J Cryptogr Eng (2013) 3:29–43
DOI 10.1007/s13389-013-0056-4

CHES 2012

Code-based cryptography on reconfigurable hardware: tweaking Niederreiter encryption for performance

Stefan Heyse · Tim Güneysu

Received: 16 November 2012 / Accepted: 28 January 2013 / Published online: 14 March 2013

Abstract Today's public-key schemes that are either based on the factorization or the discrete logarithm problem. Since both problems are closely related, a major breakthrough in cryptanalysis (e.g., with the advent of quantum computing will render nearly all currently employed security system useless. Code-based public-key schemes rely on the alternative security assumption that decoding generic linear binary codes is NP-complete. Two code-based schemes for public-key encryption are available due to McEliece and Niederreiter. Although most researchers analyzed and implemented McEliece's cryptosystem, we show in this work that the scheme by Niederreiter has some important advantages, such as smaller keys, more practical plain and ciphertext sizes and less computation complexity. In particular, we propose an efficient FPGA implementation of Niederreiter's scheme that can encrypt more than 1.5 million plaintexts per seconds on a Xilinx Virtex-6 FPGA—outperforming all known implementations of other popular public-key cryptosystems so far.

Keywords Code-based · Goppa · McEliece · Niederreiter · Embedded · FPGA

1 Introduction

Public-key cryptosystems build the foundation for virtually all advanced cryptographic requirements, such as asymmetric encryption, key exchange and digital signatures. But all established cryptosystems rely on two classes of fundamental problems, namely the factoring problem and the (elliptic curve) discrete logarithm problem. Since both are related, a cryptanalytical breakthrough will turn a large number of currently employed security systems to be completely insecure overnight. This threat is further nourished by upcoming generations of powerful quantum computers that have been shown to be very effective in computing solutions to the problems mentioned above [58]. Recently, IBM announced two improvements in quantum computing [16] so that such practical systems might already become available even in the next 15 years. Obviously, we need a larger *diversification* of public-key primitives that can maintain long-term security even in and beyond the era of quantum computers.

Therefore, some alternative cryptosystems have gathered much attention in the last years, such as multivariate-quadratic ($\mathcal{MQ}$-), lattice-based and code-based schemes. A major drawback of these constructions have been their low efficiency and practicability due to large key sizes or complex computations compared to classical RSA and ECC cryptosystems. This was particularly considered as an issue for small and embedded systems where memory and computational resources are scarce. Nevertheless, thanks to many improvements in technology over the last years, we have seen several publications that successfully demonstrated the implementation of such alternative public-key cryptography on embedded platforms [2,11,22,33,46,52].

Contribution: In this work, we present another successful transfer of an alternative public-key cryptosystem to embedded systems, namely an implementation of the code-based public-key scheme by Niederreiter [49]. The Niederreiter cryptosystem is dual to McEliece's proposal [43] and thus maintains the same level of security. Both code-based encryption schemes have proven to be secure even after more than 30 years of thorough analysis if security parameters and

S. Heyse (✉) · T. Güneysu
Horst Görtz Institute for IT-Security, Ruhr-Universitä Bochum, 44780 Bochum, Germany
e-mail: stefan.heyse@rub.de

T. Güneysu
e-mail: gueneysu@crypto.rub.de

underlying codes are appropriately chosen [8]. A recent result also demonstrated that McEliece and Niederreiter cryptosystems also resist quantum Fourier sampling algorithms [19] that are precisely those algorithms that would invalidate RSA and discrete logarithm-based cryptosystems with the advent of quantum computers.

We provide an extended view on implementations of the Niederreiter scheme on reconfigurable hardware (i.e., Xilinx Spartan-6 and Virtex-6 FPGAs) that were partially published in [34]. Exclusively, we implement and evaluate two different decoding algorithms and also demonstrate how to efficiently migrate from one to the other with minimal overhead. Both implementations provide a level of 80-bit equivalent symmetric security and can process more than 1.5 million encryption and 17,000 decryption operations per second, respectively, while using only a moderate amount of resources. Finally, we conclude that our implementations can even provide a significantly better performance and efficiency compared to established asymmetric cryptosystems, such as ECC-160 and RSA-1024. With optimizations to the constant-weight encoding algorithm and the inherent advantages of Niederreiter over McEliece, we achieve a performance that is even (a few) orders of magnitudes faster than any other asymmetric implementation reported so far.

Outline: This article is structured as follows: in Sect. 3 we provide background on codes and decoding algorithms. Section 4 provides a introduction to McEliece and Niederreiter encryption, including a brief description of modern variants of both schemes. Section 6 describes the implementations for a Xilinx Virtex-6 FPGA, however, we also provide results for other FPGA devices in Sect. 7. We conclude our work in Sect. 8.

2 Previous work

Although proposed more than 30 years ago, code-based encryption schemes have never gained much attention due to their large secret and public keys. It was common perception for quite a long time that due to their expensive memory requirements such schemes are difficult to be integrated in any (cost-driven) real-world products. The original proposal by Robert McEliece for a code-based encryption scheme suggested the use of binary Goppa codes, but in general any other linear code could be used. While other types of codes may have advantages such as a more compact representation, most proposals using different codes were proven less secure (cf. [44,50]). The Niederreiter cryptosystem is an independently developed variant of McEliece's proposal which is proven to be equivalent in terms of security [42]. In 2009, a first FPGA-based implementation of McEliece's cryptosystem was proposed targeting a Xilinx Spartan-3AN and encrypts and decrypts data in 1.07 and 2.88 ms, using security parameters achieving an equivalence of 80-bit symmetric security [22]. The authors of [60] presented another accelerator for McEliece encryption over binary Goppa codes on a more powerful Virtex5-LX110T, capable to encrypt and decrypt a block in 0.5 and 1.4 ms providing a similar level of security. The latest publication [27] based on hardware/software co-design on an Spartan3-1400AN decrypt a block in 1 ms at 92 MHz[1] at the same level of security. For x86-based platforms, a recent implementation of the McEliece scheme over binary Goppa codes is due to Biswas and Sendrier [10] achieving about 83 bit of equivalent symmetric security according to [8]. Comparing their implementation to other public-key schemes, it turns out that McEliece encryption can be faster than RSA and NTRU [7], however, at the cost of larger keys. Many proposals already tried to address this issue of large keys by replacing the originally used binary Goppa codes with (secure) codes that allow more compact representations, e.g., [15,45]. However, most of the attempts were broken [24] and for the few (still) surviving ones hardly any implementations are available [2,33]. Note further that most of these works exclusively target the McEliece cryptosystems. To the best of our knowledge, the only published implementation of Niederreiter encryption for embedded systems is an implementation for small 8-bit AVR microcontrollers that can encrypt and decrypt a block in 1.6 and 179 ms, respectively [32]. In addition to that, we are aware of just another implementation of a signature scheme in Java based on Niederreiter's concept [54].

3 Codes and algorithms

In this section, we briefly introduce relevant background information on Goppa codes and corresponding coding algorithms that we later need to implement the encryption scheme.

Definition 1 (*Codes over finite fields*) Let $\mathbb{F}_{p^m}^n$ denote a vector space of n tuples over $\mathbb{F}_{p^m}$. A (n, k)-code $\mathcal{C}$ over a finite field $\mathbb{F}_{p^m}$ is a k-dimensional subvectorspace of $\mathbb{F}_{p^m}^n$. For $p = 2$, it is called a *binary* code, otherwise it is called *p-ary*.

A convenient way to specify a code is to give a *generator matrix* G whose rows form a basis of $\mathcal{C}$. A codeword c representing a message m of length n can be easily computed as $c = m \cdot G$. Note that in general G is not uniquely determined and an equivalent code can be obtained by performing linear transformations and a permutation. Equivalently, the code can be defined using a *parity check matrix* H with $G \cdot H^{\mathrm{T}} = 0$.

[1] This work does not provide performance results for encryption.

Definition 2 (*Generator matrix and parity check matrix*) A $k \times n$ matrix G with full rank is a *generator matrix* for the (n, k)-code $\mathcal{C}$ over $\mathbb{F}$ if the rows of G span $\mathcal{C}$ over $\mathbb{F}$. The $(n-k) \times n$ matrix H is called *parity check matrix* for the code C if H^{T} is the right kernel of G. H can be used to determine whether a vector belongs to the code $\mathcal{C}$ by verifying $H \cdot c^{\mathrm{T}} = 0 = m \cdot G \cdot H^{\mathrm{T}} = m \cdot 0$, which holds for every error-free codeword of $\mathcal{C}$. The code generated by H is called *dual code* of $\mathcal{C}$ and denoted by $\mathcal{C}^{\mathrm{T}}$.

Definition 3 (*Systematic matrices*) A matrix is called *systematic* if it has the form $G = (I_k|Q)$, where I_k denotes the $k \times k$ identity matrix. If G is a generator matrix in systematic form, the parity check matrix H can be computed from G as $H = (-Q^{\mathrm{T}}|I_{n-k})$, and vice versa.

A vector $\hat{c} = c + e$ with an error vector e of $\mathrm{wt}(e) > 0$ added to the codeword c can be interpreted as an erroneous codeword.

Definition 4 (*Syndrome*) The *syndrome* of a vector $\hat{c} = c+e$ in $\mathbb{F}_q^n$ is the vector in $\mathbb{F}_q^{n-k}$ defined by $\mathcal{S}_{\hat{c}} = H \cdot \hat{c}^{\mathrm{T}} = H \cdot (c^{\mathrm{T}} + e^{\mathrm{T}}) = H \cdot e^{\mathrm{T}}$.

Generalized Reed–Solomon (GRS) Codes. GRS codes are a generalization of the very common class of Reed–Solomon (RS) codes. While RS codes are always cyclic, GRS are not necessarily cyclic. GRS codes are maximum distance separable (MDS) codes, which means that they are optimal in the sense of the *singleton bound*, i.e., the minimum distance has the maximum value possible for a linear (n, k)-code: $d_{\min} = n - k + 1$.

For some polynomial $f(x) \in \mathbb{F}_{p^m}[x]_{<k}$, pairwise distinct elements $\mathcal{L} = (\alpha_0, \ldots, \alpha_{n-1}) \in \mathbb{F}_{p^m}^n$, non-zero elements $V = (v_0, \ldots, v_{n-1}) \in \mathbb{F}_{p^m}^n$ and $0 \leq k \leq n$, GRS code can be defined as

$$GRS_{n,k}(\mathcal{L}, V) := \{c \in \mathbb{F}_{p^m}^n | c_i = v_i f(\alpha_i)\} \tag{1}$$

Alternant Codes An alternant matrix has the form $M_{i,j} = f_j(\alpha_i)$. *Alternant codes* use a parity check matrix H of alternant form and have a minimum distance $d_{\min} \geq t + 1$ and a dimension $k \geq n - mt$. For pairwise distinct $\alpha_i \in \mathbb{F}_{p^m}$, $0 \leq i \leq n-1$ and non-zero $v_i \in \mathbb{F}_{p^m}$, $0 \leq j \leq t-1$, the elements of the parity check matrix are defined as $H_{i,j} = \alpha_j^i v_i$.

Alternant codes are subfield subcodes of GRS codes, i.e., they can be obtained by restricting GRS-codes to the subfield $\mathbb{F}_p$:

$$Alt_{n,k,p}(\mathcal{L}, v) := GRS_{n,k}(\mathcal{L}, V) \cap \mathbb{F}_p^n \tag{2}$$

Goppa codes They are alternant codes over $\mathbb{F}_{p^m}$ that are restricted to a *Goppa polynomial* $g(x)$ with $\deg(g) = t$ and a *support* $\mathcal{L}$ with $g(\alpha_i) \neq 0\ \forall i$. Here, g is just another representation of the previously used tuple of non-zero elements V and polynomial $f(x)$. Hence, a definition of Goppa codes can be derived from the definition of GRS codes as follows:

$$Goppa_{n,k,p}(\mathcal{L}, g) := GRS_{n,k}(\mathcal{L}, g) \cap \mathbb{F}_p^n \tag{3}$$

The minimum distance of a Goppa code is $d_{\min} \geq t + 1$, in case of binary Goppa codes with an irreducible Goppa polynomial even $d_{\min} \geq 2t + 1$. Goppa codes [4,28] are one of the most important code classes in code-based cryptography, not only because the original proposal by McEliece was based on Goppa codes, but most notably because they belong to the few code classes that resisted all critical attacks so far. Hence, we will describe them in greater detail and use Goppa codes—more specifically, binary Goppa codes using an irreducible Goppa polynomial—to introduce the decoding algorithms developed by Patterson and by Berlekamp and Massey.

3.1 Binary Goppa codes

We begin by reiterating the above definition of Goppa for the case of binary Goppa codes, giving an explicit definition of the main ingredients.

Definition 5 Let m and t be positive integers and let the *Goppa polynomial*

$$g(x) = \sum_{i=0}^{t} g_i x^i \in \mathbb{F}_{2^m}[x]$$

be a monic polynomial of degree t and let the *support*

$$\mathcal{L} = \{\alpha_0, \ldots, \alpha_{n-1}\} \in \mathbb{F}_{2^m}^n, g(\alpha_j) \neq 0 \quad \forall 0 \leq j \leq n-1$$

be a subset of n distinct elements of $\mathbb{F}_{2^m}$. For any vector $\hat{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_{2^m}^n$, in accordance with Definition 4 we define the *syndrome* of $\hat{c}$ as

$$\mathcal{S}_{\hat{c}}(x) = -\sum_{i=0}^{n-1} \frac{\hat{c}_i}{g(\alpha_i)} \frac{g(x) - g(\alpha_i)}{x - \alpha_i} \mod g(x). \tag{4}$$

3.2 Encoding

In continuation of Eq. 3, we now define a *binary Goppa code* over $\mathbb{F}_{2^m}$ using its syndrome equation. The codeword c is a representation of a binary message m of length k which has been transformed into a n bit codeword in the encoding step by multiplying m with the generator matrix G. In particular, $c \in \mathbb{F}_{2^m}^n$ is a codeword of the code exactly if $\mathcal{S}_c = 0$:

$$Goppa_{n,k,2}(\mathcal{L}, g(x)) := \left\{ c \in \mathbb{F}_{2^m}^n \mid \mathcal{S}_c(x) = \sum_{i=0}^{n-1} \frac{c_i}{x - \alpha_i} \equiv 0 \mod g(x) \right\}.$$

If $g(x)$ is irreducible over $\mathbb{F}_{2^m}$ then $Goppa(\mathcal{L}, g)$ is called an *irreducible binary Goppa code*. If $g(x)$ has no multiple roots, then $Goppa(\mathcal{L}, g)$ is called a *separable* code and $g(x)$ as *square-free* polynomial.

According to the definition of a syndrome in Eq. (4), every element $\hat{c}_i$ of a vector $\hat{c} = c + e$ is multiplied with

$$\frac{g(x) - g(\alpha_i)}{g(\alpha_i) \cdot (x - \alpha_i)}. \tag{5}$$

Hence, given a Goppa polynomial $g(x) = g_s x^s + g_{s-1}x^{s-1} + \cdots + g_0$, according to Definition 2 the parity check matrix H can be constructed as

$$H = \begin{pmatrix} g_s & 0 & \cdots 0 \\ g_{s-1} & g_s & \cdots 0 \\ \vdots & \ddots & \vdots \\ g_1 & g_2 & \cdots g_s \end{pmatrix} \times \begin{pmatrix} \frac{1}{g(\alpha_0)} & \frac{1}{g(\alpha_1)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\ \frac{\alpha_0}{g(\alpha_0)} & \frac{\alpha_1}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}}{g(\alpha_{n-1})} \\ \vdots & \ddots & & \vdots \\ \frac{\alpha_0^{s-1}}{g(\alpha_0)} & \frac{\alpha_1^{s-1}}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^{s-1}}{g(\alpha_{n-1})} \end{pmatrix}$$
$$= H_g \times \hat{H} \tag{6}$$

where H_g has a determinant unequal to zero. Then, $\hat{H}$ is an equivalent parity check matrix to H, but having a simpler structure. Using Gauss–Jordan elimination, $\hat{H}$ can be brought to systematic form. Note that for every column swap in Gauss–Jordan, also the corresponding elements in the support $\mathcal{L}$ need to be swapped. As shown in Definition 3, the generator matrix G can be derived from the systematic parity check matrix $H = (Q|I_{n-k})$ as $(I_k| - Q^{\mathrm{T}})$

3.3 Decoding

Many different algorithms for decoding linear codes are available. The Berlekamp–Massey algorithm (BM) is one of the most popular algorithms for decoding BCH codes [3], but later found to be able to decode *any* alternant code [41]. The same applies to the Peterson decoder [53] or Peterson–Gorenstein–Zierler algorithm [29] and the more recent list decoding [63]. However, there are also specialized algorithms that decode a subclass of codes more efficiently. An important example is Patterson's algorithm [51] for binary Goppa codes, but there are also several other specialized variants of general decoding algorithms for specific code classes, such as the list decoder for binary Goppa codes [6].

3.3.1 Solving the key equation

Let E be a vector with elements in $\mathcal{F}_{p^m}$ representing the error positions, i.e., the position of ones in the error vector e. Then, by different means, both Patterson and BM compute an *error locator polynomial* $\sigma(z)$, whose roots determine the error positions in an erroneous codeword $\hat{c}$. More precisely, the roots γ_i are elements of the support $\mathcal{L}$ for the Goppa code $Goppa(\mathcal{L}, g(z))$, where the positions of these elements inside of $\mathcal{L}$ correspond to the error positions in $\hat{c}$. The error locator polynomial is defined as:

$$\sigma(z) = \prod_{i \in E}(z - \gamma_i) = \prod_{i \in E}(1 - x_i z) \tag{7}$$

In the binary case, the position holds enough information for the correction of the error, since an error value is always 1, whereas 0 means 'no error'. However, in the non-binary case, an additional *error weight polynomial* $\omega(x)$ is required for the determination of the error values. Let y_i denote the error value of the ith error. Then, the error value polynomial is defined as:

$$\omega(z) = \sum_{i \in E} y_i x_i \prod_{j \neq i \in E}(1 - x_j z). \tag{8}$$

Note that it can be shown that $\omega(x) = \sigma'(x)$ is the formal derivative of the error locator polynomial. Since the Patterson algorithm is designed only for *binary* Goppa codes, $\omega(x)$ does not occur there explicitly. Nevertheless, both algorithms implicitly or explicitly solve the following *key equation*

$$\omega(x) \equiv \sigma(x) \cdot \mathcal{S}(x) \mod g(x). \tag{9}$$

3.3.2 Patterson decoding

In 1975, Patterson presented a polynomial-time algorithm which is able to correct t errors for binary Goppa codes with a designed minimum distance $d_{\min} \geq 2t + 1$. Patterson achieves this error-correction capability by taking advantage of certain properties present in *binary* Goppa codes [23], whereas general decoding algorithms such as BM can only correct $\frac{t}{2}$ errors by default. Overbeck gives a runtime complexity estimation [50] of $\mathcal{O}(n \cdot t \cdot m^2)$, whereas BM has a runtime quadratic in the code length n.

Algorithm 1 PATTERSON ALGORITHM FOR DECODING BINARY GOPPA CODES

Require: Vector $\hat{c} = c + e$, Goppa code with an irreducible Goppa polynomial $g(x)$
Ensure: Recovered message $\hat{m}$, error vector $\hat{e}$
Compute syndrome $s = \mathcal{S}_{\hat{c}}(x)$
if $s \equiv 0 \mod g(x)$ **then**
  return $(\hat{c}, 0)$
else
  $T(x) \leftarrow s^{-1} \mod g(x)$
  if $T(x) = z$ **then**
    $\sigma(x) \leftarrow z$
  else
    $R(x) \leftarrow \sqrt{T(x) + z}$
    $(a(x), b(x)) \leftarrow \mathrm{EEA}(R(x), g(x))$
    $\sigma(x) \leftarrow a(x)^2 + z \cdot b(x)^2$
  end if
  $\tilde{e} \leftarrow$ extract_roots$(\sigma(x))$
  $\tilde{c} \leftarrow \hat{c} + \tilde{e}$
  return (map_codeword_to_message$(\tilde{c})$, $\tilde{e}$)
end if

Algorithm 1 summarizes the Patterson algorithm for decoding a vector $\hat{c} = c + e \in \mathbb{F}_{2^m}^n$ using a binary Goppa code with an irreducible Goppa polynomial $g(x)$ of degree

t. The error vector e has been added to c either intentionally like in code-based cryptography, or unintentionally, for example during the transmission of c over a noisy channel. The Patterson algorithm ensures the correction of all errors only if a maximum of t errors occurred, i.e., e has a weight $\mathrm{wt}(e) \leq t$.

Syndrome computation The algorithm computes the syndrome $\mathcal{S}_{\hat{c}}(x)$, which is equal to $\mathcal{S}_e(x)$ since $\mathcal{S}_c(x) = 0$ by definition. It can be computed either as $\mathcal{S}_{\hat{x}}(x) = H \cdot \hat{c}^{\mathrm{T}}$ according to Definition 4, or as

$$\mathcal{S}(xl) \equiv \sum_{\alpha \in \mathbb{F}_{2^m}} \frac{\hat{c}_\alpha}{x - \alpha_i} \equiv \sum_{\alpha \in \mathbb{F}_{2^m}} \frac{e_\alpha}{x - \alpha_i} \mod g(x)$$

as seen in Eq. (4). If $\mathcal{S}(x) = 0$, the codeword is error-free and the algorithm returns the unchanged codeword as well as an error vector $e = 0$.

Solving the key equation The Patterson algorithm does not directly solve the key equation. Instead, it transforms Eq. (9) to a simpler one using the property $\omega(x) = \sigma'(x)$ and the fact that $y_i = 1$ at all error positions.

$$\omega(x) \equiv \sigma(x) \cdot \mathcal{S}(x) \equiv \sum_{i \in E} x_i \prod_{j \neq i \in E} (1 - z) \mod g(x)$$

Then, $\sigma(x)$ is split into an odd and even part.

$$\sigma(x) = a(x)^2 + x b(x)^2 \tag{10}$$

Now, formal derivation and application of the original key equation yields

$$\begin{aligned} \sigma'(x) = b(x)^2 &= \omega(x) \\ &\equiv \sigma(x) \cdot \mathcal{S}(x) \mod g(x) \\ &\equiv (a(x)^2 + x b(x)^2) \cdot \mathcal{S}(x) \equiv b(x)^2 \mod g(x) \end{aligned}$$

Choosing $g(x)$ irreducible ensures the invertibility of the syndrome $\mathcal{S}$. To solve the equation for $a(x)$ and $b(x)$, we now compute an inverse polynomial $T(x) \equiv \mathcal{S}_{\hat{c}}(x)^{-1} \mod g(x)$ and obtain

$$[T(x) + x] \cdot b(x)^2 \equiv a(x)^2 \mod g(x).$$

If $T(x) = x$, we obtain the trivial solutions $a(x) = 0$ and $b(x)^2 = x b(x)^2 \cdot \mathcal{S}(x) \mod g(x)$, yielding $\sigma(x) = x$. Otherwise, we use an observation by Huber [38] for polynomials in $\mathbb{F}_{2^m}$ giving a simple expression for the polynomial $r(x)$ solving $r(x)^2 \equiv t(x) \mod g(x)$. To satisfy Huber's equation, we set $R(x)^2 \equiv T(x) + x \mod g(x)$ and obtain $R(x) \equiv \sqrt{T(x) + x}$. Finally, $a(x)$ and $b(x)$ satisfying $a(x) \equiv b(x) \cdot R(x) \mod g(x)$ can be computed using the Extended Euclidean Algorithm (EEA) and applied to Eq. 10. Equation 10 and $\deg(\sigma)(x) \leq g(x) = t$ imply that $\deg(a(x)) \leq \lfloor \frac{t}{2} \rfloor$ and $\deg(b(x)) \leq \lfloor \frac{t-1}{2} \rfloor$ [50]. Observing the iterations of the EEA, one finds that the degree of $a(x)$ is constantly decreasing from $a_0 = g(x)$ while the degree of $b(x)$ stepwise increases from zero. Hence, there is an unique point where the degree of both polynomials is below their respective bounds. Therefore, EEA can be stopped at this point, i.e., when $a(x)$ drops below $\frac{t}{2}$.

Root extraction The computation of the error locator polynomial $\sigma(x)$ is followed by the computational expensive process of extracting the roots of $\sigma(x)$. Since we know that all roots are elements of the support $\mathcal{L}$, it is possible to find the roots by brute force (i.e., testing $\sigma(\alpha_i) = 0$ for all $\alpha_i \in \mathcal{L}$). The two most commonly used methods for this are the Chien search [17] and Horner's scheme [37]. A third method as proposed in [9] cannot be easily parallelized and involves many greatest common divisor and trace computations. For hardware implementations, we therefore prefer the use of a parallelized version of the Chien search that concurrently evaluates all t coefficients.

Message recovery If $\mathrm{wt}(e) \leq t$, the Patterson algorithm is able to recover the complete error vector e. Then, $c = \hat{c} + e$ can be recovered by a simple addition, which is identical to flipping the bits in $\hat{c}$ corresponding to the error positions via exclusive-or. For $\mathrm{wt}(e) > t$, the recovered $\tilde{c}$ is incorrect and differs from the original codeword c. If a systematic generator matrix G has been used for the encoding of the original message m, c can be mapped to m just by removing the appended parity data. Otherwise, it needs to be mapped using the inverse generator matrix G^{-1} by computing $\hat{m} \leftarrow \hat{c} \cdot G^{-1}$.

3.3.3 Berlekamp–Massey and Sugiyama decoding

The BM algorithm was originally proposed by Berlekamp in 1968 and also works on general alternant codes. Applied to binary codes, the error value polynomial can be neglected as argued before. The process of BM decoding is quite simple compared to Patterson's algorithm, however, the original proof for its correctness is more complicated. Note that another decoding algorithm due to Sugiyama [64] is based on the EEA and can be regarded equivalent to the BM algorithm.

Algorithm 2 BERLEKAMP- - MASSEY- SUGIYAMA ALGORITHM TO DECODE GENERAL GOPPA CODES USING EEA

Require: Vector $\hat{c} = c + e$, Goppa code with a polynomial $g(x)$
Ensure: Recovered message $\hat{m}$, error vector $\hat{e}$
1: Compute syndrome $\mathcal{S}_{\hat{c}}(x)$
2: **if** $s \equiv 0 \mod g(x)$ **then**
3: **return** $(\hat{c}, 0)$
4: **else**
5: $(\sigma(x), \omega(x)) \leftarrow \mathrm{EEA}(\mathcal{S}(x), g(x))$
6: $\tilde{e} \leftarrow \mathrm{extract\_roots}(\sigma(x))$
7: $\tilde{c} \leftarrow \hat{c} + \tilde{e}$
8: **return** $(\mathrm{map\_codeword\_to\_message}(\tilde{c}), \tilde{e})$
9: **end if**

Algorithm 2 summarizes the Berlekamp–Massey algorithm for the decoding of a vector $\hat{c} = c + e \in \mathbb{F}_{p^m}^n$ using an alternant code with a designed minimum distance

$d_{\min} = t + 1$ and a polynomial $g(x)$, which may be a—possibly reducible—Goppa polynomial $g(x)$ of degree t. As before, c is a representation of a message m and e an error vector. In the *general* case, the BM algorithm ensures the correction of all errors only if a maximum of $\frac{t}{2}$ errors occurred, i.e., e has a weight $\mathrm{wt}(e) \leq \frac{t}{2}$. In the *binary* case it is possible to achieve t-error correction with BM by using $g(x)^2$ instead of $g(x)$ and thus working on a syndrome of double size.

Decoding general alternant codes The first step of the BM algorithm consists of the syndrome computation. In the case of Goppa codes this is identical to the computation as described in Sect. 3.3.2. In the general case, Berlekamp defines the syndrome as $\mathcal{S}(x) = \sum_{i=1}^{\infty} S_i x^m$, where only $S_1, \ldots, S_t$ are known to the decoder. Then, he constructs a relation between σ and ω and the known S_i by dividing ω by σ as defined in Eqs. (7) and (8)

$$\frac{\omega}{\sigma} = 1 + \sum_j \frac{y_j x_j z}{1 - x_j z} = 1 + \sum_{i=1}^{\infty} S_i z^m \tag{11}$$

and obtains the key equation

$$[1 + \mathcal{S}(x)] \cdot \sigma \equiv \omega \mod x^{2t+1} \tag{12}$$

already known from Sect. 3.3.1.

For solving the key equation, Berlekamp proposes a sequence of successive approximations, $\omega^{(0)}, \sigma^{(0)}, \ldots, \sigma^{(2t)}, \omega^{(2t)}$, each pair of which solves an equation of the form $(1 + \mathcal{S}(x))\sigma^{(k)} \equiv \omega^{(k)} \mod x^{k+1}$ [5]. The algorithm that Berlekamp gives for solving these equations was found to be very similar to the EEA by numerous researchers. Dornstetter proofs that the iterative version of the Berlekamp–Massey can be derived from a normalized version of Euclid's algorithm [20] and hence considers them to be equivalent. Accordingly, BM is also very similar to Sugiyama's decoding algorithm [64] which sets up the same key equation and explicitly applies EEA. However, Bras-Amorós and O'Sullivan state that BM is widely accepted to have a better performance than the Sugiyama algorithm [13]—to the contrary of [39]. In our later hardware implementation, we decided to implement and describe BM using EEA due to its simpler structure. The key equation can be solved by applying EEA to $\mathcal{S}(x), g(x))$, which returns σ and ω. Given σ, the error positions can be determined in the same way as shown in Sect. 3.3.2 for Patterson's algorithm.

Application to Code-based cryptosystems Application of the Berlekamp–Massey algorithm to the McEliece cryptosystem is straightforward, in the general as well as in the binary case. This also works in the general case when applying BM to Niederreiter's cryptosystem, with the same minor changes required as specified for Patterson's algorithm. However, a remaining problem is the BM-decoding of t errors as in Niederreiter's scheme in the binary case. Since the Niederreiter cryptosystem uses a syndrome as a ciphertext instead of a codeword, the approach of computing a syndrome of double size using BM modulo $g(z)^2$ cannot be used. Switching completely to a code over $g(z)^2$—also for the encryption process—would double the code size. In this case, Patterson's algorithm is preferable. However, we can use the following trick.[2] Recall that a syndrome s of length $n - k$ corresponding to an erroneous codeword $\hat{c}$ satisfies the equation $s = \mathcal{S}_{\hat{c}} = e \cdot H^{\mathrm{T}}$, where e is the error vector that we want to obtain by decoding s. Using the equivalence

$$Goppa(\mathcal{L}, g(x)) \equiv Goppa(\mathcal{L}, g(x)^2) \tag{13}$$

which is true for any square-free polynomial $g(x)$, we can construct a syndrome polynomial of degree $2t - 1$ based on a parity check matrix of double size for $Goppa(\mathcal{L}, g(x)^2)$. By prepending a given syndrome s—computed using the usual systematic parity matrix $H = (Q|I_{n-k})$ generated by $g(z)$—with k zeros, we obtain $(0|s)$ of length n. Then, using (13) we compute a parity check matrix H_2 modulo $g(z)^2$. Since $\deg(g(z)^2) = 2t$, the resulting parity check matrix has dimensions $2(n - k) \times n$. Computing $(0|s) \cdot H_2 = s_2$ yields a new syndrome of length $2(n - k)$, resulting in a syndrome polynomial of degree $2t - 1$, as in the non-binary case. Due to the equivalence of Goppa codes over $g(x)$ and $g(x)^2$, and the fact that $(0|s)$ and e belong to the same coset, s_2 is still a syndrome corresponding to $\hat{c}$ and having the same solution e. However, s_2 has the appropriate length for the key equation and allows Berlekamp–Massey to decode the complete error vector e.

4 Code-based cryptosystems

Code-based cryptosystems make use of the fact that decoding the syndrome of a *general* linear code is known to be $\mathcal{NP}$-hard, while efficient algorithms exist for the decoding of *specific* linear codes. Hence the definition of a trapdoor function applies. For encryption, the message is converted into a codeword by either adding random errors to the message or encoding the message in the error. Decryption recovers the plaintext by removing the errors or extracting the message from the errors. An adversary knowing the specific used code would be able to decrypt the message, therefore it is imperative to hide the algebraic structure of the code, effectively disguising it as an unknown general code.

In this work, we use the term "classical" McEliece or Niederreiter to denote the original cryptosystem as proposed by their authors. The term "modern" is used for a variant with equivalent security that we consider more appropriate for contemporary implementations. While we introduce the reader to both variants, throughout the remainder of the work

[2] Special thanks to N. Sendrier for pointing this out.

Algorithm 3 CLASSICAL MCELIECE: KEY GENERATION

Require: Fixed system parameters t, n, p, m
Ensure: private key K_{sec} , public key K_{pub}
1: Choose a binary $[n, k, d]$-Goppa code $\mathcal{C}$ capable of correcting up to t errors
2: Compute the corresponding $k \times n$ generator matrix G for code $\mathcal{C}$
3: Select a random non-singular binary $k \times k$ scrambling matrix S
4: Select a random $n \times n$ permutation matrix P
5: Compute the $k \times n$ matrix $\hat{G} = S \cdot G \cdot P$
6: Compute the inverses of S and P
7: **return** $K_{sec} = (\mathcal{D}_{Goppa}(c), S^{-1}, P^{-1}), K_{pub} = (\hat{G})$

we will consider only the "modern" variant for our actual implementation.

As shown in Algorithm 3, the key generation algorithm selects a binary Goppa code capable of correcting up to t errors. This is done by randomly choosing an irreducible Goppa polynomial of degree t and computing a corresponding generator matrix G, which is the primary part of the secret key. Given G an adversary would be able to identify the specific code and thus to decode it efficiently. Hence the algebraic structure of G needs to be hidden. For this purpose, a scrambling matrix S and a permutation matrix P are randomly selected and multiplied with G to form $\hat{G} = S \cdot G \cdot P$. Matrix S is chosen to be invertible and the permutation P effectively reorders the columns of the codeword, which needs to be reversed before decoding. Hence $\hat{G}$ is still a valid generator matrix for an equivalent code $\mathcal{C}$ (cf. [12] for details on code equivalence). The matrix $\hat{G}$ now serves as the public key and the matrices G, S and P, or equivalently S^{-1}, P^{-1}, and the decoding algorithm $\mathcal{D}_{Goppa}(c)$ for $\mathcal{C}$ compose the secret key. Canteaut and Chabaud note in [14] that the scrambling matrix S in classical McEliece has no cryptographic function, but only assures that the public matrix is not systematic in order not to reveal the plaintext bits. However, just not revealing the plaintext bits still provides no security beyond a weak form of obfuscation. Hence, a CCA2-secure conversion [15,40,48,55] is required to allow the use of a systematic matrix in a modern McEliece variant.

McEliece encryption is done as a simple vector-matrix multiplication of the k-bit message m with the $k \times n$ generator matrix $\hat{G}$ and an addition of a random error vector e with Hamming weight at most t, as shown in Algorithm 4. The multiplication adds redundancy to the codeword, resulting in a message expansion from k to n with an overhead of $\frac{n}{k}$.

Algorithm 4 CLASSICAL MCELIECE: ENCRYPTION

Require: Public key $K_{pub} = (\hat{G})$, message M
Ensure: Ciphertext c
1: Represent message M as binary string m of length k
2: Choose a random error vector e of length n with hamming weight $\leq t$
3: **return** $c = m \cdot \hat{G} + e$

The McEliece decryption shown in Algorithm 5 consists mainly of the removal of the applied errors using the known

Algorithm 5 CLASSICAL MCELIECE: DECRYPTION

Require: Ciphertext c, private key $K_{sec} = (\mathcal{D}_{Goppa}(c), S^{-1}, P^{-1})$
Ensure: Message M
1: Compute $\hat{c} = c \cdot P^{-1}$
2: Obtain $\hat{m}$ from $\hat{c}$ using the decoding algorithm $\mathcal{D}_{Goppa}(\hat{c})$ for code $\mathcal{C}$
3: Compute $m = \hat{m} \cdot S^{-1}$
4: Represent m as message M
5: **return** M

decoding algorithm $\mathcal{D}_{Goppa}(c)$ for the code $\mathcal{C}$. Before the decoding algorithm can be applied, the permutation P needs to be reversed. After the decoding step the scrambling S needs to be reversed. Decoding is the most time-consuming part of decryption and make decryption much slower than encryption.

Remember that permutation P does not affect the Hamming weight of c, and the multiplication $S \cdot G \cdot P$ with S being non-singular produces a generator matrix for a code equivalent to $\mathcal{C}$. Therefore, the decoding algorithm is able to extract the vector of *permuted* errors $e \cdot P^{-1}$ and thus $\hat{m}$ can be recovered. In order to reduce the memory requirements of McEliece and to allow a more practical implementation, the version that we call *modern McEliece*—in accordance with [36]—opts for the usage of a generator matrix in *systematic* form. In this case, the former scrambling matrix S is chosen to arrange the generator matrix into systematic form so that it does not need to be stored explicitly. Moreover, the permutation P is applied to the *code support* L instead of the generator matrix by choosing the support randomly and storing the permutation only implicitly. As a result, the public key is reduced from a $k \cdot n$ matrix to a $k \cdot (n - k)$ matrix. Apart from the smaller memory requirements, this has also positive effects on encryption and decryption speed, since the matrix multiplication needs less operations and the plaintext is just copied to and from the codeword. The private key size is also reduced: instead of storing S and P, only the permuted support L and the Goppa polynomial $g(x)$ needs to be stored. The security of modern McEliece is equivalent to the classical version, since the only modifications are a restriction of S to specific values and a different representation of P. Overbeck notes that this version requires a semantically secure conversion, but also stresses that such a conversion is needed anyway [50]. Algorithm 6 shows the key generation algorithm for the modern McEliece variant. It begins with the selection of a random Goppa polynomial $g(x)$ of degree t. The *support* L is then chosen randomly as a subset of $GF(p^m)$ elements that are no roots of $g(x)$. Often n equals p^m and $g(x)$ is chosen to be irreducible, so all elements of $GF(p^m)$ are in the support. In classical McEliece, the support is fixed and public and can be handled implicitly as long as $n = p^m$. In modern McEliece, the support is not fixed but random, and it must be kept secret. Hence it is sometimes called L_{sec}, with L_{pub} being the public support, which is only used implicitly through

Algorithm 6 MODERN MCELIECE: KEY GENERATION

Require: Fixed system parameters t, n, p, m
Ensure: private key K_{sec} , public key K_{pub}
1: Select a random goppy polynomial $g(z)$ of degree t over $GF(p^m)$
2: Randomly choose n elements of $GF(p^m)$ that are no roots of $g(z)$ as the *support* L
3: Compute the parity check matrix $\hat{H}$ according to L and $g(z)$
4: Bring H to systematic form using Gauss–Jordan elimination: $H_{sys} = \hat{S} \cdot H$
5: Compute systematic generator matrix G_{sys} from H_{sys}
6: **return** $K_{sec} = (\mathcal{D}_{Goppa}(c), L, g(z))$, $K_{pub} = (G_{sys})$

the use of G. Using the relationships discussed in Sect. 3.2, the parity check matrix H is computed according to $g(x)$ and L and brought to systematic form using Gauss–Jordan elimination. Note that for every column swap in Gauss–Jordan, also the corresponding support elements need to be swapped. Finally, the public key in the form of the systematic generator matrix G is computed from H. The private key consists of the support L and the Goppa polynomial, which form a code for that an efficient decoding algorithm $\mathcal{D}_{Goppa}(c)$ is known.

Encryption in modern McEliece (Algorithm 7) is identical to encryption in classical McEliece, but can be implemented more efficiently since the multiplication of the plaintext with the identity part of the generator matrix results in a mere copy of the plaintext to the ciphertext.

Algorithm 7 MODERN MCELIECE: ENCRYPTION

Require: Public key $K_{pub} = (G_{sys} = (I_k|Q))$, message M
Require: Ciphertext c
1: Represent message M as binary string m of length k
2: Choose a random error vector e of length n with hamming weight $\leq t$
3: **return** $c = m \cdot G_{sys} + e = (m \,||\, m \cdot Q) + e$

Decryption for the modern McEliece variant shown in Algorithm 8 consists exclusively of the removal of the applied errors using the known decoding algorithm $\mathcal{D}_{Goppa}(c)$ for the code $\mathcal{C}$. The permutation is handled implicitly through the usage of the permuted secret support. The scrambling does not need to be reversed neither, because the information bits can be read directly from the first k bits of the codeword.

Algorithm 8 MODERN MCELIECE: DECRYPTION

Require: Ciphertext c, private key $K_{sec} = (\mathcal{D}_{Goppa}(c), L, g(z))$
Ensure: Message M
1: Obtain m from c using the decoding algorithm $\mathcal{D}_{Goppa}(c)$ for code $\mathcal{C}$
2: Represent m as message M
3: **return** M

Eight years after McEliece's proposal, Niederreiter [49] developed a similar cryptosystem, apparently not aware of the previous work by McEliece. It encodes the message completely in the error vector, thus avoiding the obvious information leak of the plaintext bits not affected by the error addition as in McEliece. Since CCA2-secure conversions need to be used in all cases, this has no effect on the security, but it results in smaller plaintext blocks, what is often preferable. Moreover, Niederreiter uses the syndrome as ciphertext instead of the codeword, hence moving some of the decryption workload into the encryption (which still remains a fast operation). The syndrome calculation requires the parity check matrix as a public key instead of the generator matrix. If systematic matrices are used, this has no effect on the key size. Unfortunately, the Niederreiter cryptosystem does not allow to omit the scrambling matrix S. Instead of S, the inverse matrix S^{-1} should be stored, since only that is explicitly used. The algorithms shown in this section present the general classical Niederreiter cryptosystem and the modern variant applied to Goppa codes. Key generation works similar to McEliece, but does not require the computation of the generator matrix. Algorithm 9 shows the classical key generation algorithm for the Niederreiter cryptosystem, while Algorithm 10 presents its modern counterpart using a systematic parity check matrix and a secret support. Without the identity part, the systematic parity check matrix has the size $k \times (n-k)$ instead of $n \times (n-k)$. The inverse scrambling matrix S^{-1} is a $(n-k)(n-k)$ matrix.

Algorithm 9 CLASSICAL NIEDERREITER:KEY GENERATION

Require: Fixed system parameters t, n, p, m
Ensure: private key K_{sec} , public key K_{pub}
1: Choose a binary $[n, k, d]$-Goppa code $\mathcal{C}$ capable of correcting up to t errors
2: Compute the corresponding $(n-k) \times n$ parity check matrix H for code $\mathcal{C}$
3: Select a random non-singular binary $(n-k) \times (n-k)$ scrambling matrix S
4: Select a random $n \times n$ permutation matrix P
5: Compute the $n \times (n-k)$ matrix $\hat{H} = S \cdot H \cdot P$
6: Compute the inverses of S and P
7: **return** $K_{sec} = (\mathcal{D}_{Goppa}(c), S^{-1}, P^{-1})$, $K_{pub} = (\hat{H})$

Algorithm 10 MODERN NIEDERREITER:KEY GENERATION

Require: Fixed system parameters t, n, p, m
Ensure: private key K_{sec} , public key K_{pub}
1: Select a random goppy polynomial $g(z)$ of degree t over $GF(p^m)$
2: Randomly choose n elements of $GF(p^m)$ that are no roots of $g(z)$ as the *support* L
3: Compute the $(n-k) \times n$ parity check matrix $\hat{H}$ according to L and $g(z)$
4: Bring H to systematic form using Gauss–Jordan elimination: $H_{sys} = \hat{S} \cdot H$
5: Compute S^{-1}
6: **return** $K_{sec} = (\mathcal{D}_{Goppa}(c), L, g(z), S^{-1})$, $K_{pub} = (H_{sys})$

For encryption, the message M needs to be represented as a constant-weight (CW) encoded word of length n and hamming weight t. There exist several techniques for CW encoding, one of which will be presented in Sect. 5.2. The CW encoding is followed by a simple vector-matrix multiplication. Encryption is shown in Algorithm 11. It is identical for the classical and modern variant apart from the fact that

Algorithm 11 NIEDERREITER: ENCRYPTION

Require: Public key $K_{pub} = (\hat{H})$, message M
Ensure: Ciphertext c
1: Represent message M as binary string e of length n and weight t
Ensure: $c = \hat{H} \cdot e^{\mathrm{T}}$

the multiplication with a systematic parity check matrix can be realized more efficiently.

For the decoding algorithm, the code scrambling needs to be reverted by multiplying the syndrome with S^{-1}. Afterwards the decoding algorithm is able to extract the error vector from the syndrome. In the classical Niederreiter decryption as given in Algorithm 12, the error vector after decoding is still permuted, so it needs to be multiplied by P^{-1}. In the modern variant shown in Algorithm 13, the permutation is reverted implicitly during the decoding step. Finally, CW decoding is used to turn the error vector back into the original plaintext.

Algorithm 12 CLASSICAL NIEDERREITER: DECRYPTION

Require: Ciphertext c, private key $K_{sec} = (\mathcal{D}_{Goppa}(c), S^{-1}, P^{-1})$
Ensure: Message M
1: Compute $\hat{c} = S^{-1} \cdot c$
2: Obtain $\hat{e}$ from $\hat{c}$ using the decoding algorithm $\mathcal{D}_{Goppa}(\hat{c})$ for code $\mathcal{C}$
3: Compute $e = P^{-1} \cdot \hat{e}$
4: Represent e as message M
5: **return** M

Algorithm 13 MODERN NIEDERREITER: DECRYPTION

Require: Ciphertext c, private key $K_{sec} = (\mathcal{D}_{Goppa}(c), S^{-1})$
Ensure: Message M
1: Compute $\hat{c} = S^{-1} \cdot c$
2: Obtain e from $\hat{c}$ using the decoding algorithm $\mathcal{D}_{Goppa}(\hat{c})$ for code $\mathcal{C}$
3: Represent e as message M
4: **return** M

4.1 Niederreiter versus McEliece

The main difference between Niederreiter's and McEliece's encryption scheme is the public key. While an $(n \times k)$ generator matrix serves as public key in McEliece's scheme, Niederreiter uses a $(n \times (n-k))$ parity check matrix for this purpose. Both matrices can be used in their systematic form, leading to $((n-k) \cdot k)$ bits storage requirement in both cases. Using this method for McEliece encryption demands a CCA2 secure conversion [15,26] to remain secure, whereas Niederreiter encryption can be used without this conversion. McEliece used an n-bit code word with errors as ciphertext, whereas Niederreiter employed the $(n-k)$-bit syndrome as plaintext. This shifts the syndrome computation from the receiver to the sender of the message and therefore speeds up decryption, still maintaining a high encryption performance. At the same time, the parity check matrix and related information is no longer part of the secret key, thus reducing the secret key size. However, the Niederreiter scheme requires the scrambling matrix S which can be omitted when using McEliece encryption. Finally, Niederreiter encryption imposes less restrictions on the plaintext size, i.e., depending on the parameter sets and constant-weight encoding algorithm, Niederreiter enables plaintext blocks with a size of only hundreds of bits instead of several thousands bits as in the case of McEliece encryption. In particular, for key transportation protocols with symmetric key sizes of 128 to 256 bits, the transfer of thousands of bits as required in the case of McEliece can be an expensive overhead.

4.2 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). On the attempt to employ an alternative cryptosystem, it is of utmost important for a security engineer to being able to safely assess if this best attack has already been found. In this context, the work by Bernstein et al. [8] currently proposes the best attack on McEliece and Niederreiter cryptosystems so far reducing the work factor to break the McEliece scheme based on a (1,024, 524) Goppa code and $t = 50$ to $2^{60.55}$ bit operations. According to their findings, we summarize the security parameters for specific security levels in Table 1. The public key size column gives the size of a systematic parity check matrix and the secret key column the size of the Goppa polynomial $g(x)$, the support $\mathcal{L}$ and the inverse scrambling matrix S^{-1}. The distinction between the error capability t and the number of errors added is necessary due to list decoding [6], which can correct a few more errors than the designed error capability t

As can be clearly seen, the main caveat of code-based cryptosystems is the significant size of the public and private keys. For 80-bit security, for example, the parameters $m = 11, n = 2{,}048, t = 27, k \geq 1{,}751$ already lead to an S^{-1} of 11 KBytes and a public key size of 63 KBytes. Note that we can reduce the size of the public key from originally 74 KBytes by choosing S in such a way that it brings $\hat{H}$ to systematic form $\hat{H} = (I\, D_{n-k} \mid Q)$, where only the redundant part Q has to be stored.

5 Design decisions

In this section, we discuss the design and parameter decisions for our Niederreiter implementation on reconfigurable hardware. A primary goal of our design is high-performance, a secondary reasonable hardware costs.

5.1 Parameter selection

With the implementation of our Niederreiter cryptosystem, we aim to provide 80 bit of equivalent symmetric security, i.e., protection that is comparable to the security of ECC and RSA with approximately 160 bit and 1,024 bit,

Table 1 Security parameters for Niederreiter cryptosystems

Security level	Parameters (n, k, t), errors added	Size K_{pub} in KBits	Size $K_{sec}(g(x) \mid \mathcal{L} \mid M^{-1})$ KBits
Short-term (60 bit)	(1,024, 644, 38), 38	239	(0.37| 10| 141)
Mid-term I (80 bit)	(2,048, 1,751, 27), 27	507	(0.29| 22| 86)
Mid-term II (128 bit)	(2,690, 2,280, 56), 57	913	(0.38| 18| 164)
Long-term (256 bit)	(6,624, 5,129, 115), 117	7, 488	(1.45| 84| 2,183)

respectively. This level of security is still considered sufficient for mid-term security applications providing a reasonable cost-performance ratio and thus suitable for most embedded systems. To achieve this level of security, we selected the parameters $m = 11, n = 2{,}048, t = 27, k \geq 1{,}751$ resulting in a private and public key size of 13.5 and 63 KBytes to be stored on the device. This amount of memory is available in each Xilinx FPGA larger than the low-cost Spartan-3 XC3S1000, Virtex-5 XC5VLX30 and Virtex-6 XC6VLX75T, respectively [66]. The above security level was originally proposed to minimize public key size for a given security level and not to maximize performance. However, we stick to these parameters to be comparable with the existing code-based implementations. For practical purposes, we fixed the size of a message block to 192 bits which can be encoded into an appropriate error vector in any case. Note that the constant-weight encoding algorithm requires an input of variable length to produce a constant-weight output. Experiments showed that on average 210 bits are required to construct a valid constant-weight word. Fixing the input message to 192 bits and adding random bits as required, makes the algorithm practicable without leaking any security-relevant information. The secret key consisting of the Goppa polynomial $g(x)$ or $g^2(x)$, the support $\mathcal{L}$ and the inverse scrambling matrix S^{-1} (for Patterson decoding) or transforming matrix H_2 (for Berlekamp–Massey decoding). They are stored as part of the bitstream file which configures the FPGA. Because only the Spartan3-AN class from Xilinx offers internal Flash memory to store the bitstream internally, appropriate actions have to be taken to protect the bitstream when storing it in external memory. Note however, that also the Spartan3-AN does also not offer perfect security: Spartan3-AN FPGAs are actually assembled as stacked-die (i.e., a Flash memory on top of a separate die providing the reconfigurable logic), so an attacker can simply open the case and tap the bonding wires between the two dies to get access to the configuration data as well as the secret key. Therefore, it is mandatory to enable bitstream encryption using AES-256 which is available for larger Xilinx Spartan-6 and all Xilinx Virtex-FPGAs from Virtex-4. The (larger) public key can be stored either in internal or external memory since it does not require special protection. For our implementation, we opted to store the public key in internal BRAMs to allow immediate access for high-performance encryption.

5.2 Constant-weight encoding

Before encrypting a message with Niederreiter's cryptosystem, the message has to be encoded into an error vector. More precisely, the message needs to be transformed into a bit vector of length n and constant weight t. There exist quite a few encoding algorithms (e.g., [18,25,56]), however, they are not directly applicable to the restricted environment of embedded systems and hardware. In this work we unfolded the recursive algorithm proposed in [57] so that it can run iteratively by a simple state machine. During the encoding operation in [57], one has to compute a value $d \approx \frac{ln(2)}{t} \cdot (n - \frac{t-1}{t})$ to determine how many bits of the message are encoded into the distance to the next one-bit on the error vector. Many embedded (hardware) systems do not have a dedicated floating-point and division unit so these operations should be replaced. We therefore substituted the floating point operation and division by a simple and fast table lookup (see [32] for details). Since we still preserve all properties from [57], the algorithm will still terminate with a negligible loss in efficiency. The encoding algorithm suitable for embedded systems is given in Algorithm 14.

Algorithm 14 ENCODE A BINARY STRING IN A CONSTANT- WEIGHT WORD (BIN2CW)

Require: n, t, binary stream B
Ensure: $\Delta[0, \ldots, t-1]$
1: $\delta = 0, index = 0$
2: **while** $t \neq 0$ **do**
3: **if** $n \leq t$ **then**
4: $\Delta[index{+}{+}] = \delta$
5: $n- = 1, t- = 1, \delta = 0$
6: **end if**
7: $u \leftarrow uTable[n, t]$
8: $d \leftarrow (1 << u)$
9: **if** $read(B, 1) = 1$ **then**
10: $n- = d, \delta+ = d$
11: **else**
12: $i \leftarrow read(B, u)$
13: $\Delta[index{+}{+}] = \delta + i$
14: $\delta = 0, t- = 1, n- = (i + 1)$
15: **end if**
16: **end while**

The constant-weight decoding algorithm was adapted in a similar way, and is presented in Algorithm 15.

5.3 Inherent side channel resistance

Some research had been done regarding side channels in code-based cryptography, however, all solely focused on

Algorithm 15 DECODE A CONSTANT-WEIGHT WORD TO A BINARY STRING (CW2BIN)

Require: $n, t, \Delta[0, \ldots, t-1]$
Ensure: binary stream B
$\delta = 0, index = 0$
while $t \neq 0$ AND $n > t$ **do**
  $u \leftarrow uTable[n, t]$
  $d \leftarrow (1 << u)$
  if $\Delta[index] \geq d$ **then**
    $Write(1, B)$
    $\Delta[index] -= d$
    $n -= d$
  else
    $\delta = \Delta[index{+}{+}]$
    $Write(0|\delta, B)$
    $n -= (\delta + 1), t -= 1$
  end if
end while

implementations of McEliece encryption [35,47,59,61,62]. The advantage of Niederreiter in contrast to McEliece is that the ciphertext consists not only of a codeword with randomly added errors. Fault attacks cannot be easily performed by flipping random bits of the ciphertext assuming that the decoder either corrects one of the intentionally injected errors or fails to do so. For Niederreiter encryption, the ciphertext is a syndrome polynomial. Flipping random bits will result most likely in a decoding error without leaking any information. This renders all the attacks from [59,61,62] useless. Only power analysis attacks, like the one described in [35], which directly attacks the Goppa polynomial used in the decoding algorithm, are still possible. It also requires further investigation, if adoptions of attacks targeting the root search are possible.

6 Implementation

This section describes our implementation primarily targeting a recent Virtex-6 LX240 FPGA. Note that this device is certainly too large for our implementation but was chosen due to its availability on the Xilinx Virtex-6 FPGA ML605 Evaluation Kit for testing. Furthermore, we provide implementations for a Xilinx Spartan-3 and Xilinx Virtex-5 to allow fair comparisons with other work (cf. Table 2).

6.1 Encryption

The public key $\hat{H}$ is stored in an internal BRAM memory block and row-wise addressed by the output of the constant-weight encoder. Multiplying a binary vector with a binary matrix is equivalent to a XOR operation of each row with input vector bit equal to one. Since this operation is trivial, we focus on the implementation of the constant-weight encoding algorithm. Input data to our implementation (Fig. 1) is passed using a FIFO with a non-symmetric 8-to-1 bit aspect ratio. Hence, after a word with 8-bit length is written to the

Table 2 Comparison of our Niederreiter designs with single-core ECC and RSA implementations for 80-bit security

Scheme	Platform	Resources	Freq (MHz)	Time/Op	Cycles/byte
This work (enc)	Virtex6-LX240T	926 LUT/875 FF/17 BRAM	300	0.66 µs	8.3
This work (dec PAT)	Virtex6-LX240T	9,409 LUT/12,861 FF/9 BRAM	250	55.37 µs	576
This work (dec BM)	Virtex6-LX240T	5,567 LUT/9,166 FF/11 BRAM	220	49.72 µs	455
McEliece (enc) [22]	Spartan3-AN1400	1,044 LUT/804 FF/3 BRAM	150	1,070 µs	768
McEliece (dec) [22]	Spartan3-AN1400	9,054 LUT/12,870 FF/32 BRAM	85	21,610 µs	8,788
McEliece (dec) [27]	Spartan3-AN1400	2,979 slices	92	1,020 µs	430
This work (enc)	Spartan3-2000	1,252 LUT/869 FF/36 BRAM	150	1.3 µs	8.3
This work (dec PAT)	Spartan3-2000	15,559 LUT/13,608 FF/22 BRAM	95	145 µs	576
This work (dec BM)	Spartan3-2000	11,380 LUT/8,049 FF/26 BRAM	95	115 µs	455
McEliece (enc) [60]	Virtex5-LX110T	14,537 slices/75 BRAM	163	500 µs	389
McEliece (dec) [60]	Virtex5-LX110T	Combined with encryption	163	1,400 µs	1,091
McEliece (dec) [27]	Virtex5-LX110T	1,385 slices	190	500 µs	430
This work (enc)	Virtex5-LX50T	888 LUT/930 FF/18 BRAM	250	0.793 µs	8.2
This work (dec PAT)	Virtex5-LX50T	9,743 LUT/13,537 FF/13 BRAM	180	76.9 µs	576
This work (dec BM)	Virtex5-LX50T	7,821 LUT/9,106 FF/14 BRAM	170	64.4 µs	455
ECC-P160 (point mult.) [30]	Spartan-3 1000-4	5,764 LUT/767 FF/5 BRAM	40	5.1 ms	10,200
ECC-K163 (point mult.) [65]	Virtex5-LX110T	22,936 LUT/6,150 slices	250	5.48 µs	67.3
RSA-1024 random [31]	Spartan-3A	1,813 slices/1 BRAM	133	48.54 ms	50,436
RSA-1024 random [31]	Spartan-6	482 slices/1 BRAM	187	34.48 ms	50,373
RSA-1024 random [31]	Virtex-6	478 slices/1 BRAM	339	19.01 ms	59,258

PAT Patterson decoding, *BM* Berlekamp–Massey decoding

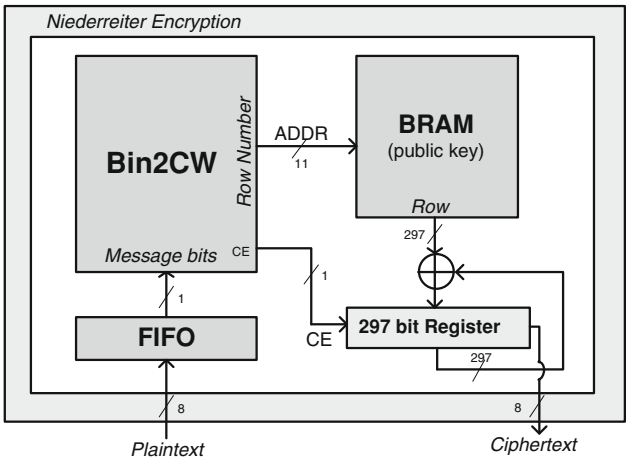

Fig. 1 Block diagram of the encryption process

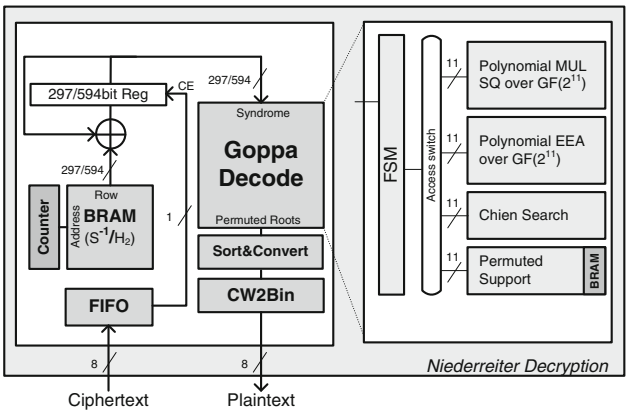

Fig. 2 Block diagram of the decryption process

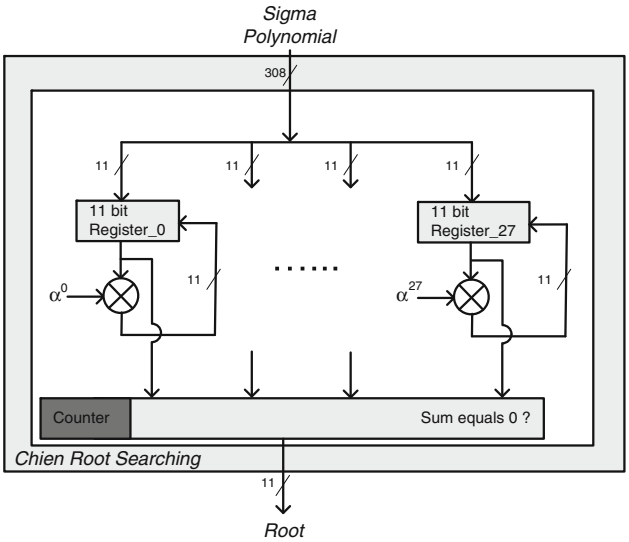

Fig. 3 Block diagram outlining the circuit of the Chien search

FIFO, it can be read out bit by bit. This is the equivalent to the binary stream reader presented in Algorithm 14. Its main part is implemented as a small finite state machine. Every time a valid $\Delta[i]$ has been computed, it is directly transferred to the vector-matrix-multiplier summing up the selected rows. By interleaving operations, we are able to process one bit from the FIFO at *every* clock cycle. After the last $\Delta[t]$ has been computed, only the last indexed row of $\hat{H}$ has to be added to the sum. Directly afterwards the encryption operation has finished and the ciphertext becomes available. Due to the very regular structure of the vector-matrix-multiplier and the small operands of the constant-weight encoder, we were able to achieve a high clock frequency of 300 MHz. Nevertheless, the logic inferred by the constant-weight encoder is still the bottleneck.

6.2 Decryption using the Patterson decoder

The first step in the decryption process is (Fig. 2) the multiplication by the inverse matrix S^{-1}. This 11-KByte large matrix is stored in an internal BRAM and addressed by an incrementing counter. Using this BRAM, the rows of the matrix are XORed into an intermediate register if the corresponding input bit of the ciphertext equals to one. After $(n-k) = 297$ clock cycles, this register contains the value $c' = S^{-1} \cdot c$ as shown in Algorithm 13. Now c' is passed on to the Goppa decoder which return the error locator polynomial $\sigma(x)$.

6.3 Decryption using the Berlekamp–Massey decoder

Instead of multiplying with S^{-1}, we have to multiply with the transformation matrix H_2 when using the Berlekamp–Massey decoder. As described above, we can use the same hardware architecture as for the Patterson decoder, with the only difference that the rows of the summed-up matrix are twice as large. Remember that we need to store only the last $n-k$ rows of H_2, because c is prefix with zeros (see Sect. 3.3.3). Because the same number of rows with twice the width have to be summed up this requires exactly the same number of cycles. The transformed syndrome is now passed to the Berlekamp–Massey decoder, which only consist of implementation of the EEA working modulo $g^2(x)$ and an stop value of $2\lfloor \frac{t}{2} \rfloor = 26$. This decoder also returns the error locator polynomial $\sigma(x)$.

Next, the roots of $\sigma(x)$ has to be computed in order to reveal the erroneous bit positions. Searching roots is quite a slow process that is highlighted by Fig. 3 showing our Chien search core. Decryption performance can be boosted by instantiating two or more of these cores in parallel and let them evaluate different support elements concurrently. Beside the additional management overhead in the controlling state machine, each of this cores requires additional 620 registers and 106 LUTs. We therefore use a sin-

gle core which evaluates one support element in two clock cycles and finishes the entire process after 4,098 clock cycles. Storing 28 look-up tables enables parallel execution of the multiplication but requires a significant amount of BRAM. Therefore, we decided to use 28 fully linearized multiplier instead, representing one output bit by a simple combinatorial circuit of the input bits.

Next, each root needs to be mapped to these bit positions for which we used a permuted support L as described above. Because the subsequent constant-weight decoding algorithm expects the distance between the error bits in ascending order, we appended a systolic implementation of bubble sort that returns sorted error positions. Simultaneously, the circuit computes the distance between two successive error positions. Finally, the error distances are translated into the binary message by a straightforward implementation of Algorithm 15 as presented in Sect. 6.1.

7 Results

We now present the results for our implementation on three different platforms to enable a fair comparison with other work. Note that most of the differences in the number of used resources for the same algorithm are due to architecture differences in the FPGA types, i.e., 4-input LUTs vs. 6-input LUTs and 18 KB BRAMs vs. 36 KB BRAMs in Spartan-3 and Virtex-5/6 FPGAs, respectively.

Encryption (Table 3) takes approximately 200 cycles or 0.66 µs on a Xilinx Virtex-6 FPGA. In applications where each encryption requires a different public key this necessitates the transfer of 1.5 million keys per second to the device. This translates to a communication interface that is capable to transfer $1.5 \cdot 10^6 \cdot 63$ Kbyte $\approx 774 \frac{\text{Gbyte}}{\text{s}}$ of data. Decryption requires 13,842 cycles and 10,940 cycles on average with Patterson decoding (Table 4) and Berlekamp–Massey decoding (Table 5), respectively. Due to the different clock rates achievable by both decoder implementations, this translates to an absolute runtime of 55 and 49 µs, respectively. Despite the slower clock frequency, Berlekamp–Massey decoding requires only 80 % of the runtime and only half of the resources compared to the implementation of the Patterson decoder.

As mentioned above, the public-key cryptosystems RSA-1024 and ECC-P160 are assumed[3] to roughly achieve an similar level of 80-bit symmetric security [21]. In Table 2 we finally compare our results to published implementations that target similar platforms (i.e., [1,22,30,31,60]). For a fair comparison with other existing implementations of code-based systems, we also implemented our code for Spartan-3 and Virtex-5 FPGAs.

Table 3 Implementation results of Niederreiter encryption with $n = 2{,}048, k = 1{,}751, t = 27$ after place and route (PAR)

Aspect	S3-2000	V5-LX50	V6-LX240
Slices	854 (2 %)	291 (4 %)	315 (1 %)
LUTs	1,252 (3 %)	888 (3 %)	926 (1 %)
FFs	869 (2 %)	930 (3 %)	875 (1 %)
BRAMs	36 (90 %)	18 (30 %)	17 (4 %)
Frequency	150 MHz	250 MHz	300 MHz
CW Encode		≈200 cycles	
Encrypt		Concurrently with CW encoding	

Table 4 Implementation results of Niederreiter decryption using Patterson decoding with $n = 2{,}048, k = 1{,}751, t = 27$ after PAR

Aspect	S3-2000	V5-LX50	V6-LX240
Slices	11,253 (54 %)	4,077 (56 %)	3,887 (10 %)
LUTs	15,559 (37 %)	9,743 (33 %)	9,409 (6 %)
FFs	13,608 (33 %)	13,537 (47 %)	12,861 (4 %)
BRAMs	22 (55 %)	13 (21 %)	9 (2 %)
Frequency	95 MHz	180 MHz	250 MHz
$c \cdot S^{-1}$		297 cycles	
$S(x)^{-1}$		4,310 cycles	
Solve key Eq.		4,854 cycles	
Search roots		4,098 cycles	
Sort&Convert		85 cycles	
CW decode		198 cycles	

Table 5 Implementation results of Niederreiter decryption using a Berlekamp–Massey decoder with $n = 2{,}048, k = 1{,}751, t = 27$ after PAR

Aspect	S3-2000	V5-LX50	V6-LX240
Slices	7,331 (35 %)	3,190 (44 %)	2,159 (5 %)
LUTs	11,380 (27 %)	7,821 (27 %)	5,567 (3 %)
FFs	8,049 (19 %)	9,106 (31 %)	9,166 (3 %)
BRAMs	26 (65 %)	14 (29 %)	11 (2 %)
Frequency	95 MHz	170 MHz	220 MHz
$syn \cdot H_2$		297 cycles	
Solve Key Eq.		6,262 cycles	
Search Roots		4,098 cycles	
Sort&Convert		85 cycles	
CW Decode		198 cycles	

[3] According to [21], RSA-1248 actually corresponds to 80-bit symmetric security. However, no implementation results for embedded systems are available for this key size.

8 Conclusions

In this work, we demonstrated the performance that can be achieved with an efficient FPGA-based implementation of Niederreiter's code-based public-key scheme. Besides practical plaintext size and smaller public keys, the very high performance with more than 1.5 million encryption and 17,000 decryption operations per second, respectively, renders the Niederreiter encryption an interesting candidate for security applications for which high throughput and many public key encryptions per second are required (and hybrid encryption should be avoided).

Acknowledgments The work described in this paper has been supported in part by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. This work has been also been supported in part by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009).

References

1. Bailey, D.V., Coffin, D., Elbirt, A., Silverman, J.H., Woodbury, A.D.: NTRU in Constrained Devices. In: Cryptographic Hardware and Embedded Systems—CHES 2001. LNCS, vol. 2162, pp. 262–272 (2001)
2. Berger, T.P., Cayrel, P.-L., Gaborit, P., Otmani, A.: Reducing key length of the McEliece cryptosystem. In: Proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology, AFRICACRYPT '09, pp. 77–97. Springer, Berlin (2009)
3. Berlekamp, B.: Nonbinary BCH decoding. IEEE Trans Inf Theory **14**(2), 242 (1968)
4. Berlekamp, E.: Goppa Codes. IEEE Trans. Inf. Theory **IT-19**(5) (1973)
5. Berlekamp, E.R.: A survey of coding theory. J. R. Stat. Soc. Ser. A (General) **135**(1) (1972)
6. Bernstein, D.J.: List decoding for binary Goppa codes. In: Proceedings of the Third International Conference on Coding and Cryptology, IWCC'11, pp. 62–80. Springer, Berlin (2011)
7. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems (2009). http://bench.cr.yp.to
8. Bernstein, D.J., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. In: Proceedings of the International Workshop on Post-Quantum Cryptography-PQCrypto '08. LNCS, vol. 5299, pp. 31–46. Springer, Berlin (2008)
9. Biswas, B., Herbert, V.: Efficient root finding of polynomials over fields of characteristic 2. In: WEWoRC 2009 (2009)
10. Biswas, B., Sendrier, N.: McEliece crypto-system: a reference implementation
11. Bogdanov, A., Eisenbarth, T., Rupp, A., Wolf, C.: Time-area optimized public-key engines: MQ-cryptosystems as replacement for elliptic curves? In: Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems-CHES 2008, vol. 5154. LNCS, pp. 45–61. Springer (2008)
12. Bouyukliev, I.G.: About the code equivalence. World Scientific, Hackensack, pp. 126–151 (2007)
13. Bras-Amors, M., O'Sullivan, M.E.: The Berlekamp-Massey algorithm and the Euclidean algorithm: A closer link. In: CoRR, Vol. abs/0908.2198 (2009)
14. Canteaut, A., Chabaud, F.: Improvements of the attacks on cryptosystems based on error-correcting codes (1995)
15. Cayrel, P.-L., Hoffmann, G., Persichetti, E.: Efficient implementation of a CCA2-secure variant of McEliece using generalized srivastava codes. In: Proceedings of the 15th International Conference on Practice and Theory in Public Key Cryptography, PKC'12, pp. 138–155. Springer, Berlin (2012)
16. Chang, K.: I.B.M. Researchers Inch Toward Quantum Computer. New York Times Article (2012).http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html?_r=1&hpw
17. Chien, R.: Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. IEEE Trans. Inf. Theor. **10**(4), 357–363 (2006)
18. Cover, T.: Enumerative source encoding **19**(1), 73–77 (1973)
19. Dinh, H., Moore, C., Russell, A.: McEliece and Niederreiter cryptosystems that resist quantum fourier sampling attacks. In: Proceedings of the 31st Annual Conference on Advances in Cryptology, CRYPTO'11, pp. 761–779. Springer, Berlin (2011)
20. Dornstetter, J.-L.: On the equivalence between Berlekamp's and Euclid's algorithms. IEEE Trans. Inf. Theory **33**(3), 428–431 (1987)
21. ECRYPT: Yearly report on algorithms and keysizes (2007–2008). Technical Report, D.SPA.28 Rev. 1.1, July 2008. http://www.ecrypt.eu.org/documents/D.SPA.10-1.1.pdf
22. Eisenbarth, T., Güneysu, T., Heyse, S., Paar, C.: Microeliece: McEliece for embedded devices. In: CHES '09: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, pp. 49–64. Springer, Berlin (2009)
23. Engelbert, D., Overbeck, R., Schmidt, A.: A summary of McEliece-type cryptosystems and their security. IACR Cryptol. ePrint Arch. **2006**, 162 (2006)
24. Faugere, J.-C., Otmani, A., Perret, L., Tillich, J.-P.: Algebraic cryptanalysis of McEliece variants with compact keys (2009)
25. Fischer, J.-B., Stern, J.: An efficient pseudo-random generator provably as secure as syndrome decoding. In: Advances in Cryptology EUROCRYPT 96, vol. 1070. Lecture Notes in Computer Science, pp. 245–255. Springer, Berlin (1996)
26. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. In: CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, pp. 537–554. Springer, London (1999)
27. Ghosh, S., Delvaux, J., Uhsadel, L., Verbauwhede, I.: A speed area optimized embedded co-processor for McEliece cryptosystem. In: 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 102–108 (2012)
28. Goppa, V.: A new class of linear correcting codes. Probl. Peredachi Inf. **6**(3), 24–30 (1969)
29. Gorenstein, D., Peterson, W.W., Zierler, N.: Two-error correcting Bose-Chaudhuri codes are quasi-perfect. Inf. Comput. **3**(3), 291–294 (1960)
30. Güneysu, T., Paar, C., Pelzl, J.: Special-purpose hardware for solving the elliptic curve discrete logarithm problem. ACM Trans. Reconfig. Technol. Syst. (TRETS) **1**(2), 1–21 (2008)
31. Helion Technology Inc.: Modular Exponentiation Core Family for Xilinx FPGA. Data Sheet, October 2008. http://www.heliontech.com/downloads/modexp_xilinx_datasheet.pdf
32. Heyse, S.: Low-Reiter: Niederreiter encryption scheme for embedded microcontrollers. In : Sendrier, N. (ed.) Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25–28, 2010. Proceedings, vol. 6061. Lecture Notes in Computer Science, pp. 165–181. Springer, Berlin (2010)
33. Heyse, S.: Implementation of McEliece based on quasi-dyadic Goppa codes for embedded devices. In: Yang, B.-Y. (ed.) Post-Quantum Cryptography, volume 7071 of Lecture Notes in Computer Science, pp. 143–162. Springer, Berlin (2011)

34. Heyse, S., Güneysu, T.: Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware. In: Prouff, E., Schaumont, P. (eds.) CHES, vol. 7428. Lecture Notes in Computer Science, pp. 340–355. Springer, Berlin (2012)
35. Heyse, S., Moradi, A., Paar, C.: Practical power analysis attacks on software implementations of McEliece. In: Sendrier, N. (ed.) Post-Quantum Cryptography, vol. 6061. Lecture Notes in Computer Science, pp. 108–125. Springer, Berlin (2010). doi:10.1007/978-3-642-12929-29
36. Hoffmann, G.: Implementation of McEliece using quasi-dyadic Goppa Codes. Bachelor thesis, TU Darmstadt (2011) http://www.cdc.informatik.tu-darmstadt.de/reports/reports/Gerhard_Hoffmann.bachelor.pdf
37. Horner, W.G.: A new method of solving numerical equations of all orders, by continuous approximation. Philosophical Transactions of the Royal Society of London **109**, 308–335 (1819)
38. Huber, K.: Note on decoding binary Goppa codes. Electron. Lett. **32**(2), 102–103 (1996)
39. Huffman, C.W., Pless, V.: Fundamentals of Error-Correcting Codes. Cambridge University Press, Cambridge (2003)
40. Kobara, K., Imai, H.: Semantically secure McEliece public-key cryptosystems-conversions for McEliece. In: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography, PKC '01, pp. 19–35, London, UK. Springer, Berlin (2001)
41. Lee, K.: Interpolation-based decoding of alternant codes. In: CoRR, vol. abs/cs/0702118 (2007)
42. Li, Y.X., Deng, R.H., wang, X.M.: On the equivalence of McEliece's and Niederreiter's public-key cryptosystems. IEEE Trans. Inf. Theor. **40**(1), 271–273 (2006)
43. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. DSN Prog. Rep. **42**(44), 114–116 (1978)
44. Minder, L.: Cryptography based on error correcting codes. PhD Thesis, École Polytechnique Fédérale de Lausanne (2007)
45. Misoczki, R., Barreto, P.S.: Compact McEliece keys from Goppa codes. In: Selected Areas in Cryptography: 16th Annual International Workshop (SAC 2009), pp. 376–392. Springer, Berlin (2009)
46. Misoczki, R., Barreto, P.S.: Selected areas in cryptography. In: Chapter Compact McEliece Keys from Goppa Codes, pp. 376–392. Springer, Berlin (2009)
47. Molter, H., Stöttinger, M., Shoufan, A., Strenzke, F.: A simple power analysis attack on a mceliece cryptoprocessor. J. Cryptogr. Eng. **1**(29–36) (2011). doi:10.1007/s13389-011-0001-3
48. Niebuhr, R., Cayrel, P.-L.: Broadcast attacks against code-based schemes. In: Armknecht, F., Lucks, S. (eds) WEWoRC, vol. 7242. Lecture Notes in Computer Science, pp. 1–17. Springer, Berlin (2011)
49. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Prob. Control Inf. Theory/Problemy Upravlen. Teor Inf. **15**(2), 159–166 (1986)
50. Overbeck, R., Sendrier, N.: Code-based cryptography. In: Bernstein, Daniel J., et al. (ed.) Post-Quantum Cryptography. First International Workshop PQCrypto 2006, Leuven, The Netherland, May 23–26, 2006, pp. 95–145. Selected Papers. Springer, Berlin (2009)
51. Patterson, N.: The algebraic decoding of Goppa codes. IEEE Trans. Inf. Theory **21**(2), 203–207 (1975)
52. Persichetti, E.: Compact McEliece keys based on Quasi-Dyadic Srivastava codes. IACR Cryptol. ePrint Arch. **2011**, 179 (2011)
53. Peterson, W.: Encoding and error-correction procedures for the Bose-Chaudhuri codes. IRE Trans. Inf. Theory **6**(4), 459–470 (1960)
54. Pierre-Louis Cayrel: Code-based cryptosystems: implementations. http://www.cayrel.net/research/code-based-cryptography/code-based-cryptosystems/
55. Pointcheval, D.: Chosen-Ciphertext security for any one-way cryptosystem. In: Imai, H., Zheng, Y. (eds.) Workshop on Practice and Theory in Public-Key Cryptography (PKC '00), vol. 1751. Lecture Notes in Computer ScienceSpringer, pp. 129–146. Melbourne, Australia (2000)
56. Sendrier, N.: Efficient generation of binary words of given weight. In: Cryptography and Coding, vol. 1025. Lecture Notes in Computer Science, pp. 184–187. Springer, Berlin (1995)
57. Sendrier, N.: Encoding information into constant weight words. In: Proceedings of International Symposium on Information Theory ISIT 2005, pp. 435–438 (2005)
58. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)
59. Shoufan, A., Strenzke, F., Molter, H., Stöttinger, M.: A timing attack against Patterson algorithm in the McEliece PKC. In: Lee, D., Hong, S. (eds.) Information, Security and Cryptology ICISC 2009, vol. 5984. Lecture Notes in Computer Science, pp. 161–175. Springer, Berlin (2010). doi:10.1007/978-3-642-14423-312
60. Shoufan, A., Wink, T., Molter, H.G., Huss, S.A., Strenzke. F.: A Novel processor architecture for McEliece cryptosystem and FPGA platforms. In: 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (2009)
61. Strenzke, F.: A timing attack against the secret permutation in the McEliece PKC. In: Sendrier, N. (ed.) Post-Quantum Cryptography, vol. 6061. Lecture Notes in Computer Science, pp. 95–107. Springer, Berlin (2010). doi:10.1007/978-3-642-12929-28
62. Strenzke, F., Tews, E., Molter, H., Overbeck, R., Shoufan, A.: Side channels in the McEliece PKC. In: 2nd workshop on post-quantum cryptography, pp. 216–229. Springer, Berlin (2008)
63. Sudan, M.: List decoding: algorithms and applications. SIGACT News **31**(1), 16–27 (2000)
64. Sugiyama, Y., Kasahara, M., Hirasawa, S., Namekawa, T.: A method for solving key equation for decoding goppa codes. Inf. Control **27**(1), 87–99 (1975)
65. Sutter, G., Deschamps, J., Imana., J.: Efficient elliptic curve point multiplication using digit-serial binary field operations. IEEE Trans. Ind. Electron. **60**(1), 217–225 (2013)
66. Xilinx Inc.: Data Sheets and Product Information for Xilinx Spartan and Virtex FPGAs. http://www.xilinx.com/support/