RUHR-UNIVERSITÄT BOCHUM

Development and Evaluation of a Code-based Cryptography Library for Constrained Devices

Hans Christoph Hudde

Master's Thesis. February 7, 2013.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Dipl.-Ing. Stefan Heyse

Abstract

Code-based cryptography is a promising candidate for the diversification of today's public-key cryptosystems, most of which rely on the hardness of either the Factorization or the Discrete logarithm problem. Both are known to be breakable using an efficient quantum algorithm due to Peter Shor. In contrast, Code-based cryptography is based on the problem of decoding unknown error-correcting codes, which is known to be $\mathcal{NP}$-hard.

There exist two basic schemes based on Code-based cryptography, which are named after their inventors Robert McEliece and Harald Niederreiter. Both share the problem of requiring huge key lengths compared to conventional cryptosystems such as RSA, which makes their implementation on embedded devices with very limited ressources challenging.

In this thesis, we present an implementation of modern variants of both schemes for AVR microcontrollers and evaluate several different methods of syndrome computation, decoding and root extraction. The implementation includes an adaption of the Berlekamp-Massey-Sugiyama algorithm to binary codes achieving the same level of error-correction as the Patterson algorithm. Moreover we implemented two conversions that turn the McEliece and Niederreiter schemes into CCA2-secure cryptosystems.

Our implementation is able to provide a security level of up to 128-bit on an ATxmega256 and hence is capable of fulfilling real-world security requirements. Moreover, the implementation outperforms comparable implementations of RSA and ECC in terms of data throughput and achieves a higher performance than previous implementations of the McEliece and Niederreiter cryptosystems. An optimal balance between memory usage and performance for specific use cases can be achieved using the flexible configuration by choosing the most suitable combination of precomputations, lookup tables or on-the-fly computations.

We demonstrate the feasibility of implementing a high-performance Code-based cryptography library on a low-cost 8-bit microcontroller and provide evidence that McEliece and Niederreiter can evolve to a fully adequate replacement for traditional cryptosystems.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Hans Christoph Hudde

Contents

1 Introduction

Two years after the first publication of an asymmetric cryptosystem — the Diffie-Hellman key exchange in 1976 — Robert McEliece developed an asymmetric cryptosystem based on error-correcting codes. Mainly due to its very long key size, McEliece's proposal has been neglected until the recent rise of interest in *Post-Quantum cryptography*.

This chapter introduces the reader to the background and objectives of this thesis. Section 1.1 motivates the need for *Post-Quantum cryptography* in general and the development of a Code-based cryptography library for constricted devices in particular. Section 1.2 gives an overview of already existing implementations of Code-based cryptosystems. Section 1.3 defines the goals of this thesis and points out the contributions of this work to the research process. Finally an outline of the thesis is given in Section 1.4.

1.1 Motivation

Impact of quantum computers on public-key cryptography Public-key cryptography is an integral part of today's digital communication. For example, it enables two or more parties to communicate confidentially without having a previously shared secret key. All popular public-key cryptosystems rely on the assumption that there is no efficient solution for the Integer Factorization problem or the Discrete Logarithm problem. According to Canteaut and Chabaud [CC95] public-key cryptography has become "dangerously dependent" on the difficulty of these two problems.

Unfortunately, both problems are known to be solvable in polynomial time on quantum computers using *Shor's algorithm* [Sho94]. Hence, a sufficiently large quantum computer would be able to break popular cryptosystems such as RSA, Elliptic Curve Cryptography (ECC) or ElGamal in polynomial time. This would "cause a tremendous shock to the worlds economy and security" wrote Overbeck [Ove07] in 2007, explaining the increased amount of research performed on the field of cryptosystems resisting quantum computers[1]. Such cryptosystems are dubbed *Post-Quantum cryptography*.

The first demonstration of Shor's algorithm was done by IBM in 2001[VSB+01] and used 7 qubits[2] to factorize the number 15. Ten years later, chinese researchers [XZL+11] were already able to factorize 143 using adiabatic quantum algorithms.

In May 2011, the quantum computing company *D-Wave Systems* claimed to have build the "world's first commercially available quantum computer" and was able to remove some of the doubts regarding the actual usage of quantum mechanical properties with an

[1] See [Nit] for a short and comprehensible introduction to quantum computers and their impact on cryptography.

[2] A qubit is a unit of quantum information, analogous to the classical bit, but allowing a superposition of its states.

Public-key cryptosystem	Example	Year
Code-based cryptography	McEliece encryption scheme	1978
Hash-based cryptography	Merkle's hash-tree signature system	1979
Lattice-based cryptography	NTRU encryption scheme	1996
Multivariate-quadratic-equations	HFE signature scheme	1996

Table 1.1: List of post-quantum cryptography candidates according to [Ber09]

article published in the renowned science magazine *Nature* [JAG+11, tec12]. In January 2012, D-Wave researchers [BCM+12] report a successful computation using 84 qubits.

At the same time, IBM announced "major advances in quantum computing device performance" [IBM12], fueling speculations that quantum computers might be available in 10 to 15 years. In September 2012, a research team led by Australian engineers succeeded in isolating, measuring and controlling an electron belonging to a single silicon atom, creating a qubit formed of just a single atom. One month later, physicists David Wineland and Serge Haroche were rewarded with a Nobel Prize for their fundamental research on quantum computers.

Code-based cryptosystems Given this pace of progress[3] regarding both theoretical and practical results of quantum computing, there is clearly a need for cryptosystems persisting in the world of quantum computers.

Fortunately there are at least four types of public-key cryptosystems, shown in Table 1.1, that are – to the best of our knowledge – not affected by quantum computers, at least not in a way as devastating as the systems affected by Shor's algorithm. Additionally, there is no hint that any popular symmetric cryptosystem is affected by quantum computers in a way that cannot be compensated by larger key sizes.

Studying these schemes is not only advantageous because it allows cryptographers to proactively deal with the challenges of post-quantum world, but also because it allows a greater diversification of cryptosystems. Building confidence in the security of alternative cryptosystems is a process spanning many years, and it is best to start this process early and without the pressure of a near breakdown of cryptography, whether it comes from quantum computers or any other kind of attack.

This thesis will concentrate on code-based cryptosystems, such as the McEliece [McE78] or Niederreiter [Nie86] scheme. Both are based on the problem of decoding a general linear code, which is known to be $\mathcal{NP}$-hard [BMvT78], hence it "presumably cannot be solved more efficiently with quantum computers" [Ove07] than with classical ones. The original McEliece scheme based on Goppa Codes has, in contrast to most other proposed variants [EOS06], resisted cryptanalysis so far. This means that while some attacks such as [Ber10, BLP08] require longer key sizes, the system as a whole remains unbroken.

By now there are also signature schemes based on McEliece-type cryptosystems, as well as conversions making McEliece and Niederreiter secure under strong security notions

[3] Wikipedia provides a timeline of quantum computing at `http://en.wikipedia.org/wiki/Timeline_of_quantum_computing`

like Indistinguishability under Adaptive Chosen Ciphertext Attacks (IND-CCA2) [KL07]. Furthermore McEliece and Niederreiter perform very well in comparative benchmarks [Be] and have a time complexity of $O(n^2)$, while RSA for example has a time complexity of $O(n^3)$ [MB09].

This makes code-based cryptography a promising candidate for a broad range of cryptographic applications.

Nevertheless there is one important drawback, namely the very large key length. For 80-bit security with RSA traditionally a 1024-*bit* key is assumed, while McEliece requires a key length of more than 437 *kilobyte* for equivalent security in the original McEliece scheme. However, in this thesis we employ a modern variant of the original scheme that reduces the key length to less than 70 kB.Further improvements exist, but must be carefully examined in order not to allow structural decoding attacks.

Embedded devices Given today's memory and network capacities the key length is not a problem for typical desktop systems, but it still is problematic for embedded devices. However, the need for security in embedded devices is continuously rising, be it in smartphones, electronic door locks, industrial controllers or medical equipment.

According to the public private partnership association ARTEMIS, which claims to represent the interests of industry and the research community in the field of embedded devices, "98% of computing devices are embedded"[4] devices. This applies to consumer electronics as well as industrial applications. Hence, efficient implementations for embedded devices are indispensable to achieve acceptance for code-based cryptosystems in practice.

1.2 Existing implementations

To the best of our knowledge only few publicly available implementations of code-based cryptosystems exist. [Cay] provides a link list of some of the implementations online.

Hybrid McEliece Encryption Scheme (HyMES) [BS08b, BS08a] is a C implementation of a McEliece variant for 32-bit architectures, published under GNU Lesser General Public License (LGPL). It it called Hybrid because it contains elements of both McEliece and Niederreiter in order to improve the information rate, and it reduces the key size by using a generator matrix in row echelon form.

FlexiProvider [fle] is a toolkit for the Java Cryptography Architecture implementing both McEliece and Niederreiter with CCA2-secure variants. It was developed by the Theoretical Computer Science Research Group at TU Darmstadt and is licensed under GNU General Public License (GPL).

Eric Arnoult [Arn10] implemented a binary and q-ary version of Niederreiter in C for 32-bit architectures.

Thomas Risse provides an implementation of McEliece [Ris11] for the python-based mathematical software Sage[5].

[4] `http://www.artemis-ju.eu/embedded_systems`
[5] `http://www.sagemath.org/`

Codecrypt [cod12] is a C++ implementation of McEliece, CFS signatures and quasi-dyadic Goppa codes. As of now it is in pre-alpha state and comes without any documentation. It is published under LGPL.

Flea ("flexible lightweight extensible algorithms", [Str12]) is an open source cryptographic library in C based on HyMES and is considered a pre-release as of now. Both Codecrypt and flea were not yet available at the start of this thesis.

An i386 assembler implementation of McEliece was done by Preneel *et al.* [BPV92], but is not freely available.

Cayrel *et al.* [CHP12] proposed and implemented an CCA2-secure variant of McEliece that does not require Constant Weight (CW) encoding. However the source code is not freely available.

The C-implementation [pro] is said to be the first publicly available implementation, however the available code is undocumented and obfuscated. Nevertheless, it has been used for a decentralized platform for censorship-resistant communication called "Entropy" similar to "The Freenet Project". This was one of the rare instances were code-based cryptography has actually been used in practice, but it has been discontinued since 2004.

There are several implementations of McEliece-type cryptosystems for different platforms by the Chair for Embedded Security at the Ruhr-University Bochum.

Heyse implemented both McEliece [Hey08, EGHP09] and Niederreiter [Hey10] for 8-bit AVR microprocessors as well as on a Xilinx Spartan-3 FPGA [Hey09, EGHP09]. The implementations perform very well, but the 8-bit implementation requires external memory to hold the public key. This thesis builds heavily upon the work of Heyse, but among other things removes the need for external memory.

The requirement for external memory has already been removed using a different approach by Heyse and Paustjan [Pau10, Hey11] using quasi-dyadic Goppa codes, which allow a compact representation of the public key. Their quasi-dyadic McEliece implementation for 8-bit AVR microcontrollers also includes a CCA2-secure conversion.

More recently, Tendyck [Ten12] implemented McEliece for an ARM CPU using the NEON instruction set, which effects in an considerable performance gain for encryption.

1.3 Contribution

While some implementations of code-based cryptography on constricted devices already exist, they mostly focus on one specific variant and leave room for optimizations. This thesis aims to provide an extensible code-based crypography library for microcontrollers. It provides implementations of both McEliece and Niederreiter including two different methods of CCA2-secure conversions. Furthermore it includes decoding via either Patterson or Berlekamp-Massey algorithm and several variants for root extraction and syndrome calculation. Finally, all the variants are evaluated and compared to existing implementations and to traditional public-key cryptosystems. Therefore, the thesis and implementation help in advancing Code-based cryptography and provide valuable hints for future implementations regarding the decision which approaches are most promising.

1.4 Outline

The remainder of the thesis is structured as follows: Chapter 2 provides a high-level introduction to Code-based cryptography. It discusses the McEliece and Niederreiter cryptosystem and examines security aspects and practical issues. Chapter 3 provides the reader with basic definitions and methods of Coding theory. Moreover, it explains the construction of the important class of Goppa codes and provides algorithms for decoding and root extraction, some of which apply to a very broad range of code classes. The implementation of these algorithms with regard to the embedded target platforms is discussed in Chapter 4. Furthermore, it covers the conversion of McEliece and Niederreiter into CCA2-secure cryptosystems. Chapter 5 evaluates the implementation with regard to memory usage and execution time of the implemented variants. Finally, a conclusion in Chapter 6 summarizes the results and provides an outlook to future research directions.

2 Code-based cryptography

This chapter introduces the reader to the basics of code-based cryptography and discusses the cryptographic and practical strengths and weaknesses of the presented systems. Section 2.1 provides a rough introduction to the fundamentals of linear codes and the basic mechanisms of Code-based cryptography, followed by a presentation of currently recommended security parameters in Section 2.2. Then, the *Classical* (Section 2.3) and *Modern* (Section 2.4) version of McEliece and of Niederreiter (Section 2.5) are discussed, without yet delving into the finer details of Coding theory. In Section 2.6 security aspects of Code-based cryptography are discussed, including the relation of Code-based cryptography to the General Decoding problem, important attack types and the notion of Semantic security. Finally, attempts at reducing the key length are briefly reviewed in Section 2.7.

2.1 Overview

Linear codes Error-detection and correction techniques have been used since the early days of telecommunication and computing to allow the transmission of information over a noisy channel, starting with the Hamming code invented by Richard Hamming in 1950. The basic idea is adding redundancy to the message, which can be used by the receiver to check the consistency and correct the errors to some extent.

Linear block codes belong to the most widely used class of error-detecting and error-correcting codes. They use a *generator matrix* G to transform a message vector m into a codeword c, and a *parity check matrix* H derived from G to check the consistency of a codeword. If $s = c \cdot H = 0$ the codeword is free of detectable errors. Otherwise s, which is called *syndrome*, can be used to identify the error positions and error values.

A linear code of length n, dimension k, and minimal code word distance[App. B.2.2] d is called an $[n, k, d]$ code. For example, a binary linear $[n, k, d]$ code has a length of n bits, it holds k bits payload and $n - k$ bits redudancy, and the minimum hamming distance[App. B.2.1] of its codewords is d. A code is efficient if it maximizes d for a given n and k. In general, such a code is able to correct $t = \lfloor \frac{d-1}{2} \rfloor$ errors.

Goppa codes are an important class of linear codes. They are defined using a polynomial $g(z)$ over the Galois Field $GF(p^m)$ with a so-called *support* L being a subset of elements of $GF(p^m)$ that has no roots in $g(z)$. They will be discussed in more detail in Chapter 3.

Cryptography based on error-detecting codes Public-key encryption schemes use two mathematically linked keys to encrypt and decrypt messages. The *public key* can only

be used to encrypt a message and the *secret key* is required to decrypt the resulting ciphertext. Such schemes can be specified by giving a triple of algorithms: key generation, encryption and decryption.

All popular public-key cryptosystems are based on one-way functions[App. B.2.3]. A one-way function can informally be defined as a function that can be computed efficiently for every input, but is hard to revert in the sense of complexity theory. A special case of a one-way function is a trapdoor function, which is hard to revert in general, but easy to revert with the help of some secret additional information.

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 pattern. 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.

The original proposal by Robert McEliece suggested the use of binary Goppa codes, but in general any other linear code could be used. While other types of code may have advantages such as a more compact representation, most proposals using different codes were proven less secure[1]. The Niederreiter cryptosystem is an indepently developed variation of McEliece which is proven to be equivalent in terms of security [LDW06].

In this thesis, the term *Classical* McEliece or Niederreiter is used to identify the original cryptosystem as proposed by its author. The term *Modern* is used for a variant with equivalent security that we consider more appropriate for actual implementations. While this chapters introduces the reader to both variants, throughout the remainder of the thesis we will always consider only the Modern variant.

2.2 Security parameters

The common system parameters for the McEliece and Niederreiter cryptosystem consist of code length n, error correcting capability t and the underlying Galois Field $GF(p^m)$. The length of the information part of the codeword is derived from the other parameters as $k = n - m \cdot t$.

In [McE78] McEliece suggested the use of binary ($p = 2$) Goppa codes with $m = 10$, $n = 1024$ and $t = 50$, hence $[n, k, d] = [p^m, n{-}m{\cdot}t, 2{\cdot}t{+}1] = [1024, 524, 101]$. The authors of [MVO96] note that $t = 38$ maximizes the computational complexity for adversaries without reducing the level of security.

There is no simple criterion neither for the choice of t with respect to n [EOS06] nor for the determination of the security level of a specific parameter set. Niebuhr *et al.* [NMBB12] propose a method to select optimal parameters providing an adequate security until a certain date. Due to newly discovered or improved attacks, the assumed

[1] See for example [Min07, OS09]

Security	m	[n, k, d]-code	t	Approximate size of systematic generator matrix ($k \cdot (n-k)$ Bit)
Insecure (60-bit)	10	[1024, 644, 77]	38	239 kBit
Short-term (~80-bit)	11	[1632, 1269, 67]	33	450 kBit
Short-term (80-bit)	11	[2048, 1751, 55]	27	507 kBit
Mid-term (128-bit)	12	[2960, 2288, 113]	5	1501 kBit
Long-term (256-bit)	13	[6624, 5129, 231]	115	7488 kBit

Table 2.1: Parameters sets for typical security levels according to [BLP08]

security level for the originally suggested parameters by McEliece fell from arround 2^{80} in 1986 to $2^{59.9}$ in 2009 [FS09]. Table 2.1 shows parameter sets for typically used security levels. The corresponding key lengths depend on the respective cryptosystem variant and the storing method and will be discussed in Section 2.7 after the presentation of the cryptosystems.

2.3 Classical McEliece cryptosystem

In this section, the algorithms for key generation, encryption and decryption as originally proposed by Robert McEliece [McE78] in 1978 are presented.

2.3.1 Key generation

As shown in Alg. 2.3.1 the key generation algorithm starts with the selection of a binary Goppa code capable of correcting up to t errors. This is done by randomly choosing a irreducible Goppa polynomial of degree t. Then the corresponding generator matrix G is computed, which is the primary part of the public 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 generated randomly and multiplied with G to form $\hat{G} = S \cdot G \cdot P$. S is chosen to be invertible and the permutation P effectively just reorders the columns of the codeword, which can be reversed before decoding. Hence $\hat{G}$ is still a valid generator matrix for an equivalent[2] code $\mathcal{C}$. $\hat{G}$ now serves as the public key and the matrices G, S and P – or equivalently S^{-1}, P^{-1} – compose the secret key.

Canteaut and Chabaud note in [CC95] 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. But not directly revealing the plaintext bits provides no security beyond a weak form of obfuscation. CCA2-secure conversions as shown in Section 2.6.3 need to be applied to address this problem and allow the intentional use of a systematic matrix as in Modern McEliece.

[2] See [Bou07] for details on code equivalence.

Algorithm 2.3.1: Classical McEliece: Key generation

Data: Fixed system parameters t, n, p, m
Result: 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** $K_{sec} = (G, S^{-1}, P^{-1}), K_{pub} = (\hat{G})$

2.3.2 Encryption

The McEliece encryption is 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 Alg. 2.3.2. The multiplication adds redundancy to the codeword, resulting in a message expansion from k to n with overhead $\frac{n}{k}$.

Algorithm 2.3.2: Classical McEliece: Encryption

Data: Public key $K_{pub} = (\hat{G})$, message M
Result: 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** $c = m \cdot \hat{G} + e$

2.3.3 Decryption

The McEliece decryption shown in Alg. 2.3.3 consists mainly of the removal of the applied errors using the known 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 makes decryption much slower than encryption. Details are given in Chapter 3.

Decryption works correctly despite of the transformation of the code $\mathcal{C}$ because the following equations hold:

$$\hat{c} = c \cdot P^{-1} \tag{2.1}$$
$$= (m \cdot \hat{G} + e) \cdot P^{-1} \tag{2.2}$$
$$= (m \cdot S \cdot G \cdot P + e) \cdot P^{-1} \tag{2.3}$$
$$= m \cdot S \cdot G \cdot P \cdot P^{-1} + e \cdot P^{-1} \tag{2.4}$$
$$= m \cdot S \cdot G \cdot + e \cdot P^{-1} \tag{2.5}$$

Remember from Section 2.3.1 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.

Algorithm 2.3.3: Classical McEliece: Decryption

Data: Ciphertext c of length n, private key $K_{sec} = (G, S^{-1}, P^{-1})$
Result: Message M

1 Compute $\hat{c} = c \cdot P^{-1}$
2 Compute the syndrome s corresponding to c
3 Obtain $\hat{m}$ of length k from s using the decoding algorithm $\mathcal{D}_{Goppa}(\hat{c})$ for code $\mathcal{C}$
4 Compute $m = \hat{m} \cdot S^{-1}$
5 Represent m as message M
6 **return** M

2.4 Modern McEliece cryptosystem

In order to reduce the memory requirements of McEliece and to allow a more practical implementation, the version that we call Modern McEliece opts for the usage of a generator matrix in systematic form. In this case, the former scrambling matrix S is chosen to bring the generator matrix to systematic form. Hence, it does not need to be stored explicitly anymore. 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 $g(z)$ 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

stresses that "such a conversion is needed anyway" [OS09]. Section 2.6.3 discusses this requirement in greater detail.

The algorithms shown in this section present the Modern McEliece variant applied to Goppa codes.

2.4.1 Key generation

Alg. 2.4.1 shows the key generation algorithm for the Modern McEliece variant.

Algorithm 2.4.1: Modern McEliece: Key generation

Data: Fixed system parameters t, n, p, m
Result: private key K_{sec} , public key K_{pub}
1 Select a random Goppa polynomial $g(z)$ of degree t over $GF(p^m)$
2 Randomly choose n elements of $GF(p^m)$ that are not roots of $g(z)$ as the *support* L
3 Compute the $k \times n$ 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 systematic $n \times (n-k)$ generator matrix G_{sys} from H_{sys}
6 **return** $K_{sec} = (L, g(z)), K_{pub} = (G_{sys})$

It starts with the selection of a random Goppa polynomial $g(z)$ of degree t. The support L is then chosen randomly as a subset of elements of $GF(p^m)$ that are not roots of $g(z)$. Often n equals p^m and $g(z)$ 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 the use of G_{sys}.

Using a relationships discussed in Section 3.3.2, the parity check matrix H is computed according to $g(z)$ 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.

Table 2.2 illustrates the relationship between the public and private versions of generator matrix, parity check matrix and support.

2.4.2 Encryption

Encryption in Modern McEliece (see Alg. 2.4.2) is identical to encryption in Classical McEliece, but can be implemented more efficiently, because 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 2.4.2: MODERN MCELIECE: ENCRYPTION

Data: Public key $K_{pub} = (G_{sys} = (I_k|Q))$, message M
Result: Ciphertext c of length n
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** $c = m \cdot G_{sys} + e = (m \,||\, m \cdot Q) + e$

2.4.3 Decryption

Decryption in the Modern McEliece variant shown in Alg. 2.4.3 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 during decoding. The 'scrambling' does not need to be reversed neither, because the information bits can be readly directly from the first k bits of the codeword.

Algorithm 2.4.3: MODERN MCELIECE: DECRYPTION

Data: Ciphertext c of length n, private key $K_{sec} = (L, g(z))$
Result: Message M
1 Compute the syndrome s corresponding to c
2 Obtain m of length k from s using the decoding algorithm $\mathcal{D}_{Goppa}(c)$ for code $\mathcal{C}$
3 Represent m as message M
4 **return** M

This works correctly and is security-equivalent to the Classical version of McEliece because all modifications can be expressed explicitly with S and P as shown in Table 2.2. G_{sys} is still a valid generator matrix for an equivalent code $\mathcal{C}$.

Alg.	Classical McEliece	Modern McEliece
Key.	$\hat{G} = S \cdot G \cdot P$	$L_{sec} = L_{pub} \cdot \hat{P}$
		$L_{sec} \Rightarrow H_{sys} = (Q^{\mathrm{T}}\|I_{n-k}) = \hat{S} \cdot H \cdot \hat{\hat{P}}$
		$H_{sys} \Rightarrow G_{sys} = (I_k\|Q) = S \cdot G \cdot P$
Enc.	$c = m \cdot \hat{G} + e$	$c = m \cdot G_{sys} + e = (m\|\|m \cdot Q) + e$
Dec.	$\hat{c} = c \cdot P^{-1}, \hat{m} = \mathcal{D}_{Goppa,L_{pub}}(\hat{c}), m = \hat{m} \cdot S^{-1}$	$\hat{m} = \mathcal{D}_{Goppa,L_{sec}}(c) = m\|\|\text{parity} = \hat{m} \cdot S^{-1}$

Table 2.2: Comparison of the modern and Classical version of McEliece

2.5 Niederreiter cryptosystem

Eight years after McEliece's proposal, Niederreiter [Nie86] developed a similar cryptosystem, apparently not aware of McEliece's work. 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 nevertheless in all cases, this has no effect on the security, but it results in smaller plaintext blocks, which is often advantageous. Moreover, Niederreiter uses the syndrome as ciphertext instead of the codeword, hence moving some of the decryption workload to 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 the omittance of 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.

2.5.1 Key generation

Key generation works similar to McEliece, but does not require the computation of the generator matrix. Alg. 2.5.1 shows the Classical key generation algorithm for the Niederreiter cryptosystem, while Alg. 2.5.2 presents the Modern variant 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 2.5.1: Classical Niederreiter: Key generation

Data: Fixed system parameters t, n, p, m
Result: 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 $K_{sec} = (H, S^{-1}, P^{-1}), K_{pub} = (\hat{H})$

Algorithm 2.5.2: Modern Niederreiter: Key generation

Data: Fixed system parameters t, n, p, m
Result: private key K_{sec} , public key K_{pub}

1 Select a random Goppa polynomial $g(z)$ of degree t over $GF(p^m)$
2 Randomly choose n elements of $GF(p^m)$ that are not 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 $K_{sec} = (L, g(z), S^{-1}), K_{pub} = (H_{sys})$

2.5.2 Encryption

For encryption, the message M needs to be represented as a CW word of length n and hamming weight t. There exist several techniques for CW encoding, one of which will be presented in Section 4.7. The CW encoding is followed by a simple vector-matrix multiplication.

Encryption is shown in Alg. 2.5.3. It is identical for the Classical and Modern variant apart from the fact that the multiplication with a systematic parity check matrix can be realized more efficiently.

Algorithm 2.5.3: NIEDERREITER: ENCRYPTION

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

2.5.3 Decryption

For the decoding algorithm to work, first the 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 Alg. 2.5.4, the error vector after decoding is still permuted, so it needs to be multiplied by P^{-1}. In the Modern variant shown in Alg. 2.5.5, 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 2.5.4: CLASSICAL NIEDERREITER: DECRYPTION

Data: Ciphertext c of length $(n - k)$, private key $K_{sec} = (H, S^{-1}, P^{-1})$
Result: 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 $e = P^{-1} \cdot \hat{e}$ of length n and weight t
4 Represent e as message M
5 return M

2.6 Security

This section provides an overview of the security of McEliece-type cryptosystems. First, the hardness of the *McEliece problem* is discussed. Then we give a rough overview on classical and side channel attacks. Finally the concept of *indistinguishability* and CCA2-secure conversions are introduced.

Algorithm 2.5.5: Modern Niederreiter: Decryption

Data: Ciphertext c of length $(n-k)$, private key $K_{sec} = (L, (g(z), S^{-1})$
Result: 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

2.6.1 Overview

The *McEliece problem* can be defined as the task of finding the message corresponding to a given ciphertext and public key according to the McEliece or Niederreiter cryptosystem.

According to Minder [Min07] there are mainly two assumptions concerning the security of the McEliece problem: The hardness of decoding a general unknown code, which is known to be $\mathcal{NP}$-hard [BMvT78], and the hardness of structural attacks reconstructing the underlying code.

- Obviously the McEliece problem can be broken by an adversary who is able to solve the General Decoding problem. On the other hand, solving the McEliece problem would presumably solve the General Decoding problem only "in a certain class of codes", since it allows only the decoding of a permutation-equivalent[3] code of a specific known code. Therefore "[w]e can not assume that the McEliece-Problem is $\mathcal{NP}$-hard" conclude Engelbert *et al.* in [EOS06]. Minder adds that $\mathcal{NP}$-hardness is a worst-case criterion and hence not very useful "to assess the hardness of an attack" [Min07]. Overbeck [OS09] points out several differentiations of the decoding problem, concluding that although there is no proof for the hardness of the McEliece problem, there is at least no sign that using McEliece-type cryptosystems with Goppa codes could 'fall into an easy case'.

- The hardness of reconstructing the underlying code given the generator matrix differs greatly across different codes. For example, the original McEliece using Goppa codes remains unbroken aside from key length adjustments, whereas the usage of Generalized Reed-Solomon Code (GRS) codes as suggested by Niederreiter turned out to be insecure due to structural attacks.

So far, there are "no known classical or quantum computer attacks on McEliece's cryptosystem which have sub-exponential running time" conclude Engelbert *et al.* in [EOS06].

2.6.2 Attacks

A public-key cryptosystem can be considered broken if it is feasible to extract the secret key or to decrypt a ciphertext without knowledge of the secret key. Note that we consider

[3] Note that attempts at more general transformations exist, see for example [BBC+11].

only attacks applicable to variants that apply a CCA2-secure conversion as described in Section 2.6.3.

Message security An adversary who is able to decode the syndrome $s = H \cdot (c + e)$ can decrypt ciphertexts of both McEliece and Niederreiter. This requires finding a linear combination of w columns of the parity check matrix H matching the syndrome s, where c is the codeword and e the error vector with hamming weight w. Syndrome decoding is known to be $\mathcal{NP}$-hard; the bruteforce complexity of this operation is $\binom{n}{w}$.

Information Set Decoding (IDS) reduces the brute-force search space using techniques from linear algebra and is often considered the "top threat" [BLP11] to McEliece-type cryptosystems. It essentially transfers the problem of syndrome decoding to the problem of finding a low-weight codeword. A basic form of this attack was already mentioned in the original proposal of the cryptosystem by McEliece in 1978. Ten years later, Lee and Brickell [LB88] systematized the concept and Stern [Ste89] discovered an information set decoding algorithm for random linear codes of length n that runs in $O(2^{0.05563n})$. This algorithm has been improved several times, for example by Canteaut [Can98] in 1998, by Bernstein *et al.* [BLP08] in 2008 and by May, Meurer *et al.* [MMT11, BJMM12] in 2011 and 2012, reducing the time complexity to $O(2^{0.0494n})$.

Statistical Decoding [Jab01] is a similar approach to information set decoding and tries to estimate the error positions by exploiting statistical information in the syndrome. Iterative Decoding [FKI07] is another similar variant which searches for a set of checksums generated by a particular key and then applies this set in an iterative bit flipping phase to every available message to test the key candidate. Although improvements to these methods exist, Engelbert *et al.* [EOS06] consider this type of attack infeasible.

Bernstein *et al.* stress in [BLP11] that even their "highly optimized attack [...] would not have been possible with the computation power available in 1978", when McEliece proposed his system, concluding that 30 years later the system has "lost little of its strength".

Key security Structural attacks typically aim at extracting the secret key from the public key or from plaintext/ciphertext pairs. For example, Sidelnikov and Shestakov [SS92] proposed a structural attack to GRS codes in 1992. Although Goppa codes are subfield subcodes of GRS codes, McEliece and Niederreiter using Goppa codes do not seem to be affected by this attack [EOS06]. This applies also to newer variants of the attack, like the extension by Wieschebrink [Wie10].

The security of McEliece-type cryptosystems is related to the problem of Code Equivalence: an adversary who is able to decide whether two generator matrices are code-equivalent may have an advantage in finding the secret key matrix. This can be accomplished using the Support Splitting Algorithm (SSA) by Sendrier [Sen00], which computes the permutation between two equivalent codes for Goppa codes and some other code classes. Attacking the McEliece cryptosystem using SSA requires the adversary to guess the secret generator matrix G, for example by testing all possible Goppa polynomials of the respective degree and checking the corresponding code using SSA.

This method is called called *Polynomial-searching attack* in [BLP11] and is considered infeasible for adequate security parameters. Using the SSA, Sendrier and Loidreau also discovered a family of weak keys [LS98] for McEliece, namely if it is used with Goppa codes generated by binary polynomials.

Petrank and Roth [PR97] propose a reduction of Code Equivalence to Graph Isomorphism, stating that even though the "Code Equivalence problem is unlikely to be $\mathcal{NP}$-complete", it is "also unlikely to be too easy". The uncertainty stems from the fact that although the Subgraph Isomorphism problem – a generalization of the Graph Isomorphism problem – is known to be $\mathcal{NP}$-complete, the computational complexity of Graph Isomorphism remains an open question[GJ79].

Variants of McEliece-type cryptosystems having highly regular structures that allow compact public key representations often fall to algebraic attacks. For example, the proposal of McEliece using quasi-cyclic codes by Berger *et al.* [BCGO09] has been broken by Otmani *et al.* [OTD10]. Another example is the attack by [Min07] against McEliece defined over elliptic curves.

Detailed overviews over all these and some other attacks are given for example by Engelbert, Overbeck and Schmidt [EOS06] or more recently by Niebuhr [Nie12].

Side channel attacks Side channel attacks attempt to extract secret information of a cryptosystem by analyzing information that a *specific implementation* leaks over side channels such as power consumption, electromagnetic emissions or timing differences. They represent a serious threat especially to devices in hostile environments, where an adversary has unconditional physical access to the device. Since this thesis is focused on embedded devices, this is probably the case for most real world use cases of this implementation.

Contrary to the previously discussed attacks, side channel attacks do not question the security of the cryptosystem itself, but only of the implementation. Nevertheless it is possible to identify attack vectors that are likely to occur in all implementations of a specific cryptosystem, for example if the system includes an algorithm whose duration depends strongly on the secret key.

The recent rise of interest in post-quantum cryptography also brought side channel analysis of McEliece-type cryptosystem more into focus and spawned several papers researching susceptibility and countermeasures. Strenzke *et al.* [Str10, Str11, SSMS10] published several papers on timing attacks against the secret permutation, syndrome inversion and the Patterson algorithm and also pointed out some countermeasures. Heyse, Moradi and Paar [HMP10] evaluated practical power analysis attacks on 8-bit implementations of McEliece provided by [Hey09]. More recently, Schacht [Sch12] evaluated timing and power analysis attacks and countermeasures with a focus on variants of the syndrome decoding algorithm.

A typical example for a side channel in McEliece based on a binary code is the bit flip attack: If an attacker toggles a random bit in the ciphertext and the bit happens to be an error bit, the decoding algorithm has one less error to correct. Without countermeasures, this typically results in a reduced runtime and hence allows the attacker to find the

complete error vector by toggling all bits one after another. Note that this attack cannot be applied straightforwardly to Niederreiter, since toggling a bit in a Niederreiter ciphertext typically renders it undecodable.

Research on side channels in code-based cryptosystems needs to be intensified, but the existing papers already provide valuable advice on common pitfalls and countermeasures. Although side channel attacks are not in the focus of this thesis, we will come back to the topic in the following chapters where necessary.

2.6.3 Ciphertext indistinguishability

The various notions of *ciphertext indistinguishability* essentially state that a computationally bounded adversary is not able to deduce *any* information about the plaintext from the ciphertext, apart from its length. The very strong security notion of *Indistinguishability under Adaptive Chosen Ciphertext Attacks (IND-CCA2)* includes the properties of *semantic security* and allows the adversary permanent access to a *decryption oracle* that he can use to decrypt arbitrary ciphertexts. The adversary chooses two distinct plaintexts, one of which is encrypted by the challenger to ciphertext c. The task of the adversary is now to decide to which of the two plaintexts c belongs, without using the decryption oracle on c. If no such an adversary can do better than guessing, the scheme is called CCA2-secure.

To fulfill the requirements of indistinguishability, encryption algorithms need to be probabilistic. Although the McEliece and Niederreiter encryption are inherently probabilistic, they are *not* inherently CCA2-secure – actually, major parts of the plaintext may be clearly visible in the ciphertext. This is especially true for McEliece with a systematic generator matrix, because then the matrix multiplication results in an exact copy of the plaintext to the codeword, just with an parity part attached. In this case, only the addition of the random error vector actually affects the value of the plaintext bits, changing a maximum of of t out of k bit positions. Therefore the plaintext "All human beings are born free and equal in dignity and rights[...]" may become "Al? huMaj beangs are0born free ?ld equal yn di?nltY and!rightq[...]", clearly leaking information. For McEliece without a systematic generator matrix and also for Niederreiter the same applies, although the information leak is less obvious.

Another problem solved by CCA2-secure conversions is the achievement of non-malleability, which means that it is infeasible to modify known ciphertexts to a new valid ciphertext whose decryption is "meaningfully related" [BS99] to the original decryption.

The McEliece cryptosystem is clearly malleable without additional protection, i.e. an attacker randomly flipping bits in a ciphertext is able to create a meaningfully related ciphertext. If he is additionally able to observe the reaction of the receiver – suggesting the name *reaction attack* for this method, in accordance with Niebuhr [Nie12] – he may also be able to reveal the original message. In the case of the Niederreiter cryptosystem, flipping bits in the ciphertext will presumably result in decoding errors. A reaction attack is still possible by adding columns of the parity check matrix to the syndrome. This can also be avoided using CCA2-secure conversions. Furthermore, they defend against broadcast attacks which were also analyzed by Niebuhr *et al.* [Nie12, NC11].

Hence, a CCA2-secure conversion is strictly required in all cases. Unfortunately, the well-known Optimal Asymmetric Encryption Padding (OAEP) scheme [Sho01] cannot be applied because it is "unsuitable for the McEliece/Niederreiter cryptosystems" [NC11], since it does not protect against the reaction attack. Instead, we implemented the Kobara-Imai-γ conversion and Fujisaki-Okamoto conversion, which are discussed in Section 4.8.

2.7 Key length

The main caveat against code-based cryptosystems is the huge key length compared to other public-key cryptosystems. This is particulary troubling in the field of embedded devices, which have low memory ressources but are an essential target platform that needs to be considered to raise the acceptance of McEliece as a real alternative.

Accordingly, much effort has been made to reduce the key length by replacing the underlying code with codes having a compact description. Unfortunately, most proposals have been broken by structural attacks. However, some interesting candidates remain.

For instance, in 2009 Barreto and Misoczki [MB09] proposed a variant based on Quasi-Dyadic Goppa codes, which has not been broken to date. It was implemented on an 8-bit AVR microcontroller by Paustjan [Pau10] and resulted in a public key size reduced by a factor t while still maintaining a higher performance than comparable RSA implementations. However, it is still unknown whether Quasi-Dyadic Goppa codes achieve the same level of security as general Goppa codes.

Using small nonbinary subfield Goppa codes with list decoding as proposed by Bernstein *et al.* [BLP11] also allows a reduction of the key size, thanks to an improved error-correction capability. The original McEliece proposal is included in this approach as the special case $p = 2$. Since the original McEliece resisted all critical attacks so far, the authors suggest that their approach may share the same security properties.

In this thesis, only general Goppa codes are considered, hence no measure to obtain compact keys has been taken apart from the usage of systematic key matrices. The resulting key size are shown in Chapter 5.

3 Coding theory

This chapter elaborates on the concepts of coding theory roughly introduced in the previous chapter. Section 3.1 gives a short overview on basic concepts that are assumed to be known to the reader. The first part of Section 3.2 provides formal definitions for the most important aspects of Coding theory, more precisely for the class of linear block codes, which is used for code-based cryptography. The second part presents several types of linear block codes and shows their relation hierarchy. Section 3.4 deals with the Patterson and Berlekamp-Massey algorithm, which allow efficient decoding of error-correcting codes. For the root extraction step required in both decoding algorithms, several methods are presented in Section 3.5.

3.1 Preliminaries

In this section, we briefly reiterate important concepts that form the basis of Coding theory. Unless otherwise noted, it is based on on several papers by Heyse (e.g. [Hey09]) and a thesis written by Hoffmann [Hof11].

Finite fields Most linear codes rely on the algebraic structure of finite fields – also named Galois Fields – to form the code alphabet.

A finite field is a set of a finite number of elements for which an abelian addition and abelian multiplication operation is defined and distributivity is satisfied. This requires that the operations satisfy closure, associativity and commutativity and must have an identity element and an inverse element.

A finite field with $q = p^m$ elements is denoted $\mathbb{F}_{p^m}$ or $\mathrm{GF}(p^m)$ or $\mathbb{F}_q$, where p is a prime number called the *characteristic* of the field and $m \in \mathcal{N}$. The number of elements is called *order*. Fields of the same order are isomorphic. $\mathbb{F}_{p^m}$ is called an *extension field* of $\mathbb{F}_p$ and $\mathbb{F}_p$ is a *subfield* of $\mathbb{F}_{p^m}$. α is called a *generator* or *primitive element* of a finite field if every element of the field $\mathbb{F}^*_{p^m} = \mathbb{F}_{p^m} \backslash \{0\}$ can be represented as a power of α. Algorithms for solving algebraic equations over finite fields exist, for example polynomial division using the Extended Euclidean Algorithm (EEA) and several algorithms for finding the roots of a polynomial. More details on finite fields can be found in [HP03].

Polynomials over Finite fields Here we present some definitions and algorithms concerning polynomials with coefficients in $\mathbb{F}$.

Definition 3.1.1 (Polynomials over finite fields) *A* polynomial *f with coefficients $c_i \in \mathbb{F}_q$ is an expression of the form $f(z) = \sum_{i=0}^{n} c_i z^i$ and is called a* polynomial in

$\mathbb{F}_{p^m}[z]$, *sometimes shortened to* polynomial in $\mathbb{F}$. *The* degree $\deg(f) = d$ *of* f *is the largest* $i < n$ *such that* p_i *is not zero. If the* leading coefficient $lc(f)$ *is 1, the polynomial is called* monic.

Definition 3.1.2 (Subspace of polynomials over $\mathbb{F}_{p^m}$) *For* $n \in \mathbb{N}$ *and* $1 \leq k \leq n$*, we denote the subspace of all polynomials over* $\mathbb{F}_{p^m}$ *of degree strict less than* k *by* $\mathbb{F}_{p^m}[z]_{<k}$.

Definition 3.1.3 (Irreducible and primitive polynomials) *A non-constant polynomial in* $\mathbb{F}$ *is said to be* irreducible *over* $\mathbb{F}_q$ *if it cannot be represented as a product of two or more non-constant polynomials in* $\mathbb{F}$*. An irreducible polynomial in* $\mathbb{F}$ *having a primitive element as a root is called* primitive*. Irreducible or often preferably primitive polynomials are used for the construction of finite fields.*

Polynomials over finite fields can be manipulated according to the well-known rules for transforming algebraic expressions such as associativity, commutativity, distributivity. Apart from the trivial addition and multiplication, also a polynomial division can be defined, which is required for the EEA that is used by the decoding algorithms shown in Section 3.4.

Definition 3.1.4 (Polynomial division) *If* f, g *are polynomials in* $\mathbb{F}$ *and* $\deg(g) \neq 0$*, then there exist unique polynomials* q *and* r *in* $\mathbb{F}$ *such that* $f = qg + r$ *with* $\deg(r) = 0$ *or* $\deg(r) < \deg(g)$*.* q *is called* quotient polynomial *and* r remainder polynomial*, which is also written* $f \mod g \equiv r$.

Definition 3.1.5 (GCD) *The* greatest common divisor $\gcd(f, g)$ *of two polynomials* f, g *in* $\mathbb{F}$ *is the polynomial of highest possible degree that evenly divides* f *and* g.

The GCD can be efficiently computed recursively using the *Euclidean algorithm*, which relies on the relation $\gcd(f, q) = \gcd(f, f + rg)$ for any polynomial r. The *Extended* Euclidean Algorithm shown in Alg. 3.1.1 additionally finds polynomials x, y in $\mathbb{F}$ that satisfy Bézout's identity

$$\gcd(a(z), b(z)) = a(z)x(z) + b(z)y(z). \tag{3.1}$$

Analogous to the usage of EEA for the calculation of a multiplicative inverse in a finite field, EEA can be used to calculate the inverse of a polynomial $a(z) \bmod b(z)$ in a field $\mathbb{F}$. Then, $x(z)$ is the inverse of $a(z) \bmod b(z)$, i.e. $a(z)x(z) \bmod b(z) \equiv const$.

3.2 Linear block codes

Algebraic coding theory has become a broad field of research and cannot be covered extensively in this thesis. Detailed accounts are given for example in [HP03, Ber72, MS78, Hof11], which are also the source for most definitions given in this section.

Algorithm 3.1.1: Extended Euclidean Algorithm (EEA)

Data: Polynomials $a(x), b(x) \in \mathbb{F}[z], \deg(a) >= \deg(b)$
Result: Polynomials $x(z), y(z)$ with $\gcd(a, b) = ax + by$

```
u(z) ← 0, u1(z) ← 1
v(z) ← 1, v1(z) ← 0
while deg a > 0 do
    (quotient, remainder) ← a(z)/b(z)
    a ← b, b ← remainder
    u2 ← u1, u1 ← u, u ← u2 − quotient · u
    v2 ← v1, v1 ← v, v ← v2 − quotient · v
return x ← u1(z), y ← v1(z)
```

3.2.1 Basic definitions

Linear block codes Linear block codes are a type of error-correcting codes that work on fixed-size data blocks to which they add some redundancy. The redundancy allows the decoder to detect errors in a recieved block and correct them by selecting the 'nearest' codeword. There exist bounds and theorems that help in finding 'good' codes in the sense of minimizing the overhead and maximizing the number of correctable errors.

Definition 3.2.1 (Linear block code) *A linear block code $\mathcal{C}$ is an injective mapping $\Sigma^k \to \Sigma^n$ of a string of length k over some alphabet Σ to a* codeword *of length n. Recall from Section 2.1 that $\mathcal{C}$ is called a $[n, k, d]$ code if the minimal code word distance*[App. B.2.2] *is $d_{min} = min_{x,y \in \mathcal{C}} dist(x, y)$, where* dist *denotes a distance function, for example Hamming distance*[App. B.2.1]*. The distance of x to the null-vector $wt(x) := dist(0, x)$ is called weight of x.*

Definition 3.2.2 (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*.*

Generator matrix and parity check matrix 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 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 $G \cdot H^{\mathrm{T}} = 0$.

Definition 3.2.3 (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 3.2.4 (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.*

Syndrome decoding 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 3.2.5 (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 $\mathcal{S}_{\hat{c}} = H \cdot \hat{c}^{\mathrm{T}} = H \cdot (c^{\mathrm{T}} + e^{\mathrm{T}}) = H \cdot e^{\mathrm{T}}$.*

As long as less errors than the error-correction capability t of the code occured, a syndrome uniquely identifies all errors, thus allowing the decoding of the erroneous codeword by the means of an syndrome decoding algorithm. The syndrome equation is satisfied by all 2^k possible error patterns e, however patterns with a low number of errors are usually more likely. Hence, in practice a *maximum-likelihood-principle* is applied, so the decoder tries to find the codeword with the minimum hamming distance to the recieved vector $\hat{c}$. Syndrome decoding uses the linearity of the code to allow a minimum distance decoding with a reduced lookup table for the nearest match. Section 3.4 presents two decoding algorithms which have been implemented for this thesis.

3.2.2 Important code classes

By now, many classes of linear block codes have been described. Some are generalizations of previously described classes, others are specialized in order to fit specific applications or to allow a more efficient decoding. Fig. 3.1 gives an overview of the hierarchy of code classes.

Polynomial codes Codes that use a fixed and often irreducible *generator polynomial* for the construction of the codeword are called *polynomial codes*. Valid codewords are all polynomials that are divisible by the generator polynomial. Polynomial division of a recieved message by the generator polynomial results in a non-zero remainder exactly if the message is erroneous.

Cyclic codes A code is called *cyclic* if for every codeword cyclic shifts of components result in a codeword again. Every cyclic code is a polynomial code.

Generalized Reed-Solomon 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

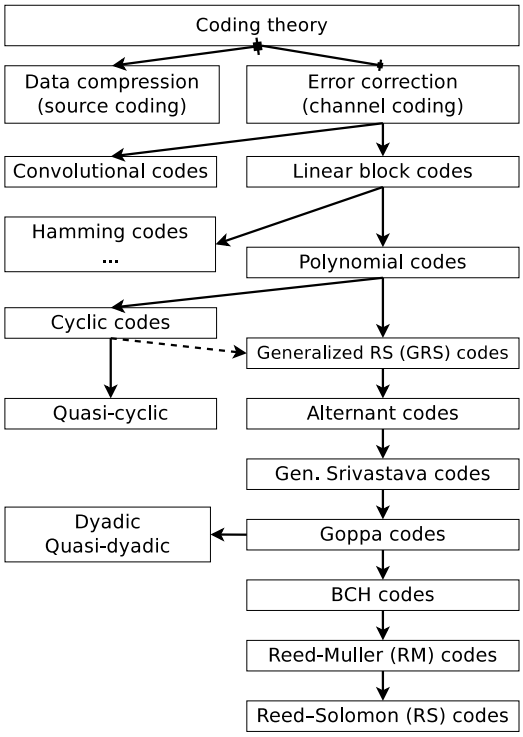

Figure 3.1: Hierarchy of code classes

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, which is $d_{min} = n - k + 1$.

For some polynomial $f(z) \in \mathbb{F}_{p^m}[z]_{<k}$, 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

$$GRS_{n,k}(\mathcal{L}, V) := \{\ c \in \mathbb{F}_{p^m}^n | c_i = v_i f(\alpha_i)\} \tag{3.2}$$

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 $\alpha_i \in \mathbb{F}_{p^m}, 0 \leq i < n$ and non-zero $v_i \in \mathbb{F}_{p^m}, 0 \leq j < t$, the elements of the parity check matrix are defined as $H_{i,j} = \alpha_j^i v_i$.

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

$$Alt_{n,k,p}(\mathcal{L}, v) := GRS_{n,k}(\mathcal{L}, V) \cap \mathbb{F}_p^n \tag{3.3}$$

Generalized Srivastava codes *Generalized Srivastava (GS) codes* [Per12] are alternant codes that use a further refined alternant form for the parity check matrix H. For $s, t \in \mathbb{N}$, let $\alpha_i \in \mathbb{F}_{p^m}, 0 \leq i < n$ and $w_i \in \mathbb{F}_{p^m}, 0 \leq i < s$ be $n + s$ pairwise distinct elements and let $v_i \in \mathbb{F}_{p^m}, 0 \leq j < t$ be non-zero. A GS code of length n over $\mathcal{F}_{p^m}$ with

order $s \cdot t$ is defined by $H = (H_1, H_2, \ldots, H_s)$, where H_i are matrices with components $h_{j,k} = v_k/(\alpha_k - w_i)^j$. GS codes include Goppa codes as a special case.

Goppa codes *Goppa codes* are alternant codes over $\mathbb{F}_{p^m}$ that are restricted to a *Goppa polynomial* $g(z)$ 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 $f(z)$. Hence, a definition of Goppa codes can be derived from the definition of GRS codes as follows:

$$Goppa_{n,k,p}(\mathcal{L}, g) := GRS_{n,k}(\mathcal{L}, g) \cap \mathbb{F}_p^n \tag{3.4}$$

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 $d_{min} \geq 2t + 1$. Details for constructing and decoding Goppa codes are given in Section 3.3.

BCH codes, RM codes, RS codes There exist several important special cases of Goppa codes (which have been first described in 1969), most prominently *BCH codes* (1959), *Reed-Muller codes* (1954) and *Reed-Solomon codes* (1960). For example, primitive BCH codes are just Goppa codes with $g(z) = z^{2t}$ [Ber73].

3.3 Construction of Goppa codes

Goppa codes [Gop69, Ber73] 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 details and use Goppa codes – more specifically, binary Goppa codes using an irreducible Goppa polynomial – to introduce the decoding algorithms developed by Patterson and Berlekamp.

3.3.1 Binary Goppa codes

We begin by reiterating the above definition (Section 3.2.2) of Goppa for the case of binary Goppa codes, giving an explicit definition of the main ingredients.

Definition 3.3.1 *Let m and t be positive integers and let the* Goppa polynomial

$$g(z) = \sum_{i=0}^{t} g_i z^i \in \mathbb{F}_{2^m}[z] \tag{3.5}$$

be a monic polynomial of degree t and let the support

$$\mathcal{L} = \{\alpha_0, \ldots, \alpha_{n-1}\} \in \mathbb{F}_{2^m}^n,\ g(\alpha_j) \neq 0 \,\forall\, 0 \leq j \leq n-1 \tag{3.6}$$

be a subset of n distinct elements of $\mathbb{F}_{2^m}$*. For any vector* $\hat{c} = (c_0, \cdots, c_{n-1}) \in \mathbb{F}_{2^m}^n$*, in accordance with Definition 3.2.5 we define the* syndrome *of* $\hat{c}$ *as*

$$\mathcal{S}_{\hat{c}}(z) = -\sum_{i=0}^{n-1} \frac{\hat{c}_i}{g(\alpha_i)} \frac{g(z) - g(\alpha_i)}{z - \alpha_i} \mod g(z). \tag{3.7}$$

In continuation of Eq. 3.4, we now define a binary Goppa code *over* $\mathbb{F}_{2^m}$ *using the syndrome equation.* $c \in \mathbb{F}_{2^m}^n$ *is a codeword of the code exactly if* $\mathcal{S}_c = 0$*:*

$$Goppa_{n,k,2}(\mathcal{L}, g(z)) := \{\ c \in \mathbb{F}_{2^m}^n \mid \mathcal{S}_c(z) = \sum_{i=0}^{n-1} \frac{c_i}{z - \alpha_i} \equiv 0 \mod g(z)\}. \tag{3.8}$$

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

3.3.2 Parity Check Matrix of Goppa Codes

According to the definition of a syndrome in Eq. 3.7, every element $\hat{c}_i$ of a vector $\hat{c} = c + e$ is multiplied with

$$\frac{g(z) - g(\alpha_i)}{g(\alpha_i) \cdot (z - \alpha_i)}. \tag{3.9}$$

Hence, given a Goppa polynomial $g(z) = g_s z^s + g_{s-1} z^{s-1} + \cdots + g_0$, according to Definition 3.2.3 the parity check matrix H can be constructed as

$$H = \begin{pmatrix} \frac{g_s}{g(\alpha_0)} & \frac{g_s}{g(\alpha_1)} & \cdots & \frac{g_s}{g(\alpha_{n-1})} \\ \frac{g_{s-1} + g_s \cdot \alpha_0}{g(\alpha_0)} & \frac{g_{s-1} + g_s \cdot \alpha_0}{g(\alpha_1)} & \cdots & \frac{g_{s-1} + g_s \cdot \alpha_0}{g(\alpha_{n-1})} \\ \vdots & \ddots & & \vdots \\ \frac{g_1 + g_2 \cdot \alpha_0 + \cdots + g_s \cdot \alpha_0^{s-1}}{g(\alpha_0)} & \frac{g_1 + g_2 \cdot \alpha_0 + \cdots + g_s \cdot \alpha_0^{s-1}}{g(\alpha_1)} & \cdots & \frac{g_1 + g_2 \cdot \alpha_0 + \cdots + g_s \cdot \alpha_0^{s-1}}{g(\alpha_{n-1})} \end{pmatrix} \tag{3.10}$$

This can be simplified to

$$H = \begin{pmatrix} g_s & 0 & \cdots & 0 \\ g_{s-1} & g_s & \cdots & 0 \\ \vdots & \ddots & & \vdots \\ g_1 & g_2 & \cdots & g_s \end{pmatrix} \times \begin{pmatrix} \frac{1}{g(\alpha_0)} & \frac{1}{g(\alpha_1)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\ \frac{\alpha_0}{g(\alpha_0)} & \frac{\alpha_1}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}}{g(\alpha_{n-1})} \\ \vdots & \ddots & & \vdots \\ \frac{\alpha_0^{s-1}}{g(\alpha_0)} & \frac{\alpha_1^{s-1}}{g(\alpha_1)} & \cdots & \frac{\alpha_{n-1}^{s-1}}{g(\alpha_{n-1})} \end{pmatrix} = H_g \times \hat{H} \tag{3.11}$$

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 3.2.4, the generator matrix G can be derived from the systematic parity check matrix $H = (Q|I_{n-k})$ as $(I_k| - Q^{\mathrm{T}})$.

3.4 Decoding algorithms

Many different algorithms for decoding linear codes are available. The Berlekamp-Massey (BM) algorithm is one of the most popular algorithms for decoding. It was invented by Berlekamp [Ber68] for decoding BCH codes and expanded to the problem of finding the shortest linear feedback shift register (LFSR) for an output sequence by Massey [Mas69], but later found to actually be able to decode *any* alternant code [Lee07]. The same applies to the Peterson decoder [Pet60] or Peterson-Gorenstein-Zierler algorithm [GPZ60] and the more recent list decoding [Sud00].

However, there are also specialized algorithms that decode only certain classes of codes, but are able to do so more efficiently. An important example is the Patterson Algorithm [Pat75] for binary Goppa codes, but there are also several specialized variants of general decoding algorithms for specific code classes, such as list decoding for binary Goppa codes [Ber11].

This thesis concentrates on Goppa codes, hence we will present the two most important algorithms that are currently available for decoding Goppa codes: Patterson and Berlekamp-Massey. The description of the Patterson algorithm is in major parts based on several papers by Heyse (e.g. [Hey09]) and on lecture notes by Huber [Hub], while the section on BM is mainly based on [HG13, Ber72, BAO09].

3.4.1 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 (ELP)* $\sigma(z)$, 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 $\mathit{Goppa}(\mathcal{L}, g(z))$, where the positions of these elements inside of $\mathcal{L}$ correspond to the error positions x_i in $\hat{c}$. The error locator polynomial is defined as:

$$\sigma(z) = \prod_{i \in E}(z - \gamma_i) = \prod_{i \in E}(1 - x_i z). \tag{3.12}$$

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 value polynomial (EVP)* $\omega(z)$ is required for the determination of the error values. Let y_i denote the error value of the i-th error. Then, the error value polynomial is defined as:

$$\omega(z) = \sum_{i \in E} y_i x_i \prod_{j \neq i \in E}(1 - x_j z). \tag{3.13}$$

Note that it can be shown that $\omega(z) = \sigma'(z)$ is the formal derivative of the error locator polynomial.

Since the Patterson algorithm is designed only for *binary* Goppa codes, $\omega(z)$ does not occur there explicitly. Nevertheless, both algorithms implicitly or explicitly solve the following *key equation*

$$\omega(z) \equiv \sigma(z) \cdot \mathcal{S}(z) \mod g(z). \tag{3.14}$$

3.4.2 Syndrome computation

The input to the decoder is a syndrome $\mathcal{S}_{\hat{c}}(z)$ for some vector $\hat{c} = c + e$, where c is a codeword representing a message m and e is an error vector. By definition, $\mathcal{S}_{\hat{c}}(z) = \mathcal{S}_e(z)$ since $\mathcal{S}_c(z) = 0$. Generally it can be computed as $\mathcal{S}_{\hat{c}}(z) = H \cdot \hat{c}^{\mathrm{T}}$ according to Definition 3.2.5. If $\mathcal{S}(z) = 0$, the codeword is free of errors, resulting in an error locator polynomial $\sigma(z) = 0$ and an error vector $e = 0$.

To avoid the multiplication with H, alternative methods of computing the syndrome can be used. For binary Goppa codes, the following syndrome equation can be derived from Eq. 3.7:

$$\mathcal{S}(z) \equiv \sum_{\alpha \in \mathbb{F}_{2^m}} \frac{\hat{c}_\alpha}{z - \alpha_i} \mod g(z) \equiv \sum_{\alpha \in \mathbb{F}_{2^m}} \frac{e_\alpha}{z - \alpha_i} \mod g(z) \tag{3.15}$$

Both methods and a third, more efficient variant were implemented and are described in Section 4.6.1.

3.4.3 Berlekamp-Massey-Sugiyama

The Berlekamp-Massey algorithm was proposed by Berlekamp in 1968 and works on general alternant codes. The application to LFSRs performed by Massey is of less importance to this thesis. Compared to the Patterson algorithm, BM can be described and implemented in a very compact form using EEA. Using this representation, it is equivalent to the Sugiyama algorithm [SKHN75].

3.4.3.1 General usage

BM returns an error locator polynomial $\sigma(z)$ and error value polynomial $\omega(z)$ satisfying the key equation Eq. 3.14. Applied to binary codes, $\sigma(z)$ does not need to be taken into account.

Algorithm 3.4.1: Berlekamp-Massey-Sugiyama algorithm

System parameters: Alternant code with generator polynomial $g(z)$
Input: Syndrome $s = \mathcal{S}_{\hat{c}}(z)$
Output: Error locator polynomial $\sigma(z)$

```
1 if s ≡ 0 mod g(z) then
2     return σ(z) = 0
3 end
4 else
5     (σ(z), ω(z)) ← EEA(S(z), G(z))
6     return (σ(z), ω(z))
7 end
```

Preliminaries Alg. 3.4.1 shows the Berlekamp-Massey-Sugiyama algorithm for decoding the syndrome 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 generator polynomial $g(z)$, which may be a – possibly reducible – Goppa polynomial $g(z)$ of degree t. In Berlekamp's original proposal for BCH codes $g(z)$ is set to $g(z) = z^{2t+1}$. In the *general* case, the BM algorithm ensures the correction of all errors only if a maximum of $\frac{t}{2}$ errors occured, 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 $g(z)^2$ instead of $g(z)$ and thus working on a syndrome of double size.

Decoding general alterant codes The input to BM is a syndrome polynomial, which can be computed as described in Section 3.4.2. In the general case, Berlekamp defines the syndrome as $\mathcal{S}(z) = \sum_{i=1}^{\infty} S_i \, z^m$, where only $S_1, \ldots, S_t$ are known to the decoder. Then, he constructs a relation between $\sigma(z)$ (Eq. 3.12) and $\omega(z)$ (Eq. 3.13) and the known S_i by dividing $\omega(z)$ by $\sigma(z)$.

$$\frac{\omega(z)}{\sigma(z)} = 1 + \sum_j \frac{y_j \, x_j \, z}{1 - x_j z} = 1 + \sum_{i=1}^{\infty} S_i \, z^m \tag{3.16}$$

where x_i are the error positions and y_i the error values known from Section 3.4.1.

Thus, he obtains the key equation

$$(1 + \mathcal{S}(z)) \cdot \sigma \equiv \omega \mod z^{2t+1} \tag{3.17}$$

already known from Section 3.4.1.

For solving the key equation, Berlekamp proposes "a sequence of successive approximations, $\omega^{(0)}, \sigma^{(0)}, \omega^{(1)}, \sigma^{(1)}, \ldots, \sigma^{(2t)}, \omega^{(2t)}$, each pair of which solves an equation of the form $(1 + \mathcal{S}(z))\sigma^{(k)} \equiv \omega^{(k)} \mod z^{k+1}$" [Ber72].

The algorithm that Berlekamp gives for solving these equations was found to be very similar to the Extended Euclidean Algorithm (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" [Dor87] and hence considers them to be equivalent. Accordingly, BM is also very similar to the Sugiyama Algorithm [SKHN75], 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 better perfomance than the Sugiyama algorithm" [BAO09]. On the contrary, the authors of [HP03] state that Sugiyama "is quite comparable in efficiency".

For this thesis, we decided to implement and describe BM using EEA in order to keep the program code size small. Then, the key equation can be solved by applying EEA to $\mathcal{S}(z), G(z))$, which returns σ and ω as coefficients of Bézouts identity given in Eq. 3.1. The error positions x_i can be determined by finding the roots of σ, as shown in Section 3.5. For non-binary codes, also ω needs to be evaluated to determine the error values. This can be done using a formula due to Forney [For65], which computes the error values as

$$e_i = -\frac{\omega(x_i^{-1})}{\sigma'(x_i^{-1})} \tag{3.18}$$

where σ' is the formal derivative of the error locator polynomial.

3.4.3.2 Decoding binary Goppa Codes

BM and t-error correction The Patterson algorithm is able to correct t errors for Goppa codes with a Goppa polynomial of degree t, because the minimum distance of a separable binary Goppa code is at least $d_{min} = 2t + 1$. This motivates the search for a way to achieve the same error-correction capability using the Berlekamp-Massey algorithm, which by default does not take advantage of the property of binary Goppa codes allowing t-error correction.

Using the well-known equivalence [MS78]

$$Goppa(\mathcal{L}, g(z)) \equiv Goppa(\mathcal{L}, g(z)^2) \tag{3.19}$$

which is true for any square-free polynomial $g(z)$, we can construct a syndrome polynomial of degree $2t$ based on a parity check matrix of double size for $Goppa(\mathcal{L}, g(z)^2)$. Recall that the Berlekamp-Massey algorithm sets up a set of syndrome equations, of which only $S_1, \ldots, S_t$ are known to the decoder. Using BM modulo $g(z)^2$ produces $2t$ known syndrome equations, which allows the algorithm to use all inherent information provided by $g(z)$. This allows the Berlekamp-Massey algorithm to correct t errors and is essentially equivalent to the splitting of the error locator polynomial into odd and even parts in the Patterson algorithm, which yields a 'new' key equation as well.

Application to binary Niederreiter A remaining problem is the decoding of t errors using BM and Niederreiter 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. Completely switching to a code over $g(z)^2$ – also for the encryption process – would double the code size without need, since we know that the Patterson algorithm is able to correct all errors using the standard code size over $g(z)$.

Instead we can use an approach described by Heyse in [HG13]. Remember that a syndrome s of length $n - k$ corresponding to an erroneous codeword $\hat{c}$ satisfies the equation $s = \mathcal{S}_{\hat{c}} = eH^{\mathrm{T}}$, where e is the error vector that we want to obtain by decoding s. Now let s be a syndrome of standard size computed modolu $g(z)$. By prepending s with k zeros, we obtain $(0|s)$ of length n. Then, using Eq. 3.19 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 $g(z)$ and $g(z)^2$, 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.

3.4.4 Patterson

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 [EOS06], whereas general decoding algorithms such as BM can only correct $\frac{t}{2}$ errors by default.

Algorithm 3.4.2: PATTERSON ALGORITHM FOR DECODING BINARY GOPPA CODES

System parameters: Goppa code with an irreducible Goppa polynomial $g(z)$
Input: Syndrome $s = \mathcal{S}_{\hat{c}}(z)$
Output: Error locator polynomial $\sigma(z)$

1 **if** $s \equiv 0 \mod g(z)$ **then**
2 **return** $\sigma(z) = 0$
3 **end**
4 **else**
5 $T(z) \leftarrow s^{-1} \mod g(z)$
6 **if** $T(z) = z$ **then**
7 $\sigma(z) \leftarrow z$
8 **end**
9 **else**
10 $R(z) \leftarrow \sqrt{T(z) + z}$ `// Huber`
11 $(a(z), b(z)) \leftarrow \text{EEA}(R(z), G(z))$ // $a(z) \equiv b(z) \cdot R(z) \mod G(z)$
12 $\sigma(z) \leftarrow a(z)^2 + z \cdot b(z)^2$
13 **end**
14 **return** $\sigma(z)$
15 **end**

Preliminaries Alg. 3.4.2 summarizes Patterson's algorithm for decoding the syndrome of a vector $\hat{c} = c + e \in \mathbb{F}_{2^m}^n$ using a binary Goppa code with an irreducible Goppa polynomial $g(z)$ of degree t. 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. The error vector e has been added to c either intentionally like in code-based cryptography, or unintendedly, 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 occured, i.e. if e has a weight $\text{wt}(e) \leq t$.

Solving the key equation The Patterson algorithm does not directly solve the key equation. Instead, it transforms Eq. 3.14 to a simpler equation using the property $\omega(z) = \sigma'(z)$ and the fact that $y_i = 1$ at all error positions.

$$\omega(z) \equiv \sigma(z) \cdot \mathcal{S}(z) \equiv \sum_{i \in E} x_i \prod_{j \neq i \in E} (1 - z) \mod g(z) \tag{3.20}$$

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

$$\sigma(z) = a(z)^2 + zb(z)^2 \tag{3.21}$$

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

$$\begin{aligned} \sigma'(z) &= b(z)^2 = \omega(z) && (3.22)\\ &\equiv \sigma(z) \cdot \mathcal{S}(z) \mod g(z) && (3.23)\\ &\equiv (a(z)^2 + zb(z)^2) \cdot \mathcal{S}(z) \equiv b(z)^2 \mod g(z) && (3.24) \end{aligned}$$

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

$$(T(z) + z) \cdot b(z)^2 \equiv a(z)^2 \mod g(z). \tag{3.25}$$

If $T(z) = z$, we obtain the trivial solutions $a(z) = 0$ and $b(z)^2 = zb(z)^2 \cdot \mathcal{S}(z) \mod g(z)$, yielding $\sigma(z) = z$. Otherwise we use an observation by [Hub96] for polynomials in $\mathbb{F}_{2^m}$ giving a simple expression for the polynomial $r(z)$ which solves $r(z)^2 \equiv t(x) \mod g(z)$. To satisfy Hubers equation, we set $R(z)^2 \equiv T(z) + z \mod g(z)$ and obtain $R(z) \equiv \sqrt{T(z) + z}$. Finally, $a(z)$ and $b(z)$ satisfying

$$a(z) \equiv b(z) \cdot R(z) \mod G(z) \tag{3.26}$$

can be computed using EEA and applied to Eq. 3.21. As $\deg(\sigma(z)) \leq g(z) = t$, the equation implies that $\deg(a(z)) \leq \lfloor \frac{t}{2} \rfloor$ and $\deg(b(z)) \leq \lfloor \frac{t-1}{2} \rfloor$ [Hey08, OS09]. Observing the iterations of EEA (Alg. 3.1.1) one finds that the degree of $a(z)$ is constantly decreasing from $a_0 = g(z)$ while the degree of $b(z)$ increases starting 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 $a(z)$ drops below $\frac{t}{2}$.

Time complexity Overbeck provides a runtime complexity estimation in [OS09]. Given a Goppa polynomial $g(z)$ of degree t and coefficients of size m, EEA takes $\mathcal{O}(t^2m^2)$ binary operations. It is used for the computation of $T(z)$ as well as for solving the key equation. $R(z)$ is computed as a linear mapping on $\mathcal{F}_{2^m}[z]/g(z)$, which takes $\mathcal{O}(t^2m^2)$ binary operations, too. Hence, the runtime of Patterson is quadratic in t and m. Note that decoding is fast compared to the subsequent root extraction.

3.5 Extracting roots of the error locator polynomial

The computation of the roots of the ELP belongs to the computationally most expensive steps of McEliece and Niederreiter. In this section, we present several methods of root extraction. For brevity, we consider only the case of t-error correcting Goppa codes with a permuted, secret support $\mathcal{L} = (\alpha_0, \ldots, \alpha_{n-1})$, but the algorithms can be easily applied to other codes.

As stated already in Section 3.4.1, the roots of $\sigma(z) = \sum_{i=0}^{t} \sigma_i z^i$ are elements of the support $\mathcal{L}$, where the position of the roots *inside* of $\mathcal{L}$ correspond to the error positions in $\hat{c}$. Let $L(i)$ denote the field element at position i in the support and $L^{-1}(i)$ the position of the element i in the support. Then, for all $0 \leq i < n$ the error vector $e = (e_0, \ldots, e_{n-1})$ is defined as

$$e_i = \begin{cases} 1 & \sigma(L(i)) \equiv 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.27}$$

3.5.1 Brute force search using the Horner scheme

The most obvious way of finding the roots is by evaluating the polynomial for all support elements, i.e. testing $\sigma(\alpha_i) = 0$ for all $\alpha_i \in \mathcal{L}$. This method, shown in Alg. 3.5.1, is not sophisticated, but can be implemented easily and may be even faster than others as long as the field size os low enough. The search can be stopped as soon as t errors have been found. Note, however, that this introduces a potential side channel vulnerability, since it makes the otherwise constant runtime dependent on the position of the roots of $\sigma(z)$ in the secret support. Since each step is independent from all others, it can be easily parallelized.

In the worst case, all n elements need to be evaluated and $\sigma(z)$ has the full degree t. Representing $\sigma(z)$ as $\sigma_0 + z(\sigma_1 + z(\sigma_2 + \cdots + z(\sigma_{t-1} + z\sigma_t)\ldots))$, the well-known Horner scheme [Rhe80] can be used for each independently performed polynomial evaluation, hence resulting in $n \times t$ additions and $n \times t$ multiplications in the underlying field.

Algorithm 3.5.1: Search for roots of $\sigma(z)$ using Horner's scheme

Data: Error locator polynomial $\sigma(z)$, support $\mathcal{L}$
Result: Error vector e

```
1  e ← 0
2  for i = 0 to n − 1 do
3      x ← L(i)
4      s ← σ_t
5      for j ← t to 0 do
6          s ← σ_j + s · x
7      end
8      if s = 0 then e_i = 1
9  end
10 return e
```

Note that it is possible to speed up the search by performing a polynomial division of $\sigma(z)$ by $(z - \mathcal{L}_i)$ as soon as $\mathcal{L}_i$ was found to be a root of $\sigma(z)$, thus lowering the degree of $\sigma(z)$ and hence the runtime of the polynomial evaluation. The polynomial division can be performed very efficiently by first bringing $\sigma(z)$ to monic form, which does not alter its roots. However, the use of the polynomial division introduces another potential

timing side channel vulnerability, similar to the stop of the algorithm after t errors have been found.

3.5.2 Bruteforce search using Chien search

A popular alternative is the Chien search [Chi06], which employs the following relation valid for any polynomial in $\mathcal{F}_{p^m}$ where α is a generator of $\mathcal{F}^*_{p^m}$:

$$\begin{aligned} \sigma(\alpha^i) &= \sigma_0 &&+ \sigma_1\alpha^i &&+ \ldots &&+ \sigma_t\,(\alpha^i)^t \\ \sigma(\alpha^{i+1}) &= \sigma_0 &&+ \sigma_1\alpha^{i+1} &&+ \ldots &&+ \sigma_t\,(\alpha^{i+1})^t \\ &= \sigma_0 &&+ \sigma_1\alpha^i\alpha &&+ \ldots &&+ \sigma_t\,(\alpha^i)^t\alpha^t \end{aligned}$$

Let $a_{i,j}$ denote $(\alpha^i)^j \cdot \sigma_j$. From the above equations we obtain $a_{i+1,j} = a_{i,j} \cdot \alpha^j$ and thus $\sigma(\alpha^i) = \sum_{j=0}^{t} a_{i,j} = a_{i,0} + a_{i,1} + \cdots + a_{i,t} = \sigma_0 + \sigma_1 \cdot \alpha^i + \cdots + \sigma_t \cdot (\alpha^i)^t$. Hence, if $\sum_{j=0}^{t} a_{i,j} = 0$, then α^i is a root of $\sigma(z)$, which determines an error at position $L^{-1}(\alpha^i)$. Note that the zero element needs special handling, since it cannot be represented as an α^i; this is *not* considered in Alg. 3.5.2.

Chien search can be used to perform a bruteforce search over all support elements, similar to the previous algorithm using Horner scheme. However, the search has to be performed in order of the support, since results of previous step are used.

For small m and some fixed t, this process can be efficiently implemented in hardware, since it reduces all multiplications to the multiplication of a precomputed constant $\alpha^j \,\forall\, 1 \leq j \leq t$ with one variable. Moreover, all multiplications of one step can be executed in parallel.

However, this is of little or no advantage for a software implementation. In the worst case, Chien search requires $(p^m - 1) \times t$ multiplications and additions, which is identical or even worse than the bruteforce approach using Horner.

As before, the search can be stopped as soon as t errors have been found, at the price of introducing a potential side channel vulnerability.

3.5.3 Berlekamp-Trace algorithm and Zinoviev procedures

The Berlekamp-Trace algorithm (BTA) [Ber71] is a factorization algorithm that can be used for finding the roots of $\sigma(z)$ since there are no multiple roots. Hence, the factorization ultimately returns polynomials of degree 1, thus directly revealing the roots.

BTA works by recursively splitting $\sigma(z)$ into polynomials of lower degree. Biswas and Herbert pointed out in [BH09] that for binary codes, the number of recursive calls of BTA can be reduced by applying a collection of algorithms by Zinoviev [Zin96] for finding the roots of polynomials of degree ≤ 10. This is in fact a tradeoff between runtime, memory and code size, and the optimal degree d_z where the BTA should be stopped and Zinovievs procedures should be used instead must be determined as the case arises. Biswas and Herbert suggest $d_z = 3$ and call the combined algorithm BTZ.

Algorithm 3.5.2: CHIEN SEARCH FOR ROOTS OF $\sigma(z)$

Data: Error locator polynomial $\sigma(z)$, support $\mathcal{L}$
Result: Error vector e

$e \leftarrow 0$
if $\sigma_0 = 0$ **then** $x = L^{-1}(0), e_x \leftarrow 1$
for $i \leftarrow 0$ **to** t **do**
    $p_i \leftarrow \sigma_i$ `// Initialize intermediate values to` $p_i = \sigma_i \cdot (\alpha^i)^0$
end
for $i \leftarrow 1$ **to** $p^m - 1$ **do**
    $s \leftarrow \sigma_0$
    for $j \leftarrow 1$ **to** t **do**
        $p_j \leftarrow p_j \cdot \alpha^j$
        $s \leftarrow s + p_j$
    end
    if $s = 0$ **then** $x = L^{-1}(\alpha^i), e_x \leftarrow 1$
end
return e

Let p be prime, $m \in \mathcal{N}$, $q = p^m$ and $f(z)$ a polynomial of degree t in $\mathcal{F}_q[z]$. BTA makes use of a Trace function, which is defined over $\mathcal{F}_q$ as

$$\mathrm{Tr}(z) = \sum_{i=0}^{m-1} z^{p^i} \tag{3.28}$$

and maps elements of $\mathcal{F}_{p^m}$ to $\mathcal{F}_p$. This can be used to uniquely represent any element $\alpha \in \mathcal{F}_{p^m}$ using a basis $B = (\beta_1, \ldots, \beta_m)$ of $\mathcal{F}_{p^m}$ over $\mathcal{F}_p$ as a tuple $(\mathrm{Tr}(\beta_i \cdot \alpha), \ldots, \mathrm{Tr}(\beta_m \cdot \alpha))$.

Berlekamp prooves that

$$f(z) = \prod_{s \in \mathcal{F}_p} \gcd(f(z), \mathrm{Tr}(\beta_i z) - s) \quad \forall\, 0 \leq j < m \tag{3.29}$$

where $\gcd(\cdot)$ denotes the *monic* common divisor of greatest degree. Moreover, he shows that at least one of these factorizations is non-trivial. Repeating this procedure recursively while iterating on $\beta_i \in B$ until the degree of each factor is 1 allows the extraction of all roots of $f(z)$ in $\mathcal{O}(mt^2)$ operations [BH09]. If BTZ is used, proceed with Zinovievs algorithms as soon as degree d_z is reached, instead of factorizing until degree 1.

Alg. 3.5.3 shows the BTZ algorithm, but omits all details of Zinoviev's algorithms. The first call to the algorithm sets $i = 1$ to select β_1 and $f = \sigma(z)$ and the error vector e to zero. Note that the polynomials $\mathrm{Tr}(\beta_i z) \mod f(z) \,\forall\, 0 \leq i < m$ can be precomputed.

3.6 Message recovery

After the extraction of the error positions – and if necessary, error values – the error vector e can be constructed. The original codeword c can be recovered simply by performing

Algorithm 3.5.3: BTZ ALGORITHM EXTRACTING ROOTS OF $\sigma(z)$

Data: Error vector e, Polynomial $f(z)$, support $\mathcal{L}$, integer i, integer d_z
Result: Error vector e

```
1 if deg(f) = 0 then return e
2 if deg(f) = 1 then
3     x = L^{-1}(-f_0/f_1), e_x ← 1    // f(z) = f_1 · z + f_0 ≡ 0
4     return e
5 end
6 if deg(f) ≤ d_z then return Zinoviev(f, L, e)
7 g ← gcd(f, Tr(β_i · z))
8 h ← f/g
9 return BTZ(e, g, L, i + 1, d_z) ∪ BTZ(e, h, L, i + 1, d_z)
```

the modular addition $c = \hat{c} + e$. For binary codes, e is a binary vector and the addition can be performed using a XOR operation, i.e. flipping all bits in $\hat{c}$ corresponding to the error positions in e.

For the McEliece cryptosystem, the message is restored from the codeword by computing a pseudo-inverse G^{-1} of the generator matrix G and multiplying it with the codeword: $\hat{m} \leftarrow c \cdot G^{-1}$. If a systematic generator matrix has been used, c can be mapped to m simply by removing the appended parity data.

In the Niederreiter cryptosystem, constant weight encoding (see Section 4.7) is used to encode the message into the error vector instead of into the codeword. To restore the message, the reverse operation needs to be applied.

4 Implementation

In this chapter, we present some aspects of the implementation of our Code-based cryptography library. It includes implementations of both the McEliece and Niederreiter cryptosystems with Berlekamp-Massey (BM) and Patterson as possible decoding algorithms, the CCA2-secure conversion by Kobara-Imai and Fujisaki-Okamoto, root searching algorithms by Chien, Berlekamp and Horner as well as several options for algorithmical details such as the manner of syndrome decoding. Parameter sets for 60-, 80-, 128- and 256-bit security are predefined, but the library is not limited to these sets. Unfortunately, to date the library is restricted to binary codes. In this chapter, we will however point out where changes are necessary to deal with q-ary codes.

The implementation is written in plain C code, apart from a small precompilation script written in Python for the sake of simplicity, and some parts in Assembly code mostly belonging to the included implementation of the SHA3 hash function.

Our main target platform is the frequently used AVR ATxmega256A3, which is an 8-bit RISC microcontroller operating at a clock frequency of up to 32 MHz. With 16 kBytes SRAM and 256 kBytes flash memory, it provides a comparably large amount of memory space, but apart from that it is a relatively simple device. This decision was made to prove that Code-based cryptography is actually usable in any context, including applications in the huge field of low-priced embedded devices with limited ressources.

The implementation was developed using the open source IDE *Code::Blocks 10.05* and compiled using the AVR 8-bit GNU Toolchain *avr-gcc 4.6.2* respectively *gcc 4.4.3* for x86 computers. Communication with the device was performed via JTAG using *avrdude 5.1* for programming and via UART using *screen* for all I/O.

This chapter is organized as follows. First the reader is introduces to some peculiarities concerning memory management on the AVR platform in Section 4.1. Section 4.2 describes consequences drawn from the time-memory conflict fueled by the large key sizes in Code-based cryptography and shortly deals with secure storage on microcontrollers. Section 4.3 first introduces the `gf_t` and `poly_t` types and then describes how lookup tables and the `FASTFIELD` switch speed up the basic blocks of our implementation. `matrix_t` is the third important data structure, which is presented in Section 4.4 before discussing the key generation and key management. Encryption is discussed in Section 4.5 and decryption including the Patterson and Berlekamp-Massey decoding algorithms in Section 4.6. Afterwards, constant weight encoding is introduced briefly in Section 4.7. Finally, we present the CCA2-conversions by Kobara-Imai and Fujisaki-Okamoto in Section 4.8.

4.1 Memory management on AVR

The Flash memory of the AVR microcontroller is organized as 16 bits words. Historically, only small amounts of internal Flash memory were available, which could be addressed by 16-bit pointers using the avr-libc functions `pgm_read_byte_near()` and `pgm_read_word_near()`. However, to address memory beyong the 64 kByte boundary, additional address registers and the slower functions `pgm_read_byte_far()` and `pgm_read_word_far()` need to be used. Due to the large key this is neccessary in our implementation. Care must be taken to place frequently used data such as lookup tables and the code support below the boundary.

The avr-gcc compiler adds to this problem by handling pointers as signed 16-bit integers, effectively halving the addressable memory space. Consequently, instead of accessing an array as usual by writing `array[index]`, it needs to be accessed by explicitly calculating the address using the base address known a compile time and an added offset, for example as `pgm_read_byte_near(array+index)`. To access memory from beyond the 64 kByte boundary, an additional macro `FAR()` needs to be used, which builds the 32-bit `uint_farptr_t` pointer.

Therefore all functions working with data from flash need to be rewritten to address the data on their own, instead of working with passed pointers.

The `PROGMEM` directive is normally used to put data in the flash memory and without copying it into the SRAM at startup. However, the treatment of pointers as signed 16-bit values restricts the size of arrays to 32 kByte, which is not enough to hold a complete key matrix for typical parameters. Instead of splitting such a matrix into several parts – introducing an additional overhead – assembler instructions can be used to put data directly in the .text section. Using the fixed base address made known at compile time using `.global VARNAME` and if neccessary the `FAR` macro, it can be used without further changes compared to usual `PROGMEM` memory access.

4.2 Design criteria

While a considerable portion of embedded devices is designed for short life cycles or holds only data of comparably low value and thus does not require long-term security [Hey10], there is an ever-growing number of devices such as smartphones dealing with valuable data that requires protection possibly beyond the lifespan of the device. Since higher security comes with longer key sizes, this consideration directly points us to the main drawback of Code-based cryptography: the huge key size compared to conventional public-key cryptosystems like RSA or ECC.

This is especially true for implementations targeting embedded devices. Consider the case of binary McEliece with 128-bit security ($n = 2960, k = 2288, t = 56$) and key matrices in systematic representation. Then, already the generator matrix requires at least $n \times (n - k)$ Bit $\approx$ 188 kB memory, exhausting nearly $^3/_4$ of available program and data memory. During key generation, even a non-systematic matrix of full size $n \times k$ Bit $\approx$ 846 kB is required, which is impossible without the use of additional external memory.

Hence, a memory-efficient implementation clearly belongs to the most important goals

of this thesis. This prohibits us for example from using time-memory tradeoffs in some places, including an broader use of function inlining. Furthermore, we decided to avoid any call to `alloc` in order to prevent heap fragmentation, which is a common problem on memory-constrained devices. Hence, all parameters need to be known at compile time. Also the choice of algorithms for decoding, root searching and so on is fixed at compile time in order to allow the compiler to ban any unneeded function from wasting program memory[1].

Nevertheless, for Code-based cryptography to be become an accepted alternative to conventional cryptosystems, implementations still need to have an acceptable speed. Since finite field arithmetic is used intensively throughout the entire system, we decided to retain the common practice of speeding up field arithmetic by using lookup tables of field elements, although they require 16 kB for parameters as above.

Another issue that needs to be considered is the secure storage of the secret key. For this purpose, we rely on the lock-bit feature provided by AVR microcontrollers. Once the lock bit for a code region is set to deny all read access, it can only be unset by a complete chip erase, removing all data from flash memory. Note that it might still be possible to extract key data using side channel attacks or sophisticated invasive attacks, given enough time and ressources.

The implementation requires a random source at several occasions, for instance in the key generation algorithm, but also during encryption. Since the AVR has no true random number source, we use the the AVR libc `rand()` function and seed it using the least significant bits of an unconnected A/D converter. As an alternative (or additional) random source, uninitialied SRAM values could be used. However, this ensures no cryptographically secure random. Realword applications should therefore use a more sophisticated approach, for example involving the AVR crypto modules to construct a cryptographically secure PRNG.

4.3 Fast finite field arithmetics

Let α be a generator of the finite field $\mathcal{F}_{p^m}$ over some irreducible or even primitive polynomial $p(z)$ in $\mathcal{F}_p$ of degree m. Any field element can be represented in an polynomial representation as well as an exponential represention, except the zero element which has no natural exponential representation.

4.3.1 Field element representations

Polynomial representation In polynomial representation, an element $a \in \mathcal{F}_{p^m}$ is written as a polynomial $a(z) = \sum_{i=0}^{m-1} a_i z^i \mod p(z)$ with coefficients $a_i \in \mathcal{F}_p$. Addition of field elements in polynomial representation can be performed efficiently as a componentwise addition mod p.

[1] The gcc compiler flags `-ffunction-sections -fdata-sections` and linker flag `-gc-sections` are used to achieve this.

In the binary case with $a_i \in 0, 1$, addition is reduced to a simple XOR operation. Defining a field element as a binary word where each bit represents one coefficient a_i allows us to perform addition word-wise instead of component-wise. More precisely, if the field element type `gf_t` is defined as `uint16_t`, the 8-bit microcontroller needs two cycles to perform the addition `gf_t a,b,c; c = a^b;` because it internally only works on 8-bit words. A single byte would not have enough space for typical field sizes such as 2^{10} to 2^{13}.

In the non-binary case, it seems reasonable for the polynomial field element presentation to define `gf_t` as an array, where – depending on the size of p – multiple coefficients are packed into one array element. For performance reasons, one finite field element should always be contained on one array element, instead of allowing overlaps from one array element to the next. This ensures that not too much memory is wasted per field element while the complexity of field operations is kept relatively low. Nevertheless, operations cannot be performed word-wise as in the binary case, hence we expect a significant performance drop. The XOR operation needs to be replaced by an addition modulo p.

The non-binary case is considered in the implementation by using inlined functions such as `gf_add` instead of directly writing XOR for addition. For the binary case, inlining ensures that the compiler optimizes the overhead away, effectively replacing the `gf_add` call with a XOR. In the non-binary case, it must be evaluated whether inlining these functions is practicable due to the increased code size, since the operation is more complex than a XOR.

Exponential representation The exponential representation uses the fact that every field element except zero can be represented as a power of the generator element α, such that $a = \alpha^i$ where $0 \leq i < p^m - 1$. This representation allows for example an efficient multiplication of two elements $a = \alpha^x, b = \alpha^y$ as $a \cdot b \equiv \alpha^x \cdot \alpha^y \equiv \alpha^{x+y \mod p^m-1}$. Division works analogously by substracting the exponents ($\alpha^{x-y \mod p^m-1}$). Squaring can be implemented as a left shift in the exponent ($(\alpha^x)^2 \equiv \alpha^{x<<1 \mod p^m-1}$). Exponentiations are computed using Square & Multiply. Square roots can be computed as a right shift if the exponent is even, otherwise by a sequence of left shifts and modulo reductions. Inversion is a simple negation of the exponent ($(\alpha^x)^{-1} \equiv \alpha^{-x \mod p^m-1}$).

For exponential representation, it is useful to store the field element in form of its exponent. For typical parameters in the binary case, the exponent happens to fit in the same two-byte value as the polynomial representation. The ability to store both representations in the same datatype `gf_t` is advantageous in many cases due to the frequent conversions, so we decided to use `gf_t` in both cases. However, in the non-binary case, a separate type for the different representations seems more useful, requiring refactoring mainly of the `gf_*` and some `poly_*` functions.

Lookup tables To allow a fast transformation from polynomial to exponential representation and vice versa, we use two lookup tables, which are typically called *log* and *antilog* or *poly* and *exp* in the literature. Calling `GFLOG(x)` returns the exponent i for $x = \alpha^i$ and

GFEXP(i) returns the element x generated by α^i. The zero element is inserted as α^{p^m-1}, which would usually refer to the first element again, since $\alpha^{p^m-1 \mod p^m-1} \equiv \alpha^0$. Hence, the tables contain GFLOG(0)= $p^m - 1$ and GFEXP $(p^m - 1) = 0$. This always needs to be handled as a special case.

The table for polynomial representation requires at least $p^m \cdot \log_2(p) \cdot m$ bits of memory. In practice, there is an overhead. In the binary case, we use two bytes per field element, thus having an overhead of $16 - m$ bits per element, where m is typically ≤ 13. In the non-binary case, the overhead also depends on the number of bits used per coefficient. For the exponential representation table, we need to store an exponent for each field element, which typically fits in two bytes. Hence the table uses $p^m \cdot 16$ bits of memory.

Since these lookup tables are too big to fit in SRAM for typical parameters, they are precomputed and stored in 'near' flash memory, below the 64 KByte boundary.

4.3.2 Avoiding duplicate conversions (The FASTFIELD switch)

Access to the lookup tables in flash memory is slow, therefore we implemented several important functions in a way that reduces unnecessary conversions, if the optimization is enabled with #define FASTFIELD TRUE. These duplicate conversions often occur as sequences of field additions and multiplications, especially during the handling of polynomials.

Polynomials in $\mathbb{F}_{p^m}[z]$ are implemented as a struct poly_t, which stores the maximum size and current degree of the polynomial and a pointer to an array of coefficients, which are field elements gf_t . A simple case of a FASTFIELD optimization is shown in Listing 4.1 for polynomial evaluation. Using this approach and neglecting the multiplications with zero, for a polynomial of degree t we need $2t+1$ lookups, whereas $3t$ lookups are needed using the standard multiplication gf_mul.

This concept can be applied in any place where some value which is constant for the duration of a code section is repeatedly multiplied with a variable value. More complex cases can be found for example in polynomial multiplication and division, as well as in the generation of the syndrome and parity check matrix.

4.4 Key management

In this section, the matrix datatype matrix_t is introduced and used to describe the storage and generation of the key matrices for McEliece and Niederreiter.

4.4.1 Matrix datatype

The matrix datatype matrix_t is used for mainly two tasks: storage of binary vector data such as a codeword or binary syndrome representation, and for matrix data such as the generator matrix and parity check matrix. It is designed as a struct holding the number of rows (rows) and columns (cols) and a pointer to a byte[2] array (data) of size

[2] More precisely, the date type can be specified defining MATRIX_DATATYPE and the bit order by setting MATRIX_DIRECTION. The datatype should be set to uint8_t since we target an 8-bit microcontroller, and setting the direction to RIGHT is recommended since some functions have been optimized to this

```
gf_t poly_eval(poly_t p, gf_t a) // Evaluate polynomial p at position a
{
    if(a==0)return p->coeff[0]; // p(a=0) = p_0 (constant term of polynomial)

    a = GFLOG(a); // switch to exponential representation of a = alpha^i
    gf_t r = poly_lc(p); // set result r to leading coefficient of polynomial

    // for all coefficients from the highest degree to 0...
    int16_t i;
    for (i = p->deg - 1; i >= 0; i--)
    {   // ... compute r += current coefficient * alpha^i
        if(r==0)r=p->coeff[i];
        else
        {   // r = gf_add(p->coeff[i], GFEXP(gf_mod1(GFLOG(r) + a)));
            gf_t t = GFLOG(r); // convert r to exponential representation
            t = gf_mod1(t+a); // multiplication of a and t in exponential representation
            t = GFEXP(t); // for addition, converting to polynomial representation is required
            r = gf_add(p->coeff[i], t);
        }
    }
    return r;
}
```

Listing 4.1: FASTFIELD in poly_eval

$rows \times \lceil cols/8 \rceil$ bytes. For convenience, $\lceil cols/8 \rceil$ is stored in an additional struct field `words_per_row`. If `cols` is not divisible by 8, each row has an overhead of up to 7 bit.

For binary matrices, access to `matrix_t` occurs mostly on bit level, but there are several instances where efficient row-wise XORs can be used, for example during Gauss-Jordan elemination or vector-matrix multiplication.

The best access time could be achieved using the straightforward approach using an array of dimension $rows \times columns$ with one element per column. While this would present an disproportionate overhead in the binary case, it may be appropriate for larger non-binary parameters. However, for small fields up to $\mathcal{F}_{15}$ it seems more reasonable to combine multiple matrix elements into one array element, as done in the binary case. Then, the `MATRIX_GET`, `MATRIX_SET` macros would need to be adapted to accept and return values of desired length.

4.4.2 Key matrices

Recall from the construction of Goppa parity check matrices presented in Section 3.3.2 that elements of H are in $\mathcal{F}_{2^m}$. The same holds for the generator matrix G, which is directly derived from H. However, the ciphertext c is computed as a vector-matrix multiplication of a message vector m and the matrix G. Since a vector-matrix multiplication can be implemented efficiently using row-wise XOR operations in the binary case, it is useful to map the $n \times t$ matrix H_{2^m} to an $n \times m \cdot t = n \times n - k$ matrix H_2. This mapping is depicted in Figure 4.1. Each element of H_{2^m} constructed according to Eq. 3.9 is then transformed into a binary column vector of m bits length. The generator matrix G_2 is

setting. These settings are assumed for the remainder of this thesis.

then derived directly from H_2. The reverse operation is performed during the generation of the syndrome polynomial from the binary syndrome vector.

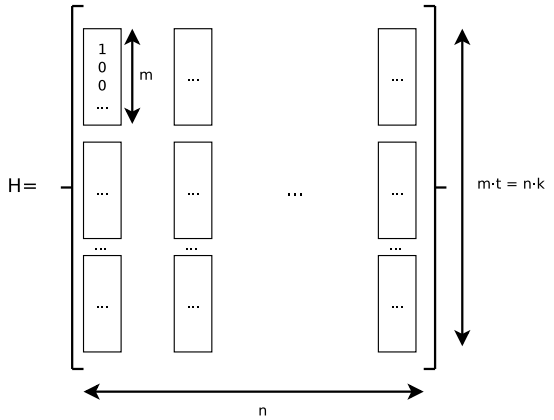

Figure 4.1: Mapping of parity check matrix elements in $GF(2^m)$ to $GF(2)$

4.4.3 Key generation

The key generation algorithms for the McEliece and Niederreiter cryptosystems have already been discussed in Chapter 2. We implemented only the 'modern' variants, since the classical variants have severe disadvantages, particularly the even higher memory requirements. The key generation procedures for McEliece and Niederreiter differ mainly in the decision which matrices need to be stored as public or private key, and which ones are only temporary. Of course, all matrices are stored in systematic form where possible, i.e. the identity part is not stored explicitly.

- In the McEliece cryptosystem, the systematic generator matrix G makes up the public key. It is derived from the parity check matrix H, which may be stored to allow a faster syndrome computation, but is typically discarded due to its size. The secret key consists mainly of the secret permuted support L and the Goppa polynomial $g(z)$.
- In the Niederreiter cryptosystem, the systematic parity check matrix H forms the public key, while the computation of G is not required. However, Niederreiter requires the matrix S to be retained, which brings the parity check matrix to systematic form, since it is needed for decryption. Hence, the secret key is identical to the McEliece secret key with the addition of S.

The key generation is typically not executed on the device due to the high memory requirements. Note that for typical parameters, it is impossible to fit the key generation data into SRAM (16 kByte) or EEPROM (4 kByte) completely. Hence, the implementation would need to write to the flash memory during execution of the code, which is possible due to a feature called Self programming Program memory that all recent AVR

microcontrollers provide. However, for most applications it seems to be sufficient to use precomputed keys.

This is accomplished by performing a precomputation run on a platform having more memory available. The `KEYSTORE` switch directs the key generation function to write all key data specific to the selected cryptosystem to a file, which is either a `.h` file containing C code or a `.s` file with Assembler instructions as discussed in Section 4.1. In the following compilation step, the `KEYSTORE` switch (which is implied by the `AVR` switch in current configuration) the key data is hardcoded and uploaded to the device together with the program code. Using the enclosed Makefile, the procedure is `make clean && make KEYSTORE=1 && ./cbc && make clean && make AVR=1 && make upload`.

Since the keys are precomputed or at least do not need to be computed frequently, performance was considered a subordinate priority to simplicity and memory usage. The procedure is roughly as follows:

- Select an irreducible monic polynomial. Although there are faster methods for constructing irreducible polynomials [Sho93], we opted for the simple yet effective method of choosing a polynomial at random and then testing its irreducibility using Rabin's algorithm [Rab80].
- Next, the support is selected. If n was chosen to be p^m, the support consists of all field elements of F_{p^m}. Otherwise, only n of p^m elements are chosen. In both cases, the elements are chosen in random order to obtain a permuted support, which inherits the role of the permutation matrix P originally used by McEliece.
- Then, the non-systematic parity check matrix H_{full} is constructed according to Eq. 3.9, as shown in a simplified version in Listing 4.2. H_{full} is then passed to the Gauss-Jordan elimination `matrix_gaussjordan()`, which takes the support as an additional parameter. The Gauss-Jordan algorithm brings a matrix to systematic form using row and column swaps. Whenever a column is swapped, also the corresponding elements in the secret support need to be swapped.

  If Niederreiter is used, the matrix passed to Gauss-Jordan is prepended with an identity matrix. Then, Gauss-Jordan does not only bring the matrix to systematic form, but additionally outputs the matrix that brings H_{full} to systematic form, which is exactly the required matrix S. In this case, special care must be taken when searching for a column to swap: since $(ID|H_{full})$ has more than n columns, corresponding support elements are only available for the first n columns, hence the remaining columns may not be swapped at all.
- If McEliece is used, the generator matrix G is derived from H according to Definition 3.2.4.

Note that all data that is not discarded needs to be allocated outside the key generation function if `alloc` is to be avoided, either as a global variable or in some superior function. For this implemetation, we decided to define macros `KEYGEN_INIT` respectively `KEYGEN_LOAD`, which allocate respectively references the required memory chunks as arrays and are

called in the main function. For easy handling, all key-related variables are combined in a struct called `cryptocontainer` which is provided as an argument to all functions that require some of the contained data. This was done mainly to provide a streamlined interface to all functions independent of the currently selected variant, without spilling the global namespace.

4.5 Encryption

Remember from Chapter 2 that encryption for both McEliece and Niederreiter involves an encoding step, a straightforward vector-matrix multiplication and for McEliece the addition of an error vector to the multiplication result.

4.5.1 Encoding

Encoding a plaintext message M of length x bits for the McEliece cryptosystem is the process of splitting M into parts of length $\leq k$ bits. Although no standard procedure has been introduced so far, the most obvious way to handle message parts with less than k bits is to pad them with zeros. As a proof of concept, this has been implemented using the functions `matrix_readbits()` and `matrix_writebits()`, which use the `matrix_t` type to provide bit-wise access to a file or a memory buffer of arbitrary length. However, for the evaluation we concentrate on the encryption of fixed-size blocks. The encryption of arbitrary length data has not been optimized or evaluated for security in any way. Instead, a focus has been placed on the CCA2-secure conversion, which also deals with the conversion of a message to an ciphertext in a secure way. It will be discussed in Section 4.8.

In the Niederreiter scheme, the encoding step is more complex, since the message needs to be encoded in a vector of length n with the constant weight t. This will be discussed in Section 4.7.

4.5.2 Multiplication

A multiplication of a plaintext vector with a matrix belonging to the public key occurs in both McEliece and Niederreiter. In McEliece, the plaintext vector is a codeword m and the matrix is the systematic generator matrix G, whereas the plaintext in Niederreiter is the error vector e and the matrix is the transposed systematic parity check matrix H^{T}.

Listing 4.3 shows the encryption in the case of McEliece and illustrates the usage of flash memory beyond the 64 kByte boundary, as described in Section 4.1. In the binary case, the multiplication $m \cdot G$ the multiplication is reduced to a row-wise XOR: if a bit in the plaintext is set, XOR the corresponding row of G to the ciphertext. Due to the `matrix_t` design, a *row-wise* XOR can be performed efficiently byte-wise, whereas the XOR of a row and a column would have to be executed bit by bit. Therefore the generator matrix is stored in transposed form. Note that only the non-identity part of G is stored, and the multiplication of the plaintext vector with the identity part has no effect. Hence,

```
// This is a simplified version of the generation of H. The original includes options to
// - construct the transposed matrix instead
// - construct only a partial matrix (for Berlekamp-Massey)
// - access the support from far flash memory (enabled by KEYLOAD switch)
void gen_matrix_H(crypt_t sk, matrix_t H){
    matrix_zero(H);
    poly_t g = sk->goppapoly;
    uint16_t i,j,k;
    for(i=0;i<GOPPA_n;i++) // iterate over all field elements belonging to the code
    {
        gf_t Li=sk->L[i]; // current field element
        gf_t denominator = poly_eval(g, Li); // g(L[i])

        #ifdef FASTFIELD
          uint8_t Li_not_0 = (Li!=0); // store whether Li is zero...
          Li=GFLOG(Li);               // ... before switching the representation
        #endif

        // iterate from leading coefficient of g(z) down to coefficient of z (i.e. no g_0)
        for(j=g->deg; j>=1; j--)
        {
            gf_t lc = poly_lc(g); // = g_s (leading coefficient)
            for(k = g->deg - 1; k >= j; k--)
            {
                // compute g_(s-i) * alpha_i^(s-i)
                #ifdef FASTFIELD
                  if(Li_not_0 && lc)
                  {
                      lc = GFLOG(lc);
                      lc = gf_mod1(lc + Li);
                      lc = GFEXP(lc);
                      lc = gf_add(g->coeff[k], lc);
                  }
                  else lc = g->coeff[k];
                #else
                  lc = gf_add(g->coeff[k], gf_mul(lc, Li));
                #endif
            }
            lc = gf_div(lc, denominator);

            for (k = (g->deg - j) * GF_m; lc != 0; k++, lc >>= 1)
            {   // bitwise copy of lc to H (H_{2^m} to H_2 mapping)
                if(lc & 1)MATRIX_SET1(H, k, i);
            }
        }
    }
}
```

Listing 4.2: Generate parity check matrix H (simplified)

the ciphertext c does not hold the *entire* codeword resulting of the multiplication but only the *parity part*, while the other part is available as the unchanged plaintext.

```
uint32_t G = FAR(KEYMATRIX); // KEYMATRIX address known at compile-time
MATRIX_DATATYPE *c = ciphertext->data; // pointer to 'data' element of matrix_t
for(i=0;i<GOPPA_k;i++){
    // if bit (i.e. column) i in plaintext is set, XOR line i of matrix G to ciphertext
    if(MATRIX_ISSET(plaintext, 0, i))
    {
        for(w=0;w<GENERATORMATRIX_WORDS_PER_ROW;w++) // iterate over every word of the current row
        {
            MATRIX_DATATYPE x = pgm_read_byte_far(G+w);
            c[w] ^= x;
        }
    }
    G+=GENERATORMATRIX_WORDS_PER_ROW; // points now to next row
}
```

Listing 4.3: McEliece encryption

The multiplication $e \cdot H^{\mathrm{T}}$ for the Niederreiter scheme is implemented similarly, but there are a few notable differences, presented in Listing 4.4. First of all, e is not available as a bit-vector. Instead, the error positions (i.e. the positions of ones in the error vector) are stored in an integer array. Moreover, the *entire codeword* needs to be constructed by the multiplication, as opposed to only the parity part as before.

```
for(i=0;i<GOPPA_t;i++){
    gf_t E=error_pos[i]; // error i signifies an one at position error_pos[i] in the error vector
    if(E<GOPPA_k) // error position in the non-identity part of H?
    {
        for(w=0;w<PARITYMATRIX_WORDS_PER_ROW;w++) // iterate over every word of the current row
        {
            MATRIX_DATATYPE x = pgm_read_byte_far(Hsys+E*PARITYMATRIX_WORDS_PER_ROW+w);
            c[w] ^= x;
        }
    }
    else { // error occured in the identity part of H
        // emulate XOR of row number E (which has only one '1' at pos E-GOPPA_k) with ciphertext
        MATRIX_ELEMENT_XOR(ciphertext, 0, E-GOPPA_k);
    }
}
```

Listing 4.4: Niederreiter encryption

In the non-binary case, the row-wise XOR needs to be replaced with the slower element-wise standard matrix multiplication modulo p, since the matrix elements belong to $\mathcal{F}_p$.

4.5.3 Error addition

As a final step, the McEliece scheme requires the addition of a random error vector to the codeword resulting from the multiplication. This is a simple XOR in the binary case, or an addition modulo p otherwise. Remember from Section 2.6.3 that the plaintext remains visible in the ciphertext at all positons without an added error. This is solved by the CCA2-conversions discussed in Section 4.8.

4.6 Decryption

Decryption in McEliece and Niederreiter is essentially the process of decoding the syndrome in order to find the error positions and if necessary the error values. In this section, we discuss implementational aspects of the decoding steps, which are roughly illustrated for all variants in the overview in Fig. 4.2.

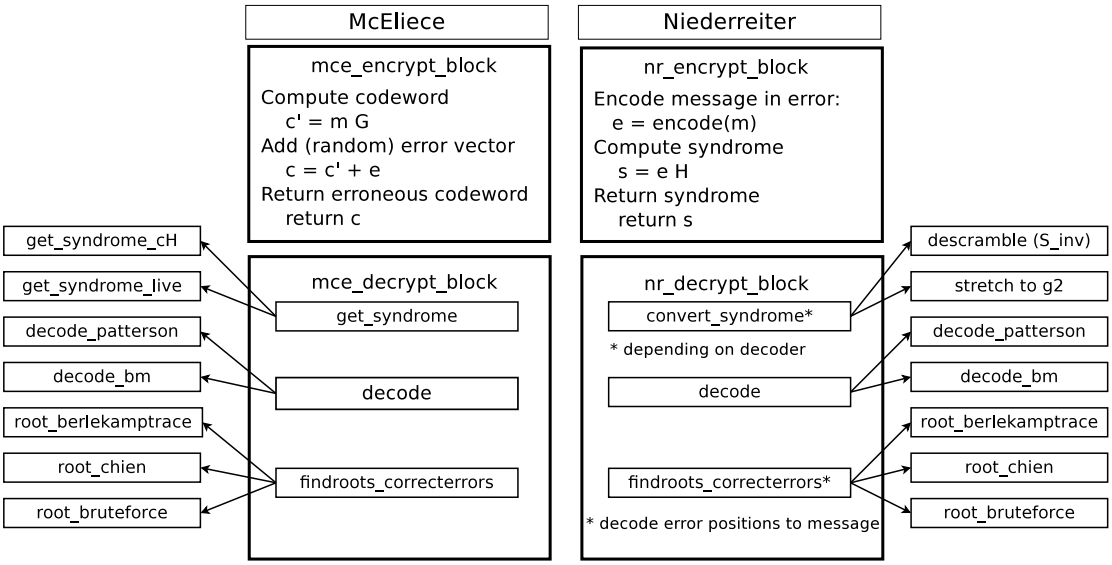

Figure 4.2: Overview on variants of the decryption process in McEliece and Niederreiter

4.6.1 Syndrome computation

In the McEliece scheme, the syndrome needs to be computed from the ciphertext. This can be done either by computing $c \cdot H^{\mathrm{T}}$ or by computing the relevant elements of H on the fly, avoiding the problem of holding the matrix in memory completely.

The ciphertext in the Niederreiter scheme *is* already a syndrome. However, the 'scrambling' caused by the transformation of the public key to a systematic first needs to be reverted. This is done by multiplying the syndrome with the reverse scrambling matrix S^{-1} which is part of the secret key. Moreover, to allow the Berlekamp-Massey (BM) algorithm to correct all errors in the binary case, the syndrome needs to be transformed to double length using Eq. 3.19.

In all cases, the result of these operations is a binary version of the syndrome, which needs to be turned into a polynomial. This is a straightforward process, since the binary version is a simple concatenation of fixed-width bit strings holding the binary representation of the polynomial coefficients.

4.6.1.1 McEliece: syndrome computation variants

Syndrome I (SYN_H) The most simple variant of computing the syndrome is the straightforward multiplication of the ciphertext $\hat{c} = c+e$ with a precomputed (or newly generated from the Goppa polynomial and support) parity check matrix H^{T}. We skip this variant here, since its implementation contains no new ideas. Note however that the multiplication must be performed using the double-sized parity check matrix based on $g(z)^2$ if BM for binary codes is used afterwards.

Syndrome II (SYN_EEA) From the syndrome definition and Eq. 3.7 we know that a syndrome satisfies the equation

$$\mathcal{S}_{\hat{c}}(z) \equiv \sum_{i=0}^{n-1} \frac{1}{z - \alpha_i} \mod g(z). \tag{4.1}$$

Hence, the syndrome can be computed by iterating over all support elements and inverting $z - \alpha_i$ using EEA, as shown in Listing 4.5.

If BM is used, `sk->goppapoly` holds $g(z)^2$ instead of $g(z)$ and the syndrome vector `syn` is of double length. The same applies to the third variant.

```
1  for(i=0;i<GOPPA_n;i++){ // iterate over all support elements respectively all positions in c
2      gf_t L_i = sk->L[i]; // lookup element in support
3
4      // the first k bits of the codeword are in sk->plaintext
5      // the remaining in sk->ciphertext (see encryption section)
6      if(i<GOPPA_k)
7          c_i = MATRIX_GET(sk->plaintext, 0, i);
8      else
9          c_i = MATRIX_GET(sk->ciphertext, 0, i-GOPPA_k);
10
11     if(c_i)
12     {
13         poly_zero(tmp); // set all coefficients of tmp to zero
14
15         // set tmp = z-alpha_i
16         tmp->coeff[1]=1; // z
17         tmp->coeff[0]=gf_sub(0, L_i); // -alpha_i
18         tmp->deg=1;
19
20         poly_inv(tmp, sk->goppapoly, tmp2); // EEA => tmp2 = (z-alpha_i)^-1
21         poly_add(syn, tmp2, syn); // sum up partial syndrome: syn = syn+tmp2
22     }
23 }
```

Listing 4.5: Syndrome decoding using EEA

Syndrome III (SYN_INVZA) The third variant is similar to the previous, but applies a trick described in [Pau10] to invert the polynomial without using EEA. Denoting the Goppa

polynomial as $g(z) = \sum_{i=0}^{t} g_i z^i$, the following relation holds:

$$\frac{1}{z - \alpha_i} \equiv \frac{1}{g(\alpha_i)} \sum_{j=s+1}^{t} g_j \alpha_i^{j-s-1} \mod g(z), \forall\, 0 \leq s < t-1 \tag{4.2}$$

Considering that $g(z)$ is monic, the syndrome can be computed as shown in Listing 4.6 using less operations than in the SYN_EEA variant.

```
1  for(i=0;i<GOPPA_n;i++){ // iterate over all support elements
2    // ... as above (cut for brevity) ...
3    if(c_i)
4    {
5      poly_zero(tmp);
6      gf_t c = gf_div(1, poly_eval(sk->goppapoly, L_i)); // 1/g(alpha_i)
7      tmp->coeff[GOPPAPOLY_DEGREE - 1]=c;
8
9      // from the highest Goppa polynomial coefficient that is not 1 downto 0
10     for(x=GOPPAPOLY_DEGREE-2; x>=0; x--)
11       tmp->coeff[x] = gf_add(gf_mul(c, sk->goppapoly->coeff[x+1]), gf_mul(L_i, tmp->coeff[x+1]));
12     poly_add(syn, tmp, syn);
13   }
14 }
```

Listing 4.6: Syndrome decoding using fast polynomial inversion

Side channel attacks on the syndrome computation The recent side channel analysis by Schacht [Sch12] also evaluates the three syndrome computation variants presented here. An attacker is able to reconstruct the secret key if he can extract the support L or Goppa polynomial $g(z)$ by observing side channels like execution time or power consumption. Given L and some valid codewords, $g(z)$ can be recovered using GCD and Eq. 3.15 [Uma11] or using the SSA [LS01]. SSA can also be used to compute L given $g(z)$ and the public key by constructing a generator matrix G' from $g(z)$ using an arbitrarily chosen $\hat{L}$ and applying SSA to G' and G [LS01].

Schacht shows that effective side channel attacks are possible on the syndrome decoding variants implemented for the *classical* McEliece scheme by reconstructing the secret permutation P. However, in the *modern* version implemented for this thesis, P does not exist anymore explicitly and hence cannot be recovered. The data-dependent instructions exploited in the attack on the classical version depend only on the *ciphertext* in the modern version. Since the ciphertext is public anyway, the attacker draws no advantage from this.

However, there are also attack vectors on the modern version. Even if any data-dependent instruction has been successfully avoided, data leaks are possible, for example due to the fact that the memory consumption of a transfer of a word on a data-bus depends on the value of the word. However, Schacht points out that these differences are far more difficult to exploit and do not provide enough information to recover the secret key completely, but only reduce the search space for an exhaustive key search.

4.6.1.2 Niederreiter: Constructing a syndrome of double length

The approach to transforming the Niederreiter-ciphertext – which is a syndrome – to a syndrome of desired length to achieve t-error correction for binary codes using the BM decoder has already been described in Section 3.4.3. A syndrome s_2 of double length is obtained by prepending the ciphertext s with zeros to construct a vector $(0|s)$ of length n, which is then multiplied with the parity check matrix H_2 computed over $g(z)^2$.

Since H_2 is constructed using the secret support, the inversion of the scrambling by multiplying the ciphertext with S^{-1} is performed implicitly during the calculation of s_2. Moreover, the first k bits of $(0|s)$ are zero. Hence the multiplication has only to consider the last $(n-k)$ bits of s and requires only the last $(n-k)$ columns of H_2. Therefore, our implementation constructs only the required parts of H_2 in order to improve computing time and memory efficiency, as shown in Listing 4.7.

```
1  void decrypt_init(crypt_t sk){
2    // ... cut for brevity
3    #if DECODE == BERLEMAS
4     poly_copy(sk->goppapoly, sk->goppapoly2); // copy g(z)
5     poly_square(sk->goppapoly2); // compute g(z) * g(z)
6     poly_swap(sk->goppapoly, sk->goppapoly2); // store in sk->goppapoly to allow access as usual
7     // the content of sk->goppapoly2 is discarded afterwards
8
9     #if CRYPTOSYSTEM == NIEDERREITER
10     // since sk->goppapoly now holds g(z)^2, this constructs H modulo g(z)^2
11     gen_matrix_H(sk, sk->Ht, TRUE, FALSE); // computes only required columns of H
12     #if defined(KEYSTORE)
13      store_hardcoded_berlemasH(sk->Ht->data, sk->Ht->words_per_row); // hardcode H for AVR
14     #endif
15    #endif
16   #endif
17   // ... cut for brevity
18 }
19
20 void nr_decrypt_block(crypt_t sk){ // niederreiter decryption
21   // ... cut for brevity
22   #if DECODE == BERLEMAS
23   // define syndrome vector of double syndrome size and set to zero
24   MATRIX_ZERO(s2, 1, 2*GOPPA_nk);
25   // more flexible access to the matrix-vector multiplication function by using the
26   // underlying 'raw access' function => (0|s) does not need to be explicitly constructed
27
28   #ifdef AVR
29   extern const MATRIX_DATATYPE *BERLEMAS_NIEDERREITER_H;uint32_t H = FAR(BERLEMAS_NIEDERREITER_H);
30   matrix_vectormul_raw_avr(sk->ciphertext->data, n-k, H, MATRIX_WORDS_PER_ROW(n-k), s2->data);
31   #else
32   matrix_vectormul_raw(sk->ciphertext->data, n-k, sk->Ht->data, sk->Ht->words_per_row, s2->data);
33   #endif
34   POLY_ZERO(s2poly, 2*t); // define "big" syndrome polynomial
35   binary_syndrome_to_polynomial(s2, s2poly, 2*t);
36   decode(sk, s2poly); // decode binary code using Berlekamp-Massey
37   #endif
38   // ... cut for brevity
39 }
```

Listing 4.7: Niederreiter: Syndrome stretching for Berlekamp-Massey decoding

Note that the H_2 – even without the first k columns – is too big to fit into AVR SRAM. Hence it either has to be precomputed and hardcoded as discussed for the other

key matrices in Section 4.4, or it needs to be live-computed. We opt for the first variant in order to avoid the repeated computational expensive computation during decryption.

4.6.2 Patterson implementation

The Patterson algorithm is a decoding algorithm for binary Gopppa codes and has already been described in Section 3.4.4. Analyzing Alg. 3.4.2 one can see that most steps for computing the error locator polynomial can be solved utilizing the standard set of polynomial functions that has been used throughout the thesis, including EEA for polynomial inversion and for solving Bézout's identity.

However one interesting step remains: Huber's algorithm [Hub96] for computing square roots, which is used to solve $R(z) \equiv \sqrt{T(z) + z} \mod g(z)$. Splitting $T(z) + z$ into an odd and even part $T(z) + z = R_0(z)^2 + z \cdot R_1(z)^2$ is a simple coefficient-wise operation. Thanks to Huber it is known that for every $g(z)$ there exists a polynomial $W(z)$ with $W(z)^2 \equiv z \mod g(z)$ such that $R(z) \equiv R_0(z) + W(z) \cdot R_1(z) \mod g(z)$.

$W(z)$ can be computed by splitting the Goppa polynomial into odd and even parts and using EEA to compute polynomials a, b such that $1 \equiv b \cdot g_0(z) + a \cdot g_1(z)$. Then, $W(z)$ can be computed as $W(z) = b(z) \cdot g_0(z) + za(z) \cdot g_1(z)$.

```
1  void poly_split(poly_t p, poly_t even, poly_t odd){
2      uint16_t d = p->deg/2;
3      for(i=0;i<d;i++){
4          even->coeff[i] = gf_sqrt(p->coeff[2*i]);
5           odd->coeff[i] = gf_sqrt(p->coeff[2*i+1]);
6      }
7      even->coeff[d] = gf_sqrt(p->coeff[2*d]);
8
9      if(xmod2(p->deg,2)) // if deg(p) odd
10         odd->coeff[d] = gf_sqrt(p->coeff[p->deg]); // 2*i+1
11 }
12
13 /// Precomputation of W(z) for decode_patterson_huber
14 void decode_patterson_huber_precompute(crypt_t sk){
15     poly_split(sk->goppapoly, even, odd); // split g(z)
16     poly_eea(even, odd, a, b, 0); // 1 = b * even(z) + a * odd(z)
17     // w(z) = b(z) * even(z) + z * a(z) * odd(z) = w0(z) + z * w1(z)
18     poly_mul(even, b, w0);
19     poly_mul(odd, a, w1);
20
21     // compute W(z)
22     W->coeff[0] = w0->coeff[0];
23     for(i=1;i<degree;i++)
24         w->coeff[i] = gf_add(w0->coeff[i], w1->coeff[i-1]);
25 }
26
27 /// Square root of p(z) mod g(z) (for rings of characteristic 2)
28 void decode_patterson_huber(crypt_t sk, poly_t p, poly_t result){
29     poly_split(p, even, odd);
30     poly_mul(odd, sk->patterson_huber_w, tmp);
31     poly_mod(tmp, sk->goppapoly);
32     poly_add(tmp, even, result);
33 }
```

Listing 4.8: Huber's algorithm in the Patterson decoder

Note that $W(z)$ depends only on the Goppa polynomial, hence it can be precomputed at startup. Listing 4.8 shows the implementation of the three relevant functions for the computation of $R(z)$.

4.6.3 Berlekamp-Massey implementation

The Berlekamp-Massey algorithm is a decoding algorithm for general Alternant codes and has already been discussed in Section 3.4.3. As soon as the syndrome is available in the required size, the computation of the error locator polynomial is reduced to a single call to EEA: `poly_eea_inv(syndrome, goppapolynomial, omega, sigma, t-1)`, where `t-1` is the stop value for EEA.

4.6.4 Root extraction using Berlekamp-Trace

The last remaining step is the evaluation of the error locator polynomial $\sigma(z)$ and the error value polynomial $\omega(z)$. For binary codes, the error value is always 1, hence $\omega(z)$ is not required. For non-binary codes it needs to be evaluated according to Eq. 3.18.

We discussed three methods for finding the roots of $\sigma(z)$ in Section 3.5. Using the detailed algorithmic description given there, implementing Chien search and Bruteforce evaluation using Horner scheme is a straightforward process, which will not be reproduced here. However, we will present some implementational details of the Berlekamp-Trace algorithm (BTA), which has been discussed in Section 3.5.3.

Recall from Alg. 3.5.3 that the basic idea of BTA is the factorization of $\sigma(z)$ by repeatedly computing a polynomial Greatest Common Divisor (GCD) until the degree of the resulting polynomials is small enough to solve the polynomial by other means. That is, in each recursion the algorithm computes $g = gcd(f, \mathrm{Tr}(\beta_i \cdot z))$, where $\mathrm{Tr}(\beta_i \cdot z)$ is a Trace polynomial computed using a basis $B = (\beta_1, \ldots, \beta_m)$ of the field $\mathcal{F}_{p^m}$.

We decided to use a *normal* basis $B = (\beta_1, \ldots, \beta_m) = (\beta_1^{p^0}, \beta_1^{p^1}, \ldots, \beta_1^{p^{m-1}})$ for our implementation. Its first element β_1 can be precomputed using the open source mathematical software SAGE as shown in Appendix B.1.2. In our implementation, the precomputed value is then defined in `config.h` as `NORMAL_BASIS` for each Finite field that is used.

The Trace polynomials can be precomputed efficiently and are stored in a sparse representation. Consider the field $\mathcal{F}_{2^{10}}$ with `NORMAL_BASIS` $= 32 = \beta_1$, which can be used to construct an instance of McEliece providing very short-term 60-bit security. Evaluating $\mathrm{Tr}(\beta_i z)$ according to Eq. 3.28 yields

$$\mathrm{Tr}(\beta_1 z) = \sum_{i=0}^{9} (\beta_1^{2^0} \cdot z)^{2^i} = \beta_1 z + (\beta_1 z)^2 + (\beta_1 z)^4 + \cdots + (\beta_1 z)^{2^9}$$

$$\mathrm{Tr}(\beta_2 z) = \sum_{i=0}^{9} (\beta_1^{2^1} \cdot z)^{2^i} = \beta_1^2 z + (\beta_1^2 z)^2 + (\beta_1^2 z)^4 + \cdots + (\beta_1^2 z)^{2^9}$$

$$\mathrm{Tr}(\beta_{10} z) = \sum_{i=0}^{9} (\beta_1^{2^9} \cdot z)^{2^i} = \beta_1^{2^9} z + (\beta_1^{2^9} z)^2 + (\beta_1^{2^9} z)^4 + \cdots + (\beta_1^{2^9} z)^{2^9}$$

One can see that z always occurs in powers of p. Hence, the Trace polynomial has degree p^{m-1}, but only m coefficients are unequal to zero. Moreover, one can see that the Trace polynomial is cyclic: the coefficient of z^2 in $\mathrm{Tr}(\beta_1 z)$ is the coefficient of z in $\mathrm{Tr}(\beta_2 z)$, and so on. Hence, the 'next' Trace polynomial can be computed from the previous one by performing a cyclic shift of the coefficients of z^{p^i} from the higher to the lower coefficient. Since $\beta_i^{p^m} = \beta_i$ in $\mathcal{F}_{2^{10}}$, the Trace polynomials repeat after m steps, i.e. $\mathrm{Tr}(\beta_{m+1} z) = \mathrm{Tr}(\beta_1 z)$.

Therefore we decided to precompute only $\mathrm{Tr}(\beta_1 z)$ and to store only the m coefficients unequal to zero. However, there is a problem remaining: if we want to retain a low memory footprint throughout the BTA procedure, we need to be able to actually work with the sparse representation during the BTA computations. Therefore, we implemented two versions of BTA:

- The computation of the polynomial greatest common divisor $gcd(\mathrm{Tr}(\beta_i \cdot z), \sigma(z))$ consists of a repeated modular reduction of the trace polynomial modulo $\sigma(z)$. Hence it is possible to perform such a reduction in advance using an algorithm that is able to work with sparse polynomial representations. Observing that our usual polynomial reduction function `poly_mod` works on a steadily moving window of only $t+1$ elements of the input and output polynomial, we implemented a sparse polynomial reduction function `poly_sparsemod`. Using this approach, the expanded polynomial is used at no stage, hence all polynomials have a degree of t or less. It can be activated using the `ROOT_BTA_SPARSE` switch.

- Otherwise, in every recursion the algorithm has to expand the sparse representation to a polynomial of full degre p^{m-1}. For 128-bit security parameters, such a polynomial has a size of 4 kB, which makes up 25% of the SRAM available on our target platform. However, a memory area shared across all recursion levels can be used for the expanded polynomial in all recursion levels. Implemented this way the big polynomial has to be allocated only once, instead of once at every recursion level.

Listing 4.9 shows relevant code parts of the Trace polynomial precomputation, usage in BTA and the modular polynomial reduction.

4.7 Constant weight encoding

Constant Weight (CW) encoding is the process of encoding a binary vector into a n-bit vector with weight t, i.e. n has exactly t bits set to '1'. Such an encoder is required for the Niederreiter cryptosystem to encode a plaintext into an error vector, but it is also required for McEliece for most CCA2-secure conversions. There exist several CW encoding algorithms, for example an enumerative and a recursive method described and compared in [OS09]. However, to suit the the constricted execution environment of microcontrollers, where deep recursions pose a problem and floating-point operations are very slow, it is recommendable to choose an algorithm optimized for such hardware.

```
/// PRECOMPUTATION of Tr(beta_1 z) for Berlekamp Trace Algorithm
void decrypt_init(crypt_t sk){ // ...
 int16_t base = NORMAL_BASIS; // first elemenet of a normal basis
 for(c=0; c < m; c++){sk->tracepoly[c]=base;base = gf_square(base);} // next element of basis
}

/// USAGE of Berlekamp Trace Algorithm
void root_berlekamptrace(poly_t sigma, crypt_t sk, uint16_t b){ // ...
#if ROOT_BTA_SPARSE == TRUE
 gf_t sparsepoly[GF_m]; // sparse polynomial with m coefficients unequal to zero, degree p^(m-1)
 for(i=0;i<m;i++)sparsepoly[i] = sk->tracepoly[xmod(i+b, GF_m)]; // shift precomputed values
 poly_sparsemod(sparsepoly, 1<<(m-1), m-1, sigma, tracepoly); // reduce modulo sigma(z)
#else
 // use global memory tracepolyShared across every recursion level
 POLY_FROM_ARRAY(tracepoly, 1<<(m-1), tracepolyShared); poly_zero(tracepoly);
 for(i=1,j=0; i<= 1<<(m-1); i*=p, j++)tracepoly->coeff[i] = sk->tracepoly[xmod(j+b, m)];
 // poly_mod(tracepoly, sigma); // not necessary!
#endif
 poly_gcd(tracepoly, sigma, sigma1); // sigma1 = gcd(sigma, Tr(beta_i z))
 poly_div(sigma, sigma1, sigma2); // sigma2 = sigma/sigma1
 b++;
 root_berlekamptrace(sigma1, sk, b); // continue recursively with sigma1 and ...
 root_berlekamptrace(sigma2, sk, b); // ... sigma2 which have a lower degree than sigma
 // ...
}

/// STANDARD modular polynomial reduction: P mod Q (P is input and output)
void poly_mod(poly_t P, poly_t Q){
  gf_t a, b; int16_t i, j, d; d = P->deg - Q->deg; // d = degree(P) - degree(Q)
  a = gf_inv(poly_lc(Q)); // a = inverse of leading coefficient
  for(i = P->deg; d >= 0; i--, d--){ // for every coefficient from P[highest degree] to P[d]
    if(P->coeff[i] != 0){ // ignore coefficient if 0
      b = gf_mul(a, P->coeff[i]);
      for(j = 0; j < Q->deg; j++) // P[j+d] = P[j+d] - b * Q[j]
        P->coeff[j+d] = gf_sub(P->coeff[j+d], gf_mul(b, Q->coeff[j]));
      P->coeff[i] = 0;
}}}

/// SPARSE modular polynomial reduction: P = sparse mod Q, deg(sparse)=sparsedeg=2^power
void poly_sparsemod(gf_t *sparse, uint8_t power, poly_t Q, poly_t P){
  gf_t a, b; int16_t i, ii, j, d; d = sparsedeg - Q->deg;
  uint16_t sparsedeg = 1<<power; // compute degree 2^power of sparse polynomial

  // initialize a cyclic cache holding the current window of sparse
  gf_t cache[GOPPA_t+1]; // holds values from i-t-1 to i-1
  for(i=0;i<=GOPPA_t;i++)cache[i]=0; // init to zero
  // set highest coefficient in cache to highest coefficient of sparse polynomial
  cache[xmod(sparsedeg, t+1)]=sparse[power]; // xmod(x,y) = x mod y
  power--;

  if (d >= 0){
    a = gf_inv(poly_lc(Q)); // a = inverse of leading coefficient
    for(i = sparsedeg; d >= 0; i--, d--){ // for every coefficient from P[highest degree] to P[d]
      if(i > GOPPA_t){ // fill cache with a new value (i-t) if neccessary
        ii = xmod((i - GOPPA_t), GOPPA_t+1);
        if((i - GOPPA_t) == (1<<power)){ // if i-t is a power of 2...
            cache[ii]=sparse[power]; // ... set value from sparse
            power--;
        }
        else cache[ii]=0;
      }

      ii = xmod(i, GOPPA_t+1);
      if(cache[ii] != 0) // ignore coefficient if 0
      {
          b = gf_mul(a, cache[ii]);
          for (j = 0; j < Q->deg; j++) // P[j+d] = P[j+d] - b * Q[j]
            uint8_t index_jd = xmod(j + d, GOPPA_t + 1);
            cache[index_jd] = gf_sub(cache[index_jd], gf_mul(b, Q->coeff[j]));
          }
          cache[ii]=0;
      }
    }
    for(i=0;i<GOPPA_t;i++)P->coeff[i] = cache[xmod(i, GOPPA_t + 1)]; // copy to output
  }
}
```

Listing 4.9: Trace polynomials in the Berlekamp-Trace Algorithm

Hence we use an approach by Heyse [Hey10] that is based on a recursive algorithm by Sendrier [Sen05], which Heyse redesigned to avoid recursion and replace slow operations.

Compared to some algorithms like Enumerative encoding [Cov06] that manage to reach the information theoretic upper bound on information that can be encoded in a constant weight word, the algorithm by Sendrier exhibits a small loss of efficiency. However, the fact that it has a complexity linear in the input length fully compensate this shortage for our purpose. Nevertheless one disadvantage remains: The length of input that can be encoded in a CW word is variable, and to the best of our knowledge, the lower bound is unknown and must be determined experimentally.

The implementation for this thesis relies heavily on the implementation by Heyse and has already been described at length in [Hey10]. However, it was hardcoded for parameters $n = 2^{11}, t = 27$, and has now been extended to work for arbitrary code parameters and adjusted to our implementation. It is able to encode approximately $\lfloor \log_2 \binom{n}{t} \rfloor$ bit into a n-bit vector with weight t. This value is precomputed, rounded to bytes and stored in the configuration macro `CWBYTES`. However, depending on n and t it has to be adjusted by a few bytes.

Apart from the rewrite to an iterative algorithm, the main idea of Heyse's modification is to avoid floating point arithmetic and division operations. This is done by modifying the procedure to compute an optimal value for the parameter d, which determines how many message bits are to be encoded into the current block of zeros. d depends on n and t, which constantly decrease until the encoding algorithm terminates. Originally, d is computed as

$$d \approx \left(n - \frac{t-1}{2}\right)\left(1 - \frac{1}{2^{1/t}}\right) \approx \frac{\ln(2)}{t}\left(n - \frac{t-1}{2}\right)$$

where the second approximation holds only for large enough t. Sendrier notes that restricting d to powers of 2 "greatly simplifies the encoding process and gives a significant advantage in speed while the loss of efficiency is very limited". Heyse follows this approach by computing d as 2^u and providing a lookup table for u that approximates the original value of d as computed using the second approximation. To keep the table small, it does not store the computed d for every possible combination of n and t, but uses a combination of n and t for the lookup, where the least significant bits hold the value of t, and the remaining bits hold the upper bits of n. Heyse showed that ignoring the lower bits of n keeps the difference to the original value d small.

Listing 4.10 shows the computation of the lookup table for u. It is written in Python since it is part of the precompilation step, which is written in Python entirely for the sake of simplicity. However, it can be easily converted to C code. The table is then written to the `precompile.h` file as an `uint8_t` array and loaded to flash memory or SRAM. For typical values such as $n = 2960, t = 56$ (128-bit security) the table size is approximately 3 kB.

If no lookup table shall be used to save memory, it is possible to approximate u by computing $(n - (t-1)/2)/t$ at runtime and mapping the result to the small range of possible values using a series of `if` and `else` conditions. This is essentially a smaller lookup table, which avoids a part of the expensive floating point arithmetic. However,

```
def bestU(N,T):
    tbits=int(math.ceil(math.log(t,2))) # number of bits of the binary representation of t
    T_MASK_LSB = ( (1<<tbits) - 1 )     # mask for selecting only the r least significant bits
    T_MASK_MSB = ctypes.c_uint32( (~T_MASK_LSB) & 0xffff).value # mask selecting all other bits
    maxindex=(N & T_MASK_MSB) + T + 1   # number of entries in the lookup table
    table=[0]*(maxindex)                # create the table, set all to 0

    i=0
    while i < maxindex: # loop over all table entries
        i+=1
        t = i & T_MASK_LSB # compute t from index i
        n = i & T_MASK_MSB # compute n from index i

        if t == 0:
            d=n
        elif n == 0:
            continue # table entry remains 0
        else:
            d = (ln2/t) * ( n - ( (t-1) / (2) ) )   # ln2=0.693147181
            # d=(n-((t-1)/2)) * (1 - math.pow(2,-1/t))

        if d > 0: # if d negative, table entry remains 0
            u = math.log(d, 2) # compute u such that d=2^u
            if u > 0:
                table[i-1]=u
    return table
```

Listing 4.10: Constant weight encoding: Generation of lookup table

the result is even less precise than the previously discussed table. Hence the number of bytes that can be encoded in a string of given weight and length may be further reduced.

4.8 CCA2-secure conversions

In Section 2.6.3 we discussed the notion of CCA2-security and its importance for the McEliece and Niederreiter cryptosystems. We will now introduce the Kobara-Imai-γ conversion (KIC) and Fujisaki-Okamoto conversion (FOC) and discuss implementational details. KIC is tailored to the McEliece cryptosystem, but can also be applied to Niederreiter. FOC is useful only for McEliece, because its main advantage is the omission of constant weight encoding, which is needed for Niederreiter anyway.

Both conversions require the use of a Hash function[App. B.2.4]. Moreover, FOC requires a Hash function providing two different output lengths. Therefore we decided to use Keccak[3] [BDPA11], which provides arbitrary output lengths. Relatively recently, Keccak has been selected as the winner of the NIST hash function competition and is now known

[3] More precisely, we use Keccak-f1600[r=1088,c=512], where f1600 is the largest of seven proposed permutations, r is the rate of processed bits per block permutation, $c = 25w - r$ is called the capacity of the hash function and $w = 2^6$ is the word size for the permutation. The authors of Keccak recommend to use smaller permutations (e.g. 25, 50, 100, 200, 400, 800) for constrained environments; moreover, it is possible to reduce w to any power of two. However, we decided to stick with the parameters proposed for the SHA-3 competition, as these are already carefully researched. Nevertheless, this should be considered for later optimizations

as SHA-3. The reference implementation also includes a version optimized for 8-bit AVR microcontrollers, which has been used for our implementation.

4.8.1 Kobara-Imai-Gamma conversion

Based on a generic conversion of Pointcheval [Poi00], Kobara and Imai [KI01] developed a CCA2-secure conversion that requires less data overhead than the generic one and can be applied to both McEliece and Niederreiter. Note that decreasing the overhead is useful without doubt, but overhead is not a major concern for public-key systems, because they are usually used only to transfer small data volumes such as key data.

KIC for McEliece Alg. 4.8.1 shows the Kobara-Imai-γ conversion applied to McEliece. It requires a constant string C, a hash function $\mathcal{H}$, a cryptographically secure pseudo random string generator Gen(*seed*) with a random seed and output of fixed length, a CW encoding and decoding function CW and CW^{-1}, and the McEliece encryption $\mathcal{E}$ and decryption $\mathcal{D}$. Note that the algorithm was simplified by omitting the optional value y_5 included in the original proposal, since it is not used in our implementation.

Algorithm 4.8.1: Kobara-Imai-γ conversion applied to McEliece

Encryption

$$
\begin{aligned}
y_1 &\leftarrow \mathrm{Gen}(r) \oplus (m||C) \\
y_2 &\leftarrow r \oplus \mathcal{H}(y_1) \\
(y_4||y_3) &\leftarrow (y_2||y_1) \\
e &\leftarrow CW(y_4) \\
\textbf{return}\, c &\leftarrow \mathcal{E}^{McEliece}_{K_{pub}}(y_3, e)
\end{aligned}
$$

Decryption

$$
\begin{aligned}
(y_3, e) &\leftarrow \mathcal{D}^{McEliece}_{K_{sec}}(c) \\
y_4 &\leftarrow CW^{-1}(e) \\
(y_2||y_1) &\leftarrow (y_4||y_3) \\
\hat{r} &\leftarrow y_2 \oplus \mathcal{H}(y_1) \\
(\hat{m}||\hat{C}) &\leftarrow y_1 \oplus \mathrm{Gen}(\hat{r}) \\
\text{IF } C = \hat{C}\, &\textbf{return}\, m \leftarrow \hat{m} \\
\text{ELSE } &\textbf{return } \perp
\end{aligned}
$$

KIC for Niederreiter KIC for McEliece has already been implemented and discussed in [Pau10]. Instead of reiterating it here again, we concentrate on the adaption of KIC

to Niederreiter, which has been implemented according to a proposal by Niebuhr and Cayrel [NC11].

KIC operates in a mode similar to a stream cipher, where Gen(*seed*) generates the keystream that is XORed to the message. Hence, only the seed needs to be encrypted directly by the Niederreiter scheme, whereas the message is encrypted by stream cipher in a way that approximates a one-time pad[App. B.2.5]. This allows the message to have a fixed, but (almost) arbitrary length, and it makes the ciphertext indistinguishable from a completely random ciphertext. The seed is cryptographically bound to the message using a Hash function. A publicly known constant string appended to the message allows the detection of modifications to the ciphertext.

The application of KIC to Niederreiter reflects the fact that in the Niederreiter scheme the plaintext is encoded only into the error vector e present in $\mathcal{E}_{K_{pub}}^{Niederreiter}(e)$. The message vector m as in $\mathcal{E}_{K_{pub}}^{McEliece}(m, e)$ is entirely missing from the Niederreiter encryption. Note that this causes a notable difference between KIC for Niederreiter and for McEliece: In the case of McEliece, the plaintext is encrypted using the *inherent* message m. Hence its length is determined by the McEliece system parameters[4]. On the contrary, KIC for Niederreiter adds an *additional* value to the ciphertext of the Niederreiter scheme and externalizes the message encryption completely.

Alg. 4.8.2 shows how KIC can be applied to the Niederreiter scheme. From the algorithm it is evident that the length of $(m||C)$ must be equal to the output length of Gen(r) and the length of the seed r must be equal to the output length of the Hash function $\mathcal{H}$. The length of m and C can be chosen almost freely, however it must be ensured that y_4 does not have a negative length. We chose C to be 20 Bytes long as suggested in the original proposal. The length of m has been set to 20 Bytes, too; however, for Niederreiter parameters achieving 256-bit security it has to be raised to a higher value. The length of the Hash output was chosen to be 32 Bytes. Table 4.1 lists all length requirements and declares the corresponding C symbols.

Note that it depends on the code parameters whether $|y_4|$ respectively $|y_3|$ is smaller or greater than $|y_2|$ respectively $|y_1|$. Hence the implementation must ensure that $(y_4||y_3) \leftarrow (y_2||y_1)$ and the respective step during decryption covers all possible cases, as shown in Listing 4.11.

4.8.2 Fujisaki-Okamoto conversion

The Fujisaki-Okamoto conversion (FOC) [FO99] is a generic CCA2-secure conversion which has been tailored to the McEliece cryptosystem by Cayrel, Hoffmann and Persichetti in [CHP12]. Using their improvements FOC does not require CW encoding, thus reducing both the design complexity and the runtime. The drawback is the need for an additional encryption operation during decryption and two Hash function calls during both encryption and decryption. However, McEliece encryption is computationally cheap, hence the decryption runtime is "still dominated by the decoding operation". Moreover, the fast encryption of the original McEliece scheme is usually affected less by

[4] Note that this can be changed by including the omitted value y_5.

Algorithm 4.8.2: KOBARA-IMAI-γ APPLIED TO NIEDERREITER

System parameters: Public constant C, Niederreiter system parameters

Encryption
Input: Binary message m
Output: Ciphertext c

$$
\begin{aligned}
y_1 &\leftarrow \mathrm{Gen}(r) \oplus (m||C) \\
y_2 &\leftarrow r \oplus \mathcal{H}(y_1) \\
(y_4||y_3) &\leftarrow (y_2||y_1) \\
e &\leftarrow CW(y_3) \\
\textbf{return}\, c &\leftarrow y_4|\mathcal{E}_{K_{pub}}^{Niederreiter}(e)
\end{aligned}
$$

Decryption
Input: Ciphertext $c = (y_4||s)$
Output: Binary message m

$$
\begin{aligned}
e &\leftarrow \mathcal{D}_{K_{sec}}^{Niederreiter}(s) \\
y_3 &\leftarrow CW^{-1}(e) \\
(y_2||y_1) &\leftarrow (y_4||y_3) \\
\hat{r} &\leftarrow y_2 \oplus \mathcal{H}(y_1) \\
(\hat{m}||\hat{C}) &\leftarrow y_1 \oplus \mathrm{Gen}(\hat{r}) \\
\text{IF } C = \hat{C} &\,\textbf{return}\, m \leftarrow \hat{m} \\
\text{ELSE } &\,\textbf{return}\, \perp
\end{aligned}
$$

Symbol	Description	Reason	C-Macro
C	Public constant	Chosen: 20 Bytes	`CONSTBYTES`
$\mathcal{H}(\cdot)$	Hash output	Chosen: 32 Bytes	`HASHBYTES`
m	Message	Chosen: 20/100 Bytes	`MESSAGEBYTES`
r	Seed	$\lvert r\rvert = \lvert\mathcal{H}(\cdot)\rvert$	`HASHBYTES`
y_1		$\lvert y_1\rvert = \lvert m\rvert + \lvert C\rvert$	`RANDBYTES`
y_2		$\lvert y_2\rvert = \lvert\mathcal{H}(\cdot)\rvert$	`HASHBYTES`
y_3		CW encoder	`CWBYTES`
y_4		$\lvert y4\rvert = \lvert y_2\rvert + \lvert y_1\rvert - \lvert y_3\rvert$	`NR_CCA2_y4`
c	Ciphertext	$\lvert c\rvert = \lvert y_4\rvert + \lvert\mathcal{E}_{K_{pub}}^{Niederreiter}(e)\rvert$	

Table 4.1: Length of parameters for Kobara-Imai-γ applied to the Niederreiter scheme

```
array_rand(seed, HASHBYTES); // generate seed
gen_rand_str(Genr, seed); // generate string of length RANDBYTES from seed

// y1 = Gen(r) xor (m||C)
for(i=0; i<MESSAGEBYTES; i++)
    y1[i] = Genr[i]^message[i];
for(i=MESSAGEBYTES; i<RANDBYTES; i++)
    y1[i] = Genr[i] ^ pubconst[i-MESSAGEBYTES];

// y2 = r xor Hash(y1)
cbc_hash(y2,y1,RANDBYTES);
for(i=0; i<HASHBYTES; i++)
    y2[i] ^= seed[i];

// y4 is leftmost NR_CCA2_y4 bytes of y2||y1
for(i=0; i<HASHBYTES && i< NR_CCA2_y4; i++)
    y4[i] = y2[i];
for(   ; i<NR_CCA2_y4; i++)
    y4[i] = y1[i-HASHBYTES];

// y3 is rightmost CWBYTES bytes of y2||y1
for(   ; i<HASHBYTES;i++)
    y3[i-NR_CCA2_y4] = y2[i];
for(   ;i<RANDBYTES+HASHBYTES;i++)
    y3[i-NR_CCA2_y4] = y1[i-HASHBYTES];

// Encode y3 into array of error positions, stored in pk->error_pos
BtoCW_it(pk, y3, CWBYTES);

// Encrypt error vector. y4 is already stored in pk->cca2_NR_KIC_y4
nr_encrypt_block(pk);
```

Listing 4.11: Kobara-Imai-γ conversion applied to Niederreiter: Encryption

two Hash function calls than by the use of CW encoding. Hence, Cayrel *et al.* argue that their construction "preserves the fast encryption better than the Kobara-Imai approach."

In the Niederreiter cryptosystem, the plaintext is encoded into the error vector, which always requires CW encoding by design. Hence, the advantage of the Fujisaki-Okamoto conversion does not apply, whereas the disadvantage of the additional encryption during decryption still applies. Therefore, we decided to implement Fujisaki-Okamoto only for McEliece.

Algorithm 4.8.3: Fujisaki-Okamoto conversion applied to McEliece

Encryption
Input: Binary message m
Output: Ciphertext c

$$\sigma \leftarrow \text{random vector of length } n \text{ and weight } t$$
$$r \leftarrow \mathcal{H}_1(\sigma||m)$$
$$c_1 \leftarrow \mathcal{E}^{McEliece}_{K_{pub}}(r, \sigma) = r \cdot G + \sigma$$
$$c_2 \leftarrow \mathcal{H}_2(\sigma) \oplus m$$
$$\textbf{return}\ c \leftarrow (c_1, c_2)$$

Decryption
Input: Ciphertext $c = (c_1||c_2)$
Output: Binary message m

$$\hat{\sigma} \leftarrow \mathcal{D}^{McEliece}_{K_{sec}}(c_1)$$
$$\textbf{return}\ \perp \text{ in case of decoding failure}$$
$$\hat{m} \leftarrow \mathcal{H}_2(\hat{\sigma}) \oplus c_2$$
$$\hat{r} \leftarrow \mathcal{H}_1(\hat{\sigma}||\hat{m})$$
$$\text{IF } c_1 = \mathcal{E}^{McEliece}_{K_{pub}}(\hat{r}, \hat{\sigma})\ \textbf{return}\ m \leftarrow \hat{m}$$
$$\text{ELSE } \textbf{return } \perp$$

Alg. 4.8.3 shows the application of FOC to McEliece, taking the improvements of Cayrel *et al.* into account. Similar to the Kobara-Imai conversion, FOC utilizes McEliece to encrypt a random seed σ which is used to generate a keystream using the Hash function $\mathcal{H}_2$. This is used to encrypt the plaintext m in a one-time pad[App. B.2.5] fashion by XORing it with the keystream. To avoid CW encoding, σ is chosen randomly such that its length is n and its weight is t. Generated this way, it can be used in place of the former error vector e without any need for encoding. Then, σ and and the plaintext m are cryptographically bound to each other using the Hash function $\mathcal{H}_1$. The result r takes the place of the former message m of the original McEliece scheme. Applying McEliece we obtain $\mathcal{E}^{McEliece}_{K_{pub}}(\dot{m} = r, \dot{e} = \sigma) = \dot{m}G_{sys} + \dot{e} = rG_{sys} + \sigma$. Since r is used only as a check value and no information on m can be derived from r, it does not matter that parts of it are visible in the McEliece ciphertext due to the usage of a systematic

generator matrix.

The decryption process reconstructs σ from the ciphertext using McEliece decryption. If decryption fails, the ciphertext may have been modified and the algorithm terminates with an error. From σ, the keystream can be recomputed to obtain the plaintext $\hat{m}$. To check whether $\hat{m}$ is the actual plaintext m without any modification, r is recomputed and fed into McEliece encryption. If the result matches c_1, the unmodified plaintext m has been decrypted successfully. Otherwise the ciphertext and hence r, σ or m have been detected to be modified and the algorithm terminates with an error.

Listing 4.12 shows the decryption of a ciphertext using FOC. Note that for high security parameters, encryption and decryption do not fit in the AVR memory at the same time. The implementation includes options to perform only encryption or decryption; however, using FOC the decryption-only switch cannot be used, since the decryption uses an encryption operation.

```
// Since the ciphertext (c1) is modified during McEliece decryption, but is required
// for later verification, it needs to be copied. Note that for efficiency reasons,
// the ciphertext is stored across the plaintext and ciphertext memory
matrix_clone(sk->plaintext, c1_copy_pt); matrix_clone(sk->ciphertext, c1_copy_ct);

// Decrypt c1 to obtain error positions (equivalent to sigma) and r.
// Note that we simply ignore r and instead recompute it from H1(sigma||m) later
mce_decrypt_block(sk);

// Construct sigma from error positions, but allocate an array large enough to hold (sigma||m)
uint8_t sigmax[n_in_bytes + MESSAGEBYTES];
MATRIX_FROM_ARRAY(sigma, 1, GOPPA_n, sigmax); // transform first part of sigmax to matrix sigma
matrix_zero(sigma); // set sigma to zero
for(i=0; i < CODE_ERRORS; i++) // iterate over error positions and set corresponding bits
    MATRIX_SET1(sigma, 0, sk->error_pos[i]);

// Compute a hash only over sigma, i.e. the first part of sigmax.
// Store hash value directly to second part of sigmax, where m will be constructed
cbc_hash(&(sigmax[n_in_bytes]), sigmax, n_in_bytes);

// m = h2(sigma) XOR c2, i.e. XOR c2 to the second part of sigmax
for(i=0;i<MESSAGEBYTES;i++) sigmax[n_in_bytes+i] ^= sk->cca2_fujimoto_c2[i];

// compute hash of sigmax and write it to r, where it is taken from for encryption
uint8_t *r = sk->plaintext->data;
cbc_hash(r, sigmax, n_in_bytes+MESSAGEBYTES); h1(sigma||m)

mce_encrypt_block(sk); // if r G + sigma == c1: m is unmodified plaintext (SUCCESS)
if( matrix_cmp(sk->plaintext, c1_copy_pt) != 0 || matrix_cmp(sk->ciphertext, c1_copy_ct) != 0)
    DIE("FAIL");

for(i=0; i<MESSAGEBYTES; i++) message[i]=sigmax[n_in_bytes+i]; // copy plaintext to output
```

Listing 4.12: Decryption of Fujisaki-Okamoto conversion applied to McEliece

5 Evaluation

This chapter evaluates the implementation of McEliece and Niederreiter over binary Goppa Codes with regard to the memory usage and runtime on an AVR ATxmega256A3 microcontroller with 256 kB flash memory and 16 kB SRAM. Section 5.1 discusses the memory usage and compares the utilized key lengths to traditional public-key cryptosystems such as RSA. Section 5.2 analyzes the runtime of the different cryptosystems, decoding methods, conversions and optimizations and finally compares it to other implementations.

5.1 Memory usage

In this section, we analyze the memory usage of the implementation, which is dominated by the size of the public key.

5.1.1 Key size

The main problem hindering the wide acceptance of Code-based cryptography is the huge key size. Table 5.1 compares key sizes for parameters used in this implementation to recommended key sizes for conventional cryptosystems that reach a comparable security level according to the current analysis[1] by ECRYPT II [Eur12].

We restrict our analysis to the 'modern' version of McEliece and Niederreiter, as presented in Section 2.4 and 2.5. Hence, G_{sys} and H_{sys} refer to the public key matrices in systematic form, both having $n \cdot (n-k)$ elements. For comparison, also the size of the

[1] Note that RSA refers here to general cryptosystems based on the integer factoring problem, DLOG to general Discrete Logarithm based cryptosystems and ECC to general cryptosystems based on Elliptic curves. A table showing an estimation of required parameters to achieve a reasonable security level for McEliece until 2050 is provided by [NMBB12], along with security-equivalent parameters for RSA and ECC.

Security	Code parameters				McEliece		Niederreiter			RSA	ECC /
level	m	n	k	t	G_{sys}	G	H_{sys}	H	S^{-1}		DLOG
60 bit	10	1024	644	38	29 kB	80 kB	29 kB	47 kB	17 kB	~816 bit	~128 bit
79 bit	11	1632	1269	33	56 kB	252 kB	56 kB	72 kB	16 kB		
80 bit	11	2048	1751	27	63 kB	437 kB	63 kB	74 kB	10 kB	1248 bit	160 bit
128 bit	12	2960	2288	56	187 kB	826 kB	187 kB	242 kb	55 kB	3248 bit	256 bit
256 bit	13	6624	5129	115	936 kB	4 MB	936 kB	1208 kB	272 kB	15424 bit	512 bit

Table 5.1: Comparison of key sizes of Code-based and conventional cryptosystems

equivalent non-systematic matrices is listed. Note that Table 5.1 shows the theoretical size, whereas the implementation stores the matrices with an overhead of up to 7 bits per row, as described in Section 4.4.1.

For the Niederreiter cryptosystem, also the inverse matrix of S is required to revert the permutation of the ciphertext that comes with the systematic parity check matrix. Hence, the $(n-k) \cdot (n-k)$ matrix S^{-1} needs to be stored as a part of the private key. Note, however, that the difference of memory consumption for H and H_{sys} is relatively small. Hence, for smaller parameter sets – where the difference amounts to only a few kilobyte – it may be reasonable to refrain from using a systematic parity check matrix. Then, S can be generated at random and only the seed for the random generator needs to be stored, while S^{-1} can be generated on-the-fly during decryption.

The private key also consists of the permuted support $\mathcal{L}$ with n elements in $\mathcal{F}_q$ and the Goppa polynomial $g(z)$ with t coefficients. Given the representation of field elements as `uint16_t`, the storage of the permuted support requires between 2 kB (60-bit security) and 13 kB (256-bit security) and the Goppa polynomial requires approximately 76 Byte to 230 Byte.

Obviously, the major problem concerning memory usage is the encryption. G_{sys}, H_{sys} and S^{-1} clearly do not fit into the 16 kB SRAM, hence they must be written to flash memory, providing slow access times. Moreover, since the matrices are mostly larger than 64 kB, they are placed in 'far' flash memory beyond the 64k-boundary, which makes access even less efficient. For 256-bit security, a public key of nearly 1 MB size is required. Hence, using this parameter set, encryption is not usable on our target platform without external memory. Using the Niederreiter cryptosystem, even decryption is not usable due to the large matrix S^{-1}. Recalling that the Fujisaki-Okamoto conversion requires an encryption step during decryption, this conversion is also barred from execution on the ATxmega256A3 for 256-bit security.

Finally, Table 5.1 also suggests not execute key generation on the target platform, especially since key generation has to deal not only with the non-systematic versions of H and G, but also with H and G at the same time if McEliece is used. Considering the security level 128-bit, the size of G is $n \cdot k = 826$ kB and the size of H is $(n-k) \cdot n = 243$ kB. Moreover, for the Niederreiter cryptosystem the matrix S^{-1} needs to be computed, which requires augmenting H and S with an identity matrix that needs to be stored explicitly during the key generation procedure for the Gauss-Jordan elimination. Considering that for most scenarios it seems reasonable to transfer precomputed keys to the device, no attempt was made at handling the key generation on the target device.

5.1.2 Message-related memory

In this subsection, the memory usage of message-related memory is discussed, which includes the plaintext and ciphertext, but also memory required for the syndrome polynomial and CCA2-secure conversions. Whereas the limitation in the previous subsection was mainly the available flash memory for holding the (constant) key data, we deal now with dynamic memory that needs to fit in the 16 kb SRAM.

Table 5.2 shows the most important elements and lists their size for 128-bit security

Conv.	McEliece		Niederreiter	
	Element	128-bit	Element	128-bit
–	Plaintext: k Bit	286 B	Plaintext: $t \cdot 2$ Byte (error positions)	112 B
			after CW encoding $\lfloor \log_2 \binom{n}{t} \rfloor$ Bit	49 B
	Ciphertext: n Bit	370 B	Ciphertext: $(n-k)$ Bit	84 B
	Syndrome: $t \cdot 2$ Byte	112 B	Syndrome: $t \cdot 2$ Byte	112 B
	with NR+BM: $2 \cdot t \cdot 2$ Byte	224 B	with NR+BM: $2 \cdot t \cdot 2$ Byte	224 B
KIC	y_1: `RANDBYTES`	323 B	y_1: `RANDBYTES`	40 B
	y_2: `HASHBYTES`	32 B	y_2: `HASHBYTES`	32 B
	y_3: reference to plaintext		y_3: reference to plaintext	
	y_4: `CWBYTES`	49 B	y_4: `NR_CCA2_y4`	23 B
	seed: `HASHBYTES`	32 B	seed: `HASHBYTES`	32 B
FOC	$(\sigma \vert M)$: k Bit + `HASHBYTES`	570 B	N/A	
	r: reference to plaintext			
	c_1: k bit	286 B		
	c_2: `HASHBYTES`	570 B		

Table 5.2: Message-related memory consumption with example size in Bytes (B) for 128-bit security

parameters as an example. Note that this table does not represent the actual memory usage, since on many occasions additional temporary memory is required and smaller structures are not considered. On the other hand, the implementation is able to save some memory by reusing the plaintext memory for the ciphertext. However, the table provides an overview over the dynamic memory consumption and demonstrates that SRAM poses no major problem to the implementation as long as care is taken that all constant parts are hold in flash memory.

5.1.3 Precomputations

Several precomputations or lookup tables are used throughout the implementation, most notably the *log* and *antilog* tables (GF tables) used for field arithmetic. For each item and parameterset, Table 5.3 lists the memory requirements, references the section where the item was introduced and states to which code path it applies and whether it can be disabled. For example, the precomputation of Huber's polynomial $W(z)$ for the Patterson algorithm cannot be disabled if Patterson is actually used, but otherwise it is not included.

For increasing parameters, some lookup tables like the GF tables quickly require too much memory to be hold in SRAM. Hence they can alternatively be accessed directly from flash memory without loading them to SRAM at startup. The table denotes whether an item is always accessed from SRAM or always from flash memory, or if both is possible (*) and determined using a compiler switch. The performance gain by accessing items from SRAM instead of from flash memory is analyzed in the next section. Note that all items shown in this table are accessed from 'near' flash memory

Item	GF tables	$W(z)$	$\mathrm{Tr}(z)$	$g(z)^2$	bestU	$\mathcal{L}^{-1}$	$g(i)$
Applies to	always	Patterson	BTA	BM	CW	Chien, BTA	`SYN_INVZA`
Optional	No	No	No	No	Yes	Yes	Yes
Section	4.3.1	4.6.2	4.6.4	4.6.1.2	4.7	3.5	4.6.1
Memory	*	SRAM	SRAM	SRAM	*	Flash	*
60 bit	2×2048 B	76 B	20 B	152 B	1063 B	2048 B	2048 B
79 bit	2×3264 B	66 B	22 B	132 B	1634 B	4096 B	3264 B
80 bit	2×4096 B	54 B	22 B	108 B	2076 B	4096 B	4096 B
128 bit	2×5920 B	112 B	24 B	224 B	3001 B	8192 B	5920 B
256 bit	2×13248 B	230 B	26 B	460 B	6646 B	16384 B	13248 B

Table 5.3: Memory consumption of precomputed items and lookup tables

below the 64k-boundary, which can be accessed more efficiently than memory beyond this boundary.

5.1.4 Program code size

AVR flash memory holds the program code as well as the data examined in the previous parts of this section. Hence, in order to be able to store large keys in flash memory, it is essential to keep the code base small. As discussed in Section 4.2 code parts that are not required for the current parameter set are removed from the executable. In this section, we examine the size[2] of the program code after substracting the memory required for the systematic key matrix G or H, the GF tables and the support and without any optional lookup tables. Note that the code base also includes UART communication functions, a few libc-functions like `memcpy` and some libgcc code apart from the Code-based cryptography functions.

Table 5.4 presents selected examples of combinations of the most important modes for two security levels:

- 80-bit security: 66540 Byte (65005 Byte without overhead) for the systematic key matrix, 8192 Byte for GF tables and 4096 Byte for support (Sum: 77293 Byte)
- 128-bit security: 192192 Byte (there is no overhead for this parameter set) for the systematic key matrix, 16384 Byte for GF tables and 5920 Byte for support (Sum: 214496 Byte)

The same compiler flags (i.e. `-Os` to optimize for size) have been used for the compilation of all examples. Some interesting and some less surprising results can be seen from these values.

- Using parameters for the 80-bit security level, the larger amount of available memory for the program code causes the compiler to produce larger code. This is done to optimize the code for speed, for example by inlining small functions.

[2]The tool `avr-size` or the `-Map` linker option can be used for this task.

System	MCE	NR	NR+KIC	MCE+KIC	MCE+FOC	MCE	MCE	MCE
Decoder	PAT	PAT	PAT	PAT	PAT	BM	PAT	PAT
Roots	HOR	HOR	HOR	HOR	HOR	HOR	BTA	CHI
80-bit	14030 B	25165 B	29597 B	18889 B	18376 B	10924 B	14846 B	14406 B
128-bit	12733 B	69074 B*	69074 B*	17658 B	17069 B	9615 B	13545 B	13119 B

Table 5.4: Size of executables without key matrices, GF tables and support

- Niederreiter has a higher code complexity than McEliece, mainly due to the fact that it requires constant weight encoding. However, if Niederreiter is used with the Kobara-Imai-γ conversion, the additional functionality causes nearly no increase in code size.
- The examples marked with an asterisk (*) show cases where the total size of the executable is larger than the available flash memory. This can be circumvented by not using encryption and decryption at the same time; however, the results would not be comparable.
- There is no significant difference in code size between McEliece with Kobara-Imai-γ or with Fujisaki-Okamoto conversion.
- The same goes for the root extraction variants, i.e. Horner scheme, Chien search and Berlekamp-Trace algorithm.
- The Patterson algorithm is more complex than Berlekamp-Massey, since BM consists mainly of a single call to EEA.

5.2 Performance

In this section, we evaluate the execution time of our implementation, comparing alternative computation methods and testing the effectivity of individual optimizations. For brevity, we denote by MCE60, ..., MCE256 the McEliece cryptosystem using parameters achieving a security level of 60-bit, ..., 256-bit according to Table 5.1, and respectively for the Niederreiter cryptosystem, denoted as NR60, ..., NR256.

5.2.1 Overview

To introduce the reader to typical performance values, we start by giving the cycle count for commonly used parameters without special optimizations. Table 5.5 presents the average amount of clock cycles for an encryption and decryption run of 80-bit McEliece and Niederreiter. Decoding is shown for both the Patterson algorithm (PAT) and the Berlekamp-Massey (BM). The syndrome in McEliece is computed using a precomputed parity check matrix H, i.e syndrome computation variant I (`SYN_H`). For root extraction, the simple bruteforce search using Horner scheme is used. For the combination of NR

McEliece			Niederreiter		
Operation	Cycles	%	Operation	Cycles	%
Encryption	994,056	13/14	Encryption	46,734	.8/.9
$c = m \cdot G$	987,615	99.35	CW encoding	16,120	34.49
$\hat{c} = c + e$	6,441	0.65	$c = eH$	30,614	65.51
Decryption (PAT)	6,196,454	86.18	Decryption (PAT)	5,577,774	99.17
Syndrome I: $s = cH$	942,940	15.22	Syndrome: $s = S^{-1}c$	141,563	2.54
Bin. syn. to polynomial	8,392	0.14	Bin. syn. to polynomial	7,982	0.13
Patterson	780,043	12.59	Patterson	854,553	15.32
$T = s^{-1}$	456,225	58.49	$T = s^{-1}$	527,110	61.68
$R = \sqrt{T + z}$	99,835	12.8	$R = \sqrt{T + z}$	99,824	11.68
EEA	217,768	27.9	EEA	221,205	25.89
$\sigma = a^2 + b^2$	6,355	0.82	$\sigma = a^2 + b^2$	6,414	0.75
Find roots & correct errors	4,434,585	71.5	Find roots & correct errors	4,493,003	80.55
			CW decoding	19,962	0.36
Decryption (BM)	6,868,866	87.38	Decryption (BM)	5,510,006	99.09
Syndrome $s = cH_2$	1,702,513	24.79	Syndrome $s_2 = sH_2$	255,228	4.63
BM (EEA)	716,809	10.44	BM (EEA)	741,172	13.45
Find roots & correct errors	4,425,157	64.42	Find roots & correct errors	4,473,280	81.18

Table 5.5: Performance of McEliece and Niederreiter (80-bit security)

and BM, syndrome stretching using a parity check matrix computed modolu $g(z)^2$ is performed. For constant weight encoding, the `bestU` lookup table is utilized.

The cycle count difference between runs with different keys amounts to few thousand cycles per encryption and decryption run. This was considered by averaging the cycle counts over several runs with different keys; however, the keys need to be written to the microcontroller manually between those runs, since key generation cannot be performed on the device. Averaging over thousands of different plaintexts is easy, since a loop count can be configured in selftest mode and plaintexts are generated at random, or can be fed into the device automatically via UART communication.

If percentages (respectively cycle counts) at the same indentation level do not sum up to 100% (respectively the value at the higher indentation level), the remainder is due to computational overhead like UART communication, or due to minor neglected steps of the algorithm, like the addition of the error vector to the codeword in McEliece. Two values are given for the percentage of encryption, where the first denotes the percentage of encryption compared to decryption using Patterson and the second the percentage compared to decryption using Berlekamp-Massey.

- **Niederreiter vs. McEliece** The most obvious result from Table 5.5 is the fact that encryption and decryption in the Niederreiter cryptosystem is faster than in the McEliece cryptosystem. Note, however, that a McEliece plaintext for these parameters is $k = 1751$ bit long, whereas the Niederreiter plaintext is only $\lfloor \log_2 \binom{2048}{27} \rfloor = 203$ bit long. Hence, McEliece needs 567.7 cycles per bit for

Cryptosystem+Decoder	MCE+PAT	MCE+BM	NR+PAT	NR+BM
Size of executable	168,739 B	240,663 B	103,408 B	111,388 B
Percentage of 256 kB	64.37 %	91.8 %	39.44 %	42.49 %

Table 5.6: Size of executables for Table 5.5 (80-bit security)

encryption and 3538.8 cycles per bit for decryption (Patterson), whereas Niederreiter needs only 230.2 cycles per bit for encryption, but 27476.7 cycles for decryption (Patterson). However, these values are not very useful for a fair comparison, since we know from Section 2.6.3 that CCA2-secure conversions are required to obtain a secure cryptosystem from McEliece and Niederreiter. We will see in the next sections that conversions affect the performance comparison considerably. Another aspect is that the performance of constant weight encoding is mostly insignificant to decryption, but not to encryption.

- **Encryption vs. Decryption** The biggest differences between McEliece and Niederreiter are found in the encryption operation. Remember that encryption is dominated by the multiplication of an vector (m or e) with a key matrix (G or H, which are both in systematic form and have the same size). However, for Niederreiter all but t positions of error vector e are zero, hence only t rows of H need to be loaded and XORed to the result vector. In contrast, a uniformly distributed random plaintext m in the McEliece on average has only $^k/_2$ zero elements, hence half of the columns of G need to be loaded and XORed to the result vector. Therefore, McEliece encryption is slow compared to Niederreiter encryption, despite of the constant weight encoding required for Niederreiter.

- **PAT vs. BM** For McEliece, the Patterson algorithm is faster than Berlekamp-Massey. For Niederreiter, BM is faster, but the difference is marginal. Moreover, BM requires a double-size parity check matrix $H_2 \bmod g(z)^2$ that has to be precomputed and stored in flash memory. As shown in Table 5.6, this causes significant differences in the flash memory consumption. For NR+BM, the first k columns of H_2 are omitted as described in Section 4.6.1.2, whereas MCE+BM requires the entire matrix. NR+PAT achieves the lowest size, since it requires only the matrices H and S^{-1}, but not H_2 or G.

- **Root finding** As previously discussed, the computationally most expensive step in decryption is the root extraction, which requires the evaluation of error polynomial $\sigma(z)$ for all elements of the support using this configuration. It also includes the correction of the error, but this can be neglected since it is very fast. Alternative methods of root finding are evaluated in Section 5.2.6.

- **Memory usage** For 80-bit security parameters, already 92% of the available internal flash memory on an ATxmega256 is used in one case. Therefore, a different configuration must be used to evaluate the 128-bit security level, and the results

PAT + KIC	McEliece				Niederreiter			
Security level	80-bit		128-bit		80-bit		128-bit	
Plaintext length	212 B		303 B		20 B		20 B	
Operation	Cycles	%	Cycles	%	Cycles	%	Cycles	%
Encryption	2,679,676	12.73	6,262,714	11.10	889,395	12.36	975,481	5.31
mG or eH	995,748	37.16	2,682,274	42.83	30,733	3.46	112,003	11.48
CW encoding	15,357	0.57	77,811	1.24	16,189	1.82	27.145	2.78
Hash	1,608,452	60.02	2,388,745	38.14	797,307	89.65	797,428	81.75
Bug[a]	27,559	1.03	972,582	15.53				
Decryption	18,378,293	87.27	50,159,835	88.90	6,306,332	87.64	17,395,092	94.69
Syndrome[b]	10,340,083	56.26	30,482,209	60.77	141,740	2.25	619,564	3.56
Patterson	773,132	4.21	2,929,471	5.84	784,868	12.45	2.925.665	16.82
$T = s^{-1}$	446,228	57.72	1,703,924	58.16	457,201	58.25	1,709,313	58.42
$R = \sqrt{T + z}$	99,996	12.93	400,179	13.66	99,900	12.73	399,984	13.67
EEA	220,444	28.51	816,574	27.87	221,275	28.19	816,368	27.90
Roots & errors	4,484,509	24.40	12,964,136	25.85	4,492,796	71.24	12,947,982	74.43
CW decoding	19,384	0.11	35,031	0.07	20,012	0.32	35,577	0.20
Hash	1,608,452	8.75	2,388,745	4.76	797,307	12.64	797,428	4.58
Bug	969,667	5.28	1,170,558	2.33				

[a] Unfortunately, there is a unresolved bug in the MCE+KIC implementation, which requires a workaround causing a significant overhead to encryption and to a lesser extent to decryption.
[b] McEliece: Syndrome means syndrome computation III (`SYN_INVZA`). Niederreiter: Syndrome means inverting the permutation of the syndrome by multiplying with S^{-1}.

Table 5.7: Performance of McEliece and Niederreiter using Kobara-Imai-γ conversion and Patterson decoder

will not be directly comparable. In particular, the parity check matrix H for the syndrome computation in McEliece must be replaced by a on-the-fly computation, as shown in Section 4.6.1. However, this comes with a significant performance penalty, as discussed in Section 5.2.4.

5.2.2 80-/128-bit security and the Kobara-Imai-γ conversion

For a comparison of different security levels and a more realistic setup using a CCA2-secure conversion, we now present McEliece and Niederreiter using the Patterson decoder and the Kobara-Imai-γ conversion (KIC) in Table 5.7. We concentrate on the 80-bit and 128-bit security level, since the 256-bit implementation cannot be run on the ATxmega256A3 and 60-bit parameters provide only very low security. Patterson was chosen since it requires less flash memory and because it is faster for McEliece and nearly as fast as BM for Niederreiter. KIC is used for the comparison, since applying the Fujisaki-Okamoto conversion (FOC) to Niederreiter is not reasonable, as argued in Section 4.8.2. Moreover, the syndrome computation variant III (`SYN_INVZA`) is used for

McEliece, because H_2 does not fit in flash memory for 128-bit security.

Using this configuration, encryption and decryption functionality can coexist in memory for all systems but 128-bit Niederreiter. As can be seen from Table 5.1, Niederreiter with 128-bit security level results in an systematic parity check matrix H of 187 kB and the reverse scrambling matrix S^{-1} of 55 kB, occupying 94.53% of the available memory. Hence, benchmarking must be performed separately for encryption and decryption in this case.

- **Niederreiter vs. McEliece** Whereas the cycle count difference between MCE and NR was relatively small in Table 5.5, a huge difference can be observed from Table 5.7. Encryption and decryption of NR128 are faster than those of MCE80 and even MCE128. The main reason is the slow live computation for the syndrome computation in the decryption procedure of McEliece, which is not required for the Niederreiter cryptosystem. Moreover, the vector-matrix multiplication eH in NR encryption is faster than mG in McEliece encryption, for the same reasons as discussed in the previous example. Note, however, that the factor by which the cycle count of the multiplication increases from 80 to 128-bit security is higher for NR than for McEliece; the same applies for the syndrome-related computations in decryption. Concerning the Patterson algorithm and root extraction,there is no significant difference between NR and MCE, as expected. The performance of constant weight encoding is now mostly insignificant for both decryption and encryption.

- **Hash function** KIC requires a Hash function for both encryption and decryption. As described in Section 4.8, Keccak-f1600[r=1088,c=512] is used. As a result, encryption is dominated by the Hash function, consuming 38% to 89% of encryption clock cycles and 4% to 13% of decryption cycles. However, the performance could be improved by utilizing lightweight parameters for Keccak. Furthermore, the input data length to the Hash function is *not* based on the security level in KIC for NR: the input length depends only on the size of a public constant value and the message length, which can both be chosen freely in some boundaries. Hence, in the above example, the Hash cycle count for NR80 and NR128 is nearly identical (Hash input length: 40B), whereas it differs for MCE80 (232B) and MCE128 (323B). The Hash output length is fixed at 32B for all cases.

- **Increasing the plaintext length** For calculating the data throughput, the cycle count must be set in relation to the processed information bits, i.e the plaintext length. For MCE+KIC, in our implementation the plaintext length depends on code parameters t (`CWBYTES`) and k. For NR+KIC, `MESSAGEBYTES` can be set to any large enough value. KIC essentially turns NR into a stream cipher, in which the Niederreiter cryptosystem is mainly used to encrypt the seed that generates the key stream. Hence, when increasing the plaintext length (`MESSAGEBYTES`), the basic NR operations are not affected. However, the larger plaintext increases the input length to the Hash function and it affects the key stream generator $Gen(r)$ and the copy and XOR operations of the conversion. The latter are very efficient (they

PAT + KIC	McEliece				Niederreiter			
Security level	80-bit		128-bit		80-bit		128-bit	
Plaintext length	212 B		303 B		212 B		303 B	
Operation	Cycles	%	Cycles	%	Cycles	%	Cycles	%
Encryption	2,679,676	12.73	6,262,714	11.10	1,674,540	19.12	2,549,599	11.48
mG or eH	995,748	37.16	2,682,274	42.83	30,682	1.83	111,647	4.38
Hash	1,608,452	60.02	2,388,745	38.14	1,576,430	94.14	2,352,123	92.25
Other	75,476	2.82	1,191,695	19.03	67,428	4.03	85,829	3.37
Cycles/Byte	12,640		20,669		7,898		8,415	
Bit/s at 32 Mhz	20,253		12,385		32,413		30,421	
Decryption	18,378,293	87.27	50,159,835	88.90	7,084,137	80,88	18,978,062	88.16
Syndrome	10,340,083	56.26	30,482,209	60.77	142,103	2.01	620,570	3.27
Patterson	773,132	4.21	2,929,471	5.84	784,420	11.07	2.925,427	15.41
Roots & errors	4,484,509	24.40	12,964,136	25.85	4,492,796	63.42	12,947,982	68.23
Hash	1,608,452	8.75	2,388,745	4.76	1,576,430	22.25	2,352,123	12.39
Other	1,172,117	6.38	1,395,274	2.78	68,356	0.96	131,960	0.7
Cycles/Byte	86,690		165,544		33,415		62,634	
Bit/s at 32 Mhz	2,953		1,546		7,661		4,087	

Table 5.8: Performance of MCE and NR with PAT+KIC using the same plaintext lengths

do not even occur explicitly in the above table), so that the only notable difference in the performance is the increased cycle count of the Hash function. Note that a similar construction can also be achieved using the McEliece cryptosystem, as mentioned in Section 4.8.

- **Data throughput** Table 5.8 shows the same measurement as the previous table, with the only difference being the plaintext size for the Niederreiter set to the same size as the plaintext of McEliece to allow a direct comparison. Moreover, the table shows the number of cycles per plaintext byte need for encryption and decryption and the data throughput at 32 Mhz. As stated before, this does not mean that McEliece is inherently slower than Niederreiter, but only that the current implementation allows the Niederreiter cryptosystem to use the stream cipher approach more efficiently. Moreover, the throughput of Niederreiter can be further increased by increasing the plaintext length until the performance is dominated by KIC-operations (Hash, XOR) instead of by the Code-based encryption and decryption procedures. However, it is often not desirable to set the plaintext length to big blocks.

5.2.3 Fujisaki-Okamoto conversion

The main advantage of the Fujisaki-Okamoto conversion (FOC) is the efficient application of a CCA2-secure conversion to the McEliece cryptosystem that avoids the need

for constant weight encoding. However, as we have already seen, on our implementation for AVR microcontrollers constant weight encoding is a minor factor, which amounts to a negligible cycle count in most cases, far below 1% of the total runtime. The only exception is shown in Table 5.5, where CW encoding amounts to 34% of the Niederreiter encryption runtime, if Niederreiter is used without a CCA2-secure conversion. However, this was due to the very fast encryption in the Niederreiter cryptosystem, which does not apply to McEliece to the same degree.

Moreover, we already saw in the previous section that the Hash function – which is called *twice* in encryption and decryption – consumes a significant percentage of cycles (although this problem could be reduced using a lightweight hash function). Furthermore, an additional call to McEliece encryption occurs during FOC decryption. While McEliece encryption is fast in general, encryption on the AVR platform is less performant than on other platforms that provide a faster access to the key matrix.

Finally, the plaintext length is determined by the output length of the Hash function $\mathcal{H}_2$. Although the output length can be arbitrarily chosen for Keccak, it needs to be fixed at compile time for the currently used implementation. Since the same Hash function is used for $\mathcal{H}_1$, whose size is determined by the parameter k of the underlying Goppa code, k determines also the plaintext length in our implementation. This prevents us from increasing the plaintext length until the performance is dominated by FOC-operations (Hash, XOR) instead of by the McEliece procedures.

Therefore, it comes with no surprise that the Fujisaki-Okamoto conversion yields a worse performance than the Kobara-Imai-γ conversion. Encryption is substantially slower due to two Hash function calls (making up roughly 80% of encryption), whereas the performance loss of decryption is less significant, since the Hash procedure amounts only to 15% to 25% of decryption. The results are summarized for 60- to 128-bit security in Table 5.9.

5.2.4 Syndrome computation variants

Two of the three syndrome computation variants for the McEliece scheme described in Section 4.6.1 have already been used during the evaluation. It turned out that Syndrome I (`SYN_H`) is very fast, but can be used only for low security parameters due to the size of the precomputed parity check matrix. On the other hand, Syndrome III (`SYN_INVZA`) computes relevant parts of the parity check matrix on the fly, thus keeping the memory footprint very low, at the expense of speed. Syndrome II (`SYN_EEA`) takes the same approach as Syndrome III, but misses the opportunity to improve the performance by replacing a generic polynomial division by a more specific procedure. Therefore it is advisable to use either Syndrome III, or – if possible – Syndrome I.

A performance evaluation of the three variants is shown in Table 5.10. Syndrome I is dominated by the costs of accessing the parity check matrix from flash memory. Syndrome II executes EEA to compute the polynomial division for every position of the ciphertext that is not zero, which is $^{n}/_{2}$ times on average for an uniformly distributed plaintext. Syndrome III evaluates the Goppa polynomial for every position of the ciphertext that is not zero and performs $3(t-1)$ field operations (addition, multiplication)

Security level	60-bit		79-bit		80-bit		128-bit	
Plaintext length	128 B		204 B		256 B		370 B	
Hash $\mathcal{H}_1$ input	209 B		363 B		456 B		570 B	
Hash $\mathcal{H}_2$ input	128 B		204 B		256 B		370 B	
Operation	Cycles	%	Cycles	%	Cycles	%	Cycles	%
Encryption	2,925,603	16.64	6,430,846	22.69	7,345,516	24.13	12,133,694	17,19
mG	447,379	15.29	838,833	13.04	980,495	13.35	111,647	21.41
Hash $\mathcal{H}_1$	1,580,303	54.02	3,141,044	48.84	3,921,804	53.39	5,481,699	45.18
Hash $\mathcal{H}_2$	797,198	27.25	2,358,724	36.68	2,359,702	32.12	3,920,769	32.31
Other	100,723	3.44	92,245	1.43	83,515	1.14	133,391	1.10
Cycles/Byte	22,856		31,523		28,693		32,793	
Bit/s at 32 Mhz	11,200		8,121		8,922		7,806	
Decryption	14,655,752	83.36	21,916,660	77.31	23,093,531	75,87	58,464,915	82.81
Syndrome	7,283,086	46.69	10,473,908	47.18	10,532,582	45.61	30,510,440	79.36
Patterson	1,393,479	9.51	1,078,481	4.92	747,599	3.24	2,905,102	4.97
Roots & errors	3,092,320	21.10	4,308,441	19.66	4,477,076	19.39	12,959,096	22.17
Hash $\mathcal{H}_1$	1,580,303	10.78	3,141,044	14.33	3,921,804	16.98	5,481,699	9.38
Hash $\mathcal{H}_2$	797,198	5.44	2,358,724	10.76	2,359,702	10.22	3,920,769	6.71
Other	509,366	3.48	908,817	4.15	1,054,768	4.57	2,687,809	4.6
Cycles/Byte	114,498		107,434		90,209		158,013	
Bit/s at 32 Mhz	2,236		2,383		2,838		1,620	

Table 5.9: Performance of McEliece using the Fujisaki-Okamoto conversion

	MCE60	MCE80	MCE128
Syndrome I	590,807	942,940	N/A
Syndrome II	35,102,622	51,627,199	144,137,682
Syndrome III	7,283,086	10,340,083	30,482,209

Table 5.10: Comparison of syndrome computation variants for McEliece

Optimization	MCE80	Gain (%)	MCE128	Gain (%)
None	11,034,662		31,475,563	
$g(z)$ lookup table in Flash memory	8,267,997	25.07	25,032,035	20.47
$g(z)$ lookup table in SRAM	8,265,963	25.09	25,003,795	20.56
GF tables in SRAM	10,870,907	1.48	too big	
Support in SRAM	10,938,429	0.87	31,425,025	0.16
FASTFIELD	10,328,010	6.40	30,581,598	2.85
All*	8,184,422	25.83	24,110,906	23.40

Table 5.11: Cycle count of Syndrome III for MCE128 using various optimizations

that rely on the GF tables.

Since Syndrome III is the most useful variant whenever Syndrome I cannot be applied, we now evaluate several optimizations to Syndrome III. The results are summarized in Table 5.11. The same keys and messages (i.e. a fixed random generator seed) have been used for all measurements to avoid any fluctuation in code cycles.

- **Goppa polynomial lookup table** A precomputed lookup table mapping all possible input values of the Goppa polynomial to their corresponding results can be stored either in Flash memory or in SRAM. The table size is $p^m \times 2$ Byte (8 kB for MCE128), since the result is an `uint16_t` and possible input values are all field elements of $GF(p^m)$. Putting the table in flash memory is a very effective optimization, since it has a significant impact on the performance and the table size is quite small compared to the available flash memory. Hence it should be applied whenever possible.

- **GF tables in SRAM** Instead of accessing the GF tables from flash memory, they can be loaded to SRAM at startup. Both table have n elements á 2 Bytes (12 kB for MCE128).

- **Support in SRAM** The same can be done to the array of support elements, which also holds n elements á 2 Bytes (6 kB for MCE128).

- **Faster field arithmetic** The FASTFIELD switch has already been described in Section 4.3.2. It reduces the number of conversions between polynomial and exponential representation of field elements during arithmetic operations.

- **Inline polynomial evaluation** Using `__inline__ poly_eval` has no effect (i.e. same cycle count and call to `poly_eval` still visible in compiled assembly), probably because of the 'optimize for code size' compilation flag. Hence the function was manually inlined, which can be enabled by setting INLINE_POLY_EVAL=TRUE. Unfortunately this actually has a negative effect on performance.

- **All* optimizations together** All optimizations cannot be used at the same time for the shown parameters because of the limited SRAM. For MCE80, 'All'

denotes FASTFIELD, Goppa polynomial lookup table in SRAM, GF tables in SRAM and support in flash memory. For MCE128, additionally the GF tables had to be left in flash memory.

Of course these optimizations can be used to improve the performance of the whole process, not only of the syndrome computation.

5.2.5 Berlekamp-Massey vs. Patterson

For 80-bit security and syndrome computation I, we have already seen in Section 5.2.1 that decrypting using BM is slightly faster than decrypting using Patterson if applied to the Niederreiter cryptosystem, whereas it is slower if applied to McEliece. From Table 5.5 one can see that the actual decoding step is in fact faster using BM in both cases, i.e. 716,809 instead of 780,043 cycles for McEliece, and 741,172 instead of 854,553 cycles for Niederreiter. However, for decoding all errors using BM, a syndrome of *double* size must be computed before the actual decoding algorithm.

For McEliece, this means computing the syndrome by multiplying the ciphertext c with the double size parity check matrix H_2 computed modulo $g(z)^2$ instead of H computed modulo $g(z)$. For the configuration used in this example, this takes takes 1,702,513 cycles for $c \cdot H_2$, instead of 942,940 cycles for $c \cdot H$. Obviously this difference cannot be compensated by the small performance advantage of BM, hence on the whole the Patterson algorithm is faster.

For Niederreiter, syndrome computation is a part of the encryption process and the ciphertext c is already a syndrome. Since c is a syndrome of standard size, it needs to be transformed to double size to allow the correction of all errors using BM, as described in Section 3.4.3.2. However, this transformation already includes the descrambling of c, which otherwise takes 141,563 cycles in this example measurement. Since the transformation takes only 255,228 cycles, the performance loss is smaller than the gain from using BM instead of Patterson. Hence, for Niederreiter BM is faster than Patterson.

However, all considerations so far are based on 80-bit security and – for McEliece – the fast syndrome computation variant I. For higher security parameters, the syndrome needs to be computed using slower on-the-fly computations. Hence, the comparison of BM and Patterson can be expected to turn out a huge advantage for Patterson if McEliece is used. The results shown in Table 5.12 confirm this assumption.

Moreover, even for Niederreiter the Patterson algorithm turns out to be faster for higher security parameters. Although Berlekamp-Massey remains slightly faster than Patterson for NR128, the performance loss during the syndrome transformation is too big to be compensated.

Hence, using Berlekamp-Massey for binary codes can only be recommended in special cases according to our implementation. Moreover, the memory requirement for BM are higher due to the larger parity check matrix.

Cryptosystem Syndrome	MCE80 Syn I	MCE80 Syn III	MCE128 Syn III	NR80 Transform	NR128 Transform
Decryption (PAT)	6,196,454	15,597,028	44,125,930	5,577,774	16,508,937
Syndrome	942,940	10,364,767	28,715,171	141,563	615,416
Patterson	780,043	779,604	2,892,273	854.553	2,909,632
Decryption (BM)	6,868,866	23,697,815	71,130,775	5,510,006	17,082,953
Syndrome	1,702,513	18,546,210	55,609,478	255,228	1,191,349
BM	716,809	716,078	2,822,964	741,172	2,892,564

Table 5.12: Comparison of Berlekamp-Massey and Patterson

5.2.6 Root extraction

Finding the roots of the error locator polynomial $\sigma(z)$ is often assumed to be the most time consuming step of decryption. Although we have already seen that this is not true for the McEliece cryptosystem if live computation of the syndrome is used, it still makes up a significant part of decryption. However, it is a step that is difficult to optimize, as Heyse states in [Hey10]: "Even an optimized implementation in assembly language reduced the runtime only marginally."

We implemented three algorithms – Bruteforce search using Horner scheme, Chien search and Berlekamp-Trace algorithm (BTA) – for root extraction which have already been described in Section 3.5. As before, we use a fixed random generator seed for the evaluation in order to avoid fluctuations.

Two optimizations can be used to improve the performance at the cost of higher flash memory consumption:

- The `FASTFIELD` switch has already been discussed and applies to all implemented algorithms.
- The `LREVERSE_LOOKUPTABLE` switch computes a reverse lookup table for the permuted secret support, which can be used in Chien search and BTA. For a given support element x the lookup table contains the position of x in the permuted support. Without the table, the support array needs to be searched iteratively until the element has been found.

Table 5.13 comparises the performance of Horner scheme, Chien search and both variations of BTA (see Section 4.6.4) as well as both optimizations. The simple bruteforce search using Horner scheme proves to be the most effective variant, and can be improved by 20.45% using the `FASTFIELD` optimization due to the excessive use of polynomial evaluation. The weaker performance of the Chien search comes with little surprise after the analysis in Section 3.5.2, showing that software implementations do not profit from the structure of the Chien search as hardware implementation do.

BTA clearly suffers from a huge overhead due to the recursion and the expensive handling of large polynomials, both in sparse and full representation. The vast amount of cycles of BTA using a sparse representation is used for the sparse polynomial reduction

	Horner scheme	Chien search	BTA (Full)	BTA (Sparse)	sparsemod[a]
No optimization	15,321,280	26,407,900	159,589,242	309,507,754	98.79 %
FASTFIELD	12,719,902	23,946,730	138,404,862	296,648,695	84.32 %
LREVERSE	N/A	24,706,570	157,887,935	290,179,466	97.93 %
Both (FF + L^{-1})	12,719,902	22,245,412	136,702,805	294,941,173	84.42 %
Gain (FF + L^{-1})	20.45 %	18.71 %	16.74 %	4.94 %	

[a]During one call to BTA (including all recursions), `poly_sparsemod` is called 76.99 times on average. The percentage denotes the percentage of runtime that BTA spends in `poly_sparsemod`.

Table 5.13: Root extraction algorithms and optimizations applied to MCE128

		Table (Flash)	Table (SRAM)	Runtime computation
MCE80+KIC	CW Encoding	15,217	14,057	40,187
CWBYTES=25	CW Decoding	19,276	18,439	44,724
MCE128+KIC	CW Encoding	77,643	75,055	128,427
CWBYTES=49	CW Decoding	34,958	33,202	87,929

Table 5.14: Comparison of cycle counts for constant weight encoding in different modes

modulo $sigma(z)$. While this saves a considerable amount of SRAM, it almost doubles the – already unacceptable long – runtime. Accordingly, BTA cannot be recommended for our set of parameters. It is however possible that it proves to be an appropriate choice for (far) larger codes. Moreover, an implementation of BTA including the Zinoviev procedures may provide better results. However, it would need to reduce the number of recursive calls to below 10% of the current calls in order to achieve the same performance as the bruteforce search using Horner scheme.

5.2.7 Constant weight encoding

All previous measurements utilizing constant weight encoding used the `bestU` lookup table described in Section 4.7. It can be accessed either from flash memory or from SRAM. Since the impact of constant weight encoding turned out to be mostly insignificant, using precious SRAM is not recommended. The third option is the computation of $(n - (t-1)/2)/t$ and a very small mapping as an approximation of u at runtime. Note that this approximation is too inaccurate to reliably encode the same number of bytes into the constant weight word.

The locally optimal value of u is computed at every step of encoding and decoding. Table 5.14 compares the resulting number of clock cycles for the three discussed variants. As expected, accessing the table from SRAM is slightly faster than from flash memory, and the runtimecomputation is slowest. Depending on the parameters, decoding or encoding may be faster than the respective operation.

5.2.8 Optimal configuration

To summarize the results discussed in the previous sections, we now evaluate an 'optimal configuration' including all optimizations that turned out to be useful. We then compute the data throughput and compare it to the performance of conventional cryptosystems.

Although the Berlekamp-Massey algorithm was shown to be slightly faster than Patterson for NR80, using all optimizations the Patterson algorithm turned out to be slightly faster even for NR80. For all other cases, Patterson is faster anyway. Comparing the conversions by Fujisaki-Okamoto and Kobaira-Imai, the latter turned out to be faster. For root extraction, the bruteforce search using Horner scheme is fastest, hence the `LREVERSE_LOOKUPTABLE` switch becomes irrelevant for the final evaluation. For syndrome computation in McEliece either Syndrome I (MCE80) or Syndrome III (MCE128) is used. If Syndrome III is used, the Goppa polynomial lookup table is used to speed up the computation. Moreover, the `FASTFIELD` switch is used in all cases. Tables are loaded into SRAM where possible.

The results for McEliece and Niederreiter at an 80-bit and 128-bit security level are presented in Table 5.15 and compared to an unoptimized run using the same keys and messages. Note that `bestU` table has been used for both the optimized and the unoptimized run, since otherwise `CWBYTES` would need to be adjusted. The major part of the decryption performance gain stems from the `FASTFIELD` switch and – in the case of McEliece – from the Goppa polynomial lookup table. Note that the impact of `FASTFIELD` is reduced by accessing the GF tables from SRAM, as done for MCE80. The big difference between MCE80 and MCE128 stems from the usage of Syndrome I for MCE80 and the on-the-fly computation for MCE128.

Encryption it not affected by any optimization and is dominated by flash memory access, XOR operations and the Hash function computation. Accessing the `bestU` table from flash improves constant weight encoding performance, but would have no significant affect on encryption performance.

5.2.9 Comparison with other implementations of Code-based and traditional cryptosystems

According to Table 5.1 RSA-1248 respectively RSA-3248 provides equivalent security to MCE80 respectively MCE 128. To the best of our knowledge, no RSA implementation for these parameters is available on a comparable 8-bit processor. However, an performance evaluation of RSA-1024 and RSA-2048 as well as ECC-160, ECC-192 and ECC-224 on an Atmel ATmega128 is available in [GPW+04]. ECC-160 corresponds to 80-bit security, wheres 128-bit security would require ECC-256. A more recent paper [LGK10] claims to perform a full RSA-1024 private-key operation in less than $76 \cdot 10^6$ cycles, also on an ATmega128.

Performance evaluations of McEliece and Niederreiter are available in [EGHP09, Hey10], as well as an evaluation of McEliece based on Quasi-Dyadic Goppa codes in [Pau10].

Table 5.16 compares the results of these implementations to the performance of our implementation of McEliece and Niederreiter. Frequencies differing from 32 Mhz (marked

McEliece	MCE80 (optimized)		MCE80		MCE128 (optimized)		MCE128	
Syndrome	I		I		III		III	
FASTFIELD	Yes		No		Yes		No	
GF tables	SRAM (8 kB)		Flash (8 kB)		Flash (16 kB)		Flash (16 kB)	
Support	SRAM (4 kB)		Flash (4 kB)		Flash (6 kB)		Flash (6 kB)	
$g(z)$ table	Flash (4 kB)		No		SRAM (8 kB)		Flash (8 kB)	
bestU table	Flash (2 kB)		Flash (2 kB)		Flash (3 kB)		Flash (3 kB)	
Plaintext	212 B		212 B		303 B		303 B	
Operation	Cycles	%	Cycles	%	Cycles	%	Cycles	%
Encryption	2,644,139	25.67	2,644,297	22.82	5,277,682	11.05	5,278,044	9.07
CW encode	15,207	0.58	15,216	0.58	77,610	1.47	77,695	1.47
Encrypt	997,067	37.71	997,131	37.71	2,679,326	50.77	2,679,439	50.77
Hash	1,608,143	60.82	1,608,160	60.82	2,388,771	45.26	2,388,771	45.26
Decryption	7,655,240	74.33	8,944,729	77.18	42,500,066	88.95	52,901,084	90.93
Syndrome	931,060	12.16	931,047	10.41	24,040,858	56.57	31,349,307	59.26
Patterson	734,672	9.60	851,024	9.51	2,904,985	6.84	3,346,153	6.33
Roots	4,194,771	54.80	5,367,754	60.01	12,956,761	30.49	15,608,332	29.50
Hash	1,608,143	21.01	1,608,160	17.98	2,388,771	5.62	2,388,771	4.52
CW decode	19,720	0.25	19,269	0.22	35,094	0.08	35,081	0.07
Total	10,299,379		11,589,026		47,777,748		58,179,128	
at 32 Mhz	C/Byte	Bit/s	C/Byte	Bit/s	C/Byte	Bit/s	C/Byte	Bit/s
Encryption	12,472	20,525	12,473	20,524	17,418	14,697	17,419	14,696
Decryption	36,110	7,090	42,192	6,067	140,264	1,825	174,591	1,466
Opt. Gain	Encryption: 0.01 %		Decryption: 16.84 %		Encryption: 0.01 %		Decryption: 24.47 %	

Niederreiter	NR80 (optimized)		NR80		NR128 (optimized)		NR128	
FASTFIELD	Yes		No		Yes		No	
GF tables	SRAM (8 kB)		Flash (8 kB)		Flash (12 kB)		Flash (12 kB)	
Support	SRAM (4 kB)		Flash (4 kB)		SRAM (6 kB)		Flash (6 kB)	
bestU table	Flash (2 kB)		Flash (2 kB)		Flash (3 kB)		Flash (3 kB)	
Plaintext	212 B		212 B		303 B		303 B	
Operation	Cycles	%	Cycles	%	Cycles	%	Cycles	%
Encryption	1,674,111	19.91	1,674,148	17.26	2,549,586	11.88	2,549,586	10.38
CW encode	15,329	0.92	15,327	0.92	27,387	1.07	27,387	1.07
Encrypt	31,279	1.87	31,278	1.87	111,714	4.38	111,714	4.38
Hash	1,573,701	94.00	1,573,701	94.00	2,352,512	92.27	2,352,512	92.27
Decryption	6,736,313	80.09	8,025,436	82.74	18,915,769	88.12	22,023,183	89.62
Descramble	142,143	2.11	141,873	1.77	620,331	3.28	620,316	2.82
Patterson	749,846	11.13	866,120	10.79	2,903,535	15.35	3,344,602	15.19
Roots	4,186,678	62.15	5,359,630	66.78	12,929,114	68.35	15,595,446	70.81
Hash	1,573,701	23.36	1,573,701	19.61	2,352,512	12.44	2,352,512	10.68
CW decode	20,084	0.30	20,070	0.25	35,633	0.19	35,633	0.16
Total	8,410,424		9,699,583		21,465,355		24,572,769	
at 32 Mhz	C/Byte	Bit/s	C/Byte	Bit/s	C/Byte	Bit/s	C/Byte	Bit/s
Encryption	7,897	32,418	7,897	32,418	8,414	30,424	8,414	30,424
Decryption	31,775	8,057	37,856	6,762	62,428	4,101	72,684	3,522
Opt. Gain	Encryption: 0.00 %		Decryption: 19.14 %		Encryption: 0.00 %		Decryption: 16.43 %	

Table 5.15: Optimized performance of McEliece and Niederreiter using Patterson decoder, Kobara-Imai-γ conversion and Horner scheme

with *) were scaled accordingly to allow a fair comparison. One can see that our implementation outperforms all other implementations: it is faster than the previous implementations of McEliece and Niederreiter, as well as comparable implementations of RSA and ECC. The difference between our and the previous Niederreiter implementation is marginal, whereas a huge improvement could be achieved over previous McEliece implementations. This is due to the fact that the previous implementations used either non-systematic key matrices or Quasi-Dyadic Goppa codes. Note, however, that Quasi-Dyadic Goppa codes have the important advantage of providing a very compact key representation, hence drastically reducing the memory requirements.

Both our Niederreiter and our McEliece implementation are faster than comparable RSA- and ECC implementations even with the CCA2-secure Kobara-Imai-γ conversion applied. This is due to the fact that KIC effectively turns McEliece and Niederreiter into a stream cipher, where Code-based public-key encryption is used only to encrypt a seed, whereas the message encryption uses fast XOR operations.

System	Cycle count	Throughput (bit/s)
80-bit McEliece using non-systematic key matrices on ATxMega192 @ 32MHz [EGHP09]		
Encryption	14,406,080	3,889
Decryption	19,751,094	2,835
80-bit Niederreiter on ATxMega192 @ 32MHz [Hey10]		
80-bit Encryption	51,247	119,890
80-bit Decryption	5,750,144	1,062
80-bit Quasi-Dyadic McEliece and KIC on ATxmega256A1 @ 32MHz [Hey11]		
Encryption	6,358,400	6,482
Decryption	33,536,000	1,229
Decryption (Syndrome on-the-fly)	50,163,200	822
RSA on ATMega128 @ 8MHz [GPW+04]		
RSA-1024 public-key $e = 2^{16} + 1$	~3,440,000	9,526 *
RSA-1024 private-key with Chinese Remainder Theorem (CRT)	~87,920,000	373 *
RSA-2048 public-key $e = 2^{16} + 1$	~15,520,000	4223 *
RSA-2048 private-key with CRT	~666,080,000	98 *
SECG-standardized ECC on ATMega128 @ 8MHz [GPW+04]		
ECC-160	~6,480,000	790 *
ECC-192	~9,920,000	619 *
ECC-224	~17,520,000	409 *
RSA on ATMega128 [LGK10]		
RSA-1024	~76,000,000	431
Our implementation on ATxmega256A1 @ 32MHz		
80-bit McEliece Encryption	994,056	56,367
80-bit McEliece Decryption	6,196,454	9,043
80-bit Niederreiter Encryption	46,734	138,999
80-bit Niederreiter Decryption	5,510,006	1,165
Our McEliece implementation including KIC on ATxmega256A1 @ 32MHz		
80-bit Encryption	2,644,139	20,525
80-bit Decryption	7,655,240	7,090
128-bit Encryption	5,277,682	14,697
128-bit Decryption	42,500,066	1,825
Our Niederreiter implementation including KIC on ATxmega256A1 @ 32MHz		
80-bit Encryption	1,674,111	32,418
80-bit Decryption	6,736,313	8,057
128-bit Encryption	2,549,586	30,424
128-bit Decryption	18,915,769	4,101

Table 5.16: Comparison of performance of our implementation and comparable implementations of McEliece, Niederreiter, RSA and ECC

6 Conclusion

This section summarizes the results of this thesis and proposes ideas for future extensions of the presented Code-based Cryptography Library.

6.1 Summary

In this thesis, we presented an implementation of a broad range of methods and techniques from Code-based cryptography, tailored to the constricted execution environment of embedded devices such as the 8-bit microcontroller AVR ATxmega256A3. Our library includes implementations of both the McEliece and Niederreiter cryptosystem and extends previous implementations providing only 80-but security to the more suitable security level of 128-bit security. Higher security levels are possible and mainly[1] limited by the amount of available memory. For example, instances providing 256-bit security have been tested successfully and would also run on AVR microcontrollers that provide enough memory (approximately 1 MB is required for encryption).

The substitution of the 'classical' McEliece and Niederreiter cryptosystems by a security-equivalent modern variant using systematic key matrices proved to be a valuable choice for reducing the high memory requirements and additionally help in improving the performance of the system.

Our library includes two CCA2-secure conversions, which are strictly required for virtually any practical application of McEliece and Niederreiter. We showed that the Kobara-Imai-γ conversion achieves a high data throughput and discussed under which conditions the Fujisaki-Okamoto conversion could provide an alternative to the Kobara-Imai conversion.

We implemented to two different decoding algorithms. The Patterson algorithm can be applied only to binary Goppa codes, but turned out to be very efficient. On the other hand, the Berlekamp-Massey-Sugiyama algorithm can be applied to general alternant codes and can be implemented in a very compact form. We demonstranted how Berlekamp-Massey can be tuned to achieve the same error-correction capacity as the Patterson algorithm for binary codes and implemented the additional steps necessary to apply it to the Niederreiter cryptosystem.

Finding the roots of the error locator polynomial and the computation of the syndrome in the McEliece cryptosystem with limited memory ressources turned out to be the computationally most expensive steps of decryption. Therefore we implemented and optimized three variants of root extraction and three methods of syndrome computation.

[1] It is also limited by the currently utilized 16-bit data types. For example, `gf_t` cannot hold elements of the field $\mathcal{F}_{2^m}$ for $m > 15$ without changes.

Depending on the parameters, a performance gain between 15% and 25% has been achieved.

An extensive evaluation has been carried out to analyze the performance of the implementation variants and optimizations. The flexible configuration of our Code-based cryptography library offers the chance to find an individually optimal balance between memory usage and performance. Several computations can optionally be speed up using precomputations and lookup tables, which can be accessed either from the fast SRAM or the slower flash memory according to the users' needs.

Our implementation shows that Code-based cryptosystems providing security levels fulfilling real-world requirements can be executed on microcontrollers with more than satisfying performance: it actually outperforms comparable implementations of conventional cryptosystems in terms of data throughput. This provides further evidence that McEliece and Niederreiter can evolve to a fully adequate replacement for traditional cryptosystems such as RSA. Hence, we continue to believe that intensifying the research on Code-based cryptography is an important step to overcome the dangerous dependence of today's cryptosystems on the difficulty of the closely related problems of Integer Factorization and Discrete Logarithm. McEliece and Niederreiter remain promising candidates for providing security in the post-quantum world, as well as for advancing the diversification of public-key cryptography.

6.2 Future Work

Several extensions and improvements to our library are possible, most notably the extension to non-binary codes. Some hints about changes necessary to allow the use of non-binary codes in our implementation have already been given in Section 4. Note that the Berlekamp-Massey algorithm is able to decode non-binary codes without further changes, since it already returns both an error locator polynomial and error value polynomial.

Moreover, the feasibility of performing key generation on the device could be evaluated.

The integration of quasi-dyadic Goppa codes into the library represents an interesting approach at reducing the memory requirements of the implementation. However, it remains unknown whether quasi-dyadic Goppa codes provide the same security as Goppa codes.

To deal with larger Galois fields, tower fields could be integrated in the library. Using tower fields, a field $\mathcal{F}_{p^{2k}}$ can be represented as a field extension over $\mathcal{F}_{p^k}$. This would allow us to perform all computations in a smaller field, requiring smaller lookup tables for field arithmetic. Note, however, that tower field arithmetic comes with a significant performance loss.

As discussed in the previous chapter, the inclusion of the additional parameter y_5 into the McEliece version of the Kobara-Imai-γ conversion could improve the data throughput of McEliece. Furthermore, the employment of a lightweight Hash function – respectively the usage of lightweight parameters for Keccak – should be evaluated to speed up both CCA2-secure conversion, in particular the Fujisaki-Okamoto conversion that uses the

Hash function twice in both encryption and decryption.

To optimize the performance of our implementation, expensive and frequently used functions such as root extraction and syndrome computation could be implemented in Assembly language. Furthermore, including Zinoviev's procedures in the Berlekamp-Trace algorithm could help in finding an efficient alternative to the root extraction using a bruteforce search with Horner scheme.

Finally, an issue concerning constant weight encoding still needs to be resolved, which causes a certain fraction of decoding operations to fail for some configurations.

A Acronyms

HyMES Hybrid McEliece Encryption Scheme

NIST National Institute of Standards and Technology

MCE McEliece cryptosystem

NR Niederreiter cryptosystem

BM Berlekamp-Massey

PAT Patterson algorithm

ELP error locator polynomial

EVP error value polynomial

BTA Berlekamp-Trace algorithm

BTZ Berlekamp-Trace algorithm using Zinovievs algorithms

HOR Horner scheme

CHI Chien search

KIC Kobara-Imai-γ conversion

FOC Fujisaki-Okamoto conversion

GRS Generalized Reed-Solomon Code

systematic Matrix in systematic form: $M = (I_k|Q)$ where I_k is a $k \times k$ identity matrix

SSA Support Splitting Algorithm

BCH Bose, Ray-Chaudhur and Hocquenghem

MDS Maximum Distance Separable

IDS Information Set Decoding

CW Constant Weight

GF Galois Field

GCD Greatest Common Divisor

EEA Extended Euclidean Algorithm

IND-CCA2 Indistinguishability under Adaptive Chosen Ciphertext Attacks

CCA2-secure see IND-CCA2

ECC Elliptic Curve Cryptography

OAEP Optimal Asymmetric Encryption Padding

LFSR linear feedback shift register

RISC Reduced instruction set computing

UART Universal Asynchronous Receiver/Transmitter

PRNG Pseudo-Random Number Generator

GPL GNU General Public License

LGPL GNU Lesser General Public License

B Appendix

B.1 Listings

B.1.1 Listing primitive polynomials for the construction of Finite fields

The open source mathematical software SAGE[1] can be used to print a list of primitive polynomials, which are required for the construction of a finite field $\mathcal{F}_{p^m}$.

```
p=2; m=1; mmax=32;
while m <= mmax:
    F.<z> = FiniteField(p^m)
    print "GF(%d^%d)" % (p,m),
    print F.polynomial()
    m+=1
```

Listing B.1: Listing primitive polynomials using SAGE

For `p=2,m=11,mmax=11` this function outputs `GF(2^11) z^11 + z^2 + 1`. Rewriting this polynomial as $1 \cdot z^{11} + \ldots 0 \cdots + 1 \cdot z^2 + 0 \cdot z^1 + 1 \cdot z^0$, we find the representation $100000000101_2 = 2053_{10}$. Hence, for each Finite field used in the implementation, we provide a definition like

```
#if GF_m == 11
 #define PRIM_POLY 2053
#endif
```

Listing B.2: Primitive polynomial definition for field construction

B.1.2 Computing a normal basis of a Finite Field using SAGE

SAGE can also help to compute a normal basis of a field $\mathcal{F}_{2^m}$.

```
def normalbase(M):
    F.<a> = GF(2^M)
    for e in range(0, 2^M):
        z=a^e
        basis = list();
        for i in range(M):
            s = bin(eval((z^(2^i)).int_repr()))[2:]
            basis.append(list(map(int,list('0'*(M-len(s))+s))))

        x = span(basis,GF(2)).matrix()
        if x.nrows() == M and x.determinant() != 0:
            return e
    return -1

```

[1] http://www.sagemath.org/

```
15 for m in range(2,16):
16     F.<a> = GF(2^m)
17     print "GF(%d^%d)" % (2,m),
18     basis = normalbase(m)
19     print a^basis, "=", (a^basis).int_repr()
20     print
```

Listing B.3: Computing a normal basis using SAGE

The code is adapted from [Ris] and outputs the first elemenet of a normal basis like `GF(2^11) a^9 = 512`, which is used to provide a definition for each used Finite field.

```
1 #if GF_m == 11
2  #define NORMAL_BASIS 512
3 #endif
```

Listing B.4: Normal basis definition for $GF(2^{11})$

B.2 Definitions

B.2.1 Hamming weight and Hamming distance

The *Hamming distance* between two words x and y is defined as the number of symbols (e.g. bits for binary strings) in which x and y differ. The *Hamming weight* $wt(x)$ is the number of non-zero symbols of x.

B.2.2 Minimum distance of a codeword

The minimum distance $d = d_{min}(\mathcal{C})$ of a linear code $\mathcal{C}$ is the smallest *Hamming distance* between distinct codewords. The code is then called a $[n, k, d]$-code.

B.2.3 One-way functions

Definitions in this section stem from Pointcheval [Poi00] and apply to polynomial time adversaries, i.e. an adversary $\mathcal{A}$ using an algorithm with a running time bounded in the algorithm's input size n by $\mathcal{O}(n^k)$ for some constant k.

One-way functions A function $f(x) = y$ is *one-way* if for any input x the output y can be computed efficiently, but it is computationally infeasible to compute x given only y. More formally, the success probability $P(f(\mathcal{A}(f(x)) = f(x))$ is negligible. If f is a permutation, it is also called *one-way permutation*.

Trapdoor functions A one-way function $f(x) = y$ is called a *one-way trapdoor function* if x can be efficiently computed from y if and only if some additional information s is known. This kind of function can be used to construct public-key cryptosystems, where s forms the secret key. If f is a permutation, it is also called *one-way trapdoor permutation*.

Partially Trapdoor functions If a trapdoor one-way function does not allow a complete inversion, but just a partial one, it is called a *partially trapdoor* one-way function. More formally, a one-way function $f(x_1, x_2) = y$ with secret s is a partially trapdoor function if given y and s, it is possible to compute a x_1 such that there exists an x_2 that satisfies $f(x_1, x_2) = y$.

B.2.4 Cryptographic Hash functions

A hash function is an algorithm mapping data sets of arbitrary length deterministically to smaller data sets of fixed length. An ideal cryptographic hash function ensures that computing the hash value is easy for any input, whereas it is infeasible

- to find a message that maps to a given hash value
- to find two different messages mapping to the same hash
- to modify a message without changing its hash value

B.2.5 One-time pad

A one-time pad (OTP) is a type of symmetric encryption which requires the key to have the same (or greater length) as the message. It encrypts using modular addition of the message and the key and decrypts by adding the same key to the ciphertext. If the key is truly random and not reused, it is provably secure, which means that a bruteforce search through the entire key space is the fastest attack possible.

List of Figures

List of Tables

List of Listings

Bibliography

[Arn10] Eric Arnoult. Implementation of a q-ary version of the niederreiter cryptosystem, August 2010. `http://www.cayrel.net/research/code-based-cryptography/code-based-cryptosystems/`.

[BAO09] Maria Bras-Amorós and Michael E. O'Sullivan. The Berlekamp-Massey Algorithm and the Euclidean Algorithm: a Closer Link. *CoRR*, abs/0908.2198, 2009.

[BBC+11] Marco Baldi, Marco Bianchi, Franco Chiaraluce, Joachim Rosenthal, and Davide Schipani. Enhanced public key security for the McEliece cryptosystem. *CoRR*, abs/1108.2462, 2011.

[BCGO09] Thierry P. Berger, Pierre-Louis Cayrel, Philippe Gaborit, and Ayoub Otmani. Reducing Key Length of the McEliece Cryptosystem. In *Proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology*, AFRICACRYPT '09, pages 77–97, Berlin, Heidelberg, 2009. Springer-Verlag.

[BCM+12] Zhengbing Bian, Fabian Chudak, William G. Macready, Lane Clark, and Frank Gaitan. Experimental determination of Ramsey numbers with quantum annealing. January 2012.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference, 2011.

[Be] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/`.

[Ber68] E. Berlekamp. Nonbinary BCH decoding. *Information Theory, IEEE Transactions on*, 14(2):242, march 1968.

[Ber71] E. R. Berlekamp. Factoring polynomials over large finite field. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, SYMSAC '71, pages 223–, New York, NY, USA, 1971. ACM.

[Ber72] E. R. Berlekamp. A Survey of Coding Theory. *Journal of the Royal Statistical Society. Series A (General)*, 135(1), 1972.

[Ber73] Elwyn Berlekamp. Goppa Codes. *IEEE Transactions on Information Theory*, IT-19(5), 1973.

[Ber09] Daniel J. Bernstein. *Introduction to post-quantum cryptography.* 2009.

[Ber10] Daniel J. Bernstein. Grover vs. McEliece. In Nicolas Sendrier, editor, *PQCrypto*, volume 6061 of *Lecture Notes in Computer Science*, pages 73–80. Springer, 2010.

[Ber11] Daniel J. Bernstein. List decoding for binary goppa codes. In *Proceedings of the Third international conference on Coding and cryptology*, IWCC'11, pages 62–80, Berlin, Heidelberg, 2011. Springer-Verlag.

[BH09] Bhaskar Biswas and Vincent Herbert. Efficient Root Finding of Polynomials over Fields of Characteristic 2. In *WEWoRC 2009*, LNCS. Springer-Verlag, 2009.

[BJMM12] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. Decoding Random Binary Linear Codes in $2^{n/20}$: How $1 + 1 = 0$ Improves Information Set Decoding. *IACR Cryptology ePrint Archive*, 2012:26, 2012.

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the McEliece cryptosystem. Cryptology ePrint Archive, Report 2008/318, 2008.

[BLP11] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Wild mceliece. In *Proceedings of the 17th international conference on Selected areas in cryptography*, SAC'10, pages 143–158, Berlin, Heidelberg, 2011. Springer-Verlag.

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems. *Information Theory, IEEE Transactions on*, 24(3):384 – 386, may 1978.

[Bou07] Iliya G. Bouyukliev. *About the code equivalence.*, pages 126–151. Hackensack, NJ: World Scientific, 2007.

[BPV92] R. Govaerts B. Preneel, A. Bosselaers and J. Vandewalle. A software implementation of the McEliece public-key cryptosystem. In *Proceedings 13th Symposium on Information Theory in the Benelux*, pages 119–126, 1992.

[BS99] Mihir Bellare and Amit Sahai. Non-malleable Encryption: Equivalence between Two Notions, and an Indistinguishability-Based Characterization. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 519–536. Springer, 1999.

[BS08a] Bhaskar Biswas and Nicolas Sendrier. The hybrid mceliece encription scheme, May 2008. `https://www.rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`.

[BS08b] Bhaskar Biswas and Nicolas Sendrier. Mceliece cryptosystem implementation: Theory and practice. In Johannes Buchmann and Jintai Ding, editors,

PQCrypto, volume 5299 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2008.

[Can98] Anne Canteaut. A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44:367–378, 1998.

[Cay] Pierre-Louis Cayrel. Code-based cryptosystems: implementations. `http://www.cayrel.net/research/code-based-cryptography/code-based-cryptosystems/`.

[CC95] Anne Canteaut and Florent Chabaud. Improvements of the Attacks on Cryptosystems Based on Error-Correcting Codes, 1995.

[Chi06] R. Chien. Cyclic decoding procedures for bose- chaudhuri-hocquenghem codes. *IEEE Trans. Inf. Theor.*, 10(4):357–363, September 2006.

[CHP12] Pierre-Louis Cayrel, Gerhard Hoffmann, and Edoardo Persichetti. 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, pages 138–155, Berlin, Heidelberg, 2012. Springer-Verlag.

[cod12] codecrypt, November 2012. `https://github.com/exaexa/codecrypt`.

[Cov06] T. Cover. Enumerative source encoding. *IEEE Trans. Inf. Theor.*, 19(1):73–77, September 2006.

[Dor87] Jean-Louis Dornstetter. On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE Transactions on Information Theory*, 33(3):428–431, 1987.

[EGHP09] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. MicroEliece: McEliece for Embedded Devices. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2009.

[EOS06] Daniela Engelbert, Raphael Overbeck, and Arthur Schmidt. A Summary of McEliece-Type Cryptosystems and their Security. *IACR Cryptology ePrint Archive*, 2006:162, 2006.

[Eur12] European Network of Excellence in Cryptology II. ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012), 9 2012. `http://www.ecrypt.eu.org/documents/D.SPA.20.pdf`.

[FKI07] M. P. C. Fossorier, K. Kobara, and H. Imai. Modeling Bit Flipping Decoding Based on Nonorthogonal Check Sums With Application to Iterative Decoding Attack of McEliece Cryptosystem. *IEEE Transactions on Information Theory*, 53(1):402 –411, jan. 2007.

[fle] Flexiprovider: PQCProvider. Theoretical Computer Science Research Group of Prof. Dr. Johannes Buchmann at the Departement of Computer Science at Technische Universität Darmstadt, Germany. `http://www.flexiprovider.de/`.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes, 1999.

[For65] Jr. Forney, G. On decoding BCH codes. *Information Theory, IEEE Transactions on*, 11(4):549 – 557, oct 1965.

[FS09] Matthieu Finiasz and Nicolas Sendrier. Security Bounds for the Design of Code-Based Cryptosystems. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 88–105, Berlin, Heidelberg, 2009. Springer-Verlag.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, January 1979.

[Gop69] V.D. Goppa. A New Class of Linear Correcting Codes. *Probl. Peredachi Inf.*, 6(3):24–30, 1969.

[GPW+04] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *CHES*, pages 119–132, 2004.

[GPZ60] Daniel Gorenstein, W. Wesley Peterson, and Neal Zierler. Two-Error Correcting Bose-Chaudhuri Codes are Quasi-Perfect. *Inf. Comput.*, 3(3):291–294, September 1960.

[Hey08] Stefan Heyse. Efficient Implementation of the McEliece Crypto System for Embedded Systems, October 2008.

[Hey09] Stefan Heyse. Code-based Cryptography: Implementing the McEliece Scheme on Reconfigurable Hardware. Diploma thesis, Ruhr-Universität-Bochum, May 2009. `http://www.emsec.rub.de/media/crypto/attachments/files/2010/04/da_heyse.pdf`.

[Hey10] Stefan Heyse. Low-Reiter: Niederreiter Encryption Scheme for Embedded Microcontrollers. In Nicolas Sendrier, editor, *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May*

25-28, 2010. Proceedings, volume 6061 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2010.

[Hey11] Stefan Heyse. Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. In *PQCrypto*, pages 143–162, 2011.

[HG13] Stefan Heyse and Tim Güneysu. Code-Based Cryptography on Reconfigurable Hardware: Tweaking Niederreiter Encryption for High-Performance. *IACR Journal of Cryptographic Engineering*, Special Issue for CHES 2012, 2013. To be published.

[HMP10] Stefan Heyse, Amir Moradi, and Christof Paar. Practical power analysis attacks on software implementations of mceliece. In Nicolas Sendrier, editor, *PQCrypto*, volume 6061 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2010.

[Hof11] Gerhard Hoffmann. Implementation of McEliece using quasi-dyadic Goppa Codes. Bachelor thesis, TU Darmstadt, Apr 2011. `www.cdc.informatik.tu-darmstadt.de/reports/reports/Gerhard_Hoffmann.bachelor.pdf`.

[HP03] Cary W. Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, August 2003.

[Hub] Klaus Huber. Algebraische Codierung für die sichere Datenübertragung. `http://www.emsec.rub.de/imperia/md/content/lectures/algebraische_codierung_huber.pdf`.

[Hub96] K. Huber. Note on decoding binary Goppa codes. *Electronics Letters*, 32(2):102 –103, jan 1996.

[IBM12] IBM. IBM Quantum Computing Press Release, February 2012. `http://ibmquantumcomputing.tumblr.com/`.

[Jab01] A. Kh. Al Jabri. A Statistical Decoding Algorithm for General Linear Block Codes. In *Proceedings of the 8th IMA International Conference on Cryptography and Coding*, pages 1–8, London, UK, UK, 2001. Springer-Verlag.

[JAG+11] M. W. Johnson, M. H. S. Amin, S. Gildert, T. Lanting, F. Hamze, N. Dickson, R. Harris, A. J. Berkley, J. Johansson, P. Bunyk, E. M. Chapple, C. Enderud, J. P. Hilton, K. Karimi, E. Ladizinsky, N. Ladizinsky, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, C. J. S. Truncik, S. Uchaikin, J. Wang, B. Wilson, and G. Rose. Quantum annealing with manufactured spins. *Nature*, 473(7346):194–198, May 2011. `http://www.nature.com/nature/journal/v473/n7346/full/nature10012.html`.

[KI01] Kazukuni Kobara and Hideki Imai. Semantically Secure McEliece Public-Key Cryptosystems-Conversions for McEliece PKC. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography:*

Public Key Cryptography, PKC '01, pages 19–35, London, UK, UK, 2001. Springer-Verlag.

[KL07] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[LB88] P. J. Lee and E. F. Brickell. An observation on the security of mceliece's public-key cryptosystem. In *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT'88*, pages 275–280, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[LDW06] Yuan Xing Li, R. H. Deng, and Xin Mei Wang. On the equivalence of McEliece's and Niederreiter's public-key cryptosystems. *IEEE Trans. Inf. Theor.*, 40(1):271–273, September 2006.

[Lee07] Kwankyu Lee. Interpolation-based Decoding of Alternant Codes. *CoRR*, abs/cs/0702118, 2007.

[LGK10] Zhe Liu, Johann Großschädl, and Ilya Kizhvatov. Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers. In *Proceedings of the 1st Workshop on the Security of the Internet of Things (SECIOT 2010)*, pages 00–00. IEEE Computer Society, 2010.

[LS98] P. Loidrean and N. Sendrier. Some weak keys in McEliece public-key cryptosystem. In *Information Theory, 1998. Proceedings. 1998 IEEE International Symposium on*, page 382, aug 1998.

[LS01] P. Loidreau and N. Sendrier. Weak keys in the McEliece public-key cryptosystem. *Information Theory, IEEE Transactions on*, 47(3):1207 –1211, mar 2001.

[Mas69] James L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15:122–127, 1969.

[MB09] Rafael Misoczki and Paulo S. Barreto. Selected areas in cryptography. chapter Compact McEliece Keys from Goppa Codes, pages 376–392. Springer-Verlag, Berlin, Heidelberg, 2009.

[McE78] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.

[Min07] Lorenz Minder. *Cryptography Based on Error Correcting Codes*. PhD thesis, Ècole Polytechnique Fédérale de Lausanne, July 2007.

[MMT11] Alexander May, Alexander Meurer, and Enrico Thomae. Decoding random linear codes in $\mathcal{O}(2^{0.054n})$. In *ASIACRYPT*, pages 107–124, 2011.

[MS78] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North-holland Publishing Company, 2nd edition, 1978.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[NC11] Robert Niebuhr and Pierre-Louis Cayrel. Broadcast Attacks against Code-Based Schemes. In Frederik Armknecht and Stefan Lucks, editors, *WEWoRC*, volume 7242 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

[Nie86] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems Control Inform. Theory/Problemy Upravlen. Teor. Inform.*, 15(2):159–166, 1986.

[Nie12] Robert Niebuhr. *Attacking and Defending Code-based Cryptosystems*. PhD thesis, Technische Universität Darmstadt, 2012.

[Nit] Abderrahmane Nitaj. Quantum and post quantum cryptography. `http://www.math.unicaen.fr/~nitaj/postquant.pdf`.

[NMBB12] Robert Niebuhr, Mohammed Meziani, Stanislav Bulygin, and Johannes Buchmann. Selecting Parameters for Secure McEliece-based Cryptosystems. *International Journal of Information Security*, 11(3):137–147, Jun 2012.

[OS09] Raphael Overbeck and Nicolas Sendrier. Code-based cryptography. Bernstein, Daniel J. (ed.) et al., Post-quantum cryptography. First international workshop PQCrypto 2006, Leuven, The Netherland, May 23–26, 2006. Selected papers. Berlin: Springer. 95-145 (2009)., 2009.

[OTD10] Ayoub Otmani, Jean-Pierre Tillich, and Léonard Dallot. Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes. *Mathematics in Computer Science*, 3(2):129–140, 2010.

[Ove07] Raphael Overbeck. *Public Key Cryptography based on Coding Theory*. Doctoral thesis, Technische Universität Darmstadt, 2007.

[Pat75] N. Patterson. The algebraic decoding of Goppa codes. *Information Theory, IEEE Transactions on*, 21(2):203 – 207, mar 1975.

[Pau10] Olga Paustjan. Post quantum cryptography on embedded devices: An efficient implementation of the mceliece key scheme based on quasi-dyadic goppa codes. Diploma thesis, Ruhr-Universität Bochum, July 2010. `https://www.emsec.rub.de/media/attachments/files/2012/10/post_quantum.pdf`.

[Per12] Edoardo Persichetti. Compact mceliece keys based on quasi-dyadic srivastava codes. *J. Mathematical Cryptology*, 6(2):149–169, 2012.

[Pet60] W. Peterson. Encoding and error-correction procedures for the Bose-Chaudhuri codes. *Information Theory, IRE Transactions on*, 6(4):459 –470, september 1960.

[Poi00] David Pointcheval. Chosen-Ciphertext Security for any One-Way Cryptosystem. In Hideki Imai and Yuliang Zheng, editors, *Workshop on Practice and Theory in Public-Key Cryptography (PKC '00)*, volume 1751 of *Lecture Notes in Computer Science*, pages 129–146, Melbourne, Australia, 2000. Springer.

[PR97] Erez Petrank and Ron M. Roth. Is Code Equivalence Easy to Decide? *IEEE Transactions on Information Theory*, 43:1602–1604, 1997.

[pro] prometheus. `http://www.eccpage.com/goppacode.c`.

[Rab80] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.*, 9(2):273–280, 1980.

[Rhe80] W.C. Rheinboldt. *Horner's Scheme and Related Algorithms*. Modules and monographs in undergraduate mathematics and its applications. Birkhauser Boston, 1980.

[Ris] Thomas Risse. SAGE, ein open source CAS vor allem auch für die diskrete mathematik. `http://www.weblearn.hs-bremen.de/risse/papers/Frege2010_03/`.

[Ris11] Thomas Risse. How SAGE helps to implement Goppa codes and McEliece PKCSs. Home page of Thomas Risse, 2011. `http://www.weblearn.hs-bremen.de/risse/papers/ICIT11/526_ICIT11_Risse.pdf`.

[Sch12] Sara Schacht. McEliece and its Vulnerabilities to Side Channel Attacks. Bachelor thesis, Ruhr-Universität-Bochum, November 2012.

[Sen00] Nicolas Sendrier. Finding the permutation between equivalent linear codes: The support splitting algorithm. *IEEE Transactions on Information Theory*, 46(4):1193–1203, 2000.

[Sen05] N. Sendrier. Encoding information into constant weight words. In *Proceedings of the International Symposium on Information Theory, ISIT 2005*, pages 435–438, 2005.

[Sho93] Victor Shoup. Fast construction of irreducible polynomials over finite fields. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, SODA '93, pages 484–492, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[Sho94] Peter W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *FOCS*, pages 124–134, 1994.

[Sho01] Victor Shoup. OAEP reconsidered. In *Journal of Cryptology*, pages 239–259. Springer-Verlag, 2001.

[SKHN75] Yasuo Sugiyama, Masao Kasahara, Shigeichi Hirasawa, and Toshihiko Namekawa. A method for solving key equation for decoding goppa codes. *Information and Control*, 27(1):87–99, 1975.

[SS92] V.M. Sidel'nikov and S.O. Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Math. Appl.*, 2(4):439–444, 1992.

[SSMS10] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger. A timing attack against patterson algorithm in the mceliece pkc. In *Proceedings of the 12th international conference on Information security and cryptology*, ICISC'09, pages 161–175, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ste89] Jacques Stern. A method for finding codewords of small weight. In *Proceedings of the 3rd International Colloquium on Coding Theory and Applications*, pages 106–113, London, UK, UK, 1989. Springer-Verlag.

[Str10] Falko Strenzke. A Timing Attack against the Secret Permutation in the McEliece PKC. In *PQCrypto'10*, pages 95–107, 2010.

[Str11] Falko Strenzke. Timing Attacks against the Syndrome Inversion in Code-based Cryptosystems. *IACR Cryptology ePrint Archive*, pages 683–683, 2011.

[Str12] Falko Strenzke. A Side-Channel Secure and Flexible Platform-Independent Implementation of the McEliece PKC, September 2012. `http://crypto-source.de/flea.html`.

[Sud00] Madhu Sudan. List decoding: algorithms and applications. *SIGACT News*, 31(1):16–27, 2000.

[tec12] technologyreview.com. The CIA and Jeff Bezos Bet on Quantum Computing, October 2012. `http://www.technologyreview.com/news/429429/the-cia-and-jeff-bezos-bet-on-quantum-computing/`.

[Ten12] Thomas Tendyck. Efficient implementation of the McEliece cryptosystem on an ARMv7 CPU utilizing the NEON instruction set. Master's thesis, Ruhr-Universität Bochum, 2012.

[Uma11] Valérie Gauthier Umaña. *Post-Quantum Cryptography.* PhD thesis, Technical University of Denmark, 2011.

[VSB+01] Lieven M. K. Vandersypen, Matthias Steffen, Gregory Breyta, Costantino S. Yannoni, Mark H. Sherwood, and Isaac L. Chuang. Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance. 2001.

[Wie10] Christian Wieschebrink. Cryptanalysis of the niederreiter public key scheme based on GRS subcodes. In *Proceedings of the Third international conference on Post-Quantum Cryptography*, PQCrypto'10, pages 61–72, Berlin, Heidelberg, 2010. Springer-Verlag.

[XZL+11] Nanyang Xu, Jing Zhu, Dawei Lu, Xianyi Zhou, Xinhua Peng, and Jiangfeng Du. Quantum factorization of 143 on a dipolar-coupling nmr system. 2011.

[Zin96] Victor Zinoviev. On the Solution of Equations of Degree ≤ 10 over finite fields $GF(2^q)$. Rapport de recherche RR-2829, INRIA, 1996.