POST QUANTUM CRYPTOGRAPHY: IMPLEMENTING ALTERNATIVE PUBLIC KEY SCHEMES ON EMBEDDED DEVICES

Preparation for the Rise of Quantum Computers

DISSERTATION

for the degree of Doktor-Ingenieur
of the Faculty of Electrical Engineering and Information Technology
at the Ruhr-University Bochum, Germany

by Stefan Heyse
Bochum, October 2013
Post Quantum Cryptography:
Implementing Alternative Public Key Schemes on Embedded Devices
Für Mandy, Captain Chaos und den Böarn.
“If you want to succeed, double your failure rate.”
(Tom Watson, IBM).

“Kaffee dehydriert den Körper nicht. Ich wäre sonst schon Staub.”
(Franz Kafka)
Abstract

Almost all of today’s security systems rely on cryptographic primitives as core components which are usually considered the most trusted part of the system. The realization of these primitives on the underlying platform plays a crucial role for any real-world deployment. In this thesis, we discuss new primitives in public-key cryptography that could serve as alternatives to the currently used RSA, ECC and discrete logarithm cryptosystems. Analyzing these primitives in the first part of this thesis from an implementer’s perspective, we show advantages of the new primitives. Moreover, by implementing them on embedded systems with restricted resources, we investigate if these schemes have already evolved into real alternatives to the current cryptosystems.

The second and main part of this work explores the potential of code-based cryptography, namely the McEliece and Niederreiter cryptosystems. After discussing the classical description and a modern variant, we evaluate different implementation possibilities, e.g., decoders, constant weight encoders and conversions to achieve CCA2-security. Afterwards, we evaluate the performance of the schemes using plain binary Goppa codes, quasi-dyadic Goppa codes and quasi-cyclic MDPC codes on smartcard class microcontrollers and a range of FPGAs. We also point out weaknesses in a straightforward implementation that can leak the secret key or the plaintext by means of side channel attacks.

The third part is twofold. At first, we investigates the most promising members of Multivariate Quadratics Public Key Scheme (MQPKS) and its variants, namely Unbalanced Oil and Vinegar (UOV), Rainbow and Enhanced TTS (enTTS). UOV resisted all kinds of attacks for 13 years and can be considered one of the best examined MQPKS. We describe implementations of UOV, Rainbow and enTTS on an 8-bit microcontroller. To address the problem of large keys, we used several optimizations and also implemented the 0/1-UOV scheme introduced at CHES 2011. To achieve a security level usable in practice on the selected device, all recent attacks are summarized and parameters for standard security levels are given. To allow judgement of scaling, the schemes are implemented for the most common security levels in embedded systems of 64, 80 and 128 bits symmetric security. This allows a direct comparison of the four schemes for the first time, because they are implemented for the same security levels on the same platform.

The second contribution is an implementation of the modern symmetric authentication protocol LaPin, which is based on Ring-Learning-Parity-with-Noise (Ring-LPN). We show that, compared to classical AES-based protocols, LaPin has a very compact memory footprint while at the same time achieving a performance at the same order of magnitude.
Keywords

Kurzfassung


Schlagworte.

Acknowledgements

This thesis would have been impossible without the inspiring working atmosphere at the EmSec group. I would like to express my gratitude to my PhD supervisor Christof Paar. He always gave me the freedom to try things out, while at the same time encouraging me to focus on the important parts. Special thanks go to all my colleagues, especially Tim Güneysu, Amir Moradi, Markus Kasper, KaOs (Timo Kasper and David Oswald), Ingo von Maurich and Thomas Poppelmann. At all times I could ask them stupid questions and they found the time to discuss them with me.

And of course I would like to thank my office mate Ralf Zimmerman. I will never get the picture of him riding a pony in a bee costume out of my head. Beside this strange imagination, we are the best rubber ducking team ever.

A great “thank you” goes to the students I supervised during my PhD time: Without them, many parts of this thesis would be much smaller. Especially I would like to thank Olga Paustjan, Peter Czypek, Hannes Hudde and Gennadi Stamm.

I am also very grateful to the international community for the support I received. A lot of mathematicians accepted me as an engineer and answered my endless stream of questions. A big “thank you” goes to Nicolas Sendrier, Christiane Peters, Paulo Baretto, Rafael Misoczky, Dan Bernstein and Tanja Lange. Very often I felt ignorant of higher math and a second later I had to explain some basic engineering facts. Thank you for the nice time.

And finally, a big “thank you” to our team assistant Irmgard Kühn. She took all the administrative load off my shoulders and always found a kind word for everyone.
# Table of Contents

Preface ....................................................................................... vii  
Abstract ..................................................................................... vii  
Kurzfassung ................................................................................ x  
Acknowledgements ....................................................................... xiii

## I The Preliminaries

1 Introduction  
1.1 Motivation ................................................................. 5  
1.2 Thesis Outline ........................................................... 6  
1.3 Summary of Research Contributions ......................... 6

2 Overview  
2.1 Alternatives to Classical PKC ..................................... 9  
2.1.1 Hash-Based Cryptography .................................. 10  
2.1.2 Code-Based Cryptography ................................ 11  
2.1.3 Multivariate-Quadratic Cryptography ................ 11  
2.1.4 Lattice-Based Cryptography ............................... 12  
2.1.5 Summary ............................................................. 13

3 Embedded Systems  
3.1 Microcontroller ....................................................... 15  
3.2 Reconfigurable Hardware .......................................... 16

4 Finite Fields  
4.1 Field Representations ............................................. 19  
4.1.1 Polynomial Representation .................................. 20  
4.1.2 Exponential Representation .............................. 21  
4.1.3 Tower Fields .................................................... 21  
4.2 A New Approach: Partial Lookup Tables ................ 24

5 Attacking Classical Schemes using Quantum Computers  
5.1 Quantum Computing .................................................. 29  
5.1.1 Mathematical Definition of Qubits and Quantum Register ................. 29
# Table of Contents

5.1.2 Operations on Qubits and Quantum Registers .......................... 31
5.2 Grover’s Algorithm: A Quantum Search Algorithm ...................... 32
   5.2.1 Attacking Cryptographic Schemes .................................. 32
   5.2.2 Formulation of the Process ......................................... 34
5.3 Shor’s Algorithm: Factoring and Discrete Logarithm ................. 36
   5.3.1 Quantum Fourier Transform ....................................... 37
   5.3.2 Factoring and RSA .................................................. 37
   5.3.3 Factoring with Shor .................................................. 38
5.4 Discrete Logarithm with Shor ............................................. 41

II Code-based Cryptography .................................................. 43

6 Introduction to Error Correcting Codes ...................................... 47
   6.1 Motivation .............................................................. 47
   6.2 Existing Implementations ............................................. 48
   6.3 Outline ................................................................. 49
   6.4 Error Correcting Codes ................................................ 49
      6.4.1 Basic Definitions ................................................. 50
      6.4.2 Punctured and Shortened Codes ................................ 51
      6.4.3 Subfield Subcodes and Trace Codes .............................. 52
      6.4.4 Important Code Classes ........................................... 53
   6.5 Construction of Goppa Codes .......................................... 55
      6.5.1 Binary Goppa Codes ............................................... 56
      6.5.2 Parity Check Matrix of Goppa Codes ............................. 56
   6.6 Dyadic Goppa Codes .................................................... 57
   6.7 Quasi-Dyadic Goppa Codes ............................................. 60
   6.8 Decoding Algorithms for Goppa Codes ................................ 61
      6.8.1 Key Equation ....................................................... 61
      6.8.2 Syndrome Computation ............................................ 62
      6.8.3 Berlekamp-Massey-Sugiyama .................................... 62
      6.8.4 Patterson ........................................................... 65
   6.9 Extracting Roots of the Error Locator Polynomial ..................... 66
      6.9.1 Brute Force Search Using the Horner Scheme .................... 67
      6.9.2 Brute Force Search using Chien Search .......................... 68
      6.9.3 Berlekamp-Trace Algorithm and Zinoviev Procedures ........... 69
   6.10 MDPC-Codes ............................................................. 70
   6.11 Decoding MDPC Codes ................................................ 70

7 Cryptosystems Based on Error Correcting Codes .......................... 71
   7.1 Overview ............................................................... 71
# Table of Contents

7.2 Security Parameters .................................................. 72
7.3 Classical McEliece Cryptosystem .................................... 73
  7.3.1 Key Generation .................................................. 73
  7.3.2 Encryption ...................................................... 73
  7.3.3 Decryption ..................................................... 74
7.4 Modern McEliece Cryptosystem ..................................... 74
  7.4.1 Key generation .................................................. 75
  7.4.2 Encryption ...................................................... 76
  7.4.3 Decryption ..................................................... 76
7.5 Niederreiter Cryptosystem ......................................... 77
  7.5.1 Key generation .................................................. 77
  7.5.2 Encryption ...................................................... 78
  7.5.3 Decryption ..................................................... 78
  7.5.4 Constant Weight Encoding .................................... 79

8 General Security Considerations and New Side-Channel Attacks 81
  8.1 Overview ........................................................... 81
  8.2 Hiding the Structure of the Private Code ....................... 82
  8.3 Attacks .............................................................. 84
    8.3.1 Message Security ............................................. 84
    8.3.2 Key Security .................................................. 84
  8.4 Side Channel Attacks ............................................. 85
    8.4.1 Introduction to DPA ........................................ 86
    8.4.2 A Practical Power Analysis Attacks on Software Implementations of McEliece 86
    8.4.3 Gains of Power Analysis Vulnerabilities ................. 91
  8.5 Ciphertext Indistinguishability .................................. 95
  8.6 Key Length ........................................................ 96

9 Conversions for CCA2-secure McEliece Variants .................. 99
  9.1 Kobara-Imai-Gamma Conversion ................................... 100
  9.2 Fujisaki-Okamoto Conversion ..................................... 103

10 Microcontroller and FPGA Implementation of Codebased Crypto Using Plain Binary Goppa Codes 107
  10.1 Previous Work .................................................... 107
  10.2 Security Parameters ............................................. 108
  10.3 8-Bit Microcontroller Implementation .......................... 108
    10.3.1 Design Decisions .......................................... 108
    10.3.2 CCA2-Secure Conversions ................................ 109
    10.3.3 $t$-error Correction Using Berlekamp-Massey Decoder .. 109
    10.3.4 Adaptions and Optimizations .............................. 110
# Table of Contents

10.3.5 µC Results ........................................... 111
10.3.6 µC Conclusions .................................... 115
10.4 FPGA Implementation of the Niederreiter Scheme .................... 117
  10.4.1 Encryption ........................................ 117
  10.4.2 Decryption Using the Patterson Decoder ....................... 118
  10.4.3 Decryption Using the Berlekamp-Massey Decoder ............... 119
  10.4.4 FPGA Results ...................................... 119
10.5 Future Work ........................................... 121

11 Codebased Crypto Using Quasi Dyadic binary Goppa Codes .......... 125
  11.1 Scheme Definition of QD-McEliece ........................................ 125
      11.1.1 Parameter Choice and Key Sizes ................................ 127
      11.1.2 Security of QD-McEliece ........................................ 127
  11.2 Implementational Aspects ........................................... 128
      11.2.1 Field Arithmetic ........................................... 129
      11.2.2 Implementation of the QD-McEliece Variant ................... 129
      11.2.3 Implementation of the KIC-γ .................................. 135
  11.3 Results on an 8-Bit Microcontroller .................................. 135
  11.4 Conclusion and Further Research .................................... 138

12 Codebased Crypto Using Quasi Cyclic Medium Density Parity Check Codes .......... 141
  12.1 McEliece Based on QC-MDPC Codes .................................... 141
  12.2 Security of QC-MDPC ........................................ 142
  12.3 Decoding (QC-)MDPC Codes ......................................... 142
  12.4 Implementation on FPGA and Microcontroller ....................... 146
      12.4.1 Decoder and Parameter Selection .............................. 146
      12.4.2 FPGA Implementation ........................................ 147
      12.4.3 Microcontroller Implementation ................................ 149
  12.5 Results ................................................ 150
      12.5.1 FPGA Results ........................................... 150
      12.5.2 Microcontroller Results .................................... 152

III Other Alternative Public Key Schemes .................................. 157

13 Multivariate Quadratics Public-Key Schemes ............................ 159
  13.1 Introduction ............................................ 159
  13.2 Multivariate Quadratic Public-Key Cryptosystems .................. 160
  13.3 Security in a Nutshell ....................................... 163
      13.3.1 Security and Parameters of UOV and 0/1-UOV ................. 164
      13.3.2 Security and Parameters of Rainbow .......................... 165
Table of Contents

13.3.3 Security and Parameters of Enhanced TTS .................................. 165
13.4 Implementation on AVR Microprocessors .................................... 166
  13.4.1 Target Platform and Tools .................................................. 166
  13.4.2 Arithmetic and Finite Field ................................................ 167
  13.4.3 Key Size and Signature Runtime Reduction ............................... 167
  13.4.4 Verify Runtime Reduction .................................................. 168
  13.4.5 RAM Requirements .......................................................... 168
  13.4.6 Key Generation ............................................................. 168
13.5 Results .................................................................................... 169
13.6 Conclusion ............................................................................... 171
  13.6.1 Further Improvements ........................................................ 171
13.7 Toy example of 0/1 UOV Key Generation ....................................... 172

14 LaPin: An Efficient Authentication Protocol Based on Ring-LPN .......... 175
  14.1 Introduction ............................................................................ 175
  14.1.1 LPN, Ring-LPN, and Related Problems ................................. 177
  14.2 Definitions ............................................................................ 178
    14.2.1 Rings and Polynomials ...................................................... 178
    14.2.2 Distributions ................................................................. 178
    14.2.3 Authentication Protocols .................................................. 179
  14.3 Ring-LPN and its Hardness ...................................................... 179
    14.3.1 Hardness of LPN and Ring-LPN ........................................ 179
  14.4 Authentication Protocol ........................................................ 181
    14.4.1 The Protocol ................................................................. 182
    14.4.2 Analysis ......................................................................... 182
  14.5 Implementation ................................................................. 185
    14.5.1 Implementation with a Reducible Polynomial ........................ 186
    14.5.2 Implementation with an Irreducible Polynomial .................... 187
    14.5.3 Implementation Results .................................................... 187
  14.6 Conclusions and Open Problems .............................................. 189
  14.7 Man-in-the-Middle Attack ....................................................... 191

IV Conclusion .............................................................................. 193

15 Conclusion and Future Work ...................................................... 195
  15.1 Conclusion ............................................................................ 195
  15.2 Future Work .......................................................................... 197
# Table of Contents

## V The Appendix

<table>
<thead>
<tr>
<th>16 Appendix</th>
<th>201</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Listings</td>
<td>201</td>
</tr>
<tr>
<td>16.1.1 Listing primitive polynomials for the construction of Finite fields</td>
<td>201</td>
</tr>
<tr>
<td>16.1.2 Computing a normal basis of a Finite Field using SAGE</td>
<td>201</td>
</tr>
<tr>
<td>16.2 Definitions</td>
<td>202</td>
</tr>
<tr>
<td>16.2.1 Hamming weight and Hamming distance</td>
<td>202</td>
</tr>
<tr>
<td>16.2.2 Minimum distance of a codeword</td>
<td>202</td>
</tr>
<tr>
<td>16.2.3 One-way functions</td>
<td>203</td>
</tr>
<tr>
<td>16.2.4 Cryptographic Hash functions</td>
<td>203</td>
</tr>
<tr>
<td>16.2.5 One-time pad</td>
<td>203</td>
</tr>
</tbody>
</table>

## Bibliography

| 203 |
| List of Figures | 223 |
| List of Tables | 226 |
| List of Abbreviations | 229 |
| About the Author | 231 |
| Publications | 233 |
Part I

The Preliminaries
# Table of Contents

1 Introduction 5
  1.1 Motivation ................................................. 5
  1.2 Thesis Outline ............................................. 6
  1.3 Summary of Research Contributions .............................. 6

2 Overview 9
  2.1 Alternatives to Classical PKC ................................. 10

3 Embedded Systems 15
  3.1 Microcontroller ............................................. 16
  3.2 Reconfigurable Hardware ...................................... 17

4 Finite Fields 19
  4.1 Field Representations ...................................... 20
  4.2 A New Approach: Partial Lookup Tables ......................... 24

5 Attacking Classical Schemes using Quantum Computers 29
  5.1 Quantum Computing ......................................... 29
  5.2 Grover’s Algorithm: A Quantum Search Algorithm ............... 32
  5.3 Shor’s Algorithm: Factoring and Discrete Logarithm ............. 36
  5.4 Discrete Logarithm with Shor .................................. 41
Chapter 1

Introduction

This chapter provides the motivation for the necessity of alternative public-key schemes and gives a brief overview of the available constructions. Afterwards the thesis is outlined and corresponding research contributions are summarized.

1.1 Motivation

In the last years, embedded systems have continuously become more important. Spanning all aspects of modern life, they are included in almost every electronic device: small tablet PCs, smart phones, domestic appliances, and even in cars. This ubiquity goes hand in hand with an increased need for embedded security. For instance, it is crucial to protect a car’s electronic doorlock from unauthorized use. These security demands can be solved by cryptography. In this context, many symmetric and asymmetric algorithms, such as AES, (3)DES, RSA, ElGamal, and ECC, are implemented on embedded devices. For many applications, where several devices communicate each with other, advanced properties of public-key cryptosystems are required. Public-key cryptosystems offer the advantage that no initial, secure exchange of one or more secret keys between sender and receiver is required. In this way, secure authentication protocols can be realized. Such protocols are used, for instance, in car-to-car communications where a previous key exchange over a secure channel is not possible. Also, asymmetric cryptography allows digital signature which is useful for code update and device authentication.

All public-key cryptosystems frequently implemented rely on the basis of the presumed hardness of one of two mathematical problems: factoring the product of two large primes (FP) and computing discrete logarithms (DLP). Both problems are closely related. Hence, solving these problems would have significantly ramifications for classical public-key cryptography, and thus, for all embedded devices that make use of this algorithms. Nowadays, both problems are believed to be computationally infeasible with an ordinary computers. However, a quantum-computer, having the ability to perform computations on a few thousand qubits, could solve both problems by using Shor’s algorithm [Sho97]. Although a quantum computer of this dimension has not been reported, it is possible within one to three decades. Hence development and cryptanalysis of alternative public-key cryptosystems seems important. Cryptosystems not suffering from the critical security loss or even a fully broken system using quantum computers
are called post-quantum cryptosystems. Beside the threat introduced by quantum computers, we want to encourage a larger diversification of cryptographic primitives in future public-key applications. However, to be accepted as real alternatives to conventional systems, such security primitives need to support efficient implementations with a comparable level of security on recent embedded platforms.

Most published post-quantum public-key schemes are focused on the following approaches [BBD08]: Hash-based cryptography (e.g., Merkle’s hash-tree public-key signature system [Mer79]), Multivariate-quadratic-equations cryptography (e.g., HFE signature scheme [Pat96]), Lattice-based cryptography (e.g., NTRU encryption scheme [HPS98a]), and Code-based cryptography (e.g., McEliece encryption scheme [McE78], Niederreiter encryption scheme [Nie86]).

During the course of this thesis, we will show how to overcome most of the practical disadvantages of post-quantum schemes and how to implement them efficiently. Finally, we show that many of them can even outperform classical public-key ciphers in terms of speed and/or size.

1.2 Thesis Outline

This thesis deals with the emerging area of alternative public-key schemes. It is divided into three principal parts. The first part gives an brief overview of the implementational properties of these systems in Section 2.1. Then the required arithmetic is introduces and a new approach to balance memory and speed is presented. It ends with an summary of the possible attacks on classical cryptographic systems using quantum computers.

The second part gives an detailed discussion of code-based public-key schemes and their implementational aspects in Section 7. Then a wide variety of schemes is evaluated on different embedded systems in Sections 10, 11 and 12. Not only the core components, but also additionally required steps (e.g., to achieve CCA2-security) are discussed and evaluated.

The third part presents two other post-quantum cryptographic algorithms. First, the implementation of several MQ-based signature schemes is presented in Section 13. Afterwards, the lightweight authentication protocol Lapin, which is based on Ring-LPN, is presented and evaluated on a smart card CPU in Section 14.

1.3 Summary of Research Contributions

This thesis gives a detailed insight into the emerging area of alternative cryptosystems. This systems aim to ensure performance and security of cryptography in the advent of quantum computers. Hence, it serves as a motivation, an introduction, and a detailed treatment of the implementation of so called post quantum cryptography. Concretely, this thesis investigates the following research topics.
1.3. Summary of Research Contributions

Finite Field Implementations

The thesis starts with a description of current methods to implement finite field arithmetic. After summarizing the existing methods and pointing out their advantages and disadvantages, a new approach is presented. This approach, called partial lookup tables, achieves a flexible trade off between memory consumption and speed. Therefore, it allows an implementer to pick an optimal setting, utilizing available memory or matching a given performance requirement.

This research contribution is based on unpublished research.

Code-based Cryptography

This thesis also describes how to identify the individual security objectives of the entities involved in a typical vehicular IT application. It describes how to deduce the corresponding security requirements that fulfill the identified security objectives and can thwart all relevant security threats properly. For this, it moreover indicates some helpful advantages and several characteristic constraints that arise when establishing IT security in the automotive domain. This comprises also several organizational security aspects from the vehicle manufacturer’s perspective.

This research contribution is based on the author’s published research in [EGHP09, Hey10, HMP10, Hey11, HG12, SH13].

Multivariate Quadratics Cryptography

This thesis further provides a solid set of practical vehicular security technologies and vehicular security mechanisms adapted for applications in the automotive domain that can implement the identified security requirements accordingly. This comprises an overview about general vehicular security technologies such as physical security measures, vehicular security modules, and vehicular security architectures, but also concretely practical security mechanisms for vehicle component identification, secure vehicle initialization, vehicle user authentication, as well as cryptographic schemes for securing in-vehicle and external vehicle communications. For this, several solutions are based on the technology of Trusted Computing, which is well-established in today’s PC world and newly emerges also into the world of embedded computing.

This research contribution is based on the author’s published research in [CHT12].

LPN based Cryptography

The thesis lastly introduces feasible vehicular security architectures capable to enable several advanced schemes for intellectual property, expertise, and software protection. It describes new schemes for secure content distribution capable for—but not limited to—applications in the automotive world and in the world of mobile computing with its characteristic constraints. Therefore, it introduces new security protocols, components, and mechanisms based on the technologies of virtualization and Trusted Computing.

This research contribution is based on the author’s published research in [HKL+ 10].
Chapter 2
Overview

The major benefits from public-key cryptography (PKC) are invaluable security services, such as non-repudiable digital signatures [RSA78] or the secure secret key exchange over untrusted communication channels [DH76]. To enable these advanced features, the security of practical PK schemes are based on so-called one-way trapdoor functions. PKC should enable everyone to make use of a cryptographic service or operation involving the public key $k_{pub}$ and the one-way function $y = f(x, k_{pub})$ to protect a message $x$. The message $x$ can only be recovered using the inverse trapdoor function $x = g(y, k_{pr})$, which requires knowledge of the secret component $k_{sec}$. One-way trapdoor functions for PKC are selected from a set of hard mathematical problems augmented with a trapdoor for easy recovery with special knowledge. One-way trapdoor functions which are used in well-established cryptosystems are based on the following mathematical problems:

**Integer Factorization Problem (FP):** For a composite integer $n = \prod p_i$ consisting of unknown primes $p_i$ it is considered hard to retrieve $p_i$ when $n$ and the primes $p_i$ are sufficiently large. This is the fundamental problem used in the RSA cryptosystem [RSA78].

**Discrete Logarithm Problem in Finite Fields (DLP):** For an element $a \in G$ and $b \in \langle a \rangle$, where $G$ is the multiplicative group of a finite field and $\langle a \rangle$ the subgroup generated by $a$, it is assumed to be hard to compute $\ell$ where $b \equiv a^\ell$ if $\langle a \rangle$ is sufficiently large. This difficult problem founds the basis for the ElGamal and Diffie-Hellman cryptosystem [ElG85].

**Elliptic Curve Discrete Logarithm Problem (ECDLP):** For an element $a \in E$ and $b \in \langle a \rangle$, where $E$ is an elliptic curve over a finite field and $\langle a \rangle$ the subgroup generated by $a$, it is assumed to be hard to compute $\ell$ where $b \equiv a^\ell$ if $\langle a \rangle$ is sufficiently large. The ECDLP is the problem used for ECC crypto systems [HMV03].

In general, all computations required by these three problems rely on arithmetic over integer rings or finite fields (i.e., either prime fields $GF(p)$ or binary extension fields $GF(2^m)$). Note that the size of operands for these operations is very large with lengths of 1024 bits more for RSA and discrete logarithms; even finite fields used in ECC require parameter lengths of over 160 bits. In this context, the modular multiplication with such very large operands plays a crucial role for all classical cryptosystems and thus represents the main burden for the underlying processing platform. More precisely, a single modular multiplication with 1024-bit
operand length performed on an 8-bit microprocessor involves thousands of 8-bit multiplication and addition instructions, making such classical cryptosystems slow and inefficient. This is why a closer look on alternative public-key crypto system (APKC)s – that possibly provide a significantly better performance – is very attractive.

2.1 Alternatives to Classical PKC

In addition to established families and problem classes of PKC schemes (see above), there exist a few more which are of interest for cryptography. Some are based on NP-complete problems, such as knapsack schemes, which, however, have been broken or are believed to be insecure. Second, there are generalizations of the established algorithms, e.g., hyperelliptic curves, algebraic varieties or non-RSA factoring based schemes. Third, there are algorithms (namely, APKCs) for which, according to our current knowledge, no attacks are known and which appear to be secure against classical cryptanalysis and cryptanalysis with quantum computers. Therefore, these are sometimes also referred to as Post Quantum Cryptography (PQC) schemes instead of APKCs. Since about 2005, there has been a growing interest in the cryptographic community in this latter class of schemes. Currently four families of algorithms, which will be introduced below, are considered the most promising candidates. Interestingly, two of them, hash-based and code-based schemes, are believed to be at least as secure as established algorithms which rely on number-theoretical assumptions. Furthermore, they are also resilient against progress in factoring or discrete log algorithms.

2.1.1 Hash-Based Cryptography

Generic hash functions are used as a base operation for generating digital signatures, usually using a directed tree graph. The idea was introduced in 1979 by Lamport [Lam79] who proposed a one-time signature scheme. The idea was improved by Winternitz to allow for more efficient signing of larger data. In 1989 Merkle published a tree-based signature scheme to enhance one-time signatures [Mer89]. The so-called Merkle signature scheme (MSS) allows for a larger number of signatures because the binary hash tree strongly decreases the amount of storage needed. The advantage of the MSS is a provable security, relying only on the security of the underlying hash function. The disadvantage of a limited number of signatures was solved by [BCD+] by constructing multiple levels of Merkle’s hash trees, allowing for a sufficiently large number of signatures for almost all practical cases. Hash-based signature schemes are adaptable to many different application scenarios (for further information, please refer to Table 2.1). Their performance, key sizes and signature sizes depend on the underlying hash function, the maximum number of signatures and other factors, allowing for various trade-offs. The MSS has a very short public key (output length of the underlying hash function), a relatively long signature length (length can be traded for computation time), and a computationally expensive key generation. Though the private key is quite large, it does not have to be stored, but parts of it can be generated on the fly. Another degree of freedom for the designer is the choice of the
underlying hash function. Dedicated algorithms such as SHA-1 or SHA-2 or block cipher-based ones are all possible. In summary, there are many design choices possible which make MSS a particularly interesting target for software and hardware implementations.

2.1.2 Code-Based Cryptography

In 1978, R. McEliece introduced a public-key encryption scheme based on error-correcting codes [McE78]. It is in fact one of the best investigated public-key schemes. The McEliece cryptosystem is based on the advantage that efficient decoders exist for some codes like general Goppa codes, but not for (unknown) general linear codes, for which decoding is known to be NP-hard. Since then, related coding-based public-key schemes have been proposed, such as the Niederreiter cryptosystem [Nie86] or the code-based signature scheme [CFS01].

The core operation for signing is matrix-vector multiplications, which makes it very efficient (in fact faster than most of the established asymmetric schemes). The main operation during decryption is decoding Goppa codes over \( \text{GF}(2^m) \) which typically requires the extended Euclidean algorithm, which is efficient for the parameters used for McEliece signatures. The key sizes for secure parameter sets vary from hundreds of kilobytes up to megabytes for the private key. Key generation involves as core operation matrix inversion which is also efficient. In summary, from a practical view point, code-based cryptosystems enjoy interesting features (fast encryption/decryption, good security reduction) but also have their drawbacks (large key sizes, encryption overhead, expensive signature generation).

Although some attacks have been proposed, the McEliece cryptosystem is considered highly secure as long as the parameters are chosen carefully and it is used correctly [Ber97]. Given that it has been in existence and analyzed for 30 years, McEliece can be considered a very trusted public-key scheme. However, the main reason why it has not been used in practice is the large key sizes. Thus, McEliece is an very interesting alternative scheme, as future technology will make it increasingly easier to deal with very long key lengths.

2.1.3 Multivariate-Quadratic Cryptography

The problem of solving multivariate quadratic equations (MQ-problem) over finite fields for building public-key schemes dates back to Matsumoto and Imai [IM85]. Independently, Shamir developed a version based on integer rings rather than small finite fields. Solving general MQ equations is known to be \( \mathcal{NP} \)-complete and the various MQ schemes attempt to approximate the general case. Both signature and encryption schemes based on the problem of solving multivariate quadratic equations have been proposed, yet only the signature schemes have survived general cryptanalysis. One class of MQ algorithms are the small-field schemes, including rather conservative schemes such as Unbalanced Oil and Vinegar (UOV) as well as more aggressively designed proposals such as Rainbow or amended TTS (amTTS). The big-field classes include HFE (Hidden Field Equations), MIA (Matsumoto Imai Scheme A) and the mixed-field class \( \ell \text{IC} – \ell \)-Invertible Cycle [DWY07] scheme. An overview over public-key schemes based on multivariate quadratics can be found in [WP05]. Some presentations of new schemes also contain...
information about reference implementations, such as [YC05]. An implementation of Rainbow
was benchmarked on different PC platforms in [BLP08a] and in [YCC04] an implementation
for 8-bit smart card processor was presented. Besides these, little is known about their imple-
mentation properties.

Although the schemes differ in details of the mathematical steps taken in signature generation,
some general principles can be identified. One crucial part is computing affine transformations,
i.e. vector addition and matrix-vector multiplication. For signature generation for schemes of
the small field class, solving linear systems of equations (LSEs) over finite fields is the major
operation, while signature verification always involves (partially) evaluating multivariate poly-
nomials over Galois fields. Depending on the finite field and the chosen scheme, key sizes can
exceed several kilobytes. In summary, a high degree of freedom exists for selecting the scheme,
the underlying finite field, and operand sizes which forms a challenging optimization problem.

2.1.4 Lattice-Based Cryptography

Lattice-based cryptography is the newest of the class of APKC schemes and is currently an
active research area [MR04, Reg09]. There are several hard problems that can be used to
build cryptosystems on lattices, the most popular is the Shortest Vector Problem (SVP). In
general, the situation for lattice-based cryptography is the opposite to MQ schemes: lattice-
based encryption schemes have been found to be more secure than digital signatures and we will
concentrate on the former. The first proposal was by Ajtai and was related to hash function
construction. The first encryption scheme was the GHQ scheme [GGH97]. Even though GHQ
and its HNF variant [Mic01] have security problems, they motivated many follow-up schemes.
The most promising candidates of a lattice-based scheme with a proof of security are currently
LWE schemes and their variants [Reg05]. However, key sizes in the range of hundreds of
kilobytes are an indication for the implementation research that is needed here.

A very different lattice-based scheme is NTRU, which exists as signature and encryption scheme.
NTRU was first proposed at the CRYPTO 1996 rump session, was described in detail in 1998
[HPS98b] and underwent subsequently several iterations. The current encryption version is
cryptographically secure and the NAEP/SVES-3 variant has certain provable security properties
[HGSSW]. This version is included in the IEEE standard 1363.1, making it one of the PQC with
a very practical outlook. NTRU encryption and decryption are very fast. They consist of one
discrete convolution and two discrete convolutions, respectively. The operands are polynomials
over an integer ring. The polynomial degree is moderate (typically below 800), and the integer
ring \( \mathbb{Z}_q \) is usually given by a prime with a binary length of 8–10 bits. Due to the convolution,
one important property of NTRU is that its bit complexity is quadratic as opposed to the
cubic bit complexity of established public-key schemes. Thus, NTRU is particularly interesting
for practice. The main operation in key generation is polynomial inversion which is achieved
through the extended Euclidean algorithms.
2.1. Alternatives to Classical PKC

2.1.5 Summary

Table 2.1 summarized the properties of APKC schemes relevant from an implementation point of view. We observe that there is a wide variety of operand types, operand sizes and algorithms needed, which makes implementation research particularly interesting.

Table 2.1: Implementation Characteristics of PQC Schemes

<table>
<thead>
<tr>
<th>Crypto Scheme</th>
<th>Signature</th>
<th>Encryption</th>
<th>Key Size (in bytes)</th>
<th>Data Types</th>
<th>Core Ops.</th>
<th>Cryptographic Maturity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash-Based</td>
<td>yes</td>
<td>no</td>
<td>≈ 20</td>
<td>hash outputs</td>
<td>hashing</td>
<td>high</td>
</tr>
<tr>
<td>Multivariate</td>
<td>yes</td>
<td>no</td>
<td>≈ 10k</td>
<td>GF(2^m)</td>
<td>matrix mult. LSE solving</td>
<td>low, medium for conservative schemes</td>
</tr>
<tr>
<td>Quadratic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lattice-Based:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NTRU</td>
<td>maybe</td>
<td>yes</td>
<td>&lt; 0.1k</td>
<td>Z_q</td>
<td>convolution</td>
<td>medium</td>
</tr>
<tr>
<td>General lattice</td>
<td>maybe</td>
<td>yes</td>
<td>≈ 100k</td>
<td>GF(2^m)</td>
<td>matrix mult.</td>
<td>medium</td>
</tr>
<tr>
<td>Code-Based</td>
<td>expensive</td>
<td>yes</td>
<td>≈ 100k</td>
<td>GF(2^m)</td>
<td>matrix mult. decoding</td>
<td>high, with precautions to implementation</td>
</tr>
</tbody>
</table>
Chapter 3

Embedded Systems

In the last years, the need for embedded systems has arisen continuously. Spanning all aspects of modern life, they are in almost every electronic device: mobile phones, smart phones, domestic appliances, digital watches and even in cars. The vast majority of today’s computing platforms are embedded systems\cite{Tur02}. This trend continues, together with a differentiation of the classical embedded systems into more subcategories with special requirements. Only a few years ago, most of these devices could only provide a few bytes of RAM and ROM which was a strong restriction for application (and security) designers. But nowadays, even many microcontroller provide enough memory to implement high security schemes. They range from small 4-bit microcontroller to large systems with multiple strong CPUs connected by a network. A typical representative of the low end systems are 8-bit microcontrollers used in many smart cards. The device used in all implementations in this thesis, the AVR microcontroller by Atmel, is introduced in Section 3.1.

The other end of the spectrum are high performance Field Programmable Gate Array (FPGA)s for real-time applications or high speed data processing. They are introduced in Section 3.2.

Figure 3.1: Growth market of embedded systems\cite{IDC,2012}
### 3.1 Microcontroller

The AVR family of microcontrollers ranges from small ATtiny types with only 512 Bytes/256 Bytes of Flash/SRAM memory up to large XMEGA devices with 384 Kbyte/32 KByte Flash/SRAM memory. They can be programmed using the freely available avr-gcc compiler in C or assembly language and offer 32 generic 8-bit working registers. Almost all instructions working on these registers are completed in one clock cycle. Beside the processor they incorporate many peripheral units and interfaces like timers, AD-, DA-converters, and USART/I2C/SPI bus controller. Together with a power supply, a single micro controller can already form a complete embedded system. In contrast to standard x86-based PCs, they are running at lower clock frequencies, have less RAM and ROM and a smaller instructions set. Together, this makes implementing APKCs a challenging task.

A block diagram of the large XMEGA256 is shown in Figure 3.2.

![Figure 3.2: XMEGA block diagram](image)

One important thing to note is that the AVR is an Harvard architecture with two separate buses for SRAM and Flash memory. Loading data from internal SRAM takes 2 clock cycles. For accessing the Flash memory 3 clock cycles are required. For frequently accessed data it is therefore advisable to copy them to SRAM at start up for faster access later on.
3.2 Reconfigurable Hardware

**FPGA** stands for Field Programmable Gate Array. It consists of a large amount of LUTs that can generate any logic combination with four (up to six in newer types) inputs and one output and basic storage elements based on FFs. Two LUTs and two FFs are packed together into a slice and four slices into a CLB. Between the CLBs is a programmable switch matrix that can connect the input and outputs of the CLBs. How LUTs and CLBs are configured is defined in a vendor specific binary file, the *bitstream*. Additionally, most modern FPGAs also contain dedicated hardware like multiplier, clock manager, and configurable block RAM.

![4-Input LUT with FF](image1)

Figure 3.3: 4-Input LUT with FF [Inca]

One advantage of FPGAs for implementing cryptographic schemes are their flexibility. The programmer is not forced to use registers of a fixed width of 8, 32 or 64 bit, but can instantiate resources in any width (e.g., an 23-bit multiplier or an 1023 bit rotate by 17). The second advantage is the possibility of parallelism. As long as there are enough free resources in the FPGAs fabric, any given component can be instantiated many times. Each of this instances is then operating truly in parallel and not pseudo parallel as in a single core CPU running multiple threads.

After writing the Very High Speed Integrated Circuit Hardware Description Language (*VHDL*) or Verilog code in an editor, it is translated to a net list. This process is called synthesis. Based on the net list the correct behaviour of the design can be verified by using a simulation tool. This both steps are completely hardware independent. The next step is mapping and translating the net list into logic resources and special resources offered by the target platform.
Due to this hardware dependency, those and the following steps need to know the exact target hardware. The final step place-and-route (PAR) then tries to find an optimum placement for the single logic blocks and connects them over the switching matrix. The output of PAR can now be converted into a bitstream file and loaded into a flash memory on the FPGA board. On most FPGAs the memory for holding the bitstream is located outside the FPGA chip and can therefore be accessed by anyone. To protect the content of the bitstream, which may include intellectual property (IP) cores or, like in our case, secret key material, the bitstream can be stored encrypted \cite{xila}. The FPGA boot-up logic then has to decrypt the bitstream before configuring the FPGA. Some special FPGAs, for example the Spartan3-AN series, contain large on-die flash memory, which can only be accessed by opening the chip physically. For the decryption algorithm the bitstream file has to be protected by one of the two methods mentioned above. Note however, that also the Spartan3-AN does also not offer perfect security: Spartan3-AN FPGAs are actually assembled as stacked-die (i.e., a Flash memory on top of a separate die providing the reconfigurable logic), so an attacker can simply open the case and tap the bonding wires between the two dies to get access to the configuration data as well as the secret key. Therefore, it is mandatory to enable bitstream encryption using AES-256 which is available for larger Xilinx Spartan-6 and all Xilinx Virtex-FPGAs starting from Virtex-4. Also note that the Xilinx specific bitstream encryption \cite{xib} was successfully attacked by side-channel analysis in \cite{mkp12}. See \cite{ies} for an updated list of broken systems. Public keys can be stored either in internal or external memory since they do not require special protection.
Chapter 4
Finite Fields

As finite fields are the basis for the arithmetic used in the systems implemented later on, this chapter introduces the necessary terms and definitions. Also different representations of the same finite field and their advantages and disadvantages are presented. Finally, we present a new approach for a time-memory trade-off, called partial tables.

Finite fields
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 $\text{GF}(p^m)$ or $\mathbb{F}_q$, where $p$ is a prime number called the characteristic of the field and $m \in \mathbb{N}$. 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 $\mathbb{F}_{p^m}$. $\alpha$ is called a generator or primitive element of a finite field if every element of the field $\mathbb{F}_{p^m}^* = \mathbb{F}_{p^m}\setminus\{0\}$ can be represented as a power of $\alpha$. 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 $\mathbb{F}$ based on [LN97, HP03].

Definition 4.0.1 (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 4.0.2 (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 4.0.3 (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
Chapter 4. Finite Fields

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 shown in Chapter 6.8.

**Definition 4.0.4 (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 4.0.5 (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 \( \gcd(f, q) = \gcd(f, rf(q)) \) for any polynomial \( r \). The **Extended Euclidean Algorithm** shown in Alg. 1 additionally finds polynomials \( x, y \) in \( \mathbb{F} \) that satisfy Bézout’s identity

\[
\gcd(a(z), b(z)) = a(z)x(z) + b(z)y(z). \tag{4.0.1}
\]

**Algorithm 1 Extended Euclidean Algorithm (EEA)**

| Input: Polynomials \( a(x), b(x) \in \mathbb{F}[z] \), \( \deg(a) \geq \deg(b) \) |
| Output: Polynomials \( x(z), y(z) \) with \( \gcd(a, b) = ax + by \) |
| 1: \( u(z) \leftarrow 0, u_1(z) \leftarrow 1 \) |
| 2: \( v(z) \leftarrow 0, v_1(z) \leftarrow 0 \) |
| 3: **while** \( \deg a > 0 \) **do** |
| 4: \( (\text{quotient, remainder}) \leftarrow a(z) \div_{\mathbb{F}[z]} b(z) \) |
| 5: \( a \leftarrow b, b \leftarrow \text{remainder} \) |
| 6: \( u_2 \leftarrow u_1, u_1 \leftarrow u, u \leftarrow u_2 - \text{quotient} \cdot u \) |
| 7: \( v_2 \leftarrow v_1, v_1 \leftarrow v, v \leftarrow v_2 - \text{quotient} \cdot v \) |
| 8: **end while** |
| 9: **return** \( x \leftarrow u_1(z), y \leftarrow v_1(z) \) |

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

**4.1 Field Representations**

Let \( \mathbb{F}_q \) denote the finite field \( \mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/p(x) \) where \( p(x) \) is an irreducible polynomial of degree \( m \) over \( \mathbb{F}_2 \). Furthermore, let \( \alpha \) denote a primitive element of \( \mathbb{F}_q \).
4.1. Field Representations

4.1.1 Polynomial Representation

Every element \( a \in \mathbb{F}_q \) has a polynomial representation \( a(x) = a_{m-1}x^{m-1} + \cdots + a_1x + a_0 \mod p(x) \) where \( a_i \in \mathbb{F}_2 \). The addition of two field elements \( a \) and \( b \) is done using their polynomial representations such that \( a + b = a(x) + b(x) \mod p(x) \equiv c(x) \) with \( c_i = a_i \oplus b_i \), \( \forall i \in \{0, \ldots, m - 1\} \). The field addition can be implemented efficiently by performing the exclusive-or operation of two unsigned \( m \)-bit values. For simplicity, the coefficient \( a_0 \) should be stored in the least significant bit and \( a_{m-1} \) in the most significant bit of an unsigned \( m \)-bit value.

4.1.2 Exponential Representation

Furthermore, any element \( a \in \mathbb{F}_q \) except the zero element can be represented as a power of a primitive element \( \alpha \in \mathbb{F}_q \) such that \( a = \alpha^i \) where \( i \in \mathbb{Z}_{2^m-1} \). The exponential representation allows to perform more complex operations such as multiplication, division, squaring, inversion, and square root extraction more efficiently than polynomial representation. The field multiplication of two field elements \( a = \alpha^i \) and \( b = \alpha^j \) is easily performed by addition of both exponents \( i \) and \( j \) such that

\[
\alpha \cdot b = \alpha^i \cdot \alpha^j \equiv \alpha^{i+j} \mod 2^{m-1} \equiv c, \ c \in \mathbb{F}_q.
\]

Analogously, the division of two elements \( a \) and \( b \) is carried out by subtracting their exponents such that

\[
\frac{a}{b} = \frac{\alpha^i}{\alpha^j} \equiv \alpha^{i-j} \mod 2^{m-1} \equiv c, \ c \in \mathbb{F}_q
\]

The squaring of an element \( a = \alpha^i \) is done by doubling its exponent and can be implemented by one left shift.

\[
a^2 = (\alpha^i)^2 \equiv \alpha^{2i} \mod 2^{m-1}
\]

Analogously, the inversion of \( a \) is the negation of its exponent.

\[
a^{-1} = (\alpha^i)^{-1} \equiv \alpha^{-i} \mod 2^{m-1}
\]

The square root extraction of an element \( a = \alpha^i \) is performed in the following manner.

- If the exponent \( i \) of \( a \) is even, then \( \sqrt{a} = (\alpha^i)^{\frac{1}{2}} \equiv \alpha^{\frac{i}{2}} \mod 2^{m-1} \).
- If the exponent \( i \) of \( a \) is odd, then \( \sqrt{a} = (\alpha^i)^{\frac{1}{2}} \equiv \alpha^{\frac{i+2^{m-1}-1}{2}} \mod 2^{m-1} \).

If the exponent of \( a \) is even the square root extraction can be implemented by one right shift of the exponent. If the exponent is odd, it is possible to extend it by the modulus \( 2^m - 1 \), which leads to an even value. Then the square root extraction is performed as before through shifting the exponent right once.

To implement the field arithmetic on an embedded microcontroller most efficiently both representations of the field elements of \( \mathbb{F}_q \), polynomial and exponential, should be precomputed and stored as log- and antilog table, respectively. Each table occupies \( m \cdot 2^m \) bits storage.
4.1.3 Tower Fields

For larger extension fields these tables become very large compared to the available memory of embedded devices. For example in \( F_{2^{16}} \), we cannot store the whole log- and antilog tables on a small microcontroller because each table is 128 Kbytes in size. Neither the SRAM memory of an ATXmega256A1 (16 Kbytes) nor the Flash memory (256 Kbytes) would be enough to implement anything else after completely storing both tables. Hence, we must make use of the slower polynomial arithmetic or the called tower fields. Efficient algorithms for arithmetic over tower fields were proposed in [Afa91], [MK89], and [Paa94].

It is possible to view the field \( F_{2^{16}} \) as a field extension of degree 2 over \( F_{2^8} \) where \( k = 1, 2, 3, \ldots \). The idea is to perform field arithmetic over \( F_{2^8} \) in terms of operations in a subfield \( F_{2^k} \). Thus, we can consider the finite field \( F_{2^{16}} = F_{(2^8)^2} \) as a tower of \( F_{2^8} \) constructed by an irreducible polynomial \( p(x) = x^2 + x + p_0 \) where \( p_0 \in F_{2^8} \). If \( \beta \) is a root of \( p(x) \) in \( F_{2^{16}} \) then \( F_{2^{16}} \) can be represented as a two dimensional vector space over \( F_{2^8} \) and an element \( A \in F_{2^{16}} \) can be written as \( A = a_1 \beta + a_0 \) where \( a_1, a_0 \in F_{2^8} \). To perform field arithmetic over \( F_{2^{16}} \) we store the log- and antilog tables for \( F_{2^8} \) and use them for fast mapping between exponential and polynomial representations of elements of \( F_{2^8} \). Each of these tables occupies only 256 bytes, reducing the required memory by a factor of 512.

The field addition of two elements \( A \) and \( B \) in \( F_{2^{16}} \) is then performed through

\[
A + B = (a_1 \beta + a_0) + (b_1 \beta + b_0) = (a_1 + b_1) \beta + (a_0 + b_0) = c_1 \beta + c_0
\]

and involves two field additions over \( F_{2^8} \) which is equal to two xor-operations of 8-bits values.

The field multiplication of two elements \( A, B \in F_{2^{16}} \) is carried out through

\[
A \cdot B = (a_1 \beta + a_0)(b_1 \beta + b_0) \mod p(x) \equiv (a_0 b_1 + b_0 a_1 + a_1 b_1) \beta + (a_0 b_0 + a_1 b_1 p_0).
\]

and involves three additions and five multiplications over \( F_{2^8} \) when reusing the value \( a_1 b_1 \) which already has been computed in the \( \beta \)-term.

The squaring is a simplified version of the multiplication of an element \( A \) by itself in a finite field of characteristic 2, and is performed as follows

\[
A^2 = (a_1 \beta + a_0)^2 \mod p(x) \equiv a_0^2 \beta^2 + a_1^2 \quad \text{and} \quad p(x) \equiv a_0^2 \beta + (a_1^2 + p_0).
\]

One squaring over \( F_{2^{16}} \) involves two square operations and one addition over \( F_{2^8} \).

The field inversion is more complicated compared to the operations described above. An efficient method for inversion in tower fields of characteristic 2 is presented in [Paa94]. The inversion of an element \( A \) is performed through

\[
A^{-1} = \left( \frac{a_1}{\Delta} \right) \beta + \frac{a_0 + a_1}{\Delta} = c_1 \beta + c_0 \quad \text{where} \quad \Delta = a_0(a_1 + a_0) + p_0 a_1^2
\]

and involves two additions, two divisions, one squaring, and two multiplications over \( F_{2^8} \), when reusing the value \( (a_0 + a_1) \).

The division of two elements \( A, B \in F_{2^{16}} \) can be performed through multiplication of \( A \) by the inverse \( B^{-1} \) of \( B \). This approach requires five additions, seven multiplications, two divisions,
and one squaring over $F_{2^8}$. To enhance the performance of the division operation we provide a slightly better method given below.

\[
A/8 = A \cdot B^{-1} = \left( \frac{a_0(b_0 + b_1) + a_1 b_1 p_0}{\Delta} \right) \beta + \left( \frac{a_0 b_1 + a_1 b_0}{\Delta} \right) \text{ where } \Delta = b_0(b_1 + b_0) + p_0 b_1^2
\]

This method involves one less addition compared to the naive approach mentioned above.

The last operation we need for the implementation of the later presented schemes (cf. Section 7) is the extraction of square roots. We could not find any formula for square root extraction over tower fields in the literature, therefore, we developed one for this purpose. For any element $A \in F_{2^{16}}$ there exists a unique square root, as the field characteristic is 2. Hence, the following holds for the square root of $A$

\[
\sqrt{A} = \sqrt{a_1 \beta + a_0} \equiv \sqrt{a_1} \sqrt{\beta} + \sqrt{a_0} \mod p(x)
= \sqrt{a_1} (\beta + \sqrt{p_0}) + \sqrt{a_0} \mod p(x)
= \sqrt{a_1} \beta + (\sqrt{a_1} \sqrt{p_0} + \sqrt{a_0}).
\]

**Proof:**

As $\text{Char}(F_{2^{16}})$ is 2,

\[
\sqrt{A} = \sqrt{a_1 \beta + a_0} \equiv (a_1 \beta + a_0)^{2^7} \mod 2^{8-1} \equiv a_1^{2^7} \beta^{2^7} + a_0^{2^7} \quad (4.1.1)
\]

For any element $y$ in $F_{2^8}$ the trace function is defined by $Tr(y) = \sum_{i=0}^{7} y^{2^i} \equiv \begin{cases} 1 & \text{if } Tr(p_0) = 1 \\ 0 & \text{if } Tr(p_0) = 0 \end{cases}$

Furthermore, $\beta$ satisfies $\beta^2 \equiv \beta + p_0$, as $\beta$ is root of $p(x)$. Hence, we can write

\[
\beta^{2^7} = (\beta^2)^{2^6} \equiv (\beta + p_0)^{2^6} \equiv (\beta + p_0^6 + p_0) \equiv \cdots \equiv \beta + \sum_{i=1}^{6} p_0^{2^i} \equiv \beta + \sum_{i=0}^{7} p_0^{2^i} + 1 + p_0^{2^7} \equiv \begin{cases} \beta + p_0^{2^7} & \text{if } Tr(p_0) = 1 \\ \beta + 1 + p_0^{2^7} & \text{if } Tr(p_0) = 0 \end{cases}
\]

We assume that $Tr(p_0) = 1$. Otherwise, the polynomial $p(x)$ would not be irreducible, and thus, unsuited for the field construction.

Applying the intermediate results to the equation 4.1.1 we obtain

\[
\sqrt{A} \equiv \sqrt{a_1^{2^7}} \mod 2^{8-1} (\beta + p_0^{2^7} \mod 2^{8-1}) + \sqrt{a_0^{2^7}} \mod 2^{8-1} \equiv a_1^{2^7-1} \mod 2^{8-1} \beta + a_1^{2^7-1} p_0^{2^7} \mod 2^{8-1} \beta + a_0^{2^7-1} \mod 2^{8-1} \equiv \sqrt{a_1} \beta + \sqrt{a_1} \sqrt{p_0} + \sqrt{a_0}
\]
4.2 A New Approach: Partial Lookup Tables

The tower fields arithmetic presented above is some kind of balance between using the full blown lookup tables as in Sec 4.1.2 and using no tables as in Sec 4.1.1. But the construction is only possible for fields of the form $\mathbb{F}_{2^k}$. As we will see later on, some systems use parameters like $\mathbb{F}_{2^{11}}$ or $\mathbb{F}_{2^{13}}$, where the memory reduction using tower fields is not possible. Together with C. Wolf, we searched for a way to reduce the memory consumption of the lookup tables while maintaining an acceptable speed.

In the classical lookup table approach, each field element in polynomials representation has a corresponding entry in the log table and vice versa. We looked for a way to store only entries for selected elements and modify others until a lookup is possible. To allow an efficient implementation on constrained microcontrollers, we decided to focus on 8-bit machines. Therefore, we tried to store only tables with at most 256 entries (addressable by 8bits) and use more than one table if necessary. To ease implementation these 8 bits are located within the lower 8 bits of the polynomial representation. The upper bits then decide if an element can be directly looked up. The challenge is to minimize the number of partial tables while at the same time minimizing the number of operation required for the modification of the elements for which no table entry exists. An exhaustive search showed, that at least three partial tables are required to be able to lookup all elements. For example in $\mathbb{F}_{2^{11}}$, all elements which have upper bits $b_{10,9,8} = [100, 010, 001]$ can be looked up directly. All other elements are squared until they fulfill the pattern $b_{10,9,8} = [100, 010, 001]$. This squares are called Search Squares (SS). We chose the squaring operation, because squaring in a polynomial basis is just inserting a zero bit between each bit followed by a reduction. Successive squaring operations will result in a cycle of generated elements. But not all possible elements are generated, which is the reason for using more than one table. The program used for the exhaustive search also counts for each elements the number of squared required until a lookup is possible. At the end it is possible to output for each field a selection of tables, which minimize the number of required squares. Note that using more than three tables also reduces the number of necessary squares at the expense of a higher memory consumption. Once we found the exponential representation, we have to revert the squares by taking the same number of square roots. But in the exponential representation, taking square roots is easy as already shown in Section 4.1.2. This square roots are called Correcting Square Roots (CSR).

The same approach can be used to generated partial tables for the mapping exponential to polynomial representation. Take square roots (called Search Square roots (SSR)) of elements in exponential representation until one suitable pattern for a lookup is found. Back in polynomial representation, correct the square roots by taking the same number of squares (Correcting Squares (CS)).

We evaluated this method in terms of memory consumption and timing on an AVR microcontroller against the polynomial and full table method from Section 4.1.1 and 4.1.2 for fields size from $\mathbb{F}_{2^9}$ to $\mathbb{F}_{2^{15}}$. To allow more flexibility we also evaluated the combination of the partial lookup only for the polynomial to exponential (called Part.Tab.Log in the figures below) lookup,
the exponential to polynomial (Part.Tab.Alog) or both (Part.Tab.Both). The respective other lookup uses the full table method from Section 4.1.2. This way, a developer can decide how much memory he or she is willing to spend to achieve a certain speed. Figure 4.1 shows the performance results and Table 4.1 the memory consumption of the proposed method compared to the two classical ones. Each block labelled with the same method is sorted from top to bottom from $F_{2^{15}}$ down to $F_{2^9}$.

These figures clearly show that for the multiplication and squaring operation the new arithmetic is always slower. But for the more complex operations (exponentiation, inversion, division and taking square roots) in fields larger than $GF(2^{11})$, partial lookup tables are faster than traditional polynomial arithmetic, while at the same time consuming less memory as full table lookups.

A further option that has to be explored is the use of a normal basis representation. In a normal basis, squaring is just a cyclic shift of the base elements, thereby speeding up the modification operation required in the new method.
Figure 4.1: Evaluation of the Partial Lookup Table Arithmetic
### 4.2. A New Approach: Partial Lookup Tables

<table>
<thead>
<tr>
<th>Field.Method</th>
<th>Code</th>
<th>Data</th>
<th>Sum of Memory</th>
<th>Maximum Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF(2^09).Poly.</td>
<td>1180 Bytes</td>
<td>0 Bytes</td>
<td>1180 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^09).Tab.</td>
<td>1524 Bytes</td>
<td>2048 Bytes</td>
<td>3572 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^09).Part.Tab.Alog</td>
<td>3900 Bytes</td>
<td>1408 Bytes</td>
<td>5308 Bytes</td>
<td>8</td>
</tr>
<tr>
<td>GF(2^09).Part.Tab.Log</td>
<td>4934 Bytes</td>
<td>1408 Bytes</td>
<td>6342 Bytes</td>
<td>8</td>
</tr>
<tr>
<td>GF(2^09).Part.Tab.Both</td>
<td>7346 Bytes</td>
<td>768 Bytes</td>
<td>8114 Bytes</td>
<td>16</td>
</tr>
<tr>
<td>GF(2^10).Poly.</td>
<td>1280 Bytes</td>
<td>0 Bytes</td>
<td>1280 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^10).Tab.</td>
<td>1616 Bytes</td>
<td>4096 Bytes</td>
<td>5712 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^10).Part.Tab.Alog</td>
<td>4304 Bytes</td>
<td>2816 Bytes</td>
<td>7102 Bytes</td>
<td>9</td>
</tr>
<tr>
<td>GF(2^10).Part.Tab.Log</td>
<td>5424 Bytes</td>
<td>2816 Bytes</td>
<td>8240 Bytes</td>
<td>9</td>
</tr>
<tr>
<td>GF(2^10).Part.Tab.Both</td>
<td>8148 Bytes</td>
<td>1536 Bytes</td>
<td>9684 Bytes</td>
<td>18</td>
</tr>
<tr>
<td>GF(2^11).Poly.</td>
<td>1340 Bytes</td>
<td>0 Bytes</td>
<td>1340 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^11).Tab.</td>
<td>1706 Bytes</td>
<td>8192 Bytes</td>
<td>9898 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^11).Part.Tab.Alog</td>
<td>4346 Bytes</td>
<td>5632 Bytes</td>
<td>9978 Bytes</td>
<td>10</td>
</tr>
<tr>
<td>GF(2^11).Part.Tab.Log</td>
<td>5514 Bytes</td>
<td>5632 Bytes</td>
<td>11146 Bytes</td>
<td>10</td>
</tr>
<tr>
<td>GF(2^11).Part.Tab.Both</td>
<td>8190 Bytes</td>
<td>3072 Bytes</td>
<td>11262 Bytes</td>
<td>20</td>
</tr>
<tr>
<td>GF(2^12).Poly.</td>
<td>1422 Bytes</td>
<td>0 Bytes</td>
<td>1422 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^12).Tab.</td>
<td>1798 Bytes</td>
<td>16384 Bytes</td>
<td>18182 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^12).Part.Tab.Alog</td>
<td>4642 Bytes</td>
<td>11264 Bytes</td>
<td>15906 Bytes</td>
<td>11</td>
</tr>
<tr>
<td>GF(2^12).Part.Tab.Log</td>
<td>5878 Bytes</td>
<td>11264 Bytes</td>
<td>17142 Bytes</td>
<td>9</td>
</tr>
<tr>
<td>GF(2^12).Part.Tab.Both</td>
<td>8758 Bytes</td>
<td>6144 Bytes</td>
<td>14902 Bytes</td>
<td>20</td>
</tr>
<tr>
<td>GF(2^13).Poly.</td>
<td>1502 Bytes</td>
<td>0 Bytes</td>
<td>1502 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^13).Tab.</td>
<td>1890 Bytes</td>
<td>32768 Bytes</td>
<td>34658 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^13).Part.Tab.Alog</td>
<td>4926 Bytes</td>
<td>22528 Bytes</td>
<td>27454 Bytes</td>
<td>12</td>
</tr>
<tr>
<td>GF(2^13).Part.Tab.Log</td>
<td>6226 Bytes</td>
<td>22528 Bytes</td>
<td>28754 Bytes</td>
<td>12</td>
</tr>
<tr>
<td>GF(2^13).Part.Tab.Both</td>
<td>9286 Bytes</td>
<td>12288 Bytes</td>
<td>21574 Bytes</td>
<td>24</td>
</tr>
<tr>
<td>GF(2^14).Poly.</td>
<td>1584 Bytes</td>
<td>0 Bytes</td>
<td>1584 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^14).Tab.</td>
<td>2066 Bytes</td>
<td>65536 Bytes</td>
<td>67602 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^14).Part.Tab.Alog</td>
<td>5258 Bytes</td>
<td>45056 Bytes</td>
<td>50314 Bytes</td>
<td>13</td>
</tr>
<tr>
<td>GF(2^14).Part.Tab.Log</td>
<td>6226 Bytes</td>
<td>45056 Bytes</td>
<td>51682 Bytes</td>
<td>13</td>
</tr>
<tr>
<td>GF(2^14).Part.Tab.Both</td>
<td>9902 Bytes</td>
<td>24576 Bytes</td>
<td>34478 Bytes</td>
<td>26</td>
</tr>
<tr>
<td>GF(2^15).Poly.</td>
<td>1664 Bytes</td>
<td>0 Bytes</td>
<td>1664 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^15).Tab.</td>
<td>2186 Bytes</td>
<td>131072 Bytes</td>
<td>133258 Bytes</td>
<td>0</td>
</tr>
<tr>
<td>GF(2^15).Part.Tab.Alog</td>
<td>5714 Bytes</td>
<td>90112 Bytes</td>
<td>95826 Bytes</td>
<td>14</td>
</tr>
<tr>
<td>GF(2^15).Part.Tab.Log</td>
<td>7192 Bytes</td>
<td>90112 Bytes</td>
<td>97304 Bytes</td>
<td>6</td>
</tr>
<tr>
<td>GF(2^15).Part.Tab.Both</td>
<td>10742 Bytes</td>
<td>49152 Bytes</td>
<td>59894 Bytes</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of Memory for Different Methods over GF(2^9) up to GF(2^15)
Chapter 5

Attacking Classical Schemes using Quantum Computers

This chapter gives a brief overview over the theory of quantum computing and the algorithms solving the discrete logarithm and factoring problem. Additionally it presents Grover’s algorithm, which is a quantum search algorithm lowering the brute force complexity of all cryptographic algorithms. It is not meant to be a in depth tutorial, but should provide an orientation how far quantum algorithms have evolved.

5.1 Quantum Computing

Quantum computation differs greatly from classical bit computation and thus the mathematics for quantum computing is different. The smallest information unit of a quantum computer is a qubit which can be in a base state, 1 or 0, or somewhere between these base states, which is a superposition of these states. A quantum system with more than one qubit is called a quantum register. Classical memories with \( n \) bits have a state dimension of \( n \), but a \( n \)-qubit system has a state dimension of \( 2^n \). As mentioned in [SS04] quantum computers can be designed to execute the same tasks with the same algorithms as classical computers, but the time for the execution is roughly the same. If algorithms use the specific properties of quantum mechanics, the quantum systems can outperform classical computers. Quantum computation can "see" all \( 2^n \) states and apply operations on them simultaneously. This feature is called quantum parallelism. One can not access all \( 2^n \) states but one has to measure the quantum system, i.e., one gets a random base state out of the superposition. The goal of quantum algorithms is to increase the probability of one desired base state which is the solution to a given problem.

5.1.1 Mathematical Definition of Qubits and Quantum Register

Compared to classical bits, one qubit can be in an arbitrary linear combination of the states 0 or 1 (see [SS04]). For a more comprehensive mathematical definition of qubit and multi qubits (quantum registers) see [Sturm2009]. The states 0 and 1 are the base states of a single quantum system and are conventionally described as the two dimensional vectors

\[
0 \equiv |0\rangle := \begin{pmatrix} 1 \\ 0 \end{pmatrix}
\]  

(5.1.1)
and

\[ |1⟩ = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \]  

(5.1.2)

Since the state of a single qubit can also be a superposition of the base states, and thus a linear combination of these base states, it can be described as

\[ ψ = λ_0 |1⟩ + λ_1 |⟩, \quad λ_j ∈ \mathbb{C} \]  

(5.1.3)

Figure 5.1 depicts the base states and one possible superposition of these states for a single qubit. One qubit is mathematically a normalized vector and we take into account that

\[ λ_0^2 + λ_1^2 = 1 \]  

(5.1.4)

Figure 5.1: States of a single qubit

In this example the amplitudes in the superposition \( λ_0 |1⟩ + λ_1 |⟩ \) for both base states equal \( \frac{1}{\sqrt{2}} \) and the probabilities are \( \frac{1}{2} \). The base states are orthogonal to each other. A system with more than one qubits encodes information in a quantum register. For example a 4-qubit register with the bit information 0101 can be visualized as

\[ |0⟩|1⟩|0⟩|1⟩ = |0101⟩ = |5⟩_4 \]  

(5.1.5)

The lower index of a register is the number of qubits and the register content can be depicted as a binary or a decimal number. Mathematically one can describe a \( n \)-qubit register as the
canonical tensor product of \( n \) two dimensional qubit vectors. The tensor product of \( n \) two dimensional vectors yields a \( 2^n \) dimensional vector. We denote the tensor product operator as \( \otimes \).

\[
|x\rangle_n := |x_1\rangle \otimes |x_2\rangle \otimes \ldots \otimes |x_n\rangle, x \in \{0, 1, \ldots, 2^n - 1\}, x_i \in \{0, 1\}
\] (5.1.6)

Since a qubit register can be somewhere between the states \( |0\ldots 0\rangle \) and \( |1\ldots 1\rangle \) it can be described as a linear combination of the base states as shown in the next example of a 2-qubit register.

\[
\psi = \lambda_0 |00\rangle + \lambda_1 |01\rangle + \lambda_2 |10\rangle + \lambda_3 |11\rangle
\] (5.1.7)

\[
= \lambda_0 \left( \begin{array}{c} 1 \\ 0 \end{array} \right) \otimes \left( \begin{array}{c} 1 \\ 0 \end{array} \right) + \lambda_1 \left( \begin{array}{c} 1 \\ 0 \end{array} \right) \otimes \left( \begin{array}{c} 0 \\ 1 \end{array} \right) + \lambda_2 \left( \begin{array}{c} 0 \\ 1 \end{array} \right) \otimes \left( \begin{array}{c} 1 \\ 0 \end{array} \right) + \lambda_3 \left( \begin{array}{c} 0 \\ 1 \end{array} \right) \otimes \left( \begin{array}{c} 0 \\ 1 \end{array} \right)
\] (5.1.8)

\[
= \lambda_0 \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \lambda_1 \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} + \lambda_2 \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \lambda_3 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}
\] (5.1.9)

A \( n \)-qubit quantum register represents the state of \( 2^n \) states at the same time, if in a superposition state, while \( \lambda_i \) is the amplitude and \( |\lambda_i|^2 \) describes the probability of the register to be in the state \( i \). Before quantum computation starts, the register is in a well defined state, e.g., \( |0\ldots 0\rangle \) or \( |0\ldots 1\rangle \). After applying operations (see Section 5.1.2) on the register, the result is a linear combination of the base states. After measuring the probability distributions a final result state will be determined.

### 5.1.2 Operations on Qubits and Quantum Registers

In quantum mechanics unitary operations, also called gates, are used. Gates are unitary transformation matrices which are applied on a qubit or a qubit register. The inverse for a gate is the conjugated-transposed gate itself. The gates described here are real and symmetric matrices, such that the gates are inversions to itself. A small subset of all available gates for quantum computing is introduced in this section. For more information on gates and their mathematical definitions see [SS09, SS04].

The \( X \)-gate is the Not-Gate for a single qubit and is defined as

\[
X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \Rightarrow X|0\rangle = |1\rangle, X|1\rangle = |0\rangle.
\] (5.1.11)

Another elementary gate is the \( H \)-gate (Hadamard-Gate) which is used to transform a well defined state, either 1 or 0, with a probability of \( \frac{1}{2} \) to be either the state 1 or 0. The idea
behind the $H$-gate is that, after applying it on a single qubit, the qubit is in a superposition of the possible states.

$$H := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$ \tag{5.1.12}

For a $n$-qubit system the state space is $2^n$. Therefore, to apply operations on the whole state space, $2^n \times 2^n$ operation matrices are needed. One can create these matrices out of the elementary gates for one qubit systems, see [SS09] for more details. There exist a few of elementary gates for two qubits. The most famous example is the $C$-gate, the $CNOT$-gate (Controlled-Not-Gate). The $C_{10}$-gate negates the right qubit if the left qubit is in the state 1. The mathematical definition is

$$C_{10} := \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad C_{10}|x\rangle|y\rangle = |x\rangle|y \oplus x\rangle$$ \tag{5.1.13}

while $x, y \in \{0, 1\}$ and $\oplus$ is a XOR operation. We also define a gate for boolean functions, the $U_f$-gate. Considering a function

$$f : \{0, 1, \ldots, 2^n - 1\} \rightarrow \{0, 1\}$$ \tag{5.1.14}

so that the $U_f$-gate operates in a $n + 1$ qubit register. Thus the definition of the gate is

$$U_f|x\rangle_n|y\rangle := |x\rangle_n|y \oplus f(x)\rangle, \forall x \in \{0, 1, \ldots, 2^n - 1\}, y \in \{0, 1\}.$$ \tag{5.1.15}

This gate is mapped to a boolean function and adds the result of this function to the first (from the right) qubit with an XOR operation.

5.2 Grover’s Algorithm: A Quantum Search Algorithm

A quantum search algorithm was introduced by Lov K. Grover in 1996 (see [Gro96, Gro97]). The complexity of this algorithm is in $O(\sqrt{N})$. It can be used to search an element, which satisfies a specific condition, in an unsorted set of elements. More than one elements can satisfy the given condition and the algorithm retrieves one of them. We focus on a set of elements where only one element satisfies the condition.

5.2.1 Attacking Cryptographic Schemes

If all elements have a bit length of $n$ bits, there exist $N = 2^n$ different elements. The algorithm works with the superposition of all $N$ elements, also call states, and uses a so called oracle function to evaluate if the condition for a given state is satisfied. We have the set $\{s|s \in \{0, 1, \ldots, 2^n - 1\}\}$ with $N$ states and the oracle function

$$f : \{0, 1, \ldots, 2^n - 1\} \rightarrow \{0, 1\}$$ \tag{5.2.1}
5.2. Grover’s Algorithm: A Quantum Search Algorithm

which is defined as

\[
    f(s) = \begin{cases} 
        1, & \text{if } s \text{ satisfies the condition} \\
        0, & \text{otherwise} 
    \end{cases} \quad (5.2.2)
\]

The algorithm iterates through different superpositions of the states in \( O(\sqrt{N}) \) steps and retrieves with a high probability one state which satisfies the condition. The concrete procedure is explained in the next chapter. Now we apply this algorithm to attack a symmetric cryptosystem. Symmetric ciphers use a secret key \( k \) to encrypt the plaintext \( x \) into the ciphertext \( y \). Lets assume we have a encryption function \( enc \) which labels a symmetric cipher, e.g., the Data Encryption Standard (DES) or the Advanced Encryption Standard (AES).

\[
y = enc_k(x), \text{ } k \text{ has a fixed size of } n \text{ bits} \quad (5.2.3)
\]

If we want to break this cipher and gain the secret key \( k \) with a known pair of plaintext and ciphertext \( (x, y) \) we need, in the worst case, to test all possible keys to find one which satisfies \( y = enc_k(x) \). That means it requires \( O(2^n) = O(N) \) steps. Grover’s algorithm finds an element which satisfies a condition in just \( O(\sqrt{N}) \) steps. If we want to break a cryptosystem with a key size of \( n \) bits with the help of Grover’s algorithm, \( O(\sqrt{2^n}) = O(2^{\frac{n}{2}}) \) steps are required, which halves the security and thus halves the key size of the cryptosystem. Figure 5.2 depicts the design of the oracle function which can be used to gain the secret key. The encryption function may be costly and may be any encryption function, but it is executed only \( O(2^{\frac{n}{2}}) \) times. This function encrypts the given plaintext with a passed key and compares the cipher text to the true cipher text of the given plaintext. The result of this function is boolean.

We can use Grover’s algorithm to attack hash functions. Hash functions have a fixed output length and an arbitrary input length and are used to compress huge data into a fingerprint with a fixed size. These fingerprints can be used for signatures. If we have a given pair of a plain message and the digital signature and we want to change the message but keep the signature valid, we need to create a message which has the same fingerprint as the original message. That means we need to find a collision. Since the input length is arbitrary and the output size is fixed, a collision must exist due to the pigeon hole principle (see [PP09]). If we have a hash
function with the output size \( n \)-bits, thus \( 2^n \) outputs exist, and we compute the fingerprint for \( 2^n + 1 \) messages, at least two messages have the same fingerprint. So the security of a hash algorithm depends on its output length. We define a function \( h \) which can be any compressing function or hash function, e.g., the Secure Hash Algorithm (SHA).

\[
z = h(x), \ z \text{ has a fixed size of } n \text{bits} \quad (5.2.4)
\]

For a given message \( x \) and the computed hash value \( z = h(x) \), we need to retrieve a state \( x' \neq x \) which satisfies the condition \( z = h(x') \). We can set the state space to \( n + m \) or even \( n \) if the hash function has weak dispersal. The algorithm retrieves a state which satisfies the condition in \( O(\sqrt{2^{n+m}}) = O(2^{\frac{n+m}{2}}) \) time steps and if \( n \gg m \) for asymptotical reasons the time complexity is just \( O(\sqrt{2^n}) = O(2^{\frac{n}{2}}) \) Figure 5.3 depicts the design of the oracle function \( f \). This function compares a passed message \( x' \) and computes the hash value. It compares the computed hash value with a given hash value and returns true if they are both equal.

**5.2.2 Formulation of the Process**

In this section we focus on the algorithm itself. The algorithm operates in a \( n + 1 \) quantum register to find an element out of \( 2^n \) which satisfies a condition. At the beginning all possible states have the same amplitude, thus the same probability, and the algorithm increases the amplitude of the desired state in each iteration by \( O(\frac{1}{\sqrt{N}}) \)[Gro96]. Executing the iteration \( O(\sqrt{N}) \) times results in an amplitude of the desired state to be \( O(1) \). The exact number of iterations can be calculated with

\[
k_0 = \left\lfloor \frac{\pi}{4} \sqrt{N} \right\rfloor \quad (5.2.5)
\]

How the above equation is retrieved is explained in [LMP03] (a good paper to understand Grover’s algorithm). The following pseudo code in Alg. 2 gives the formulation of the algorithm
and each step is explained in detail (following [LMP03]). Let \( \psi \) be the \( n + 1 \) qubit register. The register is divided into two register parts. At initialization the first part \( |\psi\rangle \) is the superposition of all \( 2^n \) possible states gained with the \( H^{(n)} \)-gate, which is a \( 2^n \times 2^n \) Hadamard matrix. The second part \( |−\rangle \) is a convenience notation for \( H|1\rangle \).

### Algorithm 2 Grover’s Search Algorithm

**Input:** \( |\psi\rangle := H^{(n)}|0\rangle_n, |−\rangle := H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \)

**Output:** private key \( K_{sec} \), public key \( K_{pub} \)

1: \( \nu_1 := |\psi\rangle |−\rangle \)
2: for \( r = 1, \ldots, k_0 \) do
3: \( \nu_{r+1} := U_f(\nu_r) \)
4: \( \nu_{r+1} = ((2|\psi\rangle\langle\psi| − |I||\psi\rangle)|−\rangle) \)
5: end for
6: return Measure of first \( n \) qubits

After initialization all possible states have the same amplitude, i.e., the same probability. The loop is the heart of the algorithm and its goal is to increase the amplitude of the desired state in the first \( n \) qubits.

Step 3 inside the loop applies the \( U_f \)-Gate on the register and \( f \) is the oracle function. After we apply the \( U_f \)-Gate the register \( \nu_{r+1} \) is updated. The first register part \( 2|\psi\rangle \) can be described as

\[
|\psi\rangle = \frac{1}{\sqrt{N}}|0, \ldots, 0\rangle_n + \cdots + \frac{1}{\sqrt{N}}|1, \ldots, 1\rangle_n = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle.
\] (5.2.6)

That means all possible (base) states have the same probability \( \lambda_1^2 = \left(\frac{1}{\sqrt{N}}\right)^2 = \frac{1}{N} \). The whole register is

\[
\nu_1 = \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle\right) |−\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle |−\rangle
\] (5.2.7)

Applying the \( U_f \)-Gate on \( \nu \) means applying it on all base states of the superposition, thus

\[
U_f(\nu_1) = U_f \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle |−\rangle\right) = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} U_f(|i\rangle |−\rangle) \] (5.2.8)

Now we take a closer look at each term of the sum.

\[
U_f(|i\rangle |−\rangle) = U_f \left(|i\rangle \left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right)\right) = \frac{U_f(|i\rangle |0\rangle) - U_f(|i\rangle |1\rangle)}{\sqrt{2}}
\] (5.2.9)

If we insert the definition of the \( U_f \)-Gate from Equation 5.1.15, we get

\[
\frac{U_f(|i\rangle |0\rangle) - U_f(|i\rangle |1\rangle)}{\sqrt{2}} = \frac{|i\rangle |0\rangle + f(i)\rangle - |i\rangle |1\rangle + f(i)\rangle}{\sqrt{2}} = (-1)^{f(i)} \left(\frac{|i\rangle |0\rangle - |i\rangle |1\rangle}{\sqrt{2}}\right) = (-1)^{f(i)} |i\rangle |−\rangle
\] (5.2.10)

Inserting the gained result in the sum of Equation 5.2.8 gives us

\[
\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} U_f(|i\rangle |−\rangle) = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} (-1)^{f(i)} |i\rangle |−\rangle
\] (5.2.11)
In Equation 5.2.11 we see that the amplitudes for all base states are the same, but for the desired state $i$, such that $f(i) = 1$ if $i$ is the state we search for and $f(i) = 0$ else, the amplitude is inversed(!). Since we assume we search for an item (e.g., a secret key), such that only one state satisfies our oracle function, only one base state has a reversed amplitude. The second part $|−⟩$ of our register $υ$ has the purpose to inverse the amplitude of our desired state. Due to the quantum parallelism, the systems ”sees” all possible states and distinguishes our desired state from the others in one single iteration. The goal is now to increase the amplitude of the desired state. After applying the $U_f$-Gate on $υ_1$ we get

$$υ′_2 = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} (-1)^{f(i)} |i⟩|−⟩ = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i⟩ - \frac{2}{\sqrt{N}} |i_0⟩ = |υ_2⟩|−⟩$$

(5.2.12)

and $i_0$ is our desired state.

Step 4 applies an operation on the first part $|ψ_r⟩$ of our register $υ_r$, which is called an inversion about the mean, the second part $|−⟩$ remains unchanged. The operation $2|ψ⟩⟨ψ|−I$ is a householder transformation matrix and mirrors the superposition state $|ψ_r⟩$ about the hyperplane $|ψ⟩$. We apply this operator on our previously calculated state $|ψ′_2⟩$ to gain the final state $|ψ_2⟩$ of this algorithm iteration.

$$|ψ_2⟩ = 2|ψ⟩⟨ψ|−I|ψ′_2⟩ = 2^{n-2} - \frac{1}{2^{n-2}} |ψ⟩ + \frac{2}{\sqrt{N}} |i_0⟩$$

(5.2.13)

The result of Equation 5.2.13 was taken from [LMP03]. After applying the inversion about the mean, the amplitudes for the superposition state $|ψ⟩$ are decreased. Thus the amplitude for our desired state $i_0$ is increased. $|ψ′_2⟩$ changed from

$$\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i⟩ - \frac{2}{\sqrt{N}} |i_0⟩ = |ψ⟩ - \frac{2}{\sqrt{N}} |i_0⟩$$

(5.2.14)

to

$$\frac{2^{n-2} - 1}{2^{n-2}} |ψ⟩ + \frac{2}{\sqrt{N}} |i_0⟩$$

(5.2.15)

In each iteration the amplitude for $|i_0⟩$ becomes bigger and bigger. To calculate the inversion about the average for one amplitude $λ_i$ individually, we can use $λ_i = (λ_{average} + (λ_i))$, while $λ_{average}$ is the average of all amplitudes. Figure 5.4 depicts the operation for four amplitudes.

Step 6: - the final step - We measure the first $n$ qubits. That means, the systems randomly collapses to one of the base states. The base state with the highest amplitude is the most likely, thus we get the correct result with a high probability. According to Grover [Gro96] we get our desired state with a probability of at least $\frac{1}{2} = O(1)$.

5.3 Shor’s Algorithm: Factoring and Discrete Logarithm

Shor introduced methods, which makes use of quantum mechanics, to solve the discrete logarithm and prime factoring problems [Sho94, Sho97]. The discussed methods have a polynomial
5.3. Shor’s Algorithm: Factoring and Discrete Logarithm

Figure 5.4: Inversion about the mean (based on [Gro97])

time complexity in the bit size, for high bit sizes the problems are infeasible with classical computers. We focus on factoring in this chapter.

5.3.1 Quantum Fourier Transform

The quantum Fourier transform is the heart of Shor’s methods. It is a unitary operation on a qubit register with \( n + 1 \) qubits, thus \( N = 2^n \) states are possible, it is applied on a base state and is described as

\[
F \otimes^n |x\rangle_n := \frac{1}{\sqrt{N}} \sum_{j=0}^{2^n-1} \exp\left(\frac{2\pi i j x}{N}\right) |j\rangle_n, x \in \{0, \ldots, N.1\} \tag{5.3.1}
\]

\( F \otimes^n \) is a \( 2^n \times 2^n \) unitary matrix and \( i \) is the imaginary part. Shor mentions that applying this operation requires polynomial amount of time steps, in bit size \( n \)[Sho97]. This operation is essentially a discrete Fourier transform. The transform is a unitary complex matrix and \( F^{-1} = F^* \), where \( F^* \) is the conjugated transposed matrix of \( F \). Essentially the inverse is the same as the original matrix, just with a ”-“ in the exponent of the exp function. Figure 5.5 shows the complex plane after we apply the transform on \(|3\rangle\). The plane is symmetrically spanned with vectors on the unit cycle.

5.3.2 Factoring and RSA

Assume we have a large number \( n \), we can factor this number in \( k \) primes such \( n = \prod_{i=0}^{k} p_i^{q_i} \).
Prime factoring is a hard problem for large numbers and is used as a one way function for the RSA cryptosystem. In the RSA cryptosystem we have a given public key \( k_{pub} = (n, e) \) and to encrypt and decrypt messages we do calculations modulo \( n \). The encryption function is defined as

\[
y \equiv x^e (\mod n) \tag{5.3.2}
\]

If we apply this rule in the decryption function of RSA we get

\[
x \equiv y^d \equiv x^{ed} \equiv x^{ed \mod \Phi(n)} \equiv x^1 (\mod n) \tag{5.3.3}
\]

Thus \( d \) must be the multiplicative inversion of \( e \mod \Phi(n) \). When choosing the public key \( e \) we have to ensure that \( gcd(e, n) = 1 \) as only then an inverse exist for \( e \). Computing the
inversion of \( e \mod \Phi(n) \) can be done with the efficient Extended Euclidean Algorithm. So when an attacker wants to break RSA, he needs the prime factorization of \( n \) to compute the secret key \( d \). In practice the bit size for the number \( n \) is about 1024 bits to make factoring infeasible for computers. The public number \( n \) contains of two prime numbers \( p \) and \( q \) such that \( n = pq \) and the Euler Phi function \( \Phi(n) = (p - 1)(q - 1) \). Thus we are interested in prime factors which are odd and are not powers of a prime. An even number \( n \) would yield the prime 2, or a power of it, to be one prime factor of \( n \) and make the prime factoring easier. For more detailed information about RSA see [PP09]. The most efficient algorithms for classical computers have an exponential time complexity, which is quite slow for large bit size \( n \). Grover’s method takes \( O((\log n)^2(\log \log n)(\log \log \log n)) \) [Sho97] time steps which polynomial complexity quite good.

5.3.3 Factoring with Shor

As mentioned in the previous section, we are interested in factoring an integer. From now on we use the notation \( N \) for the number we want to factor and \( n \) is the bit size of \( N \) \((n = \lceil \log_2 N \rceil)\). We are interested in finding the prime factors of \( N \) such that \( N = pq \) and \( p \) and \( q \) are odd prime numbers (not powers of a prime). Shor’s method gives us a result in \( O((\log N)^2(\log \log N)(\log \log \log N)) \) [Sho97] time steps.

Probabilistic non-quantum Part

Shor’s method uses a probabilistic approach and reduces the problem of factoring a prime to the problem of calculating the order for a number \( x < N \). The following description is based on
Computing the order for a random number \( x \) means gaining the smallest \( r \) such that \( x^r \equiv 1 \mod N \). Solving this problem can not be done efficient with classical computers (maybe not yet), but with quantum computing. If we take a random number \( x < N \) and calculate \( \gcd(x, N) \) we either get \( \gcd(x, N) > 1 \), thus we get a common divisor which includes factors of \( N \), or we get \( \gcd(x, N) = 1 \). In the second case we can calculate the order of \( x \) such that \( x^r \equiv 1 \mod N \). Now if \( r \) is an even number we can set
\[
x^\frac{r}{2} \equiv y \mod N
\] (5.3.4)
Notice that \( y^2 \equiv 1 \mod N \) so that we can set
\[
y^2 - 1 \equiv 0 \mod N \equiv (y - 1)(y + 1) \equiv 0 \mod N
\] (5.3.5)
This results in \((y - 1)(y - 1)\) being divisible by \( N \). \( N \) cannot divide \( y - 1 \) and \( y + 1 \) separately if \( 1 < y < N - 1 \) and thus we gain the two prime factors \( p = \gcd(y - 1, N) \) and \( q = \gcd(y + 1, N) \) if \( 0 < y - 1 < y + 1 < N \) [LMP03]. If \( \gcd(x, N) > 1 \) we got a factor of \( N \) since \( x \) and \( N \) have common factors. For the case \( \gcd(x, N) = 1 \) two conditions have to be satisfied:

1. order \( r \) of \( x \) must be even
2. \( 0 < y - 1 < y + 1 < N \)

If one condition is not satisfied we need to find another random \( x \) to proceed. This method fails if \( N \) is a power of an odd prime [Sho97, LMP03] but other efficient methods exist for this case.

Let us try to factor the number \( N = 15 \) with this new method. We get the number \( x = 7 \) and know that \( \gcd(7, 15) = 1 \), so we need to check if the conditions are satisfied. Calculating the order by hand leads us to \( 7^4 \equiv 13 \cdot 7 \equiv 1 \mod N \). We know the order \( r = 4 \) is even, thus we gain \( y \equiv 4 \equiv 7^{\frac{r}{2}} \mod N \). The condition \( 1 < y - 1 < y + 1 < N \) is also satisfied. By calculating \( \gcd(y - 1, N) = \gcd(3, 15) = 3 = p \) and \( \gcd(y + 1, N) = \gcd(5, 15) = 5 = q \) we gained the prime factorization \( 15 = 3 \cdot 5 = p \cdot q \).

The probability that our randomly chosen \( x \) yields a factor of \( N \) is \( 1 - \frac{1}{2^{k-1}} \), where \( k \) is the number of primes in \( N \) [Sho97, LMP03]. So in our case, the probability is \( 1 - \frac{1}{2} = \frac{1}{2} \) and a higher number of factors would increase our chance of hitting the right candidate.

**Quantum Part**

The goal of Shor’s quantum method is to efficiently calculate the order of a number coprime to \( N \). The following description of the method is based on [LMP03]. We need a qubit register, separated into two registers, the first one has \( t \) qubits, such that \( N^2 \leq 2^t < 2N^2 \), and the second register has \( n \) qubits.

\[
|\psi_0\rangle = |0, \ldots, 0\rangle_t |0, \ldots, 0\rangle_n
\] (5.3.6)
We put the first register into the superposition state with the \( H \)-Gate.

\[
|\psi_1\rangle = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle_t |0\rangle_n
\] (5.3.7)
We do operations on the second register mod $N$. Assume we have chosen a random $x < N$, such that $\gcd(x, N) = 1$, and we have a gate $V$ which adds a power of $x$ to the second register $V(|j⟩|k⟩ = |j⟩|k⟩ + x^j \pmod{N}$. We use this gate on $|ψ_1⟩$, which operates on all states in the superposition simultaneously:

$$|ψ_2⟩ = V|ψ_1⟩ = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} V(|j⟩|0⟩) = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j⟩|x^j \pmod{N})⟩_n$$  (5.3.8)

Quantum parallelism allows us to calculate all powers of $x$ simultaneously. At quantum levels all powers of $x$ can be "seen" and have the same amplitudes. Since we do modulo computations in the second register we have certain periods in the whole register, i.e., we have states like $|0⟩|1⟩$, $|r⟩|x^r \equiv 1 \pmod{N}⟩$, $|2r⟩|x^{2r} \equiv 1 \pmod{N}⟩$, ... etc., and we can rewrite the state $|ψ_2⟩$ as

$$|ψ_2⟩ = \frac{1}{\sqrt{2^t}}([|0⟩ + |r⟩ + |2r⟩ + \ldots |1⟩$$

$$+ (|1⟩ + |r+1⟩ + |2r+1⟩ + \ldots |x⟩)$$

$$+ (|2⟩ + |r+2⟩ + |2r+2⟩ + \ldots |x^2⟩)$$

$$\vdots$$

$$+ (|r-1⟩ \equiv -1⟩ + |2r-1 \equiv r-1⟩ + \ldots |x^{r-1}⟩ \equiv x^{-1} \pmod{N})$$

Each row has at most $\frac{2^t}{r}$ sum terms and has period $r$, which we want to find out. We apply the quantum Fourier transform, or its inverse, on all base states in the first register.

$$|ψ_3⟩ = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \left( \frac{1}{\sqrt{2^t}} \sum_{j'=0}^{2^t-1} \exp \left( 2\pi i j'j \frac{r}{2^t} \right) |j'⟩_t \right) |x^j \pmod{N})⟩$$  (5.3.9)

The quantum Fourier transform increases probabilities for estimated multiples of $\frac{2^t}{r}$. Figure 5.6 depicts a sketch of the probability distribution in the first register after applying quantum Fourier transform.
5.4. Discrete Logarithm with Shor

In the first register we measure an estimation of a random multiple of \(\frac{2^t}{r}\) and apply the continued fractions algorithm to find \(r\). Consider we have measured the value \(y\) and the continued fractions algorithm for our case has the form

\[
y \equiv \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \ldots + \frac{1}{a_p}}}}
\]  

(5.3.10)

We have to choose the convergence with a denominator smaller than \(N\). The denominator is either \(r\) or a factor \(r'\) of \(r\). In the latter case we need to compute \(xt \equiv x'r'(\mod N)\) and apply the quantum part on \(xt\) recursively to find the remaining factors of \(r\). If we get \(y = 0\) we have to rerun the algorithm again. Once we figured out what the order is, we need to check if the conditions are satisfied and compute \(y \equiv x^r\) to get prime factors \(p = \gcd(y - 1, N)\) and \(q = \gcd(y + 1, N)\).

5.4 Discrete Logarithm with Shor

Solving the discrete logarithm problem means finding an \(r\), such that \(g^r \equiv x (\mod p)\), while \(g\) is a generator and \(p\) is some prime. Discrete logarithm-based cryptosystems, such as Diffie-Hellman Key Establishment and ECC, use large primes \(p\) to be secure [PP09], since solving discrete logarithm is as hard as finding the order of an element \(r\) (previous section). ECC operates with group operations, which include several addition, multiplications and divisions, making each group operation costly to compute. Shor’s quantum method solves the discrete logarithm in polynomial time, the complexity is in \(O((\log N)^3 \log \log(N) \log \log \log(N))\)[Mos08]. The algorithm works in three quantum registers, uses two modular exponentiations and two quantum fourier transforms [Sho97]. For the concrete procedure we refer to Shor’s papers [Sho94, Sho97]. Also we point to a paper which deals with Shor’s discrete logarithm method optimizing for ECC[PZ03].
Part II

Code-based Cryptography


Table of Contents

6 Introduction to Error Correcting Codes
  6.1 Motivation .................................................. 47
  6.2 Existing Implementations .................................. 48
  6.3 Outline ...................................................... 49
  6.4 Error Correcting Codes .................................. 49
  6.5 Construction of Goppa Codes ............................. 55
  6.6 Dyadic Goppa Codes ....................................... 57
  6.7 Quasi-Dyadic Goppa Codes ................................ 60
  6.8 Decoding Algorithms for Goppa Codes .................... 61
  6.9 Extracting Roots of the Error Locator Polynomial .......... 66
  6.10 MDPC-Codes ............................................... 70
  6.11 Decoding MDPC Codes ..................................... 70

7 Cryptosystems Based on Error Correcting Codes
  7.1 Overview .................................................... 71
  7.2 Security Parameters ....................................... 72
  7.3 Classical McEliece Cryptosystem .......................... 73
  7.4 Modern McEliece Cryptosystem ............................. 74
  7.5 Niederreiter Cryptosystem ................................ 77

8 General Security Considerations and New Side-Channel Attacks
  8.1 Overview .................................................... 81
  8.2 Hiding the Structure of the Private Code .................. 82
  8.3 Attacks ..................................................... 84
Table of Contents

8.4 Side Channel Attacks .............................................. 85
8.5 Ciphertext Indistinguishability ................................. 95
8.6 Key Length ......................................................... 96

9 Conversions for CCA2-secure McEliece Variants .......... 99
  9.1 Kobara-Imai-Gamma Conversion ............................... 100
  9.2 Fujisaki-Okamoto Conversion ............................... 103

10 Microcontroller and FPGA Implementation of Codebased Crypto Using Plain Binary Goppa Codes .......... 107
  10.1 Previous Work .................................................. 107
  10.2 Security Parameters ......................................... 108
  10.3 8-Bit Microcontroller Implementation ..................... 108
  10.4 FPGA Implementation of the Niederreiter Scheme .......... 117
  10.5 Future Work .................................................. 121

11 Codebased Crypto Using Quasi Dyadic binary Goppa Codes .......... 125
  11.1 Scheme Definition of QD-McEliece ......................... 125
  11.2 Implementational Aspects .................................. 128
  11.3 Results on an 8-Bit Microcontroller ..................... 135
  11.4 Conclusion and Further Research .......................... 138

12 Codebased Crypto Using Quasi Cyclic Medium Density Parity Check Codes .................................. 141
  12.1 McEliece Based on QC-MDPC Codes ......................... 141
  12.2 Security of QC-MDPC ......................................... 142
  12.3 Decoding (QC-)MDPC Codes ................................ 142
  12.4 Implementation on FPGA and Microcontroller .......... 146
  12.5 Results ....................................................... 150
Chapter 6

Introduction to Error Correcting Codes

This chapter provides the necessary background in coding theory. Section 6.4 gives a short overview on basic concepts that are assumed to be known to the reader. The first part 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 6.8 deals with the Patterson and Berlekamp-Massey algorithm that allow efficient decoding certain codes (e.g., alternant codes and in case of Patterson, binary Goppa codes). For the root extraction step required in both decoding algorithms, several methods are presented in Section 6.9. Finally, Section 6.10 presents the class of LDPC and MDPC codes and their decoding. Please note, that only the basic variants are presented here and optimizations are discussed in the respective implementation section.

6.1 Motivation

In this part, we concentrate on code-based cryptography. The first code-based public-key cryptosystem was proposed by Robert McEliece in 1978. The McEliece cryptosystem is based on algebraic error-correcting codes, originally Goppa codes. The hardness assumption of the McEliece cryptosystem is that decoding known linear codes is easily performed by an efficient decoding algorithm, but when disguising a linear code as a general linear code by means of several secret transformations, decoding becomes NP-complete. The problem of decoding linear error-correction codes is neither related to the factorization nor to the discrete logarithm problem. Hence, the McEliece scheme is an interesting candidate for post-quantum cryptography, as it is not effected by the computational power of quantum computers.

To achieve acceptance and attention in practice, post-quantum public-key schemes have to be implemented efficiently. Furthermore, the implementations have to perform fast while keeping memory requirements small for security levels comparable to conventional schemes. The McEliece encryption and decryption do not require computationally expensive multiple precision arithmetic. Hence, it is predestined for an implementation on embedded devices.

The main disadvantage of the McEliece public-key cryptosystem is its very large public key of several hundred thousands of bits. For this reason, the McEliece PKC achieved little attention in practice yet. Particularly with regard to bounded memory capabilities of embedded systems,
it is essential to improve the McEliece cryptosystem by finding a way to reduce the public key size. There is ongoing research to replace Goppa codes by other codes having a compact and simple description. For instance, there are proposals based on quasi-cyclic codes [Gab05] and quasi-cyclic low density parity-check codes [BC07]. Unfortunately, all these proposals have been broken by structural attacks [OTD08]. Barreto and Misoczki propose in a recent work [MB09] using Goppa codes in quasi-dyadic form. When constructing a McEliece-type cryptosystem based on quasi-dyadic Goppa codes the public key size is significantly reduced. For instance, for an 80-bit security level, the public key used in the original McEliece scheme is 437.75 Kbytes large. The public key size of the quasi-dyadic variant is 2.5 Kbytes which is a factor 175 smaller compared to the original McEliece PKC. Another disadvantage of the McEliece scheme is that it is not semantically secure. The quasi-dyadic McEliece variant proposed by Barreto and Misoczki is based on systematic coding. It allows to construct CPA and CCA2 secure McEliece variants by using additional conversion schemes such as Kobara-Imai’s specific conversion $\gamma$ [NIKM08].

### 6.2 Existing Implementations

Although proposed more than 30 years ago, code-based encryption schemes have never gained much attention due to their large secret and public keys. It was common perception for quite a long time that due to their expensive memory requirements such schemes are difficult to be integrated in any (cost-driven) real-world products. The original proposal by Robert McEliece for a code-based encryption scheme suggested the use of binary Goppa codes, but in general any other linear code can be used. While other types of codes may have advantages such as a more compact representation, most proposals using different codes were proven less secure (cf. [Min07, OS09]).

The Niederreiter cryptosystem is an independently developed variant of McEliece’s proposal which is proven to be equivalent in terms of security [LDW06]. In 2009, the first FPGA-based implementation of McEliece’s cryptosystem was proposed [EGHP09], targeting a Xilinx Spartan-3AN and encrypts and decrypts data in 1.07 ms and 2.88 ms, using security parameters achieving an equivalence of 80-bit symmetric security. The authors of [SWM+09] presented another accelerator for McEliece encryption over binary Goppa codes on a more powerful Virtex5-LX110T, capable of encrypting and decrypting a block in 0.5 ms and 1.4 ms providing a similar level of security. The latest publication [GDUV12] based on hardware/software co-design on an Spartan3-1400AN decrypts a block in 1 ms at 92 MHz at the same level of security. For x86-based platforms, a recent implementation of the McEliece scheme over binary Goppa codes by Biswas and Sendrier [BS08] achieves about 83-bit of equivalent symmetric security according to [BLP08b]. Comparing their implementation to other public-key schemes, it turns out that McEliece encryption can be faster than RSA and NTRU [Be], however, at the cost of larger keys. Many proposals (e.g., [MB09, CHP12]) already tried to address this issue of large keys by replacing the original used binary Goppa codes with (secure) codes that allow more compact representations. However, most of the attempts were broken [FOPT10b] and for the few (still) surviving ones hardly any implementations are available [BCGO09, Hey11].
Note further that most of these works exclusively target the McEliece cryptosystems. To the best of our knowledge, the only published implementation of Niederreiter encryption for embedded systems is an implementation for small 8-bit AVR microcontrollers that can encrypt and decrypt a block in 1.6\text{ms} and 179\text{ms}, respectively [Hey10]. In addition, we are aware of just another implementation of a signature scheme in Java based on Niederreiter’s concept [Pie].

### 6.3 Outline

The remainder of this part is organized as follows. Chapter 6 introduces the basic concepts of coding theory. Chapter 7 presents the basic variants of McEliece’s and Niederreiter’s public-key schemes. Then, the basic security properties are discussed in Chapter 8 and algorithms necessary for CCA2 security are presented in Chapter 9. In further progress we present optimizations and implementations on microcontrollers and FPGAs using plain binary Goppa codes in Chapter 10, using quasi-dyadic Goppa codes on a microcontroller in Chapter 11 and based in MDPC codes on microcontrollers and FPGAs in Chapter 12.

### 6.4 Error Correcting Codes

Error correcting codes were first developed in the late 1940’s, by Hamming [Ham50], Golay [Gol] and Shannon [Sha48]. They were used to detect transmission errors on noisy electrical lines used to transfer telegrams and the first fax messages. They work by adding some structured redundancy to the original messages. If up to a given number of errors (e.g., flipped bits on an electrical line) occur during transmission of a message, the errors can be detected and even corrected on the receiver’s side. There are basically three methods to accomplish this task. Block codes, convolutional codes and interleaving. We will focus on the first method. Block codes got their name because the message must be divided into equal length blocks. Sometime they are referred to as \((n, k)\)-codes, because to a block of \(k\) bit information is coded into a block of \(n\) bits, the codeword. The rule how to encode a message into a codeword, can be described by a \((n \times k)\) matrix, the so called generator matrix. As for all codes presented in this thesis, any linear combination of codewords is also a codeword, these codes are also linear. This is not true for convolutional codes. To check weather a received word is a valid codeword or contains some errors, it is multiplied by a second matrix, the so called parity check matrix. If the result of this multiplications, the so called syndrome, is zero, the received word is error free and therefore a valid codeword. If the syndrome is not zero, it can be used to detect and correct the errors up to a given extend, the so called error correcting capability \(t\) of the code.

The theory behind error correcting codes 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.
6.4.1 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 received 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 6.4.1** Let $F_q$ denote a finite field of $q$ elements and $F_q^n$ a vector space of $n$ tuples over $F_q$. An $[n,k]$-linear code $C$ is a $k$-dimensional vector subspace of $F_q^n$. The vectors $(a_1, a_2, \ldots, a_q) \in C$ are called codewords of $C$.

An important property of a code is the minimum distance between two codewords.

**Definition 6.4.2** The Hamming distance $d(x,y)$ between two vectors $x,y \in F_q^n$ is defined to be the number of positions at which corresponding symbols $x_i, y_i$, $\forall 1 \leq i \leq n$ are different. The Hamming weight $wt(x)$ of a vector $x \in F_q^n$ is defined as Hamming distance $d(x,0)$ between $x$ and the zero-vector.

The minimum distance of a code $C$ is the smallest distance between two distinct vectors in $C$. A code $C$ is called $[n,k,d]$-code if its minimum distance is $d = \min_{x,y \in C} d(x,y)$. The error-correcting capability of an $[n,k,d]$-code is $t = \lceil \frac{d-1}{2} \rceil$.

The two most common ways to represent a code are either the representation by a generator matrix or a parity-check matrix.

**Definition 6.4.3** A matrix $G \in F_q^{k \times n}$ is called generator matrix for an $[n,k]$-code $C$ if its rows form a basis for $C$ such that $C = \{ x \cdot G \mid x \in F_q^k \}$. In general there are many generator matrices for a code. An information set of $C$ is a set of coordinates corresponding to any $k$ linearly independent columns of $G$ while the remaining $n-k$ columns of $G$ form the redundancy set of $C$.

If $G$ is of the form $[I_k|Q]$, where $I_k$ it the $k \times k$ identity matrix, then the first $k$ columns of $G$ form an information set for $C$. Such a generator matrix $G$ is said to be in standard (systematic) form.

**Definition 6.4.4** For any $[n,k]$-code $C$ there exists a matrix $H \in F_q^{n-k \times n}$ with $(n-k)$ independent rows such that $C = \{ y \in F_q^n \mid H \cdot y^T = 0 \}$. Such a matrix $H$ is called parity-check matrix for $C$. In general, there are several possible parity-check matrices for $C$.

If $G$ is in systematic form then $H$ can be easily computed and is of the form $[-Q^T | I_{n-k}]$ where $I_{n-k}$ is the $(n-k) \times (n-k)$ identity matrix.

Since the rows of $H$ are independent, $H$ is a generator matrix for a code $C^\perp$ called dual or orthogonal to $C$. Hence, if $G$ is generator matrix and $H$ parity-check matrix for $C$ then $H$ and $G$ are generator and parity-check matrices, respectively, for $C^\perp$. 

50
6.4. Error Correcting Codes

Definition 6.4.5 The dual of a code $C$ is defined as the $[n, n-k]$-code defined by \( \{ x \in \mathbb{F}_q^n \mid x \cdot y = 0, \forall y \in C \} \) and denoted by $C^\perp$.

Definition 6.4.6 (Codes over finite fields) Let $\mathbb{F}_{p^m}$ denote a vector space of $n$ tuples over $\mathbb{F}_{p^m}$. A $(n,k)$-code $C$ 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.

If we identify a vector $[a_{n-1}, \ldots, a_0]$ as a polynomial $a_{n-1}x^{n-1} + \cdots + a_0x^0$ over $\mathbb{F}_{p^m}$ we can define polynomial codes.

Definition 6.4.7 (Polynomial codes) For a given polynomial $g(x)$ of degree $m$ we define the polynomial code generated by $g(x)$ as the set of all polynomials of degree $n$ with $m \leq n$ that are divisible by $g(x)$.

6.4.2 Punctured and Shortened Codes

There are many possibilities to obtain new codes by modifying other codes. In this section we present two of them: punctured codes and shortened codes. These types of codes are used for the construction of the quasi-dyadic McEliece variant discussed in Chapter 11.

Let $C$ be an $[n, k, d]$-linear code over $\mathbb{F}_q$. A punctured code $C^*$ can be obtained from $C$ by deleting the same coordinate $i$ in each codeword. If $C$ is represented by the generator matrix $G$ then the generator matrix for $C^*$ can be obtained by deleting the $i$-th column of the generator matrix for $C$. The resulting code is an $[n-1, k, d^*]$-linear code with $d^* \geq d-1$ if $d > 1$, $k > 1$ and $C$ has a minimum weight codeword with a nonzero $i$-th coordinate.

It is also possible to puncture a code $C$ on several coordinates. Let $T$ denote a coordinate set of size $s$. The code $C^T$ is obtained from $C$ by deleting components indexed by the set $T$ in each codeword of $C$. The resulting code is an $[n-s, k^*, d^*]$-linear code with dimension $k^* \geq k-s$ and minimum distance $d^* \geq d-s$ by introduction.

Punctured codes are closely related to shortened codes. Consider the code $C$ and a coordinate set $T$ of size $s$. Let $C(T) \subseteq C$ be a subcode of $C$ with codewords which are zero on $T$. A shortened code $C_T$ of length $n-s$ is obtained from $C$ by puncturing the subcode $C(T)$ on the set $T$.

The relationship between shortened and punctured codes is represented by the following theorem.
Theorem 6.4.8 Let $C$ be an $[n, k, d]$-code over $F_q$ and $T$ a set of $s$ coordinates.

1. $(C^\perp)_T = (C^T)^\perp$ and $(C^\perp)^T = (C_T)^\perp$, and
2. if $s < d$ then $C^T$ has dimension $k$ and $(C^\perp)_T$ has dimension $n - s - k$
3. if $s = d$ and $T$ is the set of coordinates where a minimum weight codeword is nonzero, then $C^T$ has dimension $k - 1$ and $(C^\perp)_T$ has dimension $n - d - k + 1$

6.4.3 Subfield Subcodes and Trace Codes

Many codes can be constructed from a field $F_q$, where $q = p^m$ for some prime power $p$ and extension degree $m$, by restricting them to the subfield $F_p$. Note, that any element of $F_q = F_{p^m}$ can be written as a polynomial of degree $m - 1$ over $F_p$. For instance, every entry $h \in F_{p^m}$ of a parity check matrix, can be written as a $m$-dimensional column vector with elements from $F_p$, which represent the coefficients of this polynomial.

Definition 6.4.9 Let $F_p$ be a subfield of the finite field $F_q$ and let $C \subseteq F_q^n$ be a code of length $n$ over $F_q$. A subfield subcode $C_{SUB}$ of $C$ over $F_p$ is the vector space $C \cap F_p^n$. The dimension of a subfield subcode is $\dim(C_{SUB}) \leq \dim(C)$.

Another way to derive a code over $F_p$ from a code over $F_q$ is to use the trace mapping $\text{Tr} : F_q \rightarrow F_p$ which maps an element of $F_q$ to the corresponding element of $F_p$. The trace mapping of an element $u \in F_q$ is defined as $\text{Tr}(u) = (u_0, u_1, \ldots, u_{d-1}) \in F_p^d$, where $F_q = F_p/b(x)$ for some irreducible polynomial $b(x)$ of degree $d$.

Definition 6.4.10 Let $\text{Tr}(a)$ denote the trace of an element $a = (a_0, a_1, \ldots, a_n) \in F_q^n$ such that $\text{Tr}(a) = (\text{Tr}(a_0), \text{Tr}(a_1), \ldots, \text{Tr}(a_n)) \in F_p^n$. A Trace code $C_{TR} = C_{TR}(C) := \{\text{Tr}(c) \mid c \in C\} \subseteq F_p^n$ is a code over $F_p$ obtained from a code $C$ over $F_q$ by the trace construction. The dimension of a Trace code is $\dim(C_{TR}) \leq m \cdot \dim(C)$.

For instance, let $C$ be a code over $F_q$ defined by the parity-check matrix $H \in F_q^{t \times n}$ with elements $h_{i,j} \in F_q = F_p[x]/g(x)$ for some irreducible polynomial $g(x) \in F_p[x]$ of degree $m$.

$$H := \begin{pmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{t-1,0} & h_{t-1,1} & \cdots & h_{t-1,n-1} \end{pmatrix}$$

The elements $h_{i,j} \in F_q$ of $H$ can be represented as polynomials $h_{i,j}(x) = h_{(i,j),m-1} \cdot x^{m-1} + \cdots + h_{(i,j),1} \cdot x + h_{(i,j),0}$ of degree $m - 1$ with coefficients in $F_p$. The trace construction derives from $C$ the Trace code $C_{TR}$ by writing the $F_p$ coefficients of each element $h_{i,j}$ onto $m$ successive rows of a parity-check matrix $H_{C_{TR}} \in F_p^{mt \times n}$ for the Trace code. Consequently, $H_{C_{TR}}$ is the trace parity-check matrix for $C$. 

52
The co Trace parity check matrix $H'_{CTr}$ for $\mathcal{C}$, which is equivalent to $H_{CTr}$ $\in \mathbb{F}_p^{m t \times n}$ by a left permutation, can be obtained from $H$ analogously, by writing the $\mathbb{F}_p$ coefficients of terms of equal degree from all components on a column of $H$ onto successive rows of $H'_{CTr}$.

Subfield subcodes are closely related to Trace codes by the Delsarte-Theorem [Del75].

**Theorem 6.4.11 (Delsarte)** For a code $\mathcal{C}$ over $\mathbb{F}_q$, $(\mathcal{C}_{SUB})^\perp = (\mathcal{C}|_{\mathbb{F}_p})^\perp = Tr(\mathcal{C}^\perp)$.

That means, given an $[n, t]$-code $\mathcal{C}^\perp$ defined by the parity-check matrix $H \in \mathbb{F}_q^{t \times n}$ dual to an $[n, n - t]$-code $\mathcal{C}$ defined by the generator matrix $G \in \mathbb{F}_q^{(n-t) \times n}$ the trace construction can be used to efficiently derive from $\mathcal{C}^\perp$ a subfield subcode defined by the parity-check matrix $H_{SUB} \in \mathbb{F}_p^{d t \times n}$.

**6.4.4 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. 6.1 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 received 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. For every codeword \( c_0x^0 + \cdots + c_{n-1}x^{n-1} \) also \( c_{n-1}x^0 + c_0x^1 + \cdots + c_1x^{n-1} \) is in the code. The cyclic shift corresponds to a multiplication with \( x \mod x^n - 1 \).

Dyadic Codes A code is called dyadic if it admits a parity check matrix in dyadic form.

**Definition 6.4.12** Let \( F_q \) denote a finite field and \( h = (h_0, h_1, \ldots, h_{n-1}) \in F_q \) a sequence of \( F_q \) elements. The dyadic matrix \( \Delta(h) \in F_q^n \) is the symmetric matrix with elements \( \Delta_{ij} = h_i \oplus h_j \). The sequence \( h \) is called signature of \( \Delta(h) \) and coincides with the first row of \( \Delta(h) \). Given \( t > 0 \), \( \Delta(h,t) \) denotes \( \Delta(h) \) truncated to its first \( t \) rows.

When \( n \) is a power of 2 every \( 1 \times 1 \) matrix is a dyadic matrix, and for \( k > 0 \) any \( 2^k \times 2^k \) matrix \( \Delta(h) \) is of the form \( \Delta(h) := \begin{pmatrix} A & B \\ B & A \end{pmatrix} \) where \( A \) and \( B \) are dyadic \( 2^{k-1} \times 2^{k-1} \) matrices.

**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 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 \( d_{\text{min}} = n - k + 1 \).
For some polynomial \( f(z) \in \mathbb{F}_{p^m}[z] \), 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:

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

6.5. Construction of Goppa Codes

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 \( d_{\text{min}} \geq t + 1 \) 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 \( H_{i,j} = \alpha_i^j v_i \).

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

\[
\text{Alt}_{n,k,p}(\mathcal{L}, v) := GRS_{n,k}(\mathcal{L}, V) \cap \mathbb{F}_p^n
\]

Generalized Srivastava codes

Generalized Srivastava (GS) codes [Per11] 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 \( \mathbb{F}_{p^m} \) 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:

\[
\text{Goppa}_{n,k,p}(\mathcal{L}, g) := GRS_{n,k}(\mathcal{L}, g) \cap \mathbb{F}_p^n
\]

The minimum distance of a Goppa code is \( d_{\text{min}} \geq t + 1 \), in case of binary Goppa codes with an irreducible Goppa polynomial even \( d_{\text{min}} \geq 2t + 1 \). Details for constructing and decoding Goppa codes are given in Section 6.5.

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 \( g(z) = z^{2^t} \) [Ber73].

6.5 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.

6.5.1 Binary Goppa Codes

We begin by reiterating the above definition (Section 6.4.4) of Goppa for the case of binary Goppa codes, giving an explicit definition of the main ingredients.

**Definition 6.5.1** Let \( m \) and \( t \) be positive integers and let the Goppa polynomial
\[
    g(z) = \sum_{i=0}^{t} g_i z^i \in \mathbb{F}_{2^m}[z]
\] (6.5.1)
be a monic polynomial of degree \( t \) and let the support
\[
    L = \{ \alpha_0, \ldots, \alpha_{n-1} \} \in \mathbb{F}_{2^m}^n, g(\alpha_j) \neq 0 \quad \forall \quad 0 \leq j \leq n-1
\] (6.5.2)
be a subset of \( n \) distinct elements of \( \mathbb{F}_{2^m} \). For any vector \( \hat{c} = (c_0, \cdots, c_{n-1}) \in \mathbb{F}_{2^m}^n \), we define the syndrome of \( \hat{c} \) as
\[
    S_{\hat{c}}(z) = -\sum_{i=0}^{n-1} \frac{\hat{c}_i}{g(\alpha_i)} \frac{g(z) - g(\alpha_i)}{z - \alpha_i} \mod g(z).
\] (6.5.3)

In continuation of (6.4.3), 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 \( S_c = 0 \):
\[
    \text{Goppa}_{n,k,2}(L, g(z)) := \{ c \in \mathbb{F}_{2^m}^n \mid S_c(z) = \sum_{i=0}^{n-1} \frac{c_i}{z - \alpha_i} \equiv 0 \mod g(z) \}.
\] (6.5.4)

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

Note, that for all \( \alpha_i \in L, g(a_i) \neq 0 \).

6.5.2 Parity Check Matrix of Goppa Codes

According to the definition of a syndrome in (6.5.3), every element \( \hat{c}_i \) of a vector \( \hat{c} = c + e \) is multiplied with
\[
    \frac{g(z) - g(\alpha_i)}{g(\alpha_i) \cdot (z - \alpha_i)}.
\] (6.5.5)
Hence, given a Goppa polynomial \( g(z) = g_s z^s + g_{s-1} z^{s-1} + \cdots + g_0 \), the parity check matrix \( H \) can be constructed as

\[
H = \begin{pmatrix}
\frac{g_s}{g(\alpha_0)} & \frac{g_s}{g(\alpha_1)} & \cdots & \frac{g_s}{g(\alpha_{n-1})} \\
g_{s-1} & \frac{g_s}{g(\alpha_0)} & \cdots & \frac{g_s}{g(\alpha_{n-1})} \\
\vdots & \ddots & \ddots & \vdots \\
g_1 & g_2 & \cdots & g_s
\end{pmatrix}
\]

This can be simplified to

\[
H = \left( \begin{array}{ccc}
\frac{1}{g(\alpha_0)} & \frac{1}{g(\alpha_1)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\
g_1 & \frac{1}{g(\alpha_0)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\
\vdots & \ddots & \ddots & \vdots \\
g_s & g_{s-1} & \cdots & g_1
\end{array} \right)
\]

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 \( L \) need to be swapped. As shown in Definition 6.4.4, the generator matrix \( G \) can be derived from the systematic parity check matrix \( H = (Q|I_{n-k}) \) as \((I_k - Q^T)\).

### 6.6 Dyadic Goppa Codes

In [MB09] Barreto and Misoczki have shown how to build binary Goppa codes which admit a parity-check matrix in dyadic form. The family of dyadic Goppa codes offers the advantage of having a compact and simple description.

If \( G(x) = \prod_{i=0}^{t-1} (x - z_i) \) is a monic polynomial with \( t \) distinct roots all in \( \mathbb{F}_q \), then it is called separable over \( \mathbb{F}_q \). In case of \( q = 2^m \) the Goppa code can also correct \( t \) errors. A Goppa code generated by a separable polynomial over \( \mathbb{F}_q \) admits a parity-check matrix in Cauchy form [MS78].

**Definition 6.6.1** Given two disjoint sequences \( z = (z_0, \ldots, z_{t-1}) \in \mathbb{F}_q^t \) and \( L = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_q^n \) of distinct elements, the Cauchy matrix \( C(z, L) \) is the \( t \times n \) matrix with elements \( C_{ij} = 1/(z_i - L_j) \).

**Theorem 6.6.2** The Goppa code generated by a monic polynomial \( G(x) = (x - z_0) \cdots (x - z_{t-1}) \) without multiple zeros admits a parity-check matrix of the form \( H = C(z, L) \), i.e., \( H_{ij} = 1/(z_i - L_j) \), \( 0 \leq i < t, 0 \leq j < n \).

In this proposal the authors make an extensive use of the fact that using Goppa polynomials separable over \( \mathbb{F}_q \) the resulting Goppa code admits a parity-check matrix in Cauchy form by Theorem 6.6.2. Hence, it is possible to construct parity-check matrices which are in Cauchy and dyadic form, simultaneously.

57
Definition 6.6.3 Let $F_q$ denote a finite field and $h = (h_0, h_1, \ldots, h_{n-1}) \in F_q$ a sequence of $F_q$ elements. The dyadic matrix $\Delta(h) \in F_q^n$ is the symmetric matrix with elements $\Delta_{ij} = h_i \oplus j$. The sequence $h$ is called signature of $\Delta(h)$ and coincides with the first row of $\Delta(h)$. Given $t > 0$, $\Delta(h, t)$ denotes $\Delta(h)$ truncated to its first $t$ rows.

When $n$ is a power of 2 every $1 \times 1$ matrix is a dyadic matrix, and for $k > 0$ any $2^k \times 2^k$ matrix $\Delta(h)$ is of the form $\Delta(h) := \begin{pmatrix} A & B \\ B & A \end{pmatrix}$ where $A$ and $B$ are dyadic $2^{k-1} \times 2^{k-1}$ matrices.

Theorem 6.6.4 Let $H \in F_q^{n \times n}$ with $n > 1$ be a dyadic matrix $H = \Delta(h)$ for some signature $h \in F_q^n$ and a Cauchy matrix $C(z, L)$ for two disjoint sequences $z \in F_q^n$ and $L \in F_q^n$ of distinct elements, simultaneously. It follows that

- $F_q$ is a field of characteristic 2
- $h$ satisfies $\frac{1}{h_{i\oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}$
- the elements of $z$ are defined as $z_i = \frac{1}{h_i} + \omega$, and
- the elements of $L$ are defined as $L_i = \frac{1}{h_j} + \frac{1}{h_0} + \omega$ for some $\omega \in F_q$

It is obvious that a signature $h$ describing such a dyadic Cauchy matrix cannot be chosen completely at random. Hence, the authors suggest only choosing nonzero distinct $h_0$ and $h_i$ at random, where $i$ scans all powers of two smaller than $n$, and to compute all other values for $h$ by $h_{i\oplus j} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}$ for $0 < j < i$.

In the following an algorithm for the construction of binary Goppa codes in dyadic form is presented.
Algorithm 3 Construction of binary dyadic Goppa codes

Input: $q$ (a power of 2), $N \leq q/2$, $t$

Output: $L$, $G(x)$, $H$, $\eta$

1: $U \leftarrow U \setminus \{0\}$  
   \{Choose the dyadic signature $(h_0, \ldots, h_{n-1})$. Note that whenever $h_j$ with $j > 0$ is taken from $U$, so is $1/(1/h_j + 1/h_0)$ to prevent a potential spurious intersection between $z$ and $L$.\}

2: $h_0 \overset{\$}{\leftarrow} U$

3: $\eta_{\lfloor \lg N \rfloor} \leftarrow \frac{1}{h_0}$

4: $U \leftarrow U \setminus \{h_0\}$

5: for $r \leftarrow 0$ to $\lfloor \lg N \rfloor - 1$ do

6: \hspace{1em} $i \leftarrow 2^r$

7: \hspace{1em} $h_i \overset{\$}{\leftarrow} U$

8: \hspace{1em} $\eta_r \leftarrow \frac{1}{h_i} + \frac{1}{h_0}$

9: \hspace{1em} $U \leftarrow U \setminus \{h_i, \frac{1}{h_i}, \frac{1}{h_0}\}$

10: for $j \leftarrow 1$ to $i - 1$ do

11: \hspace{1em} $h_{i \oplus j} \leftarrow \frac{1}{h_i + h_j + h_0}$

12: \hspace{1em} $U \leftarrow U \setminus \{h_{i \oplus j}, \frac{1}{h_{i \oplus j} + h_0}\}$

13: end for

14: end for

15: $\omega \overset{\$}{\leftarrow} \mathbb{F}_q$

\{Assemble the Goppa polynomial\}

16: for $i \leftarrow 0$ to $t - 1$ do

17: \hspace{1em} $z_i \leftarrow \frac{1}{h_i} + \omega$

18: end for

19: $G(x) \leftarrow \prod_{i=0}^{t-1} (x - z_i)$

\{Compute the support\}

20: for $j \leftarrow 0$ to $N - 1$ do

21: \hspace{1em} $L_j \leftarrow \frac{1}{h_j} + \frac{1}{h_0} + \omega$

22: end for

23: $h \leftarrow (h_0, \ldots, h_{N-1})$

24: $H \leftarrow \Delta(t, h)$

25: return $L$, $G(x)$, $H$, $\eta$
Algorithm 3 takes as input three integers: \( q, N, \) and \( t \). The first integer \( q = p^d = 2^m \) where \( m = s \cdot d \) defines the finite field \( \mathbb{F}_q \) as degree \( d \) extension of \( \mathbb{F}_p = \mathbb{F}_2 \). The code length \( N \) is a power of two such that \( N \leq q/2 \). The integer \( t \) denotes the number of errors correctable by the Goppa code. Algorithm 3 outputs the support \( L \), a separable polynomial \( G(x) \), as well as the dyadic parity-check matrix \( H \in \mathbb{F}_q^{t \times N} \) for the binary Goppa code \( \Gamma(L, G(x)) \) of length \( N \) and designed minimum distance \( 2t + 1 \).

Furthermore, Algorithm 3 generates the essence \( \eta \) of the signature \( h \) of \( H \) where \( \eta_r = \frac{1}{h_2^r} + \frac{1}{h_0} \) for \( r = 0, \ldots, \lfloor \log N \rfloor - 1 \) with \( \eta_{\lfloor \log N \rfloor} = \frac{1}{h_0} \), so that, for \( i = \sum_{k=0}^{\lfloor \log N \rfloor - 1} i_k 2^k \), \( \frac{1}{h_i} = \eta_{\lfloor \log N \rfloor} + \sum_{k=0}^{\lfloor \log N \rfloor - 1} i_k \eta_k \). The first \( \lfloor \log t \rfloor \) elements of \( \eta \) together with \( \lfloor \log N \rfloor \) completely specify the roots of the Goppa polynomial \( G(x) \), namely, \( z_i = \eta_{\lfloor \log N \rfloor} + \sum_{k=0}^{\lfloor \log t \rfloor - 1} i_k \eta_k \).

The number of possible dyadic Goppa codes which can be produced by Algorithm 3 is the same as the number of distinct essences of dyadic signatures corresponding to Cauchy matrices. This is about \( \prod_{i=0}^{\lfloor \log N \rfloor} (q - 2^i) \). The algorithm also produces equivalent essences where the elements corresponding to the roots of the Goppa polynomial are only permuted. That leads to simple reordering of those roots. As the Goppa polynomial itself is defined by its roots regardless of their order, the actual number of possible Goppa polynomials is \( (\prod_{i=0}^{\lfloor \log N \rfloor} (q - 2^i)) / (\lfloor \log N \rfloor)! \).

### 6.7 Quasi-Dyadic Goppa Codes

The cryptosystems, that will be introduced in Chapter 7, cannot be securely defined using completely dyadic Goppa codes which admit a parity-check matrix in Cauchy form. By solving the overdefined linear system \( \frac{1}{H_{ij}} = z_i + L_j \) with \( nt \) equations and \( n + t \) unknowns the Goppa polynomial \( G(x) \) would be revealed immediately. Hence, Barreto and Misoczki propose using binary Goppa codes in quasi-dyadic form for cryptographic applications.

**Definition 6.7.1** A quasi-dyadic matrix is a possibly non-dyadic block matrix whose component blocks are dyadic submatrices.

A quasi-dyadic Goppa code over \( \mathbb{F}_p = \mathbb{F}_2 \) for some \( s \) is obtained by constructing a dyadic parity-check matrix \( H_{dyad} \in \mathbb{F}_q^{t \times n} \) over \( \mathbb{F}_q = \mathbb{F}_{p^m} = \mathbb{F}_2^m \) of length \( n = lt \) where \( n \) is a multiple of the desired number of errors \( t \), and then computing the co-trace matrix \( H'_{Tr} = Tr'(H_{dyad}) \in \mathbb{F}_p^{t \times l} \).

The resulting parity-check matrix for the quasi-dyadic Goppa code is a non-dyadic matrix composed of blocks of dyadic submatrices by Theorem 6.7.2.

**Theorem 6.7.2** The co-trace matrix \( H'_{Tr} \in \mathbb{F}_p^{dT \times lt} \) of a dyadic matrix \( H_{dyad} \in \mathbb{F}_q^{t \times lt} \) is quasi-dyadic and consists of dyadic blocks of size \( t \times t \) each.

Consider a dyadic block \( B \) over \( \mathbb{F}_q \) of size \( 2 \times 2 \) which is the minimum block of a dyadic parity-check matrix for a binary Goppa code.

\[
B := \begin{pmatrix}
h_0 & h_1 \\
h_1 & h_0
\end{pmatrix}
\]
6.8 Decoding Algorithms for Goppa Codes

The co-trace construction (see Section 6.4.3) derives from $B$ a matrix of the following form.

$$B'_{Tr} := \begin{pmatrix} h_{0,0} & h_{1,0} \\ h_{1,0} & h_{0,0} \\ h_{0,1} & h_{1,1} \\ h_{1,1} & h_{0,1} \end{pmatrix}$$

It is not hard to see that $B'_{Tr}$ is no more dyadic but consists of dyadic blocks over $F_p$ of size 2 × 2 each. The quasi-dyadicity of $B'_{i,Tr}$ can be shown recursively for all blocks $B_i$. Consequently, the complete co-trace matrix $Tr'(H_{dyad})$ is quasi-dyadic over $F_p$.

6.8 Decoding Algorithms for Goppa Codes

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.

6.8.1 Key Equation

Let $E$ be a vector with elements in $F_{p^m}$ 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 $\gamma_i$ are elements of the support $\mathcal{L}$ for the Goppa code $Goppa(\mathcal{L}, g(z))$, where the positions of these elements inside of $\mathcal{L}$ correspond to the error positions $x_i$ in $\hat{c}$. The error locator polynomial is defined as:

$$\sigma(z) = \prod_{i \in E}(z - \gamma_i) = \prod_{i \in E}(1 - x_i z).$$ (6.8.1)

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:

\[
\omega(z) = \sum_{i \in E} y_i x^i \prod_{j \neq i \in E} (1 - x_j z).
\] (6.8.2)

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

\[
\omega(z) \equiv \sigma(z) \cdot S(z) \mod g(z).
\] (6.8.3)

### 6.8.2 Syndrome Computation

The input to the decoder is a syndrome \( S_c(z) \) for some vector \( \hat{c} = c + e \), where \( c \) is a codeword representing a message \( m \) and \( e \) is an error vector. By definition, \( S_c(z) = S_e(z) \) since \( S_e(z) = 0 \). Generally it can be computed as \( S_e(z) = H \cdot \hat{e}^T \). If \( S(z) = 0 \), 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 (6.5.3):

\[
S(z) \equiv \sum_{\alpha \in \mathbb{F}_2 m} \frac{\hat{e}_\alpha}{z - \alpha_i} \mod g(z) \equiv \sum_{\alpha \in \mathbb{F}_2 m} \frac{e_\alpha}{z - \alpha_i} \mod g(z)
\] (6.8.4)

### 6.8.3 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.

#### General usage

BM returns an error locator polynomial \( \sigma(z) \) and error value polynomial \( \omega(z) \) satisfying the key equation (6.8.3). Applied to binary codes, \( \sigma(z) \) does not need to be taken into account.

#### Preliminaries

Alg. 4 shows the Berlekamp-Massey-Sugiyama algorithm for decoding the syndrome of a vector \( \hat{c} = c + e \in \mathbb{F}_p^n \) 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 \( g(z) = z^{2(t+1)} \). In the general case, the BM algorithm ensures the correction of all errors only if
6.8. Decoding Algorithms for Goppa Codes

Algorithm 4 Berlekamp-Massey-Sugiyama algorithm

Input: Syndrome \( s = S(z) \), Alternant code with generator polynomial \( g(z) \)
Output: Error locator polynomial \( \sigma(z) \)

1: if \( s \equiv 0 \mod g(z) \) then
2: return \( \sigma(z) = 0 \)
3: else
4: \( (\sigma(z), \omega(z)) \leftarrow \text{EEA}(S(z), G(z)) \)
5: return \( (\sigma(z), \omega(z)) \)
6: end if

6.8. Decoding Algorithms for Goppa Codes

Decoding general alternant codes

The input to BM is a syndrome polynomial, which can be computed as described in Section 6.8.2. In the general case, Berlekamp defines the syndrome as \( S(z) = \sum_{i=1}^{\infty} S_i z^m \), where only \( S_1, \ldots, S_t \) are known to the decoder. Then, he constructs a relation between \( \sigma(z) \) ((6.8.1)) and \( \omega(z) \) ((6.8.2)) and the known \( S_i \) by dividing \( \omega(z) \) by \( \sigma(z) \).

\[
\frac{\omega(z)}{\sigma(z)} = 1 + \sum_{j} y_j x_j z = 1 + \sum_{i=1}^{\infty} S_i z^m \quad (6.8.5)
\]

where \( x_i \) are the error positions and \( y_i \) the error values known from Section 6.8.1.

Thus, he obtains the key equation

\[(1 + S(z)) \cdot \sigma \equiv \omega \mod z^{2t+1} \quad (6.8.6)\]

already known from Section 6.8.1.

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 \((1 + S(z)) \sigma(k) \equiv \omega(k) \mod z^{k+1} \) [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 [SKHN76], 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 performance 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 \( S(z), G(z) \), which returns \( \sigma \) and \( \omega \) as coefficients of Bézouts identity given in (4.0.1). The error positions \( x_i \) can be determined by finding the roots of \( \sigma \), as shown in Section 6.9. For non-binary codes,
also \( \omega \) needs to be evaluated to determine the error values. This can be done using a formula due to Forney \([\text{For65}]\), which computes the error values as

\[
e_i = -\frac{\omega(x_i^{-1})}{\sigma'(x_i^{-1})}
\]

where \( \sigma' \) is the formal derivative of the error locator polynomial.

**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 \( d_{\text{min}} = 2t + 1 \). 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 \([\text{MS78}]\)

\[
Goppa(L, g(z)) \equiv Goppa(L, g(z)^2)
\]

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 \( Goppa(L, g(z)^2) \). 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 in \([\text{HG12}]\). Remember that a syndrome \( s \) of length \( n - k \) corresponding to an erroneous codeword \( \hat{c} \) satisfies the equation \( s = S\hat{c} = eH^T \), where \( e \) is the error vector that we want to obtain by decoding \( s \). Now let \( s \) be a syndrome of standard size computed modulo \( g(z) \). By prepending \( s \) with \( k \) zeros, we obtain \((0|s)\) of length \( n \). Then, using (6.8.8) 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 $e$.

### 6.8.4 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 $d_{\text{min}} \geq 2t + 1$. 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 5** PATTERSON algorithm for decoding binary Goppa codes

**Input:** Syndrome $s = S_i(z)$, Goppa code with an irreducible Goppa polynomial $g(z)$

**Output:** Error locator polynomial $\sigma(z)$

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

**Preliminaries** Alg. 5 summarizes Patterson’s algorithm for decoding the syndrome of a vector $\hat{c} = c + e \in \mathbb{F}_{2^m}$ 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 occurred, 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 (6.8.3) to a simpler equation using the property $\omega(z) = \sigma'(z)$ and the fact that $y_i = 1$ at all error positions.

$$\omega(z) \equiv \sigma(z) \cdot S(z) \equiv \sum_{i \in E} x_i \prod_{j \neq i \in E} (1 - z) \mod g(z)$$ (6.8.9)

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

$$\sigma(z) = a(z)^2 + z \cdot b(z)^2$$ (6.8.10)

65
Now, formal derivation and application of the original key equation yields
\[
\sigma'(z) = b(z)^2 = \omega(z) \\
\equiv \sigma(z) \cdot S(z) \mod g(z) \quad (6.8.11) \\
\equiv (a(z)^2 + zb(z)^2) \cdot S(z) \equiv b(z)^2 \mod g(z) \quad (6.8.13)
\]
Choosing \( g(z) \) irreducible ensures the invertibility of the syndrome \( S \). To solve the equation for \( a(z) \) and \( b(z) \), we now compute an inverse polynomial \( T(z) \equiv S_c(z)^{-1} \mod g(z) \) and obtain
\[
(T(z) + z) \cdot b(z)^2 \equiv a(z)^2 \mod g(z). \quad (6.8.14)
\]
If \( T(z) = z \), we obtain the trivial solutions \( a(z) = 0 \) and \( b(z)^2 = zb(z)^2 \cdot S(z) \mod g(z) \), yielding \( \sigma(z) = z \). Otherwise we use an observation by \([\text{Hub}]\) 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
\[
a(z) \equiv b(z) \cdot R(z) \mod G(z) \quad (6.8.15)
\]
can be computed using \text{EEA} and applied to (6.8.10). As \( \deg(\sigma(z)) \leq g(z) = t \), the equation implies that \( \deg(a(z)) \leq \lfloor \frac{t}{2} \rfloor \) and \( \deg(b(z)) \leq \lfloor \frac{t-1}{2} \rfloor \) \([\text{Hey08, OS08}]\). Observing the iterations of EEA (Alg. 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 \([\text{OS09}]\). Given a Goppa polynomial \( g(z) \) of degree \( t \) and coefficients of size \( m \), EEA takes \( \mathcal{O}(t^2 m^2) \) 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 \( \mathbb{F}_{2^m}[z]/g(z) \), which takes \( \mathcal{O}(t^2 m^2) \) 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.

### 6.9 Extracting Roots of the Error Locator Polynomial

The computation of the roots of the \text{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 6.8.1, the roots of \( \sigma(z) = \sum_{i=0}^{t-1} \sigma_i z^i \) are elements of the support \( \mathcal{L} \), where the position of the roots \textit{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

\[
e_i = \begin{cases} 
1 & \sigma(L(i)) \equiv 0 \\
0 & \text{otherwise}
\end{cases}
\] (6.9.1)

### 6.9.1 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. 6, is not sophisticated, but can be implemented easily and may be even faster than others as long as the field size is low enough. The search can be stopped as soon as \( t \) errors have been found. Note, however, that this introduces a potential timing 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 \( \sigma_0 + z(\sigma_1 + z(\sigma_2 + \cdots + z(\sigma_{t-1} + z\sigma_t) \cdots)) \), the well-known Horner scheme \([\text{Hor}19]\) 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 6 SEARCH FOR ROOTS OF \( \sigma(z) \) USING HORNER’S SCHEME

**Input:** Error locator polynomial \( \sigma(z) \), support \( \mathcal{L} \)

**Output:** Error vector \( e \)

1: \( e \leftarrow 0 \)
2: for \( i = 0 \) on \( n - 1 \) do
3: \( x \leftarrow L(i) \)
4: \( s \leftarrow \sigma_i \)
5: for \( j \leftarrow t \) to 0 do
6: \( s \leftarrow \sigma_j + s \times x \)
7: end for
8: if \( s = 0 \) then
9: \( e_i = 1 \)
10: end if
11: end for
12: return \( e \)

Note that it is possible to speed up the search by performing a polynomial division of \( \sigma(z) \) by \((z - L_i)\) as soon as \( L_i \) 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.
6.9.2 Brute Force Search using Chien Search

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

\[
\begin{align*}
\sigma(\alpha^i) &= \sigma_0 + \sigma_1 \alpha^i + \ldots + \sigma_t (\alpha^i)^t \\
\sigma(\alpha^{i+1}) &= \sigma_0 + \sigma_1 \alpha^{i+1} + \ldots + \sigma_t (\alpha^{i+1})^t \\
\end{align*}
\]

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 $\alpha^i$ 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 $\alpha^i$; this is not considered in Alg. 7.

Chien search can be used to perform a brute-force 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 brute-force 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.

\begin{algorithm}
\caption{Chien search for roots of $\sigma(z)$}
\begin{algorithmic}[1]
\State \textbf{Output:} Error locator polynomial $\sigma(z)$, support $L$
\State \textbf{Input:} Error vector $e$
\State 1: $e \leftarrow 0$
\State 2: if $\sigma_0 = 0$ then
\State 3: \quad $x = L^{-1}(0), e_x \leftarrow 1$
\State 4: end if
\State 5: for $i \leftarrow 0$ to $t$ do
\State 6: \quad $p_i \leftarrow \sigma_i$
\State 7: end for
\State 8: for $i \leftarrow 1$ to $p^m - 1$ do
\State 9: \quad $s \leftarrow \sigma_0$
\State 10: \quad for $j \leftarrow 0$ to $t$ do
\State 11: \quad \quad $p_j \leftarrow p_j \cdot \alpha^j$
\State 12: \quad \quad $s \leftarrow s + p_j$
\State 13: \quad end for
\State 14: \quad if $s = 0$ then
\State 15: \quad \quad $x = L^{-1}(\alpha^i), e_x \leftarrow 1$
\State 16: \quad end if
\State 17: end for
\State 18: return $e$
\end{algorithmic}
\end{algorithm}
6.9. Extracting Roots of the Error Locator Polynomial

6.9.3 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 \( \leq 10 \). 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 \( \text{BTZ} \).

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

\[
\text{Tr}(z) = \sum_{i=0}^{m-1} z^{p^i}
\]

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

Berlekamp proves that

\[
f(z) = \prod_{s \in \mathbb{F}_p} \gcd(f(z), \text{Tr}(\beta_i z) - s) \; \forall \; 0 \leq j < m
\]

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.

**Algorithm 8 BTZ Algorithm Extracting Roots of \( \sigma(z) \)**

**Output:** Error vector \( e \), Polynomial \( f(z) \), support \( \mathcal{L} \), integer \( i \), integer \( d_z \)

**Input:** Error vector \( e 

1: if \( \deg(f) = 0 \) then
2: \quad \text{return } e
3: \text{end if}
4: if \( \deg(f) = 1 \) then
5: \quad x = \mathcal{L}^{-1}(\frac{e_0}{f_0}), e_x \leftarrow 1
6: \quad \text{return } e
7: \text{end if}
8: if \( \deg(f) \leq d_z \) then
9: \quad \text{return } \text{Zinoviev}(f, \mathcal{L}, e)
10: \text{end if}
11: g \leftarrow \gcd(f, \text{Tr}(\beta_i \cdot z))
12: h \leftarrow f/g
13: \text{return } \text{BTZ}(e, g, \mathcal{L}, i + 1, d_z) \cup \text{BTZ}(e, h, \mathcal{L}, i + 1, d_z)
Alg. 8 shows the BTZ algorithm, but omits all details of Zinoviev’s algorithms. The first call to the algorithm sets \( i = 1 \) to select \( \beta_1 \) and \( f = \sigma(z) \) and the error vector \( e \) to zero. Note that the polynomials \( \text{Tr}(\beta_1 z) \mod f(z) \forall 0 \leq i < m \) can be precomputed.

### Chapter 6. Introduction to Error Correcting Codes

#### 6.10 MDPC-Codes

In contrast to the heavily structured codes presented above, MDPC codes have a straightforward definition:

**Definition 6.10.1 (MDPC codes)** A \((n,r,w)\)-MDPC code is a linear code of length \( n \) and co-dimension \( r \) admitting a parity check matrix with constant row weight \( w \).

When MDPC codes are quasi-cyclic, they are called \((n,r,w)\)-QC-MDPC codes. LDPC codes typically have small constant row weights (usually, less than 10). For MDPC codes, row weights scaling in \( O(\sqrt{n \log(n)}) \) are assumed.

#### 6.11 Decoding MDPC Codes

For code-based cryptosystems, decoding a codeword (i.e., the syndrome) is usually the most complex task. Decoding algorithms for LDPC/MDPC codes are mainly divided into two families. The first class (e.g., \([BMvT78a]\)) offers a better error-correction capability but is computationally more complex than the second family. Especially when handling large codes, the second family, called bit-flipping algorithms \([Gal62]\), seems to be more appropriate. In general, they are all based on the following principle:

1. Compute the syndrome \( s \) of the received codeword \( x \).
2. Check the number of unsatisfied parity-check-equations \( \#_{\text{upc}} \) associated with each codeword bit.
3. Flip each codeword bit that violates more than \( b \) equations.

This process is iterated until either the syndrome becomes zero or a predefined maximum number of iterations is reached. In that case a decoding error is returned. The main difference of the bit-flipping algorithms is how the threshold \( b \) is computed. In the original algorithm of Gallager \([Gal62]\), a new \( b \) is computed at each iteration. In \([HP03]\), \( b \) is taken as the maximum of the unsatisfied parity-check-equations \( \text{Max}_{\text{upc}} \) and \([MTSB12]\) propose to use \( b = \text{Max}_{\text{upc}} - \delta \), for some small \( \delta \). An extensive evaluation of the existing decoders and newly developed ones is presented in Chapter 12.
Chapter 7

Cryptosystems Based on Error Correcting Codes

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 7.1 provides a rough introduction to the fundamentals and basic mechanisms of code-based cryptography, followed by a presentation of currently recommended security parameters in Section 7.2. Then, the Classical (Section 7.3) and Modern (Section 7.4) version of McEliece and of Niederreiter (Section 7.5) are discussed, without yet delving into the finer details of Coding theory. In Section 13.3 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 8.6.

7.1 Overview

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 a triple of algorithms: key generation, encryption and decryption. All popular public-key cryptosystems are based on one-way functions\textsuperscript{App. 16.2.3}. 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 \textit{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.
Chapter 7. Cryptosystems Based on Error Correcting Codes

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

Table 7.1: Parameters sets for typical security levels according to \([BLP08b]\)

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 independently 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 chapter introduces the reader to both variants, throughout the remainder of the thesis we will always consider only the Modern variant.

7.2 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 \([AJM97]\) 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 \textit{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 security level for the originally suggested parameters by McEliece fell from around \(2^{80}\) in 1986 to \(2^{59.9}\) in 2009 \([FS09]\). Table 7.1 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 8.6 after the presentation of the cryptosystems.

\(^1\)See for example \([Min07, OS09]\)
7.3 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.

7.3.1 Key Generation

As shown in Alg. 9 the key generation algorithm starts with the selection of an 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 \( \hat{C} \). \( \hat{G} \) now serves as the public key and the matrices \( G, S \) and \( P \) – or equivalently \( S^{-1}, P^{-1} \) – compose the secret key.

<table>
<thead>
<tr>
<th>Algorithm 9 Classical McEliece: Key Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Fixed system parameters ( t, n, p, m )</td>
</tr>
<tr>
<td><strong>Output:</strong> private key ( K_{sec} ), public key ( K_{pub} )</td>
</tr>
<tr>
<td>1: Choose a binary ([n, k, d])-Goppa code ( \hat{C} ) capable of correcting up to ( t ) errors</td>
</tr>
<tr>
<td>2: Compute the corresponding ( k \times n ) generator matrix ( G ) for code ( \hat{C} )</td>
</tr>
<tr>
<td>3: Select a random non-singular binary ( k \times k ) scrambling matrix ( S )</td>
</tr>
<tr>
<td>4: Select a random ( n \times n ) permutation matrix ( P )</td>
</tr>
<tr>
<td>5: Compute the ( k \times n ) matrix ( \hat{G} = S \cdot G \cdot P )</td>
</tr>
<tr>
<td>6: Compute the inverses of ( S ) and ( P )</td>
</tr>
<tr>
<td>7: return ( K_{sec} = (G, S^{-1}, P^{-1}), K_{pub} = (\hat{G}) )</td>
</tr>
</tbody>
</table>

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 8.5 need to be applied to address this problem and allow the intentional use of a systematic matrix as in Modern McEliece.

7.3.2 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. 10. The multiplication adds redundancy to the codeword, resulting in a message expansion from \( k \) to \( n \) with overhead \( \frac{4}{k} \).

\(^2\)See [Bou07] for details on code equivalence.
Chapter 7. Cryptosystems Based on Error Correcting Codes

Algorithm 10 CLASSICAL McEliece: ENCRYPTION

**Input:** Public key $K_{\text{pub}} = (\hat{G})$, message $M$

**Output:** 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 $c = m \cdot \hat{G} + e$

7.3.3 Decryption

The McEliece decryption shown in Alg. 11 consists mainly of the removal of the applied errors using the known decoding algorithm $D_{\text{Goppa}}(c)$ for the code $C$. 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 6.8. Decryption works correctly despite of the transformation of the code $C$ because the following equations hold:

\[
\begin{align*}
\hat{c} &= c \cdot P^{-1} \\
&= (m \cdot \hat{G} + e) \cdot P^{-1} \\
&= (m \cdot S \cdot G \cdot P + e) \cdot P^{-1} \\
&= m \cdot S \cdot G \cdot P \cdot P^{-1} + e \cdot P^{-1} \\
&= m \cdot S \cdot G \cdot + e \cdot P^{-1}
\end{align*}
\]

Remember from Section 7.3.1 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 $C$. 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 11 CLASSICAL McEliece: DECRYPTION

**Input:** Ciphertext $c$ of length $n$, private key $K_{\text{sec}} = (G, S^{-1}, P^{-1})$

**Output:** Message $M$

1: Compute $\hat{c} = c \cdot P^{-1}$
2: Obtain $\hat{m}$ of length $k$ from $\hat{c}$ using the decoding algorithm $D_{\text{Goppa}}(\hat{c})$ for code $C$
3: Compute $m = \hat{m} \cdot S^{-1}$
4: Represent $m$ as message $M$
5: return $M$

7.4 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 8.5 discusses this requirement in greater detail.

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

### 7.4.1 Key generation

Alg. 12 shows the key generation algorithm for the Modern McEliece variant.

<table>
<thead>
<tr>
<th>Algorithm 12</th>
<th>MODERN McELIECE: KEY GENERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
<td>Fixed system parameters ( t, n, m, p = 2 )</td>
</tr>
<tr>
<td><strong>Output:</strong></td>
<td>private key ( K_{sec} ), public key ( K_{pub} )</td>
</tr>
<tr>
<td>1:</td>
<td>Select a random Goppa polynomial ( g(z) ) of degree ( t ) over ( GF(p^m) )</td>
</tr>
<tr>
<td>2:</td>
<td>Randomly choose ( n ) elements of ( GF(p^m) ) that are no roots of ( g(z) ) as the support ( L )</td>
</tr>
<tr>
<td>3:</td>
<td>Compute the parity check matrix ( H ) according to ( L ) and ( g(z) )</td>
</tr>
<tr>
<td>4:</td>
<td>Bring ( H ) to systematic form using Gauss-Jordan elimination: ( H_{sys} = S \cdot H )</td>
</tr>
<tr>
<td>5:</td>
<td>Compute systematic generator matrix ( G_{sys} ) from ( H_{sys} )</td>
</tr>
<tr>
<td>6:</td>
<td><strong>return</strong> ( K_{sec} = (L, g(z)), K_{pub} = (G_{sys}) )</td>
</tr>
</tbody>
</table>

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 6.5.2, 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 \( D_{Goppa}(c) \) is known.

Table 7.2 illustrates the relationship between the public and private versions of generator matrix, parity check matrix and support.
7.4.2 Encryption

Encryption in Modern McEliece (see Alg. 13) 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 13 Modern McEliece: Encryption

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

7.4.3 Decryption

Decryption in the Modern McEliece variant shown in Alg. 14 consists exclusively of the removal of the applied errors using the known decoding algorithm $D_{Goppa}(c)$ for the code $C$. 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 read directly from the first $k$ bits of the codeword.

Algorithm 14 Modern McEliece: Decryption

Input: Ciphertext $c$, private key $K_{sec} = (L, g(x))$
Output: Message $M$
1: Compute the syndrome $s$ corresponding to $c$
2: Obtain $m$ from $s$ using decoding algorithm for known code $C$ defined by $(L, g(x))$
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 7.2. $G_{sys}$ is still a valid generator matrix for an equivalent code $C$.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Classical McEliece</th>
<th>Modern McEliece</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key.</td>
<td>$\hat{G} = S \cdot G \cdot P$</td>
<td>$L_{sec} = L_{pub} \cdot P$</td>
</tr>
<tr>
<td></td>
<td>$L_{sec} \Rightarrow H_{sys} = (Q^T</td>
<td>I_n-k) = \hat{S} \cdot H \cdot \hat{P}$</td>
</tr>
<tr>
<td>Enc.</td>
<td>$c = m \cdot \hat{G} + e$</td>
<td>$c = m \cdot G_{sys} + e = (m</td>
</tr>
<tr>
<td>Dec.</td>
<td>$\hat{c} = c \cdot P^{-1}, \hat{m} = D_{Goppa, L_{pub}(\hat{c})}, m = \hat{m} \cdot S^{-1}$</td>
<td>$\hat{m} = D_{Goppa, L_{sec}(c)} = m</td>
</tr>
</tbody>
</table>

Table 7.2: Comparison of the modern and Classical version of McEliece
7.5 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.

7.5.1 Key generation

Key generation works similar to McEliece, but does not require the computation of the generator matrix. Alg. 15 shows the Classical key generation algorithm for the Niederreiter cryptosystem, while Alg. 16 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 15 CLASSICAL NIEDERREITER: KEY GENERATION

Input: Fixed system parameters $t, n, p, m$
Output: 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 16 MODERN NIEDERREITER: KEY GENERATION

Input: Fixed system parameters $t, n, p = 2$
Output: private key $K_{sec}$, public key $K_{pub}$
1: Select a random Goppa polynomial $g(x)$ of degree $t$ over $GF(p^m)$
2: Randomly choose $n$ elements of $GF(p^m)$ that are no roots of $g(x)$ as the support $\mathcal{L}$
3: Compute the parity check matrix $H$ according to $\mathcal{L}$ and $g(x)$
4: Bring $H$ to systematic form using Gauss-Jordan elimination: $H_{sys} = S \cdot H$
5: Compute $S^{-1}$
6: return $K_{sec} = (L, g(x), S^{-1}), K_{pub} = (H_{sys})$
Chapter 7. Cryptosystems Based on Error Correcting Codes

7.5.2 Encryption

For encryption, the message $M$ needs to be represented as a Constant Weight (CW) word of length $n$ and hamming weight $t$. There exist several techniques for CW encoding, one of which will be presented in Section 7.5.4. The CW encoding is followed by a simple vector-matrix multiplication.

Encryption is shown in Alg. 17. 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 17 Niederreiter: Encryption

htb]

Input: Public key $K_{pub} = (H_{sys})$, message $M$
Output: Ciphertext $c$
1: Represent message $M$ as binary string $e$ of length $n$ and weight $t$
2: return $c = H_{sys} \cdot e^T$

7.5.3 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. 18, the error vector after decoding is still permuted, so it needs to be multiplied by $P^{-1}$. In the Modern variant shown in Alg. 19, 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 18 Classical Niederreiter: Decryption

htb]

Input: Ciphertext $c$ of length $(n-k)$, private key $K_{sec} = (H, S^{-1}, P^{-1})$
Output: Message $M$
1: Compute $\hat{c} = S^{-1} \cdot c$
2: Obtain $\hat{e}$ from $\hat{c}$ using the decoding algorithm $D_{Goppa}(\hat{c})$ for code $C$
3: Compute $e = P^{-1} \cdot \hat{e}$ of length $n$ and weight $t$
4: Represent $e$ as message $M$
5: return $M$

Algorithm 19 Modern Niederreiter: Decryption

htb]

Input: Ciphertext $c$, private key $K_{sec} = (L, g(x), S^{-1})$
Output: Message $M$
1: Compute $\hat{c} = S^{-1} \cdot c$
2: Obtain $e$ from $\hat{c}$ using decoding algorithm for known code $C$
3: Represent $e$ as message $M$
4: return $M$
7.5. Niederreiter Cryptosystem

7.5.4 Constant Weight Encoding

Before encrypting a message with Niederreiter’s cryptosystem, the message has to be encoded into an error vector. More precisely, the message needs to be transformed into a bit vector of length \( n \) and constant weight \( t \). There exist quite a few encoding algorithms (e.g., \([\text{Cov73, Sen95, FS96}]\)), however they are not directly applicable to the restricted environment of embedded systems and hardware. We therefore unfolded the recursive algorithm proposed in \([\text{Sen05}]\) so that it can run iteratively by a simple state machine. The proposal is based on Golomb’s run-length coding \([\text{Gol66}]\) which is a form of lossless data compression for a memoryless binary source with highly unbalanced probability law, e.g., such that \( p = \text{Prob}(0) \geq 1/2 \). During the encoding operation in \([\text{Sen05}]\), one has to compute a value \( d \approx \ln(2) \cdot (n - t - 1) \) to determine how many bits of the message are encoded into the distance to the next one-bit on the error vector. Many embedded (hardware) systems do not have a dedicated floating-point and division unit so these operations should be replaced. We therefore substituted the floating point operation and division by a simple and fast table lookup (see \([\text{Hey10}]\) for details). Since we still preserve all properties from \([\text{Sen05}]\), the algorithm will still terminate with a negligible loss in efficiency. The encoding algorithm suitable for embedded systems is given in Alg. 20. The constant weight decoding algorithm was adapted in a similar way, and is presented in Alg. 21.

Compared to some algorithms like Enumerative encoding \([\text{Cov06}]\) 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 the lower bound must be determined experimentally.

---

**Algorithm 20** Encode a Binary String in a Constant-Weight Word (Bin2CW)

Input: \( n, t \), binary stream \( B \)
Output: \( \Delta[0, \ldots, t - 1] \)

1: \( \delta = 0 \), \( \text{index} = 0 \)
2: while \( t \neq 0 \) do
3: if \( n \leq t \) then
4: \( \Delta[\text{index}++] = \delta \)
5: \( n = 1, t = 1, \delta = 0 \)
6: end if
7: \( u \leftarrow uTable[n, t] \)
8: \( d \leftarrow (1 << u) \)
9: if read(\( B, 1 \)) = 1 then
10: \( n = d, \delta + = d \)
11: else
12: \( i \leftarrow \text{read}(B, u) \)
13: \( \Delta[\text{index}++] = \delta + i \)
14: \( \delta = 0, t = 1, n = (i + 1) \)
15: end if
16: end while
### Algorithm 21

**Decode a Constant-Weight Word to a Binary String (CW2Bin)**

**Input:** \( n, t, \Delta[0, \ldots, t-1] \)

**Output:** binary stream \( B \)

1. \( \delta = 0, \text{index} = 0 \)
2. while \( t \neq 0 \) AND \( n > t \) do
3. \( u \leftarrow uTable[n, t] \)
4. \( d \leftarrow (1 << u) \)
5. if \( \Delta[\text{index}] \geq d \) then
6. \( \text{Write}(1, B) \)
7. \( \Delta[\text{index}]- = d \)
8. \( n= n - d \)
9. else
10. \( \delta = \Delta[\text{index}++] \)
11. \( \text{Write}(0|\delta, B) \)
12. \( n= (\delta + 1), t= t - 1 \)
13. end if
14. end while

Listing 16.5 in the appendix shows the computation of the lookup table for \( u \). It is written in Python for the sake of simplicity. However, it can be easily converted to C code. 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, 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.
Chapter 8
General Security Considerations and New Side-Channel Attacks

This chapter 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 present a successfully side channel attacks in Section 8.4.2. Finally the concepts of indistinguishability and CCA2-secure conversions are introduced.

8.1 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 [MS07] 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 [BMvT78a], 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\(^1\) 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” [MS07]. 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.

\(^1\)Note that attempts at more general transformations exist, see for example [BBC+11].
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].

8.2 Hiding the Structure of the Private Code

The main security issue in code-based cryptography is hiding the structure of the private code. Let \( G \) denote a generator matrix for a private code \( \mathcal{C} \), and let \( \hat{\mathcal{C}} \) denote a public code obtained from \( \mathcal{C} \) by one or more secret transformations. In the following, some usual transformations are summarized, based on [OS09] and [MB09].

1. **Row Scrambler**: Multiply the generator matrix \( G \) for the private code \( \mathcal{C} \) by a random invertible matrix \( S \in \mathbb{F}_q^{k \times k} \) from the left. As \( \langle G \rangle = \langle SG \rangle \), the known error correction algorithm for \( \mathcal{C} \) can be used. Publishing a systematic generator matrix provides the same security against structural attacks as a random \( S \).

2. **Column Scrambler/ Isometry**: Multiply the generator matrix \( G \) for the private code \( \mathcal{C} \) by a random invertible matrix \( P \in \mathbb{F}_q^{n \times n} \) from the right, where \( P \) preserves the norm, e.g., \( P \) is a permutation matrix. If \( G \) and \( P \) are known then up to \( t \) errors can be corrected in \( \langle GP \rangle \).

3. **Subcode**: Let \( 0 < l < k \). Multiply the generator matrix \( G \) for the private code \( \mathcal{C} \) by a random matrix \( S \in \mathbb{F}_q^{l \times k} \) of full rank from the left. As \( \langle SG \rangle \subseteq \langle G \rangle \), the known error correction algorithm may be used.

4. **Subfield Subcode**: Take the subfield subcode \( \mathcal{C}_{SUB} \) of the secret code \( \mathcal{C} \) for a subfield \( \mathbb{F}_p \) of \( \mathbb{F}_q \). As before, one can correct errors by the error correcting algorithm for the secret code. However, sometimes one can correct errors of larger norm in the subfield subcode than in the original code.

5. **(Block-)Shortening**: Extract a shortened public code \( \mathcal{C}^T \) from a very large private code \( \mathcal{C} \) by puncturing \( \mathcal{C} \) on the set of coordinates \( T \). In particular, if \( \mathcal{C} \) is a code defined by a \( t \times N \) matrix \( H \), where \( N = l \cdot t \), such that \( H \) can be considered as a composition of \( l \) blocks of size \( t \times t \) each, then \( T_i \) contains all those coordinates of blocks which have to be deleted in order to obtain a block-shortened code \( \mathcal{C}^{T_i} \).

To protect the secret code, a combination of several transformations is used, as a rule. For instance, in the original McEliece cryptosystem a combination of transformations (1), (2) and (4) is used.

In the following, we explain the role of these transformations in hiding the structure of the private code.

In [CC95] Canteaut and Chabaud pointed out that the scrambling transformation (1) has no cryptographic function. It just sends \( G \) to another generator matrix \( G' \) for the same code to assure that the public generator matrix \( \hat{G} \) is not in systematic form. Otherwise, most bits of the
message would be revealed. Our goal is to construct a systematic public generator matrix for a binary quasi-dyadic Goppa code and to use a conversion for CCA2-secure McEliece versions (see Chapter 11.1.2) to protect the message. Hence, this transformation is neither useful nor necessary for our purpose.

In contrast, the permutation transformation (2) is essential when constructing a trapdoor function. In the following, we consider the permutation equivalence problem of two codes.

Let the symmetric group $S_n$ of order $n$ be a set of permutations of integers $\{0, \ldots, n-1\}$ and $\sigma \in S_n$ be a permutation. $P_\sigma$ denotes the $n \times n$ permutation matrix with components $p_{i,j} = 1$ if $\sigma(i) = j$ and $p_{i,j} = 0$ otherwise.

**Definition 8.2.1** Two codes $C_1$ and $C_2$ are permutation equivalent if there is a permutation matrix $P_\sigma$ such that $G_1$ is a generator matrix for $C_1$ if and only if $P \times G_2$ is the generator matrix for $C_2$. Thus, $P$ sends $C_1$ to $C_2$ by reordering the columns of $G_1$.

The permutation equivalence problem is a decisional problem defined as follows.

**Definition 8.2.2** Given two $k \times n$ matrices $G_1$ and $G_2$ over $F_q$, does there exist a permutation $\sigma$ represented as permutation matrix $P_\sigma$ such that $G_1 \times P_\sigma = G_2$?

This problem is closely related to the graph isomorphism problem which is assumed to be in $\mathcal{P}/\mathcal{NP}$ [PR97]. The Support splitting algorithm [Sen00] is the only known algorithm which solves the permutation equivalence problem of two codes in the practice. The success probability of the best known attack using the Support splitting algorithm to distinguish a Goppa code from a general linear code is negligible for all suitable McEliece parameters.

The transformation (4) is used implicitly in every McEliece-type cryptosystem based on Goppa codes because Goppa codes can be considered as subfield subcodes of Generalized Reed Solomon codes.

The last transformation (5) is of great significance for the construction of a CCA2-secure McEliece-type cryptosystem based on quasi-dyadic Goppa codes as introduced in [MB09] where the public code is equivalent to a subcode of a Reed Solomon code. Combining the transformations (2) and (5) the equivalent shortened code problem can be defined as follows.

**Definition 8.2.3** Let $H$ be a $t \times N$ matrix over $F_q$ and $\tilde{H}$ a $t \times n$ matrix over $F_q$ with $n < N$, does there exist a set of coordinates $T$ of length $N - n$ and a permutation $\sigma \in S_n$ such that $\tilde{H} = H(T) \times P_\sigma$ where $H(T)$ denotes a matrix obtained by deleting of components indexed by the set $T$ in each row of $H$?

The equivalent shortened code problem has been proven to be $\mathcal{NP}$-complete by Wieschbrink in [Wie06]. In contrast to the permutation equivalence problem the equivalent shortened code problem cannot be solved by means of the Support splitting algorithm. Hence, no efficient algorithm is known that solves this problem up to now.
8.3 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 only attacks applicable to variants that apply a CCA2-secure conversion as described in Section 8.5.

8.3.1 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 brute-force complexity of this operation is \( (2^n)^w \).

Information Set Decoding (ISD) 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 \([LB88a]\) 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 \([CC98]\) in 1998, by Bernstein et al. \([BLP08b]\) in 2008 and by May, Meurer et al. \([MMT11, BJMM12a]\) 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”.

8.3.2 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
8.4. Side Channel Attacks

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 $NP$-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 $NP$-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 [MS07] 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].

8.4 Side Channel Attacks

Side channel attacks attempt to extract secret information of a cryptosystem by analysing 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, SSMS09] published several papers on timing attacks against the secret permutation, syndrome inversion and the Patterson algorithm and also pointed out some countermeasures. We evaluated practical power analysis attacks on 8-bit implementations of McEliece [HMP10] which are described in Section 8.4.2.

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.

8.4.1 Introduction to DPA

Power analysis attacks exploit the fact that the execution of a cryptographic algorithm on a physical device leaks information about the processed data and/or executed operations through instantaneous power consumption [KJJ99]. Measuring and evaluating the power consumption of a cryptographic device allows exploiting information-dependent leakage combined with the knowledge about the plaintext or ciphertext in order to extract, e.g., a secret key. Since intermediate result of the computations are serially processed (especially in 8-,16-, or 32-bit architectures, e.g., general-purpose microcontrollers) a divide-and-conquer strategy becomes possible, i.e., the secret key could be recovered byte by byte.

A Simple Power Analysis (SPA) attack, as introduced in [KJJ99], relies on visual inspection of power traces, e.g., measured from an embedded microcontroller of a smartcard. The aim of an SPA is to reveal details about the execution of the program flow of a software implementation, like the detection of conditional branches depending on secret information. Recovering an RSA private key bit-by-bit by an SPA on square-and-multiply algorithm [KJJ99] and revealing a KeeLoq secret key by SPA on software implementation of the decryption algorithm [KKMP09] are amongst the powerful practical examples of SPA on real-world applications. Contrary to SPA, Differential Power Analysis (DPA) utilizes statistical methods and evaluates several power traces. A DPA requires no knowledge about the concrete implementation of the cipher and can hence be applied to most of unprotected black box implementations. According to intermediate values depending on key hypotheses the traces are correlated to estimated power values, and then correlation coefficients indicate the most probable hypothesis amongst all partially guessed key hypotheses [BCO04]. In order to perform a correlation-based DPA, the power consumption of the device under attack must be guessed; the power model should be defined according to the characteristics of the attacked device, e.g., Hamming weight (HW) of the processed data for a microcontroller because of the existence of a precharged/predischarged bus in microcontrollers architecture. In case of a bad quality of the acquired power consumption, e.g., due to a noisy environment, bad measurement setup or cheap equipment, averaging can be applied by decrypting(encrypting) the same ciphertext(plaintext) repeatedly and calculating the mean of the corresponding traces to decrease the noise floor.

8.4.2 A Practical Power Analysis Attacks on Software Implementations of McEliece

This research contribution is based on the author’s published research in [HMP10]. In this section we will describe an attack on the implementation published in [EGHP09].
8.4. Side Channel Attacks

The combination of the McEliece decryption Algorithms 11,14 and the Goppa decoding Algorithm 5 allows a wide range of different implementations. For our proposed attacks, the most interesting point is the specific implementation of Step 1 of Algorithm 11 and Step 1 of Algorithm 5 and whether they are merged together or not. According to these points we define four so-called implementation profiles:

Profile I performs the permutation of the ciphertext and computes the columns of $H$ as they are needed by either using the extended euclidean algorithm (EEA) or the structure given in Equation (6.5.6) or (6.5.7).

Profile II also performs the permutation, but uses the precomputed parity check matrix $H$.

Profile III does not really perform the permutation, but directly uses a permuted parity check matrix. As stated in Section 6.5, we can use $L_p = P^{-1} * L$ to compute the syndrome of the unpermuted ciphertext. This profile computes the permuted columns as needed.

Profile IV does the same as profile III, but uses a precomputed and permuted parity check matrix.

Adversary Model

In our proposed attacks we consider the following adversary model:

The adversary knows what is public like $\hat{G}, t$. Also he knows the implementation platform (e.g., type of the microcontroller used), the implementation profile, i.e., complete source code of the decryption scheme (of course excluding memory contents, precomputed values, and secret key materials). Also, he is able to select different ciphertexts and measures the power consumption during the decryption operation.

Possible Power Analysis Vulnerabilities

In order to investigate the vulnerability of the target implementation platform to power analysis attacks, a measurement setup by means of an AVR ATmega256 microcontroller which is clocked by a 16MHz oscillator is developed. Power consumption of the target device is measured using a LeCroy WP715Zi 1.5GHz oscilloscope at a sampling rate of 10GS/s and by means of a differential probe which captures the voltage drop of a $10\Omega$ resistor at VDD (5V) path.

To check the dependency of power traces on operations, different instructions including arithmetic, load, and save operations are taken into account, and power consumption for each one for different operands are collected. In contrary to 8051-based or PIC microcontrollers, which need 16, 8, or 4 clock cycles to execute an operation, an AVR ATmega256 executes the instructions in 1 or 2 clock cycles\(^2\). Therefore, the power consumption pattern of different instructions are not very different from each other. As Figure 8.1 shows, though the instructions are not certainly recognizable, load instructions are detectable amongst others. As a result the adversary may be

\(^2\)Most of the arithmetic instructions in 1 clock cycle.
able to detect the execution paths by comparing the power traces. Note that as mentioned in Section 8.4.1 if the adversary is able to repeat the measurement for a certain input, averaging helps to reduce the noise and hence improve the execution path detection procedure.

On the other hand, considering a fixed execution path, operand of instructions play a significant role in variety of power consumption values. As mentioned before, since the microcontrollers usually precharge/predischarge the bus lines, Hamming weight (HW) of the operands or HW of the results are proportional to power values.

Figure 8.2 shows the dependency of power traces on the operands for XOR, LOAD, and SAVE instructions. Note that the XOR instruction takes place on two registers, the LOAD instruction loads an SRAM location to a specified register, and the SAVE instruction stores the content of a register back to the SRAM. According to Figure 8.2, the HW of operands of SAVE instruction are more distinguishable in comparison to that of XOR and LOAD instructions. Therefore, according to the defined adversary model we suppose that the adversary considers only the leakage of the SAVE instructions. Now the question is “How precisely can the HW of the values stored by a SAVE instruction detected by an the adversary?” It should be noted that a similar question has been answered in the case of a PIC microcontroller in [RSVC09] where the adversary (which fits to our defined adversary model in addition to profiling ability) has to profile the power traces in order to correctly detect the HWs. The same procedure can be performed on our implementation platform. However, in our defined adversary model the device under attack can be controlled by the attacker in order to repeat measurements as many as needed for the same input (ciphertext). Therefore, without profiling the attacker might be able
8.4. Side Channel Attacks

Figure 8.3: Success rate of HW detection using the leakage of a SAVE instruction for different averaging and windowing parameters.

to reach the correct HWs by means of averaging and probability distribution tests\(^3\). In contrary to an algebraic side-channel attack which needs all correct HW of the target bytes to perform a successful key recovery attack [RSVC09], as we describe later in Section 8.4.3 our proposed attack is still able to recover the secrets if the attacker guesses the HWs within a window around the correct HWs. Figure 8.3 presents success rate of HW detection for different scenarios. In the figure, the number of traces for the same target byte which are used in averaging is indicated by “avg”. Further, “window” shows the size of a window which is defined around the correct HWs. As shown by Figure 8.3, to detect the correct HWs the adversary needs to repeat the measurements around 10 times, but defining a window by the size of 1 (i.e., correct HWs ±1) leads to the success rate of 100% considering only one measurement.

Differential Power Analysis

First, one may think that the best side-channel attack on implementations of the McEliece decryption scheme would be a DPA to reveal the secret key. However, the input (ciphertext) is processed in a bitwise fashion, and in contrary to symmetric block ciphers the secret key does not contribute as a parameter of a computation. Moreover, power traces for different ciphertexts would not be aligned to each other based on the computations, and execution time of decryption also varies for different ciphertexts. As a consequence, it is not possible to perform a classical DPA attack on our target implementations.

SPA on the Permutation Matrix

Considering implementation profiles I and II (defined in Section 8.4.2) the first secret information which is used in the decryption process is the permutation matrix \(P\). After permuting the ciphertext it is multiplied by the matrix \(H^T\). Since the multiplication of \(\hat{c}\) and \(H^T\) can be efficiently realized by summing up those rows of \(H\) for which corresponding bit of \(\hat{c}\) is “1” and skip all “0” bits, running time of multiplication depends on the number of “1”s (let say HW)

\(^3\)Probability distribution test here means to compare the probability distribution of the power values to the distribution of HW of random data in order to find the best match especially when highest (HW=8) or/and lowest (HW=0) is missing in measurements.
Chapter 8. General Security Considerations and New Side-Channel Attacks

Figure 8.4: Power traces of ciphertext (left) 0x0...01 and (right) 0x0...02.

Figure 8.5: Correlation vectors for ciphertexts (left) 0x0...01 and (right) 0x0...02.

As mentioned before the side-channel adversary would be able to detect the execution paths. If so, he can recover the content of $\hat{c}$ bit-by-bit by examining whether the summation is performed or not. However, HW of $\hat{c}$ is the same as HW of $c$, and only the bit locations are permuted. To recover the permutation matrix, the adversary can consider only the ciphertexts with HW=1 (2048 different ciphertexts in this case), and for each ciphertext finds the instant of time when the summation is performed (according to $\hat{c}$ bits). Sorting the time instants allows recovery whole of the permutation matrix. Figure 8.4 shows two power traces of start of decryption for two different ciphertexts. Obviously start of the summation is recognizable by visual inspection, but a general scheme (which is supposed to work independent of the implementation platform) would be similar to the scheme presented in [KKMP09]. That is, an arbitrary part of a trace can be considered as the reference pattern, and computing the cross correlation of the reference pattern and other power traces (for other ciphertexts with HW=1) reveals the positions in time when the summation takes place. Figure 8.5 presents two correlation vectors for the corresponding power traces of Figure 8.4. Note that to reduce the noise effect we have repeated the measurements and took the average over 10 traces for each ciphertext. Using this scheme for all ciphertexts with HW=1, permutation matrix is completely recovered.

SPA on the Parity Check Matrix

When implementation profiles III and IV are used, the permutation is not solely performed and hence the attack described above is not applicable. Therefore, the adversary has to take the multiplication process into account. Since in this case, execution path of multiplication does not depend on any secret, recovering the conditional branches (which only depend on ciphertext bits) would not help the attacker revealing the secrets. As a consequence the adversary has to
try revealing the content of the parity check matrix $H$. To do so, as described before he may reach (or guess) HW of the processed (or saved) data. Similarly to the last scheme the attacker can chose all ciphertexts with HW=1 and guess the HW of elements of each column of matrix $H$ separately. Since $27$ 11-bit elements of each column of $H$ are saved efficiently in a byte-wise fashion in 38-byte chunks\(^4\), and the adversary can only guess the HW of each byte, he can not certainly guess the HW of each 11-bit element of $H$. Therefore, the number of candidates for the HW of each 11-bit element is increased. As the result of this procedure, the adversary will have a set of candidates for each 11-bit element of parity matrix $H$ at row $i$ and column $j$ as follows:

$$\hat{H}_{i,j} = \left\{ h \in \{0,1\}^{11} \mid \text{HW}(h) = \text{the guessed HW by SPA } \pm \text{ window} \right\}.$$  

The effect of having an 11-bit element saved in two bytes, becomes clearer when considering the number of possible Goppa polynomials of degree 27. Overall there are $2^{11 \cdot 27} = 2^{297}$ possible polynomials. Using the known Hamming weight of the complete 11-bit coefficient, there are:

$$\left( \sum_{w=0}^{11} \left( \frac{11}{2^{11}} \right)^2 \right)^{27} = \left( \frac{88179}{256} \right)^{27} \approx 2^{227}$$  

possible candidates. But taking into account that the Hamming weight is split into a 3-bit and a 8-bit part, we have:

$$\left( \sum_{w_1=0}^{8} \left( \frac{8}{2^8} \right)^2 \right) \cdot \left( \sum_{w_2=0}^{3} \left( \frac{3}{2^3} \right)^2 \right)^{27} = \left( \frac{6435}{128} \cdot \frac{5}{2} \right)^{27} \approx 2^{188}$$  

possibilities.

**SPA on the Goppa Polynomial**

If the attacker can follow the execution path after the matrix multiplication, he would be able to measure the power consumption during the computation of the syndrome polynomial inversion (step 2 of Algorithm 5). Since at the start of this computation the Goppa polynomial is loaded, e.g., from a non-volatile memory to SRAM, similarly to the scheme explained above the adversary can predict HW of the transferred values, and hence make a list of candidates for each 11-bit element of the Goppa polynomial.

**8.4.3 Gains of Power Analysis Vulnerabilities**

This section discusses how to use the so far gathered information to perform a key recovery attack.

\(^4\)Each 11-bit can be saved in 2 bytes, but it wastes the memory and also simplifies the attack procedure by dividing the HW of an 11-bit value to the HW of two 8- and 3-bit parts.
Attack I: Knowing the Permutation Matrix

Given the permutation matrix $P$ (which is recovered by means of an SPA), we are able to completely break the system with one additional assumption. We need to know the original support $L$. In [EOS06], Section 3.1 it is stated that $L$ can be published without loss in security. Using the public key $\hat{G} = S \ast G \ast P$, we can easily recover $S \ast G$. Multiplication by a message with only a single “1” at position $i$ gives us row $S[i]$ because $G$ is considered to be in the systematic form. Therefore, by $(n - k)$ multiplications we can extract the scrambling matrix $S$ and consequently $G$ as well.

Now it is possible to recover the Goppa polynomial. According to Equation (6.5.4) we know that for a valid codeword (i.e., error free) the corresponding syndrome modulo $g(z)$ equals to zero. It means that the gcd of two different syndromes, which can now be computed by Equation (6.8.4) using $G' = S \ast G$ and the original support $L$, equals $g(z)$ with high probability. In our experiments, it never took more than one gcd-computation to recover the correct Goppa polynomial.

From this point on, we have extracted all parameters of the McEliece system, and hence are able to decrypt every ciphertext. In order to verify the revealed secrets, we executed the key generation algorithm with the extracted parameters and retrieved exactly the same secret key as in the original setup.

Attack II: Knowing the Parity Check Matrix

Without knowing the original support $L$, the attack described above is not applicable; moreover, in implementation profiles III and IV it is not possible to solely recover the permutation matrix. To overcome this problem we utilize the possible candidate lists $\hat{H}_{i,j}$ derived by an SPA attack. According to the structure of the parity check matrix $H$ in Equation (6.5.6), every column is totally defined by elements $\alpha$, $g(\alpha)$ and the coefficients of $g(z)$. We use this structure and the candidate lists in an exhaustive search. For every column $H[i]$ we randomly choose $\alpha_i$ and $g(\alpha_i)$ over all possible elements. These two elements are fixed for the entire column. Now we go recursively into the rows of column $i$. At every recursion level $j$ we have to choose a random value for $g_{t-j}$ and compute the actual value of $H[i][j]$ according to Equation (6.5.6). Only if this value is in the candidate list $\hat{H}_{i,j}$, we recursively call the search function for $H[i][j + 1]$. If a test fails, we remove the currently selected element for $g_{t-j}$ from the possible list and choose a new one. When the list gets empty, we return to one recursion level higher and try by a new element. Thereby we only go deeper into the search algorithm if our currently selected elements produce the values which are found in the corresponding candidate list. If the algorithm reaches row $[t + 1]$, with $t = 27$ in our case, we have selected candidates for $\alpha_i$, $g(\alpha_i)$, and all coefficients of the Goppa polynomial $g(z)$. Now we can check backwards whether $g(z)$ evaluates to $g(\alpha_i)$ at $\alpha_i$. If so, we have found a candidate for the Goppa polynomial and for the first support element.

While the above described algorithm continues to search new elements, we can validate the current one. By choosing another column $H[i]$ and one of the remaining $n - 1$ support elements,
we can test in \( t \) trials whether the given value exists in the corresponding candidate list. On success we additionally found another support element. Repeating this step \( n - 1 \) times reveals the order of the support \( \mathcal{L} \) and verifies the Goppa polynomial. Column four in Table 8.1 shows the average number of false \( \alpha \)s, that pass the first searched column for the right Goppa polynomial. However, these candidates are quickly sorted out by checking them against another column of \( H \). For all remaining pairs \( \langle \mathcal{L}, g(z) \rangle \) it is simply tested whether it is possible to decode an erroneous codeword.

Because a single column of \( H \) is sufficient for the first part of the attack, we could speed it up by selecting the column with the lowest number of candidates for the 27 positions. Depending on the actual matrix the number of candidates for a complete column varies between 1 000 and 25 000. It turns out that most often the column constructed by \( \alpha = 0 \) has the lowest number of candidates. So in a first try we always examine the column with lowest number of candidates with \( \alpha = 0 \) before iterating over other possibilities.

Also every information that one might know can speed up the attack. If, for example, it is known that a sparse Goppa polynomial is chosen, we can first test coefficient \( g_i = 0 \) before proceeding to other choices. For testing we generate a McEliece key from a sparse Goppa polynomial where only 4 coefficients are not zero. Table 8.1 shows the results for that key.

Even if the permutation matrix \( P \) is merged into the computation of \( H \) (implementation profiles III and IV) this attack reveals a permuted support \( \mathcal{L}_P \), which generates a parity check matrix capable of decoding the original ciphertext \( c \). As a result, although merging \( P \) and \( H \) is reasonable from a performance point of view, this eases our proposed attack.

**Attack III: Improving Attack II**

Considering the fact mentioned at the end of Section 8.4.2 knowing some information about the coefficients of \( g(z) \) dramatically reduces the number of elements to be tested on every recursion level. The use of additional information, here the HW of coefficients of \( g(z) \), significantly speeds up the attack, as shown in Table 8.2.

As mentioned in the previous section, Table 8.1 shows the results for a sparse Goppa polynomial. These result were achieved using a workstation PC equipped by two Xeon E5345 CPUs and 16 GByte RAM and gcc-4.4 together with OpenMP-3.0. The results for a full random Goppa polynomial are given in Table 8.2.

In this table a window size of \( X \) means that we do not use the information about the Goppa polynomial. Instead, we iterate over all possibilities. \( \#g(z) \) denotes the number of Goppa polynomials found until the correct one is hit, and \( \# \alpha \) indicates how many wrong elements fulfil even the first validation round. The column **CPU Time** is the time for a single CPU core.

**Countermeasures** Since the multiplication of the permuted ciphertext and parity check matrix \( H^T \) is efficiently implementing by summing up (XORing) some \( H \) rows which have “1” as the corresponding permuted ciphertext, the order of checking/XORing \( H \) rows can be changed arbitrarily. Since we have supposed that the attacker (partially) knows the program code, any fix
### Table 8.1: Runtime of the search algorithm for a sparse Goppa polynomial

<table>
<thead>
<tr>
<th>Window Size H</th>
<th>Window Size ( g(z) )</th>
<th>( # g(z) )</th>
<th>( # \alpha )</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( X ) ( &gt; 10^6 )</td>
<td>112</td>
<td>&lt; 1 sec</td>
<td>115 hours</td>
</tr>
<tr>
<td>1</td>
<td>( X ) ( &gt; 2^{32} )</td>
<td>( &gt; 2^{32} )</td>
<td>&lt; 1 sec</td>
<td>150 years</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3610</td>
<td>68</td>
<td>10 sec</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>112527</td>
<td>98</td>
<td>10 sec</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>793898</td>
<td>54</td>
<td>186 min</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( &gt; 10^6 )</td>
<td>112</td>
<td>71 days</td>
</tr>
</tbody>
</table>

### Table 8.2: Runtime of the search algorithm for a full random Goppa polynomial

<table>
<thead>
<tr>
<th>Window Size H</th>
<th>Window Size ( g(z) )</th>
<th>( # g(z) )</th>
<th>( # \alpha )</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( X ) ( &gt; 10^6 )</td>
<td>52</td>
<td>&lt; 1 sec</td>
<td>90 hours</td>
</tr>
<tr>
<td>1</td>
<td>( X ) ( &gt; 2^{32} )</td>
<td>( &gt; 2^{32} )</td>
<td>&lt; 1 sec</td>
<td>impossible</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4300</td>
<td>50</td>
<td>69 min</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101230</td>
<td>37</td>
<td>21 hours</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>( &gt; 2^{32} )</td>
<td>( &gt; 2^{32} )</td>
<td>26 days</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>( &gt; 2^{32} )</td>
<td>( &gt; 2^{32} )</td>
<td>5 years</td>
</tr>
</tbody>
</table>

change on the execution path, e.g., changing the order of summing up the \( H \) rows would not help to counteract our first attack (SPA on permutation matrix explained in Section 8.4.2). However, one can change the order of checking/XORing randomly for every ciphertext, and hence the execution path for a ciphertext in different instances of time will be different. Therefore, the adversary (which is not able to detect the random value and the selected order of computation) can not recover the permutation matrix. Note that as mentioned before if the permutation is not merely performed (e.g., in implementation profiles III and IV) our first attack is inherently defeated.

Defeating our second attack (SPA on parity check matrix explained in Section 8.4.2) is not as easy as that of the first attack. One may consider changing randomly the order of checking the \( H \) rows, which is described above, as a countermeasure against the second attack as well. According to the attack scenario the adversary examines the power traces for the ciphertexts with \( \text{HW}=1 \); then, by means of pattern matching techniques he would be able to detect at which instance of time the desired XOR operations (on the corresponding row of \( H \)) is performed. As a result, randomly changing the order to computations does not help to defeat the second attack. An alternative would be to randomly execute dummy instructions\(^5\). Though it leads to increasing the run time which is an important parameter for post quantum cryptosystems especially for software implementations, it extremely hardens our proposed attacks.

\(^5\)In our implementation platform it can be done by a random timer interrupt which runs a random amount of dummy instructions.
masking scheme may also provide robustness against our attacks. A simple way would be to randomly fill the memory location which stores the result of XORing $H$ rows before start of the multiplication (between the permuted ciphertext and the parity check matrix), and XORing the final results by the same start value. This avoids predicting HW of $H$ elements if the attacker considers only the leakage of the SAVE instructions. However, if he can use the leakage of LOAD instructions (those which load $H$ rows), this scheme does not help to counteract the attacks. One can make a randomly generated mask matrix as big as $H$, and save the masked matrix. Since in order to avoid the effect of the masking after multiplication it is needed to repeat the same procedure (multiplication) using the mask matrix, this scheme doubles the run time (for multiplication) and the area (for saving the mask matrix) as well though it definitely prevents our proposed attacks. As a result designing a masking scheme which is adopted to the limitations of our implementation platform is considered as a future work.

Note that the values in the second and last row of each table are only estimates. They are based on the progress of the search in around 2 weeks and on the knowledge of the right values. The impossible means, that there was only little progress and the estimate varied by hundreds of years.

Also it should be investigated whether the additional information from the side-channel attacks can improve one of the already known attacks, e.g., [BLP08b, LB88b, Leo88, Ste89]. The information gathered by means of side-channels ought to be useful since it downsizes the number of possibilities.

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.

### 8.5 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
Chapter 8. General Security Considerations and New Side-Channel Attacks

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? huMa? beans areborn free ?ld equal yn di?ntY 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 analysed 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. Conversions suitable for code-based cryptography are discussed in Section 9.

8.6 Key Length

The main caveat of code-based cryptosystems is the huge key length compared to other public-key cryptosystems. This is particularly troubling in the field of embedded devices, which have low memory resources 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. The implementation on a microcontroller is described in Chapter 11 and published in [Hey11]. It achieves a public key size reduction 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.
A very recent approach by Barreto and Misoczki [MTSB12] uses quasi cyclic MDPC codes. The implementation on an FPGA and a microcontroller is described in Chapter 12 and published in [SH13].

Using small non-binary 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.
Chapter 9
Conversions for CCA2-secure McEliece Variants

In [KI01] Kobara and Imai considered some conversions for achieving security against the critical attacks discussed in Section 8, and thus CCA2-security, in a restricted class of public-key cryptosystems. The authors reviewed these conversions for applicability to the McEliece public key cryptosystem and showed two of them to be convenient. These are Pointcheval’s generic conversion [Poi00] and Fujisaki-Okamoto’s generic conversion [FO99a] (Fujisaki-Okamoto Conversion (FOC)). Both convert partially trapdoor one-way functions Partially Trapdoor One-Way Function (PTOWF) \(^1\) to public-key cryptosystems fulfilling the CCA2 indistinguishability. The main disadvantage of both conversions is their high redundancy of data. Hence, Kobara and Imai developed three further specific conversions (Kobara-Imai-\(\gamma\) Conversion (KIC)) decreasing data overhead of the generic conversions even below the values of the original McEliece PKCs for large parameters.

<table>
<thead>
<tr>
<th>Conversion scheme</th>
<th>Data redundancy = ciphertext size - plaintext size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((n,k),t,r) ((2304,1280),64,160) ((2304,1280),64,256)</td>
</tr>
<tr>
<td>Pointcheval’s generic conv.</td>
<td>(n +</td>
</tr>
<tr>
<td>Fujisaki-Okamoto’s generic conversion</td>
<td>(n) 2304 2304</td>
</tr>
<tr>
<td>Kobara-Imai’s specific generic conversion</td>
<td>(n +</td>
</tr>
<tr>
<td>Kobara-Imai’s specific conversion (\alpha) and (\beta)</td>
<td>(n +</td>
</tr>
<tr>
<td>Kobara-Imai’s specific conversion (\gamma)</td>
<td>(-\lceil\log_2 (\gamma)\rceil - k) 927 1023</td>
</tr>
<tr>
<td>McEliece scheme w/o conv.</td>
<td>(n - k) 1024 1024</td>
</tr>
</tbody>
</table>

Table 9.1: Comparison between conversions and their data redundancy

Table 9.1 gives a comparison between the conversions mentioned above and their data overhead where \(r\) denotes a random value of typical length \(|r|\) equal to the output length of usual hash functions, e.g., SHA-1, SHA-256, and \(Const\) denotes a predetermined public constant of suggested length \(|Const|\)=160 bits. In addition, the data redundancy of the original McEliece system is given.

\(^1\) A PTOWF is a function \(F(x,y) \rightarrow z\) for which no polynomial time algorithm exists recovering \(x\) or \(y\) from their image \(z\) alone, but the knowledge of a secret enables a partial inversion, i.e., finding \(x\) from \(z\).
Chapter 9. Conversions for CCA2-secure McEliece Variants

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\textsuperscript{16.2.4}. Moreover, FOC requires a Hash function providing two different output lengths. Therefore we decided to use Keccak\textsuperscript{2} [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 as SHA-3. The reference implementation also includes a version optimized for 8-bit AVR microcontrollers, which has been used for our implementation.

9.1 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. 22 shows the Kobara-Imai-\(\gamma\) conversion applied to McEliece. It requires a constant string \(C\), a hash function \(H\), a cryptographically secure pseudo random string generator \(\text{Gen}(\text{seed})\) with a random seed and output of fixed length, a \(CW\) encoding and decoding function \(CW\) and \(CW^{-1}\), and the McEliece encryption \(E\) and decryption \(D\). 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.

KIC for Niederreiter KIC for McEliece has already been implemented and discussed in [Hey11]. 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 \(\text{Gen}(\text{seed})\) 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\textsuperscript{16.2.5}. 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.

\textsuperscript{2}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.
Algorithm 22 KOBARA-IMAI-γ CONVERSION APPLIED TO McELIECE

Encryption Input: Binary message \( m \), public constant \( C \)
Output: Ciphertext \( c \)

\[
\begin{align*}
y_1 & \leftarrow \text{Gen}(r) \oplus (m||C) \\
y_2 & \leftarrow r \oplus H(y_1) \\
(y_3||y_4||y_5) & \leftarrow (y_2||y_1) \\
e & \leftarrow \text{CW}(y_4) \\
\text{return } c & \leftarrow y_5||E_{\text{McEliece}}(y_3, e)
\end{align*}
\]

Decryption Input: Ciphertext \( c = (y_5||c_2) \)
Output: Binary message \( m \)

\[
\begin{align*}
(y_3, e) & \leftarrow D_{\text{McEliece}}^{\text{sec}}(c_2) \\
y_4 & \leftarrow \text{CW}^{-1}(e) \\
(y_2||y_1) & \leftarrow (y_3||y_4||y_5) \\
\hat{r} & \leftarrow y_2 \oplus H(y_1) \\
(y_4||C) & \leftarrow y_1 \oplus \text{Gen}(\hat{r}) \\
\text{IF } C = \hat{C} \text{ return } m & \leftarrow \hat{m} \\
\text{ELSE return } \bot
\end{align*}
\]

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 \( E_{\text{Niederreiter}}^K_{\text{pub}}(e) \). The message vector \( m \) as in \( E_{\text{McEliece}}^K_{\text{pub}}(m, e) \) 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\(^3\). On the contrary, KIC for Niederreiter adds an additional value to the ciphertext of the Niederreiter scheme and externalizes the message encryption completely.

Alg. 23 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 \( \text{Gen}(r) \) and the length of the seed \( r \) must be equal to the output length of the Hash function \( H \). 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 9.2 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 9.1.

\(^3\)Note that this can be changed by including the omitted value \( y_5 \).
Chapter 9. Conversions for CCA2-secure McEliece Variants

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Reason</th>
<th>C-Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Public constant</td>
<td>Chosen: 20 Bytes</td>
<td>CONSTBYTES</td>
</tr>
<tr>
<td>$H(\cdot)$</td>
<td>Hash output</td>
<td>Chosen: 32 Bytes</td>
<td>HASHBYTES</td>
</tr>
<tr>
<td>$m$</td>
<td>Message</td>
<td>Chosen: 20/100 Bytes</td>
<td>MESSAGEBYTES</td>
</tr>
<tr>
<td>$r$</td>
<td>Seed</td>
<td>$</td>
<td>r</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$</td>
<td>y_1</td>
<td>=</td>
</tr>
<tr>
<td>$y_2$</td>
<td>$</td>
<td>y_2</td>
<td>=</td>
</tr>
<tr>
<td>$y_3$</td>
<td>CW encoder</td>
<td>CWBYTES</td>
<td></td>
</tr>
<tr>
<td>$y_4$</td>
<td>$</td>
<td>y_4</td>
<td>=</td>
</tr>
<tr>
<td>$c$</td>
<td>Ciphertext</td>
<td>$</td>
<td>c</td>
</tr>
</tbody>
</table>

Table 9.2: Length of parameters for Kobara-Imai-γ applied to the Niederreiter scheme

Listing 9.1: Kobara-Imai-γ conversion applied to Niederreiter: Encryption

```c
array_rand (seed, HASHBYTES); // generate seed
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<MESSAGEBYTES && 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=0; i<MESSAGEBYTES; i++)
    y3[i-NR_CCA2_y4] = y2[i];
for (i=RANDBYTES+MESSAGEBYTES; i++)
    y3[i-NR_CCA2_y4] = y1[i-HASHBYTES];
// Encode y3 into array of error positions, stored in pk->error_pos
BtoCWit (pk, y3, CWBYTES);
// Encrypt error vector. y4 is already stored in pk->cca2_NR_KIC_y4
nr_encrypt_block (pk);
```
Algorithm 23  **KOBARA-IMAI-γ CONVERSION APPLIED TO NIEDERREITER**

**Encryption**

*Input*: Binary message $m$, public constant $C$

*Output*: Ciphertext $c$

```
y_1 \leftarrow \text{Gen}(r) \oplus (m||C)
y_2 \leftarrow r \oplus H(y_1)
(y_4||y_3) \leftarrow (y_2||y_1)
e \leftarrow CW(y_3)
```

{return} $c \leftarrow y_4|e_{Niederreiter}^{K_{pub}}(e)$

**Decryption**

*Input*: Ciphertext $c = (y_4||s)$

*Output*: Binary message $m$

```
e \leftarrow D_{Niederreiter}^{K_{sec}}(s)
y_3 \leftarrow CW^{-1}(e)
(y_2||y_1) \leftarrow (y_4||y_3)
\hat{r} \leftarrow y_2 \oplus H(y_1)
(\hat{m}||\hat{C}) \leftarrow y_1 \oplus \text{Gen}(\hat{r})$

IF $C = \hat{C}$ return $m \leftarrow \hat{m}$
ELSE return $\bot$
```

9.2  **Fujisaki-Okamoto Conversion**

The Fujisaki-Okamoto Conversion (FOC) [FO99b] 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 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.

Alg. 24 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 $\sigma$ which is used to generate a keystream using the Hash function $H_2$. This is used to encrypt the plaintext $m$ in a one-time pad [App. 16.2.5] fashion by XORing it with the keystream. To avoid CW encoding, $\sigma$ 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
Algorithm 24 Fujisaki-Okamoto conversion applied to McEliece

Encryption
Input: Binary message \( m \)
Output: Ciphertext \( c \)

\[
\begin{align*}
\sigma &\leftarrow \text{random vector of length } n \text{ and weight } t \\
r &\leftarrow \mathcal{H}_1(\sigma || m) \\
c_1 &\leftarrow \mathcal{E}_{K_{pub}}(r, \sigma) = r \cdot G + \sigma \\
c_2 &\leftarrow \mathcal{H}_2(\sigma) \oplus m \\
\text{return } c &\leftarrow (c_1, c_2)
\end{align*}
\]

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

\[
\begin{align*}
\hat{\sigma} &\leftarrow \mathcal{D}_{K_{sec}}(c_1) \\
\text{return } \bot &\text{ in case of decoding failure} \\
\hat{m} &\leftarrow \mathcal{H}_2(\hat{\sigma}) \oplus c_2 \\
\hat{r} &\leftarrow \mathcal{H}_1(\hat{\sigma} || \hat{m}) \\
\text{IF } c_1 &\leftarrow \mathcal{E}_{K_{pub}}(\hat{r}, \hat{\sigma}) \text{return } m \leftarrow \hat{m} \\
\text{ELSE return } \bot
\end{align*}
\]

encoding. Then, \( \sigma \) 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 \( \mathcal{E}_{K_{pub}}(\hat{m} = r, \hat{e} = \sigma) = \hat{m}G_{sys} + \hat{e} = rG_{sys} + \sigma \). 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 \( \sigma \) from the ciphertext using McEliece decryption. If decryption fails, the ciphertext may have been modified and the algorithm terminates with an error. From \( \sigma \), 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 \( r, \sigma \) or \( m \) have been detected to be modified and the algorithm terminates with an error.

Listing 9.2 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.
Listing 9.2: Decryption of Fujisaki-Okamoto conversion applied to McEliece

1 // Since the ciphertext (c1) is modified during McEliece decryption, but is required
2 // for later verification, it needs to be copied. Note that for efficiency reasons,
3 // the ciphertext is stored across the plaintext and ciphertext memory
4 matrix_clone(sk->plaintext, c1_copy_pt); matrix_clone(sk->ciphertext, c1_copy_ct);
5
6 // Decrypt c1 to obtain error positions (equivalent to sigma) and r.
7 // Note that we simply ignore r and instead recompute it from H1(sigma||m) later
8 mce_decrypt_block(sk);
9
10 // Construct sigma from error positions, but allocate an array large enough to hold (sigma||m)
11 uint8_t sigmax[n_in_bytes + MESSAGEBYTES];
12 MATRIX_FROM_ARRAY(sigma, 1, GOPPA_n, sigmax); // transform first part of sigmax to matrix sigma
13 matrix_zero(sigma); // set sigma to zero
14 for(i=0; i < CODE_ERRORS; i++) // iterate over error positions and set corresponding bits
15    MATRIX_SET1(sigma, 0, sk->error_pos[i]);
16
17 // Compute a hash only over sigma, \ie, the first part of sigmax.
18 // Store hash value directly to second part of sigmax, where m will be constructed
19 cbc_hash(&sigmax[n_in_bytes], sigmax, n_in_bytes);
20
21 // m = h2(sigma) XOR c2, \ie, XOR c2 to the second part of sigmax
22 for(i=0;i<MESSAGEBYTES;i++) sigmax[n_in_bytes+i] ^= sk->cca2_fujimoto_c2[i];
23
24 // compute hash of sigmax and write it to r, where it is taken from for encryption
25 uint8_t *r = sk->plaintext->data;
26 cbc_hash(r, sigmax, n_in_bytes+MESSAGEBYTES); h1(sigma||m)
27
28 mce_encrypt_block(sk); // if r G + sigma == c1: m is unmodified plaintext (SUCCESS)
29 if( matrix_cmp(sk->plaintext, c1_copy_pt) != 0 || matrix_cmp(sk->ciphertext, c1_copy_ct) != 0)
30    DIE("FAIL");
31
32 for(i=0; i<MESSAGEBYTES; i++) message[i]=sigmax[n_in_bytes+i]; // copy plaintext to output
Chapter 10

Microcontroller and FPGA Implementation of Codebased Crypto Using Plain Binary Goppa Codes

This research contribution is based on the author’s published research in [EGHP09, Hey10, Hey11, HG12]. It is joint work with Tim Güneysu and Hannes Hudde.

10.1 Previous Work

Although proposed more than 30 years ago, code-based encryption schemes have never gained much attention in practice due to their large secret and public keys. It was common perception for quite a long time that due to their expensive memory requirements such schemes are difficult to be integrated in any (cost-driven) real-world products. The original proposal by Robert McEliece for a code-based encryption scheme suggested the use of binary Goppa codes, but in general any other linear code could be used. While other types of codes may have advantages such as a more compact representation, most proposals using different codes were proven less secure (cf. eg. [MS07, OS08]). The Niederreiter cryptosystem is an independently developed variant of McEliece’s proposal which is proven to be equivalent in terms of security [LDW06]. In 2009, a first FPGA-based implementation of McEliece’s cryptosystem was proposed targeting a Xilinx Spartan-3AN and encrypts and decrypts data in 1.07 ms and 2.88 ms, using security parameters achieving an equivalence of 80-bit symmetric security [EGHP09]. The authors of [SWM+09] presented another accelerator for McEliece encryption over binary Goppa codes on a more powerful Virtex5-LX110T, capable to encrypt and decrypt a block in 0.5 ms and 1.4 ms providing a similar level of security. The latest publication [GDUV12] based on hardware/software co-design on a Spartan3-1400AN decrypts a block in 1 ms at 92 MHz\(^1\) at the same level of security. For x86-based platforms, a recent implementation of the McEliece scheme over binary Goppa codes is due to Biswas and Sendrier [BS08] achieving about 83-bit of equivalent symmetric security according to [BLP08b]. Comparing their implementation to other public-key schemes, it turns out that McEliece encryption can be faster than RSA and NTRU [BLP08a], however at the cost of larger keys. Many proposals already tried to ad-

\(^1\)This work does not provide performance results for encryption.
address this issue of large keys by replacing the original used binary Goppa codes with (secure) codes that allow more compact representations, e.g., [MB09, CHP12]. However, most of the attempts were broken [FOPT10a] and for the few (still) surviving ones hardly any implementations are available [BCGO09, Hey11]. Note further that most of these works exclusively target the McEliece cryptosystems.

### 10.2 Security Parameters.

Security parameters for cryptosystems need to be chosen in a way to provide sufficient protection against the best known attack, according to the requirements of the specific application. Due to recent improvements to Information Set Decoding, the assumed security level for the parameters originally suggested by McEliece fell from around $2^{80}$ in 1986 to $2^{59.9}$ in 2009 [FS09]. Table 10.1 shows parameter sets for typically used security levels and compares the resulting key size to conventional cryptosystems. It is clearly visible that the huge key size is the main caveat of Code-based cryptosystems.

### 10.3 8-Bit Microcontroller Implementation

In this section, we present the different aspects of our implementation. The main goal was to provide a high-performance C-implementation incorporating a broad range of methods and techniques from Code-based cryptography for arbitrary security parameters, tailored to the constrained execution environment of embedded devices. Our main target platform is the frequently used AVR ATxmega256A3, which is an 8-bit RISC microcontroller with 16 KBytes of SRAM and 256 KBytes of flash memory.

#### 10.3.1 Design Decisions

Due to the huge key lengths, implementing Code-based cryptosystems on low-memory platforms is challenging. However, such platforms are very common due to their low costs and use

<table>
<thead>
<tr>
<th>Security level</th>
<th>Code parameters</th>
<th>McEliece</th>
<th>Niederreiter</th>
<th>RSA</th>
<th>ECC / DLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 bit</td>
<td>10 1024 644 38</td>
<td>29 kB</td>
<td>80 kB</td>
<td>29 kB</td>
<td>47 kB</td>
</tr>
<tr>
<td>80 bit</td>
<td>11 2048 1751 27</td>
<td>63 kB</td>
<td>437 kB</td>
<td>63 kB</td>
<td>74 kB</td>
</tr>
<tr>
<td>128 bit</td>
<td>12 2960 2288 56</td>
<td>187 kB</td>
<td>826 kB</td>
<td>187 kB</td>
<td>242 kb</td>
</tr>
<tr>
<td>256 bit</td>
<td>13 6624 5129 115</td>
<td>936 kB</td>
<td>4 MB</td>
<td>936 kB</td>
<td>1208 kB</td>
</tr>
</tbody>
</table>

Table 10.1: Security parameters for Code-based and conventional public-key cryptosystems according to [BLP08b, Eur12]

Note that the tables states theoretical sizes, whereas our implementation stores matrices with an overhead of up to 7 bits per row.
in devices such as smartphones. Hence, efficient implementations are indispensable to improve the acceptance of Code-based cryptosystems. Therefore, memory-efficiency is necessarily an important goal. Hence we designed our implementation to provide a configurable balance between memory usage and performance, for example by optionally using precomputations and lookup tables and allowing access either from fast SRAM or slower flash memory according to the users needs.

Combinations of implemented schemes and methods can be used to the maximum possible extent. A notable exception is the Fujisaki-Okamoto conversion, which can be used only with McEliece. An adaption to the Niederreiter cryptosystem is not useful, since its biggest advantage is the absence of CW encoding, which is always required for Niederreiter. Security of key data stored on the device is ensured using 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 resources. Side channel security has been considered by avoiding data-dependent executions paths where possible.

Key generation is typically not executed on the microcontroller due to memory limitations, because multiple non-systematic matrices would need to be stored in memory at the same time.

### 10.3.2 CCA2-Secure Conversions

Our implementation includes two CCA2-secure conversions: the Kobara-Imai-γ conversion (KIC) and the Fujisaki-Okamoto conversion (FOC). KIC is based on a generic conversion by Pointcheval [Poi00] and can be applied to McEliece [KI01] and Niederreiter [NC11]. FOC [FO99b, CHP12] reduces the code complexity by removing the need for CW encoding, which is only possible for McEliece. Both conversions require the use of a Hash function, for which we use SHA-3 (Keccak) in this implementation.

Both conversions use the original cryptosystem mainly to encrypt some randomly generated string, which is used similarly to a keystream. A XOR operation is then used to encrypt a message of almost arbitrary length using the ‘keystream’. Consequently, for large message lengths the performance of both conversions is dominated mostly by the Hash function instead of by the original encryption scheme. KIC uses a publicly known constant to protect against modifications of the ciphertext, whereas FOC utilizes an additional McEliece encryption operation during encryption for this purpose. Since encryption is fast compared to decryption, this does not affect the performance considerably.

The conversion procedures are presented in Algs. 22, 23 and 24.

### 10.3.3 t-error Correction Using Berlekamp-Massey Decoder

While the Patterson algorithm is able to correct $t$ errors for binary Goppa codes, BM can usually correct only $\frac{t}{2}$ errors. However, for any square-free Goppa polynomial an equivalent code of double size can be constructed using the equivalence $Goppa(L, g(x)) \equiv Goppa(L, g(x)^2)$ (see
Section 6.8.8). A parity check matrix $H_2$ constructed from $\mathcal{L}$ and $g(x)^2$ can be used to compute a syndrome polynomial of degree $2t$ instead of $t$, which allows BM correct $t$ errors as well. This 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. Note that $H_2$ can be precomputed.

However, this approach cannot be used for Niederreiter, where the ciphertext is already a syndrome. In this case, the ciphertext needs to be transformed using a trick due to Sendrier (private communication in 2012). Recall that a syndrome $s$ of length $n - k$ corresponding to an erroneous codeword $\hat{c}$ satisfies the equation $s = S\hat{c} = e \cdot H^T$, where $e$ is the error vector that we want to obtain by decoding $s$. By prepending the a given syndrome $s$ with $k$ zeros, we obtain $(0|s)$ of length $n$, which is then multiplied with $H_2$ to obtain a new syndrome $s_2$. Since $(0|s)$ and $e$ belong to the same coset, $s_2$ is still a syndrome corresponding to $\hat{c}$ and having the same solution $e$. The first $k$ columns of $H_2$ do not need to be computed for the multiplication with $(0|s)$, reducing the memory requirements and allowing a faster implementation. Moreover, since $H_2$ is computed based on the secret support $\mathcal{L}$, the multiplication with $S^{-1}$ is not required anymore.

### 10.3.4 Adaptions and Optimizations

Several adaption to the target platform and optimizations to reach a balance between performance and memory usage have been implemented. This includes measures like the different syndrome computation variants for McEliece briefly discussed in Sec. 6.4 as well as measures taken to deal with peculiarities in the memory management on AVR microcontrollers concerning matrices larger than 32 kByte. Moreover, the implementation takes advantage of the fact that for binary codes, several operations like vector-matrix multiplication can be executed using fast word-wise XOR operations instead of element-wise additions modulo $p$.

**Fast field arithmetic**  
Field arithmetic is realized using a log and antilog lookup table containing polynomial and exponential representations of field elements. This allows fast arithmetic computations by using the most appropriate representation for each operation (e.g., polynomial for addition and exponential for multiplication), but has the disadvantage of requiring frequent conversions between representations.

Our implementation reduces unnecessary conversions by rewriting sequences of operations (e.g., loops containing alternating additions and multiplications, especially if locally constant values are involved) in important code parts. For example, the number of conversions during the evaluation of a polynomial of degree $t$ is reduced from $3t$ to $2t + 1$ using this approach. However, this results in a slightly increased code size, reducing the memory available for key data.

**Lookup tables**  
The computation of the syndrome during McEliece decryption requires an on-the-fly computation of the parity check matrix $H$ for higher security parameters, because then $H$ is too big to fit in memory. This computation is expensive and involves an evaluation of
the $g(x)$ for all support elements. This process can be sped up using a precomputed lookup table mapping all $p^n$ field elements $a_i$ to the result of $g(a_i)$, using 2 byte per table entry. This is a very efficient optimization, because it has a significant impact on the performance while occupying a comparably small amount of flash memory. Note that it applies only to McEliece used with on-the-fly syndrome computation.

10.3.5 µC Results

In this section we present performance measurements of our implementation, evaluate the effectiveness of our optimizations and compare the results to other implementations of Code-based and conventional cryptosystems. For brevity, we denote by MCE60, . . . , MCE256 the McEliece cryptosystem using parameters achieving a security level of 60-bit, . . . , 256-bit and respectively for the Niederreiter cryptosystem, denoted as NR60, . . . , NR256. The exact parameters used were given in Table 10.1.

The main memory requirements apart from the actual program code are shown in Table 10.1. One can see that the usage of systematic key matrices significantly reduces the memory requirements; nevertheless, in some cases (e.g., Niederreiter at 128-bit security) public and private keys do not fit into the available memory of our target platform at the same time. Hence, encryption and decryption have been measured separately in such cases. Note that encryption requires much more memory than decryption. Neither public nor private key fit into SRAM, so they need to be accessed from the slower flash memory.

Root extraction As previously mentioned, finding the roots of the error locator polynomial $\sigma(x)$ belongs to the most expensive computations. We implemented three methods of root extraction: Chien search, Horner scheme and Berlekamp-Trace algorithm (BTA). The latter was implemented in two variants, one dealing with large polynomials of degree $p^n-1$ and the other using with sparse polynomials, effectively dealing only with polynomials of degree $t$.

Table 10.2 compares these four algorithms and two optimizations applicable to root extraction applied to MCE128. FASTFIELD denotes the reduction of unnecessary conversions of field element representations. LREVERSE denotes a lookup table mapping support elements to their position in the permuted secret, which can be used to speed up Chien search and BTA. Horner scheme turns out to be faster than Chien search and BTA, since BTA suffers from a huge overhead due to the recursion and expensive handling of large polynomials and Chien search is an efficient solution only if it is parallelized. However, using the FASTFIELD optimization a performance gain of more than 20% is achieved.

Berlekamp-Massey vs. Patterson Table 10.3 compares the performance of decoding via BM or Patterson. Recall that BM essentially applies a single run of EEA, whereas Patterson is a more complicated process. Nevertheless, Patterson turns out to be faster in most cases. More precisely, for BM the overhead of computing a syndrome of double size respectively transforming the ciphertext to a syndrome of double size annihilates the slight performance advantages of BM over Patterson. Consequently, all in all decryption is usually faster with Patterson.
Chapter 10. Microcontroller and FPGA Implementation of Codebased Crypto Using Plain Binary Goppa Codes

<table>
<thead>
<tr>
<th>No optimization</th>
<th>Horner scheme</th>
<th>Chien search</th>
<th>BTA (Full)</th>
<th>BTA (Sparse)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15,321,280</td>
<td>26,407,900</td>
<td>159,589,242</td>
<td>309,507,754</td>
<td></td>
</tr>
<tr>
<td>FASTFIELD</td>
<td>12,719,902</td>
<td>23,946,730</td>
<td>138,404,862</td>
<td>296,648,695</td>
</tr>
<tr>
<td>LREVERSE</td>
<td>N/A</td>
<td>24,706,570</td>
<td>157,887,935</td>
<td>290,179,466</td>
</tr>
<tr>
<td>Both (FF + L⁻¹)</td>
<td>12,719,902</td>
<td>22,245,412</td>
<td>136,702,805</td>
<td>294,941,173</td>
</tr>
<tr>
<td>Gain (FF + L⁻¹)</td>
<td>20.45 %</td>
<td>18.71 %</td>
<td>16.74 %</td>
<td>4.94 %</td>
</tr>
</tbody>
</table>

Table 10.2: Cycle count of root extraction algorithms for MCE128 with different optimizations

<table>
<thead>
<tr>
<th>Cryptosystem</th>
<th>MCE80</th>
<th>MCE80</th>
<th>MCE128</th>
<th>NR80</th>
<th>NR128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syndrome</td>
<td>Precomputed $H$</td>
<td>On-the-fly</td>
<td>On-the-fly</td>
<td>Transform</td>
<td>Transform</td>
</tr>
<tr>
<td>Decryption (PAT)</td>
<td>6,196,454</td>
<td>15,597,028</td>
<td>44,125,930</td>
<td>5,577,774</td>
<td>16,508,937</td>
</tr>
<tr>
<td>Syndrome</td>
<td>942,940</td>
<td>10,364,767</td>
<td>28,715,171</td>
<td>141,563</td>
<td>615,416</td>
</tr>
<tr>
<td>Patterson</td>
<td>780,043</td>
<td>779,604</td>
<td>2,892,273</td>
<td>854,553</td>
<td>2,909,632</td>
</tr>
<tr>
<td>Decryption (BM)</td>
<td>6,868,866</td>
<td>23,697,815</td>
<td>71,130,775</td>
<td>5,510,006</td>
<td>17,082,953</td>
</tr>
<tr>
<td>Syndrome</td>
<td>1,702,513</td>
<td>18,546,210</td>
<td>55,609,478</td>
<td>255,228</td>
<td>1,191,349</td>
</tr>
<tr>
<td>BM</td>
<td>716,809</td>
<td>716,078</td>
<td>2,822,964</td>
<td>741,172</td>
<td>2,892,564</td>
</tr>
</tbody>
</table>

Table 10.3: Cycle count of decoding via Berlekamp-Massey or Patterson

**McEliece syndrome computation variants** Table 10.3 and 10.4 demonstrate the effect of the syndrome computation variants for McEliece. As expected, the computation using the precomputed parity check matrix $H$ is by far the fastest; however, it cannot be used for higher security parameters due to memory limitations. In this case, the on-the-fly computation without EEA is recommendable, since it much faster than the alternative on-the-fly computation. Nevertheless, even using the faster alternative, the syndrome on-the-fly computation is still approximately ten times slower than using $H$. However, using the Goppa polynomial lookup table described in Section 10.3.4 a performance gain of approximately 25% can be achieved.

<table>
<thead>
<tr>
<th>Syndrome computation variant</th>
<th>MCE60</th>
<th>MCE80</th>
<th>MCE128</th>
</tr>
</thead>
<tbody>
<tr>
<td>With precomputed parity check matrix $H$</td>
<td>590,807</td>
<td>942,940</td>
<td>N/A</td>
</tr>
<tr>
<td>On-the-fly computation with EEA</td>
<td>35,102,622</td>
<td>51,627,199</td>
<td>144,137,682</td>
</tr>
<tr>
<td>On-the-fly computation without EEA</td>
<td>7,283,086</td>
<td>10,340,083</td>
<td>30,482,209</td>
</tr>
</tbody>
</table>

Table 10.4: Comparison of syndrome computation variants for McEliece

**Encryption vs. Decryption** Table 10.5 shows performance results for KIC applied to McEliece and Niederreiter using several optimizations. Among other things, one can see that encryption is 2.5 to 10 times faster than decryption. This was to be expected since encryption consists mainly of a vector-matrix multiplication. Similar results can be found without the CCA2-secure conversion; in this case, encryption is even faster due to the missing Hash function call.
Note that nearly no optimizations have been applied to encryption, whereas decryption profits significantly from applied optimizations.

**McEliece vs. Niederreiter** Moreover it can be noticed that Niederreiter encryption is faster than McEliece encryption. Remember that in the McEliece scheme, a message (which may assumed to be uniformly distributed, hence having $k/2$ zero elements) is multiplied to the generator matrix, whereas in the Niederreiter scheme, an error vector with all but $t$ element set to zero is multiplied with the parity check matrix. Hence, the multiplication in Niederreiter can be implemented very efficiently, resulting in the better performance. McEliece decryption is dominated by root extraction if the precomputed parity check matrix is used, or by syndrome computation otherwise. Niederreiter decryption is always dominated by root extraction. Hence, Niederreiter is significantly faster as soon as the size of the security parameters demands the switch to on-the-fly syndrome computation in McEliece. Also without on-the-fly computations, Niederreiter turns out to be slightly faster.

**Constant weight encoding** Constant weight encoding has a significant effect on the code size, which is nearly doubled if CW encoding is used; however, its impact on the performance is negligible, as it makes up only approximately 1% of the total cycle count.

**Kobara-Imai-γ vs. Fujisaki-Okamoto** The main advantage of FOC is the avoidance of CW encoding, which makes FOC applicable only to McEliece. However, CW encoding has no significant effect on performance. Moreover, FOC uses two Hash function calls in both encryption and decryption, which consume a significant percentage of cycles (although this problem could maybe 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 due to the slow access to the key matrix in flash memory. Due to the additional encryption call, FOC requires the public key matrix also for decryption. Therefore, it comes with no surprise that FOC yields a worse performance than KIC. FOC 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 in Table 10.6.

**Comparison to other implementations** For completeness, we include a comparison of our implementation with other implementations of Code-based cryptosystems as well as comparable implementations of conventional cryptosystems such as RSA and ECC in Table 10.7. Frequencies differing from 32 Mhz (marked with *) were scaled accordingly to allow a fair comparison. One can see that our implementation outperforms previous implementations of McEliece and Niederreiter as well as comparable implementations of RSA and ECC in nearly all cases.
<table>
<thead>
<tr>
<th>McEliece</th>
<th>MCE80 (optimized)</th>
<th>MCE80</th>
<th>MCE128 (optimized)</th>
<th>MCE128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syndrome</td>
<td>I</td>
<td>I</td>
<td>III</td>
<td>III</td>
</tr>
<tr>
<td>FASTFIELD</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GF tables</td>
<td>SRAM (8 kB)</td>
<td>Flash (8 kB)</td>
<td>Flash (16 kB)</td>
<td>Flash (16 kB)</td>
</tr>
<tr>
<td>Support</td>
<td>SRAM (4 kB)</td>
<td>Flash (4 kB)</td>
<td>Flash (6 kB)</td>
<td>Flash (6 kB)</td>
</tr>
<tr>
<td>(g(x)) table</td>
<td>Flash (4 kB)</td>
<td>No</td>
<td>SRAM (8 kB)</td>
<td>Flash (8 kB)</td>
</tr>
<tr>
<td>bestU table</td>
<td>Flash (2 kB)</td>
<td>Flash (2 kB)</td>
<td>Flash (3 kB)</td>
<td>Flash (3 kB)</td>
</tr>
<tr>
<td>Plaintext</td>
<td>212 B</td>
<td>212 B</td>
<td>303 B</td>
<td>303 B</td>
</tr>
<tr>
<td>Operation</td>
<td>Cycles %</td>
<td>Cycles %</td>
<td>Cycles %</td>
<td>Cycles %</td>
</tr>
<tr>
<td>Encryption</td>
<td>2,644,139 25.67</td>
<td>2,644,297 22.82</td>
<td>5,277,682 11.05</td>
<td>5,278,004 9.07</td>
</tr>
<tr>
<td>CW encode</td>
<td>15,207 0.58</td>
<td>15,216 0.58</td>
<td>77,610 1.47</td>
<td>77,695 1.47</td>
</tr>
<tr>
<td>Encrypt</td>
<td>997,067 37.71</td>
<td>997,131 37.71</td>
<td>2,679,326 50.77</td>
<td>2,679,439 50.77</td>
</tr>
<tr>
<td>Hash</td>
<td>1,608,143 60.82</td>
<td>1,608,160 60.82</td>
<td>2,388,771 45.26</td>
<td>2,388,771 45.26</td>
</tr>
<tr>
<td>Decryption</td>
<td>7,655,240 74.33</td>
<td>8,944,729 77.18</td>
<td>42,500,066 88.95</td>
<td>52,901,084 90.93</td>
</tr>
<tr>
<td>Syndrome</td>
<td>931,060 12.16</td>
<td>931,047 10.41</td>
<td>24,040,858 56.57</td>
<td>31,349,307 59.26</td>
</tr>
<tr>
<td>Patterson</td>
<td>734,672 9.60</td>
<td>851,024 9.51</td>
<td>2,904,985 14.72</td>
<td>3,346,153 6.33</td>
</tr>
<tr>
<td>Roots</td>
<td>4,194,771 54.80</td>
<td>5,367,754 60.01</td>
<td>15,608,332 29.50</td>
<td>15,608,332 29.50</td>
</tr>
<tr>
<td>Hash</td>
<td>1,608,143 21.01</td>
<td>1,608,160 17.98</td>
<td>2,388,771 5.62</td>
<td>2,388,771 4.52</td>
</tr>
<tr>
<td>CW decode</td>
<td>19,720 0.25</td>
<td>19,269 0.22</td>
<td>35,094 0.08</td>
<td>35,081 0.07</td>
</tr>
<tr>
<td>Total</td>
<td>10,299,379</td>
<td>11,589,026</td>
<td>47,777,748</td>
<td>58,179,128</td>
</tr>
</tbody>
</table>

at 32 Mhz

<table>
<thead>
<tr>
<th>Operation</th>
<th>C/Byte Bit/s</th>
<th>C/Byte Bit/s</th>
<th>C/Byte Bit/s</th>
<th>C/Byte Bit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>12,472</td>
<td>7,897</td>
<td>12,472</td>
<td>7,897</td>
</tr>
<tr>
<td>CW encode</td>
<td>36,110</td>
<td>31,279</td>
<td>111,714</td>
<td>111,714</td>
</tr>
<tr>
<td>Hash</td>
<td>1,674,111</td>
<td>1,674,148</td>
<td>2,549,586</td>
<td>2,549,586</td>
</tr>
<tr>
<td>CW decode</td>
<td>1,573,701</td>
<td>1,573,701</td>
<td>2,352,512</td>
<td>2,352,512</td>
</tr>
<tr>
<td>Total</td>
<td>8,410,424</td>
<td>8,410,424</td>
<td>21,465,355</td>
<td>21,465,355</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>McEliece</th>
<th>MCE80 (optimized)</th>
<th>MCE80</th>
<th>MCE128 (optimized)</th>
<th>MCE128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syndrome</td>
<td>I</td>
<td>I</td>
<td>III</td>
<td>III</td>
</tr>
<tr>
<td>FASTFIELD</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GF tables</td>
<td>SRAM (8 kB)</td>
<td>Flash (8 kB)</td>
<td>Flash (12 kB)</td>
<td>Flash (12 kB)</td>
</tr>
<tr>
<td>Support</td>
<td>SRAM (4 kB)</td>
<td>Flash (4 kB)</td>
<td>Flash (6 kB)</td>
<td>Flash (6 kB)</td>
</tr>
<tr>
<td>(g(x)) table</td>
<td>Flash (4 kB)</td>
<td>No</td>
<td>SRAM (8 kB)</td>
<td>Flash (8 kB)</td>
</tr>
<tr>
<td>bestU table</td>
<td>Flash (2 kB)</td>
<td>Flash (2 kB)</td>
<td>Flash (3 kB)</td>
<td>Flash (3 kB)</td>
</tr>
<tr>
<td>Plaintext</td>
<td>212 B</td>
<td>212 B</td>
<td>303 B</td>
<td>303 B</td>
</tr>
<tr>
<td>Operation</td>
<td>Cycles %</td>
<td>Cycles %</td>
<td>Cycles %</td>
<td>Cycles %</td>
</tr>
<tr>
<td>Encryption</td>
<td>1,674,111 19.91</td>
<td>1,674,148 17.26</td>
<td>2,549,586 11.88</td>
<td>2,549,586 10.38</td>
</tr>
<tr>
<td>CW encode</td>
<td>15,329 0.92</td>
<td>15,327 0.92</td>
<td>27,387 1.07</td>
<td>27,387 1.07</td>
</tr>
<tr>
<td>Encrypt</td>
<td>31,279 1.87</td>
<td>31,278 1.87</td>
<td>111,714 3.58</td>
<td>111,714 3.58</td>
</tr>
<tr>
<td>Hash</td>
<td>1,573,701 94.00</td>
<td>1,573,701 94.00</td>
<td>2,352,512 92.27</td>
<td>2,352,512 92.27</td>
</tr>
<tr>
<td>Decryption</td>
<td>6,736,313 80.09</td>
<td>8,025,436 82.74</td>
<td>18,915,769 88.12</td>
<td>22,023,183 89.62</td>
</tr>
<tr>
<td>Descramble</td>
<td>142,143 2.11</td>
<td>141,873 1.77</td>
<td>620,331 3.28</td>
<td>620,331 3.28</td>
</tr>
<tr>
<td>Patterson</td>
<td>749,846 11.13</td>
<td>866,120 10.79</td>
<td>2,903,535 15.35</td>
<td>3,344,602 15.19</td>
</tr>
<tr>
<td>Roots</td>
<td>4,186,949 62.15</td>
<td>5,359,630 66.78</td>
<td>15,905,446 70.81</td>
<td>15,905,446 70.81</td>
</tr>
<tr>
<td>Hash</td>
<td>1,573,701 23.36</td>
<td>1,573,701 19.61</td>
<td>2,352,512 12.44</td>
<td>2,352,512 10.68</td>
</tr>
<tr>
<td>CW decode</td>
<td>20,084 0.29</td>
<td>20,070 0.25</td>
<td>35,633 0.16</td>
<td>35,633 0.16</td>
</tr>
<tr>
<td>Total</td>
<td>8,410,424</td>
<td>8,410,424</td>
<td>21,465,355</td>
<td>21,465,355</td>
</tr>
</tbody>
</table>

at 32 Mhz

<table>
<thead>
<tr>
<th>Operation</th>
<th>C/Byte Bit/s</th>
<th>C/Byte Bit/s</th>
<th>C/Byte Bit/s</th>
<th>C/Byte Bit/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>31,279</td>
<td>31,279</td>
<td>111,714</td>
<td>111,714</td>
</tr>
<tr>
<td>CW encode</td>
<td>1,573,701</td>
<td>1,573,701</td>
<td>2,352,512</td>
<td>2,352,512</td>
</tr>
<tr>
<td>Hash</td>
<td>2,549,586</td>
<td>2,549,586</td>
<td>92.27</td>
<td>92.27</td>
</tr>
<tr>
<td>CW decode</td>
<td>8,025,436</td>
<td>8,025,436</td>
<td>88.12</td>
<td>88.12</td>
</tr>
<tr>
<td>Total</td>
<td>21,465,355</td>
<td>21,465,355</td>
<td>24,572,769</td>
<td>24,572,769</td>
</tr>
</tbody>
</table>

Table 10.5: Optimized performance of McEliece and Niederreiter using Patterson decoder, KIC and Horner scheme
### Security level

<table>
<thead>
<tr>
<th>Security level</th>
<th>60-bit</th>
<th>79-bit</th>
<th>80-bit</th>
<th>128-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plaintext length</td>
<td>128 B</td>
<td>204 B</td>
<td>256 B</td>
<td>370 B</td>
</tr>
<tr>
<td>Hash $\mathcal{H}_1$ input</td>
<td>209 B</td>
<td>363 B</td>
<td>456 B</td>
<td>570 B</td>
</tr>
<tr>
<td>Hash $\mathcal{H}_2$ input</td>
<td>128 B</td>
<td>204 B</td>
<td>256 B</td>
<td>370 B</td>
</tr>
</tbody>
</table>

### Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
<th>%</th>
<th>Cycles</th>
<th>%</th>
<th>Cycles</th>
<th>%</th>
<th>Cycles</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encryption</td>
<td>2,925,603</td>
<td>16.64</td>
<td>6,430,846</td>
<td>22.69</td>
<td>7,345,516</td>
<td>24.13</td>
<td>12,133,694</td>
<td>17.19</td>
</tr>
<tr>
<td>$mG$</td>
<td>447,379</td>
<td>15.29</td>
<td>838,833</td>
<td>13.04</td>
<td>980,495</td>
<td>13.35</td>
<td>111,647</td>
<td>21.41</td>
</tr>
<tr>
<td>Hash $\mathcal{H}_1$</td>
<td>1,580,303</td>
<td>54.02</td>
<td>3,141,044</td>
<td>48.84</td>
<td>3,921,804</td>
<td>53.39</td>
<td>5,481,699</td>
<td>45.18</td>
</tr>
<tr>
<td>Hash $\mathcal{H}_2$</td>
<td>797,198</td>
<td>27.25</td>
<td>2,358,724</td>
<td>36.68</td>
<td>2,359,702</td>
<td>32.12</td>
<td>3,920,769</td>
<td>32.31</td>
</tr>
<tr>
<td>Cycles/Byte</td>
<td>22,856</td>
<td></td>
<td>31,523</td>
<td></td>
<td>28,693</td>
<td></td>
<td>32,793</td>
<td></td>
</tr>
<tr>
<td>Bit/s at 32 Mhz</td>
<td>11,200</td>
<td></td>
<td>8,121</td>
<td></td>
<td>8,922</td>
<td></td>
<td>7,806</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
<th>%</th>
<th>Cycles</th>
<th>%</th>
<th>Cycles</th>
<th>%</th>
<th>Cycles</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decryption</td>
<td>14,655,752</td>
<td>83.36</td>
<td>21,916,660</td>
<td>77.31</td>
<td>23,093,531</td>
<td>75.87</td>
<td>58,464,915</td>
<td>82.81</td>
</tr>
<tr>
<td>Syndrome</td>
<td>7,283,086</td>
<td>46.69</td>
<td>10,473,908</td>
<td>47.18</td>
<td>10,532,582</td>
<td>45.61</td>
<td>30,510,440</td>
<td>79.36</td>
</tr>
<tr>
<td>Patterson</td>
<td>1,393,479</td>
<td>9.51</td>
<td>1,078,481</td>
<td>4.92</td>
<td>747,599</td>
<td>3.24</td>
<td>2,905,102</td>
<td>4.97</td>
</tr>
<tr>
<td>Hash $\mathcal{H}_1$</td>
<td>1,580,303</td>
<td>10.78</td>
<td>3,141,044</td>
<td>14.33</td>
<td>3,921,804</td>
<td>16.98</td>
<td>5,481,699</td>
<td>9.38</td>
</tr>
<tr>
<td>Hash $\mathcal{H}_2$</td>
<td>797,198</td>
<td>5.44</td>
<td>2,358,724</td>
<td>10.76</td>
<td>2,359,702</td>
<td>10.22</td>
<td>3,920,769</td>
<td>6.71</td>
</tr>
<tr>
<td>Cycles/Byte</td>
<td>22,856</td>
<td></td>
<td>31,523</td>
<td></td>
<td>28,693</td>
<td></td>
<td>32,793</td>
<td></td>
</tr>
<tr>
<td>Bit/s at 32 Mhz</td>
<td>11,200</td>
<td></td>
<td>8,121</td>
<td></td>
<td>8,922</td>
<td></td>
<td>7,806</td>
<td></td>
</tr>
</tbody>
</table>

### Table 10.6: Performance of McEliece using the Fujisaki-Okamoto conversion

### 10.3.6 µC Conclusions

In this work, we presented an implementation of a broad range of methods and techniques from Code-based cryptography, tailored to the constrained execution environment of embedded devices such as the 8-bit microcontroller AVR ATxmega256A3. We included implementations of both the McEliece and Niederreiter cryptosystem and extended previous implementations providing only 80-bit security to the more suitable security level of 128-bit security. Higher security levels are possible and mainly 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.

The implementation 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.
## Chapter 10. Microcontroller and FPGA Implementation of Codebased Crypto Using Plain Binary Goppa Codes

Table 10.7: Comparison of performance of our implementation and comparable implementations of McEliece, Niederreiter, RSA and ECC

<table>
<thead>
<tr>
<th>System</th>
<th>Cycle count</th>
<th>Throughput (bit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80-bit McEliece using non-systematic key matrices on ATxMega192 @ 32MHz [EGHP09]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encryption</td>
<td>14,406,080</td>
<td>3,889</td>
</tr>
<tr>
<td>Decryption</td>
<td>19,751,094</td>
<td>2,835</td>
</tr>
</tbody>
</table>

| 80-bit Niederreiter on ATxMega192 @ 32MHz [Hey10] |  |  |
| Encryption | 51,247 | 119,890 |
| 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 $e = 2^{16} + 1$ | $\sim 3,440,000$ | 9,526 * |
| RSA-1024 private-key with Chinese Remainder Theorem (CRT) | $\sim 87,920,000$ | 373 * |
| RSA-2048 public-key $e = 2^{16} + 1$ | $\sim 15,520,000$ | 4223 * |
| RSA-2048 private-key with CRT | $\sim 666,080,000$ | 98 * |

| SECG-standardized ECC on ATMega128 @ 8MHz [GPW+04] |  |  |
| ECC-160 | $\sim 6,480,000$ | 790 * |
| ECC-192 | $\sim 9,920,000$ | 619 * |
| ECC-224 | $\sim 17,520,000$ | 409 * |

| RSA on ATMega128 [LGK10] |  |  |
| RSA-1024 | $\sim 76,000,000$ | 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 |
We implemented 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 demonstrated 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 resources 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. 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 work 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. We showed that 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.

Finally, we hope that this work will serve as incentive to extend the evaluation to other codes, to have a broad range of choices for future public-key schemes.

### 10.4 FPGA Implementation of the Niederreiter Scheme

This section describes our implementation primarily targeting a recent Virtex-6 LX240 FPGA. Note that this device is certainly too large for our implementation but was chosen due to its availability on the Xilinx Virtex-6 FPGA ML605 Evaluation Kit for testing. Furthermore, we provide implementations for a Xilinx Spartan-3 and Xilinx Virtex-5 to allow fair comparisons with other work (cf. Table 10.11).

#### 10.4.1 Encryption

The public key $\hat{H}$ is stored in an internal BRAM memory block and row-wise addressed by the output of the constant weight encoder. Multiplying a binary vector with a binary matrix is equivalent to a XOR operation of each row with input vector bit equal to one. Since this operation is trivial, we focus on the implementation of the constant weight encoding algorithm. Input data to our cryptosystem is passed using a FIFO with a non-symmetric 8-to-1 bit aspect.
Chapter 10. Microcontroller and FPGA Implementation of Codebased Crypto Using Plain Binary Goppa Codes

ratio. Hence, after a word with 8-bit length is written to the FIFO, it can be read out bit by bit. This is the equivalent to the binary stream reader presented in Algorithm 20. Its main part is implemented as a small finite state machine. Every time a valid \( \Delta[i] \) has been computed, it is directly transferred to the vector-matrix-multiplier summing up the selected rows. By interleaving operations we are able to process one bit from the FIFO at every clock cycle. After the last \( \Delta[t] \) has been computed, only the last indexed row of \( \hat{H} \) has to be added to the sum. Directly afterwards the encryption operation has finished and the ciphertext becomes available. Due to the very regular structure of the vector-matrix-multiplier and the small operands of the constant weight encoder, we were able to achieve a high clock frequency of 300 MHz. Nevertheless, the logic inferred by the constant-weight encoder is still the bottleneck.

![Block diagram of the encryption process.](image)

10.4.2 Decryption Using the Patterson Decoder

The first step in the decryption process is the multiplication by the inverse matrix \( S^{-1} \). This 11 KByte large matrix is stored in an internal BRAM and addressed by an incrementing counter. Using this BRAM, the rows of the matrix are XORed into an intermediate register if the corresponding input bit of the ciphertext equals to one. After \((n - k) = 297\) clock cycles, this register contains the value \( c' = S^{-1} \cdot c \) as shown in Alg. 19. Now \( c' \) is passed on to the Goppa decoder which return the error locator polynomial \( \sigma(x) \).
10.4.3 Decryption Using the Berlekamp-Massey Decoder

Instead of multiplying with $S^{-1}$, we have to multiply with the transformation matrix $H_2$ when using the Berlekamp-Massey decoder. As described above, we can use the same hardware architecture as for the Patterson decoder, with the only difference that the rows of the summed up matrix are twice as large. Remember, that we need to store only the last $n-k$ rows of $H_2$, because $c$ is prefix with zeros (see Sec. 6.8.3). Because the same amount of rows with twice the width have to be summed up this requires exactly the same number of cycles. The transformed syndrome is now passed to the Berlekamp-Massey decoder, which only consist of implementation of the EEA working modulo $g^2(x)$ and an stop value of $2\lfloor \frac{t}{2} \rfloor = 26$. This decoder also returns the error locator polynomial $\sigma(x)$.

Next, the roots of $\sigma(x)$ has to be computed in order to reveal the erroneous bit positions. Searching roots is a quite slow process that is highlighted by Fig. 10.2 showing our Chien search core. Decryption performance can be boosted by instantiating two or more of these cores in parallel and let them evaluate different support elements concurrently. Beside the additional management overhead in the controlling state machine, each of this cores requires additional 620 registers and 106 LUTs. We therefore use a single core which evaluates one support element in two clock cycles and finishes the entire process after 4098 clock cycles. Storing 28 look-up tables enables parallel execution of the multiplication but requires a significant amount of BRAM. Therefore, we decided to use 28 fully linearised multiplier instead, representing one output bit by a simple combinatorial circuit of the input bits.

Next each root needs to be mapped to these bit positions for which we used a permuted support $L$ as described above. Because the subsequent constant-weight decoding algorithm expects the distance between the error bits in ascending order, we appended a systolic implementation of bubble sort that returns sorted error positions. Simultaneously, the circuit computes the distance between two successive error positions. Finally, the error distances are translated into the binary message by a straightforward implementation of Alg. 21 as presented in Section 7.5.4.

10.4.4 FPGA Results

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

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

As mentioned above, the public-key cryptosystems RSA-1024 and ECC-P160 are assumed to roughly achieve an similar level of 80 bit symmetric security [EC08]. We finally compare our results to published implementations of these systems that target similar platforms (i.e., [EGHP09, SWM+09, GPP08, Hel08, BCE+01]). For a fair comparison with other existing implementations of code-based systems we also implemented our code for Spartan-3 and Virtex-5 FPGAs.

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

---

1 According to [EC08], RSA-1248 actually corresponds to 80 bit symmetric security. However, no implementation results for embedded systems are available for this key size.
Several extensions and improvements are possible, most notably the extension to non-binary codes alternant codes. 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. The integration of quasi-dyadic and quasi-cyclic codes represents an interesting approach at reducing the memory requirements of the implementation. However, it remains unknown whether quasi-dyadic or quasi-cyclic codes provide the same security as plain Goppa codes, due to the additional structure.

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 [Zin96] in the Berlekamp-Trace algorithm could
Table 10.9: Implementation results of Niederreiter decryption using Patterson decoding with \( n = 2048, k = 1751, t = 27 \) after PAR

<table>
<thead>
<tr>
<th>Aspect</th>
<th>S3-2000</th>
<th>V5-LX50</th>
<th>V6-LX240</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>11253 (54%)</td>
<td>4077 (56%)</td>
<td>3887 (10%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>15559 (37%)</td>
<td>9743 (33%)</td>
<td>9409 (6%)</td>
</tr>
<tr>
<td>FFs</td>
<td>13608 (33%)</td>
<td>13537 (47%)</td>
<td>12861 (4%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>22 (55%)</td>
<td>13 (21%)</td>
<td>9 (2%)</td>
</tr>
<tr>
<td>Frequency</td>
<td>95 MHz</td>
<td>180 MHz</td>
<td>250 MHz</td>
</tr>
</tbody>
</table>

\( c \cdot S^{-1} \) 297 cycles
\( S(x)^{-1} \) 4310 cycles
Solve Key Eq. 4854 cycles
Search Roots 4098 cycles
Sort&Convert 85 cycles
CW Decode 198 cycles

Table 10.10: Implementation results of Niederreiter decryption using a Berlekamp-Massey decoder with \( n = 2048, k = 1751, t = 27 \) after PAR

<table>
<thead>
<tr>
<th>Aspect</th>
<th>S3-2000</th>
<th>V5-LX50</th>
<th>V6-LX240</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>7331 (35%)</td>
<td>3190 (44%)</td>
<td>2159 (5%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>11380 (27%)</td>
<td>7821 (27%)</td>
<td>5567 (3%)</td>
</tr>
<tr>
<td>FFs</td>
<td>8049 (19%)</td>
<td>9106 (31%)</td>
<td>9166 (3%)</td>
</tr>
<tr>
<td>BRAMs</td>
<td>26 (65%)</td>
<td>14 (29%)</td>
<td>11 (2%)</td>
</tr>
<tr>
<td>Frequency</td>
<td>95 MHz</td>
<td>170 MHz</td>
<td>220 MHz</td>
</tr>
</tbody>
</table>

\( syn \cdot H_2 \) 297 cycles
Solve Key Eq. 6262 cycles
Search Roots 4098 cycles
Sort&Convert 85 cycles
CW Decode 198 cycles

help in finding an efficient alternative to the root extraction using a brute-force search with Horner scheme.
Table 10.11: Comparison of our Niederreiter designs with single-core ECC and RSA implementations for 80 bit security. Note that PAT designates Patterson decoding and BM Berlekamp-Massey decoding, respectively.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Platform</th>
<th>Resources</th>
<th>Freq</th>
<th>Time/Op</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work (enc)</td>
<td>Virtex6-LX240T</td>
<td>926 LUT/875 FF/17 BRAM</td>
<td>300 MHz</td>
<td>0.66 µs</td>
<td>8.3</td>
</tr>
<tr>
<td>This work (dec PAT)</td>
<td>Virtex6-LX240T</td>
<td>9,409 LUT/12,861 FF/9 BRAM</td>
<td>250 MHz</td>
<td>55.37 µs</td>
<td>576</td>
</tr>
<tr>
<td>This work (dec BM)</td>
<td>Virtex6-LX240T</td>
<td>5,567 LUT/9,166 FF/11 BRAM</td>
<td>220 MHz</td>
<td>49.72 µs</td>
<td>455</td>
</tr>
<tr>
<td>McEliece (enc) [EGHP09]</td>
<td>Spartan3-AN1400</td>
<td>1,044 LUT/804 FF/3 BRAM</td>
<td>150 MHz</td>
<td>1.070 µs</td>
<td>768</td>
</tr>
<tr>
<td>McEliece (dec) [EGHP09]</td>
<td>Spartan3-AN1400</td>
<td>9,054 LUT/12,870 FF/32 BRAM</td>
<td>85 MHz</td>
<td>21.610 µs</td>
<td>8,788</td>
</tr>
<tr>
<td>McEliece (dec) [GDUV12]</td>
<td>Spartan3-AN1400</td>
<td>2,979 slices</td>
<td>92 MHz</td>
<td>1.020 µs</td>
<td>430</td>
</tr>
<tr>
<td>This work (enc)</td>
<td>Spartan3-2000</td>
<td>1,252 LUT/869 FF/36 BRAM</td>
<td>150 MHz</td>
<td>1.32 µs</td>
<td>8.3</td>
</tr>
<tr>
<td>This work (dec PAT)</td>
<td>Spartan3-2000</td>
<td>15,559 LUT/13,608 FF/22 BRAM</td>
<td>95 MHz</td>
<td>1.45 µs</td>
<td>576</td>
</tr>
<tr>
<td>This work (dec BM)</td>
<td>Spartan3-2000</td>
<td>11,380 LUT/8,049 FF/26 BRAM</td>
<td>95 MHz</td>
<td>1.15 µs</td>
<td>455</td>
</tr>
<tr>
<td>McEliece (enc) [SWM+09]</td>
<td>Virtex5-LX110T</td>
<td>14,537 slices/75 BRAM</td>
<td>163 MHz</td>
<td>500 µs</td>
<td>389</td>
</tr>
<tr>
<td>McEliece (dec) [SWM+09]</td>
<td>Virtex5-LX110T</td>
<td>Combined with encryption</td>
<td>163 MHz</td>
<td>1.400 µs</td>
<td>1,091</td>
</tr>
<tr>
<td>McEliece (dec) [GDUV12]</td>
<td>Virtex5-LX110T</td>
<td>1,385 slices</td>
<td>190 MHz</td>
<td>500 µs</td>
<td>430</td>
</tr>
<tr>
<td>This work (enc)</td>
<td>Virtex5-LX50T</td>
<td>888 LUT/930 FF/18 BRAM</td>
<td>250 MHz</td>
<td>0.793 µs</td>
<td>8.2</td>
</tr>
<tr>
<td>This work (dec PAT)</td>
<td>Virtex5-LX50T</td>
<td>9,743 LUT/13,537 FF/13 BRAM</td>
<td>180 MHz</td>
<td>76.9 µs</td>
<td>576</td>
</tr>
<tr>
<td>This work (dec BM)</td>
<td>Virtex5-LX50T</td>
<td>7,821 LUT/9,106 FF/14 BRAM</td>
<td>170 MHz</td>
<td>64.4 µs</td>
<td>455</td>
</tr>
<tr>
<td>ECC-P160 (point mult.) [GPP08]</td>
<td>Spartan-3 1000-4</td>
<td>5,764 LUT/767 FF/5 BRAM</td>
<td>40 MHz</td>
<td>5.1 ms</td>
<td>10,200</td>
</tr>
<tr>
<td>ECC-K163 (point mult.) [SDI13]</td>
<td>Virtex5-LX110T</td>
<td>22,936 LUT/6,150 slices</td>
<td>250 MHz</td>
<td>5.48 µs</td>
<td>67.3</td>
</tr>
<tr>
<td>RSA-1024 random [Hel08]</td>
<td>Spartan-3A</td>
<td>1,813 slices/1 BRAM</td>
<td>133 MHz</td>
<td>48.54 ms</td>
<td>50,436</td>
</tr>
<tr>
<td>RSA-1024 random [Hel08]</td>
<td>Spartan-6</td>
<td>482 slices/1 BRAM</td>
<td>187 MHz</td>
<td>34.48 ms</td>
<td>50,373</td>
</tr>
<tr>
<td>RSA-1024 random [Hel08]</td>
<td>Virtex-6</td>
<td>478 slices/1 BRAM</td>
<td>339 MHz</td>
<td>19.01 ms</td>
<td>59,258</td>
</tr>
</tbody>
</table>
Chapter 11

Codebased Crypto Using Quasi Dyadic binary Goppa Codes

This research contribution is based on the author’s published research in [Hey11]. It is joint work with Olga Paustjan.

11.1 Scheme Definition of QD-McEliece

The main difference between the original McEliece scheme and the quasi-dyadic variant is the key generation Algorithm 25 shown below. It takes as input the system parameters $t$, $n$, and $k$ and outputs a binary Goppa code in quasi-dyadic form over a subfield $\mathbb{F}_p$ of $\mathbb{F}_q$, where $p = 2^s$ for some $s$, $q = p^d = 2^m$ for some $d$ with $m = ds$. The code length $n$ must be a multiple of $t$ such that $n = lt$ for some $l > d$.

Algorithm 25 QD-McEliece: Key generation algorithm

**Input:** Fixed common system parameters: $t$, $n = l \cdot t$, $k = n - dt$

**Output:** private key $K_{pr}$, public key $K_{pub}$

1. $(L_{dyad}, G(x) \ H_{dyad}, \eta) \leftarrow$ Algorithm 1 in [MB09] $(2^m, N, t)$, where $N >> n$, $N = l' \cdot t < q/2$

2. Select uniformly at random $l$ distinct blocks $[B_{i_0}|\cdots|B_{i_{l-1}}]$ in any order from $H_{dyad}$

3. Select $l$ dyadic permutations $\Pi^{j_0}, \cdots, \Pi^{j_{l-1}}$ of size $t \times t$ each

4. Select $l$ nonzero scale factors $\sigma_0, \ldots, \sigma_{l-1} \in \mathbb{F}_p$. If $p = 2$, then all scale factors are equal to 1.

5. Compute $H = [B_{i_0} \Pi^{j_0}|\cdots|B_{i_{l-1}} \Pi^{j_{l-1}}] \in (\mathbb{F}_q^{t \times t})^l$

6. Compute $\Sigma = \text{Diag}(\sigma_0 I_t, \ldots, \sigma_{l-1} I_t) \in (\mathbb{F}_p^{t \times t})^{l \times l}$

7. Compute the co-trace matrix $H'_{T_{\sigma}} = Tr'(H\Sigma) = Tr'(H)\Sigma \in (\mathbb{F}_p^{t \times t})^{l \times l}$

8. Bring $H'_{T_{\sigma}}$ in systematic form $H = [Q|I_{n-k}]$, e.g., by means of Gaussian elimination

9. Compute the public generator matrix $\hat{G} = [I_k|Q^T]$

10. **return** $K_{pub} = (\hat{G}, \ t)$, $K_{pr} = (H_{dyad}, L_{dyad}, \eta, \ G(x), \ (i_0, \ldots, i_{l-1}), \ (j_0, \ldots, j_{l-1}), \ (\sigma_0, \ldots, \sigma_{l-1}))$

The key generation algorithm proceeds as follows. It first runs Algorithm 1 in [MB09] to produce a dyadic code $C_{dyad}$ of length $N >> n$, where $N$ is a multiple of $t$ not exceeding the
The resulting code admits a $t \times N$ parity-check matrix $H_{dyad} = [B_0|\cdots|B_{N/t-1}]$ which can be viewed as a composition of $N/t$ dyadic blocks $B_i$ of size $t \times t$ each. In the next step the key generation algorithm uniformly selects $l$ dyadic blocks of $H_{dyad}$ of size $t \times t$ each. This procedure leads to the same result as puncturing the code $C_{dyad}$ on a random set of block coordinates $T_l$ of size $(N - n)/t$ first, and then permuting the remaining $l$ blocks by changing their order. The block permutation sequence $(i_0, \ldots, i_l)$ is the first part of the trapdoor information. It can also be described as an $N \times n$ permutation matrix $P_B$. Then the selection and permutation of $t \times t$ blocks can be done by right-side multiplication $H_{dyad} \times P_B$. Further transformations performed to disguise the structure of the private code are dyadic inner block permutations.

**Definition 11.1.1** A dyadic permutation $\Pi^j$ is a dyadic matrix whose signature is the $j$-th row of the identity matrix. A dyadic permutation is an involution, i.e., $(\Pi^j)^2 = I$. The $j$-th row (or equivalently the $j$-th column) of the dyadic matrix defined by a signature $h$ can be written as $\Delta(h)_j = h\Pi^j$.

The key generation algorithm first chooses a sequence of integers $(j_0, \ldots, j_{l-1})$ defining the positions of ones in the signatures of the $l$ dyadic permutations. Then each block $B_i$ is multiplied by a corresponding dyadic permutation $\Pi^j$ to obtain a matrix $H$ which defines a permutation equivalent code $C_H$ to the punctured code $C_{dyad}^\perp$. Since the dyadic inner-block permutations can be combined to an $n \times n$ permutation matrix $P_{dp} = \text{Diag}(\Pi^{i_0}, \ldots, \Pi^{i_{l-1}})$ we can write $H = H_{dyad} \cdot P_B \cdot P_{dp}$. The last transformation is scaling. Therefore, first a sequence $(\sigma_0, \ldots, \sigma_{l-1}) \in F_p$ is chosen, and then each dyadic block of $H$ is multiplied by a diagonal matrix $\sigma_i I_t$ such that $H' = H \cdot \Sigma = H_{dyad} \cdot P_B \cdot P_{dp} \cdot \Sigma$. Finally, the co-trace construction derives from $H'$ the parity-check matrix $H'_{Tr}$ for a binary quasi-dyadic permuted subfield subcode over $F_p$. Bringing $H'_{Tr}$ in systematic form, e.g., by means of Gaussian elimination, we obtain a systematic parity-check matrix $\hat{H}$ for the public code. $\hat{H}$ is still a quasi-dyadic matrix composed of dyadic submatrices which can be represented by a signature of length $t$ each and which are no longer associated to a Cauchy matrix. The generator matrix $\hat{G}$ obtained from $\hat{H}$ defines the public code $C_{pub}$ of length $n$ and dimension $k$ over $F_p$, while $\hat{H}$ defines a dual code $C_{pub}^\perp$ of length $n$ and dimension $k = n - dt$. The trapdoor information consisting of the essence $\eta$ of the signature $h_{dyad}$, the sequence $(i_0, \ldots, i_{l-1})$ of blocks, the sequence $(j_0, \ldots, j_{l-1})$ of dyadic permutation identifiers, and the sequence of scale factors $(\sigma_0, \ldots, \sigma_{l-1})$ relates the public code defined by $\hat{H}$ with the private code defined by $H_{dyad}$. The public code defined by $\hat{G}$ admits a further parity-check matrix $V_{L.*,G} = \text{vdm}(L^*, G(x)) \cdot \text{Diag}(G(L^*)^{-1})$ where $L^*$ is the permuted support obtained from $L_{dyad}$ by $L^* = L_{dyad} \cdot P_B \cdot P_{dp}$. Bringing $V_{L.*,G}$ in systematic form leads to the same quasi-dyadic parity-check matrix $\hat{H}$ for the code $C_{pub}$. The matrix $V_{L.*,G}$ is permutation equivalent to the parity-check matrix $V_{L,G} = \text{vdm}(L, G(x)) \cdot \text{Diag}(G(L)^{-1})$ for the shortened private code $C_{pr} = C_{dyad}^\perp$ obtained by puncturing the large private code $C_{dyad}$ on the set of block coordinates $T_l$. The support $L$ for the code $C_{pr}$ is obtained by deleting all components of $L_{dyad}$ at the positions indexed by $T_l$. Classical irreducible Goppa codes use support sets containing all elements of $F_q$. Thus, the support corresponding to such a Goppa code can be published.
11.1. Scheme Definition of QD-McEliece

while only the Goppa polynomial and the (support) permutation are parts of the secret key. In contrast, the support sets \( L \) and \( L^* \) for \( C_{pr} \) and \( C_{pub} \), respectively, are not full but just subsets of \( \mathbb{F}_q \) where \( L^* \) is a permuted version of \( L \). Hence, the support sets contain additional information and have to be kept secret.

The encryption algorithm of the QD-McEliece variant is the same as that of the original McEliece cryptosystem. First a message vector is multiplied by the systematic generator matrix \( \hat{G} \) for the quasi-dyadic public code \( C_{pub} \) to obtain the corresponding codeword. Then a random error vector of length \( n \) and hamming weight at most \( t \) is added to the codeword to obtain a ciphertext.

The decryption algorithm of the QD-McEliece version is essentially the same as that of the classical McEliece cryptosystem. The following decryption strategies are conceivable. Permute the ciphertext and undo the inner block dyadic permutation as well as the block permutation to obtain an extended permuted ciphertext of length \( N \) such that \( ct_{perm} = ct \cdot P_B \cdot P_{dp} \). Then use the decoding algorithm of the large private code \( C_{dyad} \) to obtain the corresponding codeword. Multiplying \( ct_{perm} \) by the parity-check matrix for \( C_{dyad} \) yields the same syndrome as reversing the dyadic permutation and the block permutation without extending the length of the ciphertext and using a parity-check matrix for the shortened private code \( C_{pr} \). A better method is to decrypt the ciphertext directly using the equivalent parity-check matrix \( V_{L^* G} \) for syndrome computation. Patterson’s decoding algorithm can be used to detect the error and to obtain the corresponding codeword. Since \( \hat{G} \) is in systematic form, the first \( k \) bits of the resulting codeword correspond to the encrypted message.

11.1.1 Parameter Choice and Key Sizes

For an implementation on an embedded microcontroller the best choice is to use Goppa codes over the base field \( \mathbb{F}_2 \). In this case the matrix vector multiplication can be performed most efficiently. Hence, the subfield \( \mathbb{F}_p = \mathbb{F}_{2^s} \) should be chosen to be the base field itself where \( s = 1 \) and \( p = 2 \). Furthermore, as the register size of embedded microcontrollers is restricted to 8 bits it is advisable to construct subfield subcodes of codes over \( \mathbb{F}_{2^8} \) or \( \mathbb{F}_{2^16} \). But the extension field \( \mathbb{F}_{2^8} \) is too small to derive secure subfield subcodes from codes defined over it.

Over the base subfield \( \mathbb{F}_2 \) of \( \mathbb{F}_{2^16} \) [MB09] suggests using the parameters summarized in Table 11.1.

As the public generator matrix \( \hat{G} \) is in systematic form, only its non-trivial part \( Q \) of length \( n - k = m \cdot t \) has to be stored. This part consists of \( m(l - m) \) dyadic submatrices of size \( t \times t \) each. Storing only the \( t \)-length signatures of \( Q \), the resulting public key size is \( m(l - m)t = m \cdot k \) bits in size. Hence, the public key size is a factor of \( t \) smaller compared to the generic McEliece version where the key even in systematic form is \( (n - k) \cdot k \) bits in size.

11.1.2 Security of QD-McEliece

A recent work [FOPT10a] presents an efficient attack recovering the private key in specific instances of the quasi-dyadic McEliece variant. Due to the structure of a quasi-dyadic Goppa code additional linear equations can be constructed. These equations reduce the algebraic
complexity of solving a multidimensional system of equations using Gröbner bases [AL94]. In the case of the quasi-dyadic McEliece variant there are \( l - m \) linear equations and \( l - 1 \) unknowns \( Y_i \). The dimension of the vector space solution for the \( Y'_i \)'s is \( m - 1 \). Once the unknowns \( Y_i \) are found all other unknowns \( X_i \) can be obtained by solving a system of linear equations. In our case there are 35 unknowns \( Y_i \), 20 linear equations, and the dimension of the vector space solution for the \( Y'_i \)'s is 15. The authors remark that the solution space is manageable in practice as long as \( m < 16 \). The attack was not successful with \( m = 16 \). Hence, up to now the McEliece variant using subfield subcodes over the base field of large codes over \( \mathbb{F}_{2^{16}} \) is still secure.

Conversions for CCA2-secure McEliece Variants As mentioned in Chapter 9, to achieve CCA2-security an additional conversion step is necessary. The generic conversions [Poi00, FO99a] both have the disadvantage of their high redundancy of data. Hence, Kobara and Imai developed three further specific conversions [KI01] (\( \alpha, \beta, \gamma \)) decreasing data overhead of the generic conversions even below the values of the original McEliece PKCs for large parameters. Their work shows clearly that the Kobara-Imai's specific conversion \( \gamma \) (KIC-\( \gamma \)) provides the lowest data redundancy for large parameters \( n \) and \( k \). In particular, for parameters \( n = 2304 \) and \( k = 1280 \) used in this work for the construction of the quasi-dyadic McEliece-type PKC the data redundancy of the converted variant is even below that of the original scheme without conversion.

11.2 Implementational Aspects

In this section we discuss aspects of our implementation of the McEliece variant based on quasi-dyadic Goppa codes of length \( n = 2304 \), dimension \( k = 1280 \), and correctable number of errors \( t = 64 \) over the subfield \( \mathbb{F}_2 \) of \( \mathbb{F}_{2^{16}} \) providing a security level of 80 bit. Target platform is the ATxmega256A1, a RISC microcontroller frequently used in embedded systems. This microcontroller operates at a clock frequency of up to 32 MHz, provides 16 Kbytes SRAM and 256 Kbytes Flash memory.
11.2. Field Arithmetic

To implement the field arithmetic on an embedded microcontroller most efficiently both representations of the field elements of \( \mathbb{F}_q \), polynomial and exponential, should be precomputed and stored as log- and antilog table, respectively. Each table occupies \( m \cdot 2^m \) bits of storage. Unfortunately, we cannot store the whole log- and antilog tables for \( \mathbb{F}_{2^{16}} \) because each table is 128 Kbytes in size. Neither the SRAM memory of the ATXmega256A1 (16 Kbytes) nor the Flash memory (256 Kbytes) would be enough to implement the McEliece PKC when completely storing both tables. Hence, we make use of tower field arithmetic (cf. Section 4.1.3). Efficient algorithms for arithmetic over tower fields are proposed in [Afa91, MK89, Paa94].

For the implementation it is important how to realize the mapping \( \varphi : A \rightarrow (a_1, a_0) \) of an element \( A \in \mathbb{F}_{2^{16}} \) to two elements \( (a_1, a_0) \in \mathbb{F}_{2^8} \), and the inverse mapping \( \varphi^{-1} : a_1, a_0 \rightarrow A \) such that \( A = a_1 \beta + a_0 \). Both mappings can be implemented by means of a special transformation matrix and its inverse, respectively [Paa94]. As the input and output for the McEliece scheme are binary vectors, field elements are only used in the scheme internally. Hence, we made an informed choice against the implementation of both mappings. Instead, we represent each field element \( A \) of \( \mathbb{F}_{2^{16}} \) as a structure of two uint8 values describing the elements of \( \mathbb{F}_{2^8} \) and perform all operations on these elements directly.

An element \( A \) of type \( \text{gf16}_t \) is defined by \( \text{gf16}_t \ A = \{A\cdot \text{highByte},A\cdot \text{lowByte}\} \). The tower field arithmetic can be performed through direct access to the elements \( a_1 = A\cdot \text{highByte} \) and \( a_0 = A\cdot \text{lowByte} \). The specific operations over \( \mathbb{F}_{2^8} \) are carried out through lookups in the precomputed log- and antilog tables for this field. The result of an arithmetic operation is an element of type \( \text{gf16}_t \) again.

Polynomials over \( \mathbb{F}_{2^{16}} \) are represented as arrays. For instance, we represent a polynomial \( G(x) = G_t x^t + \cdots + G_1 x + G_0 \) as an array of type \( \text{gf16}_t \) and size \( t+1 \) and store the coefficients \( G_i \) of \( G(x) \) such that \( \text{array}[i]\cdot \text{highByte} = G_{i,1} \) and \( \text{array}[i]\cdot \text{lowByte} = G_{i,0} \) where \( \varphi(G_i) = (G_{i,1},G_{i,1}) \).

The main problem when generating log- and antilog tables for a finite field is that there exist no exponential representation of the zero element, and thus, no explicit mapping \( 0 \rightarrow i \) such that \( 0 \equiv \alpha^i \), and vice versa. Hence, additional steps have to be performed within the functions for specific arithmetic operations to realize a correct zero-mapping. These additional computation steps reduce the performance of the tower field arithmetic but there is no way to avoid them.

11.2.2 Implementation of the QD-McEliece Variant

Encryption

The first step of the McEliece encryption is codeword computation. This is performed through multiplication of a plaintext \( p \) by the public generator matrix \( \hat{G} \) which serves as public key. In our case the public generator matrix \( \hat{G} = [I_k| M] \) is systematic. Hence, the first \( k \) bits of the codeword are the plaintext itself, and only the submatrix \( M \) of \( \hat{G} \) is used for the computation of the parity-check bits. \( M \in (\mathbb{F}_2^{k \times t}) \cdot d \cdot (l-d) \) can be considered as a composition of \( d \cdot (l-d) \) dyadic submatrices \( \Delta(h_{xy}) \) of size \( t \times t \) each, represented by a signature \( h_{xy} \) of length \( t \) each. It also
can be seen as a composition of \( l - d \) dyadic matrices \( \Delta(h, t) \) of size \( dt \times t \) each, represented by a signature of length \( dt = n - k \) each.

\[
M := \begin{pmatrix}
m_{0,0} & \cdots & m_{0,n-k-1} \\
\vdots & \ddots & \vdots \\
m_{t-1,0} & \cdots & m_{t-1,n-k-1} \\
m_{t,0} & \cdots & m_{t,n-k-1} \\
\vdots & \ddots & \vdots \\
m_{2t-1,0} & \cdots & m_{2t-1,n-k-1} \\
\vdots & \ddots & \vdots \\
m_{(l-d-1)t,0} & \cdots & m_{(l-d-1)t,n-k-1} \\
m_{(l-d)t,0} & \cdots & m_{(l-d)t-1,n-k-1}
\end{pmatrix}
\]

In both cases the compressed representation of \( M \) serving as public key \( K_{\text{pub}} \) for the McEliece encryption is

\[
K_{\text{pub}} = [(m_{0,0}, \cdots, m_{0,n-k-1}), \cdots, (m_{(l-d-1)t,0}, \cdots, m_{(l-d)t-1,n-k-1})].
\]

The public key is 2.5 KBytes in size and can be copied into the SRAM of the microcontroller at startup time for faster encryption. The plaintext

\[
p = (p_0, \cdots, p_{t-1}, p_t, \cdots, p_{2t-1}, \cdots, p_{(l-d-1)t}, \cdots, p_{(l-d)t-1})
\]

is a binary vector of length \( k = 1280 = 20 \cdot 64 = (l - d)t \). Hence, the codeword computation is done by adding the rows of \( M \) corresponding to the non-zero bits of \( p \). As we do not store \( M \) but just its compressed representation, only the bits \( p_i \) for all \( 0 \leq i \leq (l - d - 1) \) can be encrypted directly by adding the corresponding signatures. To encrypt all other bits of \( p \) the corresponding rows of \( M \) have to be reconstructed from \( K_{\text{pub}} \) first. The components \( h_{i,j} \) of a dyadic matrix \( \Delta(h, t) \) are normally computed as \( h_{i,j} = h_{j,i} \) which is a simple reordering of the elements of the signature \( h \). Unfortunately, we cannot use this equation directly because the public key is stored as an array of \((n - k)(l - d)/8\) elements of type \text{uint8_t}. Furthermore, for every \( t = 64 \) bits long substring of the plaintext a different length-\((n - k)\) signature has to be used for encryption. In Algorithm 26 we provide an efficient method for the codeword computation using a compressed public key.
**Algorithm 26** QD-McEliece encryption: Codeword computation

**Input:** plaintext array $p$ of type `uint8_t` and size $\lceil k/8 \rceil$ bytes, public key $K_{pub}$

**Output:** codeword array $cw$ of type `uint8_t` and size $n/8$ bytes

1. **INIT:** set the $k/8$ most significant bytes of $cw$ to $\text{MSB}_{k/8}(cw) \leftarrow p$. Set the remaining bytes of $cw$ to 0
2. for $j \leftarrow 0$ to $k/8 - 1$ by 8 do
3. Read 8 bytes = 64 bits of the plaintext
4. Determine the block key (signature of $\Delta(h_j, t)$)
5. for $i \leftarrow 0$ to 7 do
6. for all non-zero bits $x$ of $p[i]$ do
7. {compute the $(i \cdot 8 + x)$-th row of $\Delta(h_j, t)$}
8. if $x$ is odd then
9. $r_y \leftarrow (h_{j,y} \&0xAA) / 2), \forall y \in \{0, \ldots, (n - k)/8\}$
10. else
11. $r \leftarrow h_j$
12. end if
13. if $x \&0x02$ then
14. $r_y \leftarrow ((r_y \&0xCC) / 4) / ((r_y \&0x33) \cdot 4), \forall y \in \{0, \ldots, (n - k)/8\}$
15. end if
16. if $x \&0x04$ then
17. $r_y \leftarrow ((r_y \&0xF0) / 16) / ((r_y \&0x0F) \cdot 16), \forall y \in \{0, \ldots, (n - k)/8\}$
18. end if
19. {Byte permutations}
20. $\text{row}_{y,i} \leftarrow r_{y \oplus i}, \forall y \in \{0, \ldots, (n - k)/8\}$
21. {Add the row to the codeword}
22. $cw \leftarrow cw + \text{row}$
23. end for

**Decryption**

For decryption we use the equivalent shortened Goppa code $\Gamma(L^*, G(x))$ defined by the Goppa polynomial $G(x)$ and a (permuted) support sequence $L^* \subset F_{2^{16}}$. The support sequence consists of $n = 2304$ elements of $F_{2^{16}}$ and is 4.5 KBytes in size. We store the support sequence in an array of type `gf16_t` and size 2304. The Goppa polynomial is a monic separable polynomial of degree $t = 64$. As $t$ is a power of 2, the Goppa polynomial is sparse and of the form $G(x) = G_0 + \sum_{i=0}^6 G_{2^i} x^{2^i}$. Hence, it occupies just $8 \cdot 16$ bits storage space. We can store both the support sequence and the Goppa polynomial in the SRAM of the microcontroller.
Chapter 11. Codebased Crypto Using Quasi Dyadic binary Goppa Codes

Furthermore, we precompute the sequence $Diag(G(L^*_0)^{-1}, \ldots, G(L^*_{n-1})^{-1})$ for the parity-check matrix $V_{t,n}(L^*, D)$. Due to the construction of the Goppa polynomial $G(x) = \prod_{i=0}^{t-1} (x - z_i)$ where $z_i = 1/h_i + \omega$ with a random offset $\omega$, the following holds for all $G(L^*_{jt+i})^{-1}$.

\[ G(L^*_{jt+i})^{-1} = \prod_{r=0}^{t-1} (L^*_{jt+i} + z_r)^{-1} = \prod_{r=0}^{t-1} (1/h^*_{jt+i} + 1/h_r + 1/h_0)^{-1} = \prod_{r=0}^{jt+t-1} h^*_{jt+r} = \prod_{r=jt}^{jt+t-1} h^*_r \]

$h^*$ denotes a signature obtained by puncturing and permuting the signature $h$ for the large code $C_{dyad}$ such that $h^* = h \cdot P$ where $P$ is the secret permutation matrix. Hence, the evaluation of the Goppa polynomial on any element of the support block $(L^*_{jt}, \ldots, L^*_{jt+t-1})$ where $j \in \{0, \ldots, l-1\}$, $i \in \{0, \ldots, t-1\}$ leads to the same result. For this reason, only $n/t = l = 36$ values of type $\text{gf16}_t$ need to be stored. Another polynomial we need for Patterson’s decoding algorithm is $W(x)$ satisfying $W(x)^2 \equiv x \mod G(x)$. As the Goppa polynomial $G(x)$ is sparse, the polynomial $W(x)$ is also sparse and of the form $W(x) = W_0 + \sum_{i=0}^{6} W_{2i}x^{2i}$. $W(x)$ occupies $7 \cdot 16$ bits storage space.

**Syndrome Computation**

The first step of the decoding algorithm is the syndrome computation. Normally, the syndrome computation is performed through solving the equation $S_c(x) = S_e(x) \equiv \sum_{i \in E} \frac{1}{x - L^*_i} \mod G(x)$ where $E$ denotes a set of error positions. The polynomial $\frac{1}{x-L^*_i}$ satisfies the equation

\[ \frac{1}{x-L^*_i} \equiv \frac{1}{G(L^*_i)} \sum_{j=s+1}^{t} G_j L^*_{i-j}^{-s-1} \mod G(x), \ \forall 0 \leq s \leq t - 1 \quad \text{(11.2.1)} \]

The coefficients of this polynomial are components of the $i-th$ column of the Vandermonde parity-check matrix for the Goppa code $\Gamma(G(x), L^*)$. Hence, to compute the syndrome of a ciphertext $c$ we perform the on-the-fly computation of the rows of the parity-check matrix. As the Goppa polynomial is a sparse monic polynomial of the form $G(x) = G_0 + \sum_{i=0}^{6} G_{2i}x^{2i}$ with $G_{64} = 1$, we can simplify the Equation 11.2.1, and thus, reduce the number of operations needed for the syndrome computation. Algorithm 27 presents the syndrome computation procedure implemented in this work.
Algorithm 27 On-the-fly computation of the syndrome polynomial

Input: Ciphertext array \( c \) of type \texttt{uint8\_t} and size \( n/8 \) bytes, support set \( L^* \), Goppa polynomial \( G(x) = G_0 + \sum_{i=0}^{6} G_{2i} x^{2i} \) with \( G_{64} = 1 \)

Output: Syndrome \( S_c(x) = \sum_{i=0}^{l-1} S_{c,i} x^i \)

1: for \( i = 0 \) to \( n/8 \) do
2: for \( j = 0 \) to \( 7 \) do
3: if \( c_i \cdot 8 + j = 1 \) then
4: \{ compute the polynomial \( S'(x) = \frac{1}{x-L^*_i} \) mod \( G(x) \} \)
5: \( S'_{62} \leftarrow 1 \)
6: \( S'_{62} \leftarrow L^*_i \)
7: for \( r = 61 \) to \( 33 \) by \(-2\), \( s = 1 \) to \( 15 \) do
8: \( S'_r \leftarrow S'_{r+s}^2 \)
9: \( S'_{r-1} \leftarrow S'_{r+s} \cdot S'_{r+s-1} \)
10: end for
11: for \( r = 32 \) to \( 1 \) by \(-1 \) do
12: \( S'_{r-1} \leftarrow S'_r \cdot L^*_i \)
13: if \( r = 2^s \) then \{ for all powers of \( 2 \) only \}
14: \( S'_{r-1} \leftarrow S'_{r-1} + G_{2^s} \)
15: end if
16: end for
17: \( S_c(x) \leftarrow S_c(x) + S'(x)/G(L^*_i) \)
18: end if
19: end for
20: end for
21: return \( S_c(x) \)

The main advantage of this computation method is that it is performed on-the-fly such that no additional storage space is required. To speed-up the syndrome computation the parity-check matrix can be precomputed at the expense of additional \( n(n-k) = 288 \) KBytes memory. As the size of the Flash memory of ATxmega256A1 is restricted to 256 Kbytes, we cannot store the whole parity-check matrix. It is just possible to store 52 coefficients of each syndrome polynomial at most, and to compute the remaining coefficients on-the-fly. A better possibility is to work with the systematic quasi-dyadic public parity-check matrix \( \hat{H} = [Q^T | I_{n-k}] \) from which the public generator matrix \( \hat{G} = [I_k | Q] \) is obtained. To compute a syndrome the vector matrix multiplication \( \hat{H} \cdot c^T = c \cdot H^T \) is performed. For the transpose parity-check matrix \( \hat{H}^T = [Q^T | I_{n-k}]^T \) holds, where \( Q \) is the quasi-dyadic part composed of dyadic submatrices. Hence, to compute a syndrome we proceed as follows. The first \( k \) bits of the ciphertext are multiplied by the part \( Q \) which can be represented by the signatures of the dyadic submatrices. The storage space occupied by this part is 2.5 KBytes. The multiplication is performed in the same way as encryption of a plaintext (see Section 11.2.2) and results in a binary vector \( s' \)
Chapter 11. Codebased Crypto Using Quasi Dyadic binary Goppa Codes

of length \( n - k \). The last \( n - k \) bits of the ciphertext are multiplied by the identity matrix \( I_{n-k} \). Hence, we can omit the multiplication and just add the last \( n - k \) bits of \( c \) to \( s' \). To obtain a syndrome for the efficiently decodable code the vector \( s' \) first has to be multiplied by a scrambling matrix \( S \). We stress that this matrix brings the Vandermonde parity-check matrix for the private code \( \Gamma(G(x), L^*) \) in systematic form which is the same as the public parity-check matrix. Hence, \( S \) has to be kept secret. We generate \( S \) over \( \mathbb{F}_2 \) and afterwards represent it over \( \mathbb{F}_{2^{16}} \). Thus, the multiplication of a binary vector \( s' \) by \( S \) results in a polynomial \( S_c(x) \in \mathbb{F}_{2^{16}}[x] \) which is a valid syndrome. The matrix \( S \) is 128 KBytes in size and can be stored in the Flash memory of the microcontroller. The next step, which is computing the error locator polynomial \( \sigma(x) \), is implemented straightforward using Patterson’s algorithm as described in Section 6.8.4.

Searching for Roots of \( \sigma(x) \)

The last and the most computationally expensive step of the decoding algorithm is the search for roots of the error locator polynomial \( \sigma(x) \). For this purpose, we first planned to implement the Berlekamp trace algorithm [Ber70] which is known to be one of the best algorithm for finding roots of polynomials over finite fields with small characteristic. Considering the complexity of this algorithm we found out that it is absolutely unsuitable for punctured codes over a large field, because of the required computation of traces and gcds. The next root finding method we analyzed is the Chien search [Chi64] which has a theoretical complexity of \( \mathcal{O}(n \cdot t) \) if \( n = 2^m \). The Chien search scans automatically all \( 2^m - 1 \) field elements, in a more sophisticated manner than the simple polynomial evaluation method. Unfortunately, in our case \( n << 2^m \) such that the complexity of the Chien search becomes \( \mathcal{O}(2^{16} \cdot t) \) which is enormous compared to the complexity of the simple polynomial evaluation method. Another disadvantage of both the Berlekamp trace algorithm and the Chien search is that after root extraction the found roots have to be located within the support sequence to identify error positions. That is not the case when evaluating the error locator polynomial on the support set directly. In this case we know the positions of the elements \( L^*_i \) and can correct errors directly by flipping the corresponding bits in the ciphertext. The only algorithm which actually decreases the computation costs of the simple evaluation method in the case of punctured codes is the Horner scheme [Hor19]. The complexity of the Horner scheme does not depend on the extension degree of the field but on the number of possible root candidates, which is \( n \). In addition, as the Horner scheme evaluates the error locator polynomial on the support set \( L^* \), the root positions within \( L^* \) are known such that errors can be corrected more efficiently. Hence, we have implemented this root finding algorithm. After a root \( L^*_i \) of \( \sigma(x) \) has been found we perform the polynomial division of \( \sigma(x) \) by \( (x - L^*_i) \). We observed that the polynomial division by \( (x - L^*_i) \) can be performed sequentially reusing values computed in previous iteration steps. In the first step we compute the coefficient \( y_{t-2} \) of the searched polynomial \( y(x) \). In every iteration step \( j \) we use the previous coefficient \( y_{t-j+1} \) to compute \( y_{t-j} = y_{t-j+1}L^*_i + \sigma_{t-j} \). The whole procedure requires \( t - 3 \) multiplications and \( t - 2 \) additions to divide a degree-\( t \) polynomial by \( x - L^*_i \). The main advantage of performing polynomial division each time a root has been found is that the
degree of the error locator polynomial decreases. Hence, the next evaluation steps require less operations.

11.2.3 Implementation of the KIC-γ

For the implementation of Kobara-Imai’s specific conversion γ [KI01] two parameters have to be chosen: the length of the random value \( r \) and the length of the public constant \( Const \). The length of \( r \) should be equal to the output length of the used hash function. Here we choose the Blue Midnight Wish [GKK+09] (BMW) hash function, because of the availability of a fast assembly implementation. As we have \(|r| = 256\) and \(|Const| = 160\), the message to be encrypted should be of the length \(|m| \geq \left\lceil \log_2 \left( \binom{n}{k} \right) \right\rceil + k + |r| - |Const| = 1281\) bits. Hence, we encrypt messages of length 1288 bits = 161 bytes. In this case the data redundancy is even below of that of the McEliece scheme without conversion: \(1288/2304 \leq 1280/2304\).

The first steps of the KIC-γ encryption function are the generation of a random seed \( r \) for the function \( Gen(r) \), as well as the one-time-pad encryption of the message \( m \) padded with the public constant \( Const \) and the output of \( Gen(r) \). The result is a 1288 + 160 = 1448 bits = 181 bytes value \( y_1 \). In the next step the hash value of \( y_1 \) is added to the random seed \( r \) by the xor operation to obtain the value \( y_2 \). \( k = 1280\) bits from \((y_2||Y_1)\) are used as input for McEliece and from the remaining 424 bits the error vector is constructed by the constant weight encoding function \( \text{Conv[Sen05]} \) as described in Section 7.5.4.

To decrypt a ciphertext the KIC-γ first stores the first two bytes of the ciphertext in \( y_5 \). Then it calls the McEliece decryption function which returns the encrypted plaintext \( y_3 \) and the error vector \( \delta_j = i_j - i_{j-1} - 1 \) where \( i_r \) denote the error positions. To obtain part \( y_4 \) from the error vector constant weight decoding function is used. Now \((y_2||y_1) = (y_5||y_4||y_3)\) is known and the message \( m \) can be obtained.

11.3 Results on an 8-Bit Microcontroller

This section presents the results of our implementation of the McEliece variant based on [2304, 1280, 129] quasi-dyadic Goppa codes providing an 80-bit security level for the 8-bits AVR microcontroller. Due to the parameters chosen for KIC-γ the actual length of the message to be encrypted increases to 1288 bytes while the ciphertext length increases to 2312 bytes. Table 11.2 summarizes the sizes of all parameters being precomputed and used for the encryption and decryption algorithms.

Except for the matrix \( S \) which is used only within the syndrome computation method with pre-computation, all precomputed values can be copied into the faster SRAM of the microcontroller at startup time resulting in faster encryption and decryption. The performance results of our implementation were obtained from AVR Studio in version 4.18. Table 11.3 summarizes the clock cycles needed for specific operations and sub-operations for the conversion and encryption of a message. Note that we used fixed random values for the implementation of KIC-γ. The
Chapter 11. Codebased Crypto Using Quasi Dyadic binary Goppa Codes

<table>
<thead>
<tr>
<th>Encryption/Decryption</th>
<th>Parameter</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>QD-McEliece encryption</td>
<td>$K_{pub}$</td>
<td>2560 bytes</td>
</tr>
<tr>
<td>QD-McEliece decryption</td>
<td>log table for $\mathbb{F}_2^n$</td>
<td>256 bytes</td>
</tr>
<tr>
<td></td>
<td>antilog table for $\mathbb{F}_{2^8}$</td>
<td>256 bytes</td>
</tr>
<tr>
<td></td>
<td>Goppa polynomial $G(x)$</td>
<td>16 bytes</td>
</tr>
<tr>
<td></td>
<td>Polynomial $W(x)$</td>
<td>14 bytes</td>
</tr>
<tr>
<td></td>
<td>Support sequence $L^*$</td>
<td>4608 bytes</td>
</tr>
<tr>
<td></td>
<td>Array with elements $1/G(L^*_i)$</td>
<td>72 bytes</td>
</tr>
<tr>
<td></td>
<td>Matrix $S$</td>
<td>131072 bytes</td>
</tr>
<tr>
<td>KIC-$\gamma$</td>
<td>Public constant $\text{Const}$</td>
<td>20 bytes</td>
</tr>
</tbody>
</table>

Table 11.2: Sizes of tables and values in memory.

Encryption of a 1288 bits message requires 6,358,952 cycles. Hence, when running at 32 MHz, the encryption takes about 0.1987 seconds while the throughput is 6482 bits/second.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sub-operation</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td></td>
<td>15,083</td>
</tr>
<tr>
<td>CWencoding</td>
<td></td>
<td>50,667</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td>8,927</td>
</tr>
<tr>
<td>QD-McEliece encryption</td>
<td>Vector-matrix multiplication</td>
<td>6,279,662</td>
</tr>
<tr>
<td></td>
<td>Add error vector</td>
<td>4,613</td>
</tr>
</tbody>
</table>

Table 11.3: Performance of the QD-McEliece encryption including KIC-$\gamma$ on the AVR $\mu$C ATxmega256@32 MHz.

Table 11.4 presents the results of the operations and sub-operations of the QD-McEliece decryption function including KIC-$\gamma$.

Table 11.4 shows clearly that the error correction using the Horner scheme with polynomial division (PD) is about 40% faster than the Horner scheme without polynomial division. Considering the fact that the error correction is one of the most computationally expensive functions within the decryption algorithm the polynomial division provides a significant speed gain for this operation. In the case that the syndrome is computed using the precomputed matrix $S$ and the error correction is performed using the Horner scheme with polynomial division decoding of a 2312 bits ciphertext requires 33,535,287 cycles. Running at 32 MHz the decryption takes 1.0480 seconds while the ciphertext rate is 2206 bits/second\(^1\). Decryption with the on-the-fly syndrome computation method takes 50,161,743 cycles. Hence, running at 32 MHz the decryption of a ciphertext takes 1.5676 seconds in this case while the ciphertext rate is 1475 bits/second.

\(^1\)Ciphertext rate denotes number of ciphertext bits processed per second.
11.3. Results on an 8-Bit Microcontroller

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sub-operation</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>QD-McEliece</td>
<td>Syndrome computation on-the-fly</td>
<td>25,745,284</td>
</tr>
<tr>
<td></td>
<td>Syndrome computation with $S$</td>
<td>9,118,828</td>
</tr>
<tr>
<td></td>
<td>Syndrome inversion</td>
<td>3,460,823</td>
</tr>
<tr>
<td></td>
<td>Computing $\sigma(x)$</td>
<td>1,625,090</td>
</tr>
<tr>
<td></td>
<td>Error correction (HS)</td>
<td>31,943,688</td>
</tr>
<tr>
<td></td>
<td>Error correction (HS with PD)</td>
<td>19,234,171</td>
</tr>
<tr>
<td>CWdecoding</td>
<td></td>
<td>61,479</td>
</tr>
<tr>
<td>Hash</td>
<td></td>
<td>15,111</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td>19,785</td>
</tr>
</tbody>
</table>

Table 11.4: Performance of the QD-McEliece decryption on the AVR µC ATxmega256@32 MHz.

Although the on-the-fly decryption is about 1.5 times slower, no additional Flash memory is required so that a migration to cheaper devices is possible.

Table 11.5 summarizes the resource requirements of our implementation. The third column of the table refers to the decryption method with precomputed matrix $S$, the fourth to the on-the-fly syndrome decoding method. For a comparison we also provide the resource requirements for the McEliece version based on [2048,1751,55]-Goppa codes [EGHP09].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flash memory</th>
<th>External memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>QD-McEliece with KIC-γ</td>
<td>Encryption 11 Kbyte</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Decryption (with $S$) 156 Kbyte</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Decryption (on-the-fly) 21 Kbyte</td>
<td>–</td>
</tr>
<tr>
<td>McEliece[EGHP09]</td>
<td>Encryption 684 byte</td>
<td>438 Kbyte</td>
</tr>
<tr>
<td></td>
<td>Decryption 130.4 Kbyte</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 11.5: Resource requirements of QD-McEliece on the AVR µC ATxmega256@32 MHz.

As we can see, the memory requirements of the quasi-dyadic encryption routine including KIC-γ are minimal because of the compact representation of the public key. Hence, much cheaper microcontrollers such as ATxmega32 with only 4Kbytes SRAM and 32Kbytes Flash ROM could be used for encryption. In contrast, the implementation of the original McEliece version even requires 438 Kbyte external memory. The implementation of the decryption method with on-the-fly syndrome computation could also be migrated to a slightly cheaper microcontroller such as ATxmega128 with 8 Kbyte SRAM and 128 Kbyte Flash memory.

Table 11.6 gives a comparison of our implementation of the quasi-dyadic McEliece variant including KIC-γ with the implementation of the original McEliece PKC and the implementations of other public-key cryptosystems providing an 80-bit security level. RSA-1024 and ECC-160 [GPW+04] were implemented on an Atmel ATmega128 microcontroller at 8 MHz while the original McEliece version was implemented on an Atmel ATmega192 microcontroller at 32 MHz.
For a fair comparison with our implementation running at 32 MHz, we scale timings at lower frequencies accordingly.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (sec)</th>
<th>Throughput (bits/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QD-McEliece encryption</td>
<td>0.1987</td>
<td>6482</td>
</tr>
<tr>
<td>QD-McEliece decryption (with S)</td>
<td>1.0480</td>
<td>1229</td>
</tr>
<tr>
<td>QD-McEliece decryption (on-the-fly)</td>
<td>1.5676</td>
<td>822</td>
</tr>
<tr>
<td>McEliece encryption [EGHP09]</td>
<td>0.4501</td>
<td>3889</td>
</tr>
<tr>
<td>McEliece decryption [EGHP09]</td>
<td>0.6172</td>
<td>2835</td>
</tr>
<tr>
<td>ECC-160 [GPW+04]</td>
<td>0.2025</td>
<td>790</td>
</tr>
<tr>
<td>RSA-1024 $2^{16} + 1$ [GPW+04]</td>
<td>0.1075</td>
<td>9525</td>
</tr>
<tr>
<td>RSA-1024 w. CRT [GPW+04]</td>
<td>2.7475</td>
<td>373</td>
</tr>
</tbody>
</table>

Table 11.6: Comparison of the quasi-dyadic McEliece variant including KIC-$\gamma$ ($n'=2312$, $k'=1288$, $t=64$) with original McEliece PKC ($n=2048$, $k=1751$, $t=27$), ECC-P160, and RSA-1024

Although we additionally include KIC-$\gamma$ in the quasi-dyadic McEliece encryption, we were able to outperform both, the McEliece version from [EGHP09] and ECC-160, in terms of number of operations per second. In particular, the throughput of our implementation significantly exceeds that of ECC-160.

Unfortunately, we could not outperform the McEliece scheme [EGHP09] neither in throughput nor in number of operations per second for the decryption. The reason is that this implementation is based on Goppa codes with much smaller number of errors $t=27$. Due to this fact, it works with polynomials of smaller degree such that most operations within the decoding algorithm can be performed more efficiently. Another disadvantage of our implementation is that all parameters are defined over the large field $F_{2^{16}}$. As we could not store the log- and antilog tables for this field in the Flash memory, we had to implement the tower field arithmetic which significantly reduces performance. For instance, one multiplication over a tower $F_{(2^8)^2}$ involves 5 multiplications over the subfield $F_{2^8}$. Hence, much more arithmetic operations have to be performed to decrypt a ciphertext.

Nevertheless, the decryption function is still faster than the RSA-1024 private key operation and exceeds the throughput of ECC-160. Furthermore, although slower, the on-the-fly decoding algorithm requires 81% less memory compared to the original McEliece version such that migration to cheaper devices is possible.

### 11.4 Conclusion and Further Research

In this work we have implemented a McEliece variant based on quasi-dyadic Goppa codes on a 8-bits AVR microcontroller. The family of quasi-dyadic Goppa codes offers the advantage of
11.4. Conclusion and Further Research

having a compact and simple description. Using quasi-dyadic Goppa codes the public key for
the McEliece encryption is significantly reduced. Furthermore, we used a generator matrix for
the public code in systematic form resulting in an additional key reduction. As a result, the
public key size is a factor $t$ less compared to generic Goppa codes used in the original McEliece
PKC. Moreover, the public key can be kept in this compact size not only for storing but for
processing as well. However, the systematic coding necessitates further conversion to protect the
message. Without any conversions the encrypted message would be revealed immediately from
the ciphertext. Hence, we have implemented Kobara-Imai’s specific conversion $\gamma$: a conversion
scheme developed specially for CCA2 secure McEliece variants.

Our implementation out performs the implementations of [EGHP09] and ECC-160 in encryp-
tion. In particular, the quasi-dyadic McEliece encryption is 2.3 times faster than [EGHP09] and
exceeds the throughput of both, the original McEliece PKC and ECC-160, by 1.7 and 8.2 times,
respectively. In addition, our encryption algorithm requires 96.7% less memory compared to
the original McEliece version and can be migrated to much cheaper devices.

The performance of the McEliece decryption algorithm is closely related to the number of
errors added within the encryption. In our case the number of errors is 64 which is 2.4 times
greater compared to the original McEliece PKC. Hence, the polynomials used are huge and
the parity-check matrix is too large to be completely precomputed and stored in the Flash
memory. In addition, the error correction requires more time because a polynomial of degree
64 has to be evaluated. We showed in Section 11.2.2 that none of the frequently used error
correction algorithms, such as the Berlekamp trace algorithm and the Chien search, are suitable
for punctured and shortened codes obtained from codes over very large fields. Furthermore,
the tower field arithmetic significantly reduces the performance of the decoding algorithm.
Nevertheless, the decryption algorithms with precomputation and on-the-fly computation are
2.6 and 1.8 times faster than the RSA-1024 private key operation and exceed the throughput
of ECC-160. Furthermore, although slower, the on-the-fly decoding algorithm requires 81% less
memory compared to the original McEliece version such that migration to cheaper devices is
possible.
Chapter 12

Codebased Crypto Using Quasi Cyclic Medium
Density Parity Check Codes

This research contribution is based on the author’s published research in [SH13]. It is joint work with Ingo von Maurich and Tim Güneysu.

12.1 McEliece Based on QC-MDPC Codes

We now present the implementation of a variant of the McEliece cryptosystem based on \((n, r, w)\)-QC-MDPC codes with \(n = n_0 p\) and \(r = p\). To obtain such a code, we first pick a word \(h \in \mathbb{F}_2^n\) of length \(n = n_0 p\) and weight \(w\) at random. Then, the QC-MDPC code is defined by a quasi-cyclic parity-check matrix \(H \in \mathbb{F}_2^n\) of first row \(h\) and all other \(r - 1\) rows are obtained from \(r - 1\) quasi-cyclic shifts of \(h\). The parity-check matrix then has the form \(H = [H_0|H_1|...|H_{n_0-1}].\) Each block \(H_i\) has row weight \(w_i\), such that \(w = \sum_{i=0}^{n_0-1} w_i\) with a smooth distribution of \(w_i\)'s. Finally, the generator matrix \(G\) in row reduced echelon form can be easily derived from the \(H_i\) blocks. Assuming that \(H_{n_0-1}\) is non-singular (this particularly implies \(w_{n_0-1}\) being odd, otherwise the rows of \(H_{n_0-1}\) would sum up to 0), we compute \(G\) of the form \((I|Q),\) where \(I\) is the identity matrix and

\[
Q = \begin{pmatrix}
(H_{n_0-1}^{-1} \cdot H_0)^T \\
(H_{n_0-1}^{-1} \cdot H_1)^T \\
\vdots \\
(H_{n_0-1}^{-1} \cdot H_{n_0-2})^T
\end{pmatrix},
\]

In the following we detail the key-generation as well as encryption and decryption for McEliece based on QC-MDPC codes.

- **Key-Generation**: The public and private keys are generated as follows. First generate a parity-check matrix \(H \in \mathbb{F}_2^{n \times n}\) of a \(t\)-error-correcting \((n, r, w)\)-QC-MDPC code. Then generate its corresponding generator matrix \(G \in \mathbb{F}_2^{(n-r) \times n}\) in row reduced echelon form. The public key is \(G\) and the private key is \(H\). Since quasi-cyclic matrices are used, it suffices to store the first rows \(g\) and \(h\) of the circulant blocks which significantly reduces storage requirements.
Chapter 12. Code-based Crypto Using Quasi Cyclic Medium Density Parity Check Codes

- **Encryption:** To encrypt a plaintext \( m \in \mathbb{F}_2^{(n-r)} \) into \( x \in \mathbb{F}_2^n \), first generate an error vector \( e \in \mathbb{F}_2^n \) of \( \text{wt}(e) \leq t \) at random. Then compute \( x \leftarrow mG + e \).

- **Decryption:** Let \( \Psi_H \) be a \( t \)-error-correcting LDPC/MDPC decoding algorithm equipped with the sparse parity-check matrix \( H \). To decrypt \( x \in \mathbb{F}_2^n \) into \( m \in \mathbb{F}_2^{(n-r)} \) compute \( mG \leftarrow \Psi_H(mG + e) \). Finally extract the plaintext \( m \) from the first \( (n-r) \) positions of \( mG \).

12.2 Security of QC-MDPC

The description of McEliece based on QC-MDPC codes in Section 12.1 eliminates the scrambling matrix \( S \) and the permutation matrix \( P \) usually used in the McEliece cryptosystem. The use of a CCA2-secure conversion (e.g., [KI01]) allows \( G \) to be in systematic-form without introducing any security-flaws. Note that [MTSB12] states that a quasi-cyclic structure, by itself, does not imply a significant improvement for an adversary. All previous attacks on McEliece schemes are based on the combination of a quasi-cyclic/dyadic structure with some algebraic code information. To resist the best currently known attack of [BJMM12b] and also the improvements achieved by the DOOM-attack [Sen11], the authors of [MTSB12] suggest parameters as given in Table 12.1.

<table>
<thead>
<tr>
<th>Security Level</th>
<th>( n_0 )</th>
<th>( n )</th>
<th>( r )</th>
<th>( w )</th>
<th>( t )</th>
<th>Public key size</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 bit</td>
<td>2</td>
<td>9600</td>
<td>4800</td>
<td>90</td>
<td>84</td>
<td>4800 bit</td>
</tr>
<tr>
<td>80 bit</td>
<td>3</td>
<td>10752</td>
<td>3584</td>
<td>153</td>
<td>53</td>
<td>7168 bit</td>
</tr>
<tr>
<td>80 bit</td>
<td>4</td>
<td>12288</td>
<td>3072</td>
<td>220</td>
<td>42</td>
<td>9216 bit</td>
</tr>
<tr>
<td>128 bit</td>
<td>2</td>
<td>19712</td>
<td>9856</td>
<td>142</td>
<td>134</td>
<td>9856 bit</td>
</tr>
<tr>
<td>128 bit</td>
<td>3</td>
<td>22272</td>
<td>7424</td>
<td>243</td>
<td>85</td>
<td>14848 bit</td>
</tr>
<tr>
<td>128 bit</td>
<td>4</td>
<td>27200</td>
<td>6800</td>
<td>340</td>
<td>68</td>
<td>20400 bit</td>
</tr>
<tr>
<td>256 bit</td>
<td>2</td>
<td>65536</td>
<td>32768</td>
<td>274</td>
<td>264</td>
<td>32768 bit</td>
</tr>
<tr>
<td>256 bit</td>
<td>3</td>
<td>67584</td>
<td>22528</td>
<td>465</td>
<td>167</td>
<td>45056 bit</td>
</tr>
<tr>
<td>256 bit</td>
<td>4</td>
<td>81920</td>
<td>20480</td>
<td>644</td>
<td>137</td>
<td>61440 bit</td>
</tr>
</tbody>
</table>

12.3 Decoding (QC-)MDPC Codes

For code-based cryptosystems, decoding a codeword (i.e., the syndrome) is usually the most complex task. Decoding algorithms for LDPC/MDPC codes are mainly divided into two families. The first class (e.g., [BMvT78b]) offers a better error-correction capability but is computationally more complex than the second family. Especially when handling large codes, the
second family, called bit-flipping algorithms [Gal62], seems to be more appropriate. In general, they are all based on the following principle:

1. Compute the syndrome $s$ of the received codeword $x$.
2. Check the number of unsatisfied parity-check-equations $\#_{\text{upc}}$ associated with each codeword bit.
3. Flip each codeword bit that violates more than $b$ equations.

This process is iterated until either the syndrome becomes zero or a predefined maximum number of iterations is reached. In that case a decoding error is returned. The main difference of the bit-flipping algorithms is how the threshold $b$ is computed. In the original algorithm of Gallager [Gal62], a new $b$ is computed at each iteration. In [HP03], $b$ is taken as the maximum of the unsatisfied parity-check-equations $\text{Max}_{\text{upc}}$ and the authors of the QC-MDPC scheme propose to use $b = \text{Max}_{\text{upc}} - \delta$, for some small $\delta$.

Since estimating the error-correction capability of LDPC and MDPC codes generally is a hard task and is also influenced by the choice of threshold $b$, we derive different versions of the bit-flipping algorithm, evaluate their error-correcting capability and count how many iterations are required on average to decode a codeword. Because we are targeting embedded systems, we omit the variant storing $n_0$ counters for $\#_{\text{upc}}$ for each ciphertext bit. This would allow to skip the second computation of $\#_{\text{upc}}$ in some variants, but would blow up memory consumption to an unacceptable amount. We now introduce the different decoders under investigation:

**Decoder $A$** is given in [MTSB12] and computes the syndrome, then checks the number of unsatisfied parity-check-equations once to compute the maximum $\text{Max}_{\text{upc}}$ and afterwards a second time to flip all codeword bits that violate $b \geq \text{Max}_{\text{upc}} - \delta$ equations. Afterwards the syndrome is recomputed and compared to zero.

**Decoder $B$** is given in [Gal62] and computes the syndrome, then checks the number of unsatisfied parity-check-equations once per iteration $i$ and directly flips the current codeword bit if $\#_{\text{upc}}$ is larger than a precomputed threshold $b_i$. Afterwards the syndrome is recomputed and compared to zero.

We noticed that the previously proposed bit-flipping decoders recompute the syndrome after every iteration. Since this is quite costly we propose an optimization based on the following observation: If the amount of unsatisfied parity-check-equations exceeds threshold $b$, the corresponding bit in the codeword is flipped and the syndrome changes. We would like to stress that the syndrome does not change arbitrarily, but the new syndrome is equal to the old syndrome accumulated with the row $h_j$ of the parity check matrix that corresponds to the flipped codeword bit $j$. By keeping track of which codeword bits are flipped and updating the syndrome accordingly, the syndrome recomputation can be omitted. Hence, we propose and evaluate the following decoders:
Decoder $C_1$ computes the syndrome, then checks the number of unsatisfied parity-check-equations once to compute the maximum $Max_{upc}$ and afterwards a second time to flip all codeword bits that violate $b \geq Max_{upc} - \delta$ equations. If a codeword bit $j$ is flipped, the corresponding row $h_j$ of the parity check matrix is added to a temporary syndrome. At the end of each iteration the temporary syndrome is added to the syndrome, directly resulting in the syndrome of the new codeword without requiring a full recomputation.

Decoder $C_2$ computes the syndrome, then checks the number of unsatisfied parity-check-equations once to compute the maximum $Max_{upc}$ and afterwards a second time to flip all codeword bits that violate $b \geq Max_{upc} - \delta$ equations. If a codeword bit $j$ is flipped, the corresponding row $h_j$ of the parity check matrix is added directly to the current syndrome. Using this method we always work with an up-to-date syndrome and not with the one from the last iteration.

Decoder $D$ is similar to Decoder $B$ with precomputed thresholds $b_i$, but uses the direct update of the syndrome as done in Decoder $C_2$.

Decoder $E$ is similar to Decoder $C_2$ but compares the syndrome to zero after each flipped bit and aborts the current bit-flipping iteration immediately if the syndrome becomes zero.

Decoder $F$ is similar to Decoder $D$ and in addition uses the same early exit trick as Decoder $E$. 
Table 12.2: Evaluation of the performance and error correcting capability of the different decoders for a QC-MDPC code with parameters $n_0 = 2, n = 9600, r = 4800, w = 90$.

<table>
<thead>
<tr>
<th>Variant</th>
<th>#errors</th>
<th>time in $\mu$s</th>
<th>failure rate</th>
<th>avg. #iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoder $A$</td>
<td>84</td>
<td>26.8</td>
<td>0.00041</td>
<td>5.2964</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>27.3</td>
<td>0.00089</td>
<td>5.3857</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>27.9</td>
<td>0.00221</td>
<td>5.4975</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>28.7</td>
<td>0.00434</td>
<td>5.6261</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>29.3</td>
<td>0.00891</td>
<td>5.7679</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>30.1</td>
<td>0.01802</td>
<td>5.9134</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>31.0</td>
<td>0.03264</td>
<td>6.0677</td>
</tr>
<tr>
<td>Decoder $B$</td>
<td>84</td>
<td>12.6</td>
<td>0.00051</td>
<td>3.1425</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>12.9</td>
<td>0.00163</td>
<td>3.1460</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>13.4</td>
<td>0.00631</td>
<td>3.1607</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>13.9</td>
<td>0.01952</td>
<td>3.2022</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>14.6</td>
<td>0.05195</td>
<td>3.4040</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>15.1</td>
<td>0.11462</td>
<td>3.5009</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>15.7</td>
<td>0.24806</td>
<td>3.8972</td>
</tr>
<tr>
<td>Decoder $C_1$</td>
<td>84</td>
<td>22.7</td>
<td>0.00044</td>
<td>5.2862</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>23.2</td>
<td>0.00106</td>
<td>5.3924</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>23.7</td>
<td>0.00172</td>
<td>5.4924</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>24.2</td>
<td>0.00480</td>
<td>5.6260</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>25.1</td>
<td>0.00928</td>
<td>5.7595</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>25.6</td>
<td>0.01762</td>
<td>5.9078</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>26.4</td>
<td>0.03315</td>
<td>6.0685</td>
</tr>
<tr>
<td>Decoder $C_2$</td>
<td>84</td>
<td>14.0</td>
<td>0.00018</td>
<td>3.3791</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>14.1</td>
<td>0.00068</td>
<td>3.4180</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>14.2</td>
<td>0.00148</td>
<td>3.4643</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>14.6</td>
<td>0.00378</td>
<td>3.5279</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>14.8</td>
<td>0.00750</td>
<td>3.5942</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>15.1</td>
<td>0.01500</td>
<td>3.6542</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>15.4</td>
<td>0.02877</td>
<td>3.7435</td>
</tr>
<tr>
<td>Decoder $D$</td>
<td>84</td>
<td>7.02</td>
<td>0.00001</td>
<td>2.4002</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>7.04</td>
<td>0.00003</td>
<td>2.4980</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>7.24</td>
<td>0.00004</td>
<td>2.5979</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>7.53</td>
<td>0.00031</td>
<td>2.6958</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>7.78</td>
<td>0.00083</td>
<td>2.7875</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>8.13</td>
<td>0.00234</td>
<td>2.8749</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>8.31</td>
<td>0.00552</td>
<td>2.9670</td>
</tr>
<tr>
<td>Decoder $E$</td>
<td>84</td>
<td>14.15</td>
<td>0.00019</td>
<td>3.3754</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>14.14</td>
<td>0.00073</td>
<td>3.4218</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>14.77</td>
<td>0.00153</td>
<td>3.4673</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>14.63</td>
<td>0.00375</td>
<td>3.5314</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>15.11</td>
<td>0.00728</td>
<td>3.5886</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>15.15</td>
<td>0.01529</td>
<td>3.6563</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>15.68</td>
<td>0.02840</td>
<td>3.7343</td>
</tr>
<tr>
<td>Decoder $F$</td>
<td>84</td>
<td>6.68</td>
<td>0.00000*</td>
<td>2.4047</td>
</tr>
<tr>
<td></td>
<td>85</td>
<td>6.92</td>
<td>0.00002</td>
<td>2.5000</td>
</tr>
<tr>
<td></td>
<td>86</td>
<td>7.11</td>
<td>0.00008</td>
<td>2.5983</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>7.59</td>
<td>0.00039</td>
<td>2.6939</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td>7.68</td>
<td>0.00094</td>
<td>2.7912</td>
</tr>
<tr>
<td></td>
<td>89</td>
<td>7.99</td>
<td>0.00209</td>
<td>2.8793</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>8.54</td>
<td>0.00506</td>
<td>2.9630</td>
</tr>
</tbody>
</table>

* Note, this does not mean that Decoder $F$ always succeeds. It is still a probabilistic decoder that simply did not encounter any decoding failure in our evaluations.
Chapter 12. Codebased Crypto Using Quasi Cyclic Medium Density Parity Check Codes

The average number of iterations required to decode a codeword and the decoding failure rate for the different decoders with different numbers of errors are shown in Table 12.2 for a QC-MDPC code with parameters $n_0 = 2, n = 9600, r = 4800, w = 90$ (cf. first row of Table 12.1). All measurements are taken for 1000 random codes and 100,000 random decoding tries per code on a Intel Xeon E5345 CPU running at 2.33 GHz. For versions with precomputed thresholds $b_i$ we used the formula given in Appendix A of [MTSB12] to precompute the most suitable $b_i$’s for every iteration. For versions using $b = Max_{upc} - \delta$, we found by exhaustive experiments that the smallest number of iterations are required for $\delta = 5$. A decoding failure is returned when the decoder did not succeed within ten iterations.

The timings given in Table 12.2 should only be used to compare the decoders among each other. The evaluation was done in software and is not optimized for speed. It is designed to keep only the generating polynomial $h$ and not the whole parity check matrix $H$ in memory which would allow for a time/memory trade-off and faster computations. The corresponding row is derived at runtime by rotating the polynomial.

Our evaluations clearly show the superior error correcting capability of decoders $D$ and $F$ which in addition require the lowest number of iterations when compared to the other decoders (cf. Table 12.2). Decoders $A$ and $C_1$ are least efficient with an average of more than 5 bit-flipping iterations. Our new decoders $D$ and $F$ on average save 2.9 iterations compared to decoder $A$ and 0.7 iterations compared to $B$. This directly relates to the required time for decoding which is up to 4 times faster.

The small timing advantage of decoder $F$ over $D$ is due to the immediate termination if the syndrome becomes zero. Another interesting observation we made for all decoders is that if a codeword is decodable, then this is achieved after a small number of iterations. We noticed that if a codeword is not decoded within 4-6 iterations, a higher number of iterations does not lead to a successful decoding. Therefore, a early detection of a decoding failure is possible.

12.4 Implementation on FPGA and Microcontroller

In this section we discuss decoder and parameter selections and reason design choices for our QC-MDPC McEliece implementations on reconfigurable hardware and microcontrollers. The primary goal for the hardware design is high-performance while the microcontroller implementation aims for a low memory footprint. Note, the implementations of a CCA2-secure conversion and true random number generation are out of the scope of this work.

12.4.1 Decoder and Parameter Selection

Our implementations aim for a security level of 80 bit, comparable to ECC-160 and RSA-1024. Hence, we select the following QC-MDPC code parameters that provide a 80-bit security level according to Table 12.1:

$$n_0 = 2, n = 9600, r = 4800, w = 90, t = 84$$

In the latest version of [MTSB12] the authors also suggest to use $\delta \approx 5$ for the given parameters.
12.4. Implementation on FPGA and Microcontroller

Using these parameters we have a 4800-bit public key and a 9600-bit sparse secret key with 90 set bits. Such key sizes are only a fraction of the key sizes of other code-based public-key encryption schemes. During encryption a 4800-bit plaintext is encoded into a 9600-bit codeword and 84 errors are added to it. It follows from $n_0 = 2$ that the 9600-bit codeword and secret key consist of two separate 4800-bit codewords/secret keys, respectively.

As shown in Section 6.10 our decoders $D$ and $F$ require only one syndrome computation in the beginning and update the syndrome directly in the bit-flipping step. Furthermore, due to the precomputed thresholds $b_i$ the computation of the maximum number of unsatisfied parity check equations can be omitted. The decoders only differ in the way they handle the part where they check if the syndrome is zero. While decoder $F$ checks the syndrome every time the syndrome is changed in the bit-flipping step, decoder $D$ tests the syndrome at the end of each bit-flipping iteration. Note, the decoding behavior of both decoders is the same, i.e., they require the same amount of bit-flipping iterations with the difference that decoder $F$ exits as soon as the syndrome is equal to zero.

We base our QC-MDPC McEliece decryption implementation on decoder $D$ in hardware and on decoder $F$ for the microcontroller. The reason for choosing decoder $D$ to be implemented in hardware is that we sequentially rotate the codewords and secret keys in every cycle of the bit-flipping iterations. If the syndrome becomes zero during a bit-flipping iteration and we skip further computations immediately, the secret polynomials and the codewords would be misaligned. To fix this we would have to rotate them manually into their correct position which would take roughly the same amount of time as just letting the decoder finish the current iteration.

Both implementations use a maximum of five iterations before returning a decoding error and the corresponding precomputed $b_i$ are $(28, 26, 24, 22, 20)$, which are computed using the formula in the appendix of [MTSB12].

12.4.2 FPGA Implementation

For our evaluation of QC-MDPC in reconfigurable hardware we use Xilinx’s Virtex-6 FPGA device family as target platform. Virtex-6 devices are powerful FPGAs offering thousands of slices, where each slice contains four 6-input lookup tables (LUT), eight flip-flops (FF), and surrounding logic. In addition, embedded resources such as block memories (BRAM) and digital signal processors (DSP) are available. In the following we reason our design choices and describe the implementations of the QC-MDPC-based McEliece en- and decryption.

Design Considerations

Because of their relatively small size, the public and secret key do not have to be stored in external memory as it was necessary in earlier FPGA implementations of McEliece and Niederreiter using, e.g., Goppa codes. Since we aim for high-speed, we store all operands directly in FPGA logic and refrain from loading/storing them from/to internal block memories or other external memory as this would affect performance. Reading a single 4800-bit vector from a
32-bit BRAM interface would consume 150 clock cycles. However, if maximum performance is not required, the use of BRAMs could certainly reduce resource consumption significantly.

In contrast to the microcontroller implementation we do not exploit the sparsity of the secret polynomials in our FPGA design. Using a sparse representation of the secret polynomials would require to implement $w = 90$ counters with 13 bits, each indicating the position of a set bit in one of the two secret polynomials. To generate the next row of the secret key, all counters have to be increased and in case of exceeding 4799 they have to be set to 0. If a bit in the codewords $x_0$ or $x_1$ is set we have to build a 4800-bit vector from the counters belonging to the corresponding secret polynomial and XOR this vector to the current syndrome. The alternative is to read out the content of each counter belonging to the corresponding secret polynomial and flip the corresponding bit in the syndrome. These tasks, however, are time and/or resource consuming in hardware.

**Implementation**

We use a Virtex-6 XC6VLX240T FPGA as target device for a fair comparison with previous work – although all our implementations would fit smaller devices as well.

The encryption and decryption unit are equipped with a simple I/O interface. Messages and codewords are send and received bit by bit to keep the I/O overhead of our implementation small and thus get as close as possible to the actual resource consumptions of the en-/decoder.

**QC-MDPC Encryption:** In order to implement a QC-MDPC encoder we need a vector matrix multiplication to multiply message $m$ with the public key matrix $G$ to retrieve a codeword $c = mG$ and then add an error vector with $hw(e) \leq 84$ to get the ciphertext $x = c + e$. We are given a 4800-bit public key $g$ which is the first row of matrix $G$. Rotating $g$ by one bit position yields the next row of $G$ and so forth. Since $G$ is of systematic form the first half of $c$ is equal to $m$. The second half, called redundant part, is computed as follows.

We iterate over the message bit by bit and XOR the current public polynomial to the redundant part if the current message bit is set. To implement this in hardware we need three 4800-bit registers to hold the public polynomial, the message, and the redundant part. Since only one bit of the message has to be accessed in every clock cycle, we store the message in a circulant shift register which can be implemented using shift register LUTs.

**QC-MDPC Decryption:** Decryption is performed by decoding the received ciphertext, the first half of the decoded codeword is the plaintext. As QC-MDPC decoder we implement the bit-flipping decoder $D$ as described in Section 12.3. In the first step we need to compute the syndrome $s = Hx^T$ by multiplying parity check matrix $H = [H_0|H_1]$ with the ciphertext $x$.

Given the first 9600-bit row $h = [h_0|h_1]$ of $H$ and the 9600-bit codeword $x = [x_0|x_1]$ we compute the syndrome as follows. We sequentially iterate over every bit of the codewords $x_0$ and $x_1$ in parallel and rotate $h$ by rotating $h_0$ and $h_1$ accordingly. If a bit in $x_0$ and/or $x_1$ is set, we XOR the current $h_0$ and/or $h_1$ to the intermediate syndrome which is set to zero in the beginning. The syndrome computation is finished after every bit of the ciphertext has been processed.
Next we need to check if the syndrome is zero. We implement this as a logical OR tree. Since the FPGA offers 6-input LUTs, we split the syndrome into 6-bit chunks and compute their logical OR on the lowest level of the tree. The results are fed into the next level of 6-bit LUTs which again compute the logical OR of the inputs. This is repeated until we are left with a single bit that indicates if the syndrome is zero or not. In addition, we add registers after the second layer of the tree to minimize the critical path.

If the syndrome is zero, the decryption is finished. Otherwise we have to compute the number of unsatisfied parity check equations for each row \( h = [h_0 | h_1] \). We therefore compute the hamming weight of the logical AND of the syndrome and \( h_0 \) and \( h_1 \), respectively. If the hamming weight exceeds the threshold \( b_i \) for the current iteration \( i \), the corresponding bit in the codeword \( x_0 \) and/or \( x_1 \) is flipped and the syndrome is directly updated by XORing the current secret polynomial \( h_0 \) and/or \( h_1 \) to it. Then \( h_0 \) and \( h_1 \) are rotated by one bit and the process is repeated until all rows of \( H \) have been checked.

Since the computation of the number of unsatisfied parity check equations for \( h_0 \) and \( h_1 \) can be performed independently, we have two options for implementation. Either we compute the parity check violations of the first and second secret polynomial iteratively or we instantiate two hamming weight computation units and process the polynomials in parallel. The iterative version will take twice the time but using less resources. We explore both version to evaluate this time/resource trade-off.

Computing the hamming weight of a 4800-bit vector efficiently is a challenge of its own. Similar to the zero comparator we split the input into 6-bit chunks and determine their hamming weight. We then compute the overall hamming weight by building an adder tree with registers on every layer to minimize the critical path. After all rows of \( H \) have been processed, the syndrome is again compared to zero. If the syndrome is zero, the first 4800-bit of the updated codeword (i.e., \( x_0 \)) are equal to the decoded message \( m \) and are returned. Otherwise the bit-flipping is repeated with the next \( b_i \) until either the syndrome becomes zero or the maximum number of iterations is exceeded.

### 12.4.3 Microcontroller Implementation

As implementation platform we choose a ATxmega256A3 microcontroller for straightforward comparison with previous work. The microcontroller provides 16 kByte SRAM and 256 kByte program memory and can be clocked at up to 32 MHz. The main parts are written in C and we pay careful attention to implement timing critical routines as, e.g., the polynomial rotation and addition using inline assembly.

The encoding operation is straightforward. Since \( G \) is of systematic form, the first \( r \) ciphertext bits are the message itself and are simply copied. For the multiplication with the redundant part \( Q \), the message bits are parsed and the corresponding rows of \( G \) are summed up. Afterwards the current row is rotated by one bit-position to generate the next row. We implemented two different version of the encoder which differ in the way the public polynomial rotation is
implemented. In one version we use a loop to rotate the byte of the public polynomial and in the other version we unroll this process.

Usually, smartcard devices communicate over a very slow interface, e.g., 106 kByte/s \([\text{Str12}]\). In contrast to cryptosystems such as RSA and ECC, we do not need the message as a whole to start with the encryption. Therefore, an interesting option is to directly encode a byte of the message as soon as it arrives while the next message byte is still in transfer. To some extend, this allows to hide the computation time within the latency required to transfer the message.

For decoding, recall that the \(n_0 = 2\) involved secret polynomials are sparse and only 45 out of 4800 bits are set. Instead of saving 4800 coefficients in \(\frac{4800}{8} = 600\) bytes, it is sufficient to save the indices of the \(w_i = 45\) bits that are set. Each secret polynomial therefore requires only \([\log_2(4800)/8]\) \cdot 45 = 2 \cdot 45 = 90\) bytes. Additionally, rotating a polynomial by one bit-position means incrementing the 45 indices by one and handling the overflow from \(x^{4800}\) to \(x^0\). We developed a vector-(sparse-matrix) multiplication, which adds a sparse row to the syndrome by flipping the 45 indexed bits in the 4800 bit syndrome. Also the update of the syndrome can be handled this way when a ciphertext bit is flipped. In order to keep the memory consumption low while still achieving good performance we use decoder \(F\), as described in Section 6.10. Since we store the bit-position in counters, an early exit of the decoding phase can be implemented – unlike to our hardware implementation. The complete secret key therefore requires only \(2 \cdot (2 \cdot 45)\) bytes for the secret polynomials and additionally ten bytes for the precomputed thresholds \(b_i\).

Note that the precomputed thresholds \(b_i\) can be treated as public system parameter. In contrast to the encoding process, every ciphertext byte is accessed multiple times during decoding so that the ”process-while-transfer”-method described above is not applicable. Also note that during decoding no additional memory is required to store the plaintext as the first half of the ciphertext is equal to the plaintext after successful decoding.

12.5 Results

In the following we present our QC-MDPC implementation results in reconfigurable hardware and in software on a 8-bit microcontroller. Afterwards we give an overview of existing public-key encryption implementations for similar platforms and compare them to our results.

12.5.1 FPGA Results

All our results are obtained post place-and-route (PAR) for a Xilinx Virtex-6 XC6VLX240T FPGA using Xilinx ISE 14.5. For the throughput figures we assume a fast enough I/O interface is provided.

In hardware, our QC-MDPC encoder runs at 351.3 MHz and encodes a 4800-bit message in 4800 clock cycles which results in 351.3 Mbit/s. The iterative version of our QC-MDPC decoder runs at 222.5 MHz. Since the decoder does not run in constant time, we calculate the average required cycles for iterative decoding as follows. Computing the syndrome for the first time needs 4800 clock cycles and comparing the syndrome to zero takes another 2 clock cycles. For
every following bit-flipping iteration we need 9620 plus again 2 clock cycles for checking the syndrome. As shown in Table 12.2, decoder D needs 2.4002 bit-flipping iterations on average. Thus, the average cycle count for our iterative decoder is $4800 + 2 + 2.4002 \cdot (9620 + 2) = 27896.7$ clock cycles.

Our non-iterative decoder processes both secret polynomials in the bit-flipping step in parallel and runs at 190.6 MHz. We calculate the average cycles as before with the difference that every bit-flipping iteration now takes 4810 + 2 clock cycles. Thus, the average cycle count for our non-iterative decoder is $4800 + 2 + 2.4002 \cdot (4810 + 2) = 16351.8$ clock cycles.

The non-iterative decoder operates 46% faster than the iterative version while occupying 40-65% more resources. Compared to the decoders, the encoder runs 6-9 times faster and occupies 2-6 times less resources. Table 12.3 summarizes our results.

Table 12.3: Implementation results of our QC-MDPC implementations with parameters $n_0 = 2, n = 9600, r = 4800, w = 90, t = 84$ on a Xilinx Virtex-6 XC6VLX240T FPGA.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Encoder</th>
<th>Decoder (iterative)</th>
<th>Decoder (non-iterative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFs</td>
<td>14,426</td>
<td>32,974 (10%)</td>
<td>46,515 (15%)</td>
</tr>
<tr>
<td>LUTs</td>
<td>8,856</td>
<td>36,554 (24%)</td>
<td>46,249 (30%)</td>
</tr>
<tr>
<td>Slices</td>
<td>2,920</td>
<td>10,271 (27%)</td>
<td>17,120 (45%)</td>
</tr>
<tr>
<td>Frequency</td>
<td>351.3 MHz</td>
<td>222.5 MHz</td>
<td>190.6 MHz</td>
</tr>
<tr>
<td>Time/Op</td>
<td>13.66 μs</td>
<td>125.38 μs</td>
<td>85.79 μs</td>
</tr>
<tr>
<td>Throughput</td>
<td>351.3 Mbit/s</td>
<td>38.3 Mbit/s</td>
<td>55.9 Mbit/s</td>
</tr>
<tr>
<td>Encode</td>
<td>4,800 cycles</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Compute Syndrome</td>
<td>-</td>
<td>4,800 cycles</td>
<td>4,800 cycles</td>
</tr>
<tr>
<td>Check Zero</td>
<td>-</td>
<td>2 cycles</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Flip Bits</td>
<td>-</td>
<td>9,620 cycles</td>
<td>4,810 cycles</td>
</tr>
<tr>
<td>Overall average</td>
<td>4,800 cycles</td>
<td>27,896.7 cycles</td>
<td>16,351.8 cycles</td>
</tr>
</tbody>
</table>

Using the formerly proposed decoders that work without our syndrome computation optimizations (i.e., decoders A and B) would result in much slower decryptions. Decoder A would need $4802 + 5.2964 \cdot (2 \cdot 9620 + 4802) = 132138.0$ cycles in an iterative and $4802 + 5.2964 \cdot (2 \cdot 4810 + 4802) = 81186.7$ cycles in a non-iterative implementation. Decoder B saves cycles by skipping the $Max_{upc}$ computation but would still need $4802 + 3.1425 \cdot (9620 + 4802) = 50123.1$ cycles in an iterative and $4802 + 3.1425 \cdot (4810 + 4802) = 35007.7$ cycles in a non-iterative implementation.

**Comparison**

A comparison with previously published FPGA implementations of code-based (McEliece, Niederreiter), lattice-based (Ring-LWE, NTRU), and standard public-key encryption schemes (RSA, ECC) is given in Table 12.4. The most relevant metric for comparing the performance of public-key encryption schemes often depends on the application. For key exchange it is the required time per operation, given the symmetric key size is smaller or equal to the bit size that
can be transmitted in one operation. For data encryption (i.e., much more than one block), throughput in Mbit/s is typically the most interesting metric.

A hardware McEliece implementation based on Goppa codes including CCA2 conversion was presented for a Virtex5-LX110T FPGA in [SWM+10, SWM+09]. Comparing their performance to our implementations shows the advantage of QC-MDPC McEliece in both time per operation and Mbit/s. The occupied resources are similar to our resource requirements but in addition 75 block memories are required for storage. Even more important for real-world applications is the public key size. QC-MDPC McEliece requires 0.59 kByte which is only a fraction of the 100.5 kByte public key of [SWM+10].

A McEliece co-processor was recently proposed for a Virtex5-LX110T FPGA [GDUV12]. Their design goal was to optimize the speed/area ratio while we aim for high performance. With respect to decoding performance, our implementations outperform their work in both time/operation and Mbit/s. But the co-processor needs much less resources and can also be implemented on low-cost devices such as Spartan-3 FPGAs. The public keys in this work have a size of 63.5 kByte which is still much larger than the 0.59 kByte of QC-MDPC McEliece.

The Niederreiter public-key scheme was implemented in [HG12] for a Virtex6-LX240T FPGA. The work shows that Niederreiter encryption can provide high performance with a moderate amount of resources. Decryption is more expensive both in computation time as well as in required resources. The Niederreiter encryption is the superior choice for a minimum time per operation, but concerning raw throughput QC-MDPC achieves better results. Furthermore, the public key with 63.5 kByte of the Niederreiter encryption using binary Goppa codes might be too large for real-world applications.

FPGA implementations of lattice-based public-key encryption were proposed in [GFS+12] for Ring-LWE and in [KY09] for NTRU. The Ring-LWE implementation requires a huge amount of resources (in particular, exceeding the resources provided by their Virtex6-LX240T FPGA). On the other hand, NTRU as implemented in [KY09] shows that lattice-based cryptography can provide high performance at moderate resources requirements. Note further that the results are reported for an outdated Virtex-E FPGA which is hardly comparable to modern Virtex-5/-6 devices.

Efficient ECC hardware implementations for curves over $GF(p)$ and $GF(2^m)$ are [DJJ+06, GP08, RRM12a, RRM12b] which all yield good performance at moderate resource requirements. The most efficient RSA hardware implementation to date was proposed in [SM11, Suz07]. Both the time to encrypt and decrypt one block as well as the throughput are considerably worse than QC-MDPC McEliece.

12.5.2 Microcontroller Results

Our QC-MDPC encryption requires 606 byte SRAM and 3,705 byte flash memory for the iterative design and 606 byte SRAM and 5,496 byte flash memory in the unrolled version. Both versions already include the public key. The decryption unit requires 198 byte SRAM and 2,218 byte flash memory including the secret key, which is copied to SRAM at start-up for
faster access. The encoder requires 26,767,463 cycles on average or 0.8 seconds at 32 MHz. Most cycles are consumed when adding a row of $G$ to the ciphertext ($\sim 6000$ cycles each) and when rotating a row to generate the next one ($\sim 2400$ cycles).

The decoder requires 86,874,388 cycles on average or 2.7 seconds at 32 MHz. Rotating a polynomial in sparse representation takes 720 cycles and adding a sparse polynomial to the syndrome requires 2,285 cycles which clearly shows the advantage of a sparse representation. Nevertheless, computing a syndrome using the vector-(sparse-matrix)-multiplication on average requires 10,379,351 cycles. Because syndrome, ciphertext and the current row of $H$ (even in sparse form) are too large to be held in registers, they have to be stored in SRAM and are continuously loaded and stored.

**Comparison**

Table 12.5 compares our results with other implementations of McEliece and with implementations of the classical cryptosystems RSA and ECC on a similar microcontroller. For the code-based schemes, the flash memory usage includes the public and secret key, respectively. For RSA and ECC, [GPW+04] does not clearly state if the key size is included.

The main advantage of our implementations compared to other code-based schemes is the small memory footprint. Especially our decoder requires much less memory than other McEliece decoders because we only need to store the bit positions of the sparse secret polynomials instead of the full secret key.

We use the cycles/byte metric to compare our results to other implementations that handle different plaintext/ciphertext sizes. Our iterative encoder outperforms the encoders of [CHP12] and [EGHP09]. Our unrolled version is nearly as fast as [Hey11] with only half the amount of flash memory and six times less SRAM. Solely the quasi-dyadic McEliece implementation of [Hey11] outperforms our implementation, however requires much more SRAM and flash memory.
<table>
<thead>
<tr>
<th>Scheme</th>
<th>Platform</th>
<th>f [MHz]</th>
<th>Bits</th>
<th>Time/Op</th>
<th>Cycles</th>
<th>Mbit/s</th>
<th>FFs</th>
<th>LUTs</th>
<th>Slices</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work (enc)</td>
<td>XC6VLX240T</td>
<td>351.3</td>
<td>4,800</td>
<td>13.66 µs</td>
<td>4,800</td>
<td>351.3</td>
<td>14,426</td>
<td>8,856</td>
<td>2,920</td>
<td>0</td>
</tr>
<tr>
<td>This work (dec)</td>
<td>XC6VLX240T</td>
<td>190.6</td>
<td>4,800</td>
<td>85.79 µs</td>
<td>16,352</td>
<td>55.9</td>
<td>46,515</td>
<td>46,249</td>
<td>17,120</td>
<td>0</td>
</tr>
<tr>
<td>This work (dec iter.)</td>
<td>XC6VLX240T</td>
<td>222.5</td>
<td>4,800</td>
<td>125.38 µs</td>
<td>27,897</td>
<td>38.3</td>
<td>32,974</td>
<td>36,554</td>
<td>10,271</td>
<td>0</td>
</tr>
<tr>
<td>McEliece (enc) [SWM+10]</td>
<td>XC5VLX110T</td>
<td>163</td>
<td>512</td>
<td>500 µs</td>
<td>n/a</td>
<td>1.0</td>
<td>n/a</td>
<td>n/a</td>
<td>14,537</td>
<td>75²</td>
</tr>
<tr>
<td>McEliece (dec) [SWM+10]</td>
<td>XC5VLX110T</td>
<td>163</td>
<td>512</td>
<td>1,290 µs</td>
<td>n/a</td>
<td>0.4</td>
<td>n/a</td>
<td>n/a</td>
<td>14,537</td>
<td>75²</td>
</tr>
<tr>
<td>McEliece (dec) [GDUV12]</td>
<td>XC5VLX110T</td>
<td>190</td>
<td>1,751</td>
<td>500 µs</td>
<td>94,249</td>
<td>3.5</td>
<td>n/a</td>
<td>n/a</td>
<td>1,385</td>
<td>5</td>
</tr>
<tr>
<td>Niederreiter (enc) [HG12]</td>
<td>XC6VLX240T</td>
<td>300</td>
<td>192</td>
<td>0.66 µs</td>
<td>200</td>
<td>290.9</td>
<td>875</td>
<td>926</td>
<td>315</td>
<td>17</td>
</tr>
<tr>
<td>Niederreiter (dec) [HG12]</td>
<td>XC6VLX240T</td>
<td>250</td>
<td>192</td>
<td>58.78 µs</td>
<td>14,500</td>
<td>3.3</td>
<td>12,861</td>
<td>9,409</td>
<td>3,887</td>
<td>9</td>
</tr>
<tr>
<td>Ring-LWE (enc) [GFS+12]</td>
<td>XC6VLX240T</td>
<td>n/a</td>
<td>256</td>
<td>8.10 µs</td>
<td>n/a</td>
<td>15.8</td>
<td>143,396</td>
<td>298,016</td>
<td>n/a</td>
<td>0²</td>
</tr>
<tr>
<td>Ring-LWE (dec) [GFS+12]</td>
<td>XC6VLX240T</td>
<td>n/a</td>
<td>256</td>
<td>8.15 µs</td>
<td>n/a</td>
<td>15.7</td>
<td>65,174</td>
<td>124,158</td>
<td>n/a</td>
<td>0²</td>
</tr>
<tr>
<td>NTRU (enc/dec) [KY09]</td>
<td>XC1600E</td>
<td>62.3</td>
<td>251</td>
<td>1.54/1.41 µs</td>
<td>96/88</td>
<td>163/178</td>
<td>5,160</td>
<td>27,292</td>
<td>14,352</td>
<td>0</td>
</tr>
<tr>
<td>ECC-P224 [GP08]</td>
<td>XC4VFX12</td>
<td>487</td>
<td>224</td>
<td>365.10 µs</td>
<td>177,755</td>
<td>0.61</td>
<td>1,892</td>
<td>1,825</td>
<td>1,580</td>
<td>11³</td>
</tr>
<tr>
<td>ECC-163 [RRM12a]</td>
<td>XC5VLX85T</td>
<td>167</td>
<td>163</td>
<td>8.60 µs</td>
<td>1436</td>
<td>18.9</td>
<td>n/a</td>
<td>10,176</td>
<td>3,446</td>
<td>0</td>
</tr>
<tr>
<td>ECC-163 [RRM12b]</td>
<td>Virtex-4</td>
<td>45.5</td>
<td>163</td>
<td>12.10 µs</td>
<td>552</td>
<td>13.4</td>
<td>n/a</td>
<td>n/a</td>
<td>12,430</td>
<td>0</td>
</tr>
<tr>
<td>ECC-163 [DJJ+06]</td>
<td>Virtex-II</td>
<td>128</td>
<td>163</td>
<td>35.75 µs</td>
<td>4576</td>
<td>4.56</td>
<td>n/a</td>
<td>n/a</td>
<td>2254</td>
<td>6</td>
</tr>
<tr>
<td>RSA-1024 [SM11]</td>
<td>XC5VLX30T</td>
<td>450</td>
<td>1,024</td>
<td>1,520 µs</td>
<td>684,000</td>
<td>0.67</td>
<td>n/a</td>
<td>n/a</td>
<td>3,237</td>
<td>5⁴</td>
</tr>
</tbody>
</table>

Table 12.4: Performance comparison of our QC-MDPC FPGA implementations with other public-key encryption schemes.

1Occupied slices and BRAMs are only given for encryption and decryption combined. ²Calculated from synthesis results of an over-mapped device, post-PAR results are not given and will most likely be much slower. ³Additionally uses 26 DSP48s. ⁴Additionally uses 17 DSP48s.
Table 12.5: Performance comparison of our QC-MDPC microcontroller implementations with other public-key encryption schemes.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Platform</th>
<th>SRAM</th>
<th>Flash</th>
<th>Cycles/Op</th>
<th>Cycles/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work [enc]</td>
<td>ATxmega256</td>
<td>606 Byte</td>
<td>3,705 Byte</td>
<td>37,440,137</td>
<td>62,400</td>
</tr>
<tr>
<td>This work [enc unrolled]</td>
<td>ATxmega256</td>
<td>606 Byte</td>
<td>5,496 Byte</td>
<td>26,767,463</td>
<td>44,612</td>
</tr>
<tr>
<td>This work [dec]</td>
<td>ATxmega256</td>
<td>198 Byte</td>
<td>2,218 Byte</td>
<td>86,874,388</td>
<td>146,457</td>
</tr>
<tr>
<td>McEliece [enc] [EGHP09]</td>
<td>ATxmega256</td>
<td>512 Byte</td>
<td>438 kByte</td>
<td>14,406,080</td>
<td>65,781</td>
</tr>
<tr>
<td>McEliece [dec] [EGHP09]</td>
<td>ATxmega256</td>
<td>12 kByte</td>
<td>130.4 kByte</td>
<td>19,751,094</td>
<td>90,187</td>
</tr>
<tr>
<td>McEliece [enc] [Hey11]</td>
<td>ATxmega256</td>
<td>3.5 kByte</td>
<td>11 kByte</td>
<td>6,358,400</td>
<td>39,493</td>
</tr>
<tr>
<td>McEliece [dec] [Hey11]</td>
<td>ATxmega256</td>
<td>8.6 kByte</td>
<td>156 kByte</td>
<td>33,536,000</td>
<td>208,298</td>
</tr>
<tr>
<td>McEliece [enc] [CHP12]</td>
<td>ATxmega256</td>
<td>-</td>
<td>-</td>
<td>4,171,734</td>
<td>260,733</td>
</tr>
<tr>
<td>McEliece [dec] [CHP12]</td>
<td>ATxmega256</td>
<td>-</td>
<td>-</td>
<td>14,497,587</td>
<td>906,099</td>
</tr>
<tr>
<td>ECC-P160 [GPW+04]</td>
<td>ATmega128</td>
<td>282 Byte</td>
<td>3682 Byte</td>
<td>6,480,000</td>
<td>324,000</td>
</tr>
<tr>
<td>RSA-1024 random [GPW+04]</td>
<td>ATmega128</td>
<td>930 Byte</td>
<td>6292 Byte</td>
<td>87,920,000</td>
<td>686,875</td>
</tr>
</tbody>
</table>
Part III

Other Alternative Public Key Schemes
Chapter 13

Multivariate Quadratics Public-Key Schemes

This research contribution is based on the author’s published research in [CHT12]. It is joint work with Peter Czypek and Enrico Thomae

Multivariate Quadratic Public-Key Schemes (MQPKS) attracted the attention of researchers in the last decades for two reasons. First they are thought to resist attacks by quantum computers and second, most of the schemes were broken. The latter may be the reason why implementations are rare. This chapter investigates one of the most promising member of MQPKS and its variants, namely UOV, Rainbow and enTTS. UOV resisted all kinds of attacks for 13 years and can be considered one of the best examined MQPKS. We describe implementations of UOV, Rainbow and enTTS on an 8-bit microcontroller. To address the problem of large keys, we used several optimizations and also implemented the 0/1-UOV scheme introduced at CHES 2011. To achieve a practically usable security level on the selected device, all recent attacks are summarized and parameters for standard security levels are given. To allow judgement of scaling, the schemes are implemented for the most common security levels in embedded systems $2^{64}$, $2^{80}$ and $2^{128}$ bits symmetric security. This allows for the first time a direct comparison of the four schemes because they are implemented for exactly the same security levels on the same platform and also by the same developer.

Section 13.2 introduces MQ-schemes in general and UOV, Rainbow and enTTS in special. Section 13.3 summaries recent attacks and derives parameter sets to achieve $2^{64}$, $2^{80}$ and $2^{128}$ bit security. Afterwards, Section 13.4 describes our implementations before we present our results in Section 13.5. Finally, we conclude in Section 13.6 and point out some details for future improvements.

13.1 Introduction

Since Peter Shor published efficient quantum algorithms [Sho97] to solve the problem of factorization and discrete logarithm in 1995, there is a increasing demand in investigating possible alternatives. One such class of so-called post-quantum cryptosystems is based on multivariate quadratic (MQ) polynomials. We know that solving systems of MQ-polynomials is hard in the
worst case, as the corresponding $MQ$-problem is proven to be $NP$-complete \cite{GJ79}. Unfortunately all schemes proposed so far also need the Isomorphism of Polynomials (IP) problem to hide the trapdoor. It is not known how hard this problem is and indeed most $MQ$-schemes are broken this way. So for example, the balanced Oil and Vinegar scheme \cite{KS98}, Sflash \cite{BFMR11} and much more \cite{Pat95, KS99, GC00, cFJ03, CD03, WBP04}. To encapsulate, nearly all $MQ$-encryption schemes and most of the $MQ$-signature schemes are broken up to this point. There are only very few exceptions like the signature schemes HFE$^{-}$, Unbalanced Oil and Vinegar (UOV) and its layer based variants Rainbow and enTTS. Well, breaking the first seems to be a matter of time as some ideas of the attack against Sflash from Asiacrypt 2011 \cite{BFMR11} might also be applicable. On the other hand, UOV resisted all kinds of attacks for 13 years. It is thought to be the most promising member of the class of $MQ$-schemes.

\section*{Previous Work and Contribution}

Rainbow type hardware implementations got some attention during the last years. An 0.35µm ASIC, which signs in 0.012 ms, is reported in \cite{BCB08}. Further \cite{TYD11} presents an ASIC implementation, taking only 198 clock cycles for a sign operation. An ASIC implementation of enTTS(20,28) enabling sign in 0.044 seconds running at a slow clock of 100KHz, is reported in \cite{YCCC06}. The authors also report a MSP430 implementation signing in 71 ms and verifying in 726 ms and a 8051-compatible µC implementation signing in 198ms. At CHES 2004, Yang et al. describe an implementation of TTS targeting 8051-compatible µCs \cite{YCC04}. Their implementation of TTS(20,28) signs in 144ms, 170ms, 60ms and for TTS(24,32) they achieve 191ms, 227 ms, 85 ms for an i8032AH, i8051AH and W77E59, respectively. We are not aware of any implementation of UOV or Rainbow targeting small microcontrollers. This work describes implementations of the $MQ$-signature schemes, UOV, Rainbow and enTTS, on an 8-bit microcontroller. Additionally, methods to reduce the key size are evaluated and a version of UOV published at CHES 2011 (0/1-UOV \cite{PTBW11}) is introduced and also evaluated. To achieve a practically usable security level on the selected device, recent attacks are summarized and parameters for standard security levels are given. The actual implementations were all done by the same developer. This ensures, that we really compare different schemes and not just different skills of different developers.

\section{Multivariate Quadratic Public-Key Cryptosystems}

This section provides a brief introduction to UOV \cite{KPG99}, 0/1 UOV \cite{PTBW11}, Rainbow \cite{DS05} and enTTS \cite{YC05}. The general idea of all these $MQ$-signature schemes is to use a public multivariate quadratic map $P : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$ with

\[ P = \begin{pmatrix} p^{(1)}(x_1, \ldots, x_n) \\ \vdots \\ p^{(m)}(x_1, \ldots, x_n) \end{pmatrix} \]
and
\[ p^{(k)}(x_1, \ldots, x_n) := \sum_{1 \leq i \leq j \leq n} \alpha_{ij}^{(k)} x_i x_j = x^T \mathcal{P}^{(k)} x, \]
where \( \mathcal{P}^{(k)} \) is the \((n \times n)\) matrix describing the quadratic form of \( p^{(k)} \) and \( x = (x_1, \ldots, x_n)^T \). Note that we can neglect linear and constant terms as they never mix with quadratic terms and thus do not increase the security [BWP05].

The trapdoor is given by a structured central map \( F : F_q^n \to F_q^m \) with
\[
F = \begin{pmatrix}
  f^{(1)}(u_1, \ldots, u_n) \\
  \vdots \\
  f^{(m)}(u_1, \ldots, u_n)
\end{pmatrix}
\]
and
\[ f^{(k)}(u_1, \ldots, u_n) := \sum_{1 \leq i \leq j \leq n} \gamma_{ij}^{(k)} u_i u_j = u^T \mathcal{F}^{(k)} u. \]

In order to hide this trapdoor we choose two secret linear transformations \( S, T \) and define \( \mathcal{P} := T \circ F \circ S \). See Figure 13.1 for an illustration.

Figure 13.1: MQ-Scheme in general.

**Unbalanced Oil and Vinegar**

For the UOV signature scheme the variables \( u_i, i \in V := \{1, \ldots, v\} \) are called *vinegar variables* and the remaining variables \( u_i, i \in O := \{v+1, \ldots, n\} \) are called *oil variables*. The central map \( F \) is given by
\[ f^{(k)}(u_1, \ldots, u_n) := \sum_{i \in V, j \in V} \gamma_{ij}^{(k)} u_i u_j + \sum_{i \in V, j \in O} \gamma_{ij}^{(k)} u_i u_j. \]

The corresponding matrix \( \mathcal{F}^{(k)} \) is depicted in Figure 13.2.

As we have \( m \) equations in \( m + v \) variables, fixing \( v \) variables will yield a solution with high probability. Due to the structure of \( \mathcal{F}^{(k)} \), i.e., there are no quadratic terms of two oil variables, we can fix the vinegar variables at random to obtain a system of linear equations in the oil variables, which is easy to solve. This procedure is not possible for the public key, as the transformation \( S \) of variables fully mixes the variables (like oil and vinegar in a salad). Note that for UOV we can discard the transformation \( T \) of equations, as the trapdoor is invariant under this linear transformation.
Rainbow.

Rainbow uses the same idea as UOV but in different layers. Current choices of parameters $(q, v_1, o_1, o_2)$ use two layers, as it turned out to be the best choice in order to prevent MinRank attacks and preserve short signatures at the same time. We will use $q = 2^8$ throughout the paper. The central map $\mathcal{F}$ of Rainbow is divided into two layers $\mathcal{F}^{(1)}, \ldots, \mathcal{F}^{(o_1)}$ and $\mathcal{F}^{(o_1+1)}, \ldots, \mathcal{F}^{(o_1+o_2)}$ of form given in Figure 13.3.

To use the trapdoor we first solve the small UOV system $\mathcal{F}^{(1)}, \ldots, \mathcal{F}^{(o_1)}$ by randomly fixing the $v_1$ vinegar variables. The solution $u_1, \ldots, u_{v_1+o_1}$ is now used as vinegar variables of the second layer. Solving the obtained linear system yields $u_{v_1+o_1+1}, \ldots, u_{v_1+o_1+o_2}$. A formal description of Rainbow is given by the following formula.

$$f^{(k)}(u_1, \ldots, u_n) := \sum_{i \in V_1 \cup O_1} \gamma_{ij}^{(k)} u_i u_j + \sum_{i \in V_1 \cup O_1} \gamma_{ij}^{(k)} u_i u_j$$

for $k = 1, \ldots, o_1$

$$f^{(k)}(u_1, \ldots, u_n) := \sum_{i \in V_1 \cup O_1 \cup O_2} \gamma_{ij}^{(k)} u_i u_j + \sum_{i \in V_1 \cup O_1 \cup O_2} \gamma_{ij}^{(k)} u_i u_j$$

for $k = o_1 + 1, \ldots, o_1 + o_2$
0/1-Unbalanced Oil and Vinegar

At CHES 2011 Petzold et al. [PTBW11] showed that large parts of the public key are redundant in order to prevent key recovery attacks. More precisely, $S$ can be chosen of a special structure due to equivalent keys and thus large parts of the public and secret map are equal. Choosing this parts of $P$ of a special structure, such that direct attacks on the public key do not become easier, they were able to reduce the key size and running time of the verification algorithm.

Enhanced TTS

Enhanced TTS was proposed by Yang and Chen in 2005 [YC05]. The general idea is the same as for Rainbow, but as TTS was designed for high speed implementation it uses as few monomials as possible. For the purpose of evaluating the security we generalize the scheme by adding more monomials. As soon as a monomial $x_ix_j$ with $x_i \in U$ and $x_j \in V$ occur in the original TTS polynomial, we just assume that all monomials $x_ix_j$ with $x_i \in U$ and $x_j \in V$ occur. This way we easily see that TTS is a very special case of the Rainbow signature scheme. There are two different scalable central maps given in [YC05], one is called even sequence and the other odd sequence. The following equations show the odd sequence. We restrict our implementation to this case.

$$
\begin{align*}
\hat{f}(i) &= u_i + \sum_{j=1}^{2\ell-3} \gamma_{ij} u_j u_{2\ell-2+(i+j+1 \mod 2\ell-1)} & \text{for } 2\ell-2 \leq i \leq 4\ell-4, \\
\hat{f}(i) &= u_i + \sum_{j=1}^{\ell-2} \gamma_{ij} u_{i+j-(4\ell-3)} u_{i-j-2\ell} + \sum_{j=\ell-1}^{2\ell-3} \gamma_{ij} u_{i+j-3\ell+3} u_{i-j+\ell-2} & \text{for } i = 4\ell-3, 4\ell-2, \\
\hat{f}(i) &= u_i + \gamma_{i0} u_{i-2\ell+1} u_{i-2\ell-1} + \sum_{j=4\ell-1}^{i-1} \gamma_{i,j-(4\ell-2)} u_{2(i-j)-(i \mod 2)} u_j + \gamma_{i,i-4\ell+2} u_0 u_i \\
&\quad + \sum_{j=i+1}^{6\ell-3} \gamma_{i,j-(4\ell-2)} u_{4\ell-1+i-j} u_j & \text{for } 4\ell-1 \leq i \leq 6\ell-3.
\end{align*}
$$

If we generalize these equations to the Rainbow signature scheme, the central map is given by Figure 13.4.

13.3 Security in a Nutshell

To provide a fair comparison between UOV, Rainbow and enTTS regarding memory consumption and running time, we first have to choose parameters of the same level of security. Therefore we briefly revisit the latest attacks and choices of parameters of all three schemes.
Chapter 13. Multivariate Quadratics Public-Key Schemes

13.3.1 Security and Parameters of UOV and 0/1-UOV

Direct Attack.

To forge a single signature an attacker would have to solve a system of \( o \) quadratic equations in \( v \) variables over \( \mathbb{F}_q \). The usual way of finding one solution is first guessing \( v \) variables at random. This preserves one solution with high probability. The best way of solving the remaining MQ-system of \( o \) equations and variables is to guess a few further variables and then apply some Gröbner Basis algorithm like \( F_4 \) (see Hybrid Approach of Bettale et al. [BFP09]). Recently Thomae et al. showed that we can do better than guessing \( v \) variables at random [TW12].

Calculating these \( v \) variables through linear systems of equations allows to solve a system of \( o - \left\lfloor \frac{v}{2} \right\rfloor \) quadratic equations and variables afterwards. To determine the complexity of solving a MQ-system using a Groebner basis algorithm like \( F_4 \) we refer to [BFP09]. In a nutshell, we first have to calculate the degree of regularity \( d_{\text{reg}} \). For semi-regular sequences, which generic systems are assumed to be, the degree of regularity is the index of the first non-positive coefficient in the Hilbert series \( S_{m,n} \) with

\[
S_{m,n} = \frac{\prod_{i=1}^{m} (1 - z^{d_i})}{(1 - z)^n},
\]

where \( d_i \) is the degree of the \( i \)-th equation. Then the complexity of solving a zero-dimensional (semi-regular) system using \( F_4 \) [BFP09, Prop. 2.2] is

\[
\mathcal{O} \left( m \left( n + d_{\text{reg}} - 1 \right) \right)^{\alpha},
\]

with \( 2 \leq \alpha \leq 3 \) the linear algebra constant. We used \( \alpha = 2 \) throughout the paper.

Key Recovery Attacks

There are two key recovery attacks known so far. The first is a purely algebraic attack called Reconciliation attack [BBD08]. In order to obtain the secret key \( S \) we have to solve \( \left( \frac{k+1}{2} \right) o \) quadratic equations in \( kv \) variables for an optimal parameter \( k \in \mathbb{N} \). The second attack is a variant of the Kipnis-Shamir attack on the balanced Oil and Vinegar scheme [KS98]. The overall complexity of this attack is \( \mathcal{O}(q^{v-o^2-o^4}) \). Note that \( v = 2o \) is very conservative in order to prevent this attack and thus \( v \) can be chosen much smaller for \( o \) large enough. As \( k \geq 2 \) even
13.3. Security in a Nutshell

Table 13.1: Minimal 0/1-UOV parameters achieving certain levels of security. Thereby \( g \) is the optimal number of variables to guess in the hybrid approach and \( k \) is the optimal parameter selectable for the Reconciliation attack.

<table>
<thead>
<tr>
<th>security parameter ((o,v))</th>
<th>direct attack</th>
<th>Reconciliation</th>
<th>Kipnis-Shamir</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^{64}) ((21,28))</td>
<td>(2^{67}) ((g = 1))</td>
<td>(2^{131}) ((k = 2))</td>
<td>(2^{66})</td>
</tr>
<tr>
<td>(2^{80}) ((28,37))</td>
<td>(2^{85}) ((g = 1))</td>
<td>(2^{166}) ((k = 2))</td>
<td>(2^{83})</td>
</tr>
<tr>
<td>(2^{128}) ((44,59))</td>
<td>(2^{130}) ((g = 1))</td>
<td>(2^{256}) ((k = 2))</td>
<td>(2^{134})</td>
</tr>
</tbody>
</table>

Table 13.2: Minimal Rainbow parameters achieving certain levels of security. Thereby \( g \) is the optimal number of variables to guess for the hybrid approach.

<table>
<thead>
<tr>
<th>security ((v_1,o_1,o_2))</th>
<th>direct attack</th>
<th>Band</th>
<th>MinRank</th>
<th>HighRank</th>
<th>Kipnis</th>
<th>Reconciliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^{64}) ((15,10,10))</td>
<td>(2^{67}) ((g = 1))</td>
<td>(2^{70})</td>
<td>(2^{141})</td>
<td>(2^{93})</td>
<td>(2^{125})</td>
<td>(2^{242}) ((k = 6))</td>
</tr>
<tr>
<td>(2^{80}) ((18,13,14))</td>
<td>(2^{85}) ((g = 1))</td>
<td>(2^{81})</td>
<td>(2^{167})</td>
<td>(2^{126})</td>
<td>(2^{143})</td>
<td>(2^{254}) ((k = 5))</td>
</tr>
<tr>
<td>(2^{128}) ((36,21,22))</td>
<td>(2^{131}) ((g = 2))</td>
<td>(2^{131})</td>
<td>(2^{313})</td>
<td>(2^{192})</td>
<td>(2^{290})</td>
<td>(2^{523}) ((k = 7))</td>
</tr>
</tbody>
</table>

the Reconciliation attack will not badly benefit of choosing \( v \) smaller and direct attacks even suffer of such a choice.

13.3.2 Security and Parameters of Rainbow

All attacks against UOV also apply to Rainbow. Additionally the security of Rainbow relies on the MinRank-problem. Thus we also have to take MinRank and HighRank attacks, as well as the Rainbow Band Separation attack into account. See Petzold et al. [PBB10] for an overview of the attacks and the parameters to choose.

13.3.3 Security and Parameters of Enhanced TTS

All attacks against Rainbow also apply to enTTS. The only attack that seriously benefit from the changes made between Rainbow and enTTS is the Reconciliation attack with large \( k \). But as the complexities of this attacks are out of reach anyway this do not affect the security. Actually the complexity is higher than the ones of all the other attacks, so we omit it. More important is the slight benefit of the Band Separation attack. For the odd sequence enTTS we derive \( m + n - 1 \) quadratic equations in \( n - 2 \) instead of \( n \) variables.
Table 13.3: Minimal odd sequence enTTS parameters achieving certain levels of security. Thereby \( g \) is the optimal number of variables to guess for the hybrid approach.

<table>
<thead>
<tr>
<th>security ( (\ell, m, n) )</th>
<th>direct attack Band</th>
<th>MinRank</th>
<th>HighRank</th>
<th>Kipnis-Shamir</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{64} ) ( (7, 28, 40) )</td>
<td>( 2^{89} (g = 1) )</td>
<td>( 2^{68} )</td>
<td>( 2^{126} )</td>
<td>( 2^{117} )</td>
</tr>
<tr>
<td>( 2^{80} ) ( (9, 36, 52) )</td>
<td>( 2^{110} (g = 2) )</td>
<td>( 2^{85} )</td>
<td>( 2^{159} )</td>
<td>( 2^{151} )</td>
</tr>
<tr>
<td>( 2^{128} ) ( (15, 60, 88) )</td>
<td>( 2^{176} (g = 3) )</td>
<td>( 2^{131} )</td>
<td>( 2^{258} )</td>
<td>( 2^{249} )</td>
</tr>
</tbody>
</table>

13.4 Implementation on AVR Microprocessors

The goal of these implementations is a fair comparison between some of the most promising MQ-based post quantum public-key schemes. All schemes were analysed in the previous section and sets of parameters with equivalent security were defined under considerations of most recent attacks. A problem when comparing such schemes is that every implementation has its own philosophy of what is most worthy of optimization. Therefore we aim for a comparison with equal conditions for all schemes such as the same platform and implementation by the same person, also with nearly the same possible optimizations. Additionally practical figures are given in a real world scenario for signature verification and generation time. All the schemes were implemented with runtime optimization in mind.

13.4.1 Target Platform and Tools

An ATxMega128a1 on an xplain board was used as target device. This micro processor has a clock frequency of 32 MHz, 128KB flash program memory and 8KB SRAM. The code was written in C and optimized for embedded use. As compiler avr-gcc in version 4.5.1 and at some places assembler gcc-as 2.20.1 was used.

Polynomial Representation / Key Storage

When implementing MQPKS on microprocessors it is important to construct an efficient way of storing and reading the keys out of memory. All polynomials of an MQ-scheme are represented by their coefficients. It is important to decide how this coefficients are processed during runtime. The coefficients of UOV and Rainbow can be easily mapped to some readout loops. This is not that easy with enTTS as only a minimal count of coefficients are used and this few coefficients are spread over three layers and six different cyclic structures. As random access on the flash memory produces a lot of addressing overhead while calculating the address each time a serial approach was chosen. All coefficients are stored in memory in the same exact order in which they are read out. There are no gaps or zeros in memory which is also memory efficient. This memory architecture allows us to read out the keys directly and simply increment the address to reach the
13.4. Implementation on AVR Microprocessors

The AVR instruction set allows a memory readout with a post increment in one clock cycles from SRAM or two clock cycles from Flash memory. Therefore no additional address calculation is needed. The number of coefficients to store and thus the memory consumptions in bytes is $o \left( \alpha v + \frac{v(v+1)}{2} \right)$ for UOV, $o_1 \left( \alpha_1 v + \frac{v(v+1)}{2} \right) + o_2 \left( \alpha_2 (v + o_1) + \frac{(v+o_1)(v+o_1+1)}{2} \right)$ for Rainbow and $8l^2 - 6l - 3$ for enTTS. The resulting memory requirements for specific security parameters are given in Table 13.5.

13.4.2 Arithmetic and Finite Field

As the used microprocessor is based on an 8 bit architecture, working in $F_{2^8}$ seems advantageous. Multiplication is done by a table look up, each element is brought to its exponential representation, processed and then transformed back to the normal polynomial representation. Every transformation from the exponential to the basis representation costs one memory access, therefore in all implementations the exponential representation is kept as long as no $F_{2^8}$ addition takes place, which is a bitwise exclusive OR operation of two coefficients in the basis representation. As the coefficients of the keys are first read in by a multiplication, all keys are already stored in the exponential form. Random numbers are generated by the rand() gcc pseudo random number generator. This function is seeded with a value derived from uninitialized SRAM blocks which are arbitrary on every start up.

Inverting the Layers

All schemes require the inversion of multivariate systems of equations. As only linear systems of equation can be solved efficiently, we have to fix variables until the system gets linear and then perform a simple Gaussian elimination using LU decomposition. Here the exponential representation is also used where possible. For example the lower matrix and all variables were saved in exponential form. In enTTS the middle layer consists only of polynomials depending on already known variables. Therefore these polynomials can be inverted directly.

13.4.3 Key Size and Signature Runtime Reduction

The main problem of MQ-schemes are large keys, as storage space is limited on embedded devices. Large private keys come also together with long signature time, due to the processing of more data. As the signature for a fixed message is not unique, there is a lot of redundancy that can be used to reduce the secret key $S$ (cf. theory of equivalent keys). We used such minimal keys for UOV as well as for Rainbow. Note that there are no equivalent keys known for enTTS and thus the whole matrix $S$ has to be stored. The special form of $S$ has two additional side effects in addition to less space. First, also the signature time is reduced. The multiplication with the identity matrix corresponds to a copy of the signature so that only the multiplication with the remaining coefficients has to be done. For UOV this saves $\frac{v}{2}(v-1)v + \frac{v}{2}(v-1)\alpha_1$ equations and for Rainbow $\frac{v}{2}(v-1)v + \frac{v}{2}(\alpha_1-1)\alpha_2 + \frac{v}{2}(\alpha_2-1)\alpha_1$. The second observation is that due to the identity matrix in the vinegar × vinegar part, large parts of $P$ and $F$ are equal. They do not increase
security an can be seen as a system parameter (cf. [PTBW11]). As required by the authors of [PTBW11] for 0/1-UOV, also a different monomial ordering was chosen according to a minimal Turán graph. This reordering prevent easier attacks on the public key. The same procedure is probably possible for Rainbow. But as no publication exists which investigated this case, it was not implemented. For enTTS this is not possible as the Tame equations in the middle layer cause to blur the variable structure and no equivalent keys are known.

13.4.4 Verify Runtime Reduction

In the case of 0/1-UOV, choosing the coefficient from \( \mathbb{F}_2 \) has another advantage besides of less memory consumption. The verification and signature generation time can be reduced. As we know that the majority of coefficients are from \( \mathbb{F}_2 \), we can check for a one or a zero, which leads to a copy instruction in the case of one or a skip instruction in case of zero. Only otherwise we have to perform a costly multiplication in \( \mathbb{F}_{2^8} \). The effect is in our implementation not marginally visible, because the used table look up method is fast compared to a schoolbook multiplication method.

13.4.5 RAM Requirements

MQ-schemes do not need a lot of RAM, in contrast to the persistent flash memory requirements. In Table 13.4 the requirements are listed. Besides RAM needed for persistent, counting or temporary variables, only the Gaussian elimination algorithm needs a noticeable amount of RAM. As the inversion is computed in place, only one quadratic systems at time has to be stored in RAM. In case of multiple layers the maximal requirements are defined by the largest system of equations to be solved.

<table>
<thead>
<tr>
<th>security</th>
<th>2^64</th>
<th>2^80</th>
<th>2^128</th>
<th>general</th>
</tr>
</thead>
<tbody>
<tr>
<td>UOV</td>
<td>441</td>
<td>784</td>
<td>1936</td>
<td>( m^2 )</td>
</tr>
<tr>
<td>Rainbow</td>
<td>400</td>
<td>729</td>
<td>1849</td>
<td>((o_1 + o_2)^2)</td>
</tr>
<tr>
<td>enTTS</td>
<td>169</td>
<td>289</td>
<td>841</td>
<td>((2l - 1)^2)</td>
</tr>
</tbody>
</table>

Table 13.4: Minimal Ram Requirements for LES Solving in Bytes

13.4.6 Key Generation

The keys for all schemes are generated on a standard PC using a C program. Basically \( T \circ F \circ S = P \) has to be computed. Using the quadratic form, the composition can be written as in (13.4.1). An overview of the key generation process of 0/1 UOV with small parameters can be found in the appendix.
13.5 Results

Another way to generate an UOV key is described in [PTBW11]. It can be done by transforming the matrix $S$ into a matrix $A_{uov}$ and write all coefficients of $f^{(i)}$ ordered lexicographically to the rows of $Q$. Then the following equation holds: $A_{uov} \cdot Q = S^T \bar{\mathbf{s}}^{(i)} S$. With this relation inverting $A_{uov}$ is possible and therefore an inverse approach, choosing first $\mathcal{P}$ and then applying $A_{uov}$ to get $\mathcal{F}$. For the runtime optimization the reordering of monomials can take place in $A_{uov}$ instead of reorder the monomials in $\mathcal{P}$ and $\mathcal{F}$.

13.5 Results

Table 13.5 shows our achieved results. They are easy to compare because schemes are grouped by security level. For all schemes key size, runtime and code size are given. Where applicable the system parameter size is also included. The public and secret key sizes can be easily calculated. One element responds to one byte and no other overhead needs to be saved so the keys consists only of the coefficients of the public or secret maps and the linear transformations. In the case of $0/1$-UOV a large part is fixed and declared as a system parameter, but it must be anyway saved or be easy to generate in a real world scenario, therefore thus size is also listed.

Clock cycles were count internally with two concatenated 16 bit counters which are enabled to count on every clock cycle. As the count of verify operations scales with $(n \cdot (n+1) \cdot m)$ the measured times do not surprise. As enTTS uses the largest numbers of $n$ and $m$ it has the lowest verify performance and the largest public keys. Rainbow is the fastest as the parameters can be chosen relatively low. The big advantage of enTTS is the small private key. Large parts of the central map are zero and have not to be saved. In terms of theoretical public key size $0/1$-UOV performs the best. If the possibility to generate the system parameter on the device would exist, it would ensure the smallest public key. The gain of verification and signature time in comparison to the standard UOV is only minimal as the multiplication by table look up has no significant runtime difference in comparison to a multiplication with 0 or 1 as the 0 case is a special case and is checked anyway every time in a normal multiplication in $\mathbb{F}_2$. When measuring scalability for secret/public key size at the step from $2^{64}$ to $2^{128}$, UOV has an increase factor of 9/9, 0/1-UOV of 9/9, Rainbow 10/11 and enTTS of 4/10. UOV scales the best in public key size, enTTS the best in private key size. Regarding the signature size, UOV has the highest expansion factor, with a message to signature ratio of approximately 2.3, followed by Rainbow with 1.7 and enTTS with 1.4.

As a comparison of an µC with an ASIC or PC implementation is meaningless, the only MQ implementation we can compare with is the one from [YCCC06]. The authors implemented enTTS(5, 20, 28) on a MSP430 running at 8 MHz. Signing requires 17.75 ms and verifying 181.5 ms, when scaled up to our clock frequency. Although, the MSP430 is a 16 bit CPU, our implementation is a factor of 3.7 faster in signing and 5.1 times faster in verifying.
<table>
<thead>
<tr>
<th>Scheme</th>
<th>n</th>
<th>m</th>
<th>Key Size [Byte]</th>
<th>System Parameter</th>
<th>Clockcycles x 1000</th>
<th>Time [ms] @32MHz</th>
<th>Code Size [Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>private public</td>
<td>private public</td>
<td>sign verify</td>
<td>sign verify</td>
<td></td>
</tr>
<tr>
<td>enTTS(5, 20, 28)</td>
<td>28</td>
<td>20</td>
<td>1351 8120</td>
<td>*</td>
<td>*</td>
<td>153 1,126</td>
<td>4.79 35.22</td>
</tr>
<tr>
<td>enTTS(5, 20, 28)[YCCC06]</td>
<td>28</td>
<td>20</td>
<td>1417 8680</td>
<td>*</td>
<td>*</td>
<td>568 1,508^1</td>
<td>17.75 181.5^2</td>
</tr>
<tr>
<td>uov(21, 28)</td>
<td>49</td>
<td>21</td>
<td>21462 25725</td>
<td>*</td>
<td>*</td>
<td>1,615 1,690</td>
<td>50.49 52.83</td>
</tr>
<tr>
<td>0/1 uov(21, 28)</td>
<td>49</td>
<td>21</td>
<td>12936 4851</td>
<td>8526 20874</td>
<td>1,577 1,395</td>
<td>49.29 43.60</td>
<td>2258 578</td>
</tr>
<tr>
<td>rainbow(15, 10, 10)</td>
<td>35</td>
<td>20</td>
<td>9250 12600</td>
<td>*</td>
<td>*</td>
<td>848 1,010</td>
<td>26.51 31.58</td>
</tr>
<tr>
<td>enTTS(7, 28, 40)</td>
<td>40</td>
<td>28</td>
<td>2731 22960</td>
<td>*</td>
<td>*</td>
<td>332 2,558</td>
<td>10.37 49.29</td>
</tr>
<tr>
<td>uov(28, 37)</td>
<td>65</td>
<td>28</td>
<td>49728 60060</td>
<td>*</td>
<td>*</td>
<td>3,637 3,911</td>
<td>113.66 122.23</td>
</tr>
<tr>
<td>0/1 uov(28, 37)</td>
<td>65</td>
<td>28</td>
<td>30044 11368</td>
<td>19684 48692</td>
<td>3,526 3,211</td>
<td>110.20 100.37</td>
<td>2258 578</td>
</tr>
<tr>
<td>rainbow(18, 13, 14)</td>
<td>45</td>
<td>27</td>
<td>19682 27945</td>
<td>*</td>
<td>*</td>
<td>1,740 2,214</td>
<td>54.38 69.19</td>
</tr>
<tr>
<td>enTTS(9, 36, 52)</td>
<td>52</td>
<td>36</td>
<td>4591 49608</td>
<td>*</td>
<td>*</td>
<td>609 6,658</td>
<td>19.03 418.07</td>
</tr>
<tr>
<td>uov(44, 59)</td>
<td>103</td>
<td>44</td>
<td>194700 235664</td>
<td>*</td>
<td>*</td>
<td>13,314 14,134</td>
<td>416.07 441.70</td>
</tr>
<tr>
<td>0/1 uov(44, 59)</td>
<td>103</td>
<td>44</td>
<td>116820 43560</td>
<td>77880 192104</td>
<td>12,782 13,569</td>
<td>399.43 424.04</td>
<td>2258 578</td>
</tr>
<tr>
<td>rainbow(36, 21, 22)</td>
<td>79</td>
<td>43</td>
<td>97675 135880</td>
<td>*</td>
<td>*</td>
<td>8,227 9,216</td>
<td>257.11 288.01</td>
</tr>
<tr>
<td>enTTS(15, 60, 88)</td>
<td>88</td>
<td>60</td>
<td>13051 234960</td>
<td>*</td>
<td>*</td>
<td>2,142 3,0789</td>
<td>66.94 962.17</td>
</tr>
</tbody>
</table>

Table 13.5: Results

* Not applicable
1 Derived from values in original work
2 Scaled to the same clock frequency
13.6 Conclusion

Also when comparing our work with implementations of the classical signature schemes RSA and ECDSA, all four schemes perform well. For example, [GPW+04] reports 203ms for a ECC sign operation with $2^{80}$ bit security, where our implementations are two to ten times faster. For the verifying operation our work is up to three times faster. Due to the short exponent in RSA-verify, [GPW+04] verifies in the same order of magnitude. But the RSA-sign operation is at least a factor of 25 slower than our work. Table 13.6 summarizes other implementations on comparable 8 bit platforms.

Table 13.6: Overview of other implementatations on comparable platforms.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time[ms]@32MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sign</td>
</tr>
<tr>
<td>enTTS(5, 20, 28) [YCCC06]</td>
<td>17.75</td>
</tr>
<tr>
<td>ECC-P160 (SECG) [GPW+04]</td>
<td>203</td>
</tr>
<tr>
<td>ECC-P192 (SECG) [GPW+04]</td>
<td>310</td>
</tr>
<tr>
<td>ECC-P224 (SECG) [GPW+04]</td>
<td>548</td>
</tr>
<tr>
<td>RSA-1024 [GPW+04]</td>
<td>2,748</td>
</tr>
<tr>
<td>RSA-2048 [GPW+04]</td>
<td>20,815</td>
</tr>
<tr>
<td>NTRU-251-127-31 sign [DPP08]</td>
<td>143</td>
</tr>
</tbody>
</table>

1 For a fair comparison with our implementation running at 32MHz, timings at lower frequencies were scaled accordingly.

13.6 Conclusion

In this work we present the first µC implementations of the three most common MQPKS since nearly 10 years. Additionally, we implemented for the first time 0/1-UOV on a constrained device. All recent attacks were summarized and we proposed current security parameters for $2^{64}$, $2^{80}$ and $2^{128}$ bit symmetric security. Additionally, we showed that choosing $v = 2^0$ for UOV is outdated. When comparing with existing MQ implementations, ours are a factor of three and five times faster in signing and verifying, respectively. We hope our implementations will inspire follow up work, to improve acceptance of MQPKS in constrained environments.

13.6.1 Further Improvements

There is still space for improvements and the upper limit is not reached yet. A few ideas were not implemented in this work. Saving the system parameters is not optimal. Here a replacement by a pseudo random number generator or an other generator function would reduce the public key drastically, even if verification time would be increased. In our implementation all elements
of $F_2$ are saved as a byte value. It would be possible to achieve smaller keys when saving 8 elements in one byte, combined with a verification function which utilizes assembler instructions maybe even a faster verification could be possible. An overall time vs. code size trade-off is still a topic to investigate. MQ-schemes are very well scalable in regard to this trade-off.

13.7 Toy example of 0/1 UOV Key Generation
Figure 13.5: 0/1 UOV Key Generation. For details see [PTBW11].
Chapter 14

LaPin: An Efficient Authentication Protocol Based on Ring-LPN

This research contribution is based on the author’s published research in [HKL+ 10]. It is joint work with Eike Kiltz, Vadim Lyubashesky and Krzysztof Pietrzak.

This chapter proposes a new Hopper-Blum (HB) style authentication protocol that is provably secure based on a ring variant of the learning parity with noise (LPN) problem. The protocol is secure against active attacks, consists of only two rounds, has small communication complexity, and has a very small footprint which makes it very applicable in scenarios that involve low-cost, resource-constrained devices. Performance-wise, our protocol is the most efficient of the HB family of protocols and our implementation results show that it is even comparable to the standard challenge-and-response protocols based on the AES block-cipher. Our basic protocol is roughly 20 times slower than AES, but with the advantage of having 10 times smaller code size. Furthermore, if a few hundred bytes of non-volatile memory are available to allow the storage of some off-line pre-computations, then the online phase of our protocols is only twice as slow as AES.

14.1 Introduction

Lightweight shared-key authentication protocols, in which a tag authenticates itself to a reader, are extensively used in resource-constrained devices such as radio-frequency identification (RFID) tags or smart cards. The straight-forward approach for constructing secure authentications schemes is to use low-level symmetric primitives such as block-ciphers, e.g., AES [DR02]. In their most basic form, the protocols consist of the reader sending a short challenge $c$ and the tag responding with $AES_K(c)$, where $K$ is the shared secret key. The protocol is secure if AES fulfills a strong, interactive security assumption, namely that it behaves like a strong pseudo-random function.

Authentication schemes based on AES have some very appealing features: they are extremely fast, consist of only 2 rounds, and have very small communication complexities. In certain scenarios, however, such as when low-cost and resource-constrained devices are involved, the
relatively large gate-count and code size used to implement AES may pose a problem. One approach to overcome the restrictions presented by low-weight devices is to construct a low-weight block cipher (e.g., PRESENT [BKL+07]), while another approach has been to deviate entirely from block-cipher based constructions and build a provably-secure authentication scheme based on the hardness of some mathematical problem. In this work, we concentrate on this second approach.

Ideally, one would like to construct a scheme that incorporates all the beneficial properties of AES-type protocols, while also acquiring the additional provable security and smaller code description characteristics. In the past decade, there have been proposals that achieved some, but not all, of these criteria. The most notable of these proposals fall into the Hopper-Blum (HB) line of protocols, which we will survey in detail below. Our proposal can be seen as a continuation of this line of research that contains all the advantages enjoyed by HB-type protocols, while at the same time, getting even closer to enjoying the benefits of AES-type schemes.

Previous Works. Hopper and Blum [HB00, HB01] proposed a 2-round authentication protocol that is secure against passive adversaries based on the hardness of the LPN problem (we remind the reader of the definition of the LPN problem in Section 14.1.1). The characteristic feature of this protocol is that it requires very little workload on the part of the tag and the reader. Indeed, both parties only need to compute vector inner products and additions over \( \mathbb{F}_2 \), which makes this protocol (thereafter named HB) a good candidate for lightweight applications.

Following this initial work, Juels and Weis constructed a protocol called HB\(^+\) [JW05] which they proved to be secure against more realistic, so called active attacks. Subsequently, Katz et al. [KS06a, KS06b, KSS10] provided a simpler security proof for HB\(^+\) as well as showed that it remains secure when executed in parallel. Unlike the HB protocol, however, HB\(^+\) requires three rounds of communication between tag and reader. From a practical aspect, 2 round authentication protocols are often advantageous over 3 round protocols. They often show a lower latency which is especially pronounced on platforms where the establishment of a communication in every directions is accompanied by a fixed initial delay. An additional drawback of both HB and HB\(^+\) is that their communication complexity is on the order of hundreds of thousands of bits, which makes them almost entirely impractical for lightweight authentication tokens because of timing and energy constraints. (The contactless transmission of data on RFIDs or smart cards typically requires considerably more energy than the processing of the same data.)

To remedy the overwhelming communication requirement of HB\(^+\), Gilbert et al. proposed the three-round HB\(^\sharp\) protocol [GRS08a]. A particularly practical instantiation of this protocol requires fewer than two thousand bits of communication, but is no longer based on the hardness of the LPN problem. Rather than using independent randomness, the HB\(^\sharp\) protocol utilized a Toeplitz matrix, and is thus based on a plausible assumption that the LPN problem is still hard in this particular scenario.

A feature that the HB, HB\(^+\), and HB\(^\sharp\) protocols have in common is that at some point the reader sends a random string \( r \) to the tag, which then must reply with \( \langle r, s \rangle + e \), the inner product of \( r \) with the secret \( s \) plus some small noise \( e \). The recent work of Kiltz et al. [KPC+11] broke with
14.1. Introduction

this approach, and they were able to construct the first 2-round LPN-based authentication protocol (thereafter named HB^2) that is secure against active attacks. In their challenge-response protocol, the reader sends some challenge bit-string c to the tag, who then answers with a noisy inner product of a random r (which the tag chooses itself) and a session-key K(c), where K(c) selects (depending on c) half of the bits from the secret s. Unfortunately, the HB^2 protocol still inherits the large communication requirement of HB and HB^+. Furthermore, since the session key K(c) is computed using bit operations, it does not seem to be possible to securely instantiate HB^2 over structured (and hence more compact) objects such as Toeplitz matrices (as used in HB^1 [GRS08a]).

14.1.1 LPN, Ring-LPN, and Related Problems

The security of our protocols relies on the new Ring Learning Parity with Noise (Ring-LPN) problem which is a natural extension of the standard Learning Parity with Noise (LPN) problem to rings. It can also be seen as a particular instantiation of the Ring-LWE (Learning with Errors over Rings) problem that was recently shown to have a strong connection to lattices [LPR10]. We will now briefly describe and compare these hardness assumptions, and we direct the reader to Section 14.3 for a formal definition of the Ring-LPN problem.

The decision versions of these problems require us to distinguish between two possible oracles to which we have black-box access. The first oracle has a randomly generated secret vector s ∈ F_2^n which it uses to produce its responses. In the LPN problem, each query to the oracle produces a uniformly random matrix A ∈ F_2^{n×n} and a vector As + e = t ∈ F_2^n where e is a vector in F_2^n each of whose entries is an independently generated Bernoulli random variable with probability of 1 being some public parameter τ between 0 and 1/2. The second oracle in the LPN problem outputs a uniformly-random matrix A ∈ F_2^{n×n} and a uniformly random vector t ∈ F_2^n. The only difference between LPN and Ring-LPN is in the way the matrix A is generated (both by the first and second oracle). While in the LPN problem, all its entries are uniform and independent, in the Ring-LPN problem, only its first column is generated uniformly at random in F_2^n. The remaining n columns of A depend on the first column and the underlying ring R = F_2[X]/(f(X)). If we view the first column of A as a polynomial r ∈ R, then the i-th column (for 0 ≤ i ≤ n − 1) of A is just the vector representation of rX^i in the ring R. Thus when the oracle returns As + e, this corresponds to it returning the polynomial r · s + e where the multiplication of polynomials r and s (and the addition of e) is done in the ring R. In Section 14.3, we discuss how the choice of the ring R affects the security of the problem.

While the standard Learning Parity with Noise (LPN) problem has found extensive use as a cryptographic hardness assumption (e.g., [HB01, JW05, GRS08b, GRS08a, ACPS09, KSS10]), we are not aware of any constructions that employed the Ring-LPN problem. There have been some previous works that considered some relatively similar “structured” versions of LPN. The HB^2 authentication protocol of Gilbert et al. [GRS08a] made the assumption that for a random

---

1In the more common description of the LPN problem, each query to the oracle produces one random sample in F_2^n. For comparing LPN to Ring-LPN, however, it is helpful to consider the oracle as returning a matrix of n random independent samples on each query.
Toeplitz matrix $S \in \mathbb{F}_2^{m \times n}$, a uniformly random vector $a \in \mathbb{F}_2^n$, and a vector $e \in \mathbb{F}_2^m$ whose coefficients are distributed as $\text{Ber}_\tau$, the output $(a, Sa + e)$ is computationally indistinguishable from $(a, t)$ where $t$ is uniform over $\mathbb{F}_2^n$.

Another related work, as mentioned above, is the recent result of Lyubashevsky et al. [LPR10], where it is shown that solving the decisional Ring-LWE (Learning with Errors over Rings) problem is as hard as quantumly solving the worst case instances of the shortest vector problem in ideal lattices. The Ring-LWE problem is quite similar to Ring-LPN, with the main difference being that the ring $R$ is defined as $\mathbb{F}_q[X]/(f(X))$ where $f(X)$ is a cyclotomic polynomial and $q$ is a prime such that $f(X)$ splits completely into $\deg(f(X))$ distinct factors over $\mathbb{F}_q$.

Unfortunately, the security proof of our authentication scheme does not allow us to use a polynomial $f(X)$ that splits into low-degree factors, and so we cannot base our scheme on lattice problems. For a similar reason (see the proof of our scheme in Section 14.4 for more details), we cannot use samples that come from a Toeplitz matrix as in [GRS08a]. Nevertheless, we believe that the Ring-LPN assumption is very natural and will find further cryptographic applications, especially for constructions of schemes for low-cost devices.

### 14.2 Definitions

#### 14.2.1 Rings and Polynomials

For a polynomial $f(X)$ over $\mathbb{F}_2$, we will often omit the indeterminate $X$ and simply write $f$. The degree of $f$ is denoted by $\deg(f)$. For two polynomials $a, f$ in $\mathbb{F}_2[X]$, $a \mod f$ is defined to be the unique polynomial $r$ of degree less than $\deg(f)$ such that $a = fg + r$ for some polynomial $g \in \mathbb{F}_2[X]$. The elements of the ring $\mathbb{F}_2[X]/(f)$ will be represented by polynomials in $\mathbb{F}_2[X]$ of maximum degree $\deg(f) - 1$. In this paper, we will only be considering rings $R = \mathbb{F}_2[X]/(f)$ where the polynomial $f$ factors into distinct irreducible factors over $\mathbb{F}_2$. For an element $a$ in the ring $\mathbb{F}_2[X]/(f)$, we will denote by $\hat{a}$, the CRT (Chinese Remainder Theorem) representation of $a$ with respect to the factors of $f$. In other words, if $f = f_1 \ldots f_m$ where all $f_i$ are irreducible, then

$$\hat{a} \overset{\Delta}{=} (a \mod f_1, \ldots, a \mod f_m).$$

If $f$ is itself an irreducible polynomial, then $\hat{a} = a$. Note that an element $\hat{a} \in R$ has a multiplicative inverse iff, for all $1 \leq i \leq m$, $a \not\equiv 0 \mod f_i$. We denote by $R^*$ the set of elements in $R$ that have a multiplicative inverse.

#### 14.2.2 Distributions

For a distribution $D$ over some domain, we write $r \overset{\$}{\leftarrow} D$ to denote that $r$ is chosen according to the distribution $D$. For a domain $Y$, we write $U(Y)$ to denote the uniform distribution over $Y$. Let $\text{Ber}_\tau$ be the Bernoulli distribution over $\mathbb{F}_2$ with parameter (bias) $\tau \in [0,1/2]$ (i.e., $\Pr[x = 1] = \tau$ if $x \leftarrow \text{Ber}_\tau$). For a polynomial ring $R = \mathbb{F}_2[X]/(f)$, the distribution $\text{Ber}_\tau^R$ denotes the distribution over the polynomials of $R$, where each of the $\deg(f)$ coefficients of the
polynomial is drawn independently from $\textsf{Ber}_\tau$. For a ring $R$ and a polynomial $s \in R$, we write $\Lambda_{\tau,s}^R$ to be the distribution over $R \times R$ whose samples are obtained by choosing a polynomial $r \xleftarrow{} U(R)$ and another polynomial $e \xleftarrow{} \textsf{Ber}_\tau^R$, and outputting $(r, rs + e)$.

### 14.2.3 Authentication Protocols

An authentication protocol $\Pi$ is an interactive protocol executed between a Tag $T$ and a reader $R$, both PPT algorithms. Both hold a secret $x$ (generated using a key-generation algorithm $\textsf{KG}$ executed on the security parameter $\lambda$ in unary) that has been shared in an initial phase. After the execution of the authentication protocol, $R$ outputs either accept or reject. We say that the protocol has completeness error $\varepsilon_c$ if for all $\lambda \in \mathbb{N}$, all secret keys $x$ generated by $\textsf{KG}(1^\lambda)$, the honestly executed protocol returns reject with probability at most $\varepsilon_c$. We now define different security notions of an authentication protocol.

**Passive attacks.** An authentication protocol is secure against passive attacks, if there exists no PPT adversary $A$ that can make the reader $R$ return accept with non-negligible probability after (passively) observing any number of interactions between reader and tag.

**Active attacks.** A stronger notion for authentication protocols is security against active attacks. Here the adversary $A$ runs in two stages. First, she can interact with the honest tag a polynomial number of times (with concurrent executions allowed). In the second phase $A$ interacts with the reader only, and wins if the reader returns accept. Here we only give the adversary one shot to convince the verifier. An authentication protocol is $(t,q,\varepsilon)$-secure against active adversaries if every PPT $A$, running in time at most $t$ and making $q$ queries to the honest reader, has probability at most $\varepsilon$ to win the above game.

### 14.3 Ring-LPN and its Hardness

The decisional Ring-LPN$_R$ (Ring Learning Parity with Noise in ring $R$) assumption, formally defined below, states that it is hard to distinguish uniformly random samples in $R \times R$ from those sampled from $\Lambda_{\tau,s}^R$ for a uniformly chosen $s \in R$.

**Definition 14.3.1** (Ring-LPN$_R$). The (decisional) Ring-LPN$_R$ problem is $(t,Q,\varepsilon)$-hard if for every distinguisher $D$ running in time $t$ and making $Q$ queries,

\[ \left| \Pr \left[ s \xleftarrow{} R : D^{\Lambda_{\tau,s}^R} = 1 \right] - \Pr \left[ D^{U(R \times R)} = 1 \right] \right| \leq \varepsilon. \]

### 14.3.1 Hardness of LPN and Ring-LPN

One can attempt to solve Ring-LPN using standard algorithms for LPN, or by specialized algorithms that possibly take advantage of Ring-LPN’s additional structure. Some work towards

---

*By using a hybrid argument one can show that this implies security even if the adversary can interact in $k \geq 1$ independent instances concurrently (and wins if the verifier accepts in at least one instance). The use of the hybrid argument loses a factor of $k$ in the security reduction.*
constructing the latter type of algorithm has recently been done by Hanrot et al. [HLPS11], who show that in certain cases, the algebraic structure of the Ring-LPN and Ring-LWE problems makes them vulnerable to certain attacks. These attacks essentially utilize a particular relationship between the factorization of the polynomial $f(X)$ and the distribution of the noise.

**Ring-LPN with an irreducible $f(X)$**

When $f(X)$ is irreducible over $\mathbb{F}_2$, the ring $\mathbb{F}_2[X]/(f)$ is a field. For such rings, the algorithm of Hanrot et al. does not apply, and we do not know of any other algorithm that takes advantage of the added algebraic structure of this particular Ring-LPN instance. Thus to the best of our knowledge, the most efficient algorithms for solving this problem are the same ones that are used to solve LPN, which we will now very briefly recount.

The computational complexity of the LPN problem depends on the length of the secret $n$ and the noise distribution $\text{Ber}_\tau$. Intuitively, the larger the $n$ and the closer $\tau$ is to $1/2$, the harder the problem becomes. Usually the LPN problem is considered for constant values of $\tau$ somewhere between 0.05 and 0.25. For such constant $\tau$, the fastest asymptotic algorithm for the LPN problem, due to Blum et al. [BKW03], takes time $2^{\Omega(n/\log n)}$ and requires approximately $2^{\Omega(n/\log n)}$ samples from the LPN oracle. If one has access to fewer samples, then the algorithm will perform somewhat worse. For example, if one limits the number of samples to only polynomially-many, then the algorithm has an asymptotic complexity of $2^{\Omega(n/\log n)}$ [Lyu05]. In our scenario, the number of samples available to the adversary is limited to $n$ times the number of executions of the authentication protocol, and so it is reasonable to assume that the adversary will be somewhat limited in the number of samples he is able to obtain (perhaps at most $2^{40}$ samples), which should make our protocols harder to break than solving the Ring-LPN problem. Leveil and Fouque [LF06] made some optimizations to the algorithm of Blum et al. and analyzed its precise complexity. To the best of our knowledge, their algorithm is currently the most efficient one and we will refer to their results when analyzing the security of our instantiations.

In Section 14.5, we base our scheme on the hardness of the Ring-LPN$^R$ problem where $R = \mathbb{F}_2[X]/(X^{532} + X + 1)$ and $\tau = 1/8$. According to the analysis of [LF06], an LPN problem of dimension 512 with $\tau = 1/8$ would require $2^{77}$ memory (and thus at least that much time) to solve when given access to approximately as many samples (see [LF06, Section 5.1]). Since our dimension is somewhat larger and the number of samples will be limited in practice, it is reasonable to assume that this instantiation has 80-bit security.

**Ring-LPN with a reducible $f(X)$**

For efficiency purposes, it is sometimes useful to consider using a polynomial $f(X)$ that is not irreducible over $\mathbb{F}_2$. This will allow us to use the CRT representation of the elements of $\mathbb{F}_2[X]/(f)$ to perform multiplications, which in practice turns out to be more efficient. Ideally, we would like the polynomial $f$ to split into as many small-degree polynomials $f_i$ as possible, but there are some constraints that are placed on the factorization of $f$ both by the security
14.4 Authentication Protocol

proof, and the possible weaknesses that a splittable polynomial introduces into the Ring-LPN problem.

If the polynomial $f$ splits into $f = \prod_{i=1}^{m} f_i$, then it may be possible to try and solve the Ring-LPN problem modulo some $f_i$ rather than modulo $f$. Since the degree of $f_i$ is smaller than the degree of $f$, the resulting Ring-LPN problem may end up being easier. In particular, when we receive a sample $(r, rs + e)$ from the distribution $\Lambda_{R,s}^{R,s}$, we can rewrite it in CRT form as

\[
(\hat{r}, \hat{rs} + e) = (\text{mod } f_1, rs + e \text{ mod } f_1), \ldots, \text{mod } f_m, rs + e \text{ mod } f_m),
\]

and thus for every $f_i$, we have a sample

\[
(r \text{ mod } f_i, (r \text{ mod } f_i)(s \text{ mod } f_i) + e \text{ mod } f_i),
\]

where all the operations are in the ring (or field) $F_2[X]/(f_i)$. Thus solving the (decision) Ring-LPN problem in $F_2[X]/(f)$ reduces to solving the problem in $F_2[X]/(f_i)$. The latter problem is in a smaller dimension, since $\deg(s) > \deg(s \text{ mod } f_i)$, but the error distribution of $(e \text{ mod } f_i)$ is quite different than that of $e$. While each coefficient of $e$ is distributed independently as $Ber_\tau$, each coefficient of $(e \text{ mod } f_i)$ is distributed as the distribution of a sum of certain coefficients of $e$, and therefore the new error is larger. The exact which coefficients of $e$, and more importantly, how many of them, combine to form every particular coefficient of $e'$ depends on the polynomial $f_i$. For example, if

\[
f(X) = (X^3 + X + 1)(X^5 + X^2 + 1)
\]

and $e = \sum_{i=0}^{5} e_i X^i$, then,

\[
e' = e \text{ mod } (X^3 + X + 1) = (e_0 + e_3 + e_5) + (e_1 + e_3 + e_4 + e_5)X + (e_2 + e_4 + e_5)X^2,
\]

and thus every coefficient of the error $e'$ is comprised of at least 3 coefficients of the error vector $e$, and thus $\tau' > \frac{1}{2} - \frac{(1-2\tau)^3}{2}$.

In our instantiation of the scheme with a reducible $f(X)$ in Section 14.5, we used the $f(X)$ such that it factors into $f_i$’s that make the operations in CRT form relatively fast, while making sure that the resulting Ring-LPN problem modulo each $f_i$ is still around $2^{80}$-hard.

14.4 Authentication Protocol

In this section we describe our new 2-round authentication protocol and prove its active security under the hardness of the Ring-LPN problem. Detailed implementation details will be given in Section 14.5.

\[\text{If we have } k \text{ elements } e_1, \ldots, e_k \text{ } \overset{\$}{\leftarrow} \text{ Ber}_\tau, \text{ then the element } e' = e_1 + \ldots + e_k \text{ is distributed as Ber}_{\tau'}, \text{ where } \tau' = \frac{1}{2} - \frac{(1-2\tau)^3}{2}.\]
Public parameters: \( R, \pi : \{0,1\}^\lambda \rightarrow R, \tau, \tau' \)
Secret key: \( s, s' \in R \)

Tag \( T \)  
\[ r \overset{\$}{\leftarrow} R^*, e \overset{\$}{\leftarrow} \text{Ber}_\tau \in R \]

\[ z := r \cdot (s \cdot \pi(c) + s') + e (r, z) \]

\[ \text{if } r \notin R^* \text{ reject } \]
\[ e' := z - r \cdot (s \cdot \pi(c) + s') \]
\[ \text{if } \text{wt}(e') > n \cdot \tau' \text{ reject } \]
else accept

Figure 14.1: Two-round authentication protocol with active security from the \text{Ring-LPN}^R\text{assumption.}

14.4.1 The Protocol

Our authentication protocol is defined over the ring \( R = F_2[X]/(f) \) and involves a “suitable” mapping \( \pi : \{0,1\}^\lambda \rightarrow R \). We call \( \pi \) suitable for ring \( R \) if for all \( c, c' \in \{0,1\}^\lambda, \pi(c) - \pi(c') \in R \setminus R^* \) iff \( c = c' \). We will discuss the necessity and existence of such mappings after the proof of Theorem 14.4.1.

- **Public parameters.** The authentication protocol has the following public parameters, where \( \tau, \tau' \) are constants and \( n \) depend on the security parameter \( \lambda \).
  - \( R, n \) ring \( R = F_2[X]/(f) \), \( \deg(f) = n \)
  - \( \pi : \{0,1\}^\lambda \rightarrow R \) mapping
  - \( \tau \in \{0, \ldots, 1/2\} \) parameter of Bernoulli distribution
  - \( \tau' \in \{\tau, \ldots, 1/2\} \) acceptance threshold

- **Key Generation.** Algorithm KG\((1^\lambda)\) samples \( s, s' \overset{\$}{\leftarrow} R \) and returns \( s, s' \) as the secret key.

- **Authentication Protocol.** The Reader \( R \) and the Tag \( T \) share secret value \( s, s' \in R \). To be authenticated by a Reader, the Tag and the Reader execute the authentication protocol from Figure 14.1.

14.4.2 Analysis

For our analysis we define for \( x, y \in [0,1] \) the following constant:

\[ c(x, y) := \left( \frac{x}{y} \right)^x \left( 1 - \frac{x}{y} \right)^{1-x} . \]
We now state that our protocol is secure against active adversaries. Recall that active adversaries can arbitrarily interact with a Tag oracle in the first phase and tries to impersonate the Reader in the 2nd phase.

**Theorem 14.4.1** If ring mapping \( \pi \) is suitable for ring \( R \) and the Ring-LPN\(_R\) problem is \((t, q, \varepsilon)\)-hard then the authentication protocol from Figure 14.1 is \((t', q, \varepsilon')\)-secure against active adversaries, where

\[
t' = t - q \cdot \exp(R) \quad \varepsilon' = \varepsilon + q \cdot 2^{-\lambda} + c(\tau', 1/2)^{-n}
\]

and \( \exp(R) \) is the time to perform \( O(1) \) exponentiations in \( R \). Furthermore, the protocol has completeness error \( \varepsilon_c(\tau, \tau', n) \approx c(\tau', \tau)^{-n} \).

**Proof:**

The completeness error \( \varepsilon_c(\tau, \tau', n) \) is (an upper bound on) the probability that an honestly generated Tag gets rejected. In our protocol this is exactly the case when the error \( e \) has weight \( \geq n \cdot \tau' \), i.e.

\[
\varepsilon_c(\tau, \tau', n) = \Pr[\text{wt}(e) > n \cdot \tau' : e \leftarrow \text{Ber}_\tau^R]
\]

Levieil and Fouque [LF06] show that one can approximate this probability as \( \varepsilon_c \approx c(\tau', \tau)^{-n} \).

To prove the security of the protocol against active attacks we proceed in sequences of games. Game\(_0\) is the security experiment describing an active attack on our scheme by an adversary \( A \) making \( q \) queries and running in time \( t' \), i.e.

- Sample the secret key \( s, s' \leftarrow R \).
- (1st phase of active attack) \( A \) queries the tag \( T \) on \( c \in \{0, 1\}^\lambda \) and receives \((r, z)\) computed as illustrated in Figure 14.1.
- (2nd phase of active attack) \( A \) gets a random challenge \( c^* \leftarrow \{0, 1\}^\lambda \) and outputs \((r, z)\). \( A \) wins if the reader \( R \) accepts, i.e., \( \text{wt}(z - r \cdot (s \cdot \pi(c^*) + s')) \leq n \cdot \tau' \).

By definition we have \( \Pr[A \text{ wins in Game}_0] \leq \varepsilon' \).

Game\(_1\) is as Game\(_0\), except that all the values \((r, z)\) returned by the Tag oracle in the first phase (in return to a query \( c \in \{0, 1\}^\lambda \)) are uniform random elements \((r, z) \in R^2\). We now show that if \( A \) is successful against Game\(_0\), then it will also be successful against Game\(_1\).

**Claim 14.4.2** \(| \Pr[A \text{ wins in Game}_1] - \Pr[A \text{ wins in Game}_0] | \leq \varepsilon + q \cdot 2^{-\lambda} \)

To prove this claim, we construct an adversary \( D \) (distinguisher) against the Ring-LPN problem which runs in time \( t = t' + \exp(R) \) and has advantage

\[
\varepsilon \geq \left| \Pr[A \text{ wins in Game}_1] - \Pr[A \text{ wins in Game}_0] \right| - q \cdot 2^{-\lambda}
\]
In the 1st phase, \( D \) has access to a Ring-LPN oracle \( O \) and has to distinguish between \( O = \Lambda^{R,s}_\pi \) for some secret \( s \in R \) and \( O = U(R \times R) \).

- \( D \) picks a random challenge \( c^* \leftarrow \{0,1\}^\lambda \) and \( a \leftarrow R \). Next, it runs \( A \) and simulates its view with the unknown secret \( s, s' \), where \( s \in R \) comes from the oracle \( O \) and \( s' \) is implicitly defined as \( s' := -\pi(c^*) \cdot s + a \in R \).

- In the 1st phase, \( A \) can make \( q \) (polynomial many) queries to the Tag oracle. On query \( c \in \{0,1\}^\lambda \) to the Tag oracle, \( D \) proceeds as follows. If \( \pi(c) - \pi(c^*) \notin R^* \), then abort. Otherwise, \( D \) queries its oracle \( O() \) to obtain \( (r', z') \in R^2 \). Finally, \( D \) returns \((r, z)\) to \( A \), where

\[
    r := r' \cdot (\pi(c) - \pi(c^*))^{-1}, \quad z := z' + ra. \tag{14.4.2}
\]

- In the 2nd phase, \( D \) uses \( c^* \in \{0,1\}^\lambda \) to challenge \( A \). On answer \((r, z)\), \( D \) returns 0 to the Ring-LPN game if \( \text{wt}(z - r \cdot a) > n \cdot \tau' \) or \( r \notin R^* \), and 1 otherwise. Note that \( s\pi(c^*) + s' = (\pi(c^*) - \pi(c^*))s + a = a \) and hence the above check correctly simulates the output of a reader with the simulated secret \( s, s' \).

Note that the running time of \( D \) is that of \( A \) plus \( O(q) \) exponentiations in \( R \).

Let \( \text{bad} \) be the event that for at least one query \( c \) made by \( A \) to the Tag oracle, we have that \( \pi(c) - \pi(c^*) \notin R^* \). Since \( c^* \) is uniform random in \( R \) and hidden from \( A \)'s view in the first phase we have by the union bound over the \( q \) queries

\[
    \Pr[\text{bad}] \leq q \cdot \Pr_{c^* \leftarrow \{0,1\}^\lambda} [\pi(c) - \pi(c^*) \notin R \setminus R^*] = q \cdot 2^{-\lambda}. \tag{14.4.3}
\]

The latter inequality holds because \( \pi \) is suitable for \( R \).

Let us now assume \( \text{bad} \) does not happen. If \( O = \Lambda^{R,s}_\pi \) is the real oracle (i.e., it returns \((r', z')\) with \( z' = r's + e \)) then by the definition of \((r, z)\) from (14.4.2),

\[
    z = (r's + e) + ra = r(\pi(c) - \pi(c^*))s + e = r(s\pi(e) + s') + e.
\]

Hence the simulation perfectly simulates \( A \)'s view in \( \text{Game}_0 \). If \( O = U(R \times R) \) is the random oracle then \((r, z)\) are uniformly distributed, as in \( \text{Game}_1 \). That concludes the proof of Claim 14.4.2.

We next upper bound the probability that \( A \) can be successful in \( \text{Game}_1 \). This bound will be information theoretic and even holds if \( A \) is computationally unbounded and can make an unbounded number of queries in the 1st phase. To this end we introduce the minimal soundness error, \( \varepsilon_{ms} \), which is an upper bound on the probability that a tag \((r, z)\) chosen independently of the secret key is valid, i.e.

\[
    \varepsilon_{ms}(r', n) := \max_{(c, r) \in R \times R^*} \Pr_{s, s', e \leftarrow R} \left[ \text{wt}(z - r \cdot (s \cdot \pi(c^*) + s')) \leq n \tau' \right]
\]

184
As \( r \in \mathbb{R}^* \) and \( s' \in \mathbb{R} \) is uniform, also \( e' = z - r \cdot (s \cdot \pi(c^*) + s') \) is uniform, thus \( \varepsilon_{ms} \) is simply

\[
\varepsilon_{ms}(\tau', n) := \Pr[e' \leftarrow \mathbb{R} \mid \text{wt}(e') \leq n \tau']
\]

Again, it was shown in [LF06] that this probability can be approximated as

\[
\varepsilon_{ms}(\tau', n) \approx c(\tau', 1/2)^{-n}.
\]  

(14.4.4)

Clearly, \( \varepsilon_{ms} \) is a trivial lower bound on the advantage of \( \mathcal{A} \) in forging a valid tag, by the following claim in Game\(_1\) one cannot do any better than this.

**Claim 14.4.3** \( \text{Pr}[\mathcal{A} \text{ wins in Game}_1] = \varepsilon_{ms}(\tau', n) \)

To see that this claim holds one must just observe that the answers \( \mathcal{A} \) gets in the first phase of the active attack in Game\(_1\) are independent of the secret \( s, s' \). Hence \( \mathcal{A}'s \) advantage is \( \varepsilon_{ms}(\tau', n) \) by definition.

Claims 14.4.2 and 14.4.3 imply (14.4.1) and conclude the proof of Theorem 14.4.1.

We require the mapping \( \pi : \{0, 1\}^\lambda \rightarrow \mathbb{R} \) used in the protocol to be suitable for \( \mathbb{R} \), i.e., for all \( c, c' \in \{0, 1\}^\lambda, \pi(c) - \pi(c') \in \mathbb{R} \setminus \mathbb{R}^* \) iff \( c = c' \). In Section 14.5 we describe efficient suitable maps for any \( \mathbb{R} = \mathbb{F}_2[X]/(f) \) where \( f \) has no factor of degree \( \leq \lambda \). This condition is necessary, as no suitable mapping exists if \( f \) has a factor \( f_i \) of degree \( \leq \lambda \): in this case, by the pigeonhole principle, there exist distinct \( c, c' \in \{0, 1\}^\lambda \) such that \( \pi(c) = \pi(c') \mod f_i \), and thus \( \pi(c) - \pi(c') \in \mathbb{R} \setminus \mathbb{R}^* \).

We stress that for our security proof we need \( \pi \) to be suitable for \( \mathbb{R} \), since otherwise (14.4.3) is no longer guaranteed to hold. It is an interesting question if this is inherent, or if the security of our protocol can be reduced to the Ring-LPN\(_R\) problem for arbitrary rings \( \mathbb{R} = \mathbb{F}_2[X]/(f) \), or even \( \mathbb{R} = \mathbb{F}_q[X]/(f) \) (This is interesting since, if \( f \) has factors of degree \( \ll \lambda \), the protocol could be implemented more efficiently and even become based on the worst-case hardness of lattice problems). Similarly, it is unclear how to prove security of our protocol instantiated with Toeplitz matrices.

**14.5 Implementation**

There are two objectives that we pursue with the implementation of our protocol. First, we will show that the protocol is in fact practical with concrete parameters, even on extremely constrained CPUs. Second, we investigate possible application scenarios where the protocol might have additional advantages. From a practical point of view, we are particularly interested in comparing our protocol to classical symmetric challenge-response schemes employing AES. Possible advantages of the protocol at hand are (i) the security properties and (ii) improved implementation properties. With respect to the former aspect, our protocol has the obvious advantage of being provably secure under a reasonable and static hardness assumption. Even
though AES is arguably the most trusted symmetric cipher, it is “merely” computationally secure with respect to known attacks.

In order to investigate implementation properties, constrained microprocessors are particularly relevant. We chose an 8-bit AVR ATmega163 [Atma] based smartcard, which is widely used in myriads of embedded applications. It can be viewed as a typical representative of a CPU used in tokens that are in need for an authentication protocol, e.g., computational RFID tags or (contactless) smart cards. The main metrics we consider for the implementation are run-time and code size. We note at this point that in many lightweight crypto applications, code size is the most precious resource once the run-time constraints are fulfilled. This is due to the fact that EEPROM or flash memory is often heavily constrained. For instance, the WISP, a computational RFID tag, has only 8 KBytes of program memory [Wik, Ins].

We implemented two variants of the protocol described in Section 14.4. The first variant uses a ring $R = \mathbb{F}_2[X]/(f)$, where $f$ splits into five irreducible polynomials; the second variant uses a field, i.e., $f$ is irreducible. For both implementations, we chose parameters which provide a security level of $\lambda = 80$ bits, i.e., the parameters are chosen such that $\epsilon'$ in (14.4.1) is bounded by $2^{-80}$ and the completeness $\epsilon_c$ is bounded by $2^{-40}$. This security level is appropriate for the lightweight applications which we are targeting.

### 14.5.1 Implementation with a Reducible Polynomial

From an implementation standpoint, the case of reducible polynomial is interesting since one can take advantage of arithmetic based on the Chinese Remainder Theorem.

**Parameters.** To define the ring $R = \mathbb{F}_2[X]/(f)$, we chose the reducible polynomial $f$ to be the product of the $m = 5$ irreducible pentanomials specified by the following powers with nonzero coefficients: $(127, 8, 7, 3, 0)$, $(126, 9, 6, 5, 0)$, $(125, 9, 7, 4, 0)$, $(122, 7, 4, 3, 0)$, $(121, 8, 5, 1, 0)^4$. Hence $f$ is a polynomial of degree $n = 621$. We chose $\tau = 1/6$ and $\tau' = .29$ to obtain minimal soundness error $\epsilon_{ms} \approx c(\tau', 1/2)^{-n} \leq 2^{-82}$ and completeness error $\epsilon_c \leq 2^{-42}$. From the discussion of Section 14.3 the best known attack on Ring-LPN$_R^R$ with the above parameters has complexity $> 2^{80}$. The mapping $\pi : \{0, 1\}^{80} \rightarrow R$ is defined as follows. On input $c \in \{0, 1\}^{80}$, for each $1 \leq i \leq 5$, pad $c \in \{0, 1\}^{80}$ with $\deg(f_i) = 80$ zeros and view the result as coefficients of an element $v_i \in \mathbb{F}_2[X]/(f_i)$. This defines $\pi(c) = (v_1, \ldots, v_5)$ in CRT representation. Note that, for fixed $c, c^* \in \{0, 1\}^{80}$, we have that $\pi(c) - \pi(c^*) \in R \setminus R^*$ iff $c = c^*$ and hence $\pi$ is suitable for $R$.

**Implementation Details.** The main operations are multiplications and additions of polynomials that are represented by 16 bytes. We view the CRT-based multiplication in three stages. In the first stage, the operands are reduced modulo each of the five irreducible polynomials. This part has a low computational complexity. Note that only the error $e$ has to be chosen in the ring and afterwards transformed to CRT representation. It is possible to save the secret key $(s, s')$ and to generate $r$ directly in the CRT representation. This is not possible for $e$ because $e$

---

$^4$ (127, 8, 7, 3, 0) refers to the polynomial $X^{127} + X^8 + X^7 + X^3 + 1$. 

---

186
14.5. Implementation

has to come from $\text{Ber}_R^r$. In the second stage, one multiplication in each of the finite fields defined by the five pentanomials has to be performed. We used the right-to-left comb multiplication algorithm from [HMV03]. For the multiplication with $\pi(c)$ we exploit the fact that only the first 80 coefficients can be non-zero. Hence we wrote one function for normal multiplication and one for sparse multiplication. The latter is more than twice as fast as the former. The subsequent reduction takes care of the special properties of the pentanomials, thus code reuse is not possible for the different fields. The third stage, constructing the product polynomial in the ring, is shifted to the prover (RFID reader) which normally has more computational power than the tag $T$. Hence the response $(r, z)$ is sent in CRT form to the reader. If non-volatile storage — in our case we need $2 \cdot 5 \cdot 16 = 160$ bytes — is available we can heavily reduce the response time of the tag. At an arbitrary point in time, choose $e$ and $r$ according to their distribution and precompute $\text{tmp}_1 = r \cdot s$ and $\text{tmp}_2 = r \cdot s' + e$. When a challenge $c$ is received afterwards, tag $T$ only has to compute $z = \text{tmp}_1 \cdot \pi(c) + \text{tmp}_2$. Because $\pi(c)$ is sparse, the tag can use the sparse multiplication and response very quickly. The results of the implementation are shown in Table 14.1 in Section 14.5.3. Note that all multiplication timings given already include the necessary reductions and addition of a value according to Figure 14.1.

14.5.2 Implementation with an Irreducible Polynomial

Parameters. To define the field $F = \mathbb{F}_2[X]/(f)$, we chose the irreducible trinomial $f(X) = X^{532} + X + 1$ of degree $n = 532$. We chose $\tau = 1/8$ and $\tau' = .27$ to obtain minimal soundness error $\varepsilon_{\text{ms}} \approx c(\tau', 1/2)^{-n} \leq 2^{-80}$ and completeness error $\varepsilon_c \approx 2^{-55}$. From the discussion in Section 14.3 the best known attack on Ring-LPN$^F_T$ with the above parameters has complexity $> 2^{80}$. The mapping $\pi : \{0, 1\}^{80} \rightarrow F$ is defined as follows. View $c \in \{0, 1\}^{80}$ as $c = (c_1, \ldots, c_{16})$ where $c_i$ is a number between 1 and 32. Define the coefficients of the polynomial $v = \pi(c) \in F$ as zero except all positions $i$ of the form $i = 16 \cdot (j - 1) + c_j$, for some $j = 1, \ldots, 16$. Hence $\pi(c)$ is sparse, i.e., it has exactly 16 non-zero coefficients. Since $\pi$ is injective and $F$ is a field, the mapping $\pi$ is suitable for $F$.

Implementation Details. The main operation for the protocol is now a 67-byte multiplication. Again we used the right-to-left comb multiplication algorithm from [HMV03] and an optimized reduction algorithm. Like in the reducible case, the tag can do similar precomputations if $2 \cdot 67 = 134$ bytes non-volatile storage are available. Because of the special type of the mapping $v = \pi(c)$, the gain of the sparse multiplication is even larger than in the reducible case. Here we are a factor of 7 faster, making the response time with precomputations faster, although the field is larger. The results are shown in Table 14.2 in Section 14.5.3.

14.5.3 Implementation Results

All results presented in this section consider only the clock cycles of the actual arithmetic functions. The communication overhead and the generation of random bytes is excluded because they occur in every authentication scheme, independent of the underlying cryptographic func-
tions. The time for building \( e \) from \( \text{Ber}_R^R \) out of the random bytes and converting it to CRT form is included in \textit{Overhead}. Table 14.1 and Table 14.2 shows the results for the ring based and field based variant, respectively.

Table 14.1: Results for the ring based variant w/o precomputation

<table>
<thead>
<tr>
<th>Aspect</th>
<th>time</th>
<th>code size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in cycles</td>
<td>in bytes</td>
</tr>
<tr>
<td>Overhead</td>
<td>17,500</td>
<td>264</td>
</tr>
<tr>
<td>Mul</td>
<td>5 \times 13,000</td>
<td>164</td>
</tr>
<tr>
<td>sparse Mul</td>
<td>5 \times 6,000</td>
<td>170</td>
</tr>
<tr>
<td>total</td>
<td>112,500</td>
<td>1356</td>
</tr>
</tbody>
</table>

The overall code size is not the sum of the other values because, as mentioned before, the same multiplication code is used for all \textit{normal} and \textit{sparse} multiplications, respectively, while the reduction code is different for every field (\( \approx 134 \) byte each). The same code for reduction is used independently of the type of the multiplication for the same field. If precomputation is acceptable, the tag can answer the challenge after approximately 30,000 clock cycles, which corresponds to a 15 msec if the CPU is clocked at 2 MHz.

Table 14.2: Results for the field based variant w/o precomputation

<table>
<thead>
<tr>
<th>Aspect</th>
<th>time</th>
<th>code size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in cycles</td>
<td>in bytes</td>
</tr>
<tr>
<td>Overhead</td>
<td>3,000</td>
<td>150</td>
</tr>
<tr>
<td>Mul</td>
<td>150,000</td>
<td>161</td>
</tr>
<tr>
<td>sparse Mul</td>
<td>21,000</td>
<td>148</td>
</tr>
<tr>
<td>total</td>
<td>174,000</td>
<td>459</td>
</tr>
</tbody>
</table>

For the field-based protocol, the overall performance is slower due to the large operands used in the multiplication routine. But due to the special mapping \( v = \pi(c) \), here the tag can do a sparse multiplications in only 21,000 clocks cycles. This allows the tag to respond in 10.5 msec at 2 MHz clock rate if non-volatile storage is available.

As mentioned in the introduction, we want to compare our scheme with a conventional challenge-response authentication protocol based on AES. The tag’s main operation in this case is one AES encryption. The implementation in [LLS09] states 8,980 clock cycles for one encryption on a similar platform, but unfortunately no code size is given; [Tik] reports 10121 cycles per encryption and a code size of 4644 bytes.\(^5\) In comparison with these highly optimized AES implementations, our scheme is around eleven times slower when using the ring based variant without precomputations. If non-volatile storage allows precomputations, the ring based variant

---

\(^5\)An internet source [Poe] claims to encrypt in 3126 cycles with code size of 3098 bytes but since this is unpublished material we do not consider it in our comparison.
14.6 Conclusions and Open Problems

Table 14.3: Summary of implementation results

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Time (cycles)</th>
<th>Code size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours: reducible $f$ (§14.5.1)</td>
<td>30,000, 82,500</td>
<td>1,356</td>
</tr>
<tr>
<td>Ours: irreducible $f$ (§14.5.2)</td>
<td>21,000, 174,000</td>
<td>459</td>
</tr>
<tr>
<td>AES-based [LLS09, Tik]</td>
<td>10,121</td>
<td>4,644</td>
</tr>
</tbody>
</table>

is only three times slower than AES. But the code size is by a factor of two to three smaller, making it attractive for Flash constrained devices. The field based variant without precomputations is 17 to 19 times slower than AES, but with precomputations it is only twice as slow as AES, while only consuming one tenths of the code size. From a practical point of view, it is important to note that even our slowest implementation is executed in less than 100 msec if the CPU is clocked at 2 MHz. This response time is sufficient in many application scenarios. (For authentications involving humans, a delay of 1 sec is often considered acceptable.)

The performance drawback compared to AES is not surprising, but it is considerably less dramatic compared to asymmetric schemes like RSA or ECC [GPW04]. But exploiting the special structure of the multiplications in our scheme and using only a small amount of non-volatile data memory provides a response time in the same order of magnitude as AES, while keeping the code size much smaller.

Table 14.3 gives a summary of the results.

14.6 Conclusions and Open Problems

In this chapter we proposed a variant of the HB2 protocol from [KPC+11] which uses an “algebraic” derivation of the session key $K(c)$, thereby allowing to be instantiated over a carefully chosen ring $R = \mathbb{F}_2[X]/(f)$. Our scheme is no longer based on the hardness of LPN, but rather on the hardness of a natural generalization of the problem to rings, which we call Ring-LPN. The general overview of our protocol is quite simple. Given a challenge $c$ from the reader, the tag answers with $(r, z) = (r \cdot K(c) + e) \in R \times R$, where $r$ is a random ring element, $e$ is a low-weight ring element, and $K(c) = sc + s'$ is the session key that depends on the shared secret key $K = (s, s') \in R^2$ and the challenge $c$. The reader accepts if $e' = r \cdot K(c) - z$ is a polynomial of low weight, cf. Figure 14.1 in Section 14.4. Compared to the HB and HB+ protocols, ours has one less round and a dramatically lower communication complexity. Our protocol has essentially the same communication complexity as HB4, but still retains the advantage of one fewer round. And compared to the two-round HB2 protocol, ours again has the large savings in the communication complexity. Furthermore, it inherits from HB2 the simple and tight security proof that, unlike three-round protocols, does not use rewinding.

We remark that while our protocol is provably secure against active attacks, we do not have a proof of security against man-in-the-middle ones. Still, as argued in [KSS10], security against active attacks is sufficient for many use scenarios (see also [JW05, KW05, KW06]). We would like
to mention that despite man-in-the-middle attacks being outside our “security model”, we think that it is still worthwhile investigating whether such attacks do in fact exist, because it presently seems that all previous man-in-the-middle attacks against HB-type schemes along the lines of Gilbert et al. [GRS05] and of Ouafi et al. [OOV08] do not apply to our scheme. In Appendix 14.7, however, we do present a man-in-the-middle attack that works in time approximately $n^{1.5} \cdot 2^{\lambda/2}$ (where $n$ is the dimension of the secret and $\lambda$ is the security parameter) when the adversary can influence on the order of $n^{1.5} \cdot 2^{\lambda/2}$ interactions between the reader and the tag.

To resist this attack, one could simply double the security parameter, but we believe that even for $\lambda = 80$ (and $n > 512$, as it is currently set in our scheme) this attack is already impractical because of the extremely large number of interactions that the adversary will have to observe and modify.

We demonstrated that our protocol is indeed practical by providing a lightweight implementation of the tag part of the protocol. A major advantage of our protocol is its very small code size. The most compact implementation requires only about 460 bytes of code, which is an improvement by factor of about 10 over AES-based authentication. Given that EEPROM or FLASH memory is often one of the most precious resources on constrained devices, our protocol can be attractive in certain situations. The drawback of our protocol over AES on the target platform is an increase in clock cycles for one round of authentication. However, if we have access to a few hundred bytes of non-volatile data memory, our protocol allows precomputations which make the on-line phase only a factor two or three slower than AES. But even without precomputations, the protocol can still be executed in a few 100 msec, which will be sufficient for many real-world applications, e.g., remote keyless entry systems or authentication for financial transactions.

We would like to stress at this point that our protocol is targeting lightweight tags that are equipped with (small) CPUs. For ultra constrained tokens (such as RFIDs in the price range of a few cents targeting the EPC market) which consist nowadays of a small integrated circuit, even compact AES implementations are often considered too costly. (We note that virtually all current commercially available low-end RFIDs do not have any crypto implemented.) However, tokens which use small microcontrollers are far more common, e.g., low-cost smart cards, and they do often require strong authentication. Also, it can be speculated that computational RFIDs such as the WISP [Wik] will become more common in the future, and hence software-friendly authentication methods that are highly efficient such as the protocol provided here will be needed.

A number of open problems remain. Our protocol cannot be proved secure against man-in-the-middle attacks. It is possible to apply the techniques from [KPC+11] to secure it against such attacks, but the resulting protocol would lose its practical appeal in terms of code size and performance. Finding a truly practical authentication protocol, provably secure against man-in-the-middle attacks from the Ring-LPN assumption (or something comparable) remains a challenging open problem.

We believe that the Ring-LPN assumption is very natural and will find further cryptographic applications, especially for constructions of schemes for low-cost devices. In particular, we think
that if the HB line of research is to lead to a practical protocol in the future, then the security of this protocol will be based on a hardness assumption with some “extra algebraic structure”, such as Ring-LPN in this work, or LPN with Toeplitz matrices in the work of Gilbert et al. [GRS08a]. More research, however, needs to be done on understanding these problems and their computational complexity. In terms of Ring-LPN, it would be particularly interesting to find out whether there exists an equivalence between the decision and the search versions of the problem similar to the reductions that exist for LPN [BFKL93, Reg09, KS06a] and Ring-LWE [LPR10].

14.7 Man-in-the-Middle Attack

In this section, we sketch a man-in-the-middle attack against the protocol in Figure 14.1 that recovers the secret key in time approximately $O\left(n^{1.5} \cdot 2^{\lambda/2}\right)$ when the adversary is able to insert himself into that many valid interactions between the reader and the tag. For a ring $R = \mathbb{F}_2[X]/(f)$ and a polynomial $g \in R$, define the vector $\vec{g}$ to be a vector of dimension $\deg(f)$ whose $i$th coordinate is the $X^i$ coefficient of $g$. Similarly, for a polynomial $h \in R$, let $\Rot(h)$ be a $\deg(f) \times \deg(f)$ matrix whose $i$th column (for $0 \leq i < \deg(f)$) is $h \cdot X^i$, or in other words, the coefficients of the polynomial $h \cdot X^i$ in the ring $R$. From this description, one can check that for two polynomials $g, h \in R$, the product $\vec{g} \cdot \vec{h} = \Rot(g) \cdot \vec{h}$ mod 2 $= \Rot(h) \cdot \vec{g}$ mod 2.

We now move on to describing the attack. The $i$th (successful) interaction between a reader $\mathcal{R}$ and a tag $\mathcal{T}$ consists of the reader sending the challenge $c_i$, and the tag replying with the pair $(r_i, z_i)$ where $z_i - r_i \cdot (s \cdot \pi(c_i) + s')$ is a low-weight polynomial of weight at most $n \cdot \tau'$. The adversary who is observing this interaction will forward the challenge $c_i$ untouched to the tag, but reply to the reader with the ordered pair $(r_i, z_i')$ where $e_i$ is a vector that is strategically chosen with the hope that the vector $z_i' - r_i \cdot (s \cdot \pi(c_i) + s')$ is exactly of weight $n \cdot \tau'$. It’s not hard to see that it’s possible to choose such a vector $e_i$ so that the probability of $z_i' - r_i \cdot (s \cdot \pi(c_i) + s')$ being of weight $n \cdot \tau'$ is approximately $1/\sqrt{n}$. The response $(r_i, z_i')$ will still be valid, and so the reader will accept. By the birthday bound, after approximately $2^{\lambda/2}$ interactions, there will be a challenge $c_i$ that is equal to some previous challenge $c_i$. In this case, the adversary replies to the reader with $(r_i, z_i'')$, where the polynomial $z_i''$ is the polynomial $z_i'$ whose first bit (i.e., the constant coefficient) is flipped. What the adversary is hoping for is that the reader accepted the response $(r_i, z_i')$ but rejects $(r_i, z_i'')$. Notice that the only way this can happen is if the first bit of $z_i'$ is equal to the first bit of $r_i \cdot (s \cdot \pi(c_i) + s')$, and thus flipping it, increases the error by 1 and makes the reader reject. We now explain how finding such a pair of responses can be used to recover the secret key.

Since the polynomial expression $z_i' - r_i \cdot (s \cdot \pi(c_i) + s') = z_i' - r_i \cdot \pi(c_i) \cdot s - r_i \cdot s'$ can be written as matrix-vector multiplications as

$$\vec{z}_i' = Rot(r_i \cdot \pi(c_i)) \cdot \vec{s} - Rot(r_i) \cdot \vec{s}' \mod 2,$$
if we let the first bit of \( \vec{z}'_i \) be \( \beta_i \), the first row of \( \text{Rot}(r_i \cdot \pi(c_i)) \) be \( \vec{a}_i \) and the first row of \( \text{Rot}(r_i) \) be \( \vec{b}_i \), then we obtain the linear equation

\[
\langle \vec{a}_i, \vec{s} \rangle + \langle \vec{b}_i, \vec{s}' \rangle = \beta_i.
\]

To recover the entire secret \( s, s' \), the adversary needs to repeat the above attack until he obtains \( 2n \) linearly-independent equations (which can be done with \( O(n) \) successful attacks), and then use Gaussian elimination to recover the full secret.
Part IV

Conclusion
Chapter 15
Conclusion and Future Work

During the course of this thesis, we have shown how to efficiently implement a wide range of alternative cryptosystems. In the following, the main contributions are summarised and some points for future work are presented.

15.1 Conclusion

Throughout this thesis, we dedicated our research to the analysis, evaluation, evolution and implementation of practical post-quantum cryptography and especially the field of code-based cryptography. The obtained results provide strong evidence that some of the alternative cryptosystems have already evolved into full-fledged replacements for classical schemes.

Finite Field Implementation  As we showed in Section 4, the underlying field operations provide the basis for all alternative public key schemes in use. In practice, the fastest implementations use full table lookups to compute finite field operations. With increasing extension degree, the size of these tables becomes impracticable, as they grow exponentially. When lookup tables become infeasible, most implementations choose polynomial arithmetic, minimizing memory consumption at the cost of a highly increased computation time. The third possibility, using tower field arithmetic, is not available for the typical extension field degrees, e.g., $2^{11}$ or $2^{13}$.

With our proposed new implementation called partial tables (cf. Section 4.2), we add the ability to fine-tune the time-memory trade-off and thus choose the best possible implementation for specific target scenarios, which none of the previously existing implementations offered.

Code-Based Schemes on Microcontrollers  The second and main contribution of the thesis is related to the two specific code-based schemes McEliece and Niederreiter and their implementation on microcontrollers (cf. Chapter 10). While both schemes are based on the same structural elements, they excel in different use cases. Nevertheless, the previously existing implementations focused on different, single specific parameter set, e.g., the decoder, underlying code, and CCA2 secure conversion, making a direct comparison between the two schemes impossible. For
the first time, we presented an in-depth comparison with respect to these implementational properties in a wide range of security levels. During this analysis, we applied the Patterson and Berlekamp decoding algorithms. Even though Patterson's decoding algorithm is much more complex than Berlekamp, we showed that it is faster for small embedded microcontrollers. This is due to the preceding syndrome computation: in case of the Berlekamp-Massey algorithm, the computed syndrome is twice as large as the syndrome used by the Patterson algorithm and leads to a significantly higher runtime. Evaluating different root searching algorithms, we showed that - in contrast to normal PCs - the Horner scheme is not only faster than Chien search but also faster than the Berlekamp-Trace Algorithm (BTA), which has the lowest theoretical complexity. BTA suffers from a huge overhead due to the use of recursion and large polynomials, while Chien search is only efficient when parallelized.

The last part of this evaluation focuses on the two conversions to achieve CCA2 security: Kobara-Imai-γ and Fujisaki-Okamoto. The results reveal that the Kobara-Imai-γ conversion is faster by a factor of up to 2.8 during encryption and ~1.2 during decryption than the Fujisaki-Okamoto conversion and that the impact of constant weight encoding is negligible. Additionally, Kobara-Imai-γ is applicable to McEliece and Niederreiter, where Fujisaki-Okamoto only applies to the former.

Aside of the detailed evaluation, this work also provides the most complete and fastest implementation of binary Goppa code-based schemes for 8 bit microcontroller published to date.

**Different Code Constructs on Microcontrollers** As the use of plain binary Goppa codes provides the best security but also implies large key sizes, we evaluated different code constructs as a replacement to reduce these disadvantageous side-effects. Quasi-dyadic binary Goppa and quasi-cyclic MDPC codes provide much smaller key sizes and - as of today - the same security level. Implemented on microcontrollers, QD codes drastically reduce the key size (and thus the code size) with a slightly decreased performance. QC-MDPC codes push this trade-off to the limit, leading to extremely small implementations usable in highly restricted environments. This possibility comes with a price: a runtime performance close to the bounds of acceptability when involving human interaction.

**Code-Based Schemes on FPGAs** On FPGAs, we focused on the Niederreiter scheme. As on microcontrollers, we evaluated the impact of different decoders, and achieved the opposite result: We showed the advantage of the Berlekamp-Massey algorithm, which requires only 80 percent of the runtime and half of the resources compared to the implementation of the Patterson decoder. With 1.5 million encryptions and 17,000 decryption operations, respectively, we outperform all other published implementations of Goppa code-based schemes as well as the classical ECC and RSA schemes on comparable platforms.

**Different Code Constructs on FPGAs** Targeting the aspect of large key sizes, we also implemented MDPC codes on FPGAs (cf. Section 12.4). This greatly decreases the amount of
required memory and FPGA resources, which was the main drawback of previous implementations. In contrast to the microcontroller implementation, the achieved performance is highly competitive.

**Optimization of MDPC Decoders** From an algorithmic point of view, we improved the performance and error-correction capability of the known MDPC decoders. By keeping track of the syndrome changes and an early success detection method, our suggested decoders improve the performance by a factor of up to four while increasing the number of correctable errors slightly.

**Multivariate Quadratics Public Key Scheme** In the third part of the thesis, we evaluated three different MQPKS - UOV, Rainbow and enTTS - for the most common security levels in embedded systems: 64, 80 and 128 bits symmetric security. By optimizing existing constructions and including new optimizations, we are able to outperform ECC by a factor of two to ten. Compared to RSA, we are able to sign 25 times faster and verify at the same speed, even when RSA uses a short exponent.

**Lattice-based Schemes** Finally, the presented authentication scheme LaPin - which is based on the Ring-LPN problem - provides a provable secure scheme that has a much smaller code size than AES, while providing a performance in the same order of magnitude. This is achieved by exploiting the special structure of the multiplications in our scheme and using only a small amount of non-volatile data memory.

**15.2 Future Work**

Despite being more than 30 years old, code-based cryptography still has several remaining open research problems. While this thesis already addressed implementational aspects, the theoretical foundation needs improvements to serve as a solid base for future security challenges. With the exception of binary Goppa codes, no construction was subject of an in-depth security analysis. Especially the newer constructions, addressing the large key size issue, must be evaluated with respect to generic and structural attacks.

To further enhance the security, upcoming research should focus on better conversions to achieve different notions of indistinguishability, e.g., IND-CPA, IND-CCA, IND-CCA2. As of now, very few conversions are available, which are tailored to the distinct properties of the McEliece and Niederreiter algorithms. Such special conversions should offer a low data overhead, handle constant-weight encoding, and should not require encryption during decryption.

Besides encryption, digital signatures are necessary to complete the advanced properties of public-key cryptography. The few proposed code-based signature schemes share a major disadvantage: They are computationally expensive and building implementations able to compete with classical schemes is a very challenging task. Thus, this research area stays of great interest for both theoretical and implementational improvements.
Regarding essential implementational requirements, side channel resistance plays an crucial role. As the underlying arithmetic is different to block ciphers and classical public-key schemes like RSA or ECC, we need to develop new methods to reach the protection level of those schemes: There are no S-Boxes, scalar multiplication or simple exponentiation, for which efficient protection methods are already known. The future challenges are not only to analyse the vulnerabilities to side channel attacks, but also to find and evaluate possible techniques to harden code-based implementations against them.

Despite these open research questions, code-based cryptography matured over the last years to a state, where the first standardization proposals, e.g., for McEliece using binary Goppa codes, are reasonable. This will serve as a further incentive to analyze and ultimately use these promising schemes in real-life applications.

Comparing the practicability of Multivariate Quadratics Public Key Schemes with code-based schemes, MQPKS offers fast signature algorithms but lacks efficient encryption. Here, research should focus on building new encryption primitives to offer the full abilities of public-key cryptography. During this process, side channel countermeasures and security evaluations must remain in focus.

The field of lattice-based schemes like LaPin is in a very early state of development. In contrast to code-base cryptography, there are no established schemes yet and the whole area is in rapid movement. As physical attacks have come to the attention of mathematicians during the last years, many of the new protocols already take side channel aspects into account: The inherit randomization in the LaPin protocol for example allows the addition of a side channel protection layer at low costs. This work is already in progress.
Part V

The Appendix
Chapter 16

Appendix

16.1 Listings

16.1.1 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 \(\mathbb{F}_{p^m}\).

Listing 16.1: Listing primitive polynomials using SAGE

```python
1 p=2; m=1; mmmax=32;
2 while m <= mmmax:
3     F.<z> = FiniteField(p^m)
4     print “GF(%d^%d) “ % (p,m),
5     print F.polynomial(),
6     m+=1
```

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

Listing 16.2: Primitive polynomial definition for field construction

```python
1 #if GF_m == 11
2 #define PRIM_POLY 2053
3 #endif
```

16.1.2 Computing a normal basis of a Finite Field using SAGE

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

Listing 16.3: Computing a normal basis using SAGE

```python
1 def normalbase(M):
2     F.<a> = GF(2^M)
3     for e in range(0, 2^M):
```

\(^1\)\url{http://www.sagemath.org/}
Chapter 16. Appendix

4       \texttt{z=a^e}
5       \texttt{basis = list();}
6       \texttt{for i in range(M):}
7       \quad \texttt{s = bin(eval((z^(2^i)).\texttt{repr})}[2:]
8       \quad \texttt{basis.append(list(map(int, list('0'*(M-len(s))+s)))))}
9
10      \texttt{x = span(basis,GF(2)).}\texttt{matrix()}
11     \texttt{if x.nrows() == M and x.determinant() != 0:}
12     \quad \texttt{return e}
13
14
15     \texttt{for m in range(2,16):}
16     \quad \texttt{F.<a> = GF(2^m)}
17     \quad \texttt{print "GF(\%d\%d)" \% (2,m),}
18     \quad \texttt{basis = normalbase(m)}
19     \quad \texttt{print a^basis, "=" , (a^basis).}\texttt{int\_repr()}
20     \quad \texttt{print}

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

Listing 16.4: Normal basis definition for \textit{GF}(2^{11})

1     \#if GF\_m == 11
2     \#define NORMAL\_BASE 512
3     \#endif

16.2 Definitions

16.2.1 Hamming weight and Hamming distance

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

16.2.2 Minimum distance of a codeword

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

Definitions in this section stem from Pointcheval [Poi00] and apply to polynomial time adversaries, i.e., an adversary $A$ using an algorithm with a running time bounded in the algorithm’s input size $n$ by $O(n^k)$ 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 $P(f(A(f(x))) = f(x))$ 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 $f(x_1, x_2) = y$.

16.2.4 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

16.2.5 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 brute force search through the entire key space is the fastest attack possible.
Listing 16.5: Constant weight encoding: Generation of lookup table

```python
def bestU(N,T):
    tbits=int(math.ceil(math.log(t,2)))  # number of bits of the binary representation
    T_MASK_LSB = (1<<tbits)-1  # mask for selecting only the r least
    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
```

204
Bibliography


Bibliography

[LB88b] Pil Joong Lee and Ernest F. Brickell. An observation on the security of McEliece’s
public-key cryptosystem. In C. G. Günther, editor, EUROCRYPT’88, volume 330

[LDW06] Yuan Xing Li, R. H. Deng, and Xin Mei Wang. On the equivalence of McEliece’s

[Lee07] Kwankyu Lee. Interpolation-based Decoding of Alternant Codes. CoRR, abs-

[Leo88] Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large

Prisco and Moti Yung, editors, SCN 06, volume 4116 of LNCS, pages 348–359.

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

[LLS09] Hyubgun Lee, Kyounghwa Lee, and Yongtae Shin. AES implementation and per-


[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning
with errors over rings. In Henri Gilbert, editor, EUROCRYPT 2010, volume


[Lyu05] Vadim Lyubashevsky. The parity problem in the presence of noise, decoding random
linear codes, and the subset sum problem. In APPROX-RANDOM, pages

Bibliography


Bibliography


List of Figures

3.1 Growth market of embedded systems [IDC, 2012] ........................................ 15
3.2 XMEGA block diagram [Atmb] ................................................................. 16
3.3 4-Input LUT with FF [Inca] ..................................................................... 17
3.4 Simplified Overview over an FPGA [Incb] .............................................. 17

4.1 Evaluation of the Partial Lookup Table Arithmetic ............................... 26

5.1 States of a single qubit ............................................................................. 30
5.2 Oracle function to evaluate keys ............................................................. 33
5.3 Oracle function to find collision of a hash function ............................... 34
5.4 Inversion about the mean (based on [Gro97]) ........................................... 37
5.5 Quantum Fourier Transform: Complex Plane for $F^{\otimes 4} | 3 \rangle_4$ ........ 38
5.6 Sketch of probability distribution in the first register ............................ 40

6.1 Hierarchy of code classes .......................................................... .......................... 54

8.1 A power consumption trace for different instructions ....................... 88
8.2 Power consumption traces for different operands of (a) XOR, (b) LOAD, and SAVE instructions (all traces in gray and the averaged based on HWs in black) .............................. 88
8.3 Success rate of HW detection using the leakage of a SAVE instruction for different averaging and windowing parameters. ................................. 89
8.4 Power traces of ciphertext (left) 0x0...01 and (right) 0x0...02. ............. 90
8.5 Correlation vectors for ciphertexts (left) 0x0...01 and (right) 0x0...02. .... 90

10.1 Block diagram of the encryption process .............................................. 118
10.2 Block diagram outlining the circuit of the Chien search ...................... 120
10.3 Block diagram of the decryption process .............................................. 121

13.1 $\mathcal{MQ}$-Scheme in general ................................................................. 161
13.2 Central map $\mathcal{F}$ of UOV. White parts denote zero entries while grey parts denote arbitrary entries. ......................................................... 162
13.3 Central map of Rainbow $(q, v_1, o_1, o_2)$. White parts denote zero entries while gray parts denote arbitrary entries. ................................. 162
13.4 Secret map $\mathcal{F}$ of odd sequence Enhanced TTS generalized ............. 164
13.5 $0/1$ UOV Key Generation. For details see [PTBW11] ......................... 173
14.1 Two-round authentication protocol with active security from the Ring-LPN\textsuperscript{R} assumption. .......................... 182
List of Tables

2.1 Implementation Characteristics of PQC Schemes ........................................ 13
4.1 Summary of Memory for Different Methods over GF(2^9) up to GF(2^{15}) .......... 27
7.1 Parameters sets for typical security levels according to [BLP08b] .................. 72
7.2 Comparison of the modern and Classical version of McEliece ....................... 76
8.1 Runtime of the search algorithm for a sparse Goppa polynomial .................... 94
8.2 Runtime of the search algorithm for a full random Goppa polynomial ............. 94
9.1 Comparison between conversions and their data redundancy .......................... 99
9.2 Length of parameters for Kobara-Imai-γ applied to the Niederreiter scheme ...... 102
10.1 Security parameters for Code-based and conventional public-key cryptosystems according to [BLP08b, Eur12] .......................................................... 108
10.2 Cycle count of root extraction algorithms for MCE128 with different optimizations112
10.3 Cycle count of decoding via Berlekamp-Massey or Patterson ....................... 112
10.4 Comparison of syndrome computation variants for McEliece ....................... 112
10.5 Optimized performance of McEliece and Niederreiter using Patterson decoder, KIC and Horner scheme ................................................................. 114
10.6 Performance of McEliece using the Fujisaki-Okamoto conversion ................... 115
10.7 Comparison of performance of our implementation and comparable implementa-
tions of McEliece, Niederreiter, RSA and ECC ............................................. 116
10.8 Implementation results of Niederreiter encryption with n = 2048, k = 1751, t = 27 after place and route (PAR) ......................................................... 121
10.9 Implementation results of Niederreiter decryption using Patterson decoding with n = 2048, k = 1751, t = 27 after PAR ................................................. 122
10.10 Implementation results of Niederreiter decryption using a Berlekamp-Massey de-
coder with n = 2048, k = 1751, t = 27 after PAR ....................................... 122
10.11 Comparison of our Niederreiter designs with single-core ECC and RSA imple-
mentations for 80 bit security. Note that PAT designates Patterson decoding and BM Berlekamp-Massey decoding, respectively ................................ 123
11.1 Suggested parameters for McEliece variants based on quasi-dyadic Goppa codes over \( F_2 \) ............................................................. 128
11.2 Sizes of tables and values in memory ....................................................... 136
11.3 Performance of the QD-McEliece encryption including KIC-\(\gamma\) on the AVR \(\mu C\) ATxmega256@32 MHz.

11.4 Performance of the QD-McEliece decryption on the AVR \(\mu C\) ATxmega256@32 MHz.

11.5 Resource requirements of QD-McEliece on the AVR \(\mu C\) ATxmega256@32 MHz.

11.6 Comparison of the quasi-dyadic McEliece variant including KIC-\(\gamma\) \((n'=2312, k'=1288, t=64)\) with original McEliece PKC \((n=2048, k=1751, t=27)\), ECC-P160, and RSA-1024.

12.1 Parameters for different security levels for McEliece with QC-MDPC codes given by [MTSB12].

12.2 Evaluation of the performance and error correcting capability of the different decoders for a QC-MDPC code with parameters \(n_0 = 2, n = 9600, r = 4800, w = 90\).

12.3 Implementation results of our QC-MDPC implementations with parameters \(n_0 = 2, n = 9600, r = 4800, w = 90, t = 84\) on a Xilinx Virtex-6 XC6VLX240T FPGA.

12.4 Performance comparison of our QC-MDPC FPGA implementations with other public-key encryption schemes. ¹Occupied slices and BRAMs are only given for encryption and decryption combined. ²Calculated from synthesis results of an over-mapped device, post-PAR results are not given and will most likely be much slower. ³Additionally uses 26 DSP48s. ⁴Additionally uses 17 DSP48s.

12.5 Performance comparison of our QC-MDPC microcontroller implementations with other public-key encryption schemes.

13.1 Minimal 0/1-UOV parameters achieving certain levels of security. Thereby \(g\) is the optimal number of variables to guess in the hybrid approach and \(k\) is the optimal parameter selectable for the Reconciliation attack.

13.2 Minimal Rainbow parameters achieving certain levels of security. Thereby \(g\) is the optimal number of variables to guess for the hybrid approach.

13.3 Minimal odd sequence enTTS parameters achieving certain levels of security. Thereby \(g\) is the optimal number of variables to guess for the hybrid approach.

13.4 Minimal Ram Requirements for LES Solving in Bytes

13.5 Results

13.6 Overview of other implementations on comparable platforms.

14.1 Results for the ring based variant w/o precomputation

14.2 Results for the field based variant w/o precomputation

14.3 Summary of implementation results
List of Abbreviations

APKC alternative public-key crypto system
BCH Bose, Ray-Chaudhur and Hocquenghem
BM Berlekamp-Massey
BTA Berlekamp-Trace Algorithm
BTZ Berlekamp-Trace Algorithm using Zinovievs Algorithms
CCA2-secure see IND-CCA2
CLB Configurable Logic Block
CW Constant Weight
ECC Elliptic Curve Cryptography
EEA Extended Euclidean Algorithm
ELP Error Locator Polynomial
enTTS Enhanced TTS
EVP Error Value Polynomial
FF Flip-Flop
FOC Fujisaki-Okamoto Conversion
FPGA Field Programmable Gate Array
GRS Generalized Reed-Solomon Code
HW Hamming weight
I2C Inter-Integrated Circuit
IND-CCA2 Indistinguishability under Adaptive Chosen Ciphertext Attacks
IND-CCA Indistinguishability under Chosen Ciphertext Attacks
IND-CPA Indistinguishability under Chosen Plaintext Attacks
ISD Information Set Decoding
KIC Kobara-Imai-γ Conversion
Abbreviations

**LFSR** Linear Feedback Shift Register

**LUT** Look Up Table

**MDS** Maximum Distance Separable

**MQPKS** Multivariate Quadratics Public Key Scheme

**NIST** National Institute of Standards and Technology

**OAEP** Optimal Asymmetric Encryption Padding

**PKC** public-key cryptography

**PTOWF** Partially Trapdoor One-Way Function

**Ring-LPN** Ring-Learning-Parity-with-Noise

**SPI** Serial Peripheral Interface

**SSA** Support Splitting Algorithm

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

**USART** Universal Synchronous and Asynchronous Receiver/Transmitter

**UOV** Unbalanced Oil and Vinegar

**VHDL** Very High Speed Integrated Circuit Hardware Description Language
About the Author

Personal Data

- Date of Birth: January 12th, 1977
- Place of Birth: Dresden, Germany

Short Resume*

- 2009 – 2013: PhD student at Ruhr-University Bochum, Germany
- 2004 – 2009: study of IT-Security at Ruhr-University Bochum, Germany
- 1999 – 2003: Communication technician, Deutsche Bundeswehr, Germany
- 1997 – 1999: Practical training as communication technician, Deutsche Bundeswehr, Germany

*As of July 2013.
Publications

Journals


International Conferences & Workshops


- Evaluation of SHA-3 Candidates for 8-bit Embedded Processors Stefan Heyse, Ingo von Maurich, Alexander Wild, Cornel Reuber, Johannes Rave, Thomas Pöppelmann, Christof Paar, Thomas Eisenbarth. 2nd SHA-3 Candidate Conference, August 23-24, 2010, University of California, Santa Barbara, USA.


Invited Talks


■ Post-Quantum Cryptography and Quantum Algorithms: Implementations of Code-based Cryptography Lorentz Center, November 5?9, 2012, Leiden, the Netherlands