytes of SRAM and the integrated crypto accelerator engine for DES and AES [2]. The crypto accelerator is particularly useful for a fast implementation of a CPRNG that generates the scrambling matrix S^{-1} on-the-fly. Arithmetic operations in the underlying field $GF(2^{11})$ can be performed efficiently with a combination of polynomial and exponential representation. We store the coefficients of a value $a \in GF(2^{11})$ in memory using a polynomial basis with natural order. Given an $a = a_{10}\alpha^{10} + a_9\alpha^9 + a_8\alpha^8 + \cdots + a_0\alpha^0$, the coefficient $a_i \in GF(2)$ is determined by bit i of an unsigned 16 bit integer where bit 0 denotes the least significant bit. In this representation, addition is fast just by performing an exclusive-or operation on 2×2 registers. For more complex operations, such as multiplication, squaring, inversion and root extraction, an exponential representation is more suitable. Since every element except zero in $GF(2^{11})$ can be written as a power of some primitive element α, all elements in the finite field can also be represented by α^i with $i \in \mathbb{Z}_{2^m-1}$. Multiplication and squaring can then be performed by adding the exponents of the factors over $\mathbb{Z}_{2^m-1}$ such as

$$c = a \cdot b = \alpha^i \cdot \alpha^j = \alpha^{i+j} \mid a, b \in GF(2^{11}), 0 \leq i, j \leq 2^m - 2. \qquad (5)$$

If one of the elements equals zero, obviously the result is zero. The inverse of a value $d \in GF(2^{11})$ in exponential representation $d = \alpha^i$ can be obtained from a single subtraction in the exponent $d^{-1} = \alpha^{2^{11}-1-i}$ with a subsequent table-lookup. Root extraction, i.e., given a value $a = \alpha^i$ to determine $r = a^{i/2}$ is simple, when i is even and can be performed by a simple right shift on index i. For odd values of i, $m - 1 = 10$ left shifts followed by a reduction with $2^{11} - 1$ determine the square root.

To allow for efficient conversion between the two representations, we employ two precomputed tables (so called *log* and *antilog* tables) that enable fast conversion between polynomial and exponential representation. Each table consists of 2048 11-bit values that are stored as a pair of two bytes in the program memory. Hence, each lookup table consumes 4 kBytes of Flash memory. Due to frequent access, we copy the tables into the faster SRAM at startup time. Accessing the table directly from Flash memory significantly reduces performance, but allows migration to a (slightly) cheaper device with only 4 kBytes of SRAM. For multiplication, squaring, inversion, and root extraction, the operands are transformed on-the-fly to exponential representation and reverted to the polynomial basis after finishing the operation.

5.1 Generation and Storage of Matrices

All matrices as shown in Table 2 are precomputed and stored in Flash memory of the μC. We store the permutation matrix P^{-1} as an array of 2048 16-bit unsigned integers containing 11-bit indices. Matrix G is written in transposed form to simplify multiplications (i.e., all columns are stored as consecutive words in memory for straightforward index calculations). Additionally, arrays for the support of the code, its reverse mapping, and the precomputed inverse polynomials (in the order as they correspond to the ciphertext bits) reside in Flash memory as well. Since the scrambling matrix S^{-1} is too large to be stored in program memory, we opted to generate it on-the-fly from an 80-bit seed, employing the integrated DES-accelerator engine of the ATxMega as a CPRNG.

Encryption is a straightforward binary matrix-vector multiplication and does not require field arithmetic in $GF(2^{11})$. However, the large public-key matrix K_{pub} does not fit into the 192 kByte internal Flash memory. Hence, at least 512 kByte external memory are required for storing the public key $\hat{G}$. Note that the ATxMega can access external memories at the same speed as internal SRAM.

Table 2 shows the requirements of precomputed tables separated by actual size and required size in memory including the necessary 16-bit address alignment and/or padding.

5.2 System and Compiler Limitations

Due to the large demand for memory, we need to take care of some peculiarities in the memory management of the AVR microcontroller. Since originally AVR

Table 2. Sizes of tables and values in memory including overhead for address alignment

Use	Name	Actual Size	Size in Memory
Encryption	Public Key $\hat{G}$	448,256 byte	448,512 byte
Decryption	Private Key S^{-1} (IV only)	10 byte	10 byte
Decryption	Private Key P^{-1} array	2,816 byte	4,096 byte
Decoding	Syndrome table	76,032 byte	110,592 byte
Decoding	Goppa polynomial	309 bits	56 byte
Decoding	ω-polynomial	297 bits	54 byte
Decoding	Log table	22,528 bits	4,096 byte
Decoding	Antilog table	22,528 bits	4,096 byte

microcontrollers supported only a small amount of internal memory, the AVR uses 16 bit pointers to access its Flash memory. Additionally, each Flash cell comprises 16 bit of data, but the μC itself can only handle 8 bit. Hence, one bit of this address pointer must be reserved to select the corresponding byte in the retrieved word, reducing the maximal address range to 64 KByte (or 32K 16 bit words). To address memory segments beyond 64K, additional RAMP-registers need to be used. Additionally, the used avr-gcc compiler internally treats pointers as signed 16 bit integer halving again the addressable memory space. For this reason, all arrays larger than 32 Kbyte need to be split into multiple parts resulting in an additional overhead in the program code.

6 Implementation on Xilinx FPGAs

Since our target device is a low-cost Spartan-3 with moderate logic resources, we only parallelized and unrolled the most time consuming parts of the algorithms such as the polynomial multiplier and inverter. Alike the AVR implementation, we decided to implement less intensive operations of the field arithmetic (i.e., inversion, division, squaring and square roots for single field elements over $GF(2^{11})$) using precomputed log- and antilog tables which are stored in dedicated memory components (BRAM) of the FPGA (cf. Section 5.1). With such precomputed tables being available, the number of computational units in hardware can be reduced what also affects the number of required Lookup-Tables (LUT) in the Configurable Logic Blocks (CLB) of the FPGA. However, note that only 32 BRAMs are available on the Spartan-3AN 1400 FPGA (which is the largest, low-cost device of its class). This limits the option to have more than one instance of each table for allowing parallel access (besides using the dual-port feature of the BRAM). Hence, lookups to these tables need to be serialized in most cases. Since the runtime of polynomial multiplication and polynomial squaring is crucial for the overall system performance (cf. Steps 7 and 8 of Algorithm 3), we opted for a parallel polynomial multiplier instead of using the log and antilog tables as well. The polynomial multiplier consists of 27 coefficient multipliers over $GF(2^{11})$ (the topmost coefficient is treated separately) of which each coefficient multiplication is realized as logic directly in LUTs by linear combination of the input bits and the field polynomial. Hence, the multiplication of a polynomial B with a coefficient a (i.e., $C = a \cdot B \mid a \in GF(2^{11}), B, C \in \frac{F[z]}{G(z)}$) can be performed in a single clock cycle. All field operations, such as root extraction, division, and inversion can be completed in 6 clock cycles using log and antilog tables, of which two clock cycles are for the conversion to exponential representation, two are required for the corresponding operation and additional two cycles for the reverse translation. Note that for several subsequent field computations the conversion can be interleaved with the arithmetic operation so that only 4 cycles for each subsequent operations are required.

The remaining, time-critical component is the polynomial inverter which is used in step 1 and step 2 of Algorithm 3, for example to compute the parity check matrix H on-the-fly. An average of 1024 inverses need to be computed for which we implemented the binary Extended Euclidean Algorithm (EEA) over $GF(2^{11})$ in hardware. Note that each cycle of the polynomial EEA requires exactly one coefficient division (which is realized again using the log and antilog tables). In conclusion, the EEA is the largest component in our design (about 64%) and thus also comprises the critical path of the implementation. For the generation of the inverse scrambling matrix S^{-1} on the FPGA, we implemented a CPRNG based on the 80-bit low-footprint block cipher PRESENT. Note that as an alternative, we store the large static table S^{-1} in the in-system Flash memory of the FPGA. However, due to limitations of the serial SPI-Interface we only can access a single bit of S^{-1} at a maximum frequency of 50 MHz that significantly degrades our decryption performance.

This limitation also applies to the public-key matrix $\hat{G}$ which is required for the encoding process during encryption. Since this matrix is too large to fit into the 32 18 kBit BRAMs of our Spartan-3AN device, we need to store it in Flash memory. To avoid a performance penalty due to the slow SPI interface to the Flash, we could first load K_{pub} into an external DDR2-333 memory at system startup which then can be accessed via a fast memory controller to retrieve K_{pub} for encryption. With such undamped access to K_{pub}, we could gain a performance speedup for encryption by a factor of 62 (1.15 ms) with respect to loading K_{pub} directly from Flash (71.44 ms). We successfully verified this approach by testing our implementation on a test board providing external SRAM (however, no DDR2 memory). The interface between SRAM and FPGA is realized as 16 bit bus, clocked at 100 MHz and two clock cycles access time per read.

Due to the limited logic on our FPGA, we thus opted for an individual device configuration for encryption and decryption, of which one can be selected during system startup. Both configurations can be stored within the large, internal Flash memory of the FPGA. Using the multi-boot features of Spartan-3 devices, the corresponding configuration can also be loaded by the FSM (using the internal SPI-interface) during runtime whenever switching between encryption and decryption is necessary. The McEliece implementation (decryption configuration) for the Spartan-3AN FPGA is depicted in Figure 1.

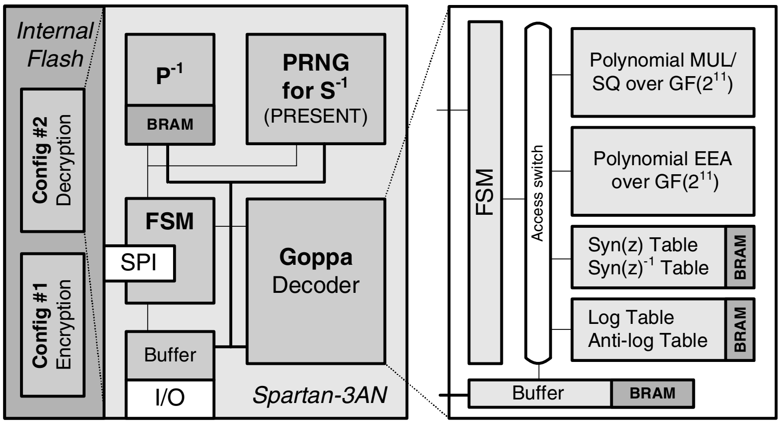

Fig. 1. McEliece implementation on a Spartan-3AN FPGA

7 Results

We now present the results for our two McEliece implementations providing 80 bit security ($n = 2048, k = 1751, t = 27$) for the AVR 8-bit microcontroller and the Xilinx Spartan-3AN FPGA. We report performance figures for the ATxMega192A1 obtained from the avr-gcc compiler v4.3.2 and a Xilinx Spartan-3AN XC3S1400AN-5 FPGA using Xilinx ISE 10.1. The resource requirements for our μC design and FPGA implementation after place-and-route (PAR) are shown in Table 3.

Table 4 summarizes the clock cycles needed for every part of the de- and encryption routines both for the FPGA and the microcontroller implementation.

In our FPGA design, the CPRNG to generate S^{-1} based on the PRESENT block cipher turns out to be a bottleneck of our implementation since the matrix generation does not meet the performance of the matrix multiplication. By replacing the CPRNG by a more efficient solution, we can untap the full

Table 3. Implementation results of the McEliece scheme with $n = 2048, k = 1751, t = 27$ on the AVR ATxMega192 μC and Spartan-3AN XC3S1400AN-5 FPGA after PAR

	Resource	Encryption	Decryption	Available
μC	SRAM	512 Byte	12 kByte	16 kByte
	Flash Memory	684 Byte	130.4 kByte	192 kByte
	External Memory	438 kByte	–	–
FPGA	Slices	668 (6%)	11,218 (100%)	11,264
	LUTs	1044 (5%)	22,034(98%)	22,528
	FFs	804 (4%)	8,977 (40%)	22,528
	BRAMs	3 (9%)	20 (63%)	32
	Flash Memory	4,644 KBits	4,644 KBits	16,896 Kbits

Table 4. Performance of McEliece implementations with $n = 2048, k = 1751, t = 27$ on the AVR ATxMega192 μC and Spartan-3AN XC3S1400AN-5 FPGA

	Aspect	**ATxMega192 μC**	**Spartan-3AN 1400**
Encrypt.	Maximum frequency	32 MHz	150 MHz
	Encrypt $c' = m \cdot \hat{G}$	14,404,944 cycles	(7,889,200)161,480 cycles
	Inject errors $c = c' + z$	1,136 cycles	398 cycles
Decryption	Maximum frequency	32 MHz	85 MHz
	Undo permutation $c \cdot P^{-1}$	275,835 cycles	combined with $Syn(z)$
	Determine $Syn(z)$	1,412,514 cycles	360,184 cycles
	Compute $T = Syn(z)^{-1}$	1,164,402 cycles	625 cycles
	Compute $\sqrt{T+z}$	286,573 cycles	487 cycles
	Solve Equation (4) with EEA	318,082 cycles	312 cycles
	Correct errors	15,096,704 cycles	312,328 cycles
	Undo scrambling $\hat{m} \cdot S^{-1}$	1,196,984 cycles	1,035,684/217,800* cycles

* This figure is an estimate assuming that an ideal PRNG for generation of S^{-1} would be available.

Table 5. Comparison of our McEliece designs with single-core ECC and RSA implementations for 80 bit security

	Method	**Platform**	**Time** ms/op	**Throughput** bits/sec
8-bit μC	McEliece encryption	ATxMega192@32MHz	450	3,889
	McEliece decryption	ATxMega192@32MHz	618	2,835
	ECC-P160 (SECG) [17]	ATMega128@8MHz	810/203[1]	197/788[1]
	RSA-1024 $2^{16}+1$ [17]	ATMega128@8MHz	430/108[1]	2,381/9,524[1]
	RSA-1024 random [17]	ATMega128@8MHz	10,990/2748[1]	93/373[1]
FPGA	McEliece encryption A	Spartan-3AN 1400-5	1.07[2]	1,626,517[2]
	McEliece encryption B	Spartan-3AN 1400-5	2.24[3]	779,948[3]
	McEliece decryption	Spartan-3AN 1400-5	21.61/10.82[4]	81,023/161,829[4]
	ECC-P160 [16]	Spartan-3 1000-4	5.1	31,200
	RSA-1024 random [18]	Spartan-3E 1500-5	51	20,275
	NTRU encryption [3]	Virtex 1000EFG860	0.005	50,876,908

[1] For a fair comparison with our implementations running at 32MHz, timings at lower frequencies were scaled accordingly.
[2] These are estimates are based on the usage of an external DDR-RAM.
[3] These are measurements based on our test setup with external SRAM running at 100MHz.
[4] These are estimates assuming that an ideal PRNG to generate S^{-1} is used.

performance of our implementation. Table 4 also gives estimates for a PRNG that does not incur any wait cycles due to throughput limitations.

The public-key cryptosystems RSA-1024 and ECC-P160 are assumed[2] to roughly achieve a similar margin of 80 bit symmetric security [1]. We finally compare our results to published implementations of these systems that target similar platforms (i.e., AVR ATMega μC and Xilinx Spartan-3 FPGAs). Note that the figures for ECC are obtained from the ECDSA signature scheme.

Embedded implementations of other alternative public key encryption schemes are very rare. The proprietary encryption scheme NTRUEncrypt has received some attention. An encryption-only hardware engine of NTRUEncrypt-251-128-3 for more advanced Xilinx Virtex platform has been presented in [3]. An embedded software implementation of the related NTRUSign performs one signature on an ATMega128L clocked at 7,37 MHz in 619 ms [13]. However, comparable performance figures of NTRU encryption and decryption for the AVR platform are not available.

Note that all throughput figures are based on the number of plaintext bits processed by each system and do not take any message expansion in the ciphertext into account.

8 Conclusions

In this paper, we described the first implementations of the McEliece public-key scheme for embedded systems using an AVR μC and a Xilinx Spartan-3AN FPGA. Our performance results for McEliece providing 80 bit security on these systems exceed the throughput but could not outperform comparable ECC cryptosystems with 160 bit in terms of number of operations per second. However, although our implementations still leave room for further optimizations, our results already show better performance than RSA-1024 on the selected platforms. Thus, we believe with growing memories in embedded systems, ongoing research and further optimizations, McEliece can evolve to a suitable and quantum computer-resistant alternative to RSA and ECC that have been extensively studied for years.

References

1. ECRYPT. Yearly Report on Algorithms and Keysizes (2007-2008). Technical report, D.SPA.28 Rev. 1.1, IST-2002-507932 ECRYPT (July 2008)
2. Atmel Corp. 8-bit XMEGA A Microcontroller. User Guide (February 2009), `http://www.atmel.com/dyn/resources/prod_documents/doc8077.pdf`
3. Bailey, D.V., Coffin, D., Elbirt, A., Silverman, J.H., Woodbury, A.D.: NTRU in Constrained Devices. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 262–272. Springer, Heidelberg (2001)

[2] According to [1], RSA-1248 actually corresponds to 80 bit symmetric security. However, no implementation results for embedded systems are available for this key size.

4. Berlekamp, E.R.: Goppa codes. IEEE Trans. Information Theory IT-19(3), 590–592 (1973)
5. Berlekamp, E.R., McEliece, R.J., van Tilborg, H.C.A.: On the inherent intractability of certain coding problems. IEEE Trans. Information Theory 24(3), 384–386 (1978)
6. Bernstein, D.J.: List Decoding for Binary Goppa Codes. Technical report (2008), `http://cr.yp.to/codes/goppalist-20081107.pdf`
7. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems, February 17 (2009), `http://bench.cr.yp.to`
8. Bernstein, D.J., Lange, T., Peters, C.: Attacking and Defending the McEliece Cryptosystem. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 31–46. Springer, Heidelberg (2008)
9. Beuchat, J.-L., Sendrier, N., Tisserand, A., Villard, G.: FPGA Implementation of a Recently Published Signature Scheme. Technical report, INRIA - Institut National de Recherche en Informatique et en Automatique (2004), `http://hal.archives-ouvertes.fr/docs/00/07/70/45/PDF/RR-5158.pdf`
10. Biswas, B., Sendrier, N.: McEliece crypto-system: A reference implementation, `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`
11. Bogdanov, A., Eisenbarth, T., Rupp, A., Wolf, C.: Time-Area Optimized Public-Key Engines: MQ-Cryptosystems as Replacement for Elliptic Curves? In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 45–61. Springer, Heidelberg (2008)
12. Chari, S., Rao, J.R., Rohatgi, P.: Template Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)
13. Driessen, B., Poschmann, A., Paar, C.: Comparison of Innovative Signature Algorithms for WSNs. In: Proceedings of ACM WiSec 2008. ACM, New York (2008)
14. Engelbert, D., Overbeck, R., Schmidt, A.: A summary of mceliece-type cryptosystems and their security (2007)
15. Freenet and Entropy. Open-Source P2P Network Applications (2009), `http://freenetproject.org` and `http://entropy.stop1984.com`
16. Güneysu, T., Paar, C., Pelzl, J.: Special-Purpose Hardware for Solving the Elliptic Curve Discrete Logarithm Problem. ACM Transactions on Reconfigurable Technology and Systems (TRETS) 1(2), 1–21 (2008)
17. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 925–943. Springer, Heidelberg (2004)
18. Helion Technology Inc. Modular Exponentiation Core Family for Xilinx FPGA. Data Sheet (October 2008), `http://www.heliontech.com/downloads/modexp_xilinx_datasheet.pdf`
19. Huber, K.: Note on decoding binary Goppa codes. Electronics Letters 32, 102–103 (1996)
20. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
21. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smartcards. Springer, Heidelberg (2007)
22. McEliece, R.J.: A Public-Key Cryptosystem Based On Algebraic Coding Theory. Deep Space Network Progress Report 44, 114–116 (1978)
23. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, New York (1996)

24. Patterson, N.: The algebraic decoding of Goppa codes. IEEE Transactions on Information Theory 21, 203–207 (1975)
25. Preneel, B., Bosselaers, A., Govaerts, R., Vandewalle, J.: A Software Implementation of the McEliece Public-Key Cryptosystem. In: Proceedings of the 13th Symposium on Information Theory in the Benelux, Werkgemeenschap voor Informatie en Communicatietheorie, pp. 119–126. Springer, Heidelberg (1992)
26. Prometheus. Implementation of McEliece Cryptosystem for 32-bit microprocessors (c-source) (2009), `http://www.eccpage.com/goppacode.c`
27. Smerdon, M.: Security Solutions Using Spartan-3 Generation FPGAs. Whitepaper (April 2008),
`http://www.xilinx.com/support/documentation/white_papers/wp266.pdf`
28. Strenzke, F., Tews, E., Molter, H., Overbeck, R., Shoufan, A.: Side Channels in the McEliece PKC. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 216–229. Springer, Heidelberg (2008)
29. Sugiyama, Y., Kasahara, M., Hirasawa, S., Namekawa, T.: A Method for Solving Key Equation for Decoding Goppa Codes. IEEE Transactions on Information and Control 27, 87–99 (1975)