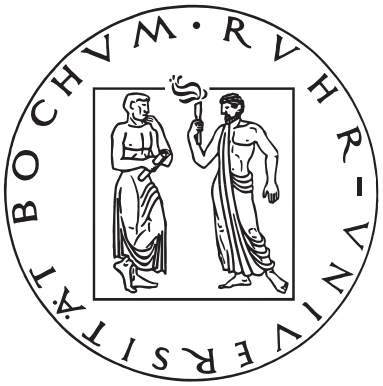
Efficient Implementation of Code- and Hash-Based Cryptography



Dissertation

---

zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

Ingo von Maurich
Bochum, Oktober 2016

To Olya and my loving parents.

Author's contact information:
ingo.vonmaurich@rub.de
https://www.sha.rub.de/group/staff/Ingo_von_Maurich/

Thesis Advisor:	**Prof. Dr.-Ing. Tim Güneysu**
	Universität Bremen & DFKI, Germany
Secondary Referee:	**Prof. Dr.-Ing. Christof Paar**
	Ruhr-Universität Bochum, Germany
Thesis submitted:	October 24, 2016
Thesis defense:	February 03, 2017
Last revision:	February 24, 2017

Abstract

In today's connected world, the majority of secure connections over the Internet are established by public-key cryptography. Common standards for public-key encryption, digital signatures as well as key-agreement and key-exchange protocols provide security services to ensure authentication, confidentiality, integrity and non-repudiation of sensitive data. The security of most public-key standards relies on the hardness of two related problems: the factorization problem in case of RSA-based cryptosystems and the (elliptic curve) discrete logarithm problem in case of DH- and ECC-based cryptosystems. Albeit unlikely, there is no guarantee that cryptanalytic advancements in solving either of the two problems (and thus breaking the current assumptions of wide-spread public-key cryptography) will not be made in the future. In addition, the availability of a scalable quantum computer would invalidate the security assumptions of established public-key cryptosystems currently deployed in the field due to Shor's quantum algorithm which efficiently solves the factorization and discrete logarithm problems. Combined with slow transitioning times to new cryptographic standards, e.g., in the banking industry, this calls for an early investigation of alternative cryptosystems. Acknowledging the current situation, the NSA Central Security Service recently announced preliminary plans to transition its Suite B family of cryptographic algorithms which protects data classified as *secret* and *top secret* to quantum-resistant algorithms and even discourages switching from RSA to ECC in favor of directly moving to quantum-resistant cryptography. Furthermore, the National Institute of Standards and Technology (NIST) initiated standardization efforts for quantum-resistant cryptography.

In this context, novel implementation techniques for alternative cryptosystems from the families of code- and hash-based cryptography for efficient public-key encryption, hybrid encryption, and digital signatures are investigated in this work. We particularly focus on exploring efficient designs tailored for embedded platforms such as microcontrollers and FPGAs and their competitiveness compared to today's RSA and ECC cryptosystems. Quantum-resistant public-key encryption in this work is based on two of the most promising and long-standing alternative cryptosystems originating from coding theory: McEliece and Niederreiter. We instantiate McEliece and Niederreiter with quasi-cyclic moderate density parity-check codes which, compared to binary Goppa codes, require much smaller keys and allow lightweight implementations. We present high-performance and area-efficient FPGA designs which can even outperform current RSA and ECC implementations. Furthermore, first results on side-channel attacks and countermeasures as well as a quantum-resistant IND-CCA-secure hybrid encryption for ARM Cortex-M microcontrollers are provided. Quantum-resistant digital signatures are achieved in this thesis through hash-based signatures by combination of the Merkle signature scheme with Winternitz one-time signatures due to their clear and tight security reductions. We propose novel algorithmic improvements for the authentication path computation and show that side-channel leakage is tightly bounded in our design.

Keywords. Public-Key Encryption, Digital Signatures, Code-Based Cryptography, Hash-Based Cryptography, Quantum-Resistance, Embedded Devices, FPGAs, Microcontrollers

Kurzfassung

Die Vielzahl verschlüsselter Verbindungen im Internet wird mit Hilfe sogenannter Public-Key Kryptographie hergestellt. Weitverbreitete Standards für Public-Key Verschlüsselung, digitale Signaturen sowie Protokolle zur Schlüsselvereinbarung und -verteilung stellen Authentizität, Vertraulichkeit, Integrität und Nicht-Zurückweisbarkeit der Verbindungen sicher. Die Sicherheit der eingesetzten Verfahren lässt sich dabei auf zwei miteinander verwandte Annahmen reduzieren: die Schwierigkeit der Primfaktorzerlegung großer Zahlen bei RSA-basierten Verfahren und dem diskreten Logarithmus-Problem bei DH- und ECC-basierten Verfahren. Wenn auch unwahrscheinlich, so ist es nicht ausgeschlossen, dass keine kryptanalytischen Fortschritte mehr bei der Lösung dieser Probleme erzielt werden und dadurch die Annahmen heute weitverbreiteter Verfahren der Public-Key Kryptographie ihre Gültigkeit verlieren. Die Verfügbarkeit eines skalierbaren Quantencomputers würde die getroffenen Annahmen ebenfalls außer Kraft setzen, da der Shor-Quantenalgorithmus beide Probleme effizient in Polynomialzeit löst. Betrachtet man zudem die langen Übergangszeiten zu neuen kryptographischen Standards, z.B. im Bankensektor, so wird deutlich, dass alternative Public-Key Kryptosysteme frühzeitig untersucht und geeignete Kandidaten identifiziert werden müssen. In Anbetracht dieser Situation hat der NSA Central Security Service kürzlich in einer Pressemitteilung angekündigt die kryptographischen Algorithmen für „Secret“ und „Top Secret“ klassifizierte Daten auf quantenresistente Algorithmen umzustellen und rät, so noch nicht geschehen, sogar davon ab den Wechsel von RSA- auf ECC-basierte Kryptographie vorzunehmen und stattdessen quantenresistente Kryptographie einzusetzen. Des Weiteren initiierte das National Institute of Standards and Technology (NIST) den Standardisierungsprozess für quantenresistente Kryptographie.

In diesem Kontext werden in der vorliegenden Arbeit neuartige Techniken zur effizienten Implementierung alternativer Kryptographieverfahren aus den Familien der codierungs- und hash-basierten Kryptographie untersucht, um Public-Key Verschlüsselung, hybride Verschlüsselung und digitale Signaturen zu realisieren. Insbesondere liegt der Fokus dabei auf maßgeschneiderten Designs für eingebettete Systeme wie FPGAs und Mikrocontroller und deren Konkurrenzfähigkeit im Vergleich zu heutigen RSA und ECC Implementierungen. Quantenresistente Public-Key Verschlüsselung wird in dieser Arbeit auf Basis zweier vielversprechender Verfahren realisiert die der Codierungstheorie entstammen: McEliece und Niederreiter. Beide Verschlüsselungsverfahren werden mit QC-MDPC Codes instanziiert, welche im Vergleich zu binären Goppa Codes kleinere Schlüssel und leichtgewichtige Implementierungen ermöglichen. Wir entwickeln hoch performante und flächeneffiziente FPGA Designs die die heutigen RSA und ECC Implementierungen leistungsmäßig übertreffen können. Zudem werden erste Seitenkanalangriffe und Gegenmaßnahmen ebenso wie IND-CCA-sichere hybride Verschlüsselung für ARM Cortex-M Mikrocontroller präsentiert. Quantenresistente digitale Signaturen werden in dieser Arbeit mithilfe von hash-basierten Signaturen durch Kombination des Merkle Signaturschemas mit Winternitz Einwegsignaturen realisiert. Wir entwickeln neuartige algorithmische Verbesserungen für die Berechnung des Authentifikationspfades und zeigen wie das Design den Verlust von Schlüsselinformationen durch Seitenkanäle begrenzt.

Schlagworte. Public-Key Verschlüsselung, Digitale Signaturen, Codierungs-basierte Kryptographie, Hash-basierte Kryptographie, Quantenresistenz, Eingebettete Systeme, FPGAs, Mikrocontroller

Table of Contents

Chapter 1

Introduction

This chapter motivates the need for alternative public-key cryptography besides the well-known RSA, DH, and ECC schemes followed by a summary of this work's research contributions on quantum-resistant cryptography. We conclude with the outline of the structure of this thesis and briefly introduce the content of each chapter.

Contents

1.1 Motivation

The majority of today's secure connections over the Internet are established by public-key cryptography. Common standards for public-key encryption and digital signatures as well as key-agreement and key-exchange protocols provide security services to ensure authentication, confidentiality, integrity, and non-repudiation of sensitive data. The security of most public-key standards relies on the hardness of two related problems: the factorization problem in case of RSA-based cryptosystems and the (elliptic curve) discrete logarithm problem in case of DH- and ECC-based cryptosystems. Albeit unlikely, there is no guarantee that cryptanalytic advancements in solving either of the two problems (and thus breaking the current assumptions of wide-spread public-key cryptography) will not be made in the future. Furthermore, Bach showed that solving the discrete logarithm problem for a composite modulus is as hard as factoring and solving it modulo primes [Bac84]. Due to the relationship between the integer factorization problem and the discrete logarithm problem, a breakthrough in solving either of the two problems could deteriorate the presumed hardness of both problems which calls for a diversification of hard problems upon which public-key cryptosystems are based.

Data which is protected by today's public-key cryptography is likely being recorded and stored for future analysis, e.g., in NSA's data storage facility in Utah[1]. With an estimated storage capacity in the range of exabytes this raises concerns about the long-term security of today's protected data. Recovery of medical records, diplomatic cables, journalists' whistle-blower sources, client-attorney communications, and many more could still have severe consequences for the involved parties if an at the time secure communication is revealed by improved cryptanalytic methods even after a long period of time, e.g., 10-20 years later.

It is well known that Shor's quantum algorithm efficiently solves the underlying problem of RSA (factoring) and can be adapted to break ECC and DH (discrete logarithms) given a scalable quantum computer capable of operating with many qubits [Sho97]. Although quantum computers can handle only few qubits so far, proof-of-concept implementations of Shor's algorithm were verified several times with 56153 (241×233) being the largest number yet which was factored into its prime factors by a quantum computer with four qubits [XZL+12, DB14]. In this context the NSA Central Security Service recently announced preliminary plans to transition its Suite B family of cryptographic algorithms to quantum-resistant algorithms in the "not too distant future"[2]. NSA's Suite B family of cryptographic algorithms was the first public cryptography standard which specified a set of algorithms to protect data classified as "Secret" or "Top Secret". For symmetric encryption, the Suite B specifies AES in CTR or GCM mode and message digests shall be computed using SHA-256/-384. Public-key cryptography is provided based on elliptic curves, namely ECDSA for digital signatures and ECDH for key agreement. According to the announcement, at least the currently recommended elliptic curve public-key cryptography will be replaced with quantum-resistant schemes. Speculations about the reasoning behind the NSA announcement were not only spurred by conspiracists but also by renown cryptographers, e.g., by Koblitz and Menezes in [KM15]. A possible explanation could be that NSA managed to develop or to acquire knowledge about a scalable quantum computer with sufficiently many qubits powerful enough to weaken the security level of the recommended ECC parameters. Another possibility is that NSA cryptanalysts identified a weakness in the presumed hardness of the elliptic curve discrete logarithm problem with advanced classical cryptanalysis. This scenario seems to be more realistic since also in the academic community progress is made towards solving elliptic curve discrete logarithms more efficiently. Recent results achieved a heuristic quasi-polynomial algorithm for discrete logarithms in finite fields of small characteristic [Jou14, BGJT14].

Although it is well-known that the factorization problem and the discrete logarithm problem can be solved in polynomial time by Shor's quantum computing algorithm, they still are the basis for virtually all public-key cryptosystems used today. Alternative cryptosystems which (a) provide the same security services, (b) have a comparable level of computational efficiency, and (c) have similar costs for storing keys, are urgently required to diversify the public-key primitives used in practice. Among the most promising alternatives to RSA and ECC public-key encryption are the code-based public-key encryption schemes by McEliece [McE78] and Niederreiter [Nie86]. The security of the McEliece and Niederreiter cryptosystems is based on variants of hard problems in coding theory without any known relation to the factorization prob-

[1] `http://www.forbes.com/sites/kashmirhill/2013/07/24/blueprints-of-nsa-data-center-in-utah-suggest-its-storage-capacity-is-less-impressive-than-thought/`, retrieved 11 October 2016.

[2] `https://www.nsa.gov/ia/programs/suiteb_cryptography/`, retrieved 11 October 2016.

lem or the discrete logarithm problem. Having been regarded for a long time as impractical for memory-constrained platforms due to their large key sizes, recent advances showed that reducing the key-sizes to practical levels is possible. McEliece encryption instantiated with *quasi-cyclic moderate density parity-check* (QC-MDPC) codes [Gal63] was introduced in [MTSB13], followed by QC-MDPC Niederreiter encryption in [BBMR14]. Compared to the original proposal of using McEliece and Niederreiter with binary Goppa codes, QC-MDPC codes allow much smaller keys and lightweight implementations. Yet it needs to be investigated if all requirements of constrained platforms can be met with code-based cryptosystems instantiated with QC-MDPC codes combined with improved decoding and implementation techniques to transform the theoretical efficiency into practice. This provides feedback to the research community and allows well-founded comparisons to other alternative cryptosystems. Furthermore, the behavior with regard to side-channel leakage is yet unknown and side-channel countermeasures need to be developed.

Another important branch of public-key cryptography are digital signatures. With the increasing popularity of contactless smart cards and near field communication, digital signatures have become a key component of many embedded system solutions. The applications of digital signatures are numerous, ranging from identification over electronic payments to firmware updates and protection against product counterfeiting. Due to the high computational requirements of today's public-key cryptography, providing efficient digital signatures on embedded microprocessors with and without dedicated co-processors is a challenge. Wide-spread classical digital signature schemes are RSA, e.g., PKCS#1 [RSA12], the digital signature algorithm DSA [NIS13], its elliptic curve equivalent ECDSA [NIS13], and the rather new EdDSA (Edwards-curve Digital Signature Algorithm) [BDL$^+$12]. The underlying problems of these digital signature schemes would however similarly be affected by advanced classical cryptanalysis and by quantum-computing attacks as their public-key encryption counterparts.

A promising candidate for alternative digital signatures is the Merkle Signature Scheme (MSS) scheme based on hash function evaluations [Mer90]. The main idea of MSS is to sign messages with a One-Time Signature Scheme (OTSS) and to authenticate the one-time verification keys using binary hash trees. It was shown in [Hül13] that the security of hash-based signature schemes can be reduced to the collision resistance or even just to the second-preimage resistance of the underlying hash function which arguably is a minimal assumption for digital signature schemes. Furthermore, hash-based signature schemes are usually built upon one-time signatures which inherently provides possibilities for leakage-resilience since the signing keys are ever-changing.

1.2 Research Contributions

The main research contributions of this thesis are the evaluation, implementation and optimization of quantum-resistant public-key encryption, hybrid encryption, and digital signature schemes on embedded devices. The focus for quantum-resistant public-key encryption is on code-based cryptography, in particular McEliece and Niederreiter with QC-MDPC codes targeting efficient designs for constrained embedded systems. We present high-performance and area-efficient FPGA designs which can even outperform current RSA and ECC implementations. Furthermore, first results on side-channel attacks and countermeasures as well as quantum-resistant IND-CCA-secure hybrid encryption for ARM Cortex-M microcontrollers are presented. Quantum-resistant digital signatures are provided through hash-based signatures by combining the Merkle signature scheme (MSS) and Winternitz one-time signatures. The main goals of our work on hash-based signatures are to provide an efficient implementation of MSS with a focus on the challenges when targeting constrained embedded systems, to design the signature scheme such that it offers protection against side-channel attacks, and to quantify and reduce the maximum side-channel leakage of the involved secrets.

The research contributions presented in this thesis were published at peer-reviewed conferences, journals, and books as listed below.

Conferences

- Indocrypt 2012 [vMG12]
- CHES 2013 [HvMG13]
- SAC 2013 [EvMY14]
- DATE 2014 [vMG14a]
- PQCrypto 2014 [vMG14b]
- ACNS 2015 [CEvMS15]
- SAC 2015 [CEvMS16b]
- PQCrypto 2016 [vMHG16]

Journals

- ACM Transactions on Embedded Computing Systems, 2015 [vMOG15]
- IEEE Transactions on Information Forensics and Security, 2016 [CEvMS16a]

Books

- Number Theory and Cryptography, 2013 [EvMPY13].

Furthermore, the author co-authored the following publications as a doctoral student at Ruhr-University Bochum. The topics covered in these publications are outside of the scope of this thesis and are therefore not included.

- CARDIS 2012 [BEE+13]
- ASAP 2013 [SvMG13]
- Journal of Signal Processing Systems, 2014 [SvMGO14]

1.3 Thesis Structure

This thesis is structured into three parts: Part I covers code-based public-key encryption and code-based hash functions. Part II presents our work on hash-based digital signatures. Part III concludes the thesis and provides a summary of the presented results.

Part I: Code-Based Public-Key Encryption and Hash Functions

In the first part of this thesis we present our work on code-based public-key encryption in Chapters 2-7 followed by an implementation of the code-based hash function RFSB in Chapter 8.

Chapter 2: Error-Correcting Codes Error-correcting codes are the foundation of code-based cryptography. This chapter provides necessary mathematical background and the notations which will be used in Part I of the thesis. After reviewing general concepts of error detection and correction with linear block codes, we introduce algebraic codes going from generalized Reed-Solomon codes over Alternant codes to Goppa codes. The chapter is concluded by the introduction of graph-based codes with a particular focus on LDPC and MDPC codes.

Chapter 3: Code-Based Public-Key Encryption Schemes This chapter introduces public-key cryptography and its basic concepts. Code-based public-key encryption is presented starting with the traditional McEliece [McE78] and Niederreiter [Nie86] cryptosystems. We survey optimizations for McEliece and Niederreiter and furthermore show how to instantiate the McEliece and Niederreiter cryptosystems with QC-MDPC codes. We conclude with a security survey of code-based cryptography followed by a summary on parameter selection.

Chapter 4: Efficient Decoding of (QC-)MDPC Codes Decryption in code-based cryptography requires decoding of received words which generally is a time-consuming task. The selection of an efficient decoding algorithm is crucial to the overall decryption performance, hence evaluation and comparison of available options and optimization investigations is essential. First we introduce LDPC and MDPC decoding techniques and evaluate their performance with concrete QC-MDPC McEliece parameters. Novel proposals are made to accelerate decoding and to effectively reduce the probability of decoding failures. We derive and evaluate several decoding variations and compare them among each other to make a justified optimal decoder selection which delivers high performance with least decoding failures.

Chapter 5: QC-MDPC McEliece for Reconfigurable Hardware High-performance and lightweight QC-MDPC McEliece en-/decryption cores are developed in this chapter targeting quantum-resistant public-key encryption in FPGA applications. Our high-performance implementation achieves $13.7\,\mu s/82.1\,\mu s$ for en-/decryption and requires 2,924/10,988 slices on Xilinx Virtex-6. Furthermore, we demonstrate that the cryptosystem can be implemented with a significantly smaller resource footprint – still achieving reasonable performance sufficient for many applications, e.g., challenge-response protocols or hybrid encryption. More precisely, our

lightweight design requires just 68 slices for the encryption unit, around 150 slices for the decryption unit and is able to en-/decrypt an input block in 2.2 ms and 13.4 ms, respectively on Xilinx Spartan-6.

Furthermore, we present horizontal and vertical side-channel analysis techniques for an implementation of the McEliece cryptosystem. Target of this side-channel attack is our lightweight and efficient QC-MDPC McEliece decryption FPGA implementation as presented in Section 5.3. The presented cryptanalysis succeeds to recover the complete private key after a few observed decryptions. It consists of a combination of a differential leakage analysis during the syndrome computation followed by an algebraic step that exploits the relation between the public and private key.

Chapter 6: QC-MDPC McEliece for Embedded Microcontrollers and General-Purpose Processors QC-MDPC McEliece for embedded microcontrollers and general-purpose processors with a focus on ARM's Cortex-M4 and Intel's Haswell architecture is presented in this chapter. Besides practical issues such as random error generation, we demonstrate side-channel attacks on straightforward implementations of QC-MDPC McEliece on embedded microcontrollers. We propose timing- and instruction-invariant coding strategies and countermeasures to strengthen QC-MDPC McEliece against timing attacks as well as simple power analysis attacks. Furthermore, we provide two implementations targeting general-purpose CPUs, a reference C implementation as well as a highly optimized implementation that makes use of vector instructions to achieve maximum performance.

Chapter 7: IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter Although QC-MDPC McEliece is a promising alternative public-key encryption scheme with practical key sizes and good performance on constrained platforms such as embedded microcontrollers and FPGAs, so far none of the QC-MDPC McEliece/Niederreiter implementations provide indistinguishability under chosen plaintext or chosen ciphertext attacks. In this chapter we close this gap by presenting (1) an efficient implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers and (2) the first implementation of Persichetti's IND-CCA hybrid encryption scheme instantiated with QC-MDPC Niederreiter for key encapsulation and AES-CBC/AES-CMAC for data encapsulation. Our implementations achieve practical performance, at 80/128-bit security levels hybrid encryption takes 16.5 ms/83.2 ms, decryption 111 ms/477.5 ms and key-generation 386.4 ms/1511.8 ms.

Chapter 8: Embedded Syndrome-Based Hashing In this chapter we present first implementations of the syndrome-based hash function RFSB-509 on an Atmel ATxmega128A1 microcontroller and a low-cost Xilinx Spartan-6 FPGA. Several trade-offs between size and speed are explored on both platforms and we show that RFSB is extremely versatile with applications ranging from lightweight to high performance. The lightweight microcontroller implementation requires just 732 bytes of ROM while still achieving a competitive performance compared to established hash functions. Our fastest FPGA implementation is based on embedded block memories available in Xilinx Spartan-6 devices and runs at 0.21 cycles/byte, with a throughput of 5.35 Gbit/s. To the best of our knowledge, this is the first time the RFSB hash function is implemented on either of these wide-spread platforms.

Part II: Hash-Based Digital Signatures

The second part of this thesis, Chapters 9-10, presents our work on hash-based digital signatures.

Chapter 9: Hash-Based Digital Signature Schemes We introduce hash-based digital signature schemes based on the Merkle signature scheme in combination with Winternitz one-time signatures. Furthermore, we explain how to efficiently generate one-time signing keys using PRNGs and provide insights into the BDS algorithm for efficient authentication path computation. This chapter concludes with a survey of the existing security arguments for hash-based signature schemes.

Chapter 10: Faster Hash-Based Signatures with Bounded Leakage Digital signatures have become a key component of many embedded system solutions and are facing strong security and efficiency requirements. At the same time side-channel resistance is essential for a signature scheme to be accepted in real-world applications. Based on the Merkle signature scheme and Winternitz one-time signatures we propose a quantum-resistant signature scheme with bounded side-channel leakage. Novel algorithmic improvements for the authentication path computation reduce the average signature computation time by nearly 50 % when compared to state-of-the-art algorithms. Furthermore, our improvements tightly bound side-channel leakage and we state the exact number of times each key is used. The proposed scheme is implemented on two platforms, an Intel Core i7 CPU and an AVR ATxmega microcontroller, with carefully optimized versions for the respective target platform. The theoretical algorithmic improvements are verified in both implementations using cryptographic hardware accelerators to achieve high performance.

Part III: Conclusion

The third part of this thesis concludes on the presented results and identifies future research.

Chapter 11: Conclusion This chapter concludes the thesis and provides a summary of the presented results. The chapter ends with an overview of further interesting research topics for alternative public-key cryptography, in particular for code-based public-key encryption and for hash-based digital signatures.

Part I

Code-Based Public-Key Encryption and Hash Functions

Chapter 2

Error-Correcting Codes

Error-correcting codes are the foundation of code-based cryptography. This chapter provides necessary mathematical background and the notations which will be used in the first part of this thesis. After reviewing general concepts of error detection and correction with linear block codes, we introduce different code representations. We survey the family of algebraic codes due to their historic importance for code-based cryptography going from generalized Reed-Solomon codes over Alternant codes to Goppa codes. This chapter concludes with the introduction of graph-based codes, particularly LDPC and MDPC codes.

Contents

In this chapter we dive into coding theory by introduction of the family of error-correcting binary linear block codes. Apart from ensuring reliable data transmission in everyday applications such as wireless networks, error-correcting codes are the foundation of code-based cryptography and are an essential basis for the first part of this thesis. In the following we provide basic concepts of coding theory, define the mathematical notion used in this thesis, and introduce the families of algebraic and graph-based codes. Further material on the introduction to coding theory can be found in [MS86, HP10].

2.1 Introduction to Coding Theory

Reliable and correct transmission of information over noisy channels is a long-standing problem with many practical applications. In 1948, Shannon formulated the basis for the mathematical theory of communication [Sha48]. Elementary notions such as information sources and

destinations, transmitters and receivers, noise sources and channels, information entropy and redundancy, as well as the word *bit* for a binary digit, 0 or 1, were introduced in his seminal work. The general setting in which a sender transmits a message m over a channel to a receiver is shown in Figure 2.1. In case the channel is noisy, an error e of some form is applied to the message in transit. Throughout this work we will work with additive errors but in general there is no restriction on the form of the error.

A typical channel in the real-world is a laser beam which reads the content of a Compact Disc (CD), Digital Versatile Disc (DVD), or Blu-ray Disc (BD). A typical form of noise in this scenario are scratches and dust particles on the lens and disc.

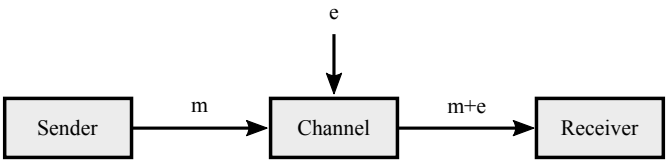


Figure 2.1: A sender transmits some message m over a noisy channel to a receiver. The noise is represented by an error vector e which is added to the message during transmission.

Error-detecting/-correcting codes were primarily developed to enable reliable communication over noisy channels. The general idea to detect or even correct errors introduced by a noisy channel is to add redundancy to the message and transmit it along with the message over the noisy channel. Adding redundancy to a message m with the help of codes is called *encoding*, the resulting output c is called a *codeword*. At the receiver's end, the process of recovering a message from a (noisy) codeword $c + e$ is called *decoding*. It consists of verifying if the received word contains errors, possibly correcting these errors, and extracting message m'. Figure 2.2 illustrates the general setting of en-/decoding messages before and after they are transmitted over a noisy channel.

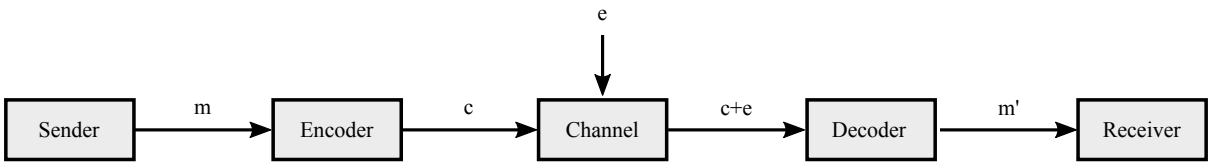


Figure 2.2: Message m is encoded into codeword c before transmitting it over a noisy channel. The channel adds an error vector e to the codeword and the result is fed into the decoder which tries to recover the original message from the noisy codeword.

Important questions here are how to generate meaningful redundancy for specific messages and how to detect and correct errors. For these tasks a multitude of codes and decoding techniques have been developed over time which are generally divided into two main categories: block codes and convolutional codes. We focus on block codes since convolutional codes do not appear to be a good choice for code-based cryptography [LT13].

2.2 Linear Block Codes

Codes which encode fixed-length messages into fixed-length codewords are called block codes in coding theory. Linear block codes are error-correcting codes whose elements are taken from the vector space $\mathbb{F}_q^n$, where $\mathbb{F}_q$ is the finite field with $q = p^m$ elements, with p being a prime number and m being a positive integer.

Definition 2.2.1. ***(Linear Block Code)***
A $[n, k]$-linear block code $\mathcal{C}$ is a linear subspace of $\mathbb{F}_q^n$ with length n and dimension k.

The vectors $c \in \mathcal{C}$ are called codewords of code $\mathcal{C}$. In case $q = 2$, we speak of a binary code. In case $q = 3$, the code is called ternary. Next follows the definition of two important metrics in coding theory, namely the *Hamming weight* and the *Hamming distance*.

Definition 2.2.2. ***(Hamming Weight)***
The number of nonzero positions of a vector $x \in \mathbb{F}_q^n$ is called the Hamming weight $\mathrm{wt}(x)$*. Equally, the Hamming weight can be defined as the Hamming distance of x to the all-zero vector,* $\mathrm{dist}(x, 0)$*.*

Definition 2.2.3. ***(Hamming Distance)***
The number of differing symbols in two vectors $\{x, y\} \in \mathbb{F}_q^n$ is called the Hamming distance $\mathrm{dist}(x, y)$*. Equally, the Hamming distance can be defined as the Hamming weight of the difference of x and y,* $\mathrm{dist}(x, y) = \mathrm{wt}(x - y)$*.*

In order to state an upper bound of how many errors can be detected and corrected by a certain code $\mathcal{C}$ we first define the *minimum distance* of a code.

Definition 2.2.4. ***(Minimum Distance)***
The minimum distance d of a linear block code $\mathcal{C}$ is the minimum Hamming distance of any two distinct codewords of $\mathcal{C}$.

$$d = \min(\mathrm{dist}(c_1, c_2)), \{c_1, c_2\} \in \mathcal{C}, c_1 \neq c_2.$$

Equally, the minimum distance is given by the lowest weight nonzero codeword of $\mathcal{C}$.

$$d = \min(\mathrm{wt}(c)), c \in \mathcal{C}, c \neq 0^n.$$

In the following we refer to a linear block code of length n, dimension k, and minimum distance d by a $[n, k, d]$-code.

Error-Detection and -Correction

A linear code $\mathcal{C}$ with minimum distance d can detect up to $d - 1$ errors since at least d errors have to be added in order to change any codeword of $\mathcal{C}$ into another valid codeword. If the received word is not a codeword, at least one error must have happened during transmission. Hence, detecting up to $d - 1$ errors can be accomplished by checking whether the received word is a codeword of $\mathcal{C}$ or not.

Furthermore, $t = \lfloor \frac{d-1}{2} \rfloor$ errors can be corrected for every linear code with minimum distance d. Since the Hamming distance of every two codewords of $\mathcal{C}$ is at least d, having a codeword with at most t errors added to it still allows to unambiguously find its nearest neighbor by selecting the codeword with the smallest Hamming distance to the received word. This phenomenon can also be explained by imagining spheres of radius t around every codeword. Because of the distance between any two codewords being at least d, all these spheres are non-intersecting. Any codeword with at most t errors lies in exactly one of these spheres and can be directly associated with the codeword in the center of the sphere. Figure 2.3 shows non-intersecting spheres of radius t around three codewords $c_1 \neq c_2 \neq c_3$ of a code $\mathcal{C}$ with minimum distance d. As long as the Hamming weight of the error vectors e_1, e_2, e_3 is at most t, the words $c_1 + e_1, c_2 + e_2, c_3 + e_3$ remain in the sphere of the respective codeword and are hence decodable. This decoding technique is known as minimum distance decoding or maximum likelihood decoding in the literature.

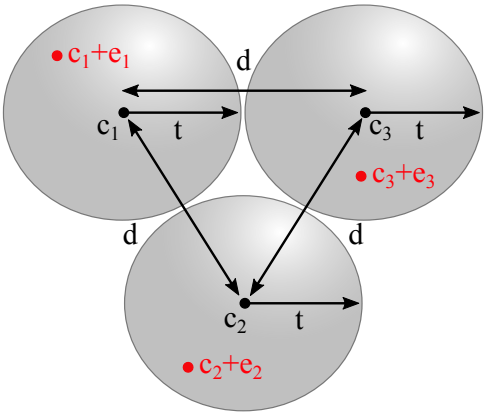


Figure 2.3: Example of a linear code $\mathcal{C}$ with minimum distance d. Non-intersecting spheres of radius $t = \lfloor \frac{d-1}{2} \rfloor$ are drawn around three codewords $c_1 \neq c_2 \neq c_3$ of $\mathcal{C}$. Error vectors e_1, e_2, e_3 of weight at most t are added to c_1, c_2, c_3. The resulting words (red) remain in the sphere of the respective codeword.

Code Representations

A code $\mathcal{C}$ is commonly described in one of two ways, either by a generator matrix or by a parity-check matrix. In general both matrices of a code are not uniquely determined.

Definition 2.2.5. ***(Generator Matrix)***
The rows of a generator matrix $G \in \mathbb{F}_q^{k \times n}$ of a linear $[n, k]$-code $\mathcal{C}$ form a basis of $\mathcal{C}$ such that

$$\mathcal{C} = \{mG \,|\, m \in \mathbb{F}_q^k\}.$$

Hence, the codewords of $\mathcal{C}$ are linear combinations of the rows of the generator matrix.

Definition 2.2.6. ***(Parity-Check Matrix)***
The parity-check matrix $H \in \mathbb{F}_q^{(n-k) \times n}$ of a linear $[n, k]$-code $\mathcal{C}$ is defined as

$$\mathcal{C} = \{Hc^T = 0^{(n-k)} \,|\, c \in \mathbb{F}_q^n\}.$$

Knowledge of either the generator or the parity-check matrix of a code is sufficient since they can be transformed into each other given the relation $HG^T = 0$.

Definition 2.2.7. ***(Syndrome)***
Given a parity-check matrix H, the syndrome $s \in \mathbb{F}_q^{n-k}$ of any vector $x \in \mathbb{F}_q^n$ is defined as

$$s = Hx^T.$$

Hence, multiplying any codeword of code $\mathcal{C}$ with its parity-check matrix H results in the all-zero vector 0^{n-k}. Likewise, the syndrome of any word that is not a codeword of code $\mathcal{C}$ differs from the all-zero vector.

Given the definitions of generator matrices, parity-check matrices, and syndromes, we can now encode messages into codewords and check whether a received word is a codeword. Encoding a message $m \in \mathbb{F}_q^k$ is accomplished by multiplying it with the generator matrix:

$$c = mG.$$

Checking whether a received word is a codeword is done by computing its syndrome and testing it for zero:

$$s = Hx^T \stackrel{?}{=} 0^{(n-k)}.$$

Decoding a received word that is not a codeword, i.e., a word whose syndrome differs from the all-zero vector, on the other hand is much more complex and requires decoding algorithms that depend on the specific codes. More details on decoding are introduced in Chapter 4.

Definition 2.2.8. ***(Systematic Generator Matrix)***
If the generator matrix is given as

$$G = [I_k \,|\, Q],$$

with I_k being the $k \times k$ identity matrix and $Q \in \mathbb{F}_q^{k \times (n-k)}$, then the generator matrix is said to be in systematic form. Note, for every $[n, k]$-code with a generator matrix that is not in systematic form there exists an equivalent $[n, k]$-code with a generator matrix in systematic form.

Having a systematic generator matrix accelerates encoding as the k positions of the message are simply copied to the first k positions of the codeword when computing $c = mG = m \cdot [I_k \,|\, Q]$. Furthermore, the corresponding parity-check matrix to a systematic generator matrix $G = [I_k|Q]$ can be computed as $H = [-Q^T|I_{n-k}]$.

Definition 2.2.9. ***(Cyclic Code)***
A linear block code is cyclic if a circular right shift of each codeword results in another codeword of the same code $\mathcal{C}$.

Hence, if $c = (c_0, \ldots, c_{n-1})$ and by right shift $c' = (c_{n-1}, c_0, \ldots, c_{n-2})$ and it holds $\forall c, c' \in \mathcal{C}$ then the code is said to be cyclic. Given $R = \mathbb{F}_2[x]/(x^n - 1)$ we can also map the codeword $c = (c_0, \ldots, c_{n-1})$ to the polynomial $c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1}$. A circular right shift of the codeword is equal to a multiplication by $x \mod (x^n - 1)$ which results in the polynomial $c_{n-1} + c_0 x + c_1 x^2 + \cdots + c_{n-2} x^{n-1}$.

Definition 2.2.10. ***(Quasi-Cyclic Code)***
A code C is quasi-cyclic (QC) if the code is closed under cyclic right shifts of its codewords by n_0 positions for some positive integer $n_0 > 0$.

Quasi-cyclicity is a generalized form of cyclic codes which allows (fixed) right shifts of more than one position. A quasi-cyclic code is equal to a cyclic code in case $n_0 = 1$. In terms of polynomials, let $c(x)$ be a codeword polynomial of code $\mathcal{C}$, then $c(x)x^{n_0} \mod (x^n - 1)$ is also a codeword polynomial of $\mathcal{C}$ if the code is quasi-cyclic.

2.3 Algebraic Codes

Algebraic codes are introduced mainly due to their historic importance for code-based cryptography and for the sake of completeness of this thesis. In the following chapters we will mostly focus on the family of graph-based codes, their applications in code-based cryptography and how they compete against classical code-based cryptosystems that are usually instantiated with binary Goppa codes which are part of the family of algebraic codes.

2.3.1 Generalized Reed-Solomon Codes

Reed-Solomon codes are a class of cyclic error-correcting block codes which were introduced in 1960 by Reed and Solomon [RS60]. After development of the Berlekamp-Massey decoding algorithm [Ber66, Mas69], Reed-Solomon codes found wide-spread applications in practice, e.g., in the standards for digital video broadcasting (DVB) and digital audio broadcasting (DAB). Reed-Solomon codes were generalized to GRS codes in [vS87].

Definition 2.3.1. ***(Generalized Reed-Solomon Codes)***
Let $0 \leq k \leq n$. Choose distinct elements $\boldsymbol{L} = \{\alpha_1, \ldots, \alpha_n\} \in \mathbb{F}^n$ and non-zero elements $\boldsymbol{v} = \{v_1, \ldots, v_n\} \in \mathbb{F}^n$ from field $\mathbb{F}$. A generalized Reed-Solomon code is defined by

$$GRS_k(\boldsymbol{L}, \boldsymbol{v}) = \{(v_1 f(\alpha_1), \ldots, v_n f(\alpha_n)) \mid f(x) \in \mathbb{F}[x]_k\},$$

where $\mathbb{F}[x]_k$ is the set of polynomials in $\mathbb{F}[x]$ of degree $< k$.

2.3.2 Alternant Codes

The family of alternant codes was defined by Helgert in 1974 [Hel74] and includes the famous Reed-Solomon codes as well as BCH codes [BRC60]. Restricting generalized Reed-Solomon codes from the extension field $\mathbb{F}_{q^m}$ to the subfield $\mathbb{F}_q$ results is the class of alternant codes which are subfield subcodes of generalized Reed-Solomon codes.

Definition 2.3.2. ***(Alternant Codes)***
Alternant codes are defined by restricting GRS codes to the subfield $\mathbb{F}_q$:

$$ALT_{k,q}(\boldsymbol{L}, \boldsymbol{v}) \coloneqq GRS_k(\boldsymbol{L}, \boldsymbol{v}) \cap \mathbb{F}_q^n.$$

2.3.3 Goppa Codes

The relations between algebraic geometry and codes were first discovered by V. D. Goppa who introduced algebraic geometric codes, better known as Goppa codes [Gop70]. We will restrict the description to the binary case in the following since for cryptographic purposes only binary Goppa codes are of interest.

Let m, t be positive integers. A binary Goppa code $\Gamma(g, \boldsymbol{L})$ is defined by its Goppa polynomial $g(z)$ and by its support $\boldsymbol{L} = \{\alpha_1, \ldots, \alpha_n\} \in \mathbb{F}_{2^m}^n$. The n distinct elements of the support $\boldsymbol{L}$ are selected such that $g(a_i) \neq 0, \forall a_i$. The Goppa polynomial $g(z)$ is a monic polynomial of degree t and is defined over the finite field $\mathbb{F}_{2^m}$ as

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

A binary Goppa code over $\mathbb{F}_{2^m}$ is defined as

$$\Gamma(g, \boldsymbol{L}) = \left\{ c \in \mathbb{F}_2^n \mid \sum_{i=0}^{n-1} \frac{c_i}{z - \alpha_i} \equiv 0 \mod g(z) \right\}.$$

2.4 Graph-Based Codes

Two prominent classes of codes in the family of graph-based codes are Low-Density Parity-Check (LDPC) and Moderate-Density Parity-Check (MDPC) codes. Instead of defining fixed structures as done for algebraic codes, LDPC and MDPC codes instead limit the Hamming weight of their parity-check matrices. In this section we introduce and define LDPC and MDPC codes before we explain proposals of using these codes in code-based cryptography in the following chapter. An extensive analysis and optimizations of several decoding techniques for this class of codes are presented in Chapter 4.

2.4.1 Low-Density Parity-Check Codes

Low-density parity-check codes were introduced in [Gal63] but did not attract much interest at first, most likely because they were considered impractical to implement at that time due to their size. Codes based on sparse parity-check matrices reappeared around 35 years later in [MN95, AL96, Mac99], attracting much more interest and serving as base for several follow-up works. Recently, LDPC codes became part of several standardized communication protocols, e.g., the second standard for digital video broadcasting over satellites (DVB-S2), the standard for 10 Gbit Ethernet (10 GbE), and the Wi-Fi standards 802.11n / 802.11ac.

LDPC codes are linear block codes which can either be represented using sparse bipartite graphs or using generator/parity-check matrices as shown for algebraic codes. Similarly as for algebraic codes, we will focus on the binary case throughout this work. For further reading on LDPC codes the author would like to refer to the in-depth descriptions given in [Rya03, Nig04].

Definition 2.4.1. ***(Low-Density Parity-Check Codes)***
A low-density parity-check code is a linear block code whose parity-check matrix is sparse. An LDPC code is regular if its parity-check matrix H consists of w_c ones in each column and $w_r = w_c(n/r)$ ones in each row, with $w_c << r$. Hence, the number of ones in each column and row is constant for regular LDPC codes. An LDPC code is irregular if its parity-check matrix H is sparse but the number of ones in columns or rows is not constant.

LDPC codes are graphically represented using bipartite graphs, commonly referred to as Tanner graphs in the LDPC context. Tanner graphs were introduced in [Tan81] and consist of variable nodes and check nodes. Given a sparse parity-check matrix of an LDPC code, the corresponding Tanner graph is constructed following two basic rules:

(1) The bipartite Tanner graph of an LDPC code consists of n variable nodes v_j for each code bit and $n-k$ check nodes c_i for each parity-check equation.

(2) Check node c_i is connected to variable node v_j iff $H_{i,j} \neq 0,\ (1 \leq i \leq n-k, 1 \leq j \leq n)$.

Furthermore, recall that by definition $Hc^T = 0$. Hence, all variable nodes v_j connected to a check node c_i have to sum up to zero.

Example: Given a $[7, 3]$ binary LDPC code with parity-check matrix

$$H_{[7,3]} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix},$$

we can construct a Tanner graph as shown in Figure 2.4. From the first column of $H_{[7,3]}$ we can derive that all check nodes c_1, c_2, c_3 are connected to v_1. Vice versa, the first row of $H_{[7,3]}$ tells us that c_1 is connected to variable nodes v_1, v_2, v_3, and so on. Using the Tanner graph we can determine that this LDPC code is irregular since its variable nodes do not have a constant number of edges connecting them to check nodes. This fact can also be seen using the matrix representation. Although this code has a constant row weight $w_r = 3$, its column weight w_c is not constant. The first column of $H_{[7,3]}$ has weight three, while the others have weight one.

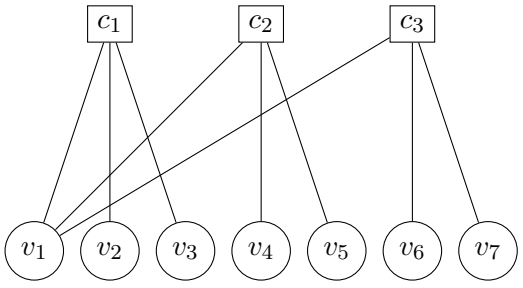


Figure 2.4: The Tanner graph of a $[7, 3]$ binary linear code.

Given a $[10, 5]$ binary LDPC code with parity-check matrix

$$H_{[10,5]} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

we can construct the codes' Tanner graph as shown in Figure 2.5. Compared to the previous example, this LDPC code is regular since its check nodes and its variable nodes both have a constant number of edges connecting them to each other (four edges for each check node and two edges for each variable node). In the matrix representation, the row and column weights are constant ($w_r = 4, w_c = 2$) and fulfill $w_r = w_c(n/r) = 2w_c$.

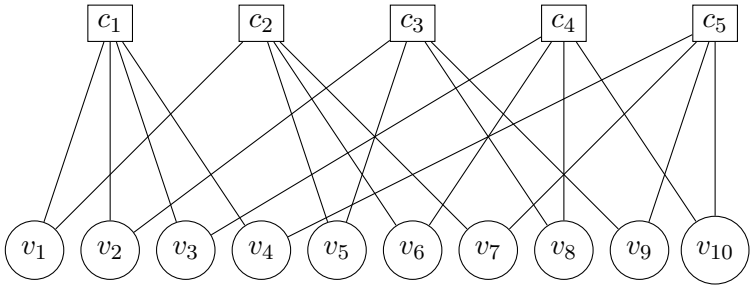


Figure 2.5: The Tanner graph of a $[10, 5]$ binary linear code.

2.4.2 Moderate-Density Parity-Check Codes

The term moderate-density parity-check code was coined in [OB09]. First applications of MDPC codes in public-key cryptography were presented a few years later in [MTSB12, MTSB13]. MDPC codes belong to the family of binary linear $[n, k]$ error-correcting codes, where n is the length, k the dimension, and $r = n - k$ the co-dimension of a code $\mathcal{C}$. Binary linear error-correcting codes are equivalently described either by their generator G or by their parity-check matrix H. The rows of generator matrix $G \in \mathbb{F}_2^{k \times n}$ form a basis of $\mathcal{C}$ while $H \in \mathbb{F}_2^{r \times n}$ describes the code as the kernel $\mathcal{C} = \{c \in \mathbb{F}_2^n \mid Hc^T = 0^\perp\}$ where $0^\perp$ represents an all-zero column vector. The syndrome of any vector $c \in \mathbb{F}_2^n$ is defined as $s = Hc^T \in \mathbb{F}_2^r$. Hence, the code $\mathcal{C}$ is comprised of all vectors $x \in \mathbb{F}_2^n$ whose syndrome is zero for a particular parity-check matrix H.

Similarly to LDPC codes, MDPC codes limit the weight of the parity-check matrix. MDPC codes are defined by only allowing a *moderate* Hamming weight $w = \mathrm{O}(\sqrt{n \log(n)})$ for each row of the parity-check matrix. The row Hamming weight is typically higher than in the case of LDPC codes but still lower compared to common block codes. By an (n, r, w)-MDPC code we refer to a binary linear $[n, k]$ code with such a constant row weight w.

Recall that a code $\mathcal{C}$ is called quasi-cyclic (QC) if for some positive integer $n_0 > 0$ the code is closed under cyclic shifts of its codewords by n_0 positions (cf. Definition 2.2.9). Furthermore, it is possible to choose the generator and parity-check matrices such that they consist of $p \times p$

circulant blocks if $n = n_0 \cdot p$ for some positive integer p. This allows to completely describe the generator and parity-check matrices by their first row. If an (n, r, w)-MDPC code is quasi-cyclic with $n = n_0 \cdot r$, we refer to it as an (n, r, w)-QC-MDPC code.

As for LDPC codes, MDPC codes can be described by a bipartite Tanner graph with n variable nodes v_j and $n - k$ check nodes c_i. The difference to LDPC codes is visible by an increased number of edges due to a higher row weight in the parity-check matrices of MDPC codes. A detailed description of how to use (QC-)MDPC codes in code-based cryptography is given in Chapter 3 and decoding of (QC-)MDPC codes is investigated in Chapter 4.

Chapter 3

Code-Based Public-Key Encryption Schemes

This chapter introduces public-key cryptography and its basic concepts. Code-based public-key encryption is presented starting with the traditional McEliece [McE78] and Niederreiter [Nie86] cryptosystems. We survey optimizations for McEliece and Niederreiter and furthermore show how to instantiate the McEliece and Niederreiter cryptosystems with QC-MDPC codes. We conclude this chapter with a security survey of code-based cryptography followed by a summary on parameter selection.

Contents

3.1 Introduction to Public-Key Cryptography

The notion of public-key encryption revolutionized cryptography in the 1970's and strongly influenced today's *modern cryptography*. Historically, sensitive information was encrypted using *secret-key* encryption algorithms. However, secret-key encryption schemes share a major drawback: they all require an initial secret channel between two parties to agree on some secret key before being able to communicate confidentially over insecure channels, a typical chicken-and-egg problem. Initial secret channels could be face-to-face meetings in a secure environment or channels provided by trusted third parties, e.g., a trusted courier who transports the secret from one communication partner to the other.

While some early secret-key schemes were used at larger scale, e.g., the military Enigma rotor cipher in World War II, secret-key schemes alone are obviously not practical in times of

Internet commerce and connected devices. Imagine a simple scenario of online shopping: a customer opens the website of a merchant, selects some desired items and proceeds to checkout and payment where he enters sensitive information about his identity as well as payment details. Clearly, such data should not be transmitted in plain to prevent fraudsters from recording and using this information for malicious activities, be it a simple analysis of buying patterns or reusing the payment details for fraudulent transactions. Encryption of sensitive information seems to be the logical solution to allow only the merchant to read and process the payment data. However, using secret-key encryption in such scenarios is non-trivial in practice. The merchant and the customer would have to agree on a secret-key in a secured environment beforehand which is not feasible with millions of customers and merchants around the world.

In the 1970's, long before the rise of the Internet and e-commerce, the secret-key distribution problem was overcome by introduction of public-key cryptography, also known as asymmetric cryptography. Instead of sharing a symmetric secret-key between two communication parties, the main idea of asymmetric cryptographic schemes is to associate two mathematically related keys with each entity. These key-pairs consist of a public-key and a private-key. The public-key is assumed to be known to everyone with some binding to the owning entity. Knowledge of the public-key and its owner allows to encrypt data which can only be decrypted by the corresponding private-key. In contrast to the public-key, the private-key is kept secret by the owner such that only the intended receiver is able to decrypt the data.

Diffie-Hellman Key Exchange

The starting point of public-key cryptography was the introduction of a key-agreement protocol published by Diffie and Hellman [DH76]. This protocol for the first time allowed two parties to agree on a secret-key without requiring an initial exchange of secret information, it merely requires that those two parties communicate over public channels. The Diffie-Hellman protocol is defined as follows: let p be a prime number and g be a generator of a multiplicative cyclic group G in $\mathbb{Z}_p^*$. Alice randomly selects a secret $a \in_R \{1, \ldots, p-1\}$ and computes her public-key $pk_A = g^a \mod p$. Bob randomly selects a secret $b \in_R \{1, \ldots, p-1\}$ and computes his public-key $pk_B = g^b \mod p$. Alice sends pk_A to Bob while Bob sends pk_B to Alice. The exchange of pk_A and pk_B is done via a public channel, hence both public-keys are not only known to Alice and Bob but to everyone who is listening to the public channel. The shared secret is derived by Alice and Bob as follows: Alice computes

$$sk = {pk_B}^a = (g^b)^a = g^{ba} \mod p$$

while Bob computes

$$sk = {pk_A}^b = (g^a)^b = g^{ab} \mod p.$$

Due to the commutativity property of exponentiation mod p, Alice and Bob compute the same secret element $g^{ba} = g^{ab} \mod p$. Any *passive* attacker capable of observing messages sent over the public channel only has knowledge of $g^a \mod p$ and $g^b \mod p$. An attacker would need to compute the discrete logarithm of either of these two values to obtain a or b which would allow to compute $g^{ab} \mod p$ as done by Alice and Bob. Note, instead of directly allowing for public-key encryption, the agreed secret of the Diffie-Hellman protocol is used to derive a secret-key under which sensitive data is encrypted using a symmetric encryption scheme. Furthermore,

the DH protocol in its basic form protects only against passive adversaries. An active adversary can exchange sent messages to insert himself as a man-in-the-middle, making Alice and Bob believe they are talking to each other while in fact their communication is being redirected and re-encrypted by an attacker which allows him to obtain the plain messages.

There are many methods available to compute discrete logarithms, among them are the baby-step giant-step [Coh93], index calculus [Adl79], number field sieve [LL93], Pollard rho [Pol78], and many more methods. However, the runtime of all of these algorithms is exponential in the group size. Solving discrete logarithms efficiently in polynomial time still remains an open problem. While it is unclear whether discrete logarithms are the only way to break the DH protocol, the equivalence of the security of the DH protocol and solving discrete logarithms was shown in [Mau94] under certain conditions. It is generally believed that there are no efficient solutions for computing discrete logarithms for carefully chosen groups and hence to attack the DH protocol. However, due to Shor there exists an efficient algorithm in the world of quantum computers which efficiently solves the discrete logarithm problem in polynomial time [Sho97].

RSA Cryptosystem

The work by Diffie and Hellman was followed by the introduction of RSA, a public-key cryptosystem for data encryption and digital signatures by Rivest, Shamir and Adleman [RSA78]. The RSA public-key encryption scheme works as follows: let p, q be prime numbers and $n = p \cdot q$. Randomly select a public-key $e \in_R \{2, \ldots, \Phi(n) - 1\}$ and compute the private key $d = e^{-1} \mod \Phi(n)$[1]. Encryption of a message $m \in \mathbb{Z}_n$ is done by computing $x = m^e \mod n$, decryption reveals the message as $m = x^d \mod n = (m^e)^d = m \mod n$. The RSA signature scheme is basically the inverse of the encryption scheme. Signing a message m is done by raising it to the private key d giving $y = m^d \mod n$. Everyone who is in possession of the public-key e can now verify signature y by raising it to the power of e and checking whether the message m', which is sent along with the signature, matches the result $m' \stackrel{?}{=} y^e \mod n$.

As shown above, the security of the Diffie-Hellman protocol is based on the hardness of the discrete logarithm problem: with p prime, g a generator of a multiplicative cyclic group G in $\mathbb{Z}_p^*$ and given $x = g^a \mod p$, find a. RSA on the other hand bases its security on the hardness of finding the e-th roots of arbitrary numbers in $\mathbb{Z}_n$. To date, the most efficient attack on RSA is to perform integer factorization: given a composite $n = p \cdot q$, find primes p or q. The fundamental theorem of arithmetic states that every integer greater than 1 is either a unique product of primes or a prime itself. Efficiently finding the prime factors of large composite numbers becomes difficult. Finding the prime factors of a composite number that consists of only two primes, a semiprime number, is to date the hardest instance of the prime factorization problem. Specialized factoring algorithms such as the special number field sieve [Pom96], Euler's factorization method [Ore48], Fermat's factorization method [Leh74], Pollard rho [Pol78] and many more are available. None of these however are able to efficiently compute the prime factors of a semiprime number in polynomial time and thus do not break RSA with properly chosen parameters. On the other hand, as in the case of discrete logarithms, there is no proof available stating that efficient prime factorization algorithms cannot exist. In fact, on quantum

[1] By $\Phi(n)$ we refer to Euler's totient function $\Phi(n) \coloneqq |\{x \in \mathbb{N} \mid 1 \leq x \leq n \wedge \gcd(x, n) = 1\}|$ which counts the number of relatively prime positive integers of n.

computers the Shor algorithm efficiently solves the prime factorization problem in polynomial time [Sho97].

Elliptic Curve Cryptography

In the 1980's, two independent proposals suggested the use of elliptic curves for cryptographic applications [Mil86, Kob87]. Elliptic curves in cryptography are defined over finite fields and are sets of points (x, y). Additive cyclic groups are defined over elliptic curves such that each point is a multiple of a generator point $\mathcal{P}$ of the group. E.g. if sets of points (x, y) fulfill the Weierstraß equation

$$y^2 = x^3 + ax + b$$

with a, b fulfilling the condition $4a^3 + 27b^2 \neq 0$, then they form an elliptic curve without singularities. In addition, ∞ is the point at infinity which acts as the neutral element of the cyclic group.

Cryptosystems based on discrete logarithms in finite fields can be transformed to elliptic curves by replacing exponentiations and multiplications in finite fields by scalar-multiplications and point-additions on elliptic curves. The elliptic curve Diffie-Hellman (ECDH) for example is a transformation of the earlier introduced Diffie-Hellman key exchange. In ECDH, Alice and Bob compute their public keys $\mathcal{Q}_a$ and $\mathcal{Q}_b$ by selecting random integers a and b and multiplying them with the generator point $\mathcal{P}$ of an agreed upon elliptic curve. Alice computes $\mathcal{Q}_a = a\mathcal{P}$ while Bob computes $\mathcal{Q}_b = b\mathcal{P}$. Alice and Bob exchange each other's public keys and compute the shared secret point $(x, y) = a\mathcal{Q}_b = b\mathcal{Q}_a = ab\mathcal{P}$ on the elliptic curve. The shared secret is then typically derived from the x-coordinate of the shared point (x, y), for example by hashing x to derive a symmetric key.

The security of elliptic curve cryptography is based on the hardness of solving the elliptic curve discrete logarithm problem (ECDLP), i.e., given two points $\mathcal{P}$ and $\mathcal{Q}$ on an elliptic curve, find a scalar n such that $n\mathcal{P} = \mathcal{Q}$. To date, solving the elliptic curve discrete logarithm problem with the baby-step giant-step [Coh93] and Pollard rho [Pol78] methods seems much harder compared to computing discrete logarithms in finite fields or solving the factorization problem. A 128-bit security level is reached for ECDH already with 256-bit keys while for RSA and DH key sizes of 3072-bit have to be used according to NIST recommendations [NIS13]. Yet again as in the case of RSA and DH, no proof exists to facilitate the hardness of computing discrete logarithms on elliptic curves. Furthermore, Shor's quantum algorithm can be transformed to also efficiently solve discrete logarithms on elliptic curves in polynomial time [Sho97].

Cryptography from Coding Theory

The first public-key encryption scheme based on algebraic codes was introduced by Robert J. McEliece in 1978 [McE78] and is usually referred to as the McEliece cryptosystem. A variation, the Niederreiter cryptosystem, was later introduced by Harald Niederreiter in [Nie86] using GRS codes instead of Goppa codes. The Niederreiter cryptosystem can be considered as the dual of the McEliece cryptosystem. Both rely on the same idea of having a secret code description and a public code description. Furthermore, McEliece and Niederreiter were shown to provide equivalent security in [LDW94].

While the secret code description allows efficient decoding, the public code description is only useful to generate valid codewords/ciphertexts which can be decoded by the secret code but knowledge of the public code does not allow for efficient decoding. Both cryptosystems rely on variants of hard problems in coding theory, namely the hardness of decoding a random linear code and the indistinguishability of the used code family from random codes in case of McEliece and the syndrome decoding problem which was proven to be NP-complete in [BMv78] in case of Niederreiter. Although the McEliece and Niederreiter cryptosystems have withstood the test of time without being seriously broken, they did not see wide adoption in practice, yet. A major drawback were their key sizes which, with cryptographically secure parameters, are much larger than those of the RSA and ECC cryptosystems at equivalent security levels. Recent advances in code-based cryptography however paved new ways for efficient public-key cryptosystems based on coding theory which combine decent performance with moderate key sizes making code-based cryptosystems serious competitors for RSA and ECC. In this context, we will take a closer look at code-based cryptosystems in the remainder of this chapter.

Outline The McEliece cryptosystem is introduced in Section 3.2, followed by the Niederreiter cryptosystem in Section 3.3. Security arguments of code-based cryptography are outlined in Section 3.4 and parameter selection is summarized in Section 3.5.

3.2 The McEliece Cryptosystem

The central idea of the McEliece cryptosystem is to transform an efficiently solvable instance of decoding a linear block code into another one which appears as a random linear code for which decoding is hard. This is achieved by scrambling and permuting the generator matrix of an efficiently decodable code.

The McEliece cryptosystem encodes a plaintext into a codeword using the generator matrix of a public code selected by the receiver and adds a randomly generated error vector of Hamming weight t to the codeword which can only be removed by the intended receiver who is in possession of the secret code description. In the following we introduce traditional McEliece in Section 3.2.1, show how to optimize McEliece in Section 3.2.2 followed by QC-MDPC McEliece in Section 3.2.3. The content of this chapter follows the notation used in [Hey13, RZ14, Mis14].

3.2.1 Traditional McEliece Encryption

Given a binary $[n, k, d]$-Goppa code $\mathcal{C}$, let G be a $k \times n$ generator matrix of $\mathcal{C}$. Further, let there be an efficient t-error correcting decoding algorithm Ψ_Δ. Such a decoding algorithms is able to decode any codeword of $\mathcal{C}$ in polynomial time which has at most t errors added to it. For binary Goppa codes, Ψ could be the decoding algorithm due to Patterson [Pat75] and Δ would be the Goppa polynomial $g(x)$ and the support $(\alpha_1, \ldots, \alpha_n)$. From such a code the McEliece cryptosystem is constructed as usual for public-key encryption systems by three algorithms for key-generation, encryption, and decryption [McE78].

Key-Generation

Select a random $n \times n$ permutation matrix P and a random non-singular $k \times k$ scrambling matrix S. The public-key G', which is a $k \times n$ generator matrix similarly to G, is computed from G as

$$G' = S \cdot G \cdot P.$$

The private-key is comprised of a scrambling matrix S, a permutation matrix P, and an efficient t-error correcting decoding algorithm Ψ_Δ, resulting in

$$sk = (S, P, \Delta).$$

More commonly, one would compute the inverse S^{-1} and P^{-1} during key generation and store them instead of S and P since only their inverses are required during decryption. Hence, the more common equivalent private-key is comprised of

$$sk = (S^{-1}, P^{-1}, \Delta).$$

Encryption

Given a message $m \in \mathbb{F}_2^k$, generate a random error $e \in_R \mathbb{F}_2^n$ with Hamming weight $\mathrm{wt}(e) \leq t$. The ciphertext $x \in \mathbb{F}_2^n$ of message m is computed as

$$x = m \cdot G' + e.$$

Decryption

Given a ciphertext $x \in \mathbb{F}_2^n$, decryption is done in three steps:

(1) Revert the permutation:
$$x' = x \cdot P^{-1}$$

(2) Decode the (still scrambled) ciphertext:
$$m' = \Psi_\Delta(x')$$

(3) Descramble the message:
$$m = m' \cdot S^{-1}$$

Note: the message is correctly recovered by the t-error correcting decoding algorithm $\Psi_\mathcal{G}$ as long as the Hamming weight of the error is less than or equal to t, even though the error is permuted in the first decoding step to

$$x' = x \cdot P^{-1} = (m \cdot G' + e) \cdot P^{-1} = m \cdot S \cdot G + e \cdot P^{-1}.$$

The important fact is that the Hamming weight of the error does not change by permutation, hence decoder Ψ_Δ is still able to remove $e \cdot P^{-1}$ from x' in the second decoding step. Inverting the linear transformation by multiplying the decoded result $m \cdot S$ with S^{-1} finally recovers the plaintext.

3.2.2 Improved McEliece Encryption

Since the keys of the McEliece cryptosystem are fairly large when using cryptographically secure parameters, efforts were made to investigate optimizations of the key sizes while still maintaining the same security level. As explained in [AF95], the scrambling matrix S of the McEliece cryptosystem does not serve a cryptographic purpose but only ensures that the public-key is not systematic. Since conversions for IND-CCA security are required nevertheless in both cases with and without scrambling matrices [OS09], the scrambling matrix can simply be removed and the public generator matrix can be brought to systematic form, i.e., $G' = [I_k \,|\, Q]$ with I_k being the $k \times k$ identity matrix. Furthermore, the permutation matrix P can be stored implicitly instead of permuting the generator matrix, e.g., by permuting the code support $\mathcal{L}$ when using Goppa codes. Thus, the permutation matrix does not have to be stored as well.

With these optimizations, the private-key size is reduced since the formerly required matrices S and P (or S^{-1} and P^{-1}) are removed. The size of the public-key benefits as well, it is reduced to a $k \times (n-k)$ matrix instead of a $k \times n$ matrix because for systematic matrices the $k \times k$ identity matrix does not have to be stored. The three algorithms for McEliece key-generation, encryption, and decryption are adapted as follows.

Key-Generation

Let G be a $k \times n$ generator matrix of a binary $[n, k, d]$-Goppa code $\mathcal{C}$ with an efficient t-error correcting decoding algorithm Ψ_Δ. Bring G to systematic form G' or equivalently, given a parity-check matrix H of $\mathcal{C}$, bring H to systematic form and compute the systematic generator matrix

$$G' = [I_k \,|\, Q]$$

from it by Gauß-Jordan elimination. The private-key is the efficient decoding algorithm Ψ_Δ, the public-key is the systematic generator matrix G'.

Encryption

Encryption complexity is reduced since message m can simply be copied to the first k positions of ciphertext x (i.e., a multiplication with I_k). The remaining $n-k$ positions are computed as mQ. After sampling an error vector e of Hamming weight $\mathrm{wt}(e) \leq t$, e is added to the concatenation of m and mQ resulting in ciphertext

$$x = (m \,|\, mQ) + e.$$

Decryption

Decryption now simply requires to decode the received ciphertext x, i.e.,

$$m = \Psi_\Delta(x).$$

3.2.3 QC-MDPC McEliece Encryption

Instantiating McEliece with t-error-correcting (QC-)MDPC codes was proposed in [MTSB12, MTSB13], mainly to significantly reduce the size of the keys while still maintaining reasonable security arguments. The proposed parameters for an 80-bit security level are $n_0 = 2, n = 9602, r = 4801, w = 90, t = 84$, which results in a much more practical public-key size of 4801 bits and a private-key size of 9602 bits compared to binary Goppa codes which require around 64 Kbytes for public-keys at the same security level.

In QC-MDPC McEliece, a r-bit plaintext block is encoded into an n-bit codeword to which t errors are added. The parity-check matrix H has constant row weight w and consists of n_0 circulant blocks, the redundant part Q of the systematic generator matrix G consists of $n_0 - 1$ circulant blocks. The public-key has a size of r bits and the private-key has a size of n bits which can be compressed since it is very sparse ($w << n$).

In the following we describe the key-generation, encryption and decryption of the McEliece cryptosystem based on t-error correcting (n, r, w)-QC-MDPC codes.

Key-Generation

The parity-check matrix H is the private-key in QC-MDPC McEliece. Since the (n, r, w)-QC-MDPC code is quasi-cyclic, the parity-check matrix consists of n_0 concatenated $r \times r$ blocks

$$H = [H_0 \,|\, \ldots \,|\, H_{n_0-1}] \,.$$

We denote the first row of each of these blocks by $h_0, \ldots, h_{n_0-1} \in \mathbb{F}_2^r$. The public-key in QC-MDPC McEliece is the corresponding generator matrix G, which is computed from H in standard form as $G = [I_k \,|\, Q]$ by concatenation of the identity matrix $I_k \in \mathbb{F}_2^{k \times k}$ with

$$Q = \begin{pmatrix} (H_{n_0-1}^{-1} \cdot H_0)^\intercal \\ (H_{n_0-1}^{-1} \cdot H_1)^\intercal \\ \cdots \\ (H_{n_0-1}^{-1} \cdot H_{n_0-2})^\intercal \end{pmatrix} .$$

The key generation starts by randomly selecting first row candidates $h_0, \ldots, h_{n_0-1} \in_R \mathbb{F}_2^r$ such that the overall row Hamming weight sums up to $w = \sum_{i=0}^{n_0-1} \mathrm{wt}(h_i)$. Since we intend to generate a code which is quasi-cyclic, the n_0 blocks of the parity-check matrix are generated from the first rows by cyclic shifts. The resulting parity-check matrix belongs to an (n, r, w)-QC-MDPC code with $n = n_0 \cdot r$. If the last block H_{n_0-1} is non-singular, i. e., if $H_{n_0-1}^{-1}$ exists, the public-key is computed as

$$G = [I_k \,|\, Q] \,.$$

Otherwise new candidates for h_{n_0-1} are generated until a non-singular H_{n_0-1} is found.

Encryption

A plaintext $m \in \mathbb{F}_2^k$ is encrypted by encoding it into a codeword using the recipient's public-key G and adding a random error vector $e \in \mathbb{F}_2^n$ of Hamming weight $\mathrm{wt}(e) \leq t$ to it. Hence, the ciphertext is computed as

$$x = (m \cdot G \oplus e) \in \mathbb{F}_2^n.$$

Decryption

Given a ciphertext $x \in \mathbb{F}_2^n$, the recipient removes the error vector e from x using a t-error correcting QC-MDPC decoding algorithm Ψ and the secret code description H yielding

$$mG = \Psi_H(x).$$

Since we have a systematic generator matrix $G = [I_k \,|\, Q]$, the first k positions after decoding mG are equal to the k-bit plaintext.

3.3 The Niederreiter Cryptosystem

The central idea of the Niederreiter cryptosystem is to encode messages into error vectors and to compute their public syndromes from which only the intended receiver who is in possession of the secret code description can recover the error and hence the message. Another difference to McEliece is that parity-check matrices are used instead of generator matrices. Because of its similarities to the McEliece cryptosystem, Niederreiter is often called the dual of McEliece. In the following we introduce traditional Niederreiter in Section 3.3.1, show how to optimize Niederreiter in Section 3.3.2 followed by QC-MDPC Niederreiter in Section 3.3.3.

3.3.1 Traditional Niederreiter Encryption

As for the McEliece cryptosystem we assume being given a binary $[n, k, d]$-Goppa code $\mathcal{C}$, this time defined by its $r \times n$ parity-check matrix H. Further, let there be an efficient t-error correcting decoding algorithm Ψ_Δ which is able to decode any codeword of $\mathcal{C}$ with at most t errors added to it. For binary Goppa codes, Ψ could be the decoding algorithm due to Patterson [Pat75] and Δ would be the Goppa polynomial $g(x)$ and the support $(\alpha_1, \ldots, \alpha_n)$. From such a code the Niederreiter cryptosystem is constructed as usual for public-key encryption systems by three algorithms for key-generation, encryption, and decryption.

Key-Generation

After generating a $r \times n$ parity-check matrix H, select a random $n \times n$ permutation matrix P and a random non-singular $r \times r$ scrambling matrix S. The public-key H' is computed as

$$H' = S \cdot H \cdot P.$$

The private-key is comprised of a permutation matrix P, a scrambling matrix S, and a secret code description Δ resulting in

$$sk = (S, P, \Delta).$$

As again only the inverses of S and P are required for decoding, more commonly their inverses are computed and stored during key-generation as well giving the equivalent private-key

$$sk = (S^{-1}, P^{-1}, \Delta).$$

Encryption

Given a message m and public-key H', the sender encodes m into a binary vector e of length n and Hamming weight $\mathrm{wt}(e) = t$. Transformation of m into a vector with constant weight is achieved through constant weight encoding [Sen05]. After transformation, the ciphertext is computed as

$$x = H' \cdot e^{\intercal}.$$

Decryption

Given a ciphertext $x \in \mathbb{F}_2^r$, decryption is accomplished similarly to McEliece in four steps:

(1) Descramble the ciphertext:

$$x' = S^{-1} \cdot x$$

(2) Decode the descrambled but still permuted ciphertext:

$$e' = \Psi_{\Delta}(x')$$

(3) Revert the permutation:

$$e = P^{-1} \cdot e'$$

(4) Recover the message by reverting the constant weight encoded e into m.

Correctness of the decryption algorithm is shown as follows:

$$\begin{aligned} e^{\intercal} &= P^{-1} \cdot \Psi_{\Delta}(S^{-1} \cdot H' \cdot e^{\intercal}) \\ &= P^{-1} \cdot \Psi_{\Delta}(S^{-1} \cdot S \cdot H \cdot P \cdot e^{\intercal}) \\ &= P^{-1} \cdot \Psi_{\Delta}(H \cdot P \cdot e^{\intercal}) \\ &= P^{-1} \cdot P \cdot e^{\intercal} \\ &= e^{\intercal}. \end{aligned}$$

3.3.2 Improved Niederreiter Encryption

Similar to the improvements applied to McEliece, it is possible for Niederreiter to have a systematic public parity-check matrix H' and to omit permutation matrix P and scrambling matrix S. With the applied optimizations, the size of the Niederreiter public parity-check matrix is reduced from $(n-k) \times n$ to $(n-k) \times k$ and the private-key is reduced to storing the secret code description Δ.

The three algorithms for Niederreiter key-generation, encryption, and decryption are adapted as follows.

Key-Generation

Public-key H' is computed from the parity-check matrix H of a randomly selected binary $[n, k, d]$-Goppa code $\mathcal{C}$ that has an efficient decoding algorithm Ψ_Δ by bringing H to systematic form

$$H'^{\mathsf{T}} = [Q \mid I_{n-k}],$$

e.g., by Gauß-Jordan elimination. The identity part I_{n-k} of H' does not have to be stored, hence reducing the size of the public-key. The private-key is the secret code description Δ, leading to

$$sk = \Delta.$$

Encryption

The encryption algorithm is not changed in optimized Niederreiter. The sender still encodes m into e by constant weight encoding where e is a binary vector of length n and Hamming weight $\mathrm{wt}(e) \leq t$, and computes

$$x = H' \cdot e^{\mathsf{T}}.$$

The only difference is that H' is of systematic form, hence multiplication of e^{T} with the identity part I_{n-k} of H' can be done implicitly by copying e^{T}.

Decryption

Given a ciphertext $x \in \mathbb{F}_2^r$, decryption is accomplished in two instead of four steps:

(1) Decode the ciphertext: $e = \Psi_\Delta(x)$.

(2) Recover the message by reverting the constant weight encoded e into m.

3.3.3 QC-MDPC Niederreiter Encryption

The Niederreiter cryptosystem's key-generation, encryption and decryption based on t-error correcting (n, r, w)-QC-MDPC codes were proposed in [BBMR14]. We introduce QC-MDPC Niederreiter following a similar notation as used in the original publication.

Key-Generation

Key-generation requires to generate a (n, r, w)-QC-MDPC code $\mathcal{C}$ with $n = n_0 r$. The private key is a composed parity-check matrix of the form

$$H = [H_0 \,|\, \ldots \,|\, H_{n_0-1}]$$

which exposes a decoding trapdoor. The public-key is a systematic parity-check matrix

$$H' = [H_{n_0-1}^{-1} \cdot H] = [H_{n_0-1}^{-1} \cdot H_0 \,|\, \ldots \,|\, H_{n_0-1}^{-1} \cdot H_{n_0-2} \,|\, I]$$

which hides the trapdoor but allows to compute syndromes of the public code.

In order to generate a (n, r, w)-QC-MDPC code with $n = n_0 r$, select the first rows $h_0, \ldots, h_{n_0-1}$ of the n_0 parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ at random with Hamming weight $\sum_{i=0}^{n_0-1} \mathrm{wt}(h_i) = w$ and check that H_{n_0-1} is invertible (which is only possible if the row weight d_v is odd). The parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ are generated by $r-1$ quasi-cyclic shifts of the first rows $h_0, \ldots, h_{n_0-1}$. Their concatenation yields the private parity-check matrix H. The public systematic parity-check matrix H' is computed by multiplication of $H_{n_0-1}^{-1}$ with all blocks H_i. Since the public and private parity-check matrices H' and H are quasi-cyclic, it suffices to store their first rows instead of the full matrices. The identity part I of the public-key is usually not stored.

Encryption

Given a public-key H' and a message $m \in \mathbb{Z}/\binom{n}{t}\mathbb{Z}$, encode m into an error vector $e \in \mathbb{F}_2^n$ with $wt(e) = t$. The ciphertext is the public syndrome

$$s' = He^\intercal \in \mathbb{F}_2^r.$$

Decryption

Given a public syndrome $s' \in \mathbb{F}_2^r$, recover its error vector using a t-error correcting (QC-)MDPC decoder Ψ_H with private key H. If

$$e = \Psi_H(s')$$

succeeds, return e and transform it back to message m. On failure of Ψ_H return $\perp$.

3.4 Security of Code-Based Cryptography

The security of cryptographic schemes based on coding theory is usually considered twofold: ciphertext security (decoding attacks) and key security (structural attacks). Decoding attacks try to recover encrypted messages from ciphertexts while structural attacks try to recover the private-key from the public code. The ciphertext security of the McEliece cryptosystem is based on the hardness of finding a codeword of an arbitrary linear code which has minimum distance to a given input vector. This is known as the general decoding problem which was proven to be

NP-complete in [BMv78]. The NP-completeness of the related problem of finding the minimum distance of a general code was proven in [Var97].

So far, the best generic message recovery attacks against McEliece and Niederreiter with binary Goppa codes are based on generic decoding attacks, so called information-set decoding (ISD) algorithms which allow to decode random linear codes. This attack was presented in the original publication of the McEliece cryptosystem [McE78]. First ISD variants were presented in [LB88, Leo88, Ste89], an improved ISD attack by Canteaut and Sendrier [CS98] found the originally proposed parameters of McEliece to not reach the proclaimed security level such that the parameters had to be adapted. Follow-up work by [BLP08, FS09, Pet10, BLP11] improved on this attack, further improved ISD attacks were presented in [MMT11, BJMM12]. However, none of these attacks are of devastating nature for code-based cryptosystems. With adapted parameters that take improved attacks into account, the McEliece and Niederreiter cryptosystems are still considered cryptographically secure public-key algorithms, especially when used with binary Goppa codes. In fact, the long time of cryptanalysis without a serious weakness in the structure of the cryptosystems is one of the strongest security arguments of McEliece and Niederreiter. This assumption is furthermore underlined by a recent first implementation of information-set decoding on special-purpose hardware which was presented in [HZP14]. Their work showed that even with special-purpose hardware implementations no significant attack speed-ups are achievable. In fact it seems that the attack realization adds non-negligible overhead to the theoretically assumed attack costs.

An early observation on the McEliece cryptosystem is that if two ciphertext c_1, c_2 with a low Hamming distance are observed by an attacker, i.e., if $\text{dist}(c_1, c_2) \leq 2t$, there is a high probability that those two ciphertexts encrypt the same plaintext. This is due to the fact that limited entropy is used during encryption ($n \gg t$). This observation first appeared in the Master thesis of Heiman [Hei87]. Later, Berson [Ber97] defined a message-resend condition for McEliece as having two ciphertext $c_1 = mSGP + e_1$ and $c_2 = mSGP + e_2$ with $e_1 \neq e_2$. While the expected Hamming distance of cryptograms of different messages is around $n/2$, the Hamming distance of c_1 and c_2 is limited to at most $\text{dist}(c_1, c_2) \leq 2t$ because both only differ in t positions from $mSGP$. Hence, in the worst case the ciphertexts c_1 and c_2 differ in $2t$ positions if the error vectors e_1 and e_2 do not share set bits in any position. If they do, these error bits cancel each other out[2], reducing the Hamming distance to less than $2t$. Note, the improved McEliece without scrambling and permutation matrices is susceptible to the same attack.

Building on this observation Berson showed in [Ber97] that it is even possible to recover the plaintext from resent encrypted messages of the form c_1, c_2 in βk^3 time, where β is a small constant. Furthermore, he generalized the attack to a related message attack, assuming two ciphertexts of the form $c_1 = m_1SGP + e_1$ and $c_2 = m_2SGP + e_2$ with $m_1 \neq m_2, e_1 \neq e_2$ and knowledge of a linear relation between m_1 and m_2. This attack succeeds as before in βk^3 time.

Another problem of the original McEliece cryptosystem is its malleability [EOS07]. Malleability is a (usually undesired) property of cryptosystems which allows an attacker to modify ciphertexts such that they result in different but valid ciphertexts whose modification is not detectable by the receiver. Malleability is quite common, e.g., the plain RSA encryption/signature scheme is malleable without a padding scheme such as PKCS#1 [RSA12]. In case of

[2]Recall that $c_1, c_2 \in \mathbb{F}_2^n$ and $1 + 1 = 0 \mod 2$.

McEliece, an attacker is able to add any number of rows of the public-key G to a ciphertext yielding another valid ciphertext. In plaintext this can be seen as the capability of adding any m' to the intended plaintext m which is encrypted to $x = mG + e$ by computing

$$x' = m'G + x = (m + m')G + e.$$

The receiver successfully decrypts the ciphertext x' to $m+m'$ without detecting a modification.

Furthermore, the complexity of ISD attacks is significantly reduced if the plaintext is partially known to an attacker [CS98]. Assuming l bits of the plaintext are known, their contributing parts to the ciphertext can be computed from the corresponding rows of G. The attacker subtracts these rows from the ciphertext. The modified ciphertext now needs to be decoded in a code of reduced dimension $k - l$ instead of dimension k which reduces the attack complexity.

The key security of the McEliece cryptosystem is based on the indistinguishability of the public generator matrix from a random matrix of the same size. Key recovery attacks in code-based cryptography are usually structural attacks which recover information about the private code from the public code description, i.e., recovery of private-key information from public-keys. McEliece and Niederreiter with binary Goppa codes did not encounter a successful key-recovery attack so far. However, there are negative examples when using different code classes, usually with some added structure. McEliece with maximum-rank-distance codes was proposed in [GPT91] and got broken in [Gib95, Gib96]. The Niederreiter scheme with generalized Reed-Solomon codes was successfully attacked in [SS92]; the attack was further improved in [Wie10]. Using binary Goppa codes instead of GRS codes was found to prevent the attack.

The suggested QC-MDPC McEliece/Niederreiter parameters in [MTSB13] account for the best currently known ISD attack of [BJMM12] and the improvements achieved by the DOOM-attack [Sen11] to counter previous attacks on McEliece schemes which were based on the combination of a quasi-cyclic/dyadic structure with some algebraic code information. Furthermore, [MTSB13] state that a quasi-cyclic structure by itself does not imply a significant improvement for an adversary. The description of McEliece based on QC-MDPC codes in Section 3.2.3 eliminates the scrambling matrix S and the permutation matrix P which were used in the original description of the McEliece cryptosystem. An IND-CCA conversion (e.g., [KI01, NIKM08]) allows G to be in systematic form without introducing security flaws. In addition, Perlner [Per14] showed that McEliece/Niederreiter with cyclo-symmetric MDPC codes as proposed in [BBMR14] do not reach the proclaimed security levels since improved information set decoding attacks were not correctly accounted for during parameter selection. It is worth noting that Perlner also states that his attack does not affect quasi-cyclic MDPC codes and even places QC-MDPC codes above CS-MDPC codes in terms of efficiency (with adapted CS-MDPC parameters). A detailed discussion of the security of QC-MDPC McEliece is given in [MTSB13].

To prevent commonly known attacks, e.g., reaction attacks or malleability attacks, cryptosystems used in practice should provide *indistinguishability under adaptive chosen-ciphertext attacks* (IND-CCA). In case of McEliece and Niederreiter, so called IND-CCA conversion can be applied. The McEliece variants proposed by Kobara and Imai [KI01] apply the IND-CCA conversions of Fujisaki and Okamoto [FO99, FO13] and Pointcheval [Poi00] to McEliece. They provide IND-CCA security and were proven as secure as the original McEliece scheme. A rather new variant is the IND-CCA secure hybrid-encryption scheme which was developed on the basis

of the Niederreiter cryptosystem by Persichetti [Per13]. An extensive list of security definitions of several security goals such as *indistinguishability under chosen-plaintext attacks* (IND-CPA) and IND-CCA as well as a closer look at IND-CCA conversions are provided in Chapter 7.3.

Although currently there are no indications of weaknesses, we would like to point out that QC-MDPC codes in combination with McEliece and Niederreiter public-key encryption is a fairly new proposal which has seen few cryptanalytic results so far. Hence, one goal of this thesis is to highlight the excellent properties in practice which are offered by QC-MDPC codes in code-based cryptosystems and to attract more attention of cryptanalysts towards these schemes.

For further reading, we recommend the detailed insights into the cryptanalytic efforts of the McEliece and Niederreiter cryptosystems which is provided in [EOS07]. An overview of existing side-channel attacks on code-based cryptosystems in given in Chapters 5.4 and 6.1.

3.5 Parameter Selection

Parameter selection is a challenging task for any cryptosystem and commonly requires a trade-off between security and practicality, e.g., performance, key size, length of the plain-and ciphertexts. From a security point-of-view it is tempting to choose parameters which provide large margins against known attacks. On the other hand, overestimated parameters commonly cause severe drawbacks with regard to performance and key/message sizes.

In practice, three security levels are commonly targeted: 80 bits, 128 bits, and 256 bits. These security levels can be seen as a way to measure and compare the resistance of a cryptosystem towards the best known attacks on this cryptosystem, e.g., integer factorization in the case of RSA. Security levels are commonly stated in bits, however they actually reflect how many "operations" are required on average by the best known attack to break a cryptosystem. These operations can be vastly different, from single CPU instructions to full-blown en-/decryptions of the cryptosystem under investigation. However, the Bachmann-Landau notation [Bac94, Lan09] is commonly used to state the security level and neglects these constant factors. In order to break a cryptosystems with parameters designed for a security level of 128 bits, an attacker needs to perform $\mathcal{O}(2^{128})$ operations.

Originally, McEliece proposed to use the cryptosystem with binary Goppa codes of size $n = 1024, k = 524, t = 50$. Using the ISD attack presented in the original work of McEliece, breaking the cryptosystem with these parameters requires $\approx 2^{81}$ operations (cf. [McE78, AM89]). Improved attacks presented in [BLP11] lowered the security level reached by the original parameters to $2^{49.69}$. Hence, the parameters were adapted as shown in Table 3.1. In [MTSB13], concrete parameters for 80-/128-/ and 256-bit security levels are proposed for QC-MDPC McEliece (cf. Table 3.2). Since small key sizes are particularly crucial for embedded systems, we select the QC-MDPC parameter sets with $n_0 = 2$ in this work. At an 80-bit security level, the following parameters are proposed: $n_0 = 2, n = 9602, r = 4801, w = 90, t = 84$. With these parameters, a 4801-bit plaintext block is encoded into a 9602-bit codeword to which $t = 84$ errors are added. The parity-check matrix H has constant row weight $w = 90$ and consists of $n_0 = 2$ circulant blocks, the redundant part Q of the generator matrix G consists of $n_0 - 1 = 1$ circulant block. The public-key has a size of 4801 bits and the private-key has a size of 9602 bits which can be compressed to 1440 bits since it is very sparse ($w \ll n$).

Table 3.1: Parameters for different security levels equivalent to symmetric security for McEliece with binary Goppa codes as proposed in [McE78, BS08, BLP08, BLP11, NMBB12]. The public-key size is given in systematic and in original form.

Security level	**n**	**k**	**t**	**PK size** systematic [kB]	**PK size** original [kB]	**Reference**
50-bit	1024	524	50	32	66	[McE78]
80-bit	2048	1696	32	73	424	[BS08]
80-bit	2048	1751	27	64	438	[BLP08]
80-bit	1687	1226	43	69	252	[NMBB12]
128-bit	4096	3604	41	217	1802	[BS08]
128-bit	3178	2384	68	231	925	[BLP11]
256-bit	6944	5208	136	1104	4415	[BLP11]

Table 3.2: Parameters for different security levels for McEliece with QC-MDPC codes as proposed in [MTSB13]. The private-key size is equal to code length n in bits.

Security level	$\mathbf{n_0}$	**n**	**r**	**w**	**t**	**PK size** [kB]
80-bit	2	9602	4801	90	84	0.59
80-bit	3	10779	3593	153	53	0.88
80-bit	4	12316	3079	220	42	1.13
128-bit	2	19714	9857	142	134	1.20
128-bit	3	22299	7433	243	85	1.81
128-bit	4	27212	6803	340	68	2.49
256-bit	2	65542	32771	274	264	4.00
256-bit	3	67593	22531	465	167	5.50
256-bit	4	81932	20483	644	137	7.50

Chapter 4

Efficient Decoding of (QC-)MDPC Codes

Compared to the relatively simple operations involved in McEliece encryption – a vector-matrix multiplication followed by a vector addition – McEliece decryption requires decoding erroneous codewords which generally is a more complex task. Since the selection of an efficient decoding algorithm is crucial to the overall McEliece decryption performance, it is imperative to evaluate and compare available options and to investigate possible optimizations. In this chapter we introduce LDPC and MDPC decoding techniques, evaluate the performance of concrete QC-MDPC McEliece parameter sets and make novel proposals to accelerate decoding and to effectively reduce the probability of decoding failures. We derive and evaluate several decoding variations and compare them among each other to make a justified optimal decoder selection which delivers high performance with least decoding failures.

The research presented in this chapter started out as a joint work with Stefan Heyse and Tim Güneysu, the results were presented at CHES'13 [HvMG13]. Subsequently the author investigated further improved decoding techniques which appeared in the ACM Transactions on Embedded Computing Systems [vMOG15].

Contents

4.1 Introduction

While this work focuses on the cryptographic applications of coding theory, efficient decoding is of general interest also for non-cryptographic coding applications. An error-correcting code whose decoding is time and memory consuming will diminish its usefulness in cryptographic and non-cryptographic coding applications alike. While conceptually it is easy to grasp the idea of decoding by identifying the nearest codeword to a received word (cf. Section 2.2), constructing efficient algorithms for this task is typically hard. In general a better performance is achieved by specifically designed decoding algorithms for a particular code class compared to more general decoding algorithms which can be applied to larger code families. Maximum likelihood decoding of LDPC/MDPC codes on binary symmetric channels is proven to be NP-complete [BMv78]. Hence, it is not possible to achieve optimal decoding for typical LDPC code sizes. On the contrary there are many sub-optimal decoders available which achieve very good results in practice. LDPC decoders can be found in many wide-spread applications today; LDPC codes are specified among others in the TV standards DVB-S2, DVB-T2, DVB-C2 and in the Wi-Fi standards 802.11n / 802.11ac.

The most common class of LDPC and MDPC decoding algorithms is the class of iterative message passing algorithms which are exchanging information back and forth between message nodes and check nodes in the codes' Tanner graph to achieve decoding (cf. Section **??**). Belief propagation decoding [Gal63] is a variant of this decoding technique which passes probabilities between message nodes and check nodes. Belief propagation decoding algorithms for LDPC and MDPC codes are mainly divided into two families commonly referred to as soft- and hard-decision decoders. The family of soft-decision decoding generally offers a better error-correction capability but is computationally more complex than the family of hard-decision bit-flipping algorithms [Gal63]. Especially when handling large codes on embedded platforms, bit-flipping decoders seem more appropriate as they do not require floating-point arithmetic and have lower memory requirements.

Contribution The main contributions of this chapter are the evaluation of several different decoding techniques for MDPC codes as well as the proposal of novel decoder optimizations in order to find an optimal decoding algorithm with regard to parameter sets of QC-MDPC McEliece and Niederreiter public-key encryption. Our decoder optimizations accelerate the syndrome computation, reduce the decoding iterations required on average, and improve the decoding failure rate.

Outline We introduce efficient decoding algorithms for LDPC codes in Section 4.2 and for MDPC codes in Section 4.3. Novel optimizations are proposed in Section 4.4 to accelerate the syndrome computation during decoding, to reduce the average number of decoding iterations, and to decrease the decoding failure rate. We derive several combinations of decoding techniques and optimizations which we evaluate and compare in Section 4.5 with a focus on the proposed QC-MDPC McEliece/Niederreiter parameters of [MTSB13] to select the best decoding algorithms as a basis for our implementations.

4.2 Decoding LDPC Codes

The first decoding algorithms for LDPC codes were proposed by Gallager [Gal63]. In the following we explain the main ideas of how decoding succeeds to eliminate errors from LDPC codewords that were transmitted over noisy channels. The explanation loosely follows [Mis14] and [Nig04]. In Section 4.3 we show how to transfer these decoding principles to MDPC codes.

The belief propagation decoding algorithms make use of the Tanner graph of the code. Probabilities are passed in the Tanner graph between message nodes and check nodes to determine which message nodes should be set to one and which should be set to zero. In each iteration of these decoding algorithms, initial probabilities are sent from message nodes to check nodes and are then adapted and returned vice versa. The term belief propagation stems from the fact that each of the message nodes "beliefs" with a certain probability whether it is supposed to be a one or a zero.

Let the word which shall be decoded be denoted as $x = [x_1, \ldots, x_n], x_i \in \mathbb{F}_2$. Before decoding starts, each received bit x_i is assigned with a probability whether $x_i = 1$. Depending on the channel, the probabilities can all be the same for each received bit or can differ significantly. The probabilities are assigned to the message nodes of the Tanner graph which send their initial probability to all connected check nodes. The check nodes make new estimates based on the received probabilities and send them back to the message nodes. This process is iterated in discrete steps either until the probability of each message node becomes negligibly close to 1 or 0, or it is iterated for a fixed number of iterations after which a hard decision is made by rounding to either 1 or 0 based on the estimated probabilities.

Picking up the example presented in Section 2.4.1, the first row of the parity-check matrix $H_{[10,5]}$ is 1111000000, i.e., the first check node is given by $c_1 = x_1 + x_2 + x_3 + x_4$. The check node receives probabilities p_1, p_2, p_3, p_4 from the connected message nodes v_1, v_2, v_3, v_4 in the Tanner graph (cf. Figure 2.5). The check node then computes new estimates p'_1, p'_2, p'_3, p'_4 from the receives probabilities as:

$$
\begin{aligned}
p'_1 &= p_2(1-p_3)(1-p_4) + p_3(1-p_2)(1-p_4) + p_4(1-p_2)(1-p_3) + p_2p_3p_4 \\
p'_2 &= p_1(1-p_3)(1-p_4) + p_3(1-p_1)(1-p_4) + p_4(1-p_1)(1-p_3) + p_1p_3p_4 \\
p'_3 &= p_1(1-p_2)(1-p_4) + p_2(1-p_1)(1-p_4) + p_4(1-p_1)(1-p_2) + p_1p_2p_4 \\
p'_4 &= p_1(1-p_2)(1-p_3) + p_2(1-p_1)(1-p_3) + p_3(1-p_1)(1-p_2) + p_1p_2p_3.
\end{aligned}
$$

The new estimates p'_1, p'_2, p'_3, p'_4 are sent back to the corresponding message nodes v_1, v_2, v_3, v_4. At the same time all other check nodes in the Tanner graph compute their updated estimates and send them back to their connected messages nodes as well. Hence, each message nodes receives multiple updated probabilities from all connected check nodes in parallel. Suppose message node v_1 receives three updated probabilities p'_1, p''_1, p'''_1 from the three connected check nodes. For the next iteration message node v_1 prepares the three responses $kp_1p''_1p'''_1$, $kp_1p'_1p'''_1$, and $kp_1p'_1p''_1$, which are returned to the first, second and third check node, respectively. The normalization factor k is computed as $k = 1/(p'_1p''_1p'''_1 + (1-p'_1)(1-p''_1)(1-p'''_1))$. This is iterated several times as discussed above until either the probabilities of all message nodes become close to either 0 or 1 or by a hard decision after a fixed number of rounds.

A simplified version of this decoding technique is the hard-decision bit-flipping decoder which was introduced in [Gal63]. The simplified version computes the number of unsatisfied parity-check equations for each bit of the received word x and compares them to a precomputed threshold b. If the threshold is exceeded, the bit of the received word is directly inverted. Thus, the previously necessary floating point arithmetic for computing updated probabilities are omitted. We discuss this decoder in more detail in the following section when decoding MDPC codes.

The remaining question is how to precompute the bit-flipping threshold b. In fact, it is not one single threshold but a series of bit-flipping thresholds b_i is precomputed for each decoding iteration i. Let P_i denote the probability of a bit being in error after i decoding iterations. The initial error probability is set to $P_0 = \frac{t}{n}$ since we want to correct a randomly generated error of length n and Hamming weight t in the case of QC-MDPC McEliece and Niederreiter. The goal is to have P_i converge to zero with increasing decoding iterations in order to determine the error locations and hence to succeed with decoding.

Assuming the probability of an unsatisfied parity-check (i.e., an odd number of errors in $w_r - 1$ positions) is

$$r_i = \frac{1 - (1 - 2P_i)^{w_r - 1}}{2},$$

[Gal63] computes the probability of a bit being in error after $i + 1$ decoding iterations as

$$P_{i+1} = P_0 \sum_{l=0}^{b-1} \binom{w_c - 1}{l} (1 - r_i)^l r_i^{w_c - 1 - l} + (1 - P_0) \sum_{l=b}^{w_c - 1} \binom{w_c - 1}{l} r_i^l (1 - r_i)^{w_c - 1 - l}.$$

Finding the smallest integer b_i for which

$$\frac{1 - P_0}{P_0} \leq \left[\frac{1 + (1 - 2P_i)^{w_r - 1}}{1 - (1 - 2P_i)^{w_r - 1}} \right]^{2b_i - w_c + 1}$$

holds then leads to the bit flipping threshold b_i for iteration i.

4.3 Decoding (QC-)MDPC Codes

In the following we explain decoding strategies applicable to (QC-)MDPC codes. In particular we propose several variations of known hard-decision bit-flipping algorithms [Gal63, HP10, MTSB13] in order to find an optimal decoding strategy. Given an input $x \in \mathbb{F}_2^n$, the hard decision bit-flipping decoding algorithms are based on the following principle:

(1) Compute the syndrome $s = Hx^T$ of the received word x.

(2) Count the unsatisfied parity-check equations $\#_{\text{upc}}$ associated with each bit of x.

(3) Flip those bits of x which violate more than b equations, where b is a bit-flipping threshold.

(4) Recompute the syndrome of the updated x.

This process is repeated until either the syndrome becomes zero or a predefined maximum number of iterations is reached upon which a decoding error is returned. The main difference between the bit-flipping algorithms is how they determine threshold b:

- In [Gal63], thresholds b_i are precomputed for each iteration i as explained in Section 4.2. Adapting Gallager's precomputation technique to MDPC codes is done by replacing w_r and w_c by w. Hence, the thresholds can be computed by finding the smallest integer b_i for which
$$\frac{1-P_0}{P_0} \leq \left[\frac{1+(1-2P_i)^{w-1}}{1-(1-2P_i)^{w-1}}\right]^{2b-w+1}$$
holds for iteration i.
- [HP10] compute the number unsatisfied parity-check equations for each received bit and set the threshold as the maximum of the unsatisfied parity-check equations $b = \max(\#_{\text{upc}})$.
- [MTSB13] slightly adapt the approach of [HP10] and propose to use $b = \max(\#_{\text{upc}}) - \delta$, for some small δ to accelerate decoding. In case of a decoding failure, δ is decreased and decoding is restarted until $\delta = 0$ where this decoder becomes equal to [HP10].

The number of unsatisfied parity-check equations is equal to the number of shared bits in a row of the parity-check matrix H and the syndrome s. Recall that the syndrome depends, by definition, only on the error e that is added to a codeword c:

$$s = Hx^T = H(c+e)^T = Hc^T + He^T = He^T$$

since $Hc^T = 0$ by definition.

4.4 Decoder Optimizations

Below we propose new ways to accelerate the syndrome computation and to reduce the decoding-failure rate. We show that these novel techniques not only accelerate decoding but also decrease the number of required decoding iterations on average.

Accelerating the Syndrome Computation Bit-flipping decoders in the literature recompute the syndrome after every decoding iteration to decide whether decoding was successful or not. The cost of one syndrome computation alone can be approximated at around twice the cost of one encoding in the context of QC-MDPC codes with $n_0 = 2$.

We propose an optimization that can be applied to all bit-flipping decoders based on the following observation: if the number of unsatisfied parity-check equations exceeds threshold b, the corresponding bit in the ciphertext is flipped and the syndrome changes. We stress that the syndrome does not change arbitrarily, but the new syndrome is equal to the old syndrome accumulated with row h_j of the parity-check matrix that corresponds to the flipped bit at position j:

$$s_{\text{new}} = s_{\text{old}} \oplus h_j.$$

By keeping track of which bits are flipped and by updating the syndrome accordingly, the syndrome recomputation can be omitted. Since only few bits are flipped in each decoding iteration, updating the syndrome requires far less additions than an ordinary syndrome computation.

Reducing Decoding Iterations There are two ways to apply the syndrome computation optimizations from the previous paragraph. One is to store all changes to the syndrome in a separate register and to add the changes at the end of a decoding iteration to the syndrome. This way, the syndrome computation is accelerated but the decoding behavior remains unchanged. The other possibility is to directly apply the changes to the syndrome whenever a ciphertext bit is flipped. This similarly accelerates the syndrome computation but it also affects the decoding behavior since the modified syndrome is used to determine the unsatisfied parity-check equations of following ciphertext bits. We explore both approaches in Section 4.4.1 and show that directly modifying the syndrome reduces the average number of decoding iterations.

Reducing Decoding Failures The decoder proposed in [Gal63] uses precomputed thresholds based on the code parameters. We found that the error-correcting capability of this decoder can be improved by incrementing the precomputed thresholds by a small Δ in case of a decoding failure and restart decoding with the adapted thresholds. When restarting, the initial syndrome does not need to be recomputed as it can be restored from the first decoding attempt. Incrementing the precomputed thresholds upon a decoding failure is similar to the approach taken by [MTSB13] when decrementing δ upon a decoding failure. We achieved the best improvements when setting $\Delta = 1$ and after every decoding failure increasing $\Delta = \Delta + 1$ until reaching a predefined $\Delta_{\max}$.

4.4.1 Investigated Decoding Techniques

Estimating the error-correction capability of LDPC and MDPC codes is non-trivial and influenced by several factors. Hence, we derive several bit-flipping algorithms, evaluate their error-correcting capability, count how many iterations are required on average to decode a codeword, and measure the execution time. Since we are mostly targeting embedded systems, we omit variants that store counters for each ciphertext bit to compute their number of unsatisfied parity-check equations $\#_{\text{upc}}$. Counters would allow to skip the second computation of $\#_{\text{upc}}$ in some decoder variants ($\mathcal{A}$, $\mathcal{C}_1$ and $\mathcal{C}_2$), but would increase the memory consumption to at least $n \cdot \lceil \log_2(w) \rceil$ bits which is unacceptable for microcontrollers and FPGAs.

The first two decoders under investigation are:

Decoder $\mathcal{A}$ is given in [MTSB13] and computes the syndrome, checks the number of unsatisfied parity-check equations once to compute $\max(\#_{\text{upc}})$ and a second time to flip all ciphertext bits that violate $\geq \max(\#_{\text{upc}}) - \delta$ equations. Afterwards, the syndrome is recomputed and compared to zero. If decoding is not successful after some fixed maximum of iterations, δ is reduced to $\delta = \delta - 1$ and the decoding process is restarted. This is repeated after each unsuccessful decoding attempt until $\delta = 0$ where the decoder becomes equal to the decoder of [HP10] which always uses $b = \max(\#_{\text{upc}})$.

Decoder $\mathcal{B}$ is given in [Gal63] and computes the syndrome, checks the number of unsatisfied parity-check equations once per iteration i and directly flips the current ciphertext bit if $\#_{\text{upc}}$ is larger than a precomputed threshold b_i. Afterwards, the syndrome is recomputed and compared to zero.

In order to evaluate our optimizations of the syndrome computation and the adaptive precomputed thresholds, we derive the following decoders:

Decoder $\mathcal{C}_1$ computes the syndrome, checks the number of unsatisfied parity-check equations once to compute $\max(\#_{\text{upc}})$ and a second time to flip all ciphertext bits that violate $\geq \max(\#_{\text{upc}}) - \delta$ equations. If a ciphertext 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, resulting in the syndrome of the modified ciphertext without requiring a full recomputation. In case of a decoding error, δ is decremented as in decoder $\mathcal{A}$.

Decoder $\mathcal{C}_2$ computes the syndrome, checks the number of unsatisfied parity-check equations once to compute $\max(\#_{\text{upc}})$ and a second time to flip all ciphertext bits that violate $\geq \max(\#_{\text{upc}}) - \delta$ equations. If a ciphertext bit j is flipped, the corresponding row h_j of the parity-check matrix is *directly* added to the current syndrome to always work with an up-to-date syndrome. In case of a decoding error, δ is decremented as in decoder $\mathcal{A}$.

Decoder $\mathcal{C}_3$ is similar to decoder $\mathcal{C}_2$ but compares the syndrome to zero after each flipped bit and aborts the current iteration immediately once it becomes zero.

Decoder $\mathcal{D}_1$ is similar to decoder $\mathcal{B}$ but uses the *direct* update of the syndrome.

Decoder $\mathcal{D}_2$ is similar to decoder $\mathcal{D}_1$ and in addition increments the precomputed thresholds in case of a decoding failure until $\Delta_{\max} = 5$.

Decoder $\mathcal{D}_3$ is similar to decoder $\mathcal{D}_2$ and in addition uses early termination as decoder $\mathcal{C}_3$.

The features of all investigated decoders are summarized in Table 4.1 to ease comparison.

4.5 Decoding Performance Evaluation

The following performance measurements are taken for randomly generated QC-MDPC codes with parameters $n_0 = 2, n = 9602, r = 4801, w = 90$. Instead of only using the proposed $t = 84$ from the parameter set of [MTSB13], we evaluate the behavior of all decoders for error weights $t = \{84, \ldots, 90\}$ to make decoding more difficult and to provoke decoding errors. A total of 1,000 random codes and 10,000 random decoding trials per code were evaluated on a computing cluster equipped with 288 AMD Opteron 6276 CPU cores running at 2.3 GHz.

For decoders with precomputed thresholds b_i we used the approach explained in Section 4.3 to precompute the b_i's for every iteration i similar to [Gal63]. We list the thresholds in Table 4.2. For decoders with $b = \max(\#_{\text{upc}}) - \delta$, we found that the smallest number of iterations are required when starting with $\delta = 5$[1]. A decoding failure is returned in case the decoder did not succeed within ten iterations.

[1] In the latest version of [MTSB12] the authors also suggest to use $\delta \approx 5$ for the given parameters.

Table 4.1: Features of the investigated decoders for (QC-)MDPC codes. The bit-flipping threshold b is either derived from the maximum number of unsatisfied parity-check equations on-the-fly or precomputed based on the parameters of the code. We also mark if the thresholds are adapted upon a decoding failure or not. The syndrome is either updated after each decoding round or after every change to the ciphertext. Comparing the syndrome to zero is done either after each decoding round or after every update of the syndrome.

Decoder	**Threshold**			**Syndrome Update**			**Syndrome Check**	
	on-the-fly	precomp.	adaptive	each round	temp.	direct	every iter.	every upd.
$\mathcal{A}$	✓		✓	✓			✓	
$\mathcal{B}$		✓		✓			✓	
$\mathcal{C}_1$	✓		✓		✓		✓	
$\mathcal{C}_2$	✓		✓			✓	✓	
$\mathcal{C}_3$	✓		✓			✓		✓
$\mathcal{D}_1$		✓				✓	✓	
$\mathcal{D}_2$		✓	✓			✓	✓	
$\mathcal{D}_3$		✓	✓			✓		✓

Table 4.2: Precomputed bit-flipping thresholds for ten decoding iterations used during the evaluation of decoders $\mathcal{B}$, $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$. The thresholds were computed for code parameters $n_0 = 2, n = 9602, r = 4801, w = 90$ and error weights $t = \{84, \ldots, 90\}$. See Section 4.3 for details about how these thresholds are computed.

Error Weight	**Bit-flipping Thresholds**
84	$[26, 24, 22, 21, 21, 21, 21, 21, 21, 21]$
85	$[26, 24, 22, 21, 21, 21, 21, 21, 21, 21]$
86	$[26, 24, 22, 21, 21, 21, 21, 21, 21, 21]$
87	$[26, 24, 22, 21, 21, 21, 21, 21, 21, 21]$
88	$[26, 24, 22, 21, 21, 21, 21, 21, 21, 21]$
89	$[26, 25, 22, 21, 21, 21, 21, 21, 21, 21]$
90	$[27, 25, 23, 21, 21, 21, 21, 21, 21, 21]$

4.5.1 Decoder Comparison

The average number of iterations required to decode a codeword with t added errors and the decoding failure rate are listed in Table 4.3 for all decoders described in Section 4.4.1 and Table 4.1. Figure 4.1a illustrates the timing behavior of the evaluated decoders, Figure 4.1b compares the number of required decoding iterations on average, and Figure 4.2 shows the observed decoding failures.

The timings given in Table 4.3 and Figure 4.1a should only be used to compare the decoders among each other. The evaluation was done in software and was not particularly optimized for speed. It was designed to keep the generating polynomial h in memory instead of the whole parity-check matrix H. All following rows of H are derived at runtime by rotating the polynomial.

Comparing the Decoders from Literature When comparing the two evaluated decoders from literature ($\mathcal{A}$ and $\mathcal{B}$), it is evident that decoder $\mathcal{B}$ requires around 40% less decoding iterations on average and around half the time to decode an erroneous codeword. On the other hand, decoder $\mathcal{B}$ encounters a higher number of decoding failures than decoder $\mathcal{A}$, which, depending on the fault tolerance of the system, might be undesirable.

Acceleration of the Syndrome Computation The acceleration of not having to recompute the syndrome becomes apparent when comparing decoder $\mathcal{A}$ with $\mathcal{C}_1$. The only difference between the two decoders is that $\mathcal{C}_1$ benefits from the accelerated syndrome update. The decoding behavior of both decoders is still the same, as the changes to the syndrome are stored in a temporary register and the syndrome is only updated after each decoding round. With this technique we gain an average reduction of the execution time by 20%.

Direct Syndrome Update Directly updating the syndrome when flipping a ciphertext bit has an even stronger impact on the decoding performance as well as on the decoding failure rate. Not only do we speed up the computation time, but we also reduce the average number of required decoding iterations by 40% (compare decoders $\mathcal{C}_1$ and $\mathcal{C}_2$). Furthermore, the number of decoding failures is highly reduced (compare decoders $\mathcal{C}_1$ to $\mathcal{C}_2$ and $\mathcal{B}$ to $\mathcal{D}_1$). We had to raise the error weight considerably during our evaluations to provoke decoding failures in case of decoder $\mathcal{C}_2$. When decoding with precomputed thresholds, decoding failures occur 80 times less using this technique (compare $\mathcal{B}$ and $\mathcal{D}_1$).

Combining Gallager's precomputed thresholds with a directly updated syndrome results in the lowest number of decoding iterations (compare decoders $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$). On average we save 2.9 iterations compared to decoder $\mathcal{A}$ and 0.7 iterations compared to $\mathcal{B}$ (cf. Figure 4.1b). Less iterations directly relate to the execution time. Combined with our syndrome update technique decoding is overall 2-4 times faster as shown in Figure 4.1a.

Adaptive Thresholds Adapting the precomputed thresholds upon a decoding error as proposed in Section 4.4 leads to the lowest decoding failure rates among all decoders under investigation (compare $\mathcal{D}_1$ with $\mathcal{D}_2/\mathcal{D}_3$). During 100,000,000 random decoding tries we only

encountered two decoding failures for decoders $\mathcal{D}_2/\mathcal{D}_3$ and we had to raise the error weight from 84 to 90 for this to happen.

The average number of decoding iterations and the average execution time increase only very slightly when using the adapted thresholds. The small timing advantage of decoders $\mathcal{C}_3/\mathcal{D}_3$ over $\mathcal{C}_2/\mathcal{D}_2$ is due to the immediate termination if the syndrome becomes zero.

Early Detection of Decoding Errors Another interesting observation for all decoders: if an erroneous codeword is decodable, it is decoded with an overwhelming probability after a small number of iterations. We noticed that if a ciphertext is not decoded within 4-6 iterations, a higher number of iterations rarely leads to a successful decoding without adapting the thresholds. Therefore, we conclude that an early detection of decoding failures is possible and that is it more beneficial to adapt the thresholds and restart decoding instead of increasing the number of decoding iterations with the same thresholds.

4.5.2 Decoding Algorithm Selection

Based on the evaluation results, we select decoders $\mathcal{D}_1/\mathcal{D}_2$ as the basis for our implementations throughout this thesis. Even though decoder $\mathcal{D}_3$ has a small timing advantage, its runtime is inherently dependent on secret data (the syndrome) which might introduce a timing side-channel. Although we are not aware of a way to exploit the information of the time it takes for the syndrome to become zero, history has shown that it is advisable to avoid leaking timing information, especially if it can be avoided at low cost.

Decoder $\mathcal{D}_1$ is summarized as:

(1) Compute the syndrome $s = Hx^T$ of the received ciphertext x.

(2) Count the number of unsatisfied parity-checks for every ciphertext bit.

(3) If the number of unsatisfied parity-checks for a ciphertext bit exceeds a precomputed threshold, flip the ciphertext bit and directly update the syndrome.

(4) If $s = 0^r$, the codeword was decoded successfully. If $s \neq 0^r$, go to Step (2) or abort after a defined maximum of iterations with a decoding error.

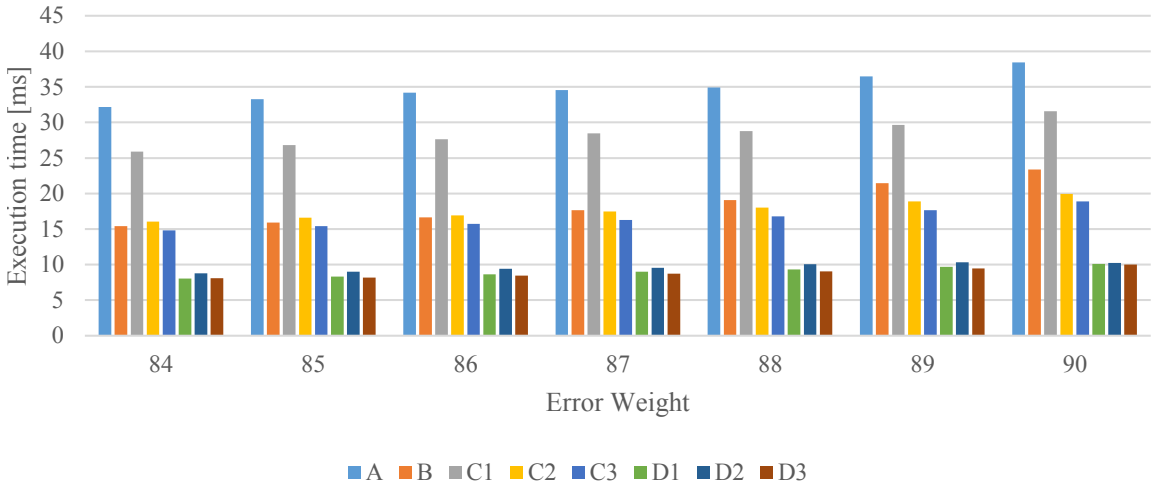
Decoder $\mathcal{D}_2$ can be seen as a wrapper around $\mathcal{D}_1$ which modifies the decoding thresholds upon a decoding error and then calls $\mathcal{D}_1$ again.

4.6 Conclusion

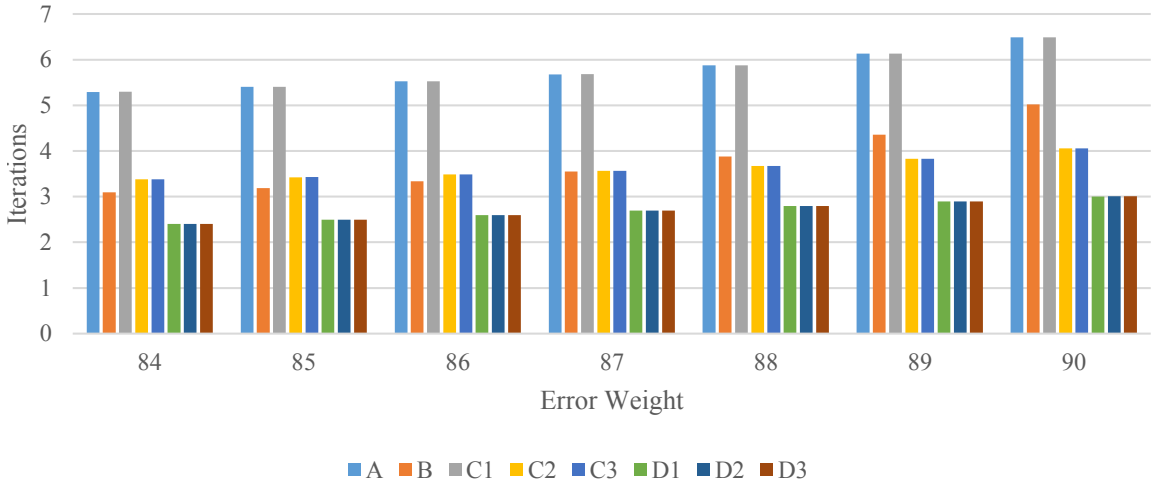
In this chapter we introduced LDPC and MDPC decoding techniques, evaluated the performance of existing QC-MDPC decoders and made novel proposals to accelerate decoding and to effectively reduce the probability of decoding failures. We derived and evaluated several decoding variations and compared them among each other to make a justified optimal decoder selection which delivers high performance with least decoding failures.

Table 4.3: Evaluation of the performance and error correcting capability of the decoders described in Section 4.4.1 for QC-MDPC codes with parameters $n_0 = 2, n = 9602, r = 4801, w = 90$ on AMD Opteron 6276 CPUs at 2.3 GHz.

Variant	#errors	time in ms	failure rate	avg. #iterations
Decoder $\mathcal{A}$	84	32.15	0.0000000	5.2922
	85	33.26	0.0000010	5.4027
	86	34.16	0.0000058	5.5234
	87	34.56	0.0000196	5.6792
	88	34.90	0.0000794	5.8728
	89	36.47	0.0002760	6.1311
	90	38.44	0.0008348	6.4876
Decoder $\mathcal{B}$	84	15.41	0.0002957	3.0936
	85	15.93	0.0012654	3.1854
	86	16.67	0.0046348	3.3343
	87	17.67	0.0138536	3.5515
	88	19.07	0.0360551	3.8790
	89	21.47	0.0798088	4.3542
	90	23.36	0.1534663	5.0191
Decoder $\mathcal{C}_1$	84	25.89	0.0000002	5.2961
	85	26.79	0.0000008	5.4014
	86	27.62	0.0000060	5.5250
	87	28.46	0.0000282	5.6822
	88	28.76	0.0000798	5.8730
	89	29.65	0.0002744	6.1354
	90	31.55	0.0008442	6.4895
Decoder $\mathcal{C}_2$	84	16.03	0.0000000	3.3780
	85	16.60	0.0000000	3.4254
	86	16.90	0.0000000	3.4864
	87	17.47	0.0000000	3.5648
	88	18.01	0.0000002	3.6726
	89	18.88	0.0000026	3.8301
	90	19.96	0.0000098	4.0596
Decoder $\mathcal{C}_3$	84	14.83	0.0000000	3.3776
	85	15.42	0.0000000	3.4263
	86	15.74	0.0000000	3.4871
	87	16.26	0.0000004	3.5656
	88	16.77	0.0000004	3.6736
	89	17.65	0.0000020	3.8308
	90	18.90	0.0000096	4.0602
Decoder $\mathcal{D}_1$	84	8.02	0.0000037	2.4019
	85	8.32	0.0000180	2.4985
	86	8.65	0.0000579	2.5975
	87	8.99	0.0001879	2.6965
	88	9.34	0.0005487	2.7928
	89	9.70	0.0014897	2.8914
	90	10.09	0.0036869	2.9992
Decoder $\mathcal{D}_2$	84	8.79	0.0000000	2.4021
	85	9.00	0.0000000	2.4982
	86	9.40	0.0000000	2.5977
	87	9.57	0.0000000	2.6962
	88	10.07	0.0000000	2.7938
	89	10.32	0.0000000	2.8950
	90	10.26	0.0000002	3.0106
Decoder $\mathcal{D}_3$	84	8.10	0.0000000	2.4021
	85	8.17	0.0000000	2.4975
	86	8.47	0.0000000	2.5964
	87	8.71	0.0000000	2.6964
	88	9.06	0.0000000	2.7941
	89	9.45	0.0000000	2.8948
	90	9.99	0.0000000	3.0109

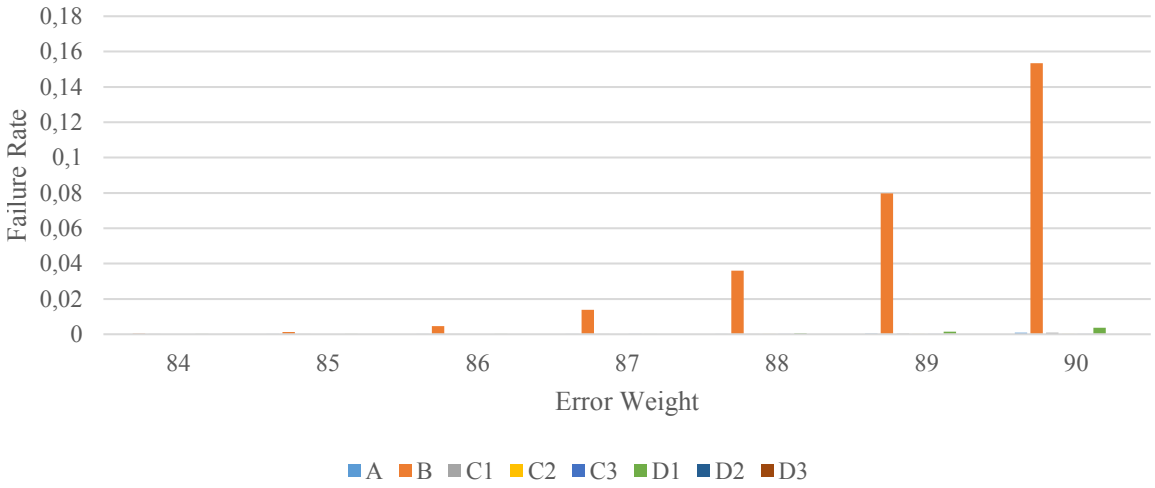


(a) Timing behavior of the evaluated decoders for error weights $t \in \{84, \ldots, 90\}$.

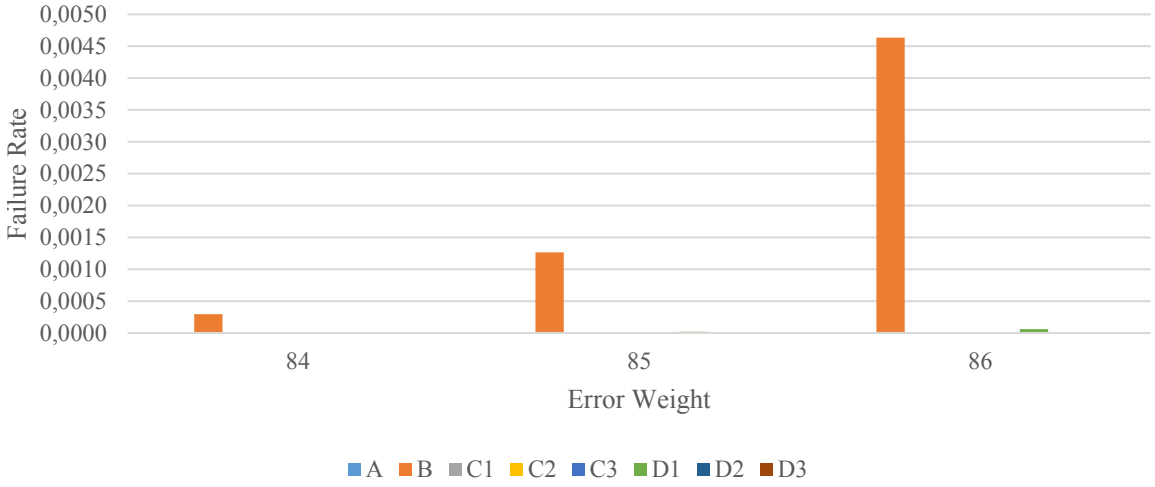


(b) Number of decoding iterations on average of the evaluated decoders for error weights $t \in \{84, \ldots, 90\}$.

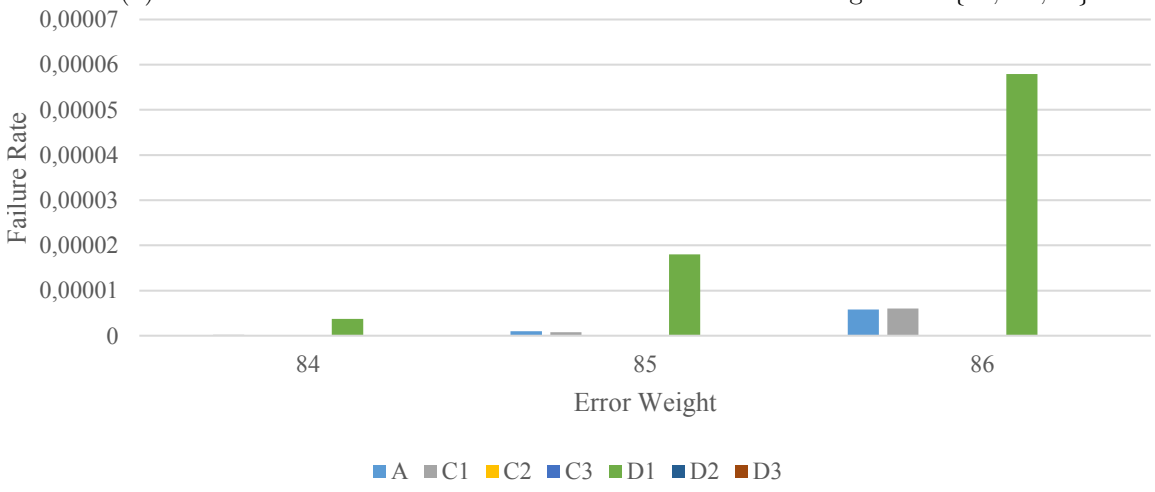
Figure 4.1: Analysis of the timing behavior and the number of decoding iterations of the evaluated decoders.



(a) Failure rates of all evaluated decoders for error weights $t = \{84, \ldots, 90\}$.



(b) Zoomed failure rates of all evaluated decoders for error weights $t = \{84, \ldots, 86\}$.



(c) Failure rates of all evaluated decoders except decoder $\mathcal{B}$ for error weights $t = \{84, \ldots, 86\}$.

Figure 4.2: Failure rates of the evaluated decoders in three different resolutions.

Chapter 5

QC-MDPC McEliece for Reconfigurable Hardware

In this chapter we develop combined QC-MDPC McEliece en-/decryption cores for high-performance and lightweight FPGA applications. Our high-performance implementation achieves $13.7\,\mu s/82.1\,\mu s$ *for en-/decryption and requires 2,924/10,988 slices on Xilinx Virtex-6. Furthermore, we demonstrate that the cryptosystem can be implemented with a significantly smaller resource footprint – still achieving reasonable performance sufficient for many applications, e.g., challenge-response protocols or hybrid encryption. More precisely, our lightweight FPGA design requires just 68 slices for the encryption unit and around 150 slices for the decryption unit. It is able to en-/decrypt an input block in 2.2 ms and 13.4 ms, respectively on Xilinx Spartan-6.*

This research was presented at CHES'13 and DATE'14 and is a joint work with Tim Güneysu. An extended version appeared in the ACM Transactions on Embedded Computing Systems [HvMG13, vMG14a, vMOG15]. The side-channel attacks and countermeasures are joint work with Cong Chen, Thomas Eisenbarth and Rainer Steinwandt. The results were presented at ACNS'15 and SAC'15 and appeared in the IEEE Transactions on Information Forensics & Security [CEvMS15, CEvMS16b, CEvMS16a].

Contents

5.1 Introduction

Field programmable gate arrays (FPGA) are reconfigurable integrated circuits which mainly consist of configurable logic blocks, *slices* in Xilinx terms. Each slice contains lookup tables (LUT), flip-flops (FF), and surrounding logic, e.g., to allow fast carry chaining. The logic blocks are interconnected by programmable switch matrices. In addition, embedded resources such as block memories (BRAM) and digital signal processors (DSP) are available on FPGAs.

Generally, FPGAs allow a faster time-to-market and have lower non-recurring costs for development compared to application-specific integrated circuits (ASIC) which are fixed integrated circuits that fulfill dedicated and unchangeable purposes. ASICs typically excel with higher performance, lower energy consumption and lower recurring costs for large product volumes in comparison to FPGAs. FPGAs are commonly used to prototype and test ASIC designs before production and for low-volume applications.

FPGA implementations of the code-based McEliece and Niederreiter cryptosystems so far are restricted to binary Goppa codes. The first implementation of a code-based cryptosystem ("MicroEliece") was proposed for a Xilinx Spartan-3 FPGA [EGHP09]. Since the storage capacity of the FPGA did not suffice, external memory had to be used to store the public-key. A hardware McEliece implementation based on Goppa codes including a CCA2 conversion was presented for a Virtex5-LX110T FPGA in [SWM+09, SWM+10]. Another McEliece co-processor was proposed for a Virtex5-LX110T FPGA by [GDUV12] with the main design goal of optimizing the speed/area ratio. Niederreiter was implemented using Goppa codes in [HG12] for a Virtex6-LX240T FPGA demonstrating that Niederreiter encryption can provide high performance with a moderate amount of resources.

Previous code-based cryptosystem implementations in reconfigurable hardware require large amounts of memory to store public-keys. Memory is either provided externally or through a large number of internal block RAM. Since storage capacity in embedded applications is typically low and expensive, the much smaller key sizes of QC-MDPC codes compared to binary Goppa codes are of high practical relevance. Hence, we explore the design space of QC-MDPC McEliece by providing high-performance and lightweight implementations of the cryptosystem targeting Xilinx's Virtex-6 and Spartan-6 FPGAs.

Contribution This chapter presents two FPGA implementations of QC-MDPC McEliece. The first implementation is designed for high-performance applications while the second implementation targets lightweight and low-cost applications. Both implementations provide encryption and decryption functionality. Our high-performance implementation of QC-MDPC in reconfigurable hardware targets Xilinx's Virtex-6 FPGAs. Virtex-6 devices are powerful FPGAs offering thousands of slices, whereas our lightweight implementation targets Xilinx's low-cost Spartan-6 family with much fewer available resources. The lightweight solution can be extremely useful for public-key operations that are executed infrequently in a lifetime of long-lasting hardware-based applications, e.g., a key (re-)establishment or firmware upgrade in elevators or avionic systems. The high-performance implementation could be used in HSMs or similar server applications where several connections have to be secured at the same time.

We investigate two decoder variants in our high-performance implementations, an iterative and a parallel design strategy. Encryption performance is 13.7 μs, decryption takes 125.4 μs/82.1 μs. Such a high performance is achieved by storing the QC-MDPC keys and intermediate results directly in FPGA logic, without requiring additional internal or external memory.

Our lightweight implementation of QC-MDPC McEliece for Xilinx FPGAs shows how the comparably small keys and intermediate results can be efficiently stored and accessed in embedded block memories to achieve a low resource consumption while still maintaining a decent performance sufficient for many applications. Since decoding is usually the most expensive operation in code-based cryptosystems, we particularly focus on implementing a lightweight design of the most efficient decoder for QC-MDPC codes according to our evaluations in Chapter 4. We show that QC-MDPC codes allow to implement public-key cryptography with very few resources while still providing excellent efficiency in terms of computational complexity for encryption and decryption on the FPGA.

Furthermore, we present horizontal and vertical side-channel analysis techniques for an implementation of the QC-MDPC McEliece cryptosystem. The target of the side-channel attacks is our lightweight QC-MDPC McEliece decryption FPGA implementation as presented in Section 5.3. The attack consists of a combination of a differential leakage analysis during the syndrome computation followed by an algebraic step that exploits the relation between the public- and private-key and succeeds to recover the complete private-key after a few observed decryptions.

Note that IND-CCA conversions and true random number generation are out of the scope of this chapter. For fair comparison between the two implementations we also implement our lightweight designs on the same Virtex-6 FPGA as the high-performance design.

Outline We present a high-performance implementation of QC-MDPC McEliece for FPGAs in Section 5.2 followed by a lightweight design in Section 5.3. In Section 5.4 we investigate side-channel attacks and countermeasures. A conclusion is drawn in Section 5.5.

5.2 High-Performance QC-MDPC McEliece for FPGAs

The following sections explain our design choices and describe the implementations of QC-MDPC McEliece in reconfigurable hardware. The primary goal of our first design is to provide a high-performance QC-MDPC McEliece public-key encryption core for Xilinx FPGAs.

5.2.1 Design Considerations

Because of their relatively small size, the QC-MDPC McEliece public- and private-keys do not necessarily have to be stored in external memory as needed in earlier FPGA implementations of McEliece and Niederreiter based on binary Goppa codes. Since we aim for high-performance, we keep all operands directly in registers and refrain from loading/storing them from/to internal block memory or other external memory as this would degrade the achievable performance.

Accessing a single 4,801-bit row of the public-key matrix via a 32-bit BRAM interface would consume at least 151 clock cycles. Storing the vector in flip-flops allows access in one clock cycle, leading to a much better performance. If maximum performance is not required, BRAMs significantly reduce the resource consumption as will be shown in Section 5.3.

Furthermore, we do not take the sparsity of the secret polynomials into account in this FPGA design. Using a sparse representation of the secret polynomials would require to implement $w = 90$ 13-bit counters, each indicating the position of a set bit in one of the two secret polynomials. To generate the next row of the private-key, all counters would have to be increased and in case of exceeding r, a counter would need to be reset to 0. If a bit in the ciphertext is set, we would have to generate a 4,801-bit vector from the counters belonging to the corresponding secret polynomial and XOR this vector to the current syndrome. An alternative would be to read out the content of each counter and flip the corresponding bit in the syndrome. These tasks, however, are time- and resource-consuming in hardware.

We base our high-performance QC-MDPC McEliece decryption implementation on decoder $\mathcal{D}_1/\mathcal{D}_2$. The reason for not choosing decoder $\mathcal{D}_3$ is that we sequentially rotate the ciphertext and private-key in every cycle of the bit-flipping iteration. 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. Furthermore, an early termination leaks timing information about the point in time at which the syndrome became zero, which is undesirable as well.

5.2.2 High-Performance FPGA Implementation

Our target device is a Virtex-6 XC6VLX240T FPGA to allow fair comparison with previous work – although all our implementations would fit smaller devices as well. The encryption and decryption units are equipped with a simple I/O interface to decrease its impact on the required FPGA resources. Messages and ciphertexts are sent and received bit-by-bit to reduce the I/O overhead.

QC-MDPC McEliece Encryption

QC-MDPC McEliece encryption requires to implement a vector matrix multiplication to multiply message m with the public-key matrix G. The resulting codeword $c = mG$ is then XORed with an error vector of Hamming weight $wt(e) \leq 84$ to produce the ciphertext $x = c \oplus e$. In QC-MDPC McEliece encryption we are given a 4801-bit public-key g which is the first row of the public matrix G. Rotating g by one bit position yields the next row of G and so forth. Since G is in systematic form, the first half of c is equal to m due to a multiplication with the identity matrix. 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. Implementing this in hardware requires three 4,801-bit registers to store 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 circular shift register which is implemented using shift register LUTs.

QC-MDPC McEliece Decryption

Decryption is performed by decoding the received ciphertext. The plaintext is obtained as the first half of the decoded codeword. We implement bit-flipping decoder $\mathcal{D}_1$ as described in Chapter 4, an algorithmic description is listed in Algorithm 1.

Algorithm 1 Decoding (QC-)MDPC Codes

Input: H, $x = mG + e$, $B = b_0, \ldots, b_{\text{max-1}}$, max
Output: Message m or DecodingFailure

```
Compute syndrome s = Hx^T
for i = 0 → max − 1 do
  for every ciphertext bit j do
    Count unsatisfied parity-check equations #upc = hw(h_j AND s)
    if #upc ≥ b_i then
      Flip ciphertext bit x_j = x_j ⊕ 1
      Update syndrome s = s ⊕ h_j
    end if
  end for
  if s = 0^r then
    return x
  end if
end for
return DecodingFailure
```

First we compute the syndrome $s = Hx^T$ by multiplying the parity-check matrix $H = [H_0 \,|\, H_1]$ with the ciphertext $x = [x_0 \,|\, x_1]$. Given the first 9,602-bit row $h = [h_0 \,|\, h_1]$ of H and the 9,602-bit ciphertext $x = [x_0 \,|\, x_1]$ the syndrome is computed as follows. We sequentially iterate over every bit of the ciphertext 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 test the syndrome for zero which is implemented using a bitwise OR tree. Since the FPGA offers 6-input LUTs, we split the syndrome into 6-bit chunks and compute their bitwise OR on the lowest level of the tree. The results are fed into another level of 6-input LUTs which again compute the bitwise OR of their inputs. This is repeated until we are left with a single bit that indicates if the syndrome is zero or not. In addition, we insert registers after the second level of the tree to minimize the critical path.

Decryption is finished once the syndrome is zero. Otherwise we determine the number of unsatisfied parity-check equations for each row $h = [h_0 \,|\, h_1]$ by computing the Hamming weight of the bitwise AND of the syndrome and h_0 and h_1, respectively. If the Hamming weight exceeds threshold b_i for the current iteration i, the corresponding bit of the ciphertext 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. Rows h_0 and h_1 are rotated by one bit and we repeat until all rows of H have been checked.

There are two options to implement counting the number of unsatisfied parity-check equations for h_0 and h_1 since they are independent of each other. Either we compute the unsatisfied parity-checks of the first and second secret polynomial iteratively or we instantiate two Hamming weight computation units to process the polynomials in parallel. The iterative version is expected to take twice the time using half the resources compared to a parallel implementation. We explore both approaches to evaluate this time/resource trade-off.

Computing the Hamming weight of a 4,801-bit vector efficiently is challenging. Similar to the zero comparator we split the input into 6-bit chunks and determine their Hamming weight using look-up tables. We then compute the overall Hamming weight by building an adder tree with registers on every layer to minimize the critical path and to enable pipelined Hamming weight computations.

The syndrome is again compared to zero after all rows of H and the corresponding changes to the ciphertext and syndrome have been processed. If the syndrome is zero, the first 4,801 bit of the updated ciphertext hold the decoded message m which is returned as the result. Otherwise the next decoding iteration $i+1$ is started with decoding threshold b_{i+1} until either the syndrome becomes zero or the maximum number of iterations is reached.

5.2.3 Implementation Results

All our results are obtained post place-and-route (PAR) for Xilinx Virtex-6 XC6VLX240T FPGAs using Xilinx ISE 14.7. The throughput figures assume an I/O interface capable of these processing speeds is provided.

Our QC-MDPC encoder runs at a maximum clock frequency of 351.7 MHz and encodes a 4,801-bit message in 4,801 clock cycles which results in a throughput of 351.7 Mbit/s. The iterative version of our QC-MDPC decoder runs at 222.5 MHz. The decoding execution time depends on how many decoding iterations for successful decoding are needed. We calculate the average required cycles for iterative decoding as follows: computing the initial syndrome requires 4,801 clock cycles and comparing the syndrome to zero takes 2 clock cycles. For every following bit-flipping iteration we need 9,622 clock cycles and additionally 2 clock cycles for comparing the syndrome to zero. As shown in Table 4.3, decoder $\mathcal{D}_1$ needs 2.4019 bit-flipping iterations on average. Thus, the average cycle count for our iterative decoder is

$$4,801 + 2 + 2.4019 \cdot (9,622 + 2) = 27,918.9 \,\text{cycles}.$$

Our parallel decoder processes both secret polynomials in the bit-flipping step in parallel and runs at 199.3 MHz. We calculate the average cycles as before with the difference that every bit-flipping iteration now takes $4,811 + 2$ clock cycles. Thus, the average cycle count for the parallel decoder is

$$4,801 + 2 + 2.4019 \cdot (4,811 + 2) = 16,363.3 \,\text{cycles}.$$

The parallel decoder operates 35% faster than the iterative version while occupying 6-26% more resources. Compared to the decoders, the encoder runs 6-9 times faster and occupies 2-5 times less resources. Table 5.1 summarizes our results.

Table 5.1: Implementation results of our QC-MDPC McEliece implementations with parameters $n_0 = 2, n = 9,602, r = 4,801$, $w = 90, t = 84$ (80-bit equivalent symmetric security) on a Xilinx Virtex-6 XC6VLX240T FPGA.

Aspect	**Encoder**	**Decoder (iterative)**	**Decoder (parallel)**
FFs	14,429 (4%)	32,962 (10%)	41,714 (13%)
LUTs	9,201 (6%)	36,502 (24%)	42,274 (28%)
Slices	2,924 (7%)	10,364 (27%)	10,988 (29%)
Frequency	351.7 MHz	222.5 MHz	199.3 MHz
Time/Op	13.7 µs	125.4 µs	82.1 µs
Throughput	351.7 Mbit/s	38.3 Mbit/s	58.5 Mbit/s
Encode	4,801 cycles	-	-
Compute Syndrome	-	4,801 cycles	4,801 cycles
Check Zero	-	2 cycles	2 cycles
Flip Bits	-	9,622 cycles	4,811 cycles
Overall average	4,801 cycles	27,918.9 cycles	16,363.3 cycles

Using the formerly proposed decoders without our optimizations (i.e., decoders $\mathcal{A}$ and $\mathcal{B}$) results in much slower decryptions. Decoder $\mathcal{A}$ needs

$$4,803 + 5.2922 \cdot (2 \cdot 9,622 + 4,803) = 132,064.5 \text{ cycles}$$

in an iterative implementation which is nearly five times slower than our iterative decoder $\mathcal{D}_1$. In a parallel implementation decoder $\mathcal{A}$ requires

$$4,803 + 5.2922 \cdot (2 \cdot 4,811 + 4,803) = 81,143.0 \text{ cycles}$$

which again is five times more cycles than our parallel implementation of decoder $\mathcal{D}_1$.

Decoder $\mathcal{B}$ saves cycles by skipping the $\max(\#_{\text{upc}})$ computation but still needs

$$4,803 + 3.0936 \cdot (9,622 + 4,803) = 49,428.2 \text{ cycles}$$

in an iterative and

$$4,803 + 3.0936 \cdot (4,811 + 4,803) = 34,544.9 \text{ cycles}$$

in a parallel implementation which are both outperformed by a factor of two by our implementations of decoder $\mathcal{D}_1$.

Comparison

A comparison with previous FPGA implementations of code-based (McEliece, Niederreiter), lattice-based (Ring-LWE, NTRU), and standard public-key encryption schemes (RSA, ECC)

is given in Table 5.2. The most relevant metric for comparing the performance of public-key encryption schemes depends on the application. For key exchange it is usually the required *time per operation* and for data encryption typically *throughput* is the most interesting metric when multiple input blocks are processed.

A hardware McEliece implementation based on Goppa codes including a CCA2 conversion was presented for Virtex5-LX110T FPGAs in [SWM+09, SWM+10]. Comparing their performance to our implementations shows the advantage of QC-MDPC McEliece in time per operation and throughput. The occupied resources are similar to our resource requirements but in addition 75 block memories are needed whereas we do not require block memories. Even more important for real-world applications is the public-key size. QC-MDPC McEliece requires 0.59 Kbytes which is only a small fraction of the 100.5 Kbytes public-key of [SWM+10].

Another McEliece co-processor was proposed by [GDUV12] for Virtex5-LX110T FPGAs. Their design goal was to optimize the speed/area ratio, while we aim for high-performance. Regarding decoding, our implementations outperform their work in both time/operation and throughput. However, [GDUV12] need fewer resources which allows an implementation on low-cost devices such as Spartan-3 FPGAs. Their public-keys have a size of 63.5 Kbytes which is still much larger than the 0.59 Kbytes of QC-MDPC McEliece.

The Niederreiter public-key scheme was implemented with binary Goppa codes by [HG12] for Virtex6-LX240T FPGAs. Their work shows that Niederreiter encryption can provide high-performance with a moderate amount of resources. Decryption is more expensive in computation time as well as in required resources compared to our work. Their Niederreiter encryption is the superior choice for a minimum time per operation while QC-MDPC McEliece achieves better throughput results. Furthermore, public-keys with a size of 63.5 Kbytes are a tough memory requirement for FPGAs.

FPGA implementations of lattice-based public-key encryption were proposed by [RVM+14, PG14b] for Ring-LWE and by [KY09] for NTRU. The Ring-LWE implementations require 1.5-2 times more time to encrypt a smaller plaintext but they decrypt ciphertexts faster and occupy less resources at the cost of using block RAMs and digital signal processors. For high-throughput applications, QC-MDPC McEliece outperforms both implementations at encryption and decryption. NTRU as implemented by [KY09] provides high-performance at moderate resources requirements. However, the selected parameters for this implementation only achieve a security level of around 64 bits. Note further that the results are reported for an outdated Virtex-E FPGA which is hardly comparable to modern Xilinx Virtex-5/-6 devices.

Efficient ECC hardware implementations for curves over $GF(p)$ and $GF(2^m)$ are [DJJ+06, GP08, RRM12, SRM12] which all yield good performance at moderate resource requirements. The most efficient RSA hardware implementation to date was proposed in [Suz07, SM11]. The time to encrypt and decrypt one block as well as the throughput are considerably worse than QC-MDPC McEliece.

Table 5.2: Performance comparison of our QC-MDPC FPGA implementations with other public-key encryption schemes. [1]Occupied resources and BRAMs are given for a combined encryption and decryption core. [2]Additionally uses 1 DSP48. [3]Additionally uses 26 DSP48s. [4]Additionally uses 17 DSP48s.

Scheme	**Platform**	f **[MHz]**	**Bits**	**Time/Op**	**Cycles**	**Mbit/s**	**FFs**	**LUTs**	**Slices**	**BRAM**
This work (enc)	XC6VLX240T	351.7	4,801	13.7 µs	4,801	351.7	14,429	9,201	2,924	0
This work (dec)	XC6VLX240T	199.3	4,801	82.1 µs	16,363	58.5	41,714	42,274	10,988	0
This work (dec iter.)	XC6VLX240T	222.5	4,801	125.4 µs	27,919	38.3	32,962	36,502	10,364	0
McEliece (enc) [SWM+10]	XC5VLX110T	163	512	500 µs	n/a	1.0	n/a	n/a	14,537	75[1]
McEliece (dec) [SWM+10]	XC5VLX110T	163	512	1,290 µs	n/a	0.4	n/a	n/a	14,537	75[1]
McEliece (dec) [GDUV12]	XC5VLX110T	190	1,751	500 µs	94,249	3.5	n/a	n/a	1,385	5
Niederreiter (enc) [HG12]	XC6VLX240T	300	192	0.66 µs	200	290.9	875	926	315	17
Niederreiter (dec) [HG12]	XC6VLX240T	250	192	58.78 µs	14,500	3.3	12,861	9,409	3,887	9
Ring-LWE (enc) [PG14b]	XC6VLX75T	262	256	26.2 µs	6,861	9.8	3,624	4,549	1,506	12[1,2]
Ring-LWE (enc) [PG14b]	XC6VLX75T	262	256	16.8 µs	4,404	15.2	3,624	4,549	1,506	12[1,2]
Ring-LWE (enc) [RVM+14]	XC6VLX75T	313	256	20.1 µs	6,300	12.7	860	1,349	n/a	2[1]
Ring-LWE (dec) [RVM+14]	XC6VLX75T	313	256	9.1 µs	2,800	28.1	860	1,349	n/a	2[1,2]
NTRU (enc/dec) [KY09]	XCV1600E	62.3	251	1.54/1.41 µs	96/88	163/178	5,160	27,292	14,352	0
ECC-P224 [GP08]	XC4VFX12	487	224	365.10 µs	177,755	0.6	1,892	1,825	1,580	11[3]
ECC-163 [RRM12]	XC5VLX85T	167	163	8.60 µs	1436	18.9	n/a	10,176	3,446	0
ECC-163 [SRM12]	Virtex-4	45.5	163	12.10 µs	552	13.4	n/a	n/a	12,430	0
ECC-163 [DJJ+06]	Virtex-II	128	163	35.75 µs	4576	4.6	n/a	n/a	2251	6
RSA-1024 [SM11]	XC5VLX30T	450	1,024	1,520 µs	684,000	0.7	n/a	n/a	3,237	5[4]

5.3 Lightweight QC-MDPC McEliece for FPGAs

Next we present a lightweight implementation of QC-MDPC McEliece for reconfigurable hardware. The goal of this work is to provide a cost effective public-key encryption engine with low resource requirements while maintaining reasonable performance.

5.3.1 Design Considerations

Intuitively, the comparably small keys of QC-MDPC McEliece should allow for small area footprint implementations. Instead of having to provide 50-100 Kbytes of memory as necessary for binary Goppa codes, the QC-MDPC public-key requires 4801 bits and the private-key 9602 bits. Apart from keys, additional data, such as the message, the ciphertext, and the syndrome, has to be stored and requires memory in the same range.

FPGAs of the Xilinx Spartan-6 and Virtex-6 family are equipped with dual-ported block memories (BRAMs), each capable of storing up to 18/36 Kbits of data. In each clock cycle two separate 32-bit words can be read from two different memory addresses, and it is even possible to write data to a memory cell in the same clock cycle after reading its content in READ_FIRST mode.

Our design of the encryption and decryption unit stores all inputs, outputs, keys and intermediate values in these block memories and processes them in 32-bit blocks to achieve a very compact structure. Below follow our design choices for the encryption and decryption cores in more detail.

QC-MDPC McEliece Encryption

Recall that for QC-MDPC McEliece encryption we have to compute $x = mG \oplus e$ which boils down to an accumulation of the rows of the generator matrix G depending on set bits in the message m and an addition of the error vector e. Hence, we have to hold the message (4801 bits), one row of the generator matrix (4801 bits), and the redundant part (second half of x, 4801 bits) in memory. The error vector e is added on-the-fly and is provided through a 32-bit interface to avoid having to store additional 9602 bits, of which at most 84 are set. In total we have to store $3 \cdot 4801$ bits, fitting one 18-Kbit BRAM. In addition to the available storage space we also have to consider that only two data ports are available for each BRAM. In a straightforward approach we would need three data ports (and thus 2 BRAMs), one for the message, one for the public-key and one for the redundant part.

Since each message bit is accessed only once as opposed to the redundant part and the rows of the public-key which are accessed 4801 times each, we store all of them in one BRAM and spend a 32-bit register to hold the current 32-bit message block which we are processing.

While the encryption unit is idle, it allows external components to access its internal BRAM to read out the encrypted ciphertext, to write a new message and, if desired, to change the public-key. When starting the encryption, the unit takes control of the BRAM and allows outside components to access the BRAM only after the encryption is finished.

QC-MDPC McEliece Decryption

For decryption we have to store the private-key (9602 bits), the received ciphertext (9602 bits), and the syndrome (4801 bits). Decoding is performed in-place, i.e., after the decoder finishes, the first 4801 bits of the decoded ciphertext hold the decrypted message. The private-key and the ciphertext consist of two separate 4801-bit vectors that can either be processed in parallel or iteratively. Since decryption is more complex than encryption we process them in parallel to not further widen the gap between encryption and decryption performance.

Concerning memory, two 18-Kbit BRAMs suffice to store all the necessary values but we have to keep in mind that each BRAM only offers two data ports. Since the private-key and the ciphertext consist of two separate 4801-bit vectors that are processed in parallel, four data ports plus one data port for the syndrome are required. To increase performance at the cost of few additional resources, we include an additional 18-Kbit BRAM to store the syndrome.

The first step during decoding is the syndrome computation. Depending on set ciphertext bits, rows of the two parity-check matrix blocks are accumulated. For comparing the syndrome to zero, we compute the OR of all 32-bit blocks of the syndrome. If the result is zero, the syndrome is zero as well. Counting the number of unsatisfied parity-check equations is done by computing the Hamming weight of the binary AND of the syndrome and the two parts of the private-key in 32-bit steps.

While the decryption unit idles, access to the ciphertext BRAM is granted to allow external components to write new ciphertexts and to read out decrypted plaintexts. External components are not allowed to access the private-key in our design. Depending on the application it might be desired to at least be able to write a new private-key which can be easily accomplished in our design by forwarding the control signals and data lines of the private-key BRAM to external components.

5.3.2 Lightweight FPGA Implementation Details

Next we detail our lightweight implementations of QC-MDPC McEliece en- and decryption based on the design decisions explained in Section 5.3.1. Note that the implementation of an IND-CCA conversion as well as the implementation of a true random number generator are out of the scope of this chapter.

QC-MDPC McEliece Encryption

Encryption usually starts by resetting the redundant part to zero. It then accumulates the rows of the generator matrix depending on the message bits and adds an error vector in the end. Our implementation combines resetting the redundant part and adding the error vector by directly loading the second half of the error vector into the redundant part and accumulating the rows of G to it. We rely on being provided a uniformly distributed error vector of weight at most $t = 84$ through a 32-bit interface.

The most performance-critical operation of the encoder is the rotation of 4801-bit vectors. More precisely, the first row g of the generator matrix has to be rotated 4801 times to iterate

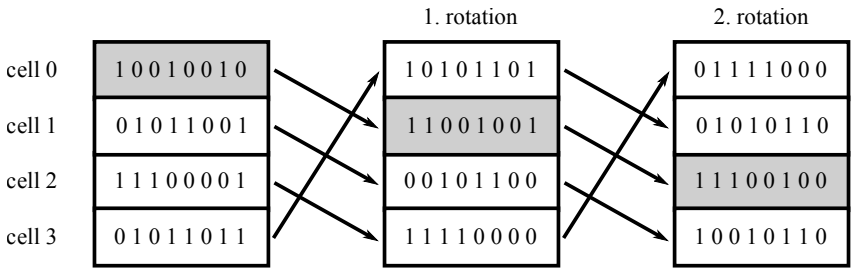


Figure 5.1: Fast vector rotation using the READ_FIRST mode in a Xilinx block RAM with 8-bit registers and four memory cells. Each rotation moves the first 8 bit of the vector (grey cells) to the following memory cell. Rotation is performed to the right.

over all rows of G. In a BRAM-based implementation, each data port can only access 32 bits per clock cycle. Hence, rotating a 4801-bit vector requires to load 152 32-bit cells[1], rotate them by one bit, and store the result.

Two clock cycles would be needed to rotate each 32-bit block in a straightforward approach with one data port. One cycle for loading and rotating the value and another cycle to store the result. When two data ports are used, one data port can be used to read blocks and the second port can be used to write blocks delayed by one clock cycle. This requires one clock cycle to rotate each 32-bit block plus a small overhead for loading the least significant bit and introducing the delay required for storing the results. However, this approach encounters a problem when having to add the current row of the generator matrix to the redundant part. Since both data ports are already occupied, we cannot load the redundant part and XOR the current row to it without spending additional clock cycles.

Instead we implement the following approach that allows to efficiently rotate g and XOR it to the redundant part at the same time if necessary with only two data ports. As described above, Xilinx BRAMs support the READ_FIRST mode which allows to first read the content of a memory cell and then to overwrite the cell with new data in the same clock cycle. After loading the least significant bit, the first 32-bit memory cell of g is read. In the next clock cycle we activate the write signal and store the rotated content of the first cell to the second cell after loading its content. By applying this trick we additionally introduce a rotation of the memory cells. The rotated 32-bit value that was previously stored in memory cell 0 is stored to memory cell 1, the rotated value of memory cell 1 is stored in cell 2, and so on. This requires to wrap the addresses after accessing the last memory cell and to keep track of which memory cell holds the beginning of the rotated vector. After one rotation, the first 32 bits are located in memory cell 1 instead of memory cell 0, after the second rotation the first 32 bits are located in cell 2, and so on. An example of this rotation technique is illustrated in Figure 5.1 for a block RAM with 8-bit registers and a total of four memory cells. This technique allows us to occupy only one data port of the BRAM while still being able to efficiently rotate a 4801-bit vector using just 153 clock cycles instead of nearly twice as many cycles with the previous approach.

We apply the same trick to the redundant part even though it does not need to be rotated. This allows us to load a 32-bit block of the redundant part, XOR the corresponding 32-bit block

[1]Rotating a 4801-bit vector that is stored in 32-bit cells requires $\lceil 4801/32 \rceil = 151$ loads plus one additional load to extract the least significant bit.

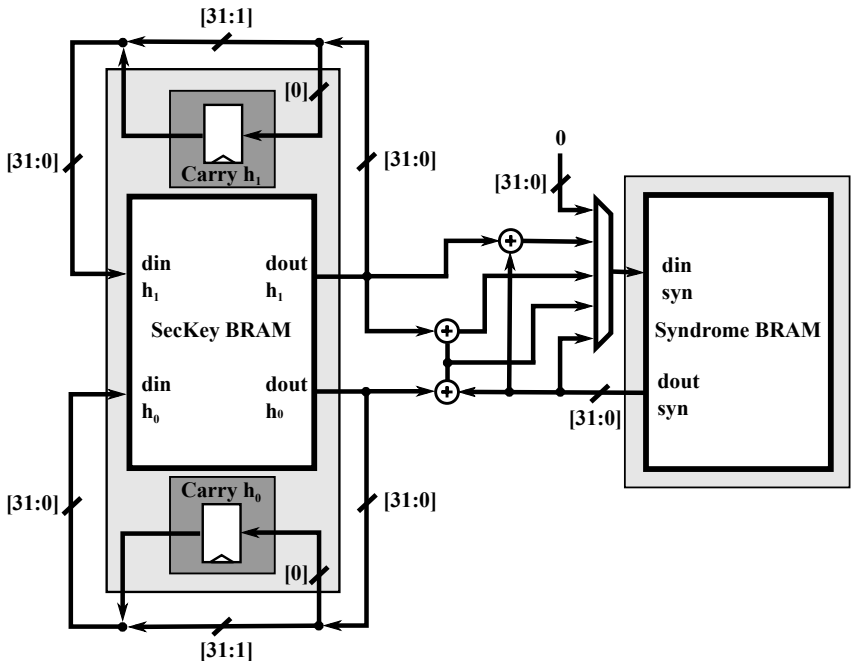


Figure 5.2: Block diagram of the syndrome computation circuit. Depending on set bits in the ciphertext, rows of both blocks of the private-key are XORed to the syndrome in 32-bit steps.

of g to it if the current message bit is set, and store the result while rotating g at the same time. Both operations can work in parallel since they only need one data port each.

After 32 rotations of row g, we XOR the current 32-bit message block with its corresponding 32-bit block of the error vector and store the result. Then we load the next 32-bit message block to a 32-bit register and repeat until all message bits are processed. The resulting ciphertext can be read out from the BRAM by external components once decoding is finished.

QC-MDPC McEliece Decryption

Decryption first computes the syndrome of the received ciphertext. After resetting the syndrome to zero, we rotate both parts of the private-key using the same trick as for rotating the public-key when encrypting. Similarly, we apply the same trick to the syndrome that we applied to the redundant part. The syndrome itself does not need to be rotated, but we benefit from the same performance gains when adding one or even both rows of the private-key to the syndrome as when adding one row of the generator matrix to the redundant part during encryption. Due to the similar structure of the syndrome computation and the encryption of a message both take nearly the same amount of clock cycles to finish. The computation would take twice as long if we would not process both parts of the private-key and the ciphertext in parallel. Figure 5.2 illustrates our syndrome computation circuit.

Testing the syndrome for zero is implemented by computing the binary OR of all 32-bit blocks of the syndrome and comparing the results to zero. To count the number of unsatisfied

parity-check equations for a ciphertext bit, we load 32 bits of the syndrome and 32 bits of the current rows of the parity-check matrix blocks and compute their binary AND. The Hamming weight of the result determines if the corresponding ciphertext bits have to be inverted. The Hamming weight is computed by splitting the 32-bit AND result into five 6-bit chunks and one 2-bit chunk, looking up their Hamming weight from tables and accumulating the results. We proceed with the following 32-bit blocks and compute the overall Hamming weights for two ciphertext bits in parallel.

Next we reload the current rows of the parity-check matrix blocks and rotate them using our previously described rotation technique. If one or two ciphertext bits caused more than b_i unsatisfied parity-check equations for the current iteration i, we invert the ciphertext bit(s) and XOR one or two rows of the parity-check matrix block to the syndrome while rotating them.

After processing $2 \cdot 32$ ciphertext bits, we store both modified parts of the ciphertext back to the BRAM and load the next 32-bit blocks to two 32-bit registers. After processing the last ciphertext bit, we again compute the binary OR of all 32-bit blocks of the syndrome and check if the result is zero. If it is we notify external components that the plaintext can now be read out, otherwise we repeat the bit-flipping decoding with adapted thresholds or signal a decoding error if the maximum number of iterations is exceeded.

5.3.3 Implementation Results

We present our implementation results in terms of occupied resources and performance for Xilinx FPGAs. Furthermore, we compare our results with the high-performance QC-MDPC McEliece FPGA implementation presented in Section 5.2 and with previous work.

The implementation results are obtained post place-and-route (PAR) and are listed in Table 5.3 for a low-cost Xilinx Spartan-6 XC6SLX4 (the smallest device in the Spartan-6 family) and for a high-end Xilinx Virtex-6 XC6VLX240T FPGA using Xilinx ISE 14.7. The encoder occupies 64-68 slices and the decoder 148-159 slices on these devices. As detailed in Section 5.3.1, the encoder uses one BRAM and the decoder uses three BRAMs to store inputs, outputs, and intermediate values. While the resource consumption is similar on both FPGAs, the design naturally runs at higher clock frequencies on the Virtex-6 FPGA.

To encrypt a message, the cycle counts listed in Table 5.4 are required. First 151 cycles are needed to load the second half of the error vector into the redundant part. Rotating g and XORing it to the redundant part if the current message bit is set takes 153 cycles and has to be repeated 4801 times. After processing 32 message bits we load the next 32-bit message block and store the previous message XORed with the corresponding 32 bits of the error vector which takes 3 cycles and has to be repeated 151 times. Finally, we store the least significant bit of the redundant part which takes one additional clock cycle. Overall,

$$151 + 4801 \cdot 153 + 151 \cdot 3 + 1 = 735,158 \text{ cycles}$$

are needed to encrypt a 4801-bit message block. This translates to 2.2 ms on the Virtex-6 FPGA and to 3.4 ms on the Spartan-6 FPGA.

Decrypting a ciphertext requires cycle counts as listed in Table 5.4. Resetting the syndrome finishes after 151 cycles. Computing the syndrome is basically the same operation as encoding

Table 5.3: Resource consumption of our lightweight QC-MDPC McEliece implementations on a low-cost Xilinx Spartan-6 XC6SLX4 and on a high-end Xilinx Virtex-6 XC6VLX240T FPGA. All results are obtained post place-and-route.

	Virtex-6 XC6VLX240T		**Spartan-6 XC6SLX4**	
Aspect	**Encryption**	**Decryption**	**Encryption**	**Decryption**
FFs	120	412	119	413
LUTs	224	568	226	605
Slices	68	148	64	159
BRAM	1	3	1	3
Frequency	334 MHz	318 MHz	213 MHz	186 MHz
Time/Op	2.2 ms	13.4 ms	3.4 ms	23.0 ms

a message. It takes 153 cycles to rotate both parts of the private-key by one bit and optionally XORing them to the syndrome which is repeated 4801 times. Loading the next two 32-bit ciphertext blocks requires one cycle and is repeated 151 times. Overall,

$$151 + 4801 \cdot 153 + 151 = 734,855 \text{ cycles}$$

are needed to compute the syndrome. Comparing the syndrome to zero takes 151 cycles. Counting the number of unsatisfied parity-check equations, i.e., computing the Hamming weight of the binary AND of the syndrome and the two current rows of the parity-check matrix blocks, takes 154 cycles and is repeated 4801 times. Loading the next two 32-bit ciphertext blocks takes 2 cycles and is repeated 151 times. After computing the Hamming weight, generating the next row of the parity-check matrix takes 153 cycles, which is also repeated 4801 times. Storing modified ciphertext blocks takes one cycle and is done 151 times before the next two 32-bit ciphertext blocks are loaded. Finally, the syndrome is again compared to zero. In summary, one iteration of the bit-flipping step takes

$$151 \cdot 2 + 4801 \cdot 154 + 4801 \cdot 153 + 151 + 151 = 1,474,511 \text{ cycles.}$$

As evaluated in Chapter 4, on average 2.4 decoding iterations are needed for successful decoding. Hence, our overall average cycle count is

$$151 + 734,855 + 151 + 2.4 \cdot 1,474,511 = 4,273,983 \text{ cycles.}$$

The design can be clocked at 318 MHz on the Virtex-6 FPGA which translates to 13.4 ms. On the Spartan-6 FPGA the design runs at 186 MHz which results in 23 ms for decrypting one message block.

Comparison

A comparison of our lightweight implementation with our high-performance implementation of QC-MDPC McEliece and other lightweight code-based FPGA implementations as well as lightweight Ring-LWE and RSA implementations is presented in Table 5.5.

Table 5.4: Required cycles for our lightweight QC-MDPC McEliece en-/decryption cores.

Encoder Operations	**Cycles**	**Decoder Operations**	**Cycles**
Load error vector	151	Reset syndrome	151
Rotate PK & XOR	153	Compute syndrome	734,704
Store & load message	3	Check syndrome	151
		Correct ciphertext bits	1,474,511
Overall average	735,000	Overall average	4,274,000

A fair comparison between the high-performance and the lightweight QC-MDPC McEliece implementations is difficult since the implementations aim for very different goals. When comparing the occupied resources it is fair to say that the lightweight goal was achieved by requiring less than 250 slices and four BRAMs for a combined en-/decryption core instead of using around 13,000 slices which allows to use much smaller and less expensive devices. As expected, the lightweight implementation is outperformed in terms of time per operation, but still provides timings in the range of a few milliseconds which seems reasonable for a large number of real-world applications.

Previous lightweight McEliece implementations [EGHP09, GDUV12] are based on Goppa codes. The first lightweight implementation of a code-based cryptosystem ("MicroEliece") was proposed for a Xilinx Spartan-3 FPGA. Since the storage capacity of the FPGA did not suffice, external memory had to be used to store the public-key. More recently, [GDUV12] proposed a lightweight McEliece decryption co-processor for Xilinx Spartan-3 and Virtex-5 FPGAs. When comparing previous work to our results it is important to keep in mind that even though all works implement McEliece, they are based on different codes. Decoding Goppa codes requires decoders which are very different from (QC-)MDPC decoders.

Our implementation uses less resources and performs at about the same speed compared to [EGHP09]. However, a direct comparison of the consumed resources is difficult since Spartan-3 FPGAs only offer 4-input LUTs as opposed to Spartan-6/Virtex-6 devices which offer 6-input LUTs. The structure of a slice has changed as well, newer Xilinx FPGAs offer more resources with each slice. But even when reducing the LUT and slice count of MicroEliece by 50%, our implementations are still smaller, especially when comparing decryption.

We need around nine times less slices in our implementation compared to [GDUV12], but also more time to decrypt. The resource consumption can be compared more or less directly since Virtex-5 and Virtex-6 FPGAs offer similar resources. Besides resource consumption and efficiency an important criterion for real-world applications is the size of the public-key. Here, the quasi-cyclic structure of QC-MDPC codes shows its advantage by reducing the public-key from 63.5 Kbytes [GDUV12] or even 437.8 Kbytes [EGHP09] to just 0.6 Kbytes.

A lightweight implementation of the lattice-based Ring-LWE scheme was recently presented in [PG14a] for a Spartan-6 XC6SLX9 FPGA. Their encryption core requires around 50% more resources but takes less time per operation. Since Ring-LWE decryption does not require complex decoding, its implementation requires fewer resources and less time to complete.

Table 5.5: Performance comparison of our lightweight QC-MDPC McEliece (McE) implementations with other lightweight public-key encryption implementations. For comparison with the high-performance QC-MDPC McEliece the iterative decryption implementation results are used. [1]Additionally uses a DSP48 block.

Scheme	Platform	Time/Op	FFs	LUTs	Slices	BRAM
Lightweight McE (enc)	XC6SLX4	3.4 ms	119	226	64	1
Lightweight McE (dec)	XC6SLX4	23.0 ms	413	605	159	3
Lightweight McE (enc)	XC6VLX240T	2.2 ms	120	224	68	1
Lightweight McE (dec)	XC6VLX240T	13.4 ms	412	568	148	3
High-performance McE (enc)	XC6VLX240T	13.7 µs	14,429	9,201	2,924	0
High-performance McE (dec)	XC6VLX240T	125.4 µs	32,962	36,502	10,364	0
McEliece [EGHP09] (enc)	XC3S1400AN	2.2 ms	804	1,044	668	3
McEliece [EGHP09] (dec)	XC3S1400AN	21.6 ms	8,977	22,034	11,218	20
McEliece [GDUV12] (dec)	XC5VLX110T	0.5 ms	n/a	n/a	1,385	5
McEliece [GDUV12] (dec)	XC3S1400AN	1.02 ms	2,505	4,878	2,979	5
Ring-LWE [PG14a] (enc)	XC6SLX9	0.9 ms	238	317	95	2[1]
Ring-LWE [PG14a] (dec)	XC6SLX9	0.4 ms	87	112	32	1[1]
RSA (Tiny32) [Hel15a]	Spartan6-3	312 ms	n/a	n/a	142	1
ECC-P233 [HB10]	XC3S50	520 ms	244	578	452	4

Helion Inc. offers a lightweight modular exponentiation core capable of performing 1024-bit RSA operations (Tiny32) [Hel15a]. They report a time/operation of 312 ms at a resource consumption of 142 slices plus one 18-Kbit BRAM on a Spartan-6 device.

A resource-efficient implementation of elliptic curve cryptography was presented in [HB10]. The resource requirements are similar to QC-MDPC McEliece but their performance is a factor of 20-150 slower. If their design would be implemented for a newer device, e.g., a Spartan-6 instead of a Spartan-3, the efficiency would presumably be improved, but usually these improvements are of a small factor.

5.4 Side-Channel Attacks and Countermeasures

In this section we are not concerned with the security of the specific QC-MDPC parameters against underlying theoretical problems but instead focus on *side-channel attacks*. Even in a post-quantum world, i. e., when scalable quantum computers are available, implementation-specific information leakage will remain a serious practical issue. So far no differential side-channel analysis such as DPA has been documented on FPGA implementations of McEliece. In fact, [HMP10] concluded that a classical DPA attack is not possible for their FPGA target implementations of McEliece with binary Goppa codes. We demonstrate that DPA can be a realistic threat for a state-of-the-art FPGA implementation of QC-MDPC McEliece and present

a horizontal and a vertical side-channel attack exploiting slightly different leakages during the syndrome computation step of the decryption implementation. The found attacks show that side-channel leakage can be efficiently exploited even if straightforward methods that work well on contemporary ciphers such as AES and RSA seem inapplicable. Hence, claims on 'free' side-channel resistance should be treated with caution. Besides showing that significant parts of the private-key can be recovered by side-channel analysis, we show that knowledge of the public-key can be utilized to recover missing key information or to correct remaining errors in hypothesized key bits. On the conceptual side it deserves to be noted that our cryptanalysis targets the decoding algorithm, and thus is not restricted to the original QC-MDPC McEliece as presented in Section 3.2.3. Our side-channel attacks are not prevented if the basic scheme is augmented with a common padding to establish stronger provable guarantees, e.g., the aforementioned IND-CCA conversions, as long as the decryption algorithm is applied to the ciphertext directly, possibly followed by some plausibility checks.

The author would like to note that the QC-MDPC McEliece side-channel attacks and countermeasures presented in this section were mainly developed by Cong Chen, Thomas Eisenbarth and Rainer Steinwandt. The author contributed to the research and co-authored the resulting publications which appeared in [CEvMS15, CEvMS16b, CEvMS16a] but does not claim the presented ideas and attacks as his own. The results are included in this thesis for sake of completeness.

5.4.1 Related Work

Side-channel leakages of McEliece have first been studied in [STM+08]. This work, as well as two follow-up studies focused on analyzing timing behavior of different parts of PC implementations of McEliece [SSMS10, Str10]. Subsequently, [AHPT11] improved over prior results, presented countermeasures and pointed out leakages in the preprocessing steps of McEliece encryption. [HMP10] performed power analysis on software implementations of classic McEliece implementations. Their work relies on simple power analysis (SPA)-based approaches, which usually do not translate well into hardware implementations, due to the increased parallel processing of data and a much smaller side-channel leakage. They also show that side-channel analysis is impeded by the large key sizes of McEliece. AVR and ARM microcontroller implementations of QC-MDPC McEliece are shown to be susceptible to SPA attacks in Section 6.3. The found weaknesses rely on secret dependent branches, which allow to recover the encrypted message as well as to recover the private key.

The conference version of this work [CEvMS15] introduced a horizontal DPA attack on our lightweight FPGA implementation of QC-MDPC McEliece. In [CEvMS16a] we introduced a novel vertical DPA that targets the leakage of the syndrome computation. While the vertical attack is less efficient than the horizontal attack (more traces are needed for full key recovery), it is less specific to the implementation and is more difficult to prevent.

5.4.2 Side-Channel Attack on QC-MDPC McEliece Encryption

Usually DPA attacks exploit an intermediate state $y = f(x, k)$ that is a function of a known data item x and a subkey k. The subkey space $\mathcal{K}$ should be small enough so that a hypothesis y

can be checked for all candidates $k \in \mathcal{K}$. Some works that elaborate on this model are [MOP07, KJJR11, WOS14]. McEliece does not offer itself for this approach, as also noted in [HMP10]. One would expect the syndrome s to serve as a potential predictable intermediate state y. However, the bits in the ciphertext x only determine which rows of the parity check matrix H are added to s, where H is the private key to be recovered. Predicting (parts of) the syndrome s requires an additional key bit hypothesis for each variation of each bit of s, i. e., each bit of s depends on l key bits after l variations, supporting the infeasibility claim of [HMP10]. A way of avoiding the exponential growth of key dependencies for each bit of the syndrome state are chosen ciphertexts of low weight. This approach is elaborated in Section 5.4.2. One of the strengths of QC-MDPC, its small private key size, stems from the fact that secret information is highly redundant: each row of H contains the same information—namely $\langle h_0 \ggg z || h_1 \ggg z \rangle$—only rotated by one bit per row, $z \in \{0, 4800\}$. This redundancy allows for an efficient recovery of key information. More important, it enables a *differential* analysis approach which greatly enhances the visibility of even faint leakages. Since the key information is reused over and over again even within the same decryption operation, the algorithm and its implementation enable what has been described as *horizontal* side-channel analysis, e.g. in the framework of [BJPW13]. Horizontal side-channel analysis has the advantage that it can utilize several leakages of the same intermediate sensitive variable from a single decryption operation, making the resulting attack potentially orders of magnitude more efficient than classical DPA attacks, usually classifiable as *vertical* side-channel analysis.

We exploit two different types of leakage, both occurring during syndrome computation. The first analysis recovers key leakage from the syndrome computation itself and requires chosen ciphertexts of low Hamming weight. It resembles classical DPA more closely and, as it only exploits one leakage sample per measurement, can be classified as a vertical side-channel analysis. The second analysis recovers a static key leakage of the key rotation operation that is completely independent of the known or chosen ciphertext input x. Since the exploited leakage occurs several times during one syndrome computation, our attack combines these leakage events, as commonly done in horizontal side-channel attacks.

Leakage Behavior

Recall that the lightweight FPGA implementation stores inputs, outputs and most intermediate values during encryption and decryption in block memories. Decryption uses three BRAMs, one BRAM stores the $2 \cdot 4801$-bit private key, one BRAM stores the $2 \cdot 4801$-bit ciphertext, and one BRAM stores the 4801-bit syndrome. Each BRAM is dual-ported and allows to read/write two 32-bit values at different addresses in one clock cycle. To compute the syndrome, set bits in the ciphertext select rows of the parity-check matrix blocks that are accumulated. Since only one row of each block is stored in the BRAM, they need to be rotated by one bit to generate the next rows. To generate all rows of H, the rotation is repeated 4801 times.

Rotating the two parts of the private key is implemented in parallel, which means that the 4801-bit rows of the first and the second part of the parity-check matrix are rotated at the same time. Efficient rotation is realized using the READ_FIRST mode of Xilinx's BRAMs which allows to read the content of a 32-bit memory cell and then to overwrite it with a new value, all within one clock cycle. The key rotation is implemented as follows: in the first clock cycle, the

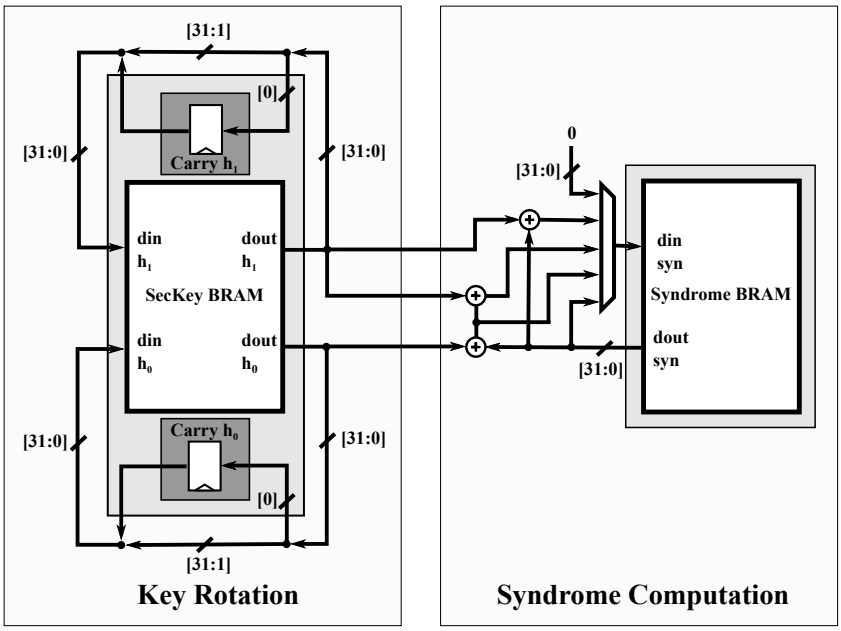


Figure 5.3: Abstract block diagram of the QC-MDPC McEliece syndrome computation circuit including key rotation as implemented in our lightweight FPGA design.

least significant bit (LSB) is loaded from the last memory cell. The first 32-bit of the row to be rotated are loaded next. In all following clock cycles, the succeeding 32-bit blocks of the row are read and overwritten by the rotated preceding 32-bit block. The LSB of each 32-bit block is delayed by a flip-flop and becomes the most significant bit (MSB) of the following block. An abstraction of this implementation is depicted in Figure 5.3. In addition to a rotation of the rows, this introduces a rotation of the memory cells. After one 4801-bit rotation, the most significant 32 bits of a parity-check matrix row do not reside in memory cell 0 but in memory cell 1. The syndrome s is computed by processing the ciphertext x in a bitwise fashion. If the j-th bit is set, i. e., $x_j = 1$, then the j-th row of H is added to the syndrome s. The implementation adds two 32-bit words in parallel: one word of the rotated h_0 and one word of h_1 are processed in each clock cycle.

The described attacks recover the key during the syndrome computation step of the decryption algorithm. The key for QC-MDPC consists of a single line of the parity check matrix H, namely $h_0||h_1$. Only this line of H, or one of its rotated versions $\langle h_0 \ggg z||h_1 \ggg z\rangle$, is stored in BRAM. The key has some noteworthy features that influence the derived DPA attacks. First, the private key is of *low weight*: both parts of the private key h_0 and h_1 are of low Hamming weight such that, $\mathrm{wt}(h_0||h_1) = w$. For the target implementation, $w = 90$ and $\mathrm{wt}(h_i) = 45$, i. e., both h_0 and h_1 have exactly 45 bits set. This means, each key bit $h_{i,j} \in \{0, 1\}$ where $i \in \{0, 1\}$ and $j \in \{0, 4800\}$ is set with probability $\Pr(h_{i,j} = 1) = w/(n_0 r) = 45/4801 \approx .94\%$. This implies *low-weight leakages*: Syndrome and key parts h_i are stored in BRAMs and are processed as 151 32-bit words. The chance of a 32-bit key word to be all-0 is still 74%, about 22% contain a single one bit, leaving the chance of having more than one bit set in a word below 5%.

The critical parts of the target implementation that feature exploitable key leakage are depicted in Figure 5.3. There are two operations that contribute to the leakage during syndrome computation. One operation is the *key rotation* (left part of Figure 5.3), which is always performed. The second operation is the *syndrome computation* (right part of Figure 5.3).

Leakage of the Key Rotation The key rotation is always performed and thus is independent of the ciphertext input x. The stored key row $\langle h_0 \ggg z || h_1 \ggg z \rangle$ is constantly rotated during the syndrome generation. In fact, it is rotated by a single bit 4801 times, where each rotation takes 151 clock cycles (plus two additional clock cycles for preprocessing and a data read-write delay, resulting in 153 clock cycles). The implementation features a separate register which stores the carry bit during rotations. In each of these clock cycles, one bit $h_{i,j}$—the LSB of the last accessed word—is written to the carry register, causing leakage $\lambda_{\text{carry}}(i,j)$. In the following clock cycle, that bit is overwritten with the LSB of the next word, $h_{i,j+32}$. Assuming a Hamming distance leakage function, this register leaks first

$$\lambda_{\text{carry}}(i,j) = w_1 \cdot \text{wt}(h_{i,j-32} \oplus h_{i,j}), \tag{5.1}$$

then, in the subsequent clock cycle, leaks $\lambda_{\text{carry}}(i, j+32) = w_1 \cdot \text{wt}(h_{i,j} \oplus h_{i,j+32})$, where $w_1 \in \mathbb{R}$ is an appropriate weight. Assuming that $h_{i,j} = 1$ and further $h_{i,j\pm32} = 0$, $\lambda_{\text{carry}}(i,j)$ gives a clearly distinguishable leakage from the case where $h_{i,j} = 0$. This leakage is the target of the described attack.

In addition to the leakage of the carry register $\lambda_{\text{carry}}(i,j)$ described in Equation (5.1), there are related leakages happening in the same clock cycles. In fact, when $h_{i,j}$ is written to the carry register, the implementation also reads the word $\langle h_{i,j+1} \dots h_{i,j+32} \rangle$ from the block memory at one address and then stores the word $\langle h_{i,j-32} \dots h_{i,j-1} \rangle$ into the block memory at the same address. Both reading and storing operations will cause leakages at different levels. Assuming a Hamming weight leakage function here, reading data and storing data words leaks as

$$\begin{aligned} \lambda_{\text{read}}(i,j) &= w_2 \cdot \text{wt}(\langle h_{i,j+1} \dots h_{i,j+32} \rangle) \text{ and} \\ \lambda_{\text{store}}(i,j) &= w_3 \cdot \text{wt}(\langle h_{i,j-32} \dots h_{i,j-1} \rangle), \end{aligned}$$

respectively. Here, $w_2 \in \mathbb{R}$ and $w_3 \in \mathbb{R}$ are appropriate weights for the different types of operations. The overall observed leakage of the *key rotation* is thus approximated as:

$$\mathcal{L}_i(j) = \lambda_{\text{carry}}(i,j) + \lambda_{\text{read}}(i,j) + \lambda_{\text{store}}(i,j) + \mathcal{N}$$

where $\mathcal{L}_i$ is the overall leakage at the clock cycle where $h_{i,j}$ is written into the carry register and $\mathcal{N}$ is noise, which is assumed to be Gaussian. Note that the target implementation processes h_0 and h_1 in parallel. This means that the leakage functions $\mathcal{L}_0$ and $\mathcal{L}_1$ for h_0 and h_1 overlap. There are two carry registers (cf. Figure 5.3), one stores $h_{0,j}$ when the other stores $h_{1,j}$. While these leakages slightly differ, we will not attempt to distinguish them. Instead we recover the combined leakages. That is, we predict the combined leakage $h_\Sigma = h_0 + h_1$, which is still sparse. Note that the addition here is *not* in $\mathbb{F}_2$, i. e., we can distinguish the case where $h_{0,j} = h_{1,j} = 1$ from the case $h_{0,j} = h_{1,j} = 0$, although this case is very rare (and will be ignored in the further description). While the model is not perfect, it describes the observed leakages well enough to base a decent key recovery on it.

We can now hypothesize the value of each key bit $h_{i,j}$ separately. We further know at which clock cycle the leakage of the carry registers (for the key rotation) occurs. Since this happens several times during the syndrome computation step of each decryption, one can build a horizontal side-channel attack, as described in Section 5.4.2.

Leakage of the Syndrome Computation Besides the key rotation, the computation of the syndrome s contributes significantly to the leakage. The target implementation processes the ciphertext x in a bitwise fashion. If the i-th bit is set, i. e., $x_i = 1$, then the i-th row of H is added to the syndrome s. The implementation can add two 32-bit words in parallel: one word of the rotated h_0 and one word of h_1 are processed each clock cycle. This means that the addition of one row of H takes 151 clock cycles (plus two additional clock cycles for preprocessing and data read-write delay, resulting again in 153 clock cycles). The syndrome s is initially zero and is only updated if at least one of the currently processed ciphertext bits x_i is set. For the first set bit $x_i = 1$, the zeroed syndrome s is overwritten with (a shifted version of) h_0 or h_1. The key bit $h_{i,j}$ is processed as part of one 32-bit word $\langle h_{i,j-l} \dots h_{i,j} \dots h_{i,j-l+31} \rangle$, where $l \in \{0, \dots, 31\}$ depends on j and the position of the set bit in x. Assuming a Hamming distance leakage, the Hamming weight of the word will leak, since it overwrites a zeroed register, i. e., the leakage of the corresponding syndrome word can be modeled as

$$\lambda_{j,\text{syn}} = w_0 \cdot \text{wt}\left(\langle h_{i,j-l} \dots h_{i,j} \dots h_{i,j-l+31} \rangle\right)$$

with an appropriate weight $w_0 \in \mathbb{R}$. Note that this leakage model is specific to the first key addition to the syndrome state s.

One problem of exploiting this leakage is caused by correlated leakages from the key rotation. Both h_0 and h_1 are rotated during the above computation, with the same key words being processed in the studied clock cycle, as described above. Since those leakages are dependent on the predicted bit, they are not independent noise that decreases by averaging, as usually happening in DPA. However, these leakages $\mathcal{L}_i(j)$ occur independently of whether the syndrome is updated or not. It is possible to remove these *constant* leakages, i.e., all leakages that occur independently of whether the syndrome is updated or not, by simply subtracting the average leakage during the corresponding clock cycles. These are the leakage of the same clock cycles when the key word is not added to the syndrome word (and the set bit in x is zero), which we refer to as $\lambda_{j,\text{const}}$. the resulting leakage observed when $h_{i,j}$ is added to the syndrome is:

$$\mathcal{L}_{j,\text{syn}} = \lambda_{j,\text{syn}} + \lambda_{j,\text{const}} + \mathcal{N}, \tag{5.2}$$

where $\mathcal{N}$ is the noise, which is assumed to be Gaussian and can be minimized by increasing the number of observations used for computing $\mathcal{L}_{j,\text{syn}}$. We know for each key bit $h_{i,j}$ at which clock cycle it is processed[2]. In fact, knowing the implementation and x, it is predictable which 32-bit word of h_i is added to the syndrome at which point in time, just as it is predictable which key bit h_i enters the carry register in which clock cycle for the key rotation.

The other disadvantage of this leakage function is that bits of h_i located close to each other have highly correlated leakage functions. In fact, since 32-bit registers are leaking, all bits in

[2]If not, several hypotheses can be checked in parallel by analyzing neighboring clock cycles, as long as the processing order is deterministic.

the same register will enter the leakage function in the same way. We will later show how this second problem can be solved. We use the leakage of the syndrome computation $\lambda_{j,\text{syn}}$ to build a vertical differential power analysis attack and hypothesize each key bit $h_{i,j}$ separately to be one, knowing that this hypothesis will be wrong 99% of the time. Based on this knowledge, one can build the following attack.

Vertical DPA of Syndrome Computation

The vertical power analysis attack targets the leakage of the syndrome during its computation. This analysis assumes the adversary sends chosen ciphertexts of weight one, i. e., all possible x such that $\text{wt}(x) = 1$. Ciphertexts of weight one ensure that a rotated version of either h_0 or of h_1 is written into a zeroed syndrome s. To recover h_0, we chose only the first 4801 bits of x to be one, yielding a total of 4801 different ciphertexts for the analysis. As detailed in Section 5.4.4, once h_0 is known the remaining part of the private key can be derived easily.

For each x we further know when a line of the key is added to the syndrome. We also know at which clock cycle during that addition the word containing $h_{i,j}$ is added. Our algorithm recovers the clock cycle where the $h_{i,j}$ is added to s for each x and the corresponding leakage in the leakage trace L. Next, we simply sum all the leakage instances of the target $h_{i,j}$ for the different x_i into a bin, as typically done by DPA. Unlike DPA, we have only one bin per key bit. However, assuming that each bit leaks similarly, we have 4756 bins that correspond to a $h_{i,j} = 0$, and only 45 bins corresponding to a bit $h_{i,j} = 1$.

Based on the leakage model derived in Equation (5.2), we can compute a *differential trace* $\Delta_{\text{syn}}(j)$ representing the syndrome leakage of each bit $h_{i,j}$. We can approximate $\lambda_{j,\text{const}}$ by simply averaging over all observed traces and compute it as $\mathcal{L}_{j,\text{const}} = \text{avg}(L_j)$. This average is then subtracted from the leakage trace for $\mathcal{L}_{j,\text{syn}}$, which is computed as

$$\Delta_{\text{syn}}(j) = \sum_{l=0}^{4800} \left(\mathcal{L}_{j,\text{syn}}(l) - \mathcal{L}_{j,\text{const}}(l)\right). \tag{5.3}$$

The resulting differential trace $\Delta_{\text{syn}}(j)$ is depicted in Figure 5.4, where the red (gray) line depicts the observed leakage while the blue (black) line depicts the leakage derived from the model as described above. From the plot as well as the model it can be observed that bits of h_i located close to each other have highly correlated leakage functions. In fact, since 32-bit registers are leaking, all bits in the same register will enter the leakage function in the same way. However, whether a given neighboring bit is in the same register depends on the row index that is currently processed, since the key bits are rotated by one bit for each row. This means that the neighboring bits will leak in a different clock cycle eventually, as the position of the set bit in x changes for different ciphertexts. The closer the bit is to the correct bit, the higher their correlation is (since they are more likely to be in the same register). We will later show that, while key bits equal to one can be detected, their exact position is harder to detect, since neighboring bits "look like" ones as well.

The plot of the differential trace in Figure 5.4 shows the highest consumption for the correct key bits. The consumption decreases linearly as the distance to the bit increases, at least for

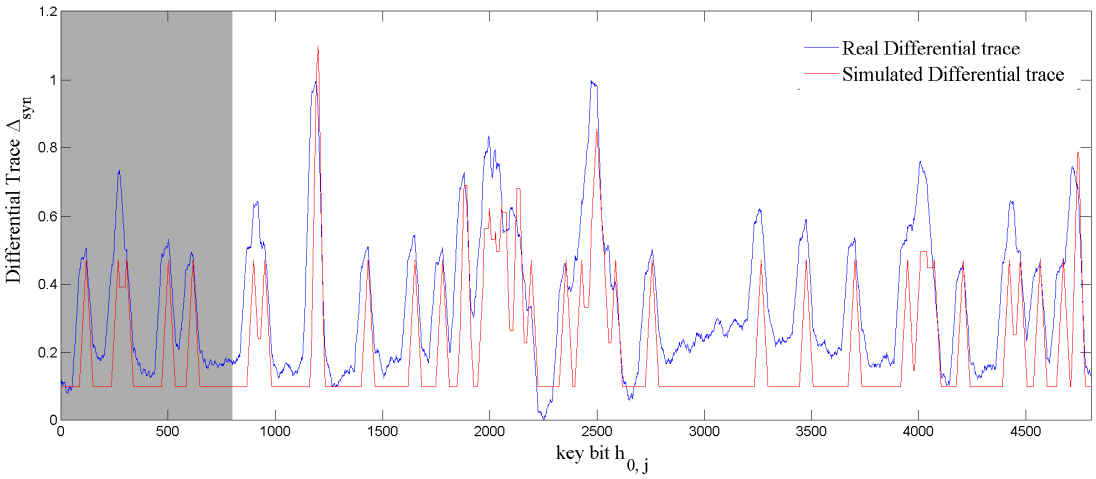


Figure 5.4: Differential leakage for syndrome computation with key part h_0 only. The plot shows the normalized leakage (vertical axis) for each key bit of h_0 (horizontal axis) for simulated leakage according to $\lambda_{j,\text{syn}}$ (blue/black line) and real measurement, i. e., empirical $\Delta_{\text{syn}}(j)$ (red/gray line). Due to correlation in the leakage of closely located bits, the shapes overlap on several positions.

key bits with a higher index. Bits at least 32 positions away from a set key bit show the lowest consumption, since they never share a leakage with a set bit. However, from the magnified version depicted in Figure 5.5 it can be seen that there is still a correlated leakage occurring that is not caught by our model. In fact, bits up to 64 bits lower than the predicted one still exhibit a correlation. We assume this to be due to the READ_FIRST mode of the BRAM. In fact, when a specific syndrome word is written to BRAM, the next one is simultaneously read, as is the corresponding part of the key. Hence, the next clock cycle's word could already be computed. While we expect this leakage to be constant, i. e., to occur independently of whether the syndrome will be updated or not, the observed leakage suggests otherwise.

In summary, the described method lets us detect leakages of h_0 and h_1 separately. It allows us to reliably distinguish set bits from zero bits. We get a single leakage observation per trace L for chosen ciphertexts of weight one. However, closely co-located bits are highly correlated, making the exact position of a bit difficult to detect.

Horizontal DPA of Key Rotation

As mentioned above, we cannot distinguish $h_{0,j}$ and $h_{1,j}$ for the key rotation operation. Instead, we predict the combined leakage $h_{\Sigma,j} = h_{0,j} + h_{1,j}$. Our key recovery works well for this combined leakage, as explained in Section 5.4.4. Note that we know for each key bit $h_{i,j}$ at which clock cycle it is processed (if not, several hypotheses can be checked in parallel by analyzing neighboring clock cycles). In fact, knowing the implementation, it is predictable which key bit $h_{i,j}$ enters the carry register in which clock cycle for the key rotation. We use this information to build a differential power analysis attack. In spite of the independence of

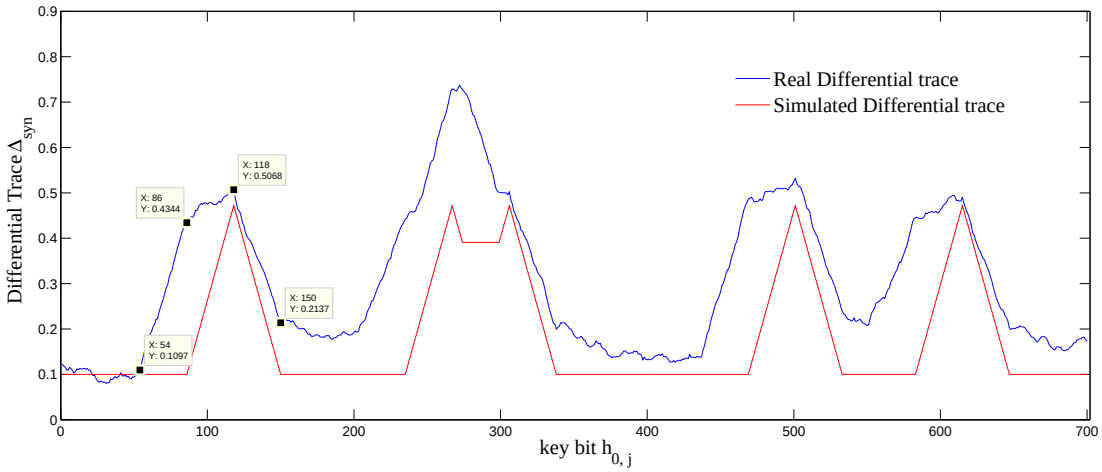


Figure 5.5: This plot is a magnification of Figure 5.4 which shows the characteristic shape of a single set key bit (left, $h_{0,118} = 1$) and two adjacent set key bits (center left, $h_{0,267} = h_{0,306} = 1$). The two shapes on the right are due to two other set key bits ($h_{0,501} = 1$ and $h_{0,616} = 1$).

the input x we claim the analysis method to be differential leakage analysis, since differential leakage traces can be computed—similar to the approach originally proposed in [KJJ99].

Our algorithm identifies all clock cycles where $h_{i,j}$ is written to or overwritten in the carry register in each trace L and extracts that leakage from L. Per processed ciphertext bit, only 150 words are rotated. The additional bit is stored in the carry register. Hence, all rotations together result in a total of $4801 \cdot 150$ carry register overwrites for each h_i. Since there are 4801 bits in h_i, each bit is written to the carry register 150 times. The corresponding clock cycles l are then identified and their corresponding leakage $\mathcal{L}_i(j, l)$ is combined, as done in horizontal SCA. The result is a differential leakage trace Δ_{carry} with only one bin per key bit. In other words, the *difference* between a key bit being zero and a key bit being one can be observed by comparing points of the leakage trace Δ_{carry} horizontally. Since the key is sparse, there are only very few bins that correspond to a bit $h_{i,j} = 1$, while most bins correspond to a bit $h_{i,j} = 0$. The implicit assumption of all bits leaking the same way is perfectly justified: each bit $h_{i,j}$ takes each column position exactly once, in a specific row. That means due to the rotation, each key bit leaks in every position exactly once, averaging out any position-specific leakages.

In order to detect whether a key bit is set, i. e., $h_{i,j} = 1$, we average over all clock cycles where $h_{i,j}$ is written to the carry register.

$$\begin{aligned}\Delta_{\text{carry}}(j) &= \frac{1}{150}\sum_{l=1}^{150}\left(\mathcal{L}_0(j,l) + \mathcal{L}_1(j,l)\right) \\ &= \text{avg}\left(\lambda_{\text{carry}}(0,j) + \lambda_{\text{read}}(0,j) + \lambda_{\text{store}}(0,j)\right. \\ &\quad \left. + \lambda_{\text{carry}}(1,j) + \lambda_{\text{read}}(1,j) + \lambda_{\text{store}}(1,j)\right)\end{aligned}$$

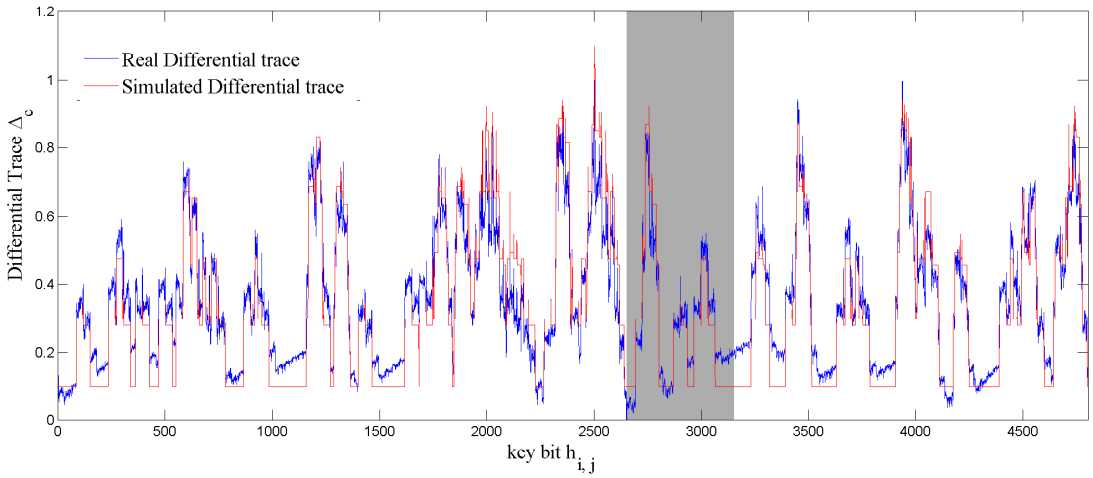


Figure 5.6: Differential leakage trace for key rotation. The plot shows the normalized leakage (vertical axis) of both key parts $h_{\Sigma,j} = h_0 + h_1$ over the key bit index (horizontal axis). The red (gray) line is the simulated leakage while the blue (black) line is the observed leakage from the target implementation.

Since $h_{i,j-32} = 0$ with very high probability, $\Delta_{\text{carry}}(j)$ depends directly on the key bit. Further, $h_{i,j} = 1$ has an even stronger influence on $\Delta_{\text{carry}}(j \pm 32)$, since it leaks through $\lambda_{\text{carry}}(i,j)$ and either $\lambda_{\text{read}}(i,j)$ or $\lambda_{\text{store}}(i,j)$. The dependence of $\Delta_{\text{carry}}(j)$ on neighboring key bits $h_{i,j\pm\delta}$, with $\delta \leq 32$, implies that each set key bit not only results in an increased leakage signal for its own position (i. e., index j), but also in the neighboring positions. Note that due to the differing weights, each set key bit imprints a characteristic shape onto the leakage trace. These shapes can (and actually will) overlap if several key bits in the same region are set.

Figure 5.6 shows the comparison of the simulated leakage trace (red(gray) line) using the power model and the real leakage trace (blue/black line). The characteristic shape is highlighted in Figure 5.7, which is a magnification of a single set bit of the key, surrounded by zeroes.

In summary, the key rotation analysis allows us to detect joint leakages of h_0 and h_1. This is due to the target implementation that processes both in parallel. The key rotation leakage features a characteristic shape with easily detectable bounds. This allows for a precise location of set key bits. Furthermore, the analysis of the key rotation is mostly input-independent, as will be discussed in Section 5.4.3. More importantly, each bit features 150 leakage observations per trace L, resulting in a very strong leakage.

Key Bit Recovery

The computation of syndrome and key rotation both cause leakages which can be analyzed in the presented differential traces. In both of the differential traces, characteristic shapes caused by set key bits can be detected and used to recover the set key bits. In the same way, the traces can be used to detect key bits that are not set. For the computation of the syndrome, the differential trace can recover the key bits of h_0 or h_1 separately, depending on the ciphertext we

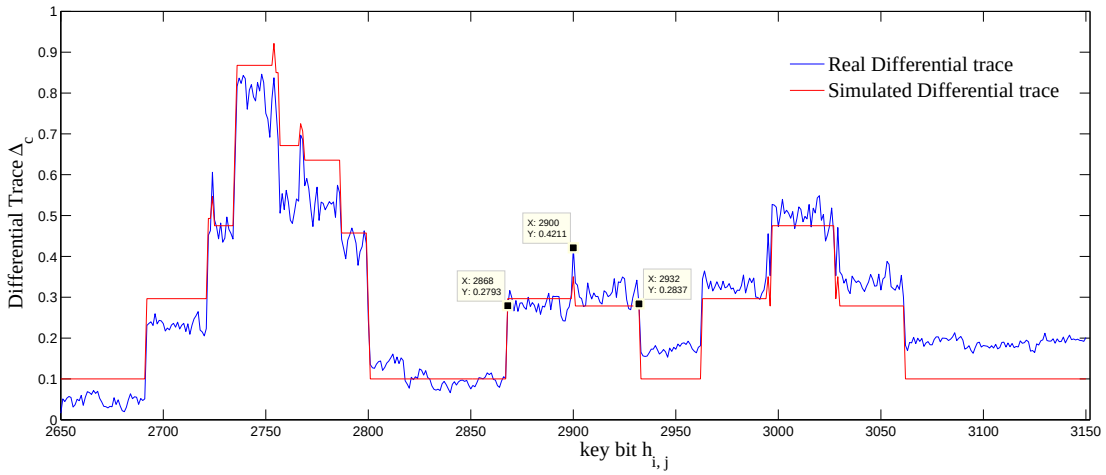


Figure 5.7: A magnified version of Figure 5.6 that highlights the characteristic shape of a single set bit (center) as well as the overlap of two (right) and three (left) "adjacent" set bits.

use. For the key rotation, since the analyzed implementation processes h_0 and h_1 in parallel, resulting in an overlap of the leakages, the differential trace actually recovers the key bits of $h_\Sigma = h_0 + h_1$.

In order to recover key bits, the characteristic shapes need to be detected. We propose a generic shape detection algorithm that works as follows:

(1) **Shape Definition** From the differential leakage trace, one singular characteristic shape can be identified and used as a template for set bits. The template is used to generate a shape threshold as shown in Figure 5.7 for the key rotation leakage and Figure 5.5 for the syndrome computation leakage. The threshold is defined by the value of features in this shape such as edges, slopes and pulses.

(2) **Shape Detection** For each key bit in the differential leakage trace, we check if this key bit together with the neighboring key bits can form a characteristic shape. This is done by checking if there are features that are beyond the threshold. If more than two features exist, it is highly probable that this key bit is set. If no feature exists, then it is highly probable that this key bit is 0. Otherwise, we mark this key bit as undetermined.

Note that the shapes will overlap if two set key bits are close to each other. Furthermore, the leakage traces are noisy, hence we can only recover parts of the key bits, leaving the other key bits undetermined. By choosing the thresholds for shape detection carefully, the number of detected bits can be maximized while keeping the number of false positive errors as low as needed.

5.4.3 Measurement Setup and Results

We ported our lightweight QC-MDPC McEliece FPGA implementation (cf. Section 5.3) to a Xilinx Virtex-5 LX50 FPGA which is mounted on a Sasebo-GII side-channel attack evaluation board. The implementation is clocked at 3 MHz by default. Measurements were performed using a Tektronix DPO 5104 oscilloscope at a sampling rate of 100 MS/s. Since our attack focuses on the syndrome computation, only the syndrome computation was recorded. The syndrome computation takes 245 ms, resulting in long traces. For the ease of analysis, a peak extraction was performed. In each clock cycle only the point of maximum power consumption is retained. The peak extraction prevents potential alignment issues and makes data handling much faster.

As mentioned in Section 5.4.2, key rotation and syndrome computation run in parallel which leads to a mixed leakage. To fully exploit the leakages, measurements were obtained in three different scenarios:

- **Known Ciphertext** In this scenario we assume the adversary to only *observe* ciphertext-leakage pairs. Hence, the ciphertexts x are chosen uniformly at random. While this can result in invalid ciphertexts, the attacker could also just generate valid ciphertexts by choosing plaintexts at will. In this scenario, a mixed leakage of key rotation and syndrome computation is obtained.
- **All-Zero Ciphertext** In order to minimize the impact of the syndrome computation and storage on the leakage, we recorded the power consumption for an all-0 ciphertext. The syndrome is never updated when the ciphertext is 0, while key rotation is always executed. Note that the all-zero word is a valid codeword without any errors. This corresponds to a chosen ciphertext side-channel attack, without the need to observe the corresponding plaintext.
- **Single-One Ciphertext** As mentioned in Section 5.4.2, the ciphertext weight is chosen to be one in this scenario, i. e., only a single bit of the ciphertext is set. This is done by adding a one bit error in each position of the all-0 ciphertext. There are 9602 such ciphertexts since both message and the redundant part have 4801 bit positions.

Results of the Vertical Attack

To extract key leakage from the syndrome computation, the single-1 ciphertexts give the main contribution. In fact, they provide the leakages of the $\mathcal{L}_{j,\mathrm{syn}}(l)$ term in Equation (5.3). The syndrome-storage independent leakage $\mathcal{L}_{j,\mathrm{const}}(l)$ can either be derived by an average of several all-0 leakage traces or the average of all used single-1 measurements. The latter approach has the advantage of not requiring additional measurements. We chose the former approach, as it is slightly less noisy. By subtraction of the two leakage terms, we derive the leakage of the syndrome computation only. Figure 5.4 shows the differential trace of the syndrome computation with respect to h_0.

The magnification of the differential trace in Figure 5.5 highlights the observed characteristic shapes imprinted by set key bits $h_{0,j} = 1$. The shape on the left is caused by a single set key bit $h_{0,118}$ with neighboring key bits set as 0. The second shape from the left is the result of two overlapping shapes of set bits in position 267 and 306, i. e., $h_{0,267} = h_{0,306} = 1$.

Table 5.6: Key bit recovery rates (#rec) and bit error rates (#error) for h_0 based on the leakage of the syndrome computation for various thresholds and number of traces. Numbers in parentheses are error occurrences that are not close to a true set bit.

Key bit value	Total # of traces	Threshold: 16		Threshold: 20		Threshold: 24		Threshold: 28	
		#rec	#error	#rec	#error	#rec	#error	#rec	#error
0	$1 \cdot 4801$	2636	0	3281	4	4089	12	4702	34
	$2 \cdot 4801$	2672	0	3143	2	3749	6	4463	17
	$5 \cdot 4801$	2681	1	3063	3	3573	6	4133	10
	$10 \cdot 4801$	2703	0	3035	3	3439	6	3931	8
1	$1 \cdot 4801$	14	12 (0)	10	7 (0)	3	2 (0)	0	0(0)
	$2 \cdot 4801$	32	25 (1)	17	13 (0)	11	8 (0)	3	2(0)
	$5 \cdot 4801$	137	118 (13)	74	59 (2)	30	21 (1)	8	5(0)
	$10 \cdot 4801$	248	225 (1)	166	145 (0)	76	60 (2)	26	15(0)

Key Extraction To actually recover the key bits from the differential trace $\Delta_{\text{syn}}(j)$, the recovery algorithm described in Section 5.4.2 is applied. The first step is to build the threshold based on features in the shape. As shown in Figure 5.5, the set key bit $h_{0,j} = 1$ for $j = 118$ caused a characteristic shape where there are two strong features. One is a rising slope from $h_{0,j-64}$ to $h_{0,j-32}$ and the other one is a falling slope from $h_{0,j}$ to $h_{0,j+32}$.

An easy way to detect slopes is by computing the backward difference of $\Delta_{\text{syn}}(j)$ as $\Delta'_{\text{syn}}(j) = \Delta_{\text{syn}}(j) - \Delta_{\text{syn}}(j-1)$, which is strictly positive for rising slopes and strictly negative for falling slopes. The number of values for which $\Delta'_{\text{syn}}(j-64)$ to $\Delta'_{\text{syn}}(j-32)$ is positive and for which $\Delta'_{\text{syn}}(j)$ to $\Delta'_{\text{syn}}(j+32)$ is negative are counted separately. If both of the features exist, $h_{0,j}$ is taken as 1. If none of the features exist, $h_{0,j}$ is taken as 0. Otherwise, it is taken as undetermined. As discussed in Section 5.4.2, due to the overlapping and noise in the differential trace, there are false positive errors in the recovered key bits. The detection works very well for set key bits that are surrounded by zeros, and less well for set bits that are located close to each other. A partial improvement can be achieved by removing (subtracting) the leakage of detected bits from the leakage trace and thereby decomposing an area of overlapping shapes into its components. However, this process turned out to be quite error-prone in itself, so that we did not further explore that direction. As we show in Section 5.4.4, such improvements to the detection algorithms are not necessary, as the recovered information is already plenty to recover the correct key.

Table 5.6 shows the results using this recovery algorithm. For each experiment, a multiple of 4801 single-1 ciphertexts are used for computing $\Delta_{\text{syn}}(j)$. As expected, a lower threshold reduces the number of detected zeros, while it increases the number of detected ones. However, with a higher number of detections, the number of false positives usually increases as well. Finally, observing a higher number of traces reduces noise and helps a cleaner shape detection. This is directly obvious from the zero recovery results, where the number of errors declines for an increased number of used measurements. For the recovery of set bits, the obvious improvement for more observations is the higher number of recovered bits. However, the number of false positives also tends to go up quickly with more measurements. This is due to the correlation

effect for closely located bits described in Section 5.4.2. The described detection based on thresholds favors the detection of correlated bits close to true one bits as well. This means that the detected errors are bits located close to a true set bit. In fact, for lower thresholds, the method returns sequences of ones, of which only one (of the center ones) is a true positive. This means that for each set key bit there will be a few false positives in the neighboring bits as well. One could say that the ones are correctly detected, but that there is remaining uncertainty of the exact location. The number in the parentheses shows the number of false positives that cannot be explained by this, i. e., false positives that are not due to the choice of the threshold. We will later show that the remaining errors in the leakage can be fixed in the final full key recovery phase in Section 5.4.4.

Results of the Horizontal Attack

Since the key rotation is independent of the ciphertext, the choice of the ciphertext could be arbitrary. However, key rotation and syndrome computation run in parallel, leading to a mixed leakage. To determine the influence of the syndrome computation, two different ciphertext scenarios are studied. One is the all-0 ciphertext to minimize the influence of the syndrome computation. In this scenario the syndrome remains all-0 throughout the entire computation. Hence, this scenario represents a chosen-ciphertext attack, just as the previously described vertical attack. The other scenario assumes random ciphertexts for each decryption, where each bit in x is set with a 50% probability. This scenario is representative of a known-ciphertext attack. For each scenario we took 256 measurements.

Next, we averaged over all considered traces in both scenarios. From the resulting average trace, $4801 \cdot 150$ peaks are extracted and used to construct the differential leakage traces Δ_{carry} as explained in Section 5.4.2. Note that averaging explicitly before the computation of Δ_{carry} or implicitly during the computation of Δ_{carry} does not influence the result. Figure 5.8 shows the differential leakage traces for the key rotation, showing the key bit position (horizontal axis) vs. the bit leakage (vertical axis) for all key bits. The blue (black) line indicates the result for the all-0 ciphertext scenario while the green (gray) line indicates the results for the random ciphertext. The latter one is slightly noisier, but nevertheless provides a well-exploitable leakage for a low number of observations. Figure 5.7 shows magnifications of the differential leakage trace to highlight the characteristic shapes, particularly the one generated by setting the key bit $h_{i,2900}$ as 1 and the neighboring key bits as 0.

The other shapes in Figure 5.7 result from the overlapping of characteristic shapes that occur when set key bits of h are close to each other. We noticed that set key bits for h_0 result in a slightly different shape than those of h_1. Since this difference cannot be distinguished as easily, we did not further try to exploit this information.

Key Extraction To extract keys from Δ_{carry}, we used the algorithm described in Section 5.4.2. The first step is to define the characteristic shape. Distinguishable features such as the rising edge, the pulse in the center and the falling edge are clearly visible in Figure 5.7 and are used to detect the shape. These features are quantified using a threshold vector. Then, for each key bit $h_{i,j}$ in Δ_{carry}, we check if there is a pulse at $h_{i,j}$, a rising edge at $h_{i,j-32}$ and a falling edge at $h_{i,j+32}$. If more than one feature exists for $h_{i,j}$, we take $h_{i,j}$ as 1. If no feature exists,

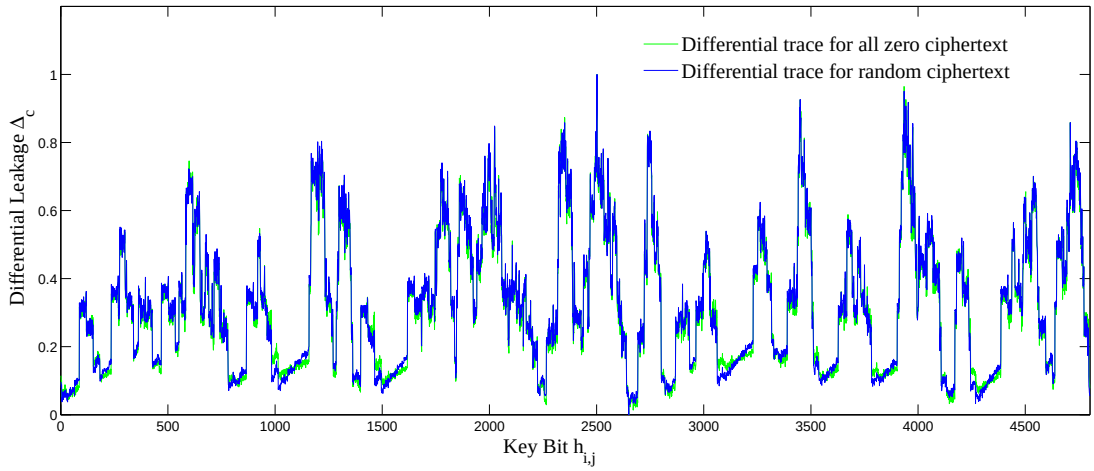


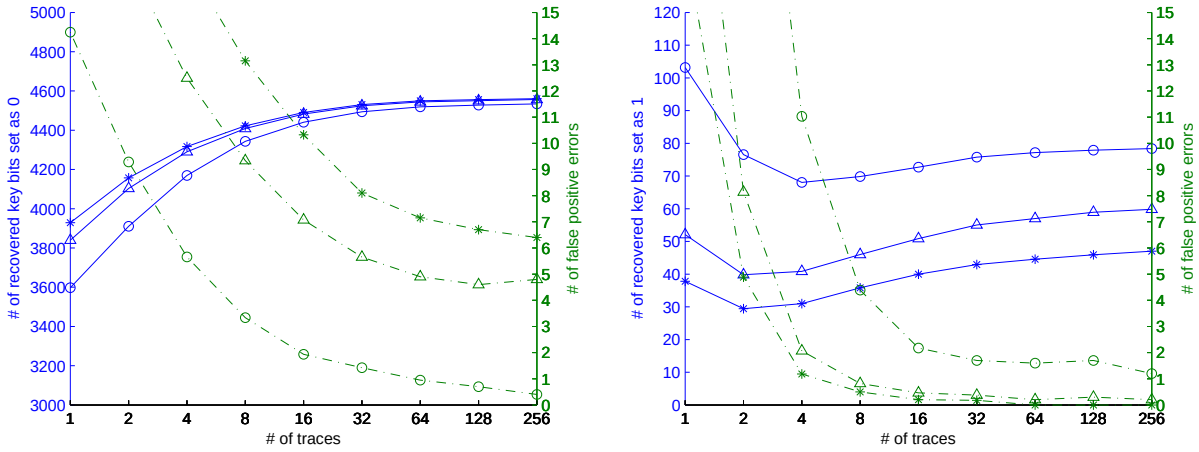
Figure 5.8: Normalized differential leakage trace Δ_{carry} for the key rotation for the bits of $h_{\Sigma,j} = h_0 + h_1$. Whether the ciphertext is known (green/gray line) or all-0 (blue/black line) has only marginal influence on the observed leakage.

$h_{i,j}$ is taken as 0. If only one feature exists, $h_{i,j}$ is left as undetermined key bit. Depending on the number of traces used for generating Δ_{carry}, it can be noisy and there will be false positive errors in recovered key bits. Errors can also be introduced by unfavorable overlapping of shapes.

Figure 5.9 shows how the chosen threshold affects the key recovery. Three different thresholds are used. The first one ($\circ$) is exactly the value extracted from the characteristic shape in Δ_{carry}. The other two ($\triangle$ and then $*$) are increased based on the first one. In Figure 5.9a, as the number of traces used to generate the differential leakage trace increases, the number of recovered 0 key bits increases and the number of false positive errors decreases for all three thresholds. However, the less aggressive the threshold is, the lower is the number of false positive errors. In contrast, Figure 5.9b shows that with the least aggressive threshold ($\circ$), more key bits of 1 can be recovered with a few more false positive errors. Hence, to recover more key bits of 0 with least false positive errors, the less aggressive threshold should be used. In contrast, to recover key bits of 1 with least false positive errors, the more aggressive threshold should be used. Note that we repeated our experiments for five different randomly generated keys to ensure the result is not key dependent. The figures show the average result for those experiments.

Figure 5.10a shows a comparison of the number of recovered key bits and false positive errors between the all-0 ciphertext and random ciphertext. As the number of traces used to generate the differential leakage trace increases, the number of recovered key bits of 0 increases and the number of false positive errors decreases for both cases. However, with the all-0 ciphertext, there are fewer positive errors. In conclusion, the all-0 ciphertext is more advantageous to the DPA of key rotation. Hence, we use the traces with the all-0 ciphertext in the other experiments.

Modern electronic devices run faster than 3 MHz which is the default clock rate for the SASEBO board and widely used in power analysis experiments. In order to validate our attack on faster platforms, the performance of the attack was measured for the same design clocked at 8 MHz and 16 MHz. The sampling rate was accordingly increased to 200 MS/s and 250 MS/s,



(a) Recovered 0 bits vs. false positives.

(b) Recovered 1 bits vs. false positives.

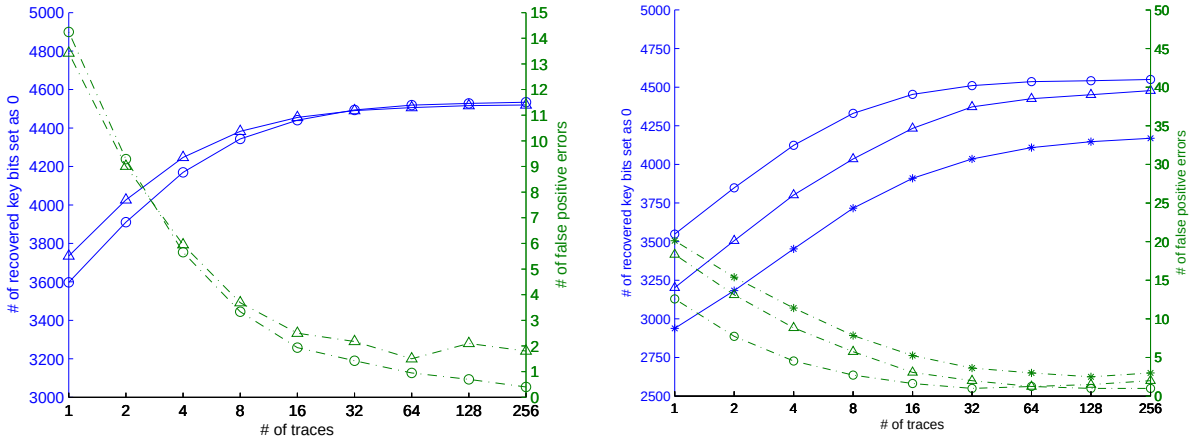
Figure 5.9: Key bit recovery rates for a range of detection thresholds for recovering 0 key bits (Figure 5.9a) and 1 key bits (Figure 5.9b). Solid line indicates the number of recovered bits (out of 90 ones and 4711 zeroes, scale on left), the dashed line indicates the number of false positives (scale on right). Markers $\circ$, then $\triangle$, and then $*$ indicate the increasing values for the threshold.

respectively. For each case, 256 traces were obtained using the all-0 ciphertext, followed by peak extraction. Figure 5.10b shows the degradation of the leakage over the increasing clock rate by comparing the number of recovered 0 key bits and false positive errors. In all three cases, the number of recovered 0 key bits increases and the number of false positive errors decreases, as the number of analyzed traces increases. However, the lower the clock rate is, the better the key bits extraction works. With a 3 MHz clock rate ($\circ$), almost 4500 of the 0 key bits can be recovered with about 1 false positive error when using all 256 traces while 4000 of the 0 bits are recovered with about 3 false positive errors at a clock rate of 16 MHz ($*$).

Overall, it can be seen that with as little as 10 measurements, more than half the key bits can be recovered with a remaining number of errors that is small enough to allow for efficient error correction. With 100 measurements and a careful choice of thresholds, the determined bits are entirely error-free at lower clock rates. This strong leakage is partially due to the fact that 150 leakages are extracted from each measurement, strongly amplifying the amount of leakage gained from each individual trace. So, in conclusion, the horizontal attack outperforms the vertical attack on the targeted unprotected implementation, but can only recover a combined leakage of h_0 and h_1.

5.4.4 Full Key Recovery

Next we analyze how to recover the full key of QC-MDPC McEliece if the adversary has knowledge of several set bits of the key as well as several zero bits of the key, possibly with few errors. We show that the structure of the key can be used to recover the remaining uncertain bits efficiently, or to detect remaining errors.



(a) Random vs. all-0 input.

(b) Varying clock rates.

Figure 5.10: Key bit recovery rates for recovering 0 key bits. Solid line indicates the number of recovered bits (out of 4711 zeroes, scale on left), the dashed line indicates the number of false positives (scale on right). Figure 5.10a compares *known* random (∘) vs. *chosen* all-0 (△) ciphertext inputs. Figure 5.10b compares the experiments for varying clock rates: ∘ 3 MHz, △ 8 MHz, and ∗ 16 MHz.

Exploiting a Connection between Private Key and Public Key

The private key consists of two related parts, h_0 and h_1. Due to the relation between the secret h_0, h_1 and the public matrix Q, we can express h_0 as:

$$h_0 = h_1 \cdot Q^T \tag{5.4}$$

Likewise, given h_0, one can compute h_1, since Q is invertible. This means that once the first half of the private key is recovered, the second half can be computed using the public key. More interestingly, this relationship can be used for *error detection* for each h_i independently: since Q is of high weight (each bit has approximately a 50% chance of being 1), even a single bit error in h_i^* will result in a high weight of a consequently derived $h_{\bar{i}}^*$, i. e., $\mathrm{wt}(h_{\bar{i}}^*) \approx r/2$. A correct h_i, however, will result in an $h_{\bar{i}}$ of low weight, in our case $\mathrm{wt}(h_{\bar{i}}) = 45$. We are currently not aware how slightly faulty or noisy information of h_0 and h_1 can be combined more efficiently without a trial and error approach using the aforementioned relationship.

If the adversary observes a combined leakage of h_0 and h_1 as is the case for the horizontal attack described in Section 5.4.2, key recovery is still possible. Adding h_1 on both sides of Equation (5.4) we obtain

$$h_0 \oplus h_1 = h_1 \cdot (Q^T \oplus I_{4801}). \tag{5.5}$$

If side-channel leakage allows us to obtain the combined leakage $h_0 \oplus h_1$ and the rank of $Q^T \oplus I_{4801}$ is high, we can solve this linear system of equations for h_1 with a computer algebra system like Magma [BCP97]—and then derive h_0 from Equation (5.4). In our experiments, the rank observed for $Q^T \oplus I_{4801}$ was 4800, resulting in two candidate solutions with only one of

them having the correct Hamming weight. So in cases where all ones can be correctly identified, Equations (5.4) and (5.5) enable a practical key recovery.

Due to noise observed in both attacks and leakage overlapping observed in the analysis of the key rotation, there are probably false positive errors in the recovered bits. Hence, error correction would be essential to correct positions that are slightly off. Guessing error positions becomes infeasible quickly, even with small improvements over an exhaustive search of $\binom{4801}{l}$ possibilities for l errors. We did not try to devise elaborate error-correction strategies, as a different attack strategy which relies on exploiting only key bits detected with a high confidence turned out to be quite effective. We explain this strategy next.

Efficient Key Recovery from Partial Information

After having identified several bits of the private key correctly with either attack strategy, we aim at an efficient way to recover remaining unknown or uncertain key bits. The following description assumes the combined leakage of h_0 and h_1, as observed in the horizontal analysis of the key rotation. For cases where the leakages of h_0 and h_1 occur separately, as is the case in the vertical analysis of the syndrome computation, the described strategy naturally carries over when Equation (5.4) (instead of Equation (5.5)) is used as starting point.

We define B_0, B_1 and B_u as index sets indicating the locations of definite zeroes, definite ones and positions of undetermined bits in $h_0 \oplus h_1$ such that

$$B_0 \,\dot\cup\, B_1 \,\dot\cup\, B_u = \{0, 1, \ldots, 4800\}\,. \tag{5.6}$$

Positions in B_0 indicate that both h_0 and h_1 are zero in that position, while positions in B_1 will mean a one in either h_0 or h_1.[3] Hence, the uncertain positions for h_1 are $B_u^1 = B_1 \,\dot\cup\, B_u$, and with Iverson's convention [Knu92] we can summarize our knowledge of $h_0 \oplus h_1$ and h_1 as $h_0 \oplus h_1 = \langle 1 \cdot [i \in B_1] + u \cdot [i \in B_u] \rangle_{0 \le i \le 4800}$ and $h_1 = \langle u \cdot [i \in B_u^1] \rangle_{0 \le i \le 4800}$, where u indicates unknown bits ("erasures"). So Equation (5.5) yields

$$\begin{aligned} &\langle 1 \cdot [i \in B_1] + u \cdot [i \in B_u] \rangle_{0 \le i \le 4800} \\ &= \left\langle u \cdot [i \in B_u^1] \right\rangle_{0 \le i \le 4800} \cdot (Q^T \oplus I_{4801}). \end{aligned}$$

As the indices in B_0 indicate definite zeroes in $h_0 \oplus h_1$ and h_1, the corresponding *rows* in the matrix $Q^T \oplus I_{4801}$ will always be multiplied with a zero coefficient. We remove these $|B_0|$ rows and the corresponding known 0-entries in h_1, obtaining an updated equation system

$$\begin{aligned} &\langle 1 \cdot [i \in B_1] + u \cdot [i \in B_u] \rangle_{0 \le i \le 4800} \\ &= \left\langle u \cdot [i \in B_u^1] \right\rangle_{i \notin B_0} \cdot Q'. \end{aligned} \tag{5.7}$$

with a (smaller) matrix $Q' \in \mathbb{F}_2^{(4801-|B_0|)\times 4801}$. There are $4801 - |B_0| - |B_1|$ unknown bits on the left- and $4801 - |B_0|$ unknown bits on the right-hand side of Equation (5.7). As we are only

[3] The (rare) case of h_0 and h_1 having a one in the same position is not considered here, as this situation is quite apparent from the side-channel leakage.

interested in finding h_1, we can try to eliminate unknown values in $h_0 \oplus h_1$ by dropping *columns* from Q'. One may hope that $|B_u|$ columns can be eliminated without Q' dropping in rank, so that we end up with a linear system of equations

$$\langle 1 \cdot [i \in B_1] \rangle_{i \notin B_u} = \left\langle u \cdot [i \in B_u^1] \right\rangle_{i \notin B_0} \cdot Q'' \tag{5.8}$$

in $4801 - |B_0|$ unknowns and a matrix $Q'' \in \mathbb{F}_2^{(4801-|B_0|)\times(4801-|B_u|)}$. If $|B_u| \leq |B_0|$ one may hope that this linear system of equations can be solved and yields a unique candidate for h_1.

To check the practical feasibility of this approach, we ran several experiments in Magma [BCP97], solving the equation system given in Equation (5.8) for several different vectors B_0 and B_1. We were particularly interested in the situation where knowledge of 1-positions in $h_0 \oplus h_1$ is ignored (i. e., $B_1 = \emptyset$), because in our measurements the 0-detection was more reliable. With $B_1 = \emptyset$, the resulting system of equations is homogeneous and thus in addition to h_1 also has the trivial solution. From Equation (5.6) we see that the condition $|B_u| \leq |B_0|$ now implies that $|B_0| \geq \lceil 4801/2 \rceil$. Staying above this threshold, in our experiments we obtained no more than 8 candidates for h_1, and the weight condition identified the correct private key uniquely.

For $|B_0| < 2400$, the kernel of the matrix Q'' in Equation (5.8) gets larger quickly and we obtain additional candidates for h_1, but finding the correct h_1 may still be feasible by looking at the Hamming weight of the candidates as long as the number of candidates is not overwhelming. The results in Section 5.4.3 show that for the target implementation the attacker can expect to recover more information from the side-channel than necessary for recovering the private key. Having $|B_0|$ comfortably above the threshold of 2400, a few false positives in B_0 can be dealt with efficiently: Instead of using all of these bit positions, one can select subsets of size 2401 at random. Assuming a hypergeometric distribution, with f false positive errors among the $|B_0|$ indices, the probability of guessing 2401 error-free positions is $\binom{|B_0|-f}{2401}/\binom{|B_0|}{2401}$. E. g., with $|B_0| = 3281$ and $f = 4$, this probability is still $\approx 2^{-7.6}$. In summary, as long as more than half the bits of the key can be recovered with a low error rate, the remaining key bits can be determined using the above-described algebraic methods. Knowledge of additional bits of $h_0 \oplus h_1$ facilitates the handling of possibly remaining errors. Not being able to recover more than half the number of key bits can make the search infeasible, although—due to the highly biased key—guessing a few additional zeroes may still be an option.

5.4.5 Preventing the Attacks

The described attacks, especially the highly efficient horizontal attack, are somewhat specific to the implementation choices of the target, but can be adjusted to other implementation parameters as well. For example, an implementation that does not process h_0 and h_1 in parallel would simplify the horizontal attack and amplify the leakage. Implementations that use a different word size (the targeted implementation processes 32-bit words due to the BRAM structure of the FPGAs) will influence the described attack as well. The smaller the word size, the more leakages per target bit, most likely facilitating both attacks further. However, a massively parallelized implementation such as the one described in Section 5.2 could impede the described attack, since all bits would always be leaking in parallel. One might still be able to exploit resource-specific leakages, e. g., leakage from a carry register.

A more reliable way to prevent this attack is provided by side-channel countermeasures. A good overview of standard DPA countermeasures is available in [MOP07]. Countermeasures are typically classified as *masking* or *hiding* countermeasures. Both classes can be applied to an implementation of (QC-)MDPC McEliece and, if done correctly, should prevent the above-mentioned attack. These countermeasure techniques can be directly applied at the logic style level, allowing the digital design to remain unchanged, or can be applied at the algorithmic level, as described next. Masking needs to be applied to the syndrome *and* the key, since both leakage sources can be targeted separately, as shown by this work. In fact, a first masked version of the analyzed core has been implemented in [CEvMS16b]. The implementation applies a threshold implementation inspired masking with two to three shares to key and syndrome during syndrome computation and decoding to achieve a protection against first-order side-channel attacks. The resulting overhead is a factor of ~ 4 on both size and performance reduction. While being quite costly, such overheads are not uncommon for reliable side-channel protection mechanisms.

Another plausible solution strategy that should impede side-channel analysis while maintaining a much lower footprint than the masking countermeasure can be based on *shuffling*. Shuffling is a hiding-based countermeasure that randomizes the execution order. It has been discussed in detail, e. g., in [TH08]. Shuffling can be applied to the order in which the ciphertext bits are processed during syndrome computation (and the order of processing syndrome in the decoding step) or the order in which the key is processed. Both described attacks take advantage of the knowledge of *when* a specific key bit is processed. This advantage only holds for deterministic execution orders. By shuffling the syndrome computation the horizontal attack is completely prevented: Ciphertext bits and key bits would be processed in a random order, requiring the implementation to be able to rotate the private key by various offsets. As a result, all key bits would leak at random points in time. Common counterattacks such as *combing* (cf. again to [TH08]) would not be helpful in this scenario, since it would require a summation over all clock cycles, making all key bits leak in parallel and thereby making them indistinguishable. The situation is slightly more complex for the vertical attack on the syndrome computation, since in the chosen single-1 ciphertext attack, the occurrence of a non-zero leakage would indicate the processing of the set ciphertext bit. Hence, to also prevent the vertical attack, the order in which the bits within key and syndrome are processed would also need to be randomized, which hinders the attacker from distinguishing the key bits.

Note that such a countermeasure would require the implementation to be able to rotate the ciphertext, the private key and the syndrome by various offsets while ensuring that these offsets are not detectable by the adversary. Implementing shuffling in such a way that no additional leakages are introduced is not a trivial task, as discussed in [VCMKS12], for instance. However, such an implementation can be realized with comparably low area overhead, since no new arithmetic units nor additional storage, e. g., for masks, would be required.

5.5 Conclusion

This chapter presented high-performance implementations of the McEliece cryptosystem instantiated with QC-MDPC codes for Xilinx Virtex-6 FPGAs and lightweight designs of the scheme for Xilinx Spartan-6 FPGAs. Our first FPGA design primarily aims for high throughput and achieves competitive results by basing on the results of the decoder evaluations from Chapter 4

and by directly implementing the design in FPGA logic without using BRAMs. We showed that it is indeed possible to realize a code-based public-key cryptosystem with moderate key sizes and high performance in reconfigurable hardware. Our second FPGA design shows that it is possible to implement the same cryptosystem in a very lightweight way. In addition to considerably reducing the resource requirements by using embedded block memories that are offered in Xilinx FPGAs, we achieved reasonable performance for both encryption and decryption. Furthermore, the key sizes remain at a level that is much more appropriate for real-world usage than the key sizes of previous code-based schemes, which is an important metric for lightweight platforms. By demonstrating the excellent properties of this novel construction for embedded applications, we hope to have provided another incentive for further cryptanalytical investigation of QC-MDPC codes in the context of code-based cryptography.

Furthermore, we presented horizontal and vertical side-channel analysis techniques for QC-MDPC McEliece. Two different leakages which occur during the syndrome computation step of the decryption are exploited. The leakage of the syndrome register gives information on the two private key halves h_0 and h_1 separately and can be exploited by a fairly generic vertical attack. Thousands of chosen ciphertext traces are necessary for a successful key recovery. The leakage of a key rotation operation which occurs during the syndrome computation step of the decryption can be exploited by a horizontal side-channel attack that recovers a combined leakage of h_0 and h_1. The resulting attack is independent of the ciphertext and succeeds with tens of traces. A significant part of the key recovery stems from the relation between the private key and public key, which can be exploited to ease key recovery. In fact, recovering only half the bits of the (highly biased) private key with a low error rate is sufficient for a full key recovery. This work inspired a follow-up masked implementation of QC-MDPC McEliece [CEvMS16b] with masking applied to the syndrome and the key. The implementation applies a threshold inspired masking with two to three shares to key and syndrome during syndrome computation and during decoding to achieve a protection against first-order side-channel attacks at the cost of a 4x area increase and a 4x performance degradation.

Chapter 6

QC-MDPC McEliece for Embedded Microcontrollers and General-Purpose Processors

This chapter presents QC-MDPC McEliece for embedded microcontrollers and for general-purpose processors with a focus on ARM's Cortex-M4 and Intel's Haswell architecture. Besides practical issues such as random error generation, we demonstrate side-channel attacks on straightforward implementations of this scheme on embedded microcontrollers. Timing- and instruction-invariant coding strategies are proposed as countermeasures to strengthen QC-MDPC McEliece against timing attacks and simple power analysis attacks. Furthermore, we provide two implementations targeting general-purpose CPUs, a reference C implementation as well as a highly optimized implementation that makes use of vector instructions to achieve maximum performance.

This research was presented at PQCrypto'14 and appeared in the ACM Transactions on Embedded Computing Systems [vMG14b, vMOG15]. It is a joint work with Tobias Oder and Tim Güneysu.

Contents

6.1 Introduction

Besides their susceptibility to quantum computing attacks, the standard public-key encryption algorithms RSA and ECC usually do not perform well when implemented on embedded microcontrollers, especially when written purely in software. Dedicated processors are available in specialized devices to allow for accelerated RSA and ECC computations which however also drive the cost of such microcontrollers since more chip area is required to realize those co-processors.

The first microcontroller implementation of QC-MDPC McEliece scheme was proposed for AVR microcontrollers in [HvMG13]. The results indicate that it seems to be challenging to provide a reasonably fast implementation of QC-MDPC codes on low-cost 8-bit AVR ATxmega256A3 microcontrollers. Encryption and decryption take 830 ms and 2.7 s on this platform, based on the former 80-bit secure parameter set from [MTSB12] ($n_0 = 2, n = 9600, r = 4800, w = 90, t = 84$). In particular, decryption is too slow to be of practical interest for many real-world applications.

Cyclo-symmetric (CS-)MDPC codes in combination with the Niederreiter cryptosystem were proposed in [BBMR14], including an implementation for a small PIC microcontroller. As for the first QC-MDPC microcontroller implementation, its largest drawback is the decryption performance of 2.8 s. Furthermore, the CS-MDPC parameters as proposed in [BBMR14] do not reach the claimed security levels as shown by Perlner [Per14].

Despite sufficient performance, other highly relevant properties need further investigation as well to enable the deployment of QC-MDPC McEliece in practical systems. First, QC-MDPC on-chip key-generation has never been implemented on constrained devices. Second, McEliece as a probabilistic scheme requires a secure random number generator capable of producing error vectors of a certain Hamming weight during the encryption operation which has not been considered yet. Third, the QC-MDPC parameter sets were slightly updated by [MTSB13] compared to [MTSB12]. Fourth, the timing and the instruction flow of all previously presented implementations of the encryption and decryption operations depend on secret data. Fifth, microcontroller implementations of QC-MDPC McEliece encryption reported have not been investigated with regard to side-channel attacks so far.

Side-channel attacks on the McEliece cryptosystem have mostly targeted Goppa codes and exploited differences in the timing behavior [SSMS10, Str10, STM+08]. Improved timing attacks and corresponding countermeasures were presented in [AHPT11]. First practical power analysis attacks on Goppa-code McEliece implementations for 8-bit microcontrollers were presented in [HMP10]. Recent work investigated differential side-channel attacks on a lightweight QC-MDPC FPGA implementation [CEvMS15, CEvMS16a, CEvMS16b] (cf. Section 5.4).

Contribution In this chapter we present an implementation of QC-MDPC McEliece encryption providing 80 bits equivalent symmetric security on a low-cost ARM Cortex-M4 microcontroller with a reasonable performance of 42 *ms* for encryption and 251-558 *ms* for decryption. The parameter set we considered for implementation takes latest advances in cryptanalysis into account and we briefly discuss how to employ true random number generation for McEliece encryption. Side-channel attacks on a straightforward implementation of this scheme are demon-

strated followed by coding strategies and countermeasures to harden against timing attacks and simple power analysis. Finally, we present a vectorized implementation of QC-MDPC McEliece for modern general-purpose processors with multiple parameter sets and security levels to demonstrate the scheme's efficiency also on non-embedded platforms.

Outline We present our implementations and their improvements compared to previous work on embedded microcontrollers in Section 6.2. Side-channel attacks on QC-MDPC McEliece are demonstrated on two microcontroller platforms in Section 6.3. We propose countermeasures to strengthen our microcontroller implementations against these attacks and provide results in Section 6.4. A vectorized implementation of QC-MDPC McEliece for state-of-the-art CPUs is presented in Section 6.5 and a conclusion is drawn in Section 6.6.

6.2 Implementing QC-MDPC McEliece for ARM Cortex-M

The STM32F4 Discovery board is equipped with a STM32F407 microcontroller which features a 32-bit ARM Cortex-M4F CPU with 1 Mbyte flash memory, 192 Kbytes SRAM and a maximum clock frequency of 168 MHz. It sells at roughly the same price of USD 5-10 as the popular 8-bit AVR microcontroller ATxmega256A3, depending on the ordered quantity. Instead of 8-bit the STM32F407 offers a 32-bit architecture, can be clocked at higher frequencies, offers more flash and SRAM storage, comes with DSP and floating point instructions, provides communication interfaces such as CAN-, USB-/ and Ethernet controllers, and has a built-in true random number generator (TRNG).

Our implementations of QC-MDPC McEliece for the STM32F407 microcontroller cover key generation, encryption, and decryption with the main goal of achieving a reasonable time/memory trade-off.

Key Generation

Private-key generation starts by selecting a first row candidate for H_{n_0-1} with w/n_0 set bits. The indexes at which bits are set are generated using the microcontroller's TRNG in the range of $0 \leq i \leq r-1$. Since $r = 4801$ is not a power of two, we sample error indexes e_i with $\lceil \log_2(r) \rceil = 13$ bits from the TRNG and use them only if $e_i \leq r-1$ (i.e., rejection sampling).

The public-key computation requires that $H_{n_0-1}^{-1}$ exists. Hence, we apply the extended Euclidean algorithm to the first row candidate and $x^r - 1$. If the inverse does not exist, we select a new first row candidate for H_{n_0-1} and repeat. If the inverse exists, the first row of H_{n_0-1} is converted into a sparse representation where w/n_0 indexes point to the positions of set bits. These indexes are stored as part of the private-key.

Next, we generate random first rows for $H_0, \ldots, H_{n_0-2}$ with w/n_0 set bits as described for H_{n_0-1}, convert and store them in their sparse representation, and compute $(H_{n_0-1}^{-1} H_i)^\intercal, 0 \leq i \leq n_0 - 2$. Note that since the involved matrices are quasi-cyclic, the result is quasi-cyclic as well. The computed generator matrix is not sparse and hence its first row is stored in full length.

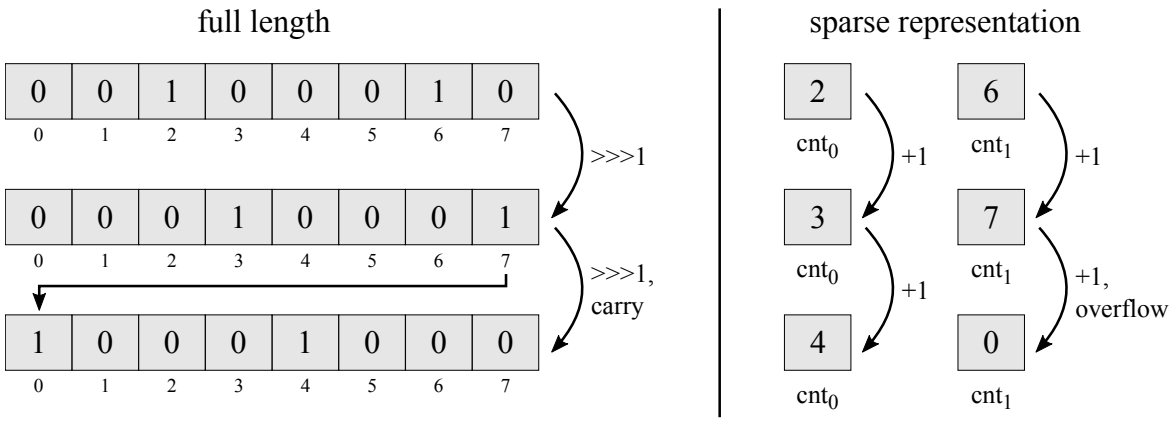


Figure 6.1: Example of an 8-bit register with two set bits in sparse and full length representation. Both values are rotated one bit to the right (>>> 1), twice. The second rotation demonstrates how a carry/overflow is handled in both representations.

Encryption

Encryption is divided into encoding a message and adding an error of weight t to the resulting codeword. To compute the redundant part of the codeword, set bits in message m select rows of the generator matrix G that have to be XORed. Starting from the first row of the generator matrix, we parse m bit-by-bit and decide whether or not to XOR the current row to the redundant part. For the next message bit the following row is generated by rotating it one bit to the right. This implementation approach was originally introduced in [HvMG13].

After computing the redundant part of the codeword, it is appended to the message and t random indexes are generated at which the codeword bits are inverted to transform the codeword into a ciphertext (i.e., the error addition). We retrieve the indexes from the microcontroller's internal TRNG and again use rejection sampling, this time with $\lceil \log_2(n) \rceil = 14$-bit random numbers, to achieve a uniform distribution of the error positions. In Section 6.3.2 we describe the shortcomings of this implementation approach with regard to side-channel attacks and present corresponding countermeasures in Section 6.4.1.

Decryption

We implement decoder $\mathcal{D}_1$ as described in Chapter 4 to decrypt ciphertexts. First, the syndrome is computed, which is a similar operation to encoding a message, except that the private-key is stored in a sparse representation. Each of the n_0 rows of the private-key is stored using a series of counters that point to the positions of set bits (here: 2×45 counters). To generate the next row, all counters are incremented by one. If a counter exceeds r, it overflowed and has to be reset to zero which in the full length representation is equal to the carry bit of a rotated row. As an example imagine a sparse 8-bit value with two set bits. Its corresponding sparse representation requires two counters cnt_0 and cnt_1 to store the positions of set bits. Figure 6.1 illustrates this example and shows how rotation is performed in both representations using 8-bit registers.

The ciphertext is split into n_0 parts which correspond to the n_0 blocks of the parity-check matrix. The ciphertext blocks are processed in parallel bit-by-bit. If a ciphertext bit is set, the corresponding row of the parity-check matrix is added to the syndrome otherwise the syndrome remains unchanged. The following rows of the parity-check matrix blocks are generated directly in the sparse representation by incrementing the counters. If a counter overflows, i.e., the counter value equals r, the counter is reset to zero.

If the computed syndrome $s \neq 0^r$, we proceed by counting how many parity-check equations are violated by a ciphertext bit. This is given by the number of bits that are set in both the syndrome and the row of the parity-check matrix block which corresponds to the ciphertext bit. If the number of unsatisfied parity-check equations exceeds a precomputed threshold b_i, the ciphertext bit is flipped and the row of the parity-check matrix block is added to the syndrome.

If the syndrome is zero after a decoding iteration, decoding was successful. Otherwise we continue with further iterations until we either reach successful decoding or a fixed maximum of iterations upon which a decoding error is returned. In Section 6.3.3 we describe the shortcomings of this implementation approach with regard to side-channel attacks and present corresponding countermeasures in Section 6.4.2.

6.3 Side-Channel Attacks

In the following we present power analysis attacks on the QC-MDPC McEliece encryption and decryption implementations and describe how two development boards were modified to allow meaningful power measurements. We attack our implementations for the STM32F407 and compiled the source code from [HvMG13] for the Atmel AVR XMEGA-A1 Xplained board which we attack as well. The AVR Xplained board features an 8-bit Atmel ATxmega128A1 microcontroller which can be clocked at a maximum frequency of 32 MHz. Its internals are equivalent to the ATxmega256A3 used in [HvMG13] except for less flash and SRAM memory.

Power analysis attacks exploit the fact that when cryptographic operations are executed on a physical device, information about the processed data and the executed instructions may be recovered from the consumed electrical energy at different points in time. Simple power analysis (SPA) attacks [KJJ99] are based on the idea that certain operations can be distinguished from each other by visual inspection or automated pattern recognition.

In this work we develop two side-channel attack (SCA) scenarios: first, a message recovery attack demonstrates that on-chip generated secret messages, e.g., symmetric secret-keys for hybrid encryption, can be obtained using significant single-trace leakage during encryption. Second, we present an SPA attack on the decryption operation which identifies the private-key.

6.3.1 Preparing the Evaluation Boards

Since our goal is to observe power traces from two microcontroller development boards, we modify the boards to allow clean power measurements as explained below. We only modify external components on the board, leaving the microcontrollers untouched.

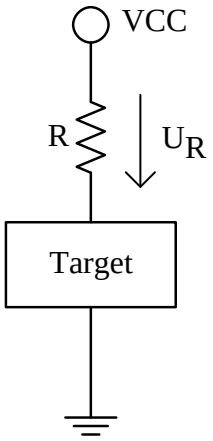


Figure 6.2: A measurement resistor R is inserted into the VCC path of the target device to measure the target's power consumption by measuring voltage U_R.

For our measurements we use a PicoScope 5203 with two channels that can obtain 500 MS/s for each channel sampling a bandwidth of 250 MHz. One probe measures the power consumptions at an inserted measurement resistor in the VCC path (cf. Figure 6.2), the other probe is used to signal the beginning and end of the cryptographic operation via an I/O pin of the respective microcontroller (i.e., a trigger signal).

Atmel AVR XMEGA-A1 Xplained Board

We removed all capacitors[1] connected between the microcontroller's VCC and GND and we placed a 2.7 Ω resistor onto the power supply measurement header that connects the board's 3.3 V to the VCC pins of the microcontroller. Furthermore, we added three capacitors in parallel (100 μF, 100 nF, 10 nF) right before our measurement resistor between the board's 3.3 V and GND to account for the removed capacitors. The modified AVR board is shown in Figure 6.3a.

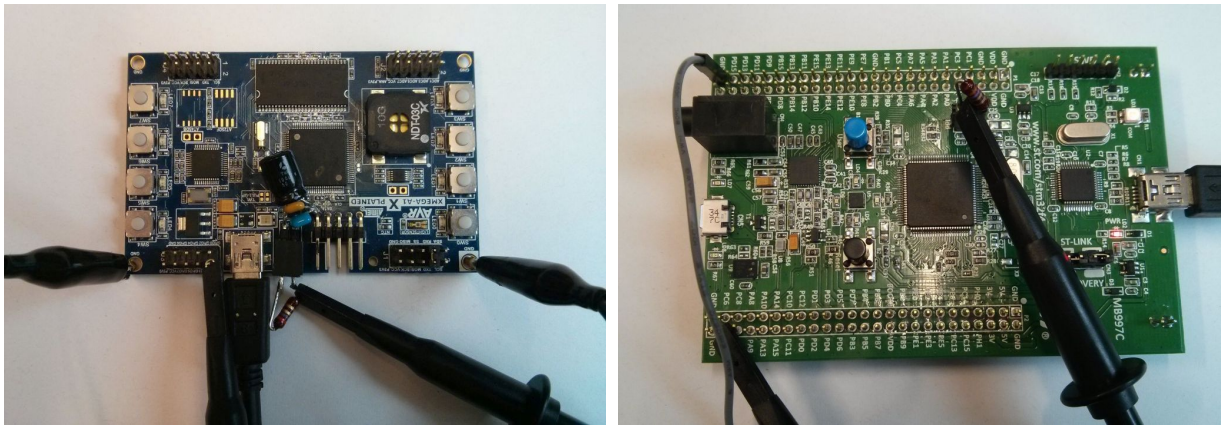
STM32F4 Discovery Board

Again, we removed all capacitors and coils[2] between the microcontroller's VDD pins and GND and placed a 2.7 Ω resistor onto the power supply measurement header (IDD) that connects the board's 3 V to the VDD pins of the microcontroller. Similarly, we added three capacitors in parallel (100 μF, 100 nF, 10 nF) right before our measurement resistor between the board's 3 V and GND. The modified STM32 board is shown in Figure 6.3b.

6.3.2 Message Recovery Attack

Imagine an implementation in which the microcontroller generates a symmetric key to encrypt bulk data. The symmetric key is encrypted under the public-key of the intended receiver using

[1] A total of ten 100nF capacitors (C102-C111) were removed, cf. [Atm10].
[2] One coil (L1) and 16 capacitors (C21-C26,C28-C37) were removed, cf. [STM14].



(a) Modified Atmel AVR XMEGA-A1 Xplained board with connected probes.

(b) Modified STM32F4 Discovery board with connected probes.

Figure 6.3: Measurement setups for our side-channel attacks.

public-key encryption. After exchanging the symmetric key, the communication is encrypted using a symmetric encryption scheme for performance reasons.

If an attacker is able to perform a message recovery attack on the public-key encryption, he is in possession of the symmetric (session-)key which allows to decrypt and forge ciphertexts until the symmetric key is updated. Although this attack is often not considered in SCA-related works, it is of practical relevance.

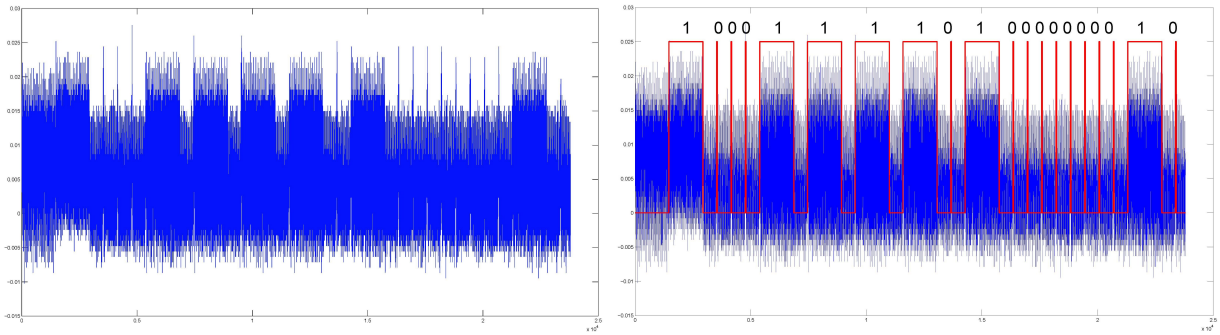
General Considerations

Recall that when encrypting a message m using QC-MDPC McEliece, the message is multiplied with the generator matrix G and an error e is added to the result.

$$x = m \cdot G + e$$

Message m selects rows of G which are accumulated to compute the redundant part of the codeword. A message recovery attack is successful if it is possible to detect if a certain row of G is accumulated or not, since each accumulation can be directly mapped to a specific message bit. Another approach would be to recover the error indexes when they are generated or when the error is added to the codeword. The recovered error together with the ciphertext could then be used to remove the error from the codeword.

The devices under test perform QC-MDPC McEliece encryptions as follows: if a message bit is set, the corresponding row of G is added to the redundant part, otherwise this step is skipped. Afterwards, the next row of G is generated and the process is repeated for the following message bit. The addition of one row of G to the redundant part involves hundreds of `load`, `xor`, and `store` operations on both platforms. Hence, our goal is to detect if this memory-intense operation is being executed or not by inspection of the power trace.



(a) Plain power trace. (b) Power trace with marked message bits.

Figure 6.4: Power trace of the encryption of a message starting with `0x8F402...` on an ATxmega128A1 microcontroller.

Experiments with the ATxmega Microcontroller

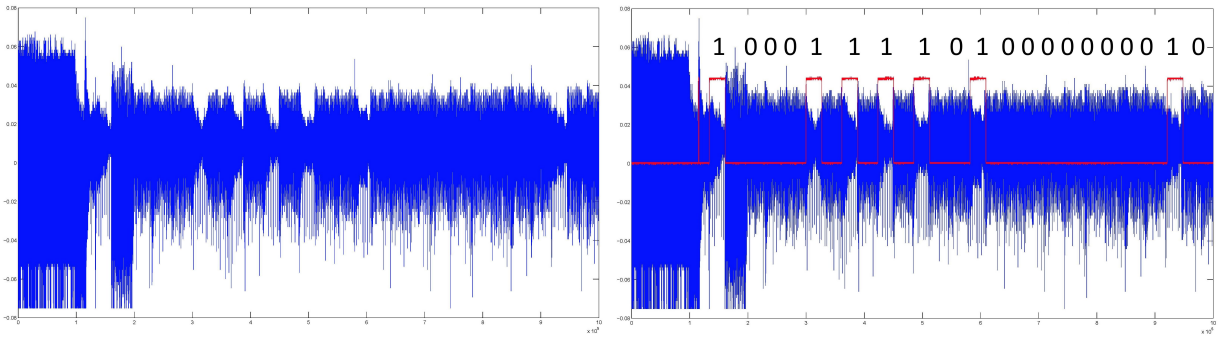
We recorded a power trace while encrypting a randomly selected message starting with `0x8F402...` under a valid public-key on the ATxmega128A1 microcontroller clocked at 8 MHz. The power trace shown in Figure 6.4a allows to distinguish three reoccurring patterns. Two of these patterns can be attributed to the performed or skipped row accumulation from G, the third pattern corresponds to the generation of the next row of G. Since the addition of a row of G corresponds to a set message bit, the message that is encrypted can be read more or less directly from a single power trace. We highlight the different patterns and message bits in Figure 6.4b. The attack is independent of the public-key under which the message is encrypted.

Experiments with the STM32 Microcontroller

We repeated the attack on the STM32F407 microcontroller with the same message and public-key as before. The power trace is shown in Figure 6.5a, the device was clocked at 42 MHz. The patterns cannot be identified as clearly as on the ATxmega, but still an observable difference in the power trace exists when a row of G is added to the redundant part of the codeword. We highlight the repeating pattern in Figure 6.5b and map the corresponding message bits to the power trace. Since in this case no visible pattern for a message bit being zero exists, we use the distance between two set message bits to determine how many zeros lie in-between. This is done by cross-correlating the ”one”-pattern with the recorded power trace and then dividing the distance from peak to peak by the time it takes to skip one accumulation and generate the next row of G. The exact duration of skipping one accumulation is obtained in a profiling phase which only has to be done once when setting up the attack.

6.3.3 Private-Key Recovery Attack

For the private-key recovery attack we assume that we are given a device which decrypts some known ciphertext. Knowledge of the corresponding plaintext is not required. The goal is to recover the private-key from the observed power consumption of the device during decryption.



(a) Plain power trace. (b) Power trace with marked message bits.

Figure 6.5: Power trace of the encryption of a message starting with `0x8F402...` on an STM32F407 microcontroller.

General Considerations

Recall that the syndrome s of the received ciphertext x is computed by multiplying the private parity-check matrix H with $x^\intercal$ at the beginning of a QC-MDPC McEliece decryption.

$$s = H \cdot x^\intercal$$

Since we are in a quasi-cyclic setting with $n_0 = 2$, the first rows of the two parity-check matrix blocks define the parity-check matrix. Further recall that the rows of the parity-check matrix are stored in a sparse representation using counters (cf. Figure 6.1).

Using SPA, at least two things should be observable from a power trace that is recorded during syndrome computation:

(1) A set ciphertext bit determines if a row of the private-key is being added to the syndrome or not (similar to the message recovery attack described in Section 6.3.2). Since the ciphertext usually is assumed to be known to an attacker, recovering the ciphertext bits from a power trace does not yield a meaningful attack.

(2) Incrementing the counters that resemble parts of the private-key must include an overflow check such that the counters are reset to zero if a carry occurs. If it is possible to detect an overflow, this might reveal the positions of set bits in the private-key which in turn could be used to build a full key recovery attack.

The AVR and the ARM implementations store the position of the private-key bits in counters which are incremented to generate the next rows of the quasi-cyclic parity-check matrix blocks. The counters are ordered such that the last counter stores the position of the most significant bit in the private-key. When rotating a row of the private-key there are conditional branches depending on whether the last counter overflowed or not. If an overflow occurred, all counter values are moved to the next counter and the first counter is reset. This reduces the overall complexity to test only the last counter on the overflow condition. Figure 6.6 depicts an example of this rotation technique for small parameters.

We set the ciphertext to the all-zero vector in our experiments to remove the influence of additions of private-key rows to the syndrome from the power traces. Our attack still works

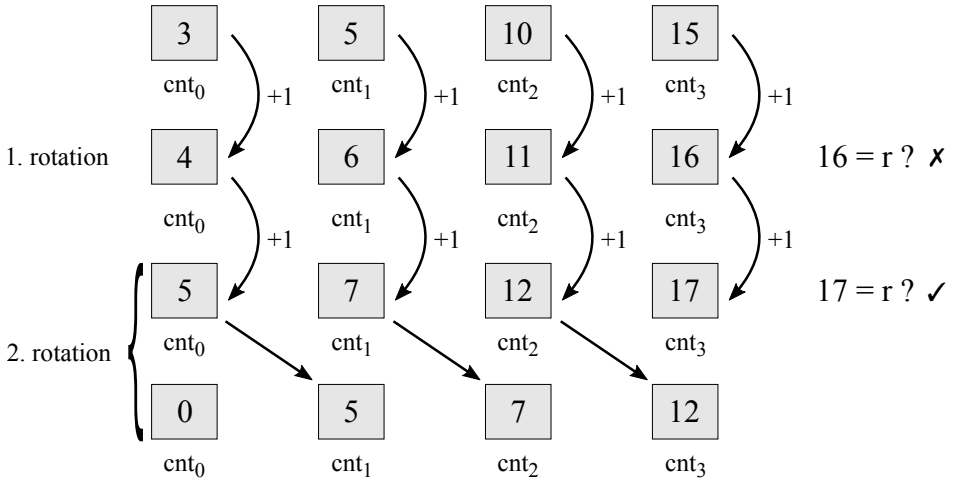


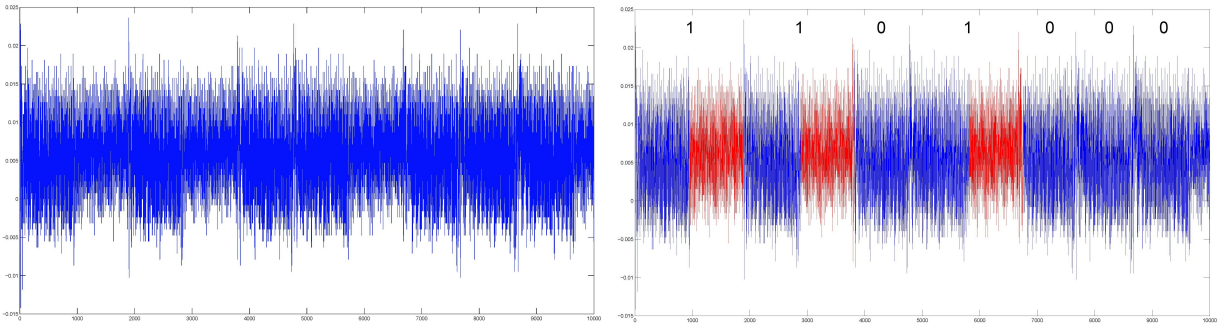
Figure 6.6: Example of the implemented rotation of vectors stored in sparse representations. Length r is set to 17 in this example. Counter cnt_3 always holds the most significant bit. If cnt_3 is equal to r after being incremented, the counter values are moved to the next counter (cnt_3 is overwritten first) and cnt_0 is reset to zero.

if any other ciphertext is used and only requires to profile the time it takes to add a row of the private-key to the syndrome once. Another option would be to only set bits at the end of the ciphertext, extract the private-key up to this point and then find the remaining private-key bits by smart brute-force which takes the relation between private- and public-key into account (cf. [CEvMS15]). Note that our attacks are independent of the implemented decoding algorithm since we attack the syndrome computation which all bit-flipping decoders execute as their first step.

Experiments with the ATxmega Microcontroller

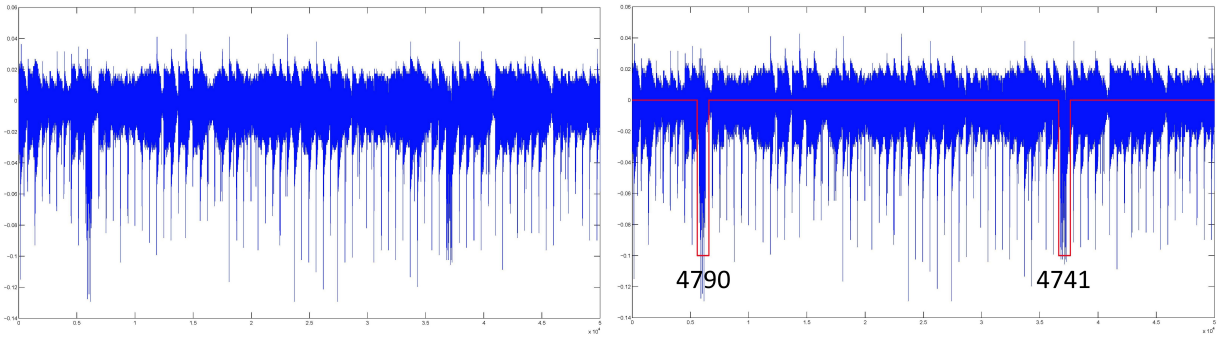
A power trace of the first few rounds of the syndrome computation is shown in Figure 6.7a for a private-key starting with $(1101000\ldots)_2$ on the ATxmega128A1 microcontroller. Two different repeating patterns can be distinguished in the power trace. Our experiments show that the first pattern occurs when the device is checking whether the current ciphertext bit is set (which is never the case since we set the ciphertext to the all-zero vector) and all counters are incremented by one. The second pattern only occurs in the power trace if the highest counter overflowed. Hence, we can distinguish an overflow which represents a carry bit in the private-key. In case both patterns appear after each other, the highest counter overflowed. If only the first pattern appears, the highest counter did not overflow.

An overflow means that the most significant bit of the private-key was set. Since the private-key is rotated bit-by-bit, every bit of the private-key will be the most-significant bit at some point during the syndrome computation. Hence, it is possible to recover the private-key from a power trace as shown in Figure 6.7b in which we highlight the two patterns and mark the corresponding private-key bits.



(a) Plain power trace. (b) Power trace with marked private-key bits.

Figure 6.7: Power traces recorded during syndrome computation on an ATxmega128A1 microcontroller. The first part of the private-key in this example starts with $(1101000\ldots)_2$.



(a) Plain power trace. (b) Power trace with marked private-key bits.

Figure 6.8: Power traces recorded during syndrome computation on a STM32F407 microcontroller. The first part of the private-key starts with set bits at positions 4790 and 4741.

Experiments with the STM32 Microcontroller

Figure 6.8a shows the beginning of a power trace that was recorded during syndrome computation of some ciphertext on the STM32F407 microcontroller. The first part of the private-key in this example has the first two set bits at positions 4790 and 4741.

Again, two different patterns can be distinguished. Both patterns are negative peaks in the power trace which differ in length compared to reoccurring shorter peaks. Our experiments show that the short peaks appear if there is no counter overflow and the long peaks appear if there is a counter overflow. Thus, it is again possible to map the power trace to bits of the private-key. We highlight the two set bits at positions 4790 and 4741 in Figure 6.8b. In between the two set bits there are 49 small peaks, which translate to 49 zeros in the private-key.

6.4 Countermeasures and Implementation Results

In this section we describe countermeasures that mitigate the attacks presented in Section 6.3 and take other potential information leaks into account as well. The countermeasures are implemented for the STM32F4 microcontroller using the ARM Thumb-2 assembly language to allow full control over the timings and the instruction flow.

6.4.1 Protecting the Encryption

As shown in Section 6.3.2, the encrypted message can be recovered from a single power trace if it is possible to decide whether a row of G is being accumulated or not. Our proposed countermeasure is always to perform an addition to the redundant part, independent of whether the corresponding message bit is set. Of course we cannot simply accumulate all rows of the generator matrix, as this would map all messages to the same codeword.

Since the addition of a row of G to the redundant part is done in 32-bit steps on the ARM microcontroller, we use the current message bit m_i to compute a 32-bit mask $(0-m_i)$. If $m_i = 0$, then the mask is zero, otherwise all 32 bits of the mask are set. Before the 32-bit blocks of the current row of G are XORed to the redundant part, we compute the logical AND of them with the mask. This either results in the current row being added if the message bit is set, or in zero being added if the message bit is not set.

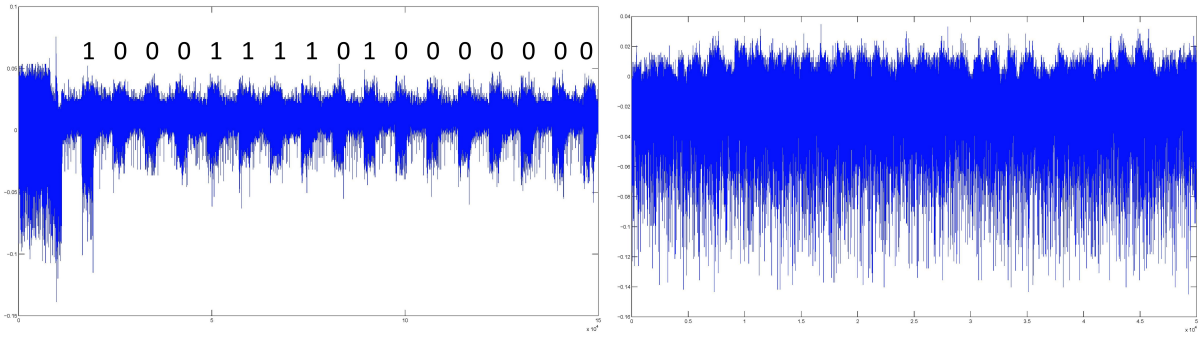
This countermeasure leads to a runtime that is independent of the message and the public-key. Furthermore, as the same instructions are executed for set and cleared message bits, a constant program flow is achieved. Hence, it is not possible to extract the message bits from timing information and also not by distinguishing different instruction flows (cf. Fig 6.9a).

6.4.2 Protecting the Decryption

As shown in Section 6.3.3, the private-key leaks while it is being rotated in an unprotected implementation. A possible countermeasure would be to simply refrain from rotating the rows of the private-key and instead to store the full parity-check matrix in memory. However, storing H would require $2 \times (4801 \times 4801)$ bits $= 5.5$ Mbytes. Since this is infeasible on the platform under investigation, we are protecting the rotation of a row of the private-key.

The protected private-key rotation still uses counters that point to set private-key bits, but the concept of having ordered counters is removed and thus we eliminate the need to move counter values after an overflow. We check for an overflow by comparing the incremented counter values to the maximum r. We load the negative flag `N` from the program status register, use it to compute a 32-bit mask $(0 - \mathtt{N})$, and store the logical AND of the counter value and the mask back to the counter. If the counter value is smaller than r, the `N` flag is set and the incremented counter value is stored. Otherwise the `N` flag is zero and the counter is reset to zero.

This countermeasure removes timing dependencies based on overflowed counters and executes the same program flow independent of whether a counter is reset or not. Figure 6.9b shows the same part of the syndrome computation as was shown for the unprotected version in Figure 6.8b.



(a) Power trace of the protected encryption on the STM32F407 microcontroller. The message starts with `0x8F402`, the first bits are given as reference.

(b) Power trace of the protected syndrome computation on the STM32F407 microcontroller. The private-key starts with set bits at positions 4790 and 4741.

Figure 6.9: Power traces recorded during encryption and decryption with enabled countermeasures.

With the leakage mitigation of the private-key rotation one important step towards SPA-resistant implementations is achieved. However, there are more dependencies on secret data when decoding. Even though we are currently not aware of how these dependencies could be exploited, we avoid them in order to harden the implementation against future attacks.

After syndrome computation and after every decoding iteration the syndrome is compared to zero to check whether decoding succeeded. This comparison should be constant-time, as an early abort of the comparison could leak information about the current state of the syndrome (e.g., about the first non-zero position). We implemented the comparison by computing the OR of all 32-bit blocks of the syndrome and then check whether the result is zero or not.

Counting unsatisfied parity-check equations for a ciphertext bit is the same as counting how many bits are set at the same positions in the current row of the private-key and in the syndrome. Since we know the position of set bits in the private-key from the counters that represent the current row of the private-key, we extract the bits of the syndrome at the same positions and accumulate them. This is done by loading the 32-bit part of the syndrome which holds the bit the counter is pointing to and by shifting and masking the 32-bit part such that the bit in question is singled out and moved to the least significant bit position. We accumulate the result which is either 0 or 1. Since we use 16-bit counters for the private-key and operate on a 32-bit architecture, the upper 11 bits can be used to address a 32-bit memory cell of the syndrome. The remaining 5 bits point to the bit position within the cell. This approach computes the number of unsatisfied parity-check equations with an instruction flow and hence a timing that is independent of the syndrome and the current row of the private-key.

Comparing the number of unsatisfied parity-check equations to the threshold for the current decoding iteration is implemented as

$$\text{ge\_u32}(x, y) = (1 \oplus ((x \oplus ((x \oplus y) | ((x - y) \oplus y))) >> 31))$$

which returns 1 if x is greater or equal to y and 0 otherwise in constant time (x and y are assumed to be unsigned 32-bit integers). The result of this comparison decides whether we have

to invert a ciphertext bit and to update the syndrome with the current row of the private-key or not. If an attacker would be able to trace the points in time when these operations are executed, he likely would be able to recover the error that was added to the codeword and hence reconstruct the plaintext from the ciphertext. To circumvent this leakage, we always XOR the ciphertext bit at the current position with the comparison result which is either 1 or 0. In addition, we always perform the syndrome update by XORing the bit that resulted from the comparison to the positions of the syndrome which are stored in the private-key counters. Since an XOR of a value with zero results in the same value, we actually do not change the ciphertext and the syndrome in case the number of unsatisfied parity-check equations is below the decoding threshold but still execute the same instructions.

Last but not least, the decoding algorithm lasts a variable number of iterations before it terminates. In most cases decoding is finished after two or three decoding iterations (on average 2.4 iterations, cf. Chapter 4) and in rare cases it requires up to a fixed maximum of five iterations. We remark that it is unclear yet if secret data can be recovered only from the number of decoding iterations. This needs to be investigated in future work. To be on the safe side we propose an implementation where we simply do not test the syndrome for zero after a decoding iteration. The decoding algorithm always performs the specified maximum number of iterations. It automatically stops modifying the ciphertext once the syndrome becomes zero. In combination with the techniques introduced above this leads to a fully constant-time and instruction-invariant implementation of the bit-flipping decoder.

6.4.3 Implementation Results

The results of our implementations are listed in Table 6.1. Encrypting a message takes 42 *ms* and decrypting a ciphertext takes 558 *ms* in a fully constant-time implementation. Key-generation takes 884 *ms* on average, but usually key-generation performance is not an issue on small embedded devices since they generate few (if more than one) key-pairs in their lifetime. The combined code of key-generation, encryption, and decryption requires 5.7 Kbytes (0.6%) flash memory and 2.7 Kbytes (1.4%) SRAM, including the public- and private-key. Since $w << r$ for all QC-MDPC parameter sets, storing the private-key in a sparse representation saves memory and at the same time allows fast row rotations. For the 80-bit parameter set with $n_0 = 2$ we only need $w = 90$ 16-bit counters to store the private-key (1440 bits instead of 9602 bits).

Compared to the vulnerable C implementation of the encryption, we are able to achieve a speed up of 50%, to achieve an execution time and an instruction flow which is independent of secret data, and to generate and add true random error vectors.

Our hardened implementations of the decoder are between 1.1-2.5 times slower than the vulnerable C implementation but mitigate the side-channel attacks from Section 6.3 and take further possible information leaks into account. Version ct_3 is completely constant-time and independent of the ciphertext and private-key. Version ct_2 accelerates the first syndrome computation by skipping accumulations if ciphertext bits are not set. As discussed in Section 6.3.3, the computation only depends on set bits in the ciphertext (selecting which rows of the parity-check matrix are XORed) which is usually assumed to be known to an attacker anyways. Version ct_1 of the decoder tests the syndrome for zero after each decoding iteration and exits if decoding was successful before reaching the maximum iterations. Since it is unknown so far if leaking

the number of decoding iterations helps to recover secret information, we advise against using decoder version ct_1 despite its performance advantage.

Compared to the QC-MDPC McEliece implementation in [HvMG13], our encryption function is 20 times faster and includes true random error additions. Decryption performance is improved to a much more realistic 251-558 ms instead of 2.7 s. Furthermore, our implementations are protected against timing attacks and simple power analysis attacks. Other McEliece microcontroller implementations based on Goppa codes [EGHP09, Hey11] and Srivastava codes [CHP12] have much higher memory requirements and all need more time per operation.

An instantiation of the Niederreiter cryptosystem with CS-MDPC codes was presented in [BBMR14] along with a microcontroller implementation targeting a PIC24FJ32 microcontroller clocked at 32 MHz. Their memory requirements are similar to our implementation, key generation and encryption perform similar as well. With 2.8 s their decryption routine is around five times slower. Two important remarks have to be made regarding this implementation. First, it is not designed to run in constant-time or with an invariant instruction flow and is hence not protected against timing and simple power analysis attacks. Second, as recently shown in [Per14], the CS-MDPC parameters as proposed in [BBMR14] do not reach the claimed security levels. For an 80-bit security level the former 128-bit CS-MDPC parameter set has to be used which expands the CS-MDPC public-key from 3,072 to 7,232 bits (4,801 bits for QC-MDPC) and the ciphertext from 9,216 bits to 21,696 bits (9,602 bits for QC-MDPC). Due to this fact, CS-MDPC codes lose their advantage in terms of public-key and ciphertext size to QC-MDPC codes. Furthermore, the performance of key-generation, encryption, and decryption implementations with adjusted parameters will be much lower compared to the results presented in [BBMR14].

Microcontroller implementations of the pre-quantum cryptosystems RSA and ECC were presented for an ATmega128 microcontroller by [LGK10, LWG14]. ECC is somewhat competitive in terms of cycles, but takes more time due to the slower platform. RSA is clearly beaten by QC-MDPC McEliece in terms of performance.

Note that the microarchitecture of the STM32F407 used in this work and the ATxmega256 in [HvMG13] are completely different – but similarly expensive in terms of cost which is usually the most relevant factor for practical applications. The implementations are made available online to allow independent verification and refinement of our results[3].

6.5 QC-MDPC McEliece on General-Purpose Processors

Next we present a vectorized implementation of the QC-MDPC McEliece encryption scheme for general-purpose processors. The target platform is an Intel Core i7-4770 CPU running at 3.40 GHz. The CPU is based on the Haswell architecture and provides a true random number generator (TRNG) which complies to the standards NIST SP800-90A, B, and C, as well as FIPS-140-2 and ANSI X9.82 [Int14]. We employ the TRNG to provide randomness for the key- and error-generation.

[3] `http://www.sha.rub.de/research/projects/code/`

Our vectorized implementation targets modern processors that support the Streaming SIMD Extensions 4 (SSE4). We also develop an unvectorized implementation that can be run systems which do not offer SSE4. In addition, the unvectorized implementation serves as a baseline to evaluate the achieved speed-ups using vector instructions. Our implementations are written in C and we make use of several intrinsic functions to access SSE4 instructions in the vectorized implementation. The software implementations support multiple parameter sets, which allows to easily switch from the 80-bit security parameters to parameter sets that are designed for 128-bit or 256-bit security levels (cf. Section 3.5). The following description mainly focuses on the vectorization of QC-MDPC McEliece.

6.5.1 Vectorized Implementation of QC-MDPC McEliece

While SSE4 features 128-bit integer vectors, Haswell processors also support the AVX2 instruction set which is capable of handling 256-bit integer vectors. However, to ensure a wider compatibility of our vectorized implementation, we decided to apply SSE4. Additionally, we exploit the carry-less multiplication instruction `CLMUL` to accelerate the implementation. This instruction operates on 128-bit vectors and hence would imply expensive conversions if our implementation would be based on 256-bit vectors.

The CPU-internal TRNG has a hardware entropy source that samples thermal noise. The output of this entropy source is used as input for an AES-CBC-MAC that generates the seed for a deterministic random bit generator (DRBG). The DRBG then provides 16, 32 or 64 random bits when calling the corresponding `RDRAND` instruction.

Key Generation

Keys are generated similarly as for the microcontroller implementations with $n_0 = 2$. We generate a random, invertible first row of $H_{n_0-1} = H_1$ with $w/n_0 = w/2$ set bits, a similarly random first row of H_0 and compute the corresponding first row of G. Since the Intel CPU has access to much more memory than the microcontrollers, we do not use a compressed sparse representation for the private-key. All polynomials are stored in full length and we generate the complete matrix H to avoid polynomial shifts during decryption. Since the public matrix G has to be transmitted between communication partners and possibly several different public-keys have to be stored, we do not expand G yet.

Encryption

Encryption of a message starts by first expanding public-key G. This speeds up the actual encryption and is done only once per public-key. All following encryptions under the same public-key reuse the already expanded matrix. In contrast to our other implementations, we rotate the first row by 64 positions and store the result. We repeat this step $\lceil N/64 \rceil$ times and end up with a matrix 64 times smaller than a fully expanded generator matrix.

When multiplying the message by the public-key, the omitted intermediate rotations are performed implicitly using the `CLMUL` instruction that performs a carry-less multiplication of two

64-bit values and returns the 128-bit result. By replacing the bit-by-bit checks with this instruction and working with 128-bit vectors, we are able to accelerate the vector-matrix multiplication by 25 times compared to the unvectorized implementation of this subroutine. Additionally, using the `CLMUL` instruction avoids the previously discussed timing dependency on secret data as the carry-less multiplication is always executed and has a constant reciprocal throughput.

Afterwards, we append the computed redundant part to the message and add a random error vector of weight t. We generate a 64-bit random number using the `RDRAND` instruction and derive four 14-bit, four 15-bit or three 17-bit indexes for the 80/128/256-bit parameter sets, respectively (cf. Section 3.5). If a resulting index i is in the range $0 \leq i < n$, we invert the codeword bit at index i and repeat until t bits are flipped (i.e., rejection sampling).

Decryption

Decoder $\mathcal{D}_2$ is used in this implementation (cf. Section 4.4.1). We first compute the syndrome of the ciphertext. Similar to the multiplication of the plaintext by the public-key, we employ the `CLMUL` instruction to avoid bit-by-bit checks. Since the private-key matrix has been generated by rotating two independent polynomials, the two halves of H are stored separately. Therefore, we have to pay attention to the center element of the ciphertext. As we are processing 64 bits of the ciphertext at once, the center element has to be multiplied with H_0 and H_1. While multiplying the center element by H_0, the bits of the center element that will be multiplied by H_1 have to be set to zero, and vice versa. Compared to the unvectorized implementation of the syndrome computation we achieve a 44 times improved performance with this approach. This even exceeds the speed-up with respect to the multiplication during encryption, since the unvectorized implementation computes the syndrome using a sparse and compressed representation of H.

The plaintext is recovered similarly to the microcontroller implementation. We check whether the syndrome is zero or not. If not, then we identify the number of violated parity-check equations for each ciphertext bit. For this purpose, we employ the `POPCNT` instruction that returns the Hamming weight of a 64-bit word. The number of violated equations is compared to a precomputed threshold. If it exceeds the threshold, we flip the responsible bit and the corresponding row of the private-key matrix is added to the syndrome. In case the syndrome is not zero after reaching the maximum number of decoding iterations, we slightly increase the decoding thresholds and start another decryption attempt. Note, there are no table look-ups depending on secret data in our implementation to reduce the risk of cache timing attacks.

6.5.2 Implementation Results

Table 6.2 lists the cycle counts of our vectorized and non-vectorized implementations for parameter sets designed for 80/128/256-bit equivalent symmetric security. Using vectorization turns out to be significantly faster than using a non-vectorized implementation. Key generation is accelerated by a factor of 2; encryption is nearly 10 times faster; and decryption is 3 times faster. The vectorization speeds up almost all subroutines, except for the error addition which is slower since it uses true random number generation.

The cycle counts naturally rise for higher security levels. Increasing the security level from 80 to 128 bits incurs a performance penalty of a factor of 3-6. The cycle counts for the 256-bit security level are about 10 times higher compared to the 128-bit security level.

Table 6.3 compares our work with implementations of other public-key cryptosystems on similar platforms. The eBACS benchmarking project [eBA15a] contains a McEliece implementation by [BS08] (`mceliece`), RSA implementations (`ronald1024`, `ronald3072`) and an NTRU implementation (`ntruees787ep1`). Compared to the binary Goppa code McEliece implementation by [BS08], our implementation operates twice as fast for encryption and around three times slower for decryption. With respect to the cycles per byte metric, QC-MDPC McEliece benefits from its larger block sizes although encrypting large data using public-key schemes is a rare use case. Again, public-keys are considerably larger than for QC-MDPC codes. The NTRU implementation is only reported for a 256-bit security level and requires less cycles for one operation at this security level. The optimized implementation of the KEM/DEM scheme based on the Niederreiter cryptosystem with Goppa codes by [BCS13] is able to decrypt faster compared to our QC-MDPC implementation. Unfortunately, their cycle counts for key generation and encryption are not reported. For real-world applications, public-key sizes still play an important role since they need to be transferred to remote parties and are stored in embedded devices. At a security level of 128-bit, QC-MDPC McEliece has public-keys of size 1.2 Kbytes while the Goppa code-based Niederreiter implemented by [BCS13] has a public-key of 221 Kbytes.

6.6 Conclusion

In this chapter we presented implementations of QC-MDPC McEliece key-generation, encryption, and decryption providing 80 bits of equivalent symmetric security on low-cost ARM Cortex-M4-based microcontrollers with a reasonable performance for encryption and decryption, respectively. We demonstrated side-channel attacks on a straightforward implementation of this scheme and proposed timing- and instruction-invariant coding strategies and countermeasures to strengthen it against timing attacks and simple power analysis. Furthermore, we presented implementations of QC-MDPC McEliece on general-purpose CPUs and showed how SSE4 vector instruction can be employed to speed-up the computations significantly. Future work includes investigations with respect to fault-injection attacks and with respect to deriving private-key information from only knowing how many iterations are required to decode known or choosable ciphertexts.

Table 6.1: Results of our microcontroller implementations of the QC-MDPC McEliece (McE) cryptosystem. The compiler optimization level was set to `-O2` which gave the best code-size/performance trade-off. [1]Flash and SRAM memory requirements are reported for a combined implementation of key generation, encryption, and decryption. Our constant-time (ct) decoder ct_3 runs completely in constant-time. Decoder ct_2 skips row accumulations during syndrome computation if ciphertext bits are not set. Decoder ct_1 tests the syndrome for zero after each decoding iteration.

Scheme	**Platform**	**SRAM**	**Flash**	**Cycles/Op**	**Time/Op**
This work [enc]	STM32F407	2.7 Kbytes[1]	4,1 Kbytes[1]	16,771,239	100 ms
This work [dec]	STM32F407	2.7 Kbytes[1]	4,1 Kbytes[1]	37,171,833	221 ms
This work [enc, ct]	STM32F407	2.7 Kbytes[1]	5.7 Kbytes[1]	7,018,493	42 ms
This work [dec, ct_1]	STM32F407	2.7 Kbytes[1]	5.7 Kbytes[1]	42,129,589	251 ms
This work [dec, ct_2]	STM32F407	2.7 Kbytes[1]	5.7 Kbytes[1]	85,571,555	509 ms
This work [dec, ct_3]	STM32F407	2.7 Kbytes[1]	5.7 Kbytes[1]	93,745,754	558 ms
This work [keygen]	STM32F407	2.7 Kbytes[1]	5.7 Kbytes[1]	148,576,008	884 ms
McE [enc] [HvMG13]	ATxmega256	606 Bytes	5.5 Kbytes	26,767,463	836 ms
McE [dec] [HvMG13]	ATxmega256	198 Bytes	2.2 Kbytes	86,874,388	2.71 s
McE [enc] [EGHP09]	ATxmega256	512 Bytes	438 Kbytes	14,406,080	450 ms
McE [dec] [EGHP09]	ATxmega256	12 Kbytes	130.4 Kbytes	19,751,094	617 ms
McE [enc] [Hey11]	ATxmega256	3.5 Kbytes	11 Kbytes	6,358,400	199 ms
McE [dec] [Hey11]	ATxmega256	8.6 Kbytes	156 Kbytes	33,536,000	1.1 s
McE [enc] [CHP12]	ATxmega256	-	-	4,171,734	130 ms
McE [dec] [CHP12]	ATxmega256	-	-	14,497,587	453 ms
Nie [enc] [BBMR14]	PIC24FJ32	2.6 Kbytes[1]	5.6 Kbytes[1]	7,200,000	900 ms
Nie [enc] [BBMR14]	PIC24FJ32	2.6 Kbytes[1]	5.6 Kbytes[1]	200,000	25 ms
Nie [dec] [BBMR14]	PIC24FJ32	2.6 Kbytes[1]	5.6 Kbytes[1]	22,400,000	2,800 ms
ECC-P160 [LWG14]	ATmega128	556 Bytes	14.7 Kbytes	9,044,084	1,220 ms
RSA-1024 [LGK10]	ATmega128	-	-	75,680,000	10.3 s

Table 6.2: Cycle counts of our QC-MDPC McEliece implementations on an Intel Core i7-4770 CPU for 100,000 runs en-/decryption and 1,000 runs for the key generation. The compiler optimization level was set to `-O3` since we aim to optimize our implementation for speed. TurboBoost and hyper-threading were disabled during measurements.

Operation	**80-bit non-vectorized**	**80-bit SSE4**	**128-bit SSE4**	**256-bit SSE4**
Key Generation	32,139,668	14,234,347	54,379,733	526,096,652
Encryption	292,432	34,123	106,871	971,605
Decryption	10,114,096	3,104,624	18,825,103	193,922,410
Multiply by public-key	267,913	10,742	44,114	478,152
Add Error	2,528	11,761	18,837	50,114
Compute Syndrome	1,178,512	26,654	95,144	959,382
Rotate left by one position	586	115	196	562
Rotate left sparse	288	-	-	-
AND+Hamming weight	3,723	123	233	735

Table 6.3: Comparison of our QC-MDPC McEliece PC implementation with other McEliece, RSA, and NTRU implementations. We list the required cycles to en-/decrypt one block as well as the required cycles/byte. *eBACS reports cycles for en-/decrypting 59 bytes. We scaled the cycles/byte metric to the full block size.

Implementation	**Platform**	**Sec.** [bits]	**Enc.** [cycles]	**Dec.** [cycles]	**Enc.** [cyc./byte]	**Dec.** [cyc./byte]	**Blocks** [bits]
This work	Haswell	80	34,123	3,104,624	56.86	5,173	4801
This work	Haswell	128	106,871	18,825,103	86.74	15,278	9857
This work	Haswell	256	971,605	193,922,410	237.19	47,340	32771
McBits [BCS13]	Ivy Bridge	81	-	24,051	-	109.88	1751
McBits [BCS13]	Ivy Bridge	129	-	60,493	-	134.27	3604
McBits [BCS13]	Ivy Bridge	263	-	306,102	-	452.40	5413
`mceliece` [eBA15a]	Haswell	83	63,522	1,139,808	300*	5,376*	1696
`ronald1024` [eBA15a]	Haswell	80	45,452	1,288,172	355*	10,064*	1024
`ronald3072` [eBA15a]	Haswell	128	165,832	15,181,669	432*	39,536*	3072
`ntruees787ep1` [eBA15a]	Haswell	256	322,240	513,852	4,958*	7,905*	520

Chapter 7

IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter

Although QC-MDPC McEliece is a promising alternative public-key encryption scheme with practical key sizes and good performance on constrained platforms such as embedded microcontrollers and FPGAs, so far none of the QC-MDPC McEliece/Niederreiter implementations provide indistinguishability under chosen plaintext or chosen ciphertext attacks. In this chapter we close this gap by presenting (1) an efficient implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers, and (2) the first implementation of Persichetti's IND-CCA hybrid encryption scheme instantiated with QC-MDPC Niederreiter for key encapsulation and AES-CBC/AES-CMAC for data encapsulation. Our implementations achieve practical performance: at 80/128-bit security hybrid encryption takes 16.5 ms/83.2 ms, decryption takes 111 ms/477.5 ms and key-generation takes 386.4 ms/1511.8 ms.

This research was presented at PQCrypto'16 [vMHG16] and is a joint work with Lukas Heberle and Tim Güneysu.

Contents

7.1 Introduction

The previous chapters provided novel insights into achieving efficiency when using new codes in the McEliece cryptosystem with improved decoding techniques and optimized implementations. This chapter highlights another important aspect of public-key encryption schemes which is *indistinguishability under chosen-plaintext attacks* (IND-CPA) and *indistinguishability under adaptive chosen-ciphertext attacks* (IND-CCA). These attack models describe the capabilities of different adversaries and allow to formulate and proof security features of public-key encryption schemes.

Our previous implementations of the plain McEliece and Niederreiter cryptosystems do not provide IND-CCA security on their own, using QC-MDPC codes does not change this fact. However, McEliece/Niederreiter can be integrated into existing frameworks which provide IND-CPA or IND-CCA security, e.g., [KI01, NIKM08]. Another approach is to plug Niederreiter into an IND-CCA secure hybrid encryption scheme as recently proposed by Persichetti [Per13]. It is the first hybrid encryption scheme with assumptions from coding theory, and it was proven to provide IND-CCA security and *indistinguishability of keys under adaptive chosen-ciphertext attacks* (IK-CCA) in the random oracle model in [Per13].

Using QC-MDPC codes in code-based cryptography was proposed in [MTSB13] for the McEliece cryptosystem; a corresponding description of QC-MDPC Niederreiter was published in [BBMR14]. Commonly the Niederreiter cryptosystem has the drawback of requiring message transformations into error vectors of fixed weight using constant weight encoding (e.g., [Sen05]) before encryption. However, in a hybrid encryption scheme the public-key encryption scheme just transmits a random symmetric key, hence constant weight encoding can be omitted without affecting security.

Using a hybrid encryption scheme furthermore allows efficient encryption of large plaintexts without the need to share a symmetric secret-key beforehand. Still it is not clear how efficient Persichetti's scheme is in practice, especially when implemented for constrained processors of embedded devices.

Contribution This work provides the first implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers for which we also deploy Persichetti's recently proposed hybrid encryption scheme. We base Persichetti's hybrid encryption scheme on QC-MDPC Niederreiter and extend it with standard symmetric components to handle arbitrary plaintext lengths. Our implementations provide 80-bit and 128-bit security levels and we give an outlook on how to achieve a 256-bit security level.

Outline We summarize the background on QC-MDPC Niederreiter in Section 7.2. Security definitions are given in Section 7.3. Hybrid encryption with Niederreiter based on [Per13] is presented in Section 7.4. Our implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers is detailed in Section 7.5 followed by our implementation of Persichetti's hybrid encryption scheme in Section 7.6. Results and comparisons are given in Section 7.7. We conclude in Section 7.8.

7.2 The QC-MDPC Niederreiter Cryptosystem

We introduce the Niederreiter cryptosystem's key-generation, encryption and decryption based on t-error correcting (n, r, w)-QC-MDPC codes as proposed in [BBMR14] (cf. Section 3.3.3) and provide an overview of how to efficiently decode QC-MDPC codes in the Niederreiter setting.

QC-MDPC Niederreiter Key-Generation

Key-generation requires to generate a (n, r, w)-QC-MDPC code $\mathcal{C}$ with $n = n_0 r$. The private-key is a composed parity-check matrix of the form

$$H = [H_0 \,|\, \ldots \,|\, H_{n_0-1}]$$

which exposes a decoding trapdoor. The public-key is a systematic parity-check matrix

$$H' = [H_{n_0-1}^{-1} \cdot H] = [H_{n_0-1}^{-1} \cdot H_0 \,|\, \ldots \,|\, H_{n_0-1}^{-1} \cdot H_{n_0-2} \,|\, I]$$

which hides the trapdoor but allows to compute syndromes of the public code.

In order to generate a (n, r, w)-QC-MDPC code with $n = n_0 r$, select the first rows $h_0, \ldots, h_{n_0-1}$ of the n_0 parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ at random with Hamming weight $\sum_{i=0}^{n_0-1} \mathrm{wt}(h_i) = w$ and check that H_{n_0-1} is invertible (which is only possible if the row weight d_v is odd). The parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ are generated by $r-1$ quasi-cyclic shifts of the first rows $h_0, \ldots, h_{n_0-1}$. Their concatenation yields the private parity-check matrix H. The public systematic parity-check matrix H' is computed by multiplication of $H_{n_0-1}^{-1}$ with all blocks H_i. Since the public and private parity-check matrices H' and H are quasi-cyclic, it suffices to store their first rows or columns instead of the full matrices. The identity part I of the public-key is usually not stored.

QC-MDPC Niederreiter Encryption

Given a public-key H' and a message $m \in \mathbb{Z}/\binom{n}{t}\mathbb{Z}$, encode m into an error vector $e \in \mathbb{F}_2^n$ with $wt(e) = t$. The ciphertext is the public syndrome

$$s' = He^\intercal \in \mathbb{F}_2^r.$$

QC-MDPC Niederreiter Decryption

Given a public syndrome $s' \in \mathbb{F}_2^r$, recover its error vector using a t-error correcting (QC-)MDPC decoder Ψ_H with private-key H. If

$$e = \Psi_H(s')$$

succeeds, return e and transform it back to message m. On failure of Ψ_H return $\perp$.

Parameters

We use the following parameters in our QC-MDPC Niederreiter implementations as proposed in [MTSB13] for QC-MDPC McEliece. The parameters offer the same security levels for QC-MDPC Niederreiter as explained in [BBMR14]. For an 80-bit security level we use

$$n_0 = 2, n = 9602, r = 4801, w = 90, t = 84.$$

For a 128-bit security level the parameters are

$$n_0 = 2, n = 19714, r = 9857, w = 142, t = 134.$$

By $d_v = w/n_0$ we denote the Hamming weight of each row of the n_0 private parity-check matrix blocks[1]. With these parameters the private parity-check matrix H consists of $n_0 = 2$ circulant blocks, each with constant row weight d_v. The public parity-check matrix H' consists of $n_0 - 1 = 1$ circulant block concatenated with the identity matrix. The public-key has a size of r bits and the private-key has a size of n bits which can be compressed since $w \ll n$. Plaintexts are encoded into vectors of length n and Hamming weight t; ciphertexts have length r.

7.2.1 Decoding for QC-MDPC Niederreiter

Several decoders were evaluated to efficiently decode (QC-)MDPC codes in the McEliece cryptosystem in Chapter 4. We found that bit-flipping decoders as introduced by Gallager in [Gal63] in combination with our proposed improvements are the most suitable decoders for constrained devices. Hence, we transfer the decoder $\mathcal{D}_2$ and several optimizations from QC-MDPC McEliece to the QC-MDPC Niederreiter setting. The Niederreiter decoder is presented in Algorithm 2.

In QC-MDPC Niederreiter the decoder receives a private parity-check matrix H and a public syndrome s' as input and computes the private syndrome $s = H_{n_0-1}s'^{\intercal}$. Decoding runs in several iterations which are summarized as follows: the inner loop iterates over all columns of a block of the private parity-check matrix and counts the number of unsatisfied parity-checks $\#_{\text{upc}}$ by counting the number of shared set bits of each column $H_i[j]$ and the private syndrome s. If $\#_{\text{upc}}$ exceeds a certain threshold[2], the decoder likely has found an error position and inverts the corresponding bit in a zero-initialized error candidate $e_{\text{cand}} \in \mathbb{F}_2^n$, thus the name *bit-flipping* decoder. In addition, we include the optimization of directly updating the syndrome s through an addition of $H_i[j]$ to the syndrome in case of a bit-flip as proposed in Chapter 4. This modification improves the decoding behavior to take less decoding iterations and reduces the probability of decoding failures. Furthermore, decoding is accelerated since syndrome recomputations after decoding iterations are avoided.

The inner loop is repeated for every block H_i of H until all blocks have been processed. Afterwards, the public syndrome of the error candidate is computed and compared to the initial public syndrome s'. On a match, the correct error vector was found and is returned. Otherwise the decoder continues with the next iteration. After a fixed maximum of iterations, decoding is restarted with incremented thresholds as proposed for QC-MDPC McEliece. The failure symbol $\perp$ is returned if even after δ_{max} threshold adaptations the correct error vector is not found.

[1] 80-bit: $d_v = 45$, 128-bit: $d_v = 71$. Note that $n_0 = 2$ and w is even for the parameters used in this chapter.
[2] The bit-flipping thresholds used in Algorithm 2 are precomputed as proposed in [Gal63], cf. Section 4.3.

Algorithm 2 Syndrome Decoder for QC-MDPC codes. Returns Error Vector e or Failure $\perp$.

Input: H, s', $\text{iterations}_{\max}$, $\delta_{\max}$, threshold
Output: e

```
Compute the private syndrome s ← H_{n0-1} s'^T
δ ← 0, e_cand ← 0^n
while δ < δ_max do
  iterations ← 0
  while iterations < iterations_max do
    for i in n0 do
      for j in r do
        hw ← HammingWeight(H_i[j] & s)
        if hw ≥ (threshold[iterations] + δ) then
          e_cand[i·r + j] ← e_cand[i·r + j] ⊕ 1
          s ← H_i[j] ⊕ s
        end if
      end for
    end for
    s'_cand ← H' e_cand^T
    if s' = s'_cand then
      return e ← e_cand
    end if
    iterations ← iterations + 1
  end while
  δ ← δ + 1
  s ← H_{n0-1} s'^T
end while
return ⊥
```

7.3 Background

We present necessary definitions to construct the IND-CCA and IK-CCA secure hybrid encryption scheme of [Per13] on the basis of Niederreiter public-key encryption. Assumptions about the security of the Niederreiter framework and definitions of properties of the plaintext, ciphertext and key indistinguishability for public-key encryption schemes (IND-CPA, IND-CCA, IK-CCA) are introduced. Furthermore, we introduce and define key derivation functions (KDF) and message authentication codes (MAC) together with their desired security property of providing *existential unforgeability under chosen message attacks* (EUF-CMA).

7.3.1 Niederreiter Security Assumptions

The security of the Niederreiter cryptosystem is based on two assumptions, the indistinguishability of scrambled matrices from random matrices and the hardness of the syndrome decoding problem. Note: we will name probabilistic polynomial time algorithms in short as ppt algorithms and negligible functions as $negl()$.

Assumption 7.3.1. *(**Indistinguishability**, based on Assumption 1 in [Per13])*
Let (H, w) be a public-key of the Niederreiter cryptosystem and H be the scrambled $(n-k) \times n$ parity-check matrix of a $[n, k]$ linear code over $\mathbb{F}_q$. Then, H is computationally indistinguishable from a uniformly chosen $(n-k) \times n$ matrix R. The advantage of a ppt attacker A is

$$Adv^{ind}_{A,H}(k) = \Big| Pr[A(R, w) = 1] - Pr[A((H, w) \leftarrow Gen_{NR}(1^k)) = 1] \Big| \leq negl(k)$$

for a sufficiently large k.

Assumption 7.3.2. *(**Syndrome Decoding Problem**, based on Assumption 2 in [Per13])*
Let H be the $(n-k) \times n$ parity-check matrix of a $[n, k]$ linear code over $\mathbb{F}_q$ and $s \in_R \mathbb{F}_q^{(n-k)}$ be chosen uniformly at random. Then, it is hard to find a vector $e \in \mathbb{F}_q^n$ with $\mathrm{wt}(e) \leq w$ such that $He^\intercal = s$. The advantage of a ppt attacker A is

$$Adv^{SDP}_{A,SD_w}(k) = Pr[A(H, w, s) = e, \mathrm{wt}(e) \leq w] \leq negl(k)$$

for a sufficiently large k.

The hardness of the syndrome decoding problem (SDP) was proven to be NP-complete in [BMv78]. Next, we define the IND-CCA and IK-CCA security goals and the corresponding security games in Section 7.3.3 and Section 7.3.4, respectively. We also include the definition of the weaker model of IND-CPA security in Section 7.3.2.

7.3.2 IND-CPA Security

Indistinguishability under chosen-plaintext attacks (IND-CPA) is a property of public-key encryption schemes which aim to provide semantic security. The task for an attacker is to pick two plaintexts, send them to an encryption oracle which randomly encrypts one of the two plaintexts, and then to distinguish which plaintext was encrypted given only the corresponding ciphertext. If there exists no attacker capable of winning this game with a considerably higher probability than the guessing probability of 1/2, the encryption scheme provides IND-CPA security. We formally define IND-CPA security for public-key encryption schemes in the following Definition 7.3.3.

Definition 7.3.3. ***(IND-CPA Security)***
A public-key cryptosystem $\pi = (Gen_\pi, Enc_\pi, Dec_\pi)$ is IND-CPA secure if the advantage $Adv^{IND\text{-}CPA}_{A,\pi}(n)$ of any ppt attacker A is

$$Adv^{IND\text{-}CPA}_{A,\pi}(n) = \Big| Pr[PubK^{IND\text{-}CPA}_{A,\pi}(n) = 1] - \tfrac{1}{2} \Big| \leq negl(n)$$

for a sufficiently large security parameter n.

The security game $\text{PubK}^{\text{IND-CPA}}_{A,\pi}(n)$ used in the definition of IND-CPA security is modeled in Figure 7.1. Attacker A receives the security parameter 1^n and a valid public-key pk from the challenger, picks two messages m_0 and m_1 from the message space M, and sends them back to the challenger. The challenger randomly picks a bit $b \in_R \{0,1\}$, encrypts message m_b under pk to $c = \text{Enc}_{\text{pk}}(m_b)$ and sends the result to A. The attacker has to decide which plaintext $m_{b'}$ was encrypted by the challenger. Attacker A wins the game if he returns bit $b' = b$; the game is lost if $b' \neq b$. This step can be repeated several times by the attacker; the number of repetitions and the computational power of A is only bound to be polynomial in the size of the security parameter. If the attacker's advantage of winning this game is negligible compared to guessing, the encryption scheme provides semantic security.

Note that encryption oracles are inherently available to all attackers who target public-key cryptosystems since encryption can be performed by anyone who has access to public-key pk.

$$
\begin{array}{lcl}
\textbf{PubK}^{\textbf{IND-CPA}}_{A,\pi}(n) & & \textbf{Attacker A} \\
(\text{pk}, \text{sk}) \leftarrow \text{Gen}(1^n) & \xrightarrow{(1^n, \text{pk})} & \\
b \in_R \{0,1\} & \xleftarrow{(m_0, m_1)} & m_0, m_1 \in M \\
c = \text{Enc}_{\text{pk}}(m_b) & \xrightarrow{c} & \\
\text{Output:} & \xleftarrow{b'} & b' \in \{0,1\} \\
= \begin{cases} 1 & b = b' \\ 0 & \text{else} \end{cases} & &
\end{array}
$$

Figure 7.1: The IND-CPA security game $\text{PubK}^{\text{IND-CPA}}_{A,\pi}(n)$.

7.3.3 IND-CCA Security

Indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA) is a stronger property of public-key encryption schemes compared to IND-CPA. Its definition extends the IND-CPA security game by providing attacker A with access to a decryption oracle before and after receiving the challenge ciphertext. The decryption oracle returns the plaintexts of adaptively chosen ciphertexts to the attacker, except the plaintext of the challenge ciphertext. If there exists no ppt attacker capable of winning this game with considerably higher probability than the guessing probability of 1/2, the encryption scheme provides IND-CCA. We formally define IND-CCA security for public-key encryption schemes in Definition 7.3.4.

Definition 7.3.4. ***(IND-CCA Security)***
A public-key cryptosystem $\pi = (Gen_\pi, Enc_\pi, Dec_\pi)$ is IND-CCA secure if the advantage $Adv^{IND\text{-}CCA}_{A,\pi}(n)$ of any ppt attacker A is

$$Adv^{IND\text{-}CCA}_{A,\pi}(n) = \left| Pr[PubK^{IND\text{-}CCA}_{A,\pi}(n) = 1] - \tfrac{1}{2} \right| \leq negl(n)$$

for a sufficiently large security parameter n.

The security game $\text{PubK}_{A,\pi}^{\text{IND-CCA}}(n)$ used in the definition of IND-CCA security is modeled in Figure 7.2. Compared to the IND-CPA security game (cf. Figure 7.1), attacker A can send arbitrary ciphertexts c_i' from the ciphertext space C to the challenger who decrypts them and returns the corresponding plaintexts m_i' before and after receiving the challenge ciphertext c. The trivial exception is that the challenge ciphertext c is not allowed to be queried to the decryption oracle. The number of ciphertexts that can be queried is bound in the security parameter n by polynomials $p(n)$ and $q(n)$. Similar to the IND-CPA security game, the attacker has to decide whether message m_0 or m_1 was encrypted by sending $b' \in \{0,1\}$. The game is won if $b' = b$ and lost if $b' \neq b$. If the attacker's advantage of winning this game is negligible compared to guessing, the encryption scheme provides IND-CCA security.

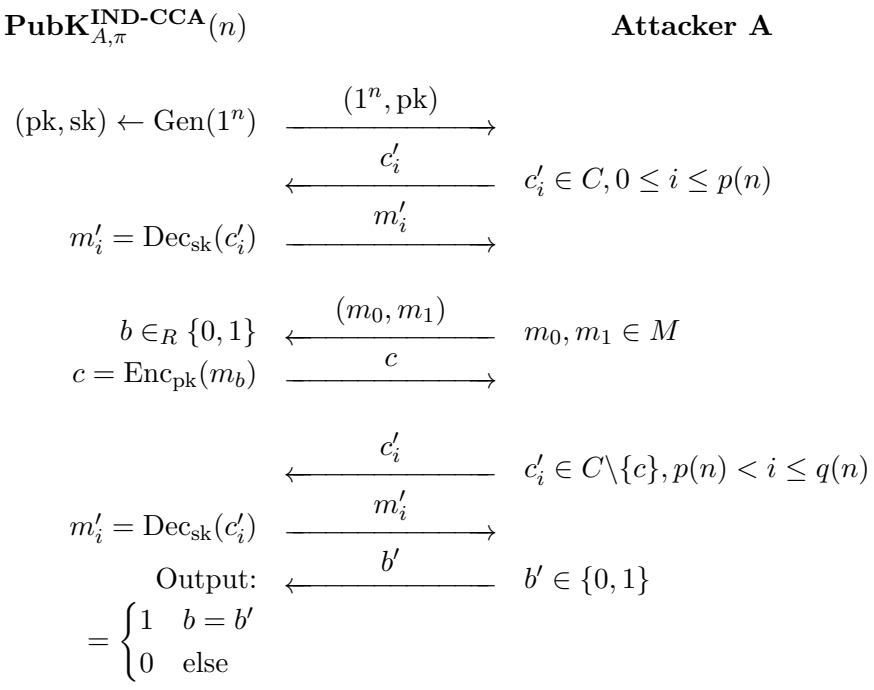


Figure 7.2: The IND-CCA security game $\text{PubK}_{A,\pi}^{\text{IND-CCA}}(n)$.

7.3.4 IK-CCA Security

Indistinguishability of keys under adaptive chosen-ciphertext attacks (IK-CCA) is a property of public-key encryption schemes which aim to provide key privacy, i.e., a ppt attacker is not able to distinguish which public-key out of a set of known public-keys was used to encrypt a message chosen by the adversary. The modeled attacker is similar to the attacker of the IND-CCA security game. Attacker A can perform arbitrary encryptions and is provided with a decryption oracle for two distinct public-key/private-key pairs before and after receiving the challenge ciphertext. We formally define IK-CCA security for public-key encryption schemes in Definition 7.3.5.

Definition 7.3.5. ***(IK-CCA Security)***
A public-key cryptosystem $\pi = (Gen_\pi, Enc_\pi, Dec_\pi)$ provides IK-CCA security if the advantage $Adv^{IK\text{-}CCA}_{A,\pi}(n)$ of any ppt attacker A is

$$Adv^{IK\text{-}CCA}_{A,\pi}(n) = \left| Pr[PubK^{IK\text{-}CCA}_{A,\pi}(n) = 1] - \tfrac{1}{2} \right| \leq negl(n)$$

for a sufficiently large security parameter n.

The security game $\text{PubK}^{\text{IK-CCA}}_{A,\pi}(n)$ used in the definition of IK-CCA security is modeled in Figure 7.3. Compared to the IND-CCA security game (cf. Figure 7.2), the challenger generates two public-key/private-key pairs $((pk_0, sk_0), (pk_1, sk_1))$ in the security parameter 1^n and sends both public-keys to the attacker. The attacker is able to encrypt arbitrary plaintexts under both provided public-keys and in addition can query a decryption oracle with arbitrary ciphertexts $c'_i \in C$, where C is the set of all valid ciphertexts. Furthermore, the attacker can select which private-key $sk_{s_i}, s_i \in \{0,1\}$ shall be used by the decryption oracle to decrypt c'_i to m'_i.

$\mathbf{PubK}^{\mathbf{IK\text{-}CCA}}_{A,\pi}(n)$		**Attacker A**
$(\text{pk}_0, \text{sk}_0) \leftarrow \text{Gen}(1^n)$		
$(\text{pk}_1, \text{sk}_1) \leftarrow \text{Gen}(1^n)$	$\xrightarrow{(1^n, \text{pk}_0, \text{pk}_1)}$	
		$s_i \in \{0,1\}$
	$\xleftarrow{(c'_i, s_i)}$	$c'_i \in C, 0 \leq i \leq p(n)$
$m'_i = \text{Dec}_{\text{sk}_{s_i}}(c'_i)$	$\xrightarrow{m'_i}$	
$b \in_R \{0,1\}$	$\xleftarrow{m}$	$m \in M$
$c = \text{Enc}_{\text{pk}_b}(m)$	$\xrightarrow{c}$	
		$s_i \in \{0,1\}$
	$\xleftarrow{(c'_i, s_i)}$	$c'_i \in C \backslash \{c\}, p(n) < i \leq q(n)$
$m'_i = \text{Dec}_{\text{sk}_{s_i}}(c'_i)$	$\xrightarrow{m'_i}$	
Output:	$\xleftarrow{b'}$	$b' \in \{0,1\}$
$= \begin{cases} 1 & b = b' \\ 0 & \text{else} \end{cases}$		

Figure 7.3: The IK-CCA security game $\text{PubK}^{\text{IK-CCA}}_{A,\pi}(n)$.

The challenge for the attacker in this security game is slightly different from the previous games. This time the attacker selects a specific message m in the message space M to be encrypted by the challenger. The challenger decides by fair coin toss $b \in_R \{0,1\}$ under which public-key pk_b message m is encrypted. The results $c = \text{Enc}_{\text{pk}_b}(m)$ is returned to A who continues querying ciphertexts $c'_i \neq c$ to the decryption oracle. Again, attacker A is bound in

the security parameter n by polynomials $p(n)$ and $q(n)$. Finally, A has to decide whether pk_0 or pk_1 was used to encrypt m. If the attacker returns $b' = b$ the game is won, otherwise if $b' \neq b$ the game is lost. If the attacker's advantage of winning this game is negligible compared to guessing, the encryption scheme provides IK-CCA security.

7.3.5 EUF-CMA Security

Existential unforgeability under adaptive chosen-message attacks (EUF-CMA) is a property of cryptographic signature schemes which was introduced in [GMR88]. A forger is given a public-key and access to a signing oracle $\mathrm{Sig}_{\mathrm{sk}}(\cdot)$. The forger's task is to output a signature σ for a message m which has not been queried to the oracle and which successfully verifies with $\mathrm{Ver}_{\mathrm{pk}}(m, \sigma) = 1$. We formally define EUF-CMA security in Definition 7.3.6.

Definition 7.3.6. ***(EUF-CMA Security)***
A signature scheme $\pi = (Gen_\pi, Sig_\pi, Ver_\pi)$ is EUF-CMA secure if the advantage $Adv_{F,\pi}^{EUF\text{-}CMA}(n)$ of any ppt forger F is

$$Adv_{F,\pi}^{EUF\text{-}CMA}(n) = Pr[Sig_{F,\pi}^{EUF\text{-}CMA}(n) = 1] \leq negl(n)$$

for a sufficiently large security parameter n.

The security game $\mathrm{Sig}_{F,\pi}^{\mathrm{EUF\text{-}CMA}}(n)$ used in the definition of EUF-CMA security is modeled in Figure 7.4. After the challenger generates a key-pair (pk, sk), he sends the public-key and the security parameter to the forger. Afterwards, F selects arbitrary messages m_i from the message space M and queries the signature oracle to provide valid signatures for the messages m_i. The number of queries is bound in the security parameter by polynomial $p(n)$. In the end, F has to return a message m together with a forged signature σ which was not queried to the signature oracle. The game is won if the verification of the forged signature σ succeeds under public-key pk for message m, i.e., $\mathrm{Ver}_{\mathrm{pk}}(m, \sigma) = 1$, otherwise the game is lost. If the forger's advantage of winning this game is negligible, the signature scheme is EUF-CMA secure.

7.3.6 Key Derivation Functions

In addition to the security properties, we define key derivation functions (KDF) which will be used to compute symmetric keys in the hybrid encryption scheme.

Definition 7.3.7. ***(Key Derivation Function**, based on Definition 3 in [Per13])*
Let x be a string of arbitrary length and l be a positive integer. A function KDF(x,l) is a key derivation function which outputs a bit string y of length l that is computationally indistinguishable from a random bit string r of the same length l. The advantage of a ppt attacker A is

$$Adv_{A,KDF}^{KDF}(n) = |Pr[A(x, l, y) = 1] - Pr[A(x, l, r) = 1]| \leq negl(n)$$

for a sufficiently large security parameter n.

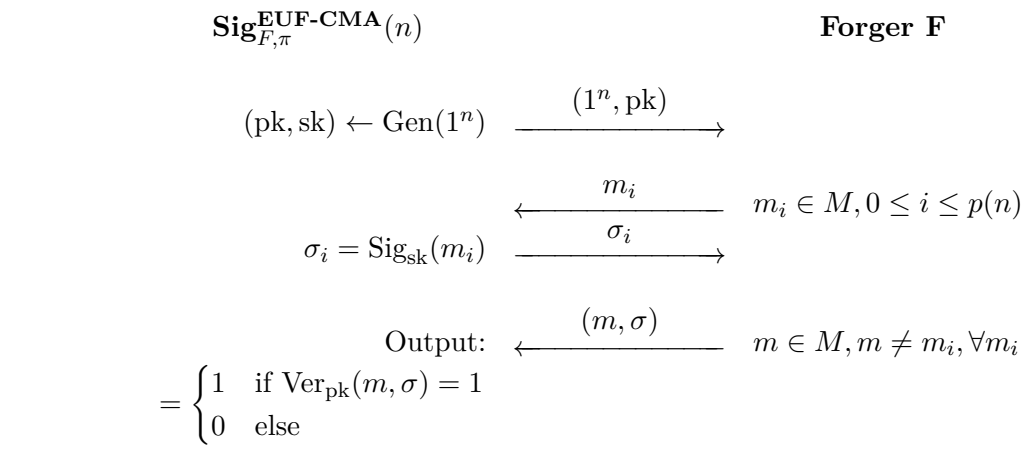


Figure 7.4: The EUF-CMA security game $\mathrm{Sig}_{F,\pi}^{\mathrm{EUF\text{-}CMA}}(n)$.

7.3.7 Message Authentication Codes

Furthermore, we define message authentication codes (MAC) to authenticate the symmetric ciphertexts in the hybrid encryption scheme. Message authentication codes are desired to provide existential unforgeability against chosen message attacks (cf. Figure 7.4).

Definition 7.3.8. *(**Message Authentication Code**, based on Definition 4 in [Per13])*
An algorithm that authenticates a message by a short tag is called a message authentication code. It is defined by a function $Ev(k, T)$ that takes as input a key k of length l_{MAC} and a string T of arbitrary length. Function $Ev(k, T)$ returns an authentication tag τ of length l_{TAG} which is appended to the message.

Definition 7.3.9. *(**EUF-CMA MAC Security**)*
A message authentication scheme $\pi = (Ev_\pi)$ provides EUF-CMA security if the advantage $Adv_{F,\pi}^{EUF\text{-}CMA}(n)$ of any ppt forger F is

$$Adv_{F,\pi}^{EUF\text{-}CMA}(n) = Pr[Sig_{F,\pi}^{EUF\text{-}CMA}(n) = 1] \leq negl(n)$$

for a sufficiently large security parameter n.

7.4 Niederreiter Hybrid Encryption

Hybrid encryption schemes were introduced in [CS03]. They are divided into two independent components: a key encapsulation mechanism (KEM) and a data encapsulation mechanism (DEM). The KEM is a public-key encryption scheme which encrypts randomly generated session keys under the public-key of the intended receiver. The DEM then encrypts the plaintexts under the randomly generated session keys using a symmetric encryption scheme. Hybrid encryption is beneficial in practice since symmetric encryption is orders of magnitude more efficient than public-key encryption, especially for large plaintexts. On the other hand symmetric schemes alone are not practical due to their key distribution problem. Hybrid encryption benefits from efficient symmetric encryption and asymmetric key distribution.

7.4.1 Key and Data Encapsulation Mechanisms

A general hybrid encryption scheme consists of a key encapsulation mechanism combined with a data encapsulation mechanism. The KEM generates and encrypts a symmetric key which is used by the DEM to encrypt a message. The KEM is a public-key encryption scheme, while a symmetric encryption scheme is used for the DEM.

Definition 7.4.1. *(**Hybrid Encryption Scheme**, based on [Per13])*
A hybrid encryption scheme $\pi_{HY} = (\pi_{KEM}, \pi_{DEM})$ consists of a KEM $\pi_{KEM} = (Gen_{KEM}, Enc_{KEM}, Dec_{KEM})$ and a DEM $\pi_{DEM} = (Enc_{DEM}, Dec_{DEM})$ defined as follows:

- ***Gen_{KEM}** is a probabilistic key generation algorithm that generates a public-key key pair (pk, sk) from the security parameter 1^n.*
- ***Enc_{KEM}** is a probabilistic public-key encryption algorithm that generates a random symmetric key k of length l_k and encrypts this key under public-key pk. It returns the symmetric key and the ciphertext (k, c).*
- ***Dec_{KEM}** is a deterministic public-key decryption algorithm that decrypts ciphertext c with private-key sk. It either returns the decrypted symmetric key k or failure symbol $\perp$.*
- ***Enc_{DEM}** is a deterministic symmetric encryption algorithm that encrypts a plaintext m under symmetric key k and returns its ciphertext c^*.*
- ***Dec_{DEM}** is a deterministic symmetric decryption algorithm that decrypts plaintext m from ciphertext c^* using key k. It either outputs the plaintext or failure symbol $\perp$.*

A hybrid scheme $\pi_{HY} = (Gen_{HY}, Enc_{HY}, Dec_{HY})$ then is a combination of the aforementioned algorithms of π_{KEM} and π_{DEM}.

- ***Gen_{HY}** invokes Gen_{KEM} and returns the resulting key-pair.*
- ***Enc_{HY}** receives the public-key pk and a plaintext m as input. First, the algorithm invokes $Enc_{KEM,pk}()$ and obtains a secret-key and its ciphertext (k, c). Then, plaintext m is encrypted by invoking $Enc_{DEM,k}(m)$. The resulting ciphertext c^* is concatenated with c and is output as $\tilde{c} = (c \,||\, c^*)$.*
- ***Dec_{HY}** receives ciphertext $\tilde{c}$ and the private-key sk. It divides $\tilde{c}$ into c and c^* and recovers the symmetric key k by invoking $Dec_{KEM,sk}(c)$. If the failure symbol $\perp$ is returned, Dec_{HY} returns $\perp$ as well. Afterwards, the plaintext m is decrypted by invoking $Dec_{DEM,k}(c^*)$. The output is either the decrypted plaintext or failure symbol $\perp$.*

In summary, a hybrid encryption scheme consists of:

- **Key generation:** $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathrm{Gen}_{\mathrm{HY}}(1^n)$, returning $(\mathrm{pk}, \mathrm{sk})$.
- **Encryption:** $(k, c) \leftarrow \mathrm{Enc}_{\mathrm{KEM,pk}}()$, $c^* \leftarrow \mathrm{Enc}_{\mathrm{DEM},k}(m)$, returning $(c \,||\, c^*)$.
- **Decryption:** $k = \mathrm{Dec}_{\mathrm{KEM,sk}}(c)$, $m = \mathrm{Dec}_{\mathrm{DEM},k}(c^*)$, returning m or $\perp$.

It was proven in [CS03] that the advantage of any ppt attacker on the IND-CCA security of the hybrid scheme is at most the sum of the advantages of two ppt attackers A_1 and A_2 on the IND-CCA security of the KEM and DEM, respectively (see Theorem 7.4.2). The hybrid scheme can be proven IND-CCA secure in the random oracle model even if the public-key encryption scheme used as KEM itself is not IND-CCA secure.

Theorem 7.4.2. *(**Security of Hybrid Encryption Schemes**, based on [CS03])*
A hybrid encryption scheme $\pi_{HY} = (\pi_{KEM}, \pi_{DEM})$ is IND-CCA secure if π_{KEM} and π_{DEM} are IND-CCA secure. For any ppt attacker A, there exist two attackers A_1 and A_2 such that

$$Adv^{IND\text{-}CCA}_{A,\pi_{HY}}(n) \leq Adv^{IND\text{-}CCA}_{A_1,\pi_{KEM}}(n) + Adv^{IND\text{-}CCA}_{A_2,\pi_{DEM}}(n).$$

The advantages $\mathrm{Adv}^{\text{IND-CCA}}_{A_1,\pi_{\text{KEM}}}(n)$ and $\mathrm{Adv}^{\text{IND-CCA}}_{A_2,\pi_{\text{DEM}}}(n)$ of attackers A_1 and A_2 on the IND-CCA security of the KEM and DEM are defined next.

Definition 7.4.3. *(**IND-CCA Security of KEMs**)*
A public-key KEM $\pi_{KEM} = (Gen_{KEM}, Enc_{KEM}, Dec_{KEM})$ is IND-CCA secure if the advantage of any ppt attacker A_1 is

$$Adv^{IND\text{-}CCA}_{A_1,\pi_{KEM}}(n) = \left| Pr[KEM^{IND\text{-}CCA}_{A_1,\pi_{KEM}}(n) = 1] - \tfrac{1}{2} \right| \leq negl(n)$$

for a sufficiently large security parameter n.

Definition 7.4.4. *(**IND-CCA Security of DEMs**)*
A symmetric DEM $\pi_{DEM} = (Enc_{DEM}, Dec_{DEM})$ is IND-CCA secure if the advantage of any ppt attacker A_2 is

$$Adv^{IND\text{-}CCA}_{A_2,\pi_{DEM}}(n) = \left| Pr[DEM^{IND\text{-}CCA}_{A_2,\pi_{DEM}}(n) = 1] - \tfrac{1}{2} \right| \leq negl(n)$$

for a sufficiently large security parameter n.

The security game $\mathrm{KEM}^{\text{IND-CCA}}_{A_1,\pi_{\text{KEM}}}$ used in the definition of the IND-CCA security of KEMs is modeled in Figure 7.5. The challenger generates a key-pair (pk, sk) and provides the attacker with the security parameter and the public-key. As in earlier IND-CCA security games, the attacker is allowed to send arbitrary ciphertexts c'_i from the ciphertext space C to a decryption oracle $\mathrm{Dec}_{\text{sk}}(\cdot)$ before and after receiving the challenge. The number is again bound in the security parameter by polynomials $p(n)$ and $q(n)$. To generate the challenge of this game, the challenger invokes the KEM's encryption algorithm $\mathrm{Enc}_{\text{pk}}()$ and receives (k, c). Then he generates $b \in_R \{0,1\}$ by fair coin toss. If $b = 1$ he sets $k^* = k$, else k^* is set to a uniform random string of the same length as k. The challenge (k^*, c) is sent to the attacker who has to decide whether k^* is the correct plaintext for ciphertext c or if k^* is simply a random string. The game is won by A_1 if $b' = b$, otherwise the game is lost. A KEM is IND-CCA secure if there exists no ppt attacker A_1 with a considerably higher probability of winning this game than the guessing probability $1/2$.

The security game $\mathrm{DEM}^{\text{IND-CCA}}_{A_2,\pi_{\text{DEM}}}$ used in the definition of the IND-CCA security of a DEM is equivalent to the security game $\mathrm{PrivK}^{\text{IND-CCA}}_{A,\pi}$ of the IND-CCA security of symmetric encryption schemes which in turn is similar to the security game $\mathrm{PubK}^{\text{IND-CCA}}_{A,\pi}$ of the IND-CCA security of public-key encryption schemes (cf. Figure 7.2). The only difference is the addition of a second oracle that provides arbitrary encryptions of adaptively chosen messages to the attacker. This addition is necessary because the attacker cannot perform symmetric encryptions without knowing the symmetric key. In the public-key setting encryptions can be performed by

anyone using the public-key. Otherwise the security games are the same and thus a symmetric encryption schemes is IND-CCA secure if there exists no ppt attacker A_2 with a considerably higher probability of winning $\text{PrivK}^{\text{IND-CCA}}_{A_2,\pi}$ than the guessing probability of 1/2. Hence, a DEM provides IND-CCA if the symmetric encryption scheme provides IND-CCA. In addition, [CS03] showed that it is possible to construct an IND-CCA symmetric encryption scheme from an IND-CPA symmetric encryption scheme by combination with a secure one-time MAC.

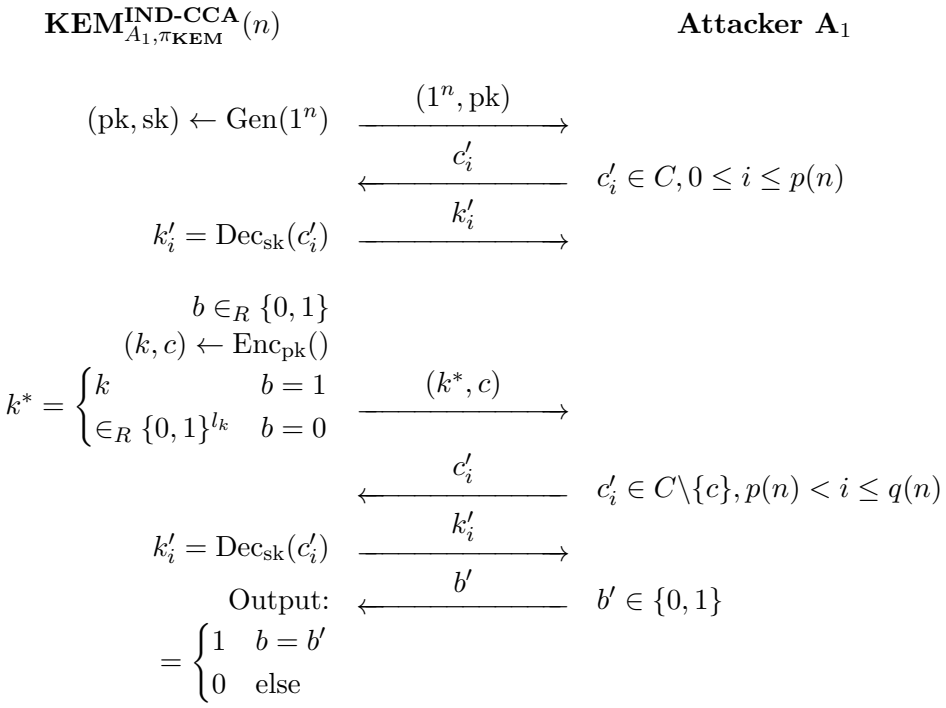


Figure 7.5: The KEM IND-CCA security game $\text{KEM}^{\text{IND-CCA}}_{A_1,\pi_{\text{KEM}}}(n)$.

7.4.2 Constructing Hybrid Encryption from Niederreiter

We introduce the Niederreiter hybrid encryption scheme as proposed in [Per13] in which Persichetti focuses on the realization of an IND-CCA secure KEM and assumes being provided with an IND-CCA symmetric encryption scheme for the DEM.

The Niederreiter KEM

Let $\mathcal{F}$ be the family of t-error correcting $[n,k]$-linear codes over $\mathbb{F}_q$ and let n, k, q, t be fixed system parameters. The Niederreiter KEM $\pi_{\text{NR\_KEM}} = (\text{Gen}_{\text{NR\_KEM}}, \text{Enc}_{\text{NR\_KEM}}, \text{Dec}_{\text{NR\_KEM}})$ follows the definition of a generic Niederreiter scheme.

- $\mathbf{Gen}_{\mathbf{NR\_KEM}}$ Pick a random code $\mathcal{C} \in \mathcal{F}$ with parity-check matrix $H' = (M \,|\, I_{n-k})$. Output H' (or M) as public-key and the code description Δ as private-key.

- **$\mathbf{Enc_{NR\_KEM}}$** Given a public-key H', generate a random error $e \in_R \mathbb{F}_q^n$ of weight $\mathrm{wt}(e) = t$ and compute its public syndrome $s' = H'e^\intercal$. The symmetric key k of length l_k is generated from e by a key-derivation function as $k = (k_1 \,||\, k_2) = \mathrm{KDF}(e, l_k)$. The output is (k, s').
- **$\mathbf{Dec_{NR\_KEM}}$** Decode ciphertext s' to $e = \Psi_\Delta(s')$ using the code description Δ and decoding algorithm Ψ. Derive symmetric key $k = \mathrm{KDF}(e, l_k)$ if decoding succeeds. Otherwise, k is set to a pseudorandom string of length l_k, [Per13] suggests to set $k = \mathrm{KDF}(s', l_k)$.

The Standard DEM

Let $\mathrm{Enc}_{k_1}^{\mathrm{SE}}(\cdot)$ and $\mathrm{Dec}_{k_1}^{\mathrm{SE}}(\cdot)$ denote the en-/decryption operations of a symmetric encryption scheme under key k_1 and let $\mathrm{Ev}_{k_2}(\cdot)$ denote the evaluation of a keyed message authentication code under key k_2 which returns a fixed length message authentication tag τ. The standard DEM $\pi_{\mathrm{DEM}} = (\mathrm{Enc}_{\mathrm{DEM}}, \mathrm{Dec}_{\mathrm{DEM}})$ is the combination of a symmetric encryption scheme with a message authentication code[3].

- **$\mathbf{Enc_{DEM}}$** Given a plaintext m and key $k = (k_1 \,||\, k_2)$, encrypt m to $T = \mathrm{Enc}_{k_1}^{\mathrm{SE}}(m)$ and compute the message authentication tag $\tau = \mathrm{Ev}_{k_2}(T)$ of ciphertext T under k_2. The output is $c^* = (T \,||\, \tau)$.
- **$\mathbf{Dec_{DEM}}$** Given a ciphertext c^* and key k, split c^* into T, τ and k into k_1, k_2. Verify the correctness of the MAC by evaluating $\mathrm{Ev}_{k_2}(T) \stackrel{?}{=} \tau$. If the MAC is correct, plaintext $m = \mathrm{Dec}_{k_1}^{\mathrm{SE}}(T)$ is decrypted and returned. In case of a MAC mismatch, $\perp$ is returned.

The Niederreiter Hybrid Encryption Scheme

The Niederreiter hybrid encryption scheme $\pi_{\mathrm{HY}} = (\mathrm{Gen}_{\mathrm{HY}}, \mathrm{Enc}_{\mathrm{HY}}, \mathrm{Dec}_{\mathrm{HY}})$ is a combination of the Niederreiter KEM $\pi_{\mathrm{NR\_KEM}}$ with the DEM π_{DEM}.

- **$\mathbf{Gen_{HY}}$** invokes $\mathrm{Gen}_{\mathrm{NR\_KEM}}()$ and returns the generated key-pair.
- **$\mathbf{Enc_{HY}}$** receives plaintext m and public-key H' and first invokes $s' = \mathrm{Enc}_{\mathrm{NR\_KEM}}(H')$. The returned symmetric keys k_1 and k_2 are used to encrypt the message to $T = \mathrm{Enc}_{k_1}^{\mathrm{SE}}(m)$ and to compute the authentication tag $\tau = \mathrm{Ev}_{k_2}(T)$. The overall ciphertext is $(s' \,||\, T \,||\, \tau)$.
- **$\mathbf{Dec_{HY}}$** receives ciphertext $(s' \,||\, T \,||\, \tau)$ and invokes $\mathrm{Dec}_{\mathrm{NR\_KEM}}(s')$ to decrypt the symmetric key $k = (k_1 \,||\, k_2)$. Then it verifies the correctness of the MAC by evaluating if $\mathrm{Ev}_{k_2}(T)$ matches τ. If the MAC is correct, plaintext $m = \mathrm{Dec}_{k_1}^{\mathrm{SE}}(T)$ is decrypted and returned. In case of a MAC mismatch, $\perp$ is returned.

7.4.3 QC-MDPC Niederreiter Hybrid Encryption

Our instantiation of the Niederreiter hybrid encryption scheme of [Per13] realizes the KEM using QC-MDPC Niederreiter as defined in Section 7.2. We construct the DEM based on the AES symmetric encryption standard [NIS01] which enables the DEM to handle arbitrary

[3] In [Per13], the DEM is assumed as a fixed length one-time pad of the size of m combined with a standard MAC. Hence, $\mathrm{Enc}_{k_1}^{\mathrm{SE}}(m) = m \oplus k_1$ and $\mathrm{Dec}_{k_1}^{\mathrm{SE}}(T) = T \oplus k_1$ with m, T, k_1 having the same fixed length.

plaintext lengths compared to the impractical one-time pad DEM used in [Per13]. We target 80-bit and 128-bit security levels in this work. Our DEM uses AES-128 in CBC-mode for message en-/decryptions and AES-128 in CMAC-mode for MAC computations hereby following the *encrypt-then-MAC* paradigm. Furthermore, we employ SHA-256 for the key derivation of k_1 and k_2 from s'. For an overall 256-bit security level, appropriate parameters for QC-MDPC Niederreiter should be used (cf. [MTSB13], Section 3.5) combined with AES-256-CBC for encryption, AES-256-CMAC for MAC computations, and SHA-512 for key derivation.

Hybrid Key-Generation

Hybrid key-generation uses QC-MDPC Niederreiter key-generation (cf. Section 7.2).

Hybrid Encryption

Hybrid encryption generates a random error vector $e \in_R \mathbb{F}_2^n$ with Hamming weight t, encrypts e using QC-MDPC Niederreiter encryption to s' and derives two 128-bit symmetric sessions keys $k = (k_1 \,||\, k_2) = \text{SHA-256}(e)$. Message m is encrypted under k_1 by AES-128 in CBC-mode to T starting from a random initialization vector IV. A MAC tag τ is computed over T under k_2 using AES-128 CMAC. The ciphertext is $(s' \,||\, T \,||\, \tau \,||\, IV)$.

Hybrid Decryption

Hybrid decryption extracts the symmetric session keys k_1 and k_2 from the QC-MDPC Niederreiter cryptogram, verifies the provided AES-128 CMAC authentication tag under k_2 and finally decrypts the symmetric ciphertext using k_1 with AES-128 in CBC-mode. The scheme is illustrated in Figure 7.6.

Security

Proof for the IND-CCA security of the hybrid scheme is given in [Per13] assuming IND-CCA secure symmetric encryption. Furthermore, [CS03] showed that it is possible to construct IND-CCA symmetric encryption from IND-CPA symmetric encryption (AES-CBC with random IVs [BDJR97]) by combining it with a standard MAC (AES-CMAC).

7.5 QC-MDPC Niederreiter on ARM Cortex-M4

The following implementation of QC-MDPC Niederreiter targets ARM Cortex-M4 microcontrollers since they are a modern wide-spread representative of embedded computing platforms. Our implementation covers key-generation, encryption, and decryption. Details on the implementations of the hybrid encryption scheme based on QC-MDPC Niederreiter are presented in Section 7.6.

We use the same microcontroller that was used to implement QC-MDPC McEliece in Chapter 6 to allow fair comparison with previous work. The STM32F417VG microcontroller features

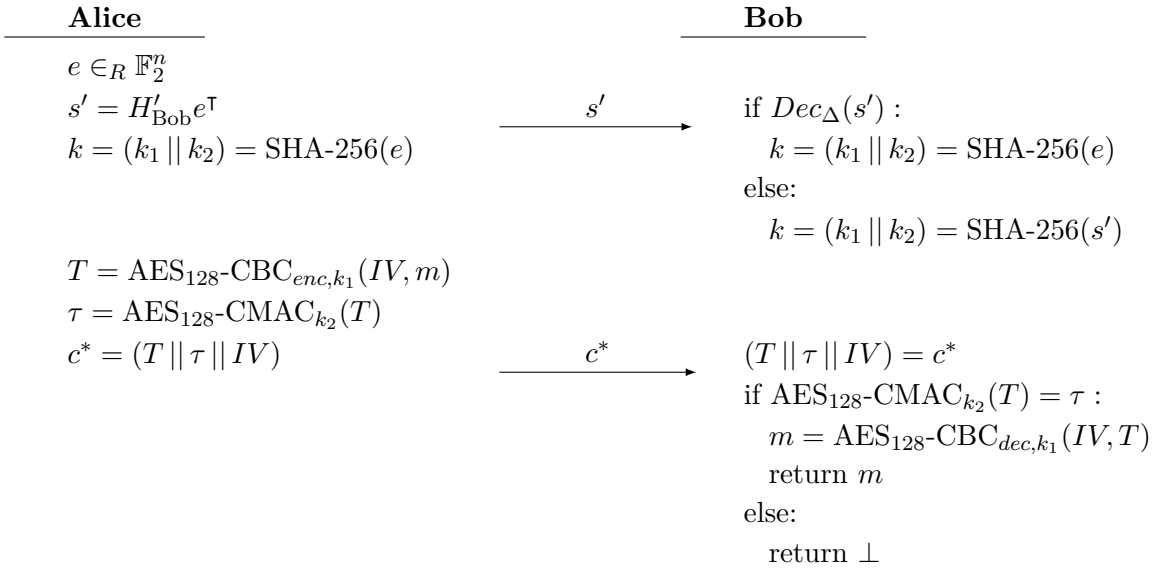


Figure 7.6: Alice encrypts plaintext m for Bob using QC-MDPC Niederreiter hybrid encryption with public-key H'_{Bob}. We split the transfer of s' and c^* for illustration purposes.

an ARM Cortex-M4 CPU with a maximum clock frequency of 168 MHz, 1 MB of flash memory and 192 kB of SRAM. The microcontroller is based on a 32-bit architecture and offers hardware co-processors for acceleration of AES, 3DES, MD5, SHA-1, and true random number generation. Our implementations are written in *Ansi-C* with partial use of Thumb-2 assembly for critical functions. The primary optimization goal is performance; the secondary goal is memory consumption, e.g., we make limited use of unrolling only when it has high performance impacts.

7.5.1 Polynomial Representations

Our implementations use three different polynomial representations. Each representation has advantages which we utilize in different parts of our implementations.

- *poly_t*: is the naïve way to store a polynomial. It simply stores each bit of the polynomial after each other. Its size depends on the polynomial's length and is independent of the weight of the polynomial.
- *sparse_t*: stores the positions of the polynomial's set bits. This representation requires less memory than *poly_t* if few bits are set. Furthermore, it allows fast iterations of set bits in the polynomial without having to test all its positions.
- *sparse_double_t*: stores the polynomial similarly to the *sparse_t* representation but allocates twice the size of the actually required memory. The yet unused memory is prepended. In addition, it holds a pointer indicating the start of the polynomial. This representation is beneficial when rotating sparse polynomials compared to the *sparse_t* representation. Its benefits will be explained in more detail when we explain efficient decoding in Section 7.5.4.

7.5.2 QC-MDPC Niederreiter Key-Generation

Generating a random first row candidate h_{n_0-1} for block H_{n_0-1} of length r and Hamming weight d_v is done using the microcontroller's TRNG as source of entropy. Its outputs are used as indexes at which we set bits in the polynomial. Since r is prime and hence not a power of two, we use rejection sampling to ensure a uniform distribution of the sampled indexes. The TRNG provides 32 random bits per call but only $\lceil \log_2(r) \rceil$ random bits (13 bits at an 80-bit security level, 14 bits at an 128-bit security level) are needed to determine an index in the range of $0 \leq i \leq r-1$. Hence we derive two random indexes per TRNG call. As stated in Section 7.2, we have to ensure that H_{n_0-1} is invertible. We therefore apply the *extended Euclidean algorithm* to generated first row candidates until an invertible h_{n_0-1} is found.

We generate the remaining first rows h_i similar to h_{n_0-1} but skip the inverse checking as only H_{n_0-1} has to be invertible. After private-key generation, we compute the corresponding public-key which is the systematic parity-check matrix $H' = H_{n_0-1}^{-1} \cdot H = [H_{n_0-1}^{-1} \cdot H_0 | \ldots | I]$. All we need to do is to compute $H_1^{-1} \cdot H_0$ and append the identity matrix since the selected parameter sets always have $n_0 = 2$. The private-key has few set bits ($d_v \ll r$), hence we store it in sparse representation. The public-key is stored in polynomial representation due to its high density. Since the code is quasi-cyclic, we only store the first columns of both matrices. The different representations ease and accelerate later usage of the polynomials.

7.5.3 QC-MDPC Niederreiter Encryption

Given a public-key H' and an error vector[4] $e \in \mathbb{F}_2^n$ of weight $\mathrm{wt}(e) = t$, we compute the public syndrome $s' = H'e^\intercal$. Computing s' is done by iterating over set bits in the error vector and accumulating the corresponding columns of H'. Since the error vector is stored in sparse representation, the index of each bit in the error vector specifies the number of cyclic shifts of the first column of H'. To avoid repeated shifting, we reuse the previous shifted column and shift it only by the difference to the next bit index. Multiplication of $e^\intercal$ by the identity part of H' is skipped. As the public syndrome has high density, we store it in *poly_t* representation.

7.5.4 QC-MDPC Niederreiter Decryption

For decryption we implement two decoder variants: Dec_1 and Dec_2. They differ in their implementation; the decoding behavior of both remains as explained in Section 7.2.1. We start with Dec_1 and subsequently explain the improvements made in Dec_2 to accelerate decryption. Furthermore, we discuss general implementation optimizations.

Dec_1 Decoder Dec_1 starts by computing the private syndrome $s = H_{n_0-1}s'^\intercal$ from the public syndrome s' and the private-key H. This is the same operation as encryption, however we use the *sparse_t* representation for the private-key. Recovery of the error vector e starts from a zero-initialized error candidate e_{cand} of length n. For each column of the private parity-check matrix

[4] We do not implement constant weight encoding since it is not needed in the hybrid encryption scheme. Encrypting a message $m \in \mathbb{Z}/\binom{n}{t}\mathbb{Z}$ requires to encode it into an error-vector $e \in \mathbb{F}_2^n$ of weight $\mathrm{wt}(e) = t$ and to reverse the encoding after decryption.

blocks we observe the number of positions which are different from the private syndrome s, i.e., counting unsatisfied parity-checks. We implement this step by computing the binary *AND* of the current column of the private parity-check matrix block with s followed by a Hamming weight computation of the result.

If the Hamming weight exceeds the decoding threshold $b_{\text{iteration}}$, we invert the corresponding bit in e_{cand}. The position is determined by the current column i and block j with $pos = j * r + i$. Additionally, we *XOR* the current column onto the private syndrome for a direct update every time a bit is flipped in e_{cand}. Updating the syndrome while decoding was shown to drastically increase decoding performance in Chapter 4 for QC-MDPC McEliece; the results similarly apply to QC-MDPC Niederreiter.

We iterate over the private-key column by column from the first block to the last by taking the first column of each block and performing successive cyclic shifts. The *sparse_t* representation allows efficient shifting as we only have to increment d_v indexes to effectively shift the polynomial. However, we have to check for overflows of incremented indexes which translate to carry transfers in the regular *poly_t* representation. An overflow results in additional effort, as we have to transfer every value in memory so that the position of the highest bit is always stored in the highest counter.

After iterating over all columns of the private-key, we compute the public syndrome of the current error candidate, i.e., we encrypt e_{cand} to $s'_{\text{cand}} = H' e^{\mathsf{T}}_{\text{cand}}$ and compare s'_{cand} to the initial public syndrome s'. On a match, the error vector was found and decryption finishes by returning e. On a mismatch, we continue with the next decoding iteration. After a fixed number of iterations[5], we abort and restart decoding with the original private syndrome and increased decoding thresholds similar to the optimized QC-MDPC McEliece decoder $\mathcal{D}_2$ (cf. Section 4.4.1).

Dec_2 The Dec_1 decoding approach has two downsides. First, the public-key has to be known during decryption which diverges from standard crypto APIs. Second, costly encryptions have to be performed after each decoding iteration to check whether the current error candidate is the correct error vector. Our Dec_2 decoder addresses these drawbacks as described in the following.

The first optimization is to transform the private-key from *sparse_t* to *sparse_double_t* polynomial representation. This structure allows efficient handling of overflows during column rotations. A cyclic shift without carry is equivalent to the *sparse_t* representation in which we increment every bit index of the polynomial. In case of a carry, we pop the last value of the array (with value r), move all array elements by one position, and insert a new value in the beginning (with value 0). We illustrate this operation in Figure 7.7.

Using *sparse_double_t* we avoid direct manipulation of the array in case of a carry which is the costly part of the *sparse_t* representation. Instead, we decrement the pointer by one and insert a zero at the first element. The last element is ignored since the polynomial has known fixed weight d_v and thereby known length. While the previous approach needs d_v operations, this approach breaks it down to two operations, independent of the polynomial's length. We illustrate the carry handling in *sparse_double_t* representation in Figure 7.8.

[5]We found the number of iterations experimentally and set it to five, cf. Section 4.5

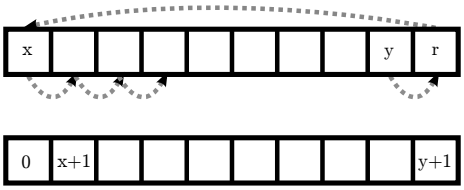


Figure 7.7: Carry handling during cyclic polynomial rotation in *sparse_t* representation.

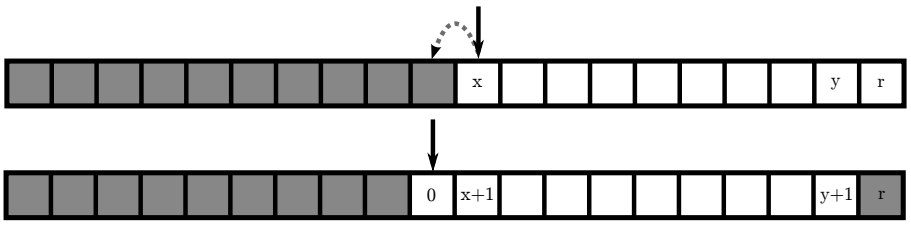


Figure 7.8: Carry handling during cyclic polynomial rotation in *sparse_double_t* representation. The pointer position is indicated by the black arrow.

Our second optimization checks if the Hamming weight of the error candidate matches the expected Hamming weight $\mathrm{wt}(e) = t$ instead of encrypting e_{cand} after every decoding iteration. If the Hamming weights do not match, we continue with the next decoding iteration immediately. Since Hamming weight computation of a vector is a much cheaper operation than vector matrix multiplication, decryption performance improves.

The third optimization completely eliminates the need to encrypt the error candidates to determine whether the correct error vector was found. Instead we test the private syndrome for zero at the end of each decoding iteration. Since the private syndrome is updated every time a bit-flip occurs, it becomes zero once the correct error vector was recovered.

Other general optimizations include writing hot code of the decryption routine in ARM Thumb-2 assembly giving us full control of the executed instructions and allowing us to pay close attention to the instruction execution order to avoid pipeline stalls by interleaving instructions which decreases the number of wasted clock cycles. Furthermore, we store two 16-bit indexes in one 32-bit field of the *sparse_double_t* type[6]. As we indicate the start by a pointer, we do not need to actually shift the values in memory in case of an overflow. A shift by 16 bits would be expensive on a 32-bit architecture. Furthermore, this allows us to increment two values with one `ADD` instruction, and we process twice the data with each load and store instruction. To benefit from the burst mode of the load and store instructions (`LDMIA` and `STMIA`), i.e., loading and storing multiple words from and to SRAM, we have to ensure that the memory pointers are 32-bit word aligned. This however is not the case at every second overflow since we decrement the *sparse_double_t* pointer in 16-bit steps. A flag variable is used to deal with this issue. If the flag is set, we temporarily decrease the pointer for alignment.

[6] 16 bits are sufficient to store the position for the 80-bit and 128-bit security levels.

7.6 Hybrid Encryption on ARM Cortex-M4

This section details our implementation of the IND-CCA secure QC-MDPC Niederreiter hybrid encryption scheme for ARM Cortex-M4 microcontrollers as introduced in Section 7.4.3. We describe hybrid key-generation, hybrid encryption, and hybrid decryption based on our implementation of QC-MDPC Niederreiter (cf. Section 7.5).

7.6.1 Hybrid Key-Generation

The hybrid encryption scheme requires an asymmetric key-pair for the KEM and two symmetric keys for the DEM. One symmetric key is used to ensure confidentiality through encryption; the other key is used to ensure message authentication. However, only the asymmetric key pair is permanent. The symmetric keys are randomly generated during encryption. Thus, the implementation of the hybrid key-generation is equal to QC-MDPC Niederreiter key-generation (cf. Section 7.5.2).

7.6.2 Hybrid Encryption

On input of a plaintext $m \in \mathbb{F}_2^*$ and a QC-MDPC Niederreiter public-key H', we generate a random error vector $e \in_R \mathbb{F}_2^n$ with $\mathrm{wt}(e) = t$ using the microcontroller's TRNG and encrypt e under H' using QC-MDPC Niederreiter encryption (cf. Section 7.5.3). Additionally, a SHA-256 hash is derived from e and is split into two 128-bit keys $k = (k_1 \,||\, k_2) = \text{SHA-256}(e)$.

After generation of k_1 and k_2 the key encapsulation is finished, and we continue with data encapsulation. We generate a random 16-byte IV using the microcontroller's TRNG and encrypt message m under k_1 to $T = \text{AES-128-CBC}_{enc,k_1}(IV, m)$. Ciphertext T is then fed into AES-128-CMAC, generating a 16-byte tag τ under key k_2. Finally, we concatenate the outputs to $x = (s' \,||\, T \,||\, \tau \,||\, IV)$.

To accelerate AES operations we make use of the AES crypto co-processor offered by the STM32F417 microcontroller for encryption and MAC generation. Unfortunately, the crypto co-processor only offers SHA-1 acceleration which we refrain from to not lower the overall security level. Thus we created a software implementation of SHA-256 for hashing.

7.6.3 Hybrid Decryption

Hybrid decryption receives ciphertext $x = (s' \,||\, T \,||\, \tau \,||\, IV)$ and decrypts the public syndrome s' using QC-MDPC Niederreiter decryption with the KEM private-key to recover the error vector e (cf. Section 7.5.4). After successful decryption of e, we derive sessions keys k_1 and k_2 by hashing the error vector with SHA-256. We compute the AES-128-CMAC tag τ^* of the symmetric ciphertext T under k_2. If $\tau^* \neq \tau$ we abort decryption; otherwise we decrypt T under k_1 using AES-128-CBC to recover plaintext m.

We make use of the microcontroller's AES crypto co-processor to accelerate decryption and MAC computation. For SHA-256 we use the same software implementation as during encryption.

7.7 Implementation Results

Below we present our implementation results of QC-MDPC Niederreiter and of the hybrid encryption scheme from [Per13] instantiated with QC-MDPC Niederreiter. Both implementations target ARM Cortex-M4 embedded microcontrollers. We list code size as well as execution time, evaluate the impact of our optimizations, and compare the results with previous work. Our code was built with GCC for embedded ARM (arm-eabi v.4.9.3) at optimization level `-O2`.

7.7.1 QC-MDPC Niederreiter Results

In order to measure the performance of QC-MDPC Niederreiter key-generation, encryption and decryption, we use randomly chosen instances throughout the measurements. We generate 500 random key-pairs and measure for each key-pair 500 en-/decryptions of randomly chosen plaintexts of n-bit length and Hamming weight t, resulting in 250,000 executions over which we average the execution time. Furthermore, we measure cyclic shifting in *poly_t* compared to the sparse polynomial representations to verify our optimizations in more detail. The execution times are listed for 80-bit security. The results for 128-bit security are given in parenthesis.

QC-MDPC Niederreiter key-generation takes 376.1 ms (1495.8 ms); encryption takes 15.6 ms (81.7 ms), and decryption takes 109.6 ms (477.7 ms) with decoder Dec_2 on average. With decoder Dec_1, decryption takes 697.9 ms (3830.2 ms) on average. Both decoders require 2.35 (3.25) decoding iterations on average until decoding succeeds. As embedded microcontrollers usually generate few key pairs in their lifespan; key-generation performance is of less practical relevance.

Generating the full private parity-check matrix from its first column in the straightforward *poly_t* representation takes 83.4 ms (345.8 ms). Our *sparse_t* representation accelerates this to 11.6 ms (34.0 ms), and the *sparse_double_t* representation allows even faster rotations with 7.9 ms (21.2 ms) for the same task. By storing private-keys in sparse representation with two 16-bit counters in one 32-bit word we reduce the required memory per private-key by 85% (88.5%) from 9602 bits (19714 bits) to 1440 bits (2272 bits) compared to storing the polynomials in their full length.

The code size of 80-bit QC-MDPC Niederreiter including key-generation, encryption and decryption with Dec_1 requires 14 KiB flash memory (1.3%) and additional 4 KiB SRAM (2.0%). For the 128-bit parameter set we need 19 KiB flash memory (1.9%) and 4 KiB SRAM (2.0%). The same implementation with decoder Dec_2 requires 16 KiB flash (1.6%) and 3 KiB SRAM (1.5%). For 128-bit security we measure 20 KiB flash memory (2.0%) and 3 KiB SRAM (1.5%) with Dec_2. Table 7.1 lists the code size of each function separately. Note that the sum of the separate code sizes is greater than the combined implementation due to code reuse.

7.7.2 QC-MDPC Niederreiter Hybrid Encryption Results

The execution time of hybrid encryption schemes is dominated by the public-key cryptosystem which is used for key en-/decapsulation. Hence, we employ QC-MDPC decoder Dec_2 for key decapsulation as it operates much faster than Dec_1. We generate 500 random key pairs and en-/decrypt 500 randomly chosen 32-byte plaintexts for each key pair with the hybrid encryption

scheme. We measure short plaintexts for worst-case cycles/byte performance. Longer plaintexts marginally affect performance since they are only processed by symmetric components. Below we list our results for 80-bit security and 128-bit security (in parenthesis).

Key-generation of the hybrid encryption scheme requires 386.4 ms (1511.8 ms); hybrid encryption takes 16.5 ms (83.2 ms), and hybrid decryption takes 111.0 ms (477.5 ms) on average. Compared to pure QC-MDPC Niederreiter, the symmetric operations (en-/decryption, MACing, hashing) add very little to the overall execution time ($< 5\%$) although the hybrid encryption scheme seems more complex at first. The AES computations are hardware accelerated which results in a further speedup but even if a Cortex-M4 microcontroller without an AES co-processor would be used we would see only a slight increase in the overall execution time. The required code size of the complete hybrid encryption scheme (QC-MDPC Niederreiter, AES-128-CBC, AES-128-CMAC, SHA-256) is 25 KiB flash (2.4%) and 4 KiB SRAM (2.0%) at 80-bit security and 30 KiB flash (2.8%) and 4 KiB SRAM (2.0%) at 128-bit security.

7.7.3 Comparison with Related Work

Implementation results reported in related work are listed in Table 7.1. A direct comparison of QC-MDPC McEliece (cf. Chapter 6, [vMG14b]) with our hybrid QC-MDPC Niederreiter implemented on a similar ARM Cortex-M4 microcontroller shows that hybrid QC-MDPC Niederreiter is around 2.5 times faster at the same security level. In addition it provides IND-CCA security and the possibility to efficiently handle large plaintexts. However, the QC-MDPC McEliece implementation features constant runtime which adds to its execution time. Compared to QC-MDPC McEliece implemented on an ATxmega256, our encryption runs 50 times faster, and decryption runs 25 times faster. In addition we provide IND-CCA security through hybrid encryption. Comparing implementations on ATxmega256 with implementations on STM32F417 is not a fair comparison, however both microcontrollers come at a similar price which makes the comparisons relevant for practical applications. Publication of our work [vMHG16] led to a follow-up work by Chou [Cho16] who uses a bit-sliced implementation to provide constant-time key generation, encryption, and decryption for 80-bit security parameters in a similar hybrid encryption scenario. The bit-sliced implementation achieves 15% higher encryption performance and 20% higher decryption performance at the cost of a code-size of 62 Kbytes compared to our 16 Kbytes on a similar Cortex-M4 microcontroller. The key generation is 2.3 times slower but offers constant-time computations.

We refrain from comparing our work to the CS-MDPC Niederreiter implementation on a PIC24FJ32GA002 microcontroller as presented in [BBMR14]. It was shown in [Per14] that the proposed CS-MDPC parameters do not reach the proclaimed security levels and need adaptation. McEliece implementations based on binary Goppa codes targeting the ATxmega256 microcontroller were presented in [EGHP09] and [Hey11]. Again, our implementations outperform both by factors of 5-28. In addition, binary Goppa code public-keys are much larger (64 Kbytes vs. 4801 bits) and impractical for devices with constraint memory. The CCA2-secure McEliece implementation based on Srivastava codes presented in [CHP12] also targets the ATxmega256 and is just 4-8 times slower than our hybrid QC-MDPC Niederreiter which appears as good competitor if implemented on the same microcontroller.

Table 7.1: Performance and code size of our implementations of QC-MDPC Niederreiter using Dec_2 compared to other implementations of similar public-key encryption schemes on embedded microcontrollers. We abbreviate Niederreiter (NR) and McEliece (McE). [1]Flash and SRAM memory requirements are reported for a combined implementation of key generation, encryption, and decryption. [2]Flash requirements are reported for a combined implementation of key generation, encryption, and decryption, SRAM memory requirements are not available. Without symmetric primitives the implementation is reported at 38 Kbytes of flash.

Scheme	**Platform**	**SRAM [bytes]**	**Flash [bytes]**	**Cycles/Op**	**Time/Op [ms]**
QC-MDPC $NR_{80\text{-bit,enc}}$	STM32F417	2,048	3,064	2,623,432	16
QC-MDPC $NR_{80\text{-bit,dec}}$	STM32F417	2,048	8,621	18,416,012	110
QC-MDPC $NR_{80\text{-bit,keygen}}$	STM32F417	3,136	8,784	63,185,108	376
QC-MDPC $NR_{80\text{-bit,combined}}$	STM32F417	3,136	16,124	-	-
QC-MDPC $NR_{128\text{-bit,enc}}$	STM32F417	2,048	4,272	13,725,688	82
QC-MDPC $NR_{128\text{-bit,dec}}$	STM32F417	2,048	8,962	80,260,696	478
QC-MDPC $NR_{128\text{-bit,keygen}}$	STM32F417	3,136	12,096	251,288,544	1496
QC-MDPC $NR_{128\text{-bit,combined}}$	STM32F417	3,136	20,416	-	-
QC-MDPC $McE_{80\text{-bit,enc}}$	STM32F407	2,700[1]	5,700[1]	7,018,493	42
QC-MDPC $McE_{80\text{-bit,dec}}$	STM32F407	2,700[1]	5,700[1]	42,129,589	251
QC-MDPC $McE_{80\text{-bit,keygen}}$	STM32F407	2,700[1]	5,700[1]	148,576,008	884
QC-MDPC $NR_{80\text{-bit,enc}}$ [Cho16]	STM32F407	-	62,000[2]	2,244,489	13
QC-MDPC $NR_{80\text{-bit,dec}}$ [Cho16]	STM32F407	-	62,000[2]	14,679,937	87
QC-MDPC $NR_{80\text{-bit,keygen}}$ [Cho16]	STM32F407	-	62,000[2]	140,372,822	836
QC-MDPC $McE_{80\text{-bit,enc}}$ [HvMG13]	ATxmega256	606	5,500	26,767,463	836
QC-MDPC $McE_{80\text{-bit,dec}}$ [HvMG13]	ATxmega256	198	2,200	86,874,388	2,710
Goppa McE_{enc} [EGHP09]	ATxmega256	512	438,000	14,406,080	450
Goppa McE_{dec} [EGHP09]	ATxmega256	12,000	130,400	19,751,094	617
Goppa McE_{enc} [Hey11]	ATxmega256	3,500	11,000	6,358,400	199
Goppa McE_{dec} [Hey11]	ATxmega256	8,600	156,000	33,536,000	1,100
Srivastava McE_{enc} [CHP12]	ATxmega256	-	-	4,171,734	130
Srivastava McE_{dec} [CHP12]	ATxmega256	-	-	14,497,587	453

7.8 Conclusion

This work presented the first implementations of QC-MDPC Niederreiter and of Persichetti's hybrid encryption scheme for ARM Cortex-M4 microcontrollers. We extended Persichetti's hybrid encryption scheme by choosing well-known symmetric components for data encapsulation in order to handle plaintexts of arbitrary length. We achieved reasonable performance using a combination of new implementation optimizations and transferred known techniques from QC-MDPC McEliece. Furthermore, our implementations operate with practical key sizes which addresses a long-standing drawback of code-based cryptography. IND-CCA security and performance through hybrid encryption are important features for real-world applications. Resistance against quantum computing attacks is an additional provided feature that will be required for next-gen cryptographic applications. This work provides a possible solution which is feasible even on constraint embedded microcontrollers.

Chapter 8

Embedded Syndrome-Based Hashing

This chapter presents first implementations of the syndrome-based hash function RFSB-509 on an Atmel ATxmega128A1 microcontroller and a low-cost Xilinx Spartan-6 FPGA. We explore several trade-offs between size and speed on both platforms and show that RFSB is extremely versatile with applications ranging from lightweight to high performance. The lightweight microcontroller implementation requires just 732 bytes of ROM while still achieving a competitive performance compared to established hash functions. Our fastest FPGA implementation is based on embedded block memories available in Xilinx Spartan-6 devices. It runs at 0.21 cycles/byte and with a throughput of 5.35 Gbit/s. To the best of our knowledge, this is the first time the RFSB hash function is implemented on either of these wide-spread platforms.

This research was presented at Indocrypt'12 [vMG12] and is a joint work with Tim Güneysu.

Contents

8.1 Introduction

Cryptographic hash functions are used in a wide range of applications where a secure mapping of an arbitrary amount of data to a fixed-length bit string is required. Examples are digital signatures, messages authentication codes, data integrity checks, and password protection. Prominent and widely deployed hash functions such as MD5 [Riv92], SHA-1 [NIS12], the SHA-2 family [NIS12] and SHA-3 [NIS14] are used in various products and implementations whose security depends on the collision and pre-image resistance of those hash functions. However, (chosen-prefix) collision attacks have been published for MD5 [SSA$^+$09, XLF13] and SHA-1 [WYY05] over the last years and are already exploited in the real-world. A major attack based on MD5 collisions was performed by the Flame espionage malware which injects itself into the Microsoft Windows operating system. The Flame malware code is signed by a rogue Microsoft certificate and disguises itself as a Microsoft Windows update. The rogue certificate was obtained using a previously unknown chosen-prefix collision attack on a Microsoft Terminal Server Licensing Service certificate which still used the outdated MD5 hashing algorithm [Jon12].

Although the SHA-2 family withstands critical attacks so far, its similar structure to SHA-1 and ever improving attacks on round-reduced version of SHA-256/-512 (e.g., [MNS13, EMS14]) raise concerns about its long-term security. Therefore, the National Institute of Standards and Technology (NIST) announced the public SHA-3 competition in the end of 2007 [NIS07]. A total of 64 candidates entered the competition out of which 14 advanced to the second round; five candidates entered the final round, and Keccak [BDPv11] was selected as the SHA-3 standard [UN12, NIS14]. Apart from security, the main selection criteria were hardware and software speeds as well as scalability. The announcement of the SHA-3 competition explicitly demands efficiency in 8-bit microcontrollers as well as in FPGAs and in hardware to account for the wide range of application in which hash functions are used.

Embedded 8-bit microcontrollers are a common representative of low-cost and energy efficient computation units used in many real-world applications, e.g., in the automotive industry, digital signature smart cards, and wireless sensor networks. Field-Programmable Gate Arrays (FPGA) on the other hand allow reconfigurable implementations in hardware, usually yielding a much better performance than achievable with 8-bit microcontrollers or PCs. FPGA device classes range from low-cost (e.g., Xilinx Spartan family) to high-end/high-speed (e.g., Xilinx Virtex family). Since microcontrollers and FPGAs are both used for applications handling sensitive data, secure and efficient cryptographic primitives are essential on both platforms.

Code-based cryptography offers a variety of cryptographic primitives that are built upon the hardness of well-known NP-complete problems in coding theory. Besides public-key encryption and digital signatures, coding theory can also be applied to realize cryptographic hash functions. The Fast Syndrome-Based (FSB) hash function [AFG$^+$08] is such a code-based hash function. FSB was one of the candidates in the SHA-3 competition but due to its inefficiency compared to other candidates, FSB did not advance to the second round. The Really Fast Syndrome-Based (RFSB) hash function [BLPS11] is an improved version of FSB that aims to be more efficient and thus overcomes the main drawback of FSB.

Contribution With code-based public-key encryption and digital signature schemes proven to be feasible in hard- and software, it still is an open question how code-based hash functions perform on these platforms. We set out to answer this question by evaluating the feasibility and achievable performance of RFSB-509 in embedded systems. We explore different design choices for embedded microcontrollers and reconfigurable hardware by targeting the wide-spread 8-bit Atmel ATxmega microcontroller and Xilinx Spartan-6 FPGAs. We show that RFSB-509 can be efficiently implemented on both platforms and that RFSB can, in contrast to its predecessor FSB, keep up with the SHA-3 finalists and other hash standards. Source code for both platforms is made publicly available to allow independent verification of our implementations and to inspire future work[1].

Outline This chapter is organized as follows. We present related work on code-based hash functions and the history that led to the development of RFSB in Section 8.2. After briefly introducing general specifications of the RFSB hash function, we detail on the concrete proposal RFSB-509 and give an implementer's view on RFSB-509 in Section 8.3. Next, our design considerations for implementations targeting embedded microcontrollers and reconfigurable hardware are presented in Section 8.4 and Section 8.5. We evaluate our results in Section 8.6 and draw a conclusion in Section 8.7.

8.2 Related Work

Augot, Finiasz, Gaborit, Manuel, and Sendrier entered the SHA-3 competition with the Fast Syndrome Based (FSB) hash function [AFG+08] that relies on the syndrome decoding problem for linear codes. Previous attempts to build such a hash function by Augot, Finiasz, and Sendrier [AFS03, AFS05], and Finiasz, Gaborit, and Sendrier [FGS07] turned out to be flawed and were broken by Coron and Joux [CJ04], Saarinen [Saa07], and Fouque and Leurent [FL08]. Hence, FSB was adjusted to withstand these attacks for the SHA-3 submission and to date the updated version remains unbroken. However, FSB did not advance to the second round of the SHA-3 competition mainly because it lacks in efficiency compared to other candidates.

Meziani, Dagdelen, Cayrel, and El Yousfi Alaoui used the ideas of FSB and combined them with a sponge construction instead of the Merkle-Damgård principle [Dam90] to construct the S-FSB hash function [MDCE11]. Their main goal was to improve the performance compared to FSB, and they reported a C implementation of S-FSB-256 on an Intel Core 2 Duo CPU running at 183 cycles/byte. Compared to FSB requiring 264 cycles/byte on the same CPU, S-FSB is about 30% faster, but when looking at the overall picture, S-FSB is still an order of magnitude slower than the current hash standard SHA-256 which runs at 15.49 cycles/byte on a similar CPU according to eBASH[2] [eBA15b]. Optimized implementations that make use of SSE CPU extensions have been reported for FSB and S-FSB in [CMNS14]. Although the authors were able achieve better cycle/byte ratios for both hash functions, 204 cycles/byte for FSB-256 and 172 cycles/byte for S-FSB-256 still remains an order of magnitude slower than SHA-256.

[1] `http://www.sha.rub.de/research/projects/code/`

[2] (6fd); 2007 Intel Core 2 Duo E4600; 2 x 2400MHz; cobra, supercop-20111120

Bernstein, Lange, Peters, and Schwabe introduced the Really Fast Syndrome-Based (RFSB) hash function as an improved version of FSB and proposed concrete parameters (RFSB-509) in [BLPS11]. The authors report an implementation of RFSB-509 that outperforms the current hash standard SHA-256 on Intel Core 2 Quad Q9550 CPUs at 13.62 vs. 15.26 cycles/byte. According to measurements on eBASH[3], an updated implementation by the same authors computes RFSB-509 even faster at 10.64 cycles/byte while SHA-256 remains at 15.31 cycles/byte on the same CPU.

Another software implementation of RFSB in Java and C is reported by Rothamel and Weiel [RW11] for x86 CPUs. In addition to RFSB-509, the authors suggest parameter sets RFSB-227, RFSB-379, and RFSB-1019 and provide performance measurements for all four variants. They report RFSB-509 to run at 120.5 cycles/byte on an Intel i7 CPU. A vectorized implementation of RFSB-509 is reported in [CMNS14] to be running at 19.27 cycles/byte on an AMD Phenom II X2 550 CPU. However, both results do not reach the performance reported in the original RFSB publication. The implementation by Schwabe and Bernstein runs at 9.06 cycles/byte on an Intel Core i7-4770[4].

8.3 The RFSB Hash Function

The RFSB hash function is constructed similarly to the FSB hash function [AFG+08]. Both are designed to be used inside a collision resistant hash function. A fixed length compression function is combined with the Merkle-Damgård domain extender [Dam90] to enable processing data of arbitrary length. An initialization vector (IV) is compressed together with the first message block. The output is used as a chaining value together with the second message block, and is again fed into the compression function. This continues until the second to last message block has been processed. The last block is padded by appending a single 1 bit followed by sufficiently many zeros and a 64-bit message length counter. After all input blocks have been processed, a final output filter (called final compression function in FSB terms) is applied. In case of FSB, Whirlpool is used as final compression function. The authors of RFSB suggest to use SHA-256 or an AES-based output filter. The basic hashing principle of FSB and RFSB is illustrated in Figure 8.1.

8.3.1 The RFSB Compression Function

The RFSB compression function is defined by four parameters: an odd prime r, positive integers b and w, and a compressed matrix of size $2^b \times r$ bits. The compression function takes a bw-bit string as input which is interpreted as a sequence of $\lceil bw/8 \rceil$ bytes $(m_1, m_2, \ldots, m_w)$, where each $m_i \in \{0, 1, \ldots, 2^b - 1\}$. The output is a r-bit string that is interpreted as a sequence of $\lceil r/8 \rceil$ bytes. Both input and output are interpreted in the little-endian format. The compressed matrix consists of constants $c[0], c[1], \ldots, c[2^b - 1]$, where each of the constant has a length of r-bit.

[3](10677); 2008 Intel Core 2 Quad Q9550; 4 x 2833MHz; berlekamp, supercop-20120704
[4](306c3); 2013 Intel Core i7-4770; 4 x 3400MHz; wintermute, supercop-20140505

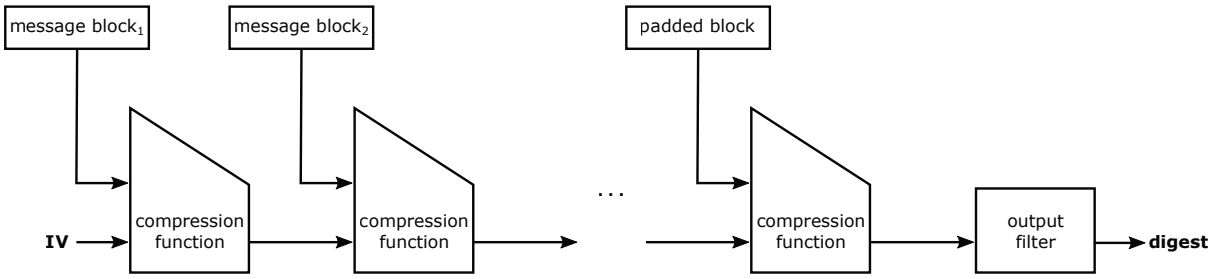


Figure 8.1: Illustration of the basic hashing principle based on the Merkle-Damgård domain extender used by FSB and RFSB. The initialization vector (IV) is set to zero in RFSB.

The uncompressed RFSB matrix is derived from these constants by defining

$$c_i[j] = c[j]\, x^{128(w-i)}, 1 \leq i \leq w,\ 0 \leq j \leq 2^b - 1$$

in the ring $\mathbb{F}_2[x]/(x^r - 1)$ which essentially are rotations of the compressed matrix constants.

The input is mapped to the output using the message bytes m_i as indices of the uncompressed matrix constants c_i. The constants are summed up by exclusive-or (XOR) addition to form the output as follows:

$$(m_1, m_2, \ldots, m_w) \mapsto c_1[m_1] \oplus c_2[m_2] \oplus \cdots \oplus c_w[m_w].$$

When using the compressed matrix notation, the mapping from input to output is given by:

$$(m_1, m_2, \ldots, m_w) \mapsto c[m_1]\, x^{128(w-1)} \oplus c[m_2]\, x^{128(w-2)} \oplus \cdots \oplus c[m_w]$$

in $\mathbb{F}_2[x]/(x^r - 1)$.

8.3.2 A Concrete Proposal: RFSB-509

RFSB-509 is a concrete parameter proposal by the designers of RFSB and has been shown to allow for fast software implementations. The authors of RFSB-509 claim to provide a collision resistance of more than 2^{128} and the proposed parameters are $r = 509, b = 8$, and $w = 112$. Hence, the RFSB-509 message block size is 896 bits (112 bytes) and the output size is 509 bits. The compressed matrix is of size $2^b \times r = 256 \times 509$ bits which roughly amounts to 16 Kbytes. A recent result by Kirchner [Kir11] claims to lower the complexity to about 2^{79} using an improved generalized birthday attack. Thus, the parameters need to be adjusted if a collision resistance of more than 79-bit is required.

Each element of the compressed matrix is generated using a concatenation of the ciphertexts that result from four AES-128 [NIS01] encryptions using the fixed all-zero key and a plaintext which is a function of the index of the constant. We denote AES encryption by $y = \mathrm{AES}_k(x)$, where y is a 16-byte ciphertext, k is a 16-byte key, and x is a 16-byte plaintext. The plaintext is set to zero except for the last two bytes. The second to last byte is set to $0 \leq j \leq 255$ which

is the index of the constant. The last byte of the plaintext is a counter which is increased with each AES-128 encryption from 0 to 3. In total this results in the 512-bit string

$$c'[j] = \mathrm{AES}_0(0,\ldots,0,j,0)\,||\mathrm{AES}_0(0,\ldots,0,j,1)\,||\ldots||\mathrm{AES}_0(0,\ldots,0,j,3)$$

which is reduced to

$$c[j] = c'[j] \mod (x^{509}-1)$$

to remain in the ring $\mathbb{F}_2[x]/(x^{509}-1)$. The 112-byte input block $(m_1, m_2, \ldots, m_{112})$ with each $m_i \in \{0, 1, \ldots, 255\}$ is mapped to the 509-bit output by computing

$$(m_1, m_2, \ldots, m_{112}) \mapsto c[m_1]\,x^{128(112-1)} \oplus c[m_2]\,x^{128(112-2)} \oplus \cdots \oplus c[m_{111}]\,x^{128} \oplus c[m_{112}]$$

in $\mathbb{F}_2[x]/(x^{509}-1)$.

8.3.3 RFSB-509 from an Implementer's Point of View

A few aspects have to be considered when designing RFSB-509 for embedded systems. Our detailed considerations and optimizations of RFSB-509 for embedded devices follow below.

The constants' matrix, albeit compressed, still has a size of 16 Kbytes. As memory usually is a scarce resource in embedded systems, it might be challenging to store this matrix. Due to the computability of the constants, one of two choices can be made. Either memory is spent to store the matrix or each constant is, probably multiple times, generated on-the-fly when needed. On-the-fly generation of each constant requires four AES-128 encryptions, thus a total of $4 \times 112 = 448$ AES encryptions are required for one RFSB-509 compression.

When compressing a message block, the rotations applied to each constant depend on the position of the current message byte. For example, the first mapping in RFSB-509 is $c[m_1]\,x^{128(112-1)} = c[m_1]\,x^{14208}$ which requires to rotate $c[m_1]$ by 14208 bit positions. When using 512-bit wide registers, the amount of different rotations performed during RFSB compression can be reduced to just four since $128(112-i) \equiv 384i \mod 512 \in \{384, 256, 128, 0\}$. Hence, the RFSB compression s_1 of the first four messages bytes (m_1, m_2, m_3, m_4) can be rewritten as

$$s_1 = \mathrm{ROL}_{384}(c[m_1]) \oplus \mathrm{ROL}_{256}(c[m_2]) \oplus \mathrm{ROL}_{128}(c[m_3]) \oplus c[m_4]$$

where ROL_j denotes a j-bit rotation to the left (towards the most significant bit) of a 512-bit register. The four different rotations and their exclusive-or sum can be seen as the *basic compression unit* of RFSB-509 which we generalized to

$$s_i = \mathrm{ROL}_{384}(c[m_{4i+1}]) \oplus \mathrm{ROL}_{256}(c[m_{4i+2}]) \oplus \mathrm{ROL}_{128}(c[m_{4i+3}]) \oplus c[m_{4i+4}].$$

In order to process all 112 input message bytes, the basic compression unit is repeated 28 times. Accumulation of the intermediate results s_i then yields the output of the compression function

$$\begin{aligned}\mathrm{compress}_{509}(m_1,\ldots,m_{112}) &= \sum_{i=1}^{28} s_i \mod (x^{509}-1)\\ &= \sum_{i=0}^{27}\sum_{j=1}^{4} \mathrm{ROL}_{512-128j}(c[m_{4i+j}]) \mod (x^{509}-1)\end{aligned}$$

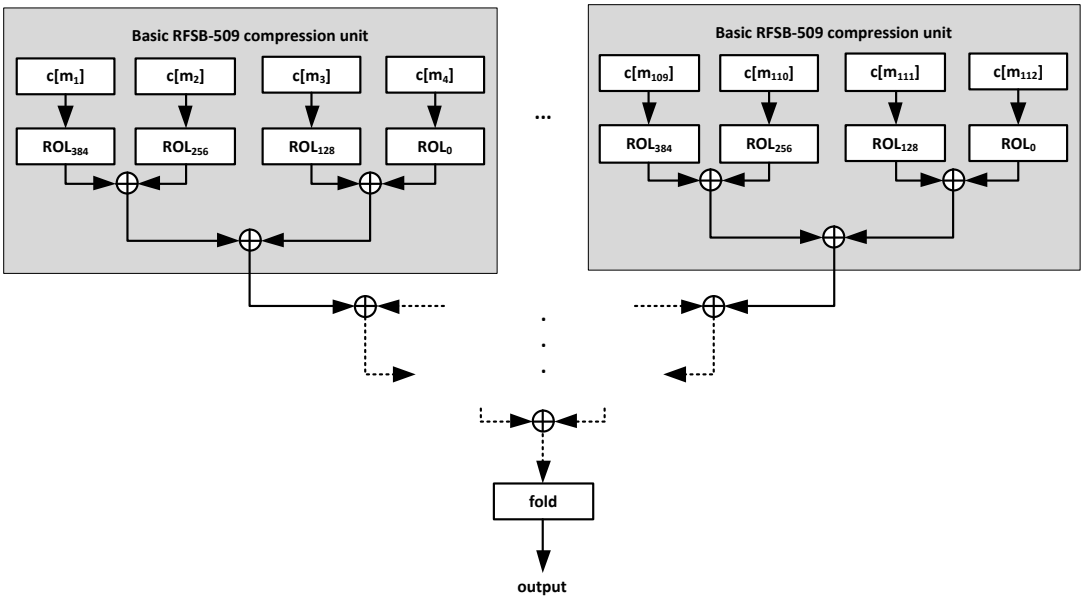


Figure 8.2: The basic compression unit of RFSB-509 consists of looking up four constants, rotating them according to their position by either 384, 256, 128, or 0 bits and xoring the results. The *fold* unit represents the reduction modulo $x^{509} - 1$.

in which the sums are formed using exclusive-or addition. Figure 8.2 illustrates the tree-like structure of the RFSB-509 compression function and shows multiple basic compression units.

One further important detail is the computation of the reduction modulo $x^{509} - 1$ for 512-bit registers. It is achieved by folding the three most significant bits onto the three least significant bits and setting the three most significant bits to zero. Such a reduction does not pose a problem on both platforms and can be realized at minimal cost.

8.4 Designing RFSB-509 for Embedded Microcontrollers

Our design of RFSB-509 for embedded microcontrollers targets the wide-spread 8-bit Atmel ATxmega microcontroller family which are low-cost yet powerful enough for a wide range of applications. Apart from the usual features offered by such devices (analog to digital conversion, timers, counters, several communication interfaces, etc.) the ATxmega offers dedicated hardware accelerators for the encryption standards DES [NIS99] and AES-128 [NIS01].

All following designs are split into three basic functions: *init*, *update*, and *final*. During *init* we reset the internal state, the output and the counter to zero. The *update* function implements the Merkle-Damgård domain extender, processes new message blocks and updates the internal state accordingly until the last message block is reached. The last message block is processed by the finalization function which pads the message, appends the length counter, compresses the last block and returns the overall output.

As detailed in Section 8.3.3, there are two ways of realizing the RFSB compression function. Either the constants are stored in a table or the constants are generated on-the-fly when

needed. One could also think of a hybrid mode, in which the constants are not stored in the program memory (ROM) but are generated once and stored in volatile SRAM when booting the device. We explore these three possibilities and give details about the design approach for each version in the following. The AES- and ROM-based implementations target the Atmel ATxmega128A1 microcontroller, the SRAM-based implementation requires to use Atmel's ATxmega384C3 microcontroller as it provides more SRAM.

8.4.1 On-the-Fly Constant Generation

On-the-fly constant generation is required for a lightweight implementation of RFSB-509 since the compressed constant matrix already consumes 16 Kbytes of program memory which renders a lightweight implementation impossible. Especially the hardware accelerated AES-128 offered in ATxmega devices is useful in this setting. The AES-128 crypto module runs concurrently to the CPU and takes 375 clock cycles after loading the key and the plaintext block into the module to en-/decrypt a 128-bit block. When taking loading and storing of key, plaintext and ciphertext into account, an AES-128 encryption takes about 500 clock cycles or 31.25 cycles/byte. Thus, when running at its maximum clock frequency of 32 MHz the ATxmega is able to achieve a plain AES-128 encryption throughput of around 8 Mbit/s.

Our lightweight implementation of the RFSB-509 compression function is built around a parameterizable constant generation function that is capable of providing rotation widths of $\{384, 256, 128, 0\}$ bits. During each iteration of the constant generation function, four AES encryptions are computed. After each AES encryption the ciphertext is transferred to 16 general purpose registers and immediately afterwards the next plaintext and key (which is the all-zero key for all encryptions but has to be reloaded before every encryption nevertheless) are loaded into the AES module and the next encryption is started. While waiting for the encryption to finish, we concurrently process the previous ciphertext by accumulating it to the output and reducing the computed constant modulo $x^{509} - 1$. Thus, we make use of otherwise wasted cycles while the AES encryption is running in parallel. In order to maintain a reasonable performance, parts of the code are unrolled, e.g., storing and loading data to and from the AES crypto module are unrolled since these parts are critical for the overall runtime.

Optimization Proposal

If the constants would be generated using DES instead of AES-128, the performance of the on-the-fly constant generation could be further improved. Since the output length of DES is half the output length of AES-128, twice as many DES encryptions would be required. However, at 16 cycles per DES encryption after loading the key and plaintext to the corresponding registers, this would still be an order of magnitude faster than AES-128 encryption on an ATxmega microcontroller. Since the performance of the encryption function is the limiting factor in such an implementation, the overall performance would greatly benefit from this modification.

Note, the short 56-bit keys of DES and its vulnerability to brute-force attacks do not pose a threat to the security of RFSB-509 since all plaintexts and keys are public by definition in RFSB. As stated in the original RFSB publication: "The full security of AES is certainly not

required for RFSB: all that we need is a function generating a few elements of $\mathbb{F}_2[x]/(x^r - 1)$ without any obvious linear structure" [BLPS11].

8.4.2 ROM-Based Lookup Table

A total of 16 Kbytes of program memory is required when storing the precomputed constants in the ROM of the microcontroller. Each of the 256 entries in the table consists of 64 bytes, thus we multiply each message byte by 2^6 to compute the index of the required constant. Instead of first reading out the constant and then rotating it according to the position of the current message byte, we adjust the table pointer beforehand to directly read out the rotated constant. This is possible since all rotation widths are a multiple of 8 and the basic addressable unit in our 8-bit microcontroller is a byte. For example, if a constant has to be rotated by 384 bit, we add $384/8 = 48$ to the current index, read out 16 bytes, then subtract 64 from the index and read out the remaining 48 bytes of the constant. Thus, we get nearly free rotations by pointer arithmetic. We repeat this process for all message bytes and rotation widths. The result is reduced modulo $x^{509} - 1$ after all constants have been read out and accumulated.

We explore two different approaches, a rolled and an unrolled version. The unrolled version removes all loops inside the basic compression unit which computes the intermediate output of four consecutive message bytes with four different rotation widths (cf. Figure 8.2).

8.4.3 RAM-Based Lookup Table

In order to estimate the maximum achievable performance, we move the constants from the program memory to the faster SRAM. Accessing a byte in the program memory of the ATxmega takes 3 clock cycles while accessing the internal SRAM takes 2 clock cycles. Since $112 \times 64 = 7168$ bytes have to be looked up when compressing one message block, this small difference can have a larger impact on the overall runtime as one might expect on first sight. The compression itself is constructed similarly to the previously described setup with some minor adjustments which account for the modified memory locations. For this evaluation we use the Atmel ATxmega384C3 microcontroller since it offers 32 Kbytes SRAM. Devices offering 8 or 16 Kbytes SRAM do not suffice in this scenario since the current state and the next message block have to be held in the SRAM in addition to the constants.

The remaining question is how to place the RFSB-509 constants into the SRAM. Since SRAM is volatile memory, its content has to be reloaded after every power cycle. As designers we are left with two choices. Either we store the constants in the program memory as done in Section 8.4.2 and copy them into SRAM at every power up, or we generate the constants once at every power up and store them in the SRAM. The decision which of the proposed methods to choose depends on two factors. On the one hand, it has to be considered how much time is available after a power cycle before the compression function has to be used for the first time. Generating the constants takes longer than just copying them from program memory. On the other hand, the decision depends on the available program memory. The generation function takes up much less program memory compared to a 16 Kbytes table. In our implementation, we generate the constants after each power up and thus avoid redundant tables in SRAM and ROM. Again we explore two approaches: a rolled and an unrolled version similarly to the ROM-table design.

8.5 Designing RFSB-509 for Reconfigurable Hardware

Our evaluation of RFSB-509 in reconfigurable hardware targets the low-cost Xilinx Spartan-6 device family. Spartan-6 devices offer hundreds to (ten-)thousands of slices, where each slice contains four 6-input/1-output look-up tables (LUT), eight flip-flops (FF), and surrounding logic. In addition to the general purpose logic, embedded resources such as block memories (BRAM) and digital signal processors (DSP) are available.

Our designs of RFSB-509 for reconfigurable hardware follow two different strategies to implement the compressed matrix constants. In one architecture we generate the constants when needed using on-the-fly AES computations, and in the other architecture we make use of the embedded block memories to store precomputed matrix constants.

Since different choices for the constant look-ups only affect the compression function of RFSB-509, all implementations share the same top-level component that takes care of handling the input and output through FIFOs as well as controlling the Merkle-Damgård construction which is also shared across our FPGA designs. Hence, our design is a modular system in which the compression function can be easily exchanged. We detail on the different compression function implementations below.

8.5.1 Implementing RFSB-509 with Embedded Block Memories

Spartan-6 FPGAs feature dual-ported block memories (BRAM) each capable of storing up to 18 Kbits. The BRAMs can be configured to represent one out of five different memory types. For our purpose we choose to configure the BRAMs as dual-port read-only memories since we do not need to write new constants. Dual-ported BRAMs allow to read two separate values from two different memory addresses in each clock cycle.

Minimal BRAM Consumption

Since the compressed constants' matrix has a size of about 15.9 Kbytes, a minimum of $\frac{15.9 \cdot 8}{18} = 7.07$ BRAMs is required. However, a wide-access port of 509 bits for each constant is not supported by the BRAM primitives. The maximum native supported port width is 32-bit (36-bit when using the parity bits) or when combining both ports 64-bit (72-bit when using the parity bits). To achieve a minimal block memory usage, we use eight BRAMs to store the constants as shown in Figure 8.3.

We configure the BRAMs to store 512 values of 32 bits each. The RFSB constants are divided into eight 64-bit chunks and are distributed to the BRAMs. The 64-bit chunks are again split and stored in two consecutive memory slots. Hence, BRAM_1 holds the topmost 64 bits of all 256 RFSB constants, BRAM_2 the following 64 bits of all RFSB constants and so forth.

The index into the table is the current message byte m_i appended by a zero and a one bit to address both memory slots. Because of the dual-port layout of the block memories, both 32-bit memory slots can be read out simultaneously. This is done for all block memories at the same time and the results are concatenated and rotated to form the constant $\text{ROL}_x\left(c\left[m_i\right]\right)$. This setup allows to read out a complete and already rotated RFSB-509 constant in one clock cycle.

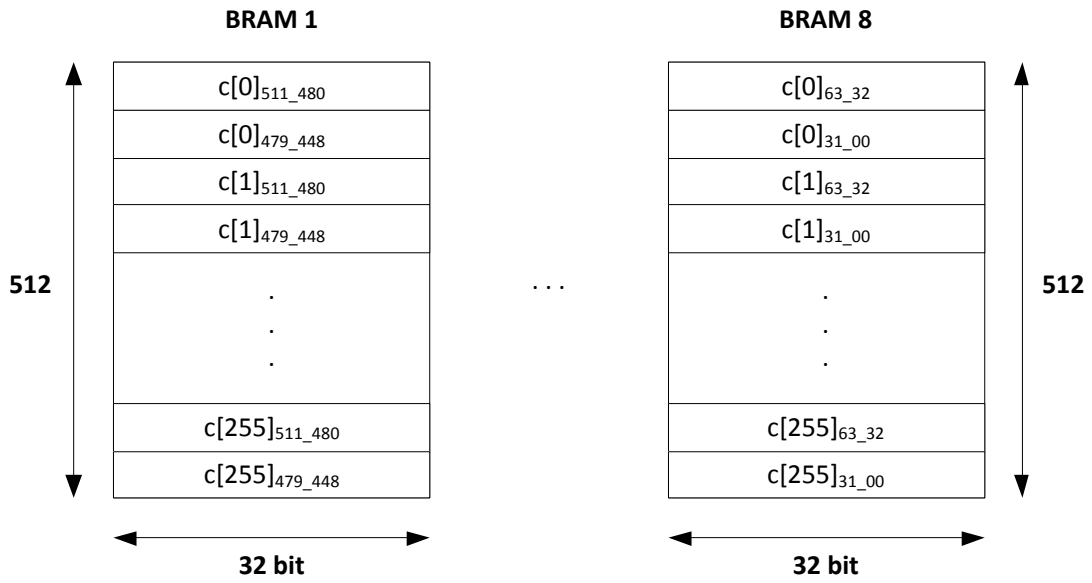


Figure 8.3: Our smallest BRAM-based FPGA implementation of RFSB-509 requires 8 block memories configured as 512×32 bit dual-port ROM. Every BRAM holds a 64-bit chunk of the 509-bit constants (prepended by three zero bits) which is split into two 32-bit parts. Since two memory cells of each BRAM can be read out in one clock cycle, one constant can be read out in one clock cycle.

We sequentially iterate over all input message bytes, accumulate the corresponding constants, and reduce the result after all message bytes have been processed.

Due to its tree-like structure, RFSB allows very scalable designs which can process multiple message bytes in one clock cycle since the inputs to the block memories are independent of each other. Below we explore designs that implement multiple constant look-ups per clock cycle.

Wide-Access Block Memories

This architecture uses block memories with wide-access ports to provide the matrix constants. Creating a 256×509-bit table using the Xilinx block memory generator results in 15 occupied 18-Kbit BRAMs. This architecture allows to read out two RFSB-509 constants in one clock cycle, thus reducing the necessary cycles spent for table look-ups from 112 to 56 cycles.

The internal compression module handles two bytes at once and applies two different rotations to the read-out constants depending on the position of the message byte in the input string. In the first mode, the first constant is rotated by 384-bit, the second constant by 256-bit. In the second mode the first constant is rotated by 128-bit and the second constant is not rotated at all. Both constants are accumulated to the intermediate result. The rotation mode is switched with every input message pair and after the complete input block has been processed, the result is reduced modulo $x^{509} - 1$.

Multiple Table Instances

For high-performance applications we explore architectures in which we duplicate the wide-access block memories that contain the compressed matrix constants. We only go so far that we remain within reasonable resource boundaries (i.e., realizable on Spartan-6 devices).

In the first setting we use two tables which allow to process four message bytes in one clock cycle, essentially representing the *basic compression unit* introduced in Section 8.3.3 and Figure 8.2. Furthermore, it is now possible to hard-wire the rotations applied to the constants because the output of each of the block memory ports only handles either $c[m_{4i+1}]$, $c[m_{4i+2}]$, $c[m_{4i+3}]$, or $c[m_{4i+4}]$, $0 \leq i \leq 27$. The two tables require 29 block memories and again halve the required cycles to 28 clock cycles for the constant look-ups of a 112-byte input block.

In a second design we use four separate instances of the constant table, requiring 58 BRAMs. This architecture allows to look up eight message bytes per clock cycle and finishes the look-ups after 14 clock cycles.

The third design quadruples the amount of block memories and thus contains eight instances of the constants table. This requires 116 BRAMs and allows to look-up 16 constants at the same time which means all constants are retrieved after just 7 clock cycles.

8.5.2 Implementing RFSB-509 with AES-128

To complete our design exploration, we include an on-the-fly generation of the matrix constants using an AES-128 FPGA implementation. Since the key is always fixed to the all-zero key, the key-schedule does not have to be implemented as the round-keys can be precomputed. This is only true if the AES core is not used for other applications which require the key to be adjustable at run-time. The AES in use is a T-table based implementation that occupies eight block memories for storing the tables.

The constant computation unit uses a straightforward approach. It receives a message byte and starts four consecutive AES-128 encryptions with the respective input blocks as described in Section 8.3.2. Each result is XORed to an internal output signal and after the fourth encryption is finished, a modular reduction is performed, and the constant is returned. The higher level unit receives the constant, rotates it according to the position of the current message byte and passes the next message byte to the constant computation unit.

8.6 Results and Comparison

Our implementations are verified against test-vectors generated using the reference implementation of RFSB-509 by Schwabe which was submitted to the ECRYPT Benchmarking of Cryptographic Systems (eBACS) [eBA15a]. The results for embedded microcontrollers are reported based on Atmel's AVR Studio 6. In addition to simulation, our implementations were tested on a real device, namely on the AVR XPLAIN board equipped with an ATxmega128A1 microcontroller. The FPGA results are post place-and-route results reported by Xilinx' ISE Design Suite 14.1, and the target device is a Spartan-6 FPGA XC6SLX100.

We omit the output filter because a wide range of SHA-256 implementation is already available in hard- and software. The output filter arguably does not affect the performance measurements when hashing long messages since it is only applied once to the output of the RFSB-509 compression function. In the following we present our microcontroller and FPGA results and compare them to other hash function implementations on similar devices.

8.6.1 Embedded Microcontrollers

Table 8.1 shows the results of our implementations of RFSB-509 on the embedded microcontroller ATxmega128A1. The performance is measured in cycles/byte, the amount of clock cycles required for calling the *update* function is divided by 48 bytes since only 48 message bytes enter each compression function due to the Merkle-Damgård construction. Thus, these figures represent the realistic performance for hashing long messages.

Table 8.1: Implementation results of RFSB-509 on ATxmega128A1 microcontrollers. *Results for the SRAM table based implementations are measured on an ATxmega384C3 since it provides more SRAM.

Design	**ROM [bytes]**	**RAM [bytes]**	**Cycles/ byte**	**Used ROM**	**Used RAM**
HW-AES	732	232	4,753.1	0.5%	2.8%
ROM table	602+16384	232	1,573.9	12.2%	2.8%
ROM table unrolled	3,100+16384	232	1,114.9	14.0%	2.8%
RAM table*	996	232+16,384	1,424.5	4.2%	50.7%
RAM table unrolled*	3,494	232+16,384	965.6	4.9%	50.7%

All implementations require 232 bytes of SRAM, split into a 112-byte state, a 48-byte input, a 64-byte output and an 8-byte counter. Additional 16 Kbytes of ROM/SRAM are used by the ROM/SRAM-based table implementations to store the constants table. The fastest microcontroller implementation is running at 965.6 cycles/byte, but is so far only realizable on a few 8-bit AVR microcontrollers since at least the ATxmega384 device has to be used to meet the RAM requirements. The fastest ROM-based implementation computes one RFSB-509 round at 1114.9 cycles/byte. The rolled version does not seem to be a good choice, since program memory at this size is not a problem for current microcontrollers, and spending additional 2.5 Kbytes of ROM (+1.6%) seems to be worth the 460 cycles/byte (30%) performance improvement.

Our smallest implementation, which is based on on-the-fly AES encryptions, only requires 732 bytes ROM and falls into the lightweight cryptography category. If ROM memory is scarce, the current version could be implemented even smaller by removing unrolled loops which currently improve performance. Since for every constant the AES encryption is called four times, 448 AES encryptions are needed during each compression. Assuming 500 clock cycles for each AES encryption we get a lower bound of 224,000 clock cycles or 4,666.7 cycles/byte for the encryptions, not counting rotations, modular reductions and the combination of looked-up constants to form the output. Our result of 4,753.1 cycles/byte comes very close to this lower bound.

Table 8.2: Comparison of the lightweight RFSB-509 implementation with lightweight implementations of wide-spread hash functions as presented in [BEE+13].

Hash	**Platform**	**ROM [bytes]**	**RAM [bytes]**	**Cycles/ byte**
RFSB-509	ATxmega128	732	232	4,753
SHA-256	ATtiny45	1,090	143	532
BLAKE-256	ATtiny45	1,166	193	562
Grøstl-256	ATtiny45	1,400	201	686
JH-256	ATtiny45	1,020	234	5,062
Keccak	ATtiny45	868	244	1,432
Skein-256	ATtiny45	988	232	4,788
PHOTON	ATtiny45	1,244	78	6,210
SPONGENT	ATtiny45	364	101	50,908

Although RFSB-509 fits well on current embedded microcontrollers and performs at a decent speed, beating implementations of the SHA-3 candidates is not possible due to memory requirements caused by the size of the constants' matrix. When comparing the lightweight AES-based implementation to the results of an ECRYPT initiative that aimed to provide a comprehensive collection of lightweight implementations of hash functions [BEE+13], RFSB-509 beats well known hash functions such as SHA-256, BLAKE-256, JH-256, and Skein-256 in terms of code size and outperforms JH-256 and sponge-based construction such as PHOTON and SPONGENT (cf. Table 8.2). However, it has to be noted that the other implementations do not make use of crypto accelerators since they target the AVR ATtiny device family.

8.6.2 Reconfigurable Hardware

Table 8.3 shows our FPGA results taken from post place-and-route reports. The designs that use BRAM tables are named RFSB-509$_x$, where x denotes the amount of used block memories. To measure the performance of our implementations we count the required clock cycles to load new message bits into the Merkle-Damgård state, compress the current state and update the state accordingly. We divide the number of clock cycles by 48 since in the Merkle-Damgård construction only 48 new message bytes enter each 112-byte compression. In addition, we compute the achieved throughput of each implementation as $\mathrm{Tp} = \frac{\text{clock frequency} \times 8}{\text{cycles/byte}}$.

The amount of utilized block memories directly correlates with the achievable performance. When using the minimum of 8 BRAMs we reach a throughput of 805.1 Mbit/s. Our fastest implementation runs at 5.35 Gbit/s and consumes 116 block memories. A designer is thus left with the decision of how many block memories to spend to reach a certain performance goal.

The required area on an FPGA is measured in terms of flip-flops, LUTs, and BRAMs. We also include the number of occupied slices for comparison even though this number has to be considered with care since the slice count itself does not reveal the actual degree of used logic inside the slice and neglects the number of occupied embedded resources (e.g., DSPs and BRAMs). The overall slice count stays on the same level for nearly all of our implementations.

Table 8.3: Implementation results of different designs of RFSB-509 for Xilinx Spartan-6 XC6SLX100 FPGAs. We report the occupied slices, flip-flops (FF), 6-input look-up tables (LUT), and the maximum clock frequency f. The performance is reported in terms of cycles/byte, throughput (Tp), and throughput/area ratio (Tp/Area).

Design	**Slices**	**FFs**	**LUTs**	**18-Kbit BRAM**	f **[MHz]**	**Cycles/ byte**	**Tp [Mbit/s]**	**Tp/Area** $[\frac{\textbf{Mbit/s}}{\textbf{Slices}}]$
AES-based	1,526	5,793	4,920	8	260.2	213.8	9.3	0.01
RFSB-509_8	1,402	4,621	4,316	8	259.4	2.46	805.1	0.57
RFSB-509_{15}	1,381	4,106	4,277	15	234.7	1.25	1,432.8	1.04
RFSB-509_{29}	1,409	4,101	4,309	29	223.0	0.65	2,633.9	1.87
RFSB-509_{58}	1,447	4,070	3,709	58	171.1	0.38	3,480.2	2.41
RFSB-509_{116}	2,112	4,071	4,690	116	146.2	0.21	5,354.0	2.54

Only the fastest implementation occupies more slices, but the amount of used flip-flops and LUTs does not increase on the same scale. This is due to fact that block memories are spread out across the FPGA. Usually this leaves more freedom of where to place an implementation on the FPGA, but when combining more than just a few BRAMs, the design is spread across the FPGA which leads to partially used slices. This also increases the critical path which explains the decreasing clock frequency for the 58 and 116 BRAM variants.

Note, the performance and size of the AES-based design is inherently depended on the underlying AES core. Nevertheless, using on-the-fly constant generation on an FPGA does not seem to be a good choice since the required resources are nearly the same as in our smallest BRAM implementation plus additional logic for the AES core (393 flip-flops, 326 LUTs, 130 slices, 8 BRAMs, and 21 clock cycles for one encryption). The performance is two orders of magnitude lower. A possible scenario in which an AES-based implementation could be favorable is if no block memories are available in the FPGA or if they are already occupied by other parts of the application. This would of course require a none BRAM-based AES implementation as well.

We compare our results to an evaluation of the hardware performance of the five SHA-3 finalists [GHR+12] and a recent implementation of the lattice-based hash function SWIFFTX [GCHB12] in Table 8.4. When comparing the numbers one has to keep in mind that our implementation results are achieved on low-cost Xilinx Spartan-6 devices while the other results are measured on high-end Virtex-5 and Virtex-6 devices. Nevertheless, our implementations keep up with most implementations and get clearly outperformed only by the Keccak-256 implementation.

8.7 Conclusion

We presented the first implementations of RFSB-509 for embedded microcontrollers and reconfigurable hardware. Lightweight to high-performance designs have been evaluated and proven feasible on both platforms with competitive results in code size/area and performance. Our

Table 8.4: This table compares our results to other hash functions implemented in FPGAs. The results of [GHR+12] are given for high-end Xilinx Virtex-6 devices, [GCHB12] for Xilinx Virtex-5 and our results for the low-cost Xilinx Spartan-6.

Hash Function	**Slices**	**Tp [Gbit/s]**	**Tp/Area** $[\frac{\textbf{Mbit/s}}{\textbf{Slices}}]$	**Device [Xilinx]**
RFSB-509_{58}	1,447	3.48	2.41	Spartan-6
RFSB-509_{116}	2,112	5.34	2.54	Spartan-6
SWIFFTX [GCHB12]	16,645	4.85	0.29	Virtex-5
SHA-256 [GHR+12]	239	1.63	6.83	Virtex-6
Helion Fast SHA-256 [Hel15b]	214	1.5	7.01	Spartan-6
BLAKE-256 [GHR+12]	2,530	8.06	3.18	Virtex-6
Grøstl-256 [GHR+12]	898	4.20	4.68	Virtex-6
JH-256 [GHR+12]	849	5.41	6.37	Virtex-6
Keccak-256 [GHR+12]	1,474	18.80	12.76	Virtex-6
Skein-256 [GHR+12]	1,628	6.21	3.82	Virtex-6

result show that code-based hash functions are practical and suitable candidates even for applications involving embedded systems.

In light of NIST's decision to select Keccak as SHA-3, one of the most important selection criteria seems to have been to pick a hash function which is not based on previous SHA-1/-2 structures and neither on an AES-inspired design. Presumably the selection was made to spread risk across a larger variety of cryptographic functions so that a successful attack on one of the cryptographic primitives does not affect other NIST standards. Code-based hash functions could add to this variety with RFSB-509. Furthermore, RFSB performance considerably improved over the SHA-3 candidate FSB which was ruled out mainly due to its inefficiency.

Part II

Hash-Based Digital Signatures

Chapter 9

Hash-Based Digital Signature Schemes

This chapter introduces hash-based digital signature schemes with a focus on the Merkle signature scheme in combination with Winternitz one-time signatures. Furthermore, we explain efficient generation of one-time signing keys using PRNGs and provide insights into the BDS algorithm for efficient authentication path computation. The chapter concludes with a survey of the security arguments for hash-based signature schemes.

Contents

9.1 Introduction to Hash-Based Signatures

The first part of this thesis focused on alternative public-key encryption based on coding theory. In the second part we shift the focus to another important use-case of public-key cryptography: *digital signatures*. With the increasing popularity of contactless smart cards and near field communication, digital signature engines have become a key component of many embedded system solutions. The applications of digital signatures are numerous, ranging from identification over electronic payments to firmware updates and protection against product counterfeiting. Due to the high computational requirements for public-key cryptography, providing efficient digital signatures on embedded microprocessors with and without dedicated co-processors remains a challenging task.

Wide-spread digital signature schemes are RSA (e.g., PKCS#1 [RSA12]), the digital signature algorithm DSA [NIS13], and its elliptic-curve equivalent ECDSA [NIS13]. A new contender are Edwards-curve digital signatures EdDSA [BDL+12], which allow faster computations and easier achievable secure software implementations compared to ECDSA. The underlying hard problems of these digital signature schemes however are known to broken by quantum computers [Sho97]. Even if scalable quantum computers would never be built, and the hardness of the factorization and discrete logarithm problems continue to hold, exploring alternative signatures schemes is still worthwhile to identify schemes which provide stronger security arguments, are more efficient, or offer inherent countermeasures against side-channel attacks.

Popular candidates for alternative digital signatures are grouped into different families similarly as alternative public-key encryption schemes. An early example is Stern's identification scheme [Ste94] which bases its security on the syndrome decoding problem from coding theory. The required interaction between prover and verifier in Stern's scheme can be removed using the Fiat-Shamir transform [FS87], but this leads to a somewhat inefficient signature scheme.

The first digital signature scheme from coding theory that could be considered practical is CFS [CFS01]. CFS is based on the Niederreiter cryptosystem [Nie86] and its security stems from the syndrome decoding problem and the indistinguishability of binary Goppa codes. An unpublished attack by Bleichenbacher based on the Generalized Birthday Problem [Wag02] reduced the assumed security level of the original CFS parameter set as it lowers the attack complexity from $2^{r/2}$ to $2^{r/3}$. An adapted scheme (Parallel-CFS) was introduced [Fin11] since CFS becomes impractical when adjusting the parameters to account for the attack due to large keys in the range of gigabytes. CFS and Parallel-CFS were implemented on a desktop PC in [LS12] followed by a bitsliced implementation whose signing time is independent of secret data [BCS13]. The implementation results show that code-based signature schemes are becoming more efficient but so far cannot compete with classical digital signature schemes, especially regarding the public-key size and the signing time which are in the range of megabytes and few signatures per second, even with highly optimized implementations on powerful CPUs.

Digital signature schemes based on hash function evaluations were introduced in [Mer90]. The main idea of the Merkle Signature Scheme (MSS) is to sign messages with a One-Time Signature Scheme (OTSS) and to authenticate the one-time verification keys using binary hash trees. Several improvements to different parts of Merkle's signature scheme were proposed over time, recent proposals of hash-based signature schemes include XMSS [BDH11], XMSS$^+$ [HBB13], XMSS^{MT} [HRB13], and SPHINCS [BHH+15].

In this work we explore hash-based digital signature scheme for mainly two reasons. First, it was shown in [Hül13] that the security of hash-based signature schemes can be reduced to the collision resistance or even to the second-preimage resistance of the underlying hash function which arguably is a minimal assumption for digital signature schemes. Second, hash-based signature schemes are usually built upon one-time signature schemes which inherently provides the possibility of leakage-resilience since the signing keys are ever-changing.

An additional advantage of hash-based signature schemes is that they allow for low-effort disaster recovery in the unlikely case that the employed hash function is broken by an attack. The hash function can simply be replaced by any other cryptographically secure hash function which provides at least the same output length. Other parts of the scheme remain unchanged which is not as easily possible with classical signature schemes.

The main goals of this work are to provide an efficient implementation of MSS with a focus on the challenges when implementing on constrained embedded systems, to design the scheme such that it offers protection against side-channel attacks, and to quantify and reduce the maximum side-channel leakage of involved secrets.

In the following we introduce the foundations of hash-based digital signatures schemes. The Merkle signature scheme, Winternitz one-time signatures, efficient private-key generation, and the authentication path computation which is the main computational challenge when signing a message are explained.

9.2 The Merkle Signature Scheme

The Merkle signature scheme is a popular hash-based signature scheme that was introduced in [Mer90]. A detailed description of the Merkle signature scheme and its variants can be found in [BDS09].

MSS is based on the availability of an at least second preimage resistant, undetectable n-bit one-way function f and a cryptographic m-bit hash function g:

$$f : \{0,1\}^n \to \{0,1\}^n, \;\; g : \{0,1\}^* \to \{0,1\}^m .$$

The height of the Merkle tree H predetermines the number of signatures that are verifiable with a MSS verification key (2^H signatures). The number of signatures can be extended with the concept of tree chaining which was introduced in [BGD+06] and extended to virtually unlimited signatures in [BDK+07].

9.2.1 MSS Key Generation

Let the nodes of the Merkle tree be denoted by $\nu_h[s]$ with $h \in \{0, \ldots, H\}$ being the height of the node and $s \in \{0, \ldots, 2^{H-h} - 1\}$ being the node index on height h.

First, 2^H one-time signing key-pairs (X_i, Y_i) are generated using $\mathsf{KeyGen}_{\mathrm{OTS}}(1^n)$, the key-generation algorithm of the underlying OTSS. The 2^H leaves of the Merkle tree are defined to be digests $g(Y_i)$ of one-time verification keys Y_i which correspond to one-time signing keys X_i, $0 \leq i < 2^H$. Starting from the leaves, the root of the Merkle tree $\nu_H[0]$ is generated as

$$\nu_{h+1}[i] = g(\nu_h[2i] \,||\, \nu_h[2i+1]), \;\; 0 \leq h < H, \;\; 0 \leq i < 2^{H-h-1}.$$

Hence, a parent node is generated by hashing the concatenation of its two child nodes. The root of the tree is defined to be the MSS verification key. A Merkle tree of height $H = 3$ is illustrated in Figure 9.1.

Even for small tree heights H storing all nodes of the tree quickly becomes costly, especially in memory constraint environments such as embedded microcontrollers. The Treehash algorithm [Mer90, Szy04] provides a memory efficient way of computing the root node and requires to store at most H nodes at the same time. We list the Treehash algorithm in Algorithm 3. The algorithm stores nodes on a stack and computes parent nodes as soon as both children

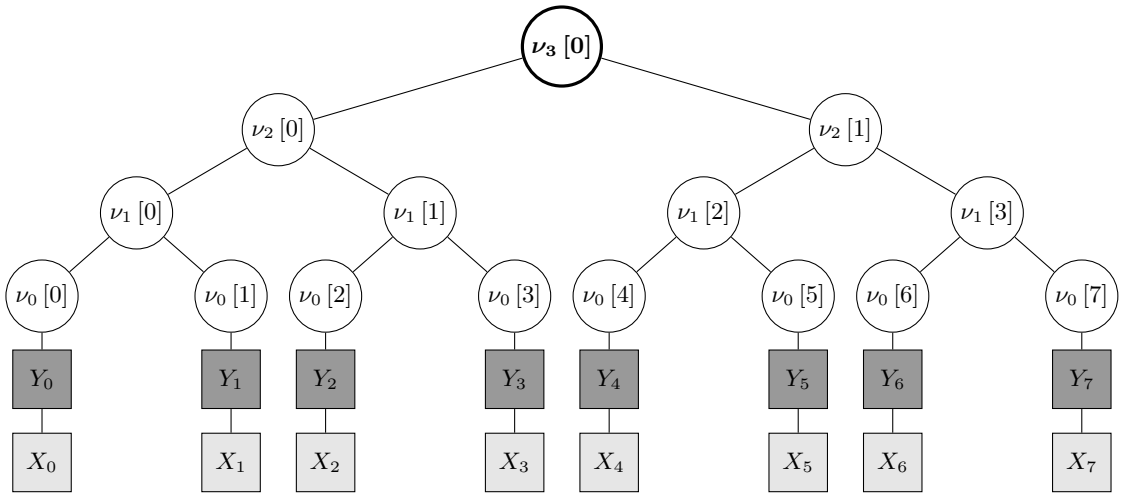


Figure 9.1: A Merkle tree of height $H = 3$. The leaves $\nu_0\,[i] = g(Y_i)$ are computed by hashing the one-time verification keys Y_i. Inner nodes are computed by hashing the concatenation of its two children, e.g., $\nu_1\,[0] = g(\nu_0\,[0]\,||\,\nu_0\,[1])$. The MSS verification key is the root node $\nu_3\,[0]$.

are available. The child nodes are removed from the stack if they are no longer required, i.e., after their parent was computed. Tree leaves are computed on-the-fly using the generic method LEAFCALC(i) which computes the i-th leaf of the Merkle tree. For now we can assume this to be $\mathsf{KeyGen}_{\mathrm{OTS}}$, more details on this topic will be provided in Section 9.4. For a Merkle tree of height H, the TREEHASH algorithm calls the method LEAFCALC in total 2^H times to compute all leaves. Hash function g is called $2^H - 1$ times to compute the root of the tree. Figure 9.2 illustrates the order in which nodes are generated by the TREEHASH algorithm for the same Merkle tree of height $H = 3$ as in Figure 9.1.

Algorithm 3 TREEHASH [MER90, SZY04]

Input: Tree height $H \geq 2$
Output: Tree root $\nu_H\,[0]$

```
for i = 0 → 2^H − 1 do
    ν_0 [i] ← LEAFCALC(j)
    while NODE_1 has the same height as the top node on STACK do
        NODE_2 ← STACK.pop()
        Compute parent: NODE_1 ← g(NODE_2 || NODE_1)
    end while
    Push parent to stack: STACK.push(NODE_1)
end for
return STACK.pop()
```

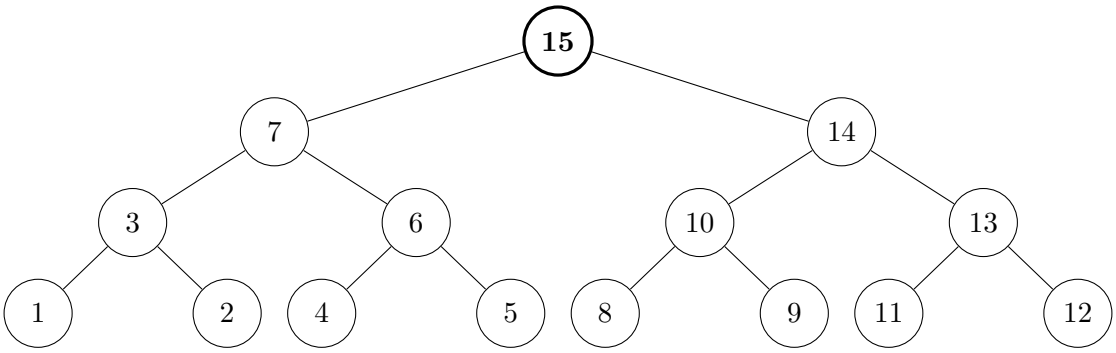


Figure 9.2: Given a Merkle tree of height $H = 3$, the TREEHASH algorithm (Algorithm 3) computes the nodes $\nu_h[i]$ of the tree in the listed order. The leaves are computed using LEAFCALC, all other nodes of the tree are the results of hashing its two child nodes.

9.2.2 MSS Signature Generation

A Merkle signature σ_s of a message M is computed as follows. The signature $\sigma_s(d)$ of a digest $d = g(M)$ consists of a signature index s, a one-time signature σ_{OTS}, a one-time verification key Y_s, and an authentication path $(\text{AUTH}_0, \ldots, \text{AUTH}_{H-1})$ that allows the verification of the one-time signature with respect to the public MSS verification key, hence the signature $\sigma_s(d)$ is defined as

$$\sigma_s(d) = (s, \sigma_{\text{OTS}}, Y_s, (\text{AUTH}_0, \ldots, \text{AUTH}_{H-1})).$$

The signature index $s \in \{0, \ldots, 2^H - 1\}$ is incremented with every issued signature. The one-time signature scheme is used to sign the digest under signing key X_s to generate the one-time signature

$$\sigma_{\text{OTS}} = \mathsf{Sign}_{\text{OTS}}(d, X_s)$$

of the message digest d. The authentication path for the s-th leaf are all sibling nodes AUTH_h, $h \in \{0, \ldots, H-1\}$ on the path from leaf $\nu_0[s]$ to the root node $\nu_H[0]$. It enables the verifier to recompute the root node of the Merkle tree and hence to verify the authenticity of the current one-time signature even though the verifier has not seen the one-time verification key Y_i beforehand. An example is given in Figure 9.3 in which the authentication nodes for leaf $\nu_0[1]$ are marked.

We would like to stress that the signature generation reflects the structure of an online/offline signature scheme. The authentication path only depends on the OTSS verification key Y_s which is known prior to the message. Hence, the authentication path can be precomputed. The online phase can then be processed faster by only hashing the message and signing the hash with the one-time signature scheme.

9.2.3 MSS Signature Verification

Given a digest $d = g(M)$ and its signature $\sigma_s(d)$ the verifier plugs the one-time signature σ_{OTS} into the underlying one-time signature verification algorithm $\mathsf{Verify}_{\text{OTS}}(d, \sigma_s(d))$ to verify the

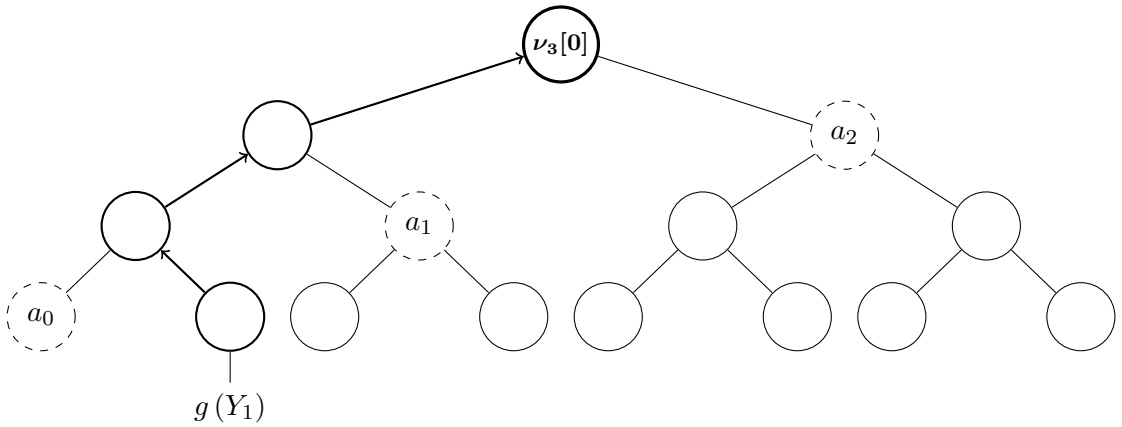


Figure 9.3: The authentication path for leaf $\nu_0[1] = g(Y_1)$ in a Merkle tree of height $H = 3$ is $A_1 = (\text{AUTH}_0, \text{AUTH}_1, \text{AUTH}_2) = (\nu_0[0], \nu_1[1], \nu_2[1])$. Given Y_1 and A_1, it is possible to reconstruct the root node $\nu_3[0]$ and to verify the authenticity of Y_1.

validity of a provided signature. If the verification succeeds, the root node is reconstructed using the provided authentication path.

$$\phi_{h+1} = \begin{cases} g(\phi_h \,||\, \text{AUTH}_h), \text{ if } \lfloor s/2^h \rfloor \equiv 0 \bmod 2 \\ g(\text{AUTH}_h \,||\, \phi_h), \text{ if } \lfloor s/2^h \rfloor \equiv 1 \bmod 2 \end{cases}, \quad \phi_0 = \nu_0[s], \quad h = 0, \ldots, H-1.$$

The MSS signature is accepted if the one-time signature σ_{OTS} is successfully verified and if the root node of the Merkle tree was reconstructed, i.e., if ϕ_H is equal to the root node $\nu_H[0]$.

9.3 Winternitz One-Time Signatures

Winternitz one-time signatures (W-OTS) [DSS05] are a convenient choice for the one-time signature scheme in MSS as they reduce the overall signature size compared to, e.g., the Lamport-Diffie one-time signature scheme [Lam79]. The Winternitz parameter $w \geq 2$ determines how many bits are signed simultaneously. Parameter t is defined as

$$t = t_1 + t_2, \quad t_1 = \left\lceil \frac{n}{w} \right\rceil, \quad t_2 = \left\lceil \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rceil$$

and determines of how many random n-bit strings x_i the Winternitz signing keys consist.

9.3.1 W-OTS Key Generation

A W-OTS signing key $X = (x_0, \ldots, x_{t-1})$ is generated by selecting t random bit strings $x_i \in \{0,1\}^n$, $0 \leq i < t$. The W-OTS verification key $Y = g(y_0 \,||\, \ldots \,||\, y_{t-1})$ is computed from the signing key by applying f $2^w - 1$ times to each x_i giving $y_i = f^{2^w-1}(x_i)$, $0 \leq i < t$ and computing the hash of the concatenated y_i. Hence, the verification key is computed as

$$Y = g(y_0 \,||\, \ldots \,||\, y_{t-1}) = g\left(f^{2^w-1}(x_0) \,||\, \ldots \,||\, f^{2^w-1}(x_{t-1})\right).$$

Note, the superscript denotes multiple executions of f, e.g., $f^2(x_i) = f(f(x_i))$ and $f^0(x_i) = x_i$.

9.3.2 W-OTS Signature Generation

A signature for a message M is created by signing its digest $d = g(M)$ under key X. Digest d is divided into t_1 blocks $b_0, \ldots, b_{t_1-1}$ of length w, and a checksum $c = \sum_{i=0}^{t_1-1} (2^w - b_i)$ is computed. Checksum c is divided into t_2 blocks $b_{t_1}, \ldots, b_{t-1}$ of length w (zero-padding to the left is applied if c or d are not multiples of w). The W-OTS signature $\sigma_{\text{W-OTS}} = (\sigma_0, \ldots, \sigma_{t-1})$ is computed with $\sigma_i = f^{b_i}(x_i)$, $0 \leq i < t$. Hence, the W-OTS signature is computed as

$$\sigma_{\text{W-OTS}} = (\sigma_0, \ldots, \sigma_{t-1}) = \left(f^{b_0}(x_0), \ldots, f^{b_{t-1}}(x_{t-1})\right).$$

9.3.3 W-OTS Signature Verification

Given a message digest $d = g(M)$, a signature $\sigma_{\text{W-OTS}}$, and a verification key Y_s, the verifier generates blocks $b_0, \ldots, b_{t-1}$ from d as done during signature generation and reconstructs

$$\begin{aligned} Y'_s &= g\left(f^{2^w-1-b_0}(\sigma_0) \mid\mid \ldots \mid\mid f^{2^w-1-b_{t-1}}(\sigma_{t-1})\right) \\ &= g\left(f^{2^w-1-b_0}\left(f^{b_0}(x_0)\right) \mid\mid \ldots \mid\mid f^{2^w-1-b_{t-1}}\left(f^{b_{t-1}}(x_{t-1})\right)\right) \\ &= g\left(f^{2^w-1}(x_0) \mid\mid \ldots \mid\mid f^{2^w-1}(x_{t-1})\right). \end{aligned}$$

If Y'_s equals Y_s the signature is valid, otherwise it has to be rejected. When using W-OTS signatures in MSS, transmitting Y_s and comparing Y_s to Y'_s can be omitted. Y'_s can simply be used together with the nodes of the authentication path to recompute the root of the Merkle tree. If the recomputed root equals the MSS public-key, then Y'_s is a valid OTS verification key.

9.4 Signing Key Generation

Providing enough memory to store 2^H one-time signature keys can quickly become problematic on constrained devices even for small tree heights. Instead of storing the keys, a PRNG can be used to generate the keys when needed as proposed in [RED+08] resulting in significantly reduced storage requirements. On input of a seed k_i the PRNG outputs a random string r_{i+1} and an updated seed k_{i+1}, each of length n.

$$\textsc{Prng} : \{0,1\}^n \rightarrow \{0,1\}^n \times \{0,1\}^n, k_i \rightarrow (k_{i+1}, r_{i+1}) \tag{9.1}$$

The MSS signing key is reduced to the initial seed $\textsc{Seed}_0 \in_R \{0,1\}^n$ which is given to the PRNG. This initial seed is used in a two stage process to generate W-OTS signing keys X_i. First, PRNG seeds for all W-OTS signing keys denoted as $\textsc{Seed}_{\text{W-OTS}_i}$ are derived from $\textsc{Seed}_0$:

$$(\textsc{Seed}_{i+1}, \textsc{Seed}_{\text{W-OTS}_i}) \leftarrow \textsc{Prng}(\textsc{Seed}_i), \;\; 0 \leq i < 2^H. \tag{9.2}$$

Then, the W-OTS signing keys $X_i = (x_0, \ldots, x_{t-1})$, $0 \leq i < 2^H$ are generated by the PRNG starting from $\textsc{Seed}_{\text{W-OTS}_i}$. More specifically, the t n-bit strings of the i-th W-OTS signing key $X_i = (x_0, \ldots, x_{t-1})$, $0 \leq i < 2^H$ are generated by

$$(\textsc{Seed}_{\text{W-OTS}_i}, x_j) \leftarrow \textsc{Prng}(\textsc{Seed}_{\text{W-OTS}_i}), \;\; 0 \leq j < t. \tag{9.3}$$

9.5 Authentication Path Computation

As already mentioned in Section 9.2.2, the authentication path allows to link one-time verification keys to the overall public-key in MSS (cf. Figure 9.3). Its computation can be costly with naïve approaches such as simply storing all nodes of the tree or recomputing the Merkle tree every time an authentication path is required.

Creating an authentication path for a specific leaf s can be easily accomplished by storing all tree nodes in memory and looking up the required nodes when needed. However, this approach is infeasible for reasonable applications because of the exponential growth of nodes in the tree height H. Hence, it is necessary to compute and update the authentication path when signing messages.

A straightforward approach would be to simply compute the Merkle tree when signing a message and storing the authentication nodes. Although the memory requirements of this approach are moderate (the stack in TREEHASH and the authentication nodes), the computational complexity is very high since almost the complete Merkle tree has to be computed for each MSS signature.

The best known algorithm for on-the-fly computation of authentication nodes is the BDS algorithm [BDS09] (cf. Algorithm 4). It makes use of several treehash algorithm instances TREEHASH_h for heights $0 \leq h \leq H - K - 1$ and allows to efficiently create (parts of) Merkle trees. In the BDS algorithm each instance is initialized with a leaf index s for which it computes the corresponding node value. Each instance is updated until the required authentication node is computed. During a treehash update the next leaf is created and parent nodes are computed if possible.

The generation of the authentication path is divided into two parts that go alongside with the key and signature generation of MSS. During key generation all treehash instances TREEHASH_h are initialized with $\nu_h\,[3]$, and the first authentication path is stored:

$$\text{AUTH}_h = \nu_h\,[1]\,,\ 0 \leq h \leq H - 1.$$

The BDS algorithm generates left authentication nodes either by computing the leaf value or by one hash function evaluation of a concatenation of two previously computed nodes from memory. Right authentication nodes are more expensive to generate since they are computed from the leaf up. Since right nodes close to the top are most expensive to compute, a positive integer $K \geq 2, (H - K$ even) decides how many of these nodes are stored in RETAIN_h during key generation, $H - K \leq h \leq H - 2$.

The authentication nodes change every 2^h steps for height h. During signature generation the treehash instances are updated and if an authentication node from a treehash instance is used, the instance is re-initialized to compute the next authentication node for that particular height.

Algorithm 4 Algorithm for BDS Authentication Path Computation [BDS09]

Input: $s \in \left\{0, \ldots, 2^H - 2\right\}, H, K$, and the algorithm state.
Output: Authentication path A_{s+1} for leaf $s+1$.

1: Let $\tau = 0$ if leaf s is a left node or let τ be the height of the first parent of leaf s which is a left node: $\tau \leftarrow \max\{h : 2^h | (s+1)\}$

2: If the parent of leaf s on height $\tau + 1$ is a left node, store the current authentication node on height τ in KEEP_τ:
if $\lfloor s/2^{\tau+1} \rfloor$ is even **and** $\tau < H - 1$ **then** $\text{KEEP}_\tau \leftarrow \text{AUTH}_\tau$

3: If leaf s is a left node, it is required for the authentication path of leaf $s+1$:
if $\tau = 0$ **then** $\text{AUTH}_0 \leftarrow \text{LEAFCALC}(s)$

4: Otherwise, if leaf s is a right node, the auth. path for leaf $s+1$ changes on heights $0, \ldots, \tau$:
if $\tau > 0$ **then**

a) The authentication path for leaf $s+1$ requires a new left node on height τ. It is computed using the current authentication node on height $\tau - 1$ and the node on height $\tau - 1$ previously stored in $\text{KEEP}_{\tau-1}$. The node stored in $\text{KEEP}_{\tau-1}$ can then be removed:
$\text{AUTH}_\tau \leftarrow g\left(\text{AUTH}_{\tau-1} \,\|\, \text{KEEP}_{\tau-1}\right)$, remove $\text{KEEP}_{\tau-1}$

b) The authentication path for leaf $s+1$ requires new right nodes on heights $h = 0, \ldots, \tau - 1$. For $h < H - K$ these nodes are stored in TREEHASH_h and for $h \geq H - K$ in RETAIN_h:
for $h = 0$ **to** $\tau - 1$ **do**
if $h < H - K$ **then** $\text{AUTH}_h \leftarrow \text{TREEHASH}_h.\text{pop}()$
if $h \geq H - K$ **then** $\text{AUTH}_h \leftarrow \text{RETAIN}_h.\text{pop}()$

c) For heights $0, \ldots, \min\{\tau - 1, H - K - 1\}$ the Treehash instances must be initialized anew. The Treehash instance on height h is initialized with the start index $s + 1 + 3 \cdot 2^h < 2^H$:
for $h = 0$ **to** $\min\{\tau - 1, H - K - 1\}$ **do**
$\text{TREEHASH}_h.\text{initialize}(s + 1 + 3 \cdot 2^h)$

5: Next we spend the budget of $(H - K)/2$ updates on the Treehash instances to prepare upcoming authentication nodes:
repeat $(H - K)/2$ **times**

a) We consider only stacks which are initialized and not finished. Let k be the index of the Treehash instance whose lowest tail node has the lowest height. In case there is more than one such instance we choose the instance with the lowest index:
$k \leftarrow \min\left\{h : \text{TREEHASH}_h.\text{height}() = \min_{j=0,\ldots,H-K-1}\left\{\text{TREEHASH}_j.\text{height}()\right\}\right\}$

b) The Treehash instance with index k receives one update: $\text{TREEHASH}_k.\text{update}()$

6: The last step is to output the authentication path for leaf $s+1$:
return $\text{AUTH}_0, \ldots, \text{AUTH}_{H-1}$.

9.6 Security of Hash-Based Signature Schemes

The security properties of the MSS signature scheme are discussed in [BDS09]. Specifically, the work shows that Lamport-Diffie one-time signatures [Lam79] are existentially unforgeable under adaptive chosen message attacks (i.e., provide EUF-CMA security), if the chosen one-way function is preimage resistant. The Merkle signature scheme is also CMA-secure if the underlying OTS scheme is CMA-secure and if the underlying hash function is collision resistant. For increased efficiency and shorter signatures we choose Winternitz OTS rather than the classic Lamport-Diffie OTS. The security of the Winternitz one-time signatures is discussed in [DSS05, BDE+11, Hül13]. The findings in [BDE+11] and [Hül13] show that Winternitz OTS are CMA-secure if used with pseudo-random functions or collision-resistant, undetectable one-way functions, respectively. The level of equivalent symmetric security lost by using a small Winternitz-parameter w is in both cases rather small. In our case, the biggest Winternitz parameter is $w = 4$, hence we still provide a security level of approx. 95 bits for a 128-bit PRF or 116 bits for W-OTS+ [Hül13]). Related discussions for a similar MSS scheme can also be found in [BDH11].

Another important aspect is that most hash-based signature schemes are stateful, i.e., it is required to maintain persistent information about which of the OTS keys were already used. When signing a message, it is crucial that the signer ensures not to reuse a one-time signing key. While in a mathematical description this does not pose a problem, in practice there are a few obstacles to overcome. Commonly the state maintenance would be realized by an always increasing index stored in non-volatile memory which points to the first unused one-time signing key. The index should be incremented prior to issuing a signature such that an index increment cannot be skipped by fault injection attacks after signature generation. State maintenance becomes more difficult in case multiple parties should be able to sign messages with the same signing key, in case of restoring key backups, virtual machine images, and so forth. A possible solution proposed by Goldreich are Merkle trees with a huge number of leaves in the range of the security parameter [Gol03]. The index can then be chosen at random or could be derived from a hash of the message that is signed. Efficiently handling such large Merkle trees can be achieved using the technique of tree-chaining as proposed for example in GMSS [BDK+07]. Extending on these ideas [BHH+15] recently introduced the SPHINCS stateless hash-based signature scheme which uses a hyper-tree structure that combines the approaches of Merkle and Goldreich. Instead of one-time signatures so called few-time signatures are used to reduce the total tree height. Further state management techniques for different scenarios, e.g., a hybrid stateless/stateful approach, are presented and analyzed in [MKF+16].

Chapter 10

Faster Hash-Based Signatures with Bounded Leakage

Digital signatures have become a key component of many embedded system solutions and are facing strong security and efficiency requirements. At the same time side-channel resistance is essential for a signature scheme to be accepted in real-world applications. Based on the Merkle signature scheme and Winternitz one-time signatures we propose a quantum-resistant signature scheme with bounded side-channel leakage. Novel algorithmic improvements for the authentication path computation reduce the average signature computation time by nearly 50 % when compared to state-of-the-art algorithms. Furthermore, our improvements tightly bound side-channel leakage and we state the exact number of times each key is used.
The proposed scheme is implemented on two platforms, an Intel Core i7 CPU and an AVR ATxmega microcontroller, with carefully optimized versions for the respective target platform. The theoretical algorithmic improvements are verified in both implementations using cryptographic hardware accelerators to achieve high performance.

This research was presented at SAC'13 and appeared in the book Number Theory and Cryptography [EvMPY13, EvMY14]. It is joined work with Thomas Eisenbarth, Xin Ye, and Christof Paar.

Contents

10.1 Introduction

Side-channel attacks are considered a serious threat for cryptographic implementations in embedded devices. Adding effective protection against attacks such as power analysis or EM analysis is costly in terms of space and computation time. Hence, side-channel resistant public-key engines are often just too bulky for wide-spread adoption. Exploring public-key digital signature schemes that are efficient on embedded platforms and offer inherent side-channel resistance can be a superior alternative to the prevailing choices of (EC-)DSA and RSA.

New research directions in theoretical cryptography, namely *leakage resilient* cryptographic schemes, suggest that performing cryptographic algorithms in a different way might make them inherently resistant against side-channel attacks without the need of further implementational countermeasures. Instead of protecting a key that is used over and over again, these schemes limit the leakage that an attacker can observe for a given key (or state) by limiting the number of accesses to it. The groundbreaking work of Faust et al. [FKPR10] shows a scheme that provides a selectable number of leakage resilient signatures. The approach builds on a signature scheme that only leaks an admissible amount of information when executed up to three times. The scheme does not explicitly propose or recommend an underlying signature scheme. But when instantiated with one of the prevailing signature schemes, the leakage resilient signature engine becomes practically infeasible: each generated leakage-resilient signature requires three signature generations and two key generations of the underlying signature scheme.

Prior work by Rohde et al. [RED+08] as well as by Hülsing et al. [HBB13] suggest that the Merkle Signature Scheme (MSS) in combination with Winternitz One-Time Signatures (W-OTS) [DSS05] is a possible choice for a time-limited signature scheme and can be efficiently implemented in embedded systems. We analyze and extend the proposal by Rohde et al. and propose several modifications that lead to significant performance improvements and bounded side-channel leakage. A key component of the analyzed MSS engine is the PRNG which is used to generate the private signing key. The PRNG is a self-contained component and is desired to be leakage resilient. Another building block for the one-time signatures is a one-way function that needs to have bounded leakage. Other parts of the engine, e.g., a collision resistant hash function needed for the Merkle tree, only process public data and are thus leakage-agnostic.

Contribution Compared to the state-of-the-art, our proposed scheme provides bounded leakage at comparable cost to an *unprotected* ECC engine. We implement the proposed signature scheme on two wide-spread platforms: an Intel Core i7 CPU and a low-cost AVR 8-bit microcontroller. We target a security level of 80-bit and make use of available cryptographic hardware accelerators to gain maximum efficiency. In addition, we evaluate existing algorithms to compute the authentication path of the Merkle signatures and propose improvements that balance the number of times each leaf is computed and thus limit side-channel leakage. These improvements also halve the average computation time required to compute the authentication path. We quantify how often each leaf is computed, show that previous algorithms have a strong bias in their leaf computations, and explain how we distribute and balance the load across all leaves.

Outline This work is outlined as follows. We start by investigating bounded leakage features of our MSS design in Section 10.2. The optimized authentication path computation which

balances computations across all leaves is presented in Section 10.3. Implementation details and a leakage analysis are given in Section 10.4. We draw a conclusion in Section 10.5.

10.2 Bounded Leakage for MSS

The presented design has several features that bound leakage of secret information. First, the design consists of many one-time signatures with *independent* keys. This means there is no key reuse, and leakage of one OTS key does not reveal information about the other keys. Major parts of the performed computations are in the Merkle tree. Since the Merkle tree is public, computations within the tree do not leak any secret information. Hence, leakage of g is not an issue.

Secret information is only processed during *signing* and *key generation*. Key generation usually takes place in a secure environment, as key generation is usually too costly to be performed on the embedded system. However, even if key generation leaks, it is a single sequence of leakage for all parts of the key, i.e., all one-time keys leak exactly once. Critical information leakage can only happen during signing. If all OTS keys would be stored, they could be chosen independently and would leak exactly once, when used for signing (assuming that *only computation leaks information* [MR04]). In this case, an adversary would get, at most, two observations per key (one during key generation and one at signing), outperforming the scheme described in [FKPR10]. However, as described in Section 9.4, the OTS keys are generated on-the-fly using a PRNG to achieve a scheme suited for embedded devices. In this case each signing operation consists of three steps: (i) performing an OTS, (ii) updating the state which requires recomputation of verification keys, and (iii) computing the authentication path. Since the Merkle tree is public, no secret information is revealed during authentication path computation. The OTS itself only leaks information about the current OTS key, i.e., one additional leakage for each key. The main leakage occurs during the state updates, which result in repeated execution of the PRNG and recomputation of verification keys that leak information about the corresponding OTS key.

Each PRNG update reveals information about one OTS key and the internal state of the PRNG. As the described scheme generates several one-time keys more than once, the PRNG can be executed l times on the *same* input, where l is determined by the parameters of the BDS algorithm. That is, each $\textsc{Seed}_i$ has up to l leakages as PRNG input. The OTS keys x_i are derived from an initial seed $\textsc{Seed}_{\text{W-OTS}_i}$ by the same PRNG. The x_i serve as input for the one-way function f. That is, each $\textsc{Seed}_{\text{W-OTS}_i}$ has up to l leakages as input to PRNG; each x_i is either known by the adversary as part of the signature or has up to l leakages as input of f during verification key recomputation and signing.

10.3 Optimized Authentication Path Computation

Since the Merkle tree is not stored, the parts of the Merkle tree needed for the authentication path must be generated. One optimized algorithm for this purpose is the BDS algorithm [BDS09]. Its goal is to minimize costly leaf computations. However, to minimize the

leakage, it is also important to balance leaf computations. In the following we describe further optimizations that reduce the number of computations for each individual leaf, thereby minimizing the maximum leakage per private-key computation. We furthermore reduce the overall computation time by nearly 50%, at the cost of a slightly increased memory usage.

10.3.1 Authentication Path Computation

The authentication path consists of nodes of the Merkle tree. For the computation of upcoming authentication nodes we use several stacks of nodes for different heights of the tree. Treehash instances $\textsc{Treehash}_h$ are used for heights $0 \leq h \leq H-K-1$. Each instance is initialized with a leaf index s and is updated in Algorithm 4 until the required authentication node is computed. During a treehash update the next leaf is created and parent nodes are computed by hashing previously created nodes if possible. Authentication nodes change every 2^h steps for height h and if an authentication node is used from a treehash instance, this instance is re-initialized to compute the following authentication node for that height.

Preliminaries

The total number of leaf computations that occur during execution of Algorithm 4 can be calculated by counting all invocations of Leafcalc, a function that on input s outputs leaf $\nu_0[s]$. As mentioned in [BDS09] it is possible to omit Leafcalc in Step 3 of Algorithm 4 since the s-th W-OTS key pair is used to sign the current message, hence the verification key can be computed from the signature and one additional hash computation yields leaf $\nu_0[s]$. If a different OTSS is used, the verification key is part of the OTS, and can be hashed to create $\nu_0[s]$. This saves 2^{H-1} Leafcalc invocations. Careful analysis of Algorithm 4 leads to the total number of leaf computations in the BDS algorithm

$$N_{H,K_{\text{total}}} = \sum_{h=0}^{H-K-1} \left(2^{H-1} - 2^{h+1}\right) = (H-K)\,2^{H-1} - 2^{H-K+1} + 2.$$

To count how often a specific leaf s is computed during Algorithm 4 we have to consider all occurrences of s as parameter of Leafcalc, except for when s is a left leaf (Step 3, Algorithm 4), as explained above. To determine if leaf s is computed in treehash instance $\textsc{Treehash}_h$ we make the following observation: $\textsc{Treehash}_0$ computes leaves $(5), (7), (9), \ldots$, $\textsc{Treehash}_1$ computes leaves $(10, 11), (14, 15), \ldots$, $\textsc{Treehash}_2$ computes leaves $(20, 21, 22, 23), (28, 29, 30, 31), \ldots$ and so forth. Hence, the total number of computations for leaf s is given by

$$N_{H,K}(s) = \sum_{h=0}^{H-K-1} \left\lfloor \frac{s \bmod 2^{h+1}}{2^h} \right\rfloor \cdot \left\lceil \frac{\left\lfloor \frac{s}{5 \cdot 2^h} \right\rfloor}{2^H} \right\rceil.$$

Drawbacks

A drawback of the BDS algorithm (Algorithm 4) is that it does not balance the computation of leaf nodes. Some leaves are generated multiple times, others are only computed once. On

average each leaf of the Merkle tree is computed $\overline{N_{H,K}} = N_{H,K_{\text{total}}}/2^H \approx \frac{1}{2}(H-K)$ times. However, the computations per leaf deviate from the average as shown in Figure 10.1 for a Merkle tree with 1024 leaves ($H = 10, K = 2$).

10.3.2 Balanced Authentication Path Computation

Since the rightmost nodes of each treehash instance are calculated most frequently, we propose to cache and reuse them for balancing the leaf computations. We use an array RIGHTNODES to store those nodes. Note, the root of each treehash instance and the complete treehash instance TREEHASH_0 are not stored since lower treehash instances do not require those nodes. Besides reducing the side-channel leakage for heavy duty leaves, this also leads to a significantly reduced computation time, at the cost of an increased memory consumption.

In general, h nodes $\nu_j[2^{2+h-j}-1]$, $j = 0, \ldots, h-1$ are stored for each instance TREEHASH_h, $1 \leq h \leq H-K-1$ (TREEHASH_1: node $\nu_0[7]$, TREEHASH_2: nodes $\nu_1[7], \nu_0[15]$, etc.). The required storage space is

$$S_{\text{RightNodes}}(H,K) = \sum_{h=1}^{H-K-1} h = \binom{H-K}{2} = \triangle_{H-K-1}.$$

Table 10.1 lists the required memory to store right nodes for common parameter sets. The RIGHTNODES array is initialized when computing the root of the Merkle tree. Algorithm 5 formalizes the adjusted initial setup.

Table 10.1: Storage space required by the RIGHTNODES array where the rightmost nodes of each treehash instance TREEHASH_h, $h = 1, \ldots, H-K-1$ are stored for reusage by lower treehash instances.

$H-K$	$\triangle_{H-K-1}$	128-bit digest	160-bit digest	256-bit digest
6	15	240 bytes	300 bytes	480 bytes
8	28	448 bytes	560 bytes	896 bytes
10	45	720 bytes	900 bytes	1,440 bytes
12	66	1,056 bytes	1,320 bytes	2,112 bytes
14	91	1,456 bytes	1,820 bytes	2,912 bytes
16	120	1,920 bytes	2,400 bytes	3,840 bytes
18	153	2,448 bytes	3,060 bytes	4,896 bytes

The treehash instances are updated in case they are initialized but not yet finished (Step 5, Algorithm 4). Each update computes one new leaf. If possible, higher nodes are generated by hashing concatenated nodes from the stack. The rightmost leaf of a treehash instance is computed when the instance receives its last update before finishing. Afterwards, all consecutive rightmost nodes of this instance are generated. If the leaf index $s \equiv 2^h - 1 \mod 2^h$ in instance TREEHASH_h, we store the following nodes in the RIGHTNODES array starting from offset $h(h-1)/2$. Algorithm 6 presents our treehash update algorithm.

The RIGHTNODES array holds the authentication node for every other treehash instance TREEHASH_h, $h = 0, \ldots, H-K-2$ since it was already computed by instance TREEHASH_{h+1}.

Algorithm 5 Key Generation and Initial Setup for the Improved Traversal Algorithm.

Input: H, K

Output: Public-key $\nu_H[0]$, Authentication path, RIGHTNODES array, TREEHASH stacks, RETAIN stacks

1: **Public-Key**
Calculate and publish tree root, $\nu_H[0]$.

2: **Initial Right Nodes**
$i = 0$
for $h = 1$ **to** $H - K - 1$ **do**
for $j = 0$ **to** $h - 1$ **do**
Set RIGHTNODES$[i] = \nu_j \left[2^{2+h-j} - 1\right]$.
$i = i + 1$

3: **Initial Authentication Nodes**
for each $h \in \{0, 1, \ldots, H-1\}$ **do**
Set AUTH$_h = \nu_h[1]$.

4: **Initial Treehash Stacks**
for each $h \in \{0, 1, \ldots, H-K-1\}$ **do**
Setup TREEHASH$_h$ stack with $\nu_h[3]$.

5: **Initial Retain Stacks**
for each $h \in \{H-K, \ldots, H-2\}$ **do**
for each $j \in \left\{2^{H-h-1}, \ldots, 0\right\}$ **do**
RETAIN$_h$.push($\nu_h[2j+3]$).

Algorithm 6 Improved Treehash Update

Input: Height h, current index s, RIGHTNODES array

Output: Updated RIGHTNODES array, updated Treehash instance TREEHASH$_h$

Compute the s-th leaf: NODE$_1$ ←LEAFCALC(s)
if $s \equiv 2^h - 1 \left(\mod 2^h\right)$ **and** NODE$_1$.height() $< h$ **then**
offset $= h(h-1)/2$
RIGHTNODES[offset] ←NODE$_1$
end if
while NODE$_1$ has the same height as the top node on TREEHASH$_h$ **do**
Pop the top node from the stack: NODE$_2$ ←TREEHASH$_h$.*pop*()
Compute their parent node: NODE$_1 \leftarrow g($NODE$_2 \| $NODE$_1)$
if $s \equiv 2^h - 1 \left(\mod 2^h\right)$ **then**
offset = offset + 1
RIGHTNODES[offset] ←NODE$_1$
end if
end while
Push the parent node on the stack: TREEHASH$_h$.*push*(NODE$_1$)

Hence, it is possible to copy the authentication node from the RIGHTNODES array instead of computing it during every other initialization of treehash instance TREEHASH_h. The authentication node can be copied from the RIGHTNODES array if $s+1 \equiv 0 \mod 2^{h+2}$ and if $s+1 \equiv 2^{h+1} \mod 2^{h+2}$ the authentication node has to be computed. If nodes can be reused, the authentication node (root of TREEHASH_h) is copied from the RIGHTNODES array along with its rightmost child nodes. This way we can reuse them in instances TREEHASH_j, $j < h$. This improvement can be easily integrated into the BDS algorithm by modifying Step 4c accordingly.

Comparison

In order to quantify our improvements, we count the overall number of leaf computations and develop formulas with which we can count how often a specific leaf s is computed. As before, 2^h leaves are computed by each instance TREEHASH_h but since we are able to copy the authentication node from the RIGHTNODES array during every other re-initialization for treehash instances TREEHASH_h, $h = 0, \ldots, H-K-2$, half of the LEAFCALC computations can be omitted and only $2^{H-h-2}-1$ re-initializations are required. Hence, these treehash instances make $2^{H-2}-2^h$ calls to LEAFCALC each. The exception is the treehash instance TREEHASH_{H-K-1} because it cannot copy nodes from the RIGHTNODES array. This is due to the fact that it is the highest instance and its nodes have not been computed before by any other treehash instance. Thus, this instance remains unchanged and makes $2^{H-1}-2^{H-K}$ calls to LEAFCALC. Overall, the total number of leaf computations is

$$\begin{aligned} N'_{H,K_{\text{total}}} &= \sum_{h=0}^{H-K-2} \left(2^{H-2}-2^h\right) + 2^{H-1} - 2^{H-K} \\ &= (H-K+1)\,2^{H-2} - 3 \cdot 2^{H-K-1} + 1. \end{aligned}$$

This is nearly a 50% reduction compared to $N_{H,K_{\text{total}}}$ of the BDS algorithm.

The number of leaf computations for a specific leaf s in the improved algorithm depends on whether s is a left leaf or a right leaf. If s is even, it is a left leaf and can be computed from the current one-time signature or verification key as mentioned in Section 10.3.1 for Step 3 of Algorithm 4. If s is odd, it is a right leaf and thus LEAFCALC is not executed directly. To determine if s is computed in treehash instance TREEHASH_h, $h = 0, \ldots, H-K-2$, we have to consider that s is copied instead of being computed during every other initialization. We construct function $\delta'_{H,K}(s)$ that returns the number of times leaf s is computed in treehash instances $\text{TREEHASH}_h, h = 0, \ldots, H-K-2$.

$$\delta'_{H,K}(s) = \sum_{h=0}^{H-K-2} \left\lfloor \frac{s \bmod 2^{h+1}}{2^h} \right\rfloor \cdot \left\lceil \frac{\left\lfloor \frac{s}{5 \cdot 2^h} \right\rfloor}{2^H} \right\rceil \cdot \left(1 - \left\lfloor \frac{s \bmod 2^{h+2}}{2^{h+1}} \right\rfloor\right)$$

Since the highest treehash instance TREEHASH_{H-K-1} cannot copy nodes from the RIGHTNODES array, we count the number of computations for this instance as for the BDS algorithm by evaluating $\delta_{H,K}(s, H-K-1)$ for leaf s. Overall, leaf s is generated

$$N'_{H,K}(s) = \left\lfloor \frac{s \bmod 2^{H-K}}{2^{H-K-1}} \right\rfloor \cdot \left\lceil \frac{\left\lfloor \frac{s}{5 \cdot 2^{H-K-1}} \right\rfloor}{2^H} \right\rceil + \delta'_{H,K}(s)$$

times during the computation of all authentication nodes. On average each leaf is now computed $\overline{N'_{H,K}} = N'_{H,K_{\text{total}}}/2^H \approx \frac{1}{4}(H-K+1)$ times. The reduced number of computations for each leaf is shown in Figure 10.2. Visual comparison between Figure 10.1 and Figure 10.2 gives an intuition of the reduction and balancing of leaf computations. For further comparisons see Figure 10.3.

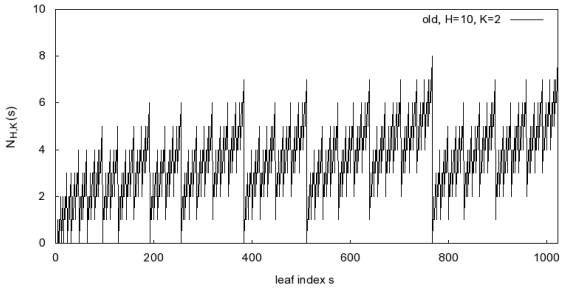


Figure 10.1: Number of times each leaf is computed by the original BDS algorithm for a Merkle tree of height $H = 10$ and $K = 2$.

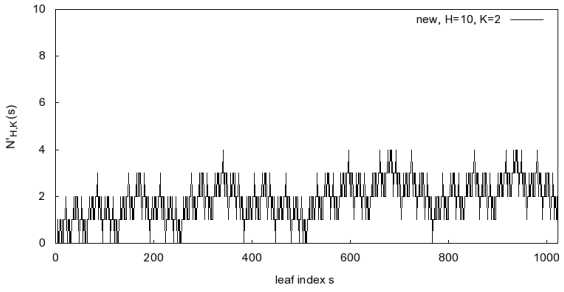


Figure 10.2: Number of times each leaf is computed by our variation for a Merkle tree of height $H = 10$ and $K = 2$.

We compare the overall, average and worst-case number of leaf computations in Table 10.2 for common parameters sets (H, K). The total number of leaf computations as well as the average computations per leaf are decreased by about $38 - 48\%$ for the chosen parameters of H and K. Both the worst-case computation time as well as the average signature computation time are decreased. For example, battery-powered devices benefit from a reduced computation time, which directly relates to the overall power consumption.

Table 10.2: Comparison of the required computations for a Merkle tree with common parameter sets (H, K). We also list the average and worst-case number of leaf computations $\overline{N_{H,K}}$ and $\overline{N'_{H,K}}$, as well as the variance $\sigma^2_{H,K}$ and $\sigma'^2_{H,K}$ of $N_{H,K}(s)$ and $N'_{H,K}(s)$.

H	K	$N_{H,K_{\text{total}}}$	$N'_{H,K_{\text{total}}}$	$\overline{N_{H,K}}$	$\overline{N'_{H,K}}$	%	$\sigma^2_{H,K}$	$\sigma'^2_{H,K}$	%	max. $N_{H,K}(s)$	max. $N'_{H,K}(s)$	%
10	2	3,586	1,921	3.50	1.88	46.4	2.24	0.73	67.3	8	4	50.0
10	4	2,946	1,697	2.88	1.66	42.4	1.60	0.50	68.5	6	3	50.0
10	6	2,018	1,257	1.97	1.23	37.7	1.02	0.33	67.9	4	2	50.0
16	2	425,986	221,185	6.50	3.38	48.1	3.75	1.11	70.4	14	7	50.0
16	4	385,026	206,849	5.88	3.16	46.3	3.11	0.88	71.6	12	6	50.0
16	6	325,634	178,689	4.97	2.73	45.1	2.53	0.71	72.1	10	5	50.0
20	2	8,912,898	4,587,521	8.50	4.38	48.5	4.75	1.36	71.4	18	9	50.0
20	4	8,257,538	4,358,145	7.88	4.16	47.2	4.11	1.13	72.5	16	8	50.0
20	6	7,307,266	3,907,585	6.97	3.73	46.5	3.53	0.96	72.9	14	7	50.0

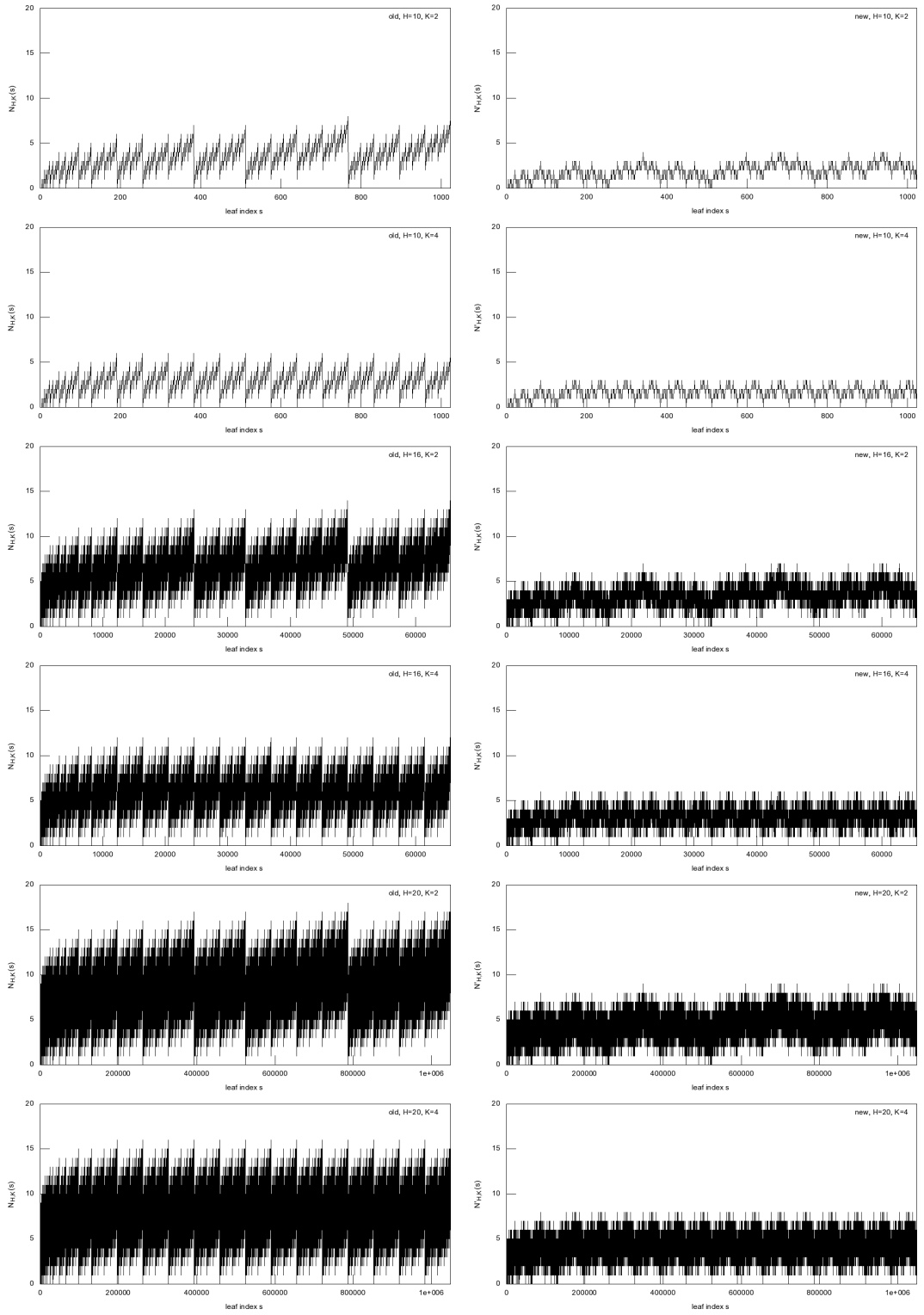


Figure 10.3: Comparison of $N_{H,K}(s)$ (on the left) and $N'_{H,K}(s)$ (on the right) for $H = \{10, 16, 20\}$ and $K = \{2, 4\}$ for all leaves s of the respective tree.

Since all but the topmost treehash instances need to be computed only every second time, the number of updates per signature (Step 5, Algorithm 4) can be reduced from $\lceil (H-K)/2 \rceil$ to $\lceil (H-K+1)/4 \rceil$. As a result, the average update time is much better balanced than in Algorithm 4 and the worst case computation time is also improved. The BDS algorithm needs to store $3H + \lfloor H/2 \rfloor - 3K + 2^K - 2$ tree nodes and $2(H-K)+1$ PRNG seeds as signature keys. Due to storing the rightmost nodes our improved algorithm increases the number of tree nodes that have to be stored by $\binom{H-K}{2}$. Even if the additional memory is used to increase K for the original BDS algorithm, the speedup is still significant. E.g., our algorithm with $(H, K) = (16, 4)$ and BDS with $(16, 6)$ have comparable storage requirements, but our algorithm still achieves a speedup of 36% over BDS. The verification key and signature sizes remain unaffected: the verification key size is m and the signature size remains at $t \cdot n + H \cdot m$.

10.4 Implementation Details and Leakage Analysis

We describe our choices for the cryptographic primitives which is used to implement the proposed signature scheme described in Sections 9.2 and 10.3. We detail on the target platforms and give performance figures for key and signature generation as well as signature verification.

10.4.1 A Bounded Leakage Merkle Signature Engine

We implement two versions with different hash functions g for the Merkle tree. Both versions use AES-128 in an MJH construction [LS11]. Using AES-128 as the block cipher is favorable from a performance perspective as existing AES co-processors can be used. MJH is collision resistant for up to $\mathcal{O}(2^{\frac{2n}{3} - \log n})$ queries when instantiated with a n-bit block cipher. With AES-128 as an ideal cipher this results in 80 bits security [LS11]. On the downside, MJH produces 256-bit hash outputs which in the MSS setting leads to an increased key and signature size. Hence, we also implement a version that shortens the 256-bit output of MJH to 160-bit, resulting in smaller key and signature sizes. This also reduces the number of times the AES engine needs to be used when creating nodes in the Merkle tree. Remember: leakage of g is not an issue since g only processes public information.

One-way function f is implemented based on AES-128 in an MMO [MMO85, MOV01] construction: $f(x_i) := \mathrm{AES}_{\mathrm{IV}}(x_i) \oplus x_i$. Unlike the PRNG, f is keyless. Hence, for independent inputs its leakage is inherently 1-limiting and f can thus be viewed as uniformly seed-preserving. The PRNG defined in Equation (9.1) in Section 9.4 is implemented based on the leakage-2-limiting PRNG proposed in [SPY+10]. In particular,

$$\mathrm{PRNG}(k_i) := (\mathrm{AES}_{k_i}(0^{128}),\ \mathrm{AES}_{k_i}(0^{127}||1)),$$

where AES_{k_i} denotes AES-128 with a 128-bit key k_i, used as seed-preserving function.

Both PRNG and f handle secret inputs. The PRNG processes each $\textsc{Seed}_s$ and $\textsc{Seed}_{\text{W-OTS}_s}$ as well as the x_i for s exactly $N'_{H,K}(s)$ times during state updates and one time during signing OTS_s. We exclude the key generation in this analysis, as it is performed off-chip, presumably in a secure environment. Both PRNG and f rely on AES-128 as cryptographic building block. The PRNG executes AES twice under the same secret-key (i.e., the PRNG is 2-limiting) while f

touches the secret input only once, making the signature engine overall leakage-2-limited. The strongest leakage will be observed for the SEED_i, resulting in a total of $l = 2 \cdot (\max(N'_{H,K}(s)) + 1)$ leakages. These l observations are on 2 different inputs, hence there are $l/2 = \max(N'_{H,K}(s)) + 1$ observations under the same input, i.e., leakage will only differ by noise. Classical side-channel attacks are further mitigated by the fact that intermediate values SEED_i of the key generation PRNG are not output. The adversary will only get access to a limited number of x_i.

10.4.2 Implementation Platforms

We implement the signature scheme on two different platforms: a lightweight and low-cost 8-bit Atmel ATxmega microcontroller and a powerful Intel Core i7 CPU.

Intel Core i7-2620M 64-bit CPU

Intel's off-the-shelf Core i7-2620M 64-bit Sandy Bridge CPU features two cores running at 2.70 GHz (with Turbo Boost technology up to 3.40 GHz). For accurate measurement, we disable Turbo Boost and hyper-threading during our benchmarks. The CPU incorporates recent extensions to the x86 instruction set. An important extension in our context is the AES-NI extension which consists of six additional instructions that improve the performance when en-/decrypting data using AES [Cor10]. All standardized key lengths (128 bits, 192 bits, 256 bits) are supported for a block size of 128 bits.

Atmel AVR ATxmega128A1 8-bit Microcontroller

We are using the Atmel evaluation board AVR XPLAIN that features an ATxmega128A1 microcontroller. The ATxmega offers hardware accelerators for DES and AES and is clocked at 32 MHz. The hardware acceleration is limited to AES with 128-bit key and block sizes. A leakage analysis has been performed on this processor in Section 10.4.4, as it is a typical example for a low-power embedded platform.

10.4.3 Performance Results

In the following we give performance figures of the signature scheme for selected Merkle tree parameters H and K as well as Winternitz parameter w on both platforms.

CPU Performance

On the Intel CPU we measure the time to create the root node of Merkle trees, i.e., the verification key generation. We iterate over all leaves and sign random messages to measure the average computation time that is needed to create a valid MSS signature. Additionally, we measure the time it takes to verify an MSS signature. Signature computation includes creating the signing key, performing a one-time signature with the signing key, and generating the next authentication path. The last step can be precomputed between two signing operations since

Table 10.3: Performance figures of a Merkle tree with parameters $H = 16, K = 2, w = 2$ on an Intel i7 CPU and $H = 10, K = 2, w = 2$ on an ATxmega microcontroller. One-way function f is implemented using a hardware-accelerated AES-128 (AES-NI instructions, ATxmega crypto accelerator) in MMO construction. Hash function g is implemented using AES-128 in an MJH-256 construction and with the output truncated to 160 bits. The Intel CPU runs at 2.7 GHz and the ATxmega at 32 MHz.

Hash g		MJH-256 w/ AES-128			MJH-160 w/ AES-128		
Target		[RED+08]	our	impr.	[RED+08]	our	impr.
Core i7	KeyGen	6546.9 ms	6037.5 ms	8%	4218.7 ms	3,886.3 ms	8%
Core i7	Sign	743.9 us	401.3 us	46%	487.1 us	256.2 us	47%
Core i7	Verify	76.1 us	78.1 us	-3%	50.8 us	49.3 us	3%
AVR	Sign	110.0 ms	64.9 ms	41%	70.7 ms	41.7 ms	41%
AVR	Verify	18.4 ms	18.4 ms	0%	11.0 ms	11.0 ms	0%

it is independent of the signed message. The measurement is done for tree height $H = 16$ with $K = 2$ and $w = 2$. Note, due to the binary tree structure computation of the root node can be parallelized if more than one CPU core is available. This would bring down the required computation time by roughly the factor of used cores.

We compare our results against the originally proposed signature scheme [RED+08] in Table 10.3. Our improved algorithm in combination with the exchanged PRNG yields on average a performance gain of 46-47 % for signature generation compared to the results of [RED+08]. The new PRNG improves the computation time on average by 8%, the algorithmic changes of the authentication path computation yield 38-39% points.

When generating verification keys an 8% improvement can be observed. This is due to the exchanged PRNG which uses a hardware-accelerated AES engine since our algorithmic improvements do not affect key generation. Signature verification is more or less stable, regardless of cipher/algorithm combinations and is about a factor of 5 faster than signature generation.

Microcontroller Performance

On the microcontroller we measure the average computation time that is needed to create a valid MSS signature (including next authentication path computation) and the time it takes to verify an MSS signature. We omit the generation of the verification key since for reasonable tree heights it is an infeasible task for the microcontroller. Verification keys have to be computed once on a more powerful platform when initializing the microcontroller. The code was compiled using `avr-gcc` version 3.3.0. We found optimization stage `-O2` to achieve the best tradeoff between runtime and code size.

The results on the microcontroller are in accordance with the results observed on the Intel CPU. The average signature generation time improves by 41 % when using our proposed changes. Signature verification remains stable and is four times faster than signature generation. The memory consumption is listed in Table 10.4. Compared to the setting of [RED+08] we need more

Table 10.4: Required memory on the ATxmega128A1 microcontroller. In total 128 Kbytes flash memory and 8 Kbytes SRAM are available on this device. Memory consumption is reported in bytes and includes the verification and signature keys.

		MJH-256 w/ AES-128				MJH-160 w/ AES-128			
		[RED+08]		our		[RED+08]		our	
H	K	Flash	SRAM	Flash	SRAM	Flash	SRAM	Flash	SRAM
10	2	10,608	1,486	12,070	2,382	10,204	1,066	11,352	1,626
10	4	10,726	1,604	11,768	2,084	10,250	1,112	11,138	1,412
10	6	11,994	2,874	12,752	3,066	11,018	1,878	11,726	1,998

Table 10.5: Comparison of signing key (sk), verification key (vk), and signature size (sig) between [RED+08], our improvement, and XMSS+ [HBB13] for common (H, K, w) parameter sets. All sizes are reported in bytes.

			MJH-256 our			MJH-160 our			MJH-256 [RED+08]			MJH-160 [RED+08]			XMSS+ [HBB13]		
H	K	w	sk	vk	sig	sk	vk	sig	sk	vk	sig	sk	vk	sig	sk	vk	sig
16	2	2	5,335	32	2,640	3,547	20	1,680	2,423	32	2,640	1,727	20	1,680	3,760	544	3,476
16	2	4	5,335	32	1,584	3,547	20	1,008	2,423	32	1,584	1,727	20	1,008	3,200	512	1,892
20	4	2	7,049	32	2,768	4,649	20	1,760	3,209	32	2,768	2,249	20	1,760	4,303	608	3,540
20	4	4	7,049	32	1,712	4,649	20	1,088	3,209	32	1,712	2,249	20	1,088	3,744	576	1,956

flash and SRAM memory due to the additional storage for the RIGHTNODES array. Table 10.5 compares key and signature sizes for different MSS implementations. Note that the increased signature sizes of [HBB13] enable on-card key generation.

10.4.4 Leakage Analysis

The AVR ATxmega processors has been analyzed with respect to power analysis in [Kiz09]. The found leakage is weak: the best attack needs more than 3000 measurements on random known inputs for secret-key recovery. However, the applied method is not the most powerful[1].

In order to get a more thorough leakage analysis of the target platform, we performed own side-channel experiments. Since all AES computations with critical leakage are performed by the AES co-processor of the ATxmega processor, we analyzed the leakage of that co-processor. Instead of a correlation based DPA, we applied a (univariate) template attack [CRR03], the de-facto standard for power leakage evaluation [SMY09]. The profiled intermediate state is $\Delta = p_0 \oplus k_0 \oplus p_1 \oplus k_1$, where one template was created for each possible Δ. This is the same intermediate state that was targeted in [Kiz09]. It appears to be the intermediate state with the strongest leakage. Each recovered Δ reveals one byte of key information. The maximum

[1] Targeting the key xor and using correlation attacks are not considered optimal methods of leakage extraction.

observable leakage is that of the 2-limiting PRNG, which is, at most, executed 10 times each on two different inputs for MSS parameters $(H, K) = (20, 2)$. To capture the maximum leakage, the experiment builds univariate templates from 10,000 traces and tests over two groups of 10 traces where each group shares the same input. A total of 5,000 experiments are conducted, resulting in a Guessing Entropy [SMY09] of 85.06 or 6.41 bits for the correct Δ. This means that the adversary still has to test more than 85 hypotheses for that byte on average. The reduction in entropy is hence less than 0.6 bits[2], resulting in well above 100 bits of remaining key entropy when considering univariate side-channel attacks.

An alternative to plain template attacks are algebraic side-channel attacks [RSVC09] which do not require a known input and output and would be more applicable to attack the PRNG in this work. While being able to exploit several leakages during a single execution of AES (close to 1000 in [RSVC09]), these methods are very sensitive to noise and need a much stronger leakage than the one observed here. Often, an almost noise-free Hamming weight leakage is assumed, which is more than 2.5 bits of information on a byte. This kind of information is not provided by the observed leakage of the hardware AES of the ATxmega processor.

Another location of potential leakage is the computation of the Winternitz signature, where the adversary actually gets access to hash outputs and some outputs of the PRNG used to generate the one-time keys. The observed leakage (10 observations for the same single input, same setup as for the PRNG) has a guessing entropy of 99.53, i.e., less than 0.4 bits of information per byte are revealed. Not much prior work on side-channel attacks on one-way functions has been performed which is most likely due to the fact that the adversary gets only single observations of the leakage.

10.5 Conclusion

We presented novel algorithmic improvements for computing the authentication path in MSS that balance leaf computations, accelerate the overall authentication path generation, and reduce side-channel leakage. The proposed improvements have been implemented on two platforms and were compared to previous proposed algorithms showing significant improvements. We gave explicit formulas to quantify the number of leaf computations when using MSS and showed that the leakage of the secret state is bounded throughout the scheme. The leakage analysis of the ATxmega AES engine showed that no significant information can be extracted about the secret state due to the bounded number of executions under the same key.

We stated theoretically achievable performance gains and verified them practically. The algorithmic improvements decrease the required computation time for signature creation in theory as well as in practice. The performance figures show that Merkle signatures are not only practical, but also resource-friendly and fast. Furthermore, the scheme inherently bounds side-channel leakage. As such it can be an advantageous choice for, e.g., digital signature smartcards.

[2]Note that the guessing entropy for a byte with 2^8 equiprobable states is 128, i.e., 7 bits as guessing entropy looks for the expected number of guesses.

Part III

Conclusion

Chapter 11

Conclusion

This chapter concludes the thesis and provides a summary of the presented results. The thesis ends with an overview of further interesting research topics for alternative public-key cryptography, in particular for code-based public-key encryption and for hash-based digital signatures. Future research ideas include further exploration of side-channel and fault-injection attacks, the NIST call for standardization of quantum-resistant cryptography, and the investigation of other hash-based signature schemes, e.g., the recently proposed SPHINCS signature scheme.

Contents

11.1 Conclusion

This thesis provided novel designs of code-based public-key encryption and hash-based digital signatures schemes targeting resource constraint FPGAs and microcontrollers.

McEliece and Niederreiter encryption will likely be the first code-based cryptographic schemes chosen for practical applications. Their good track record of being fundamentally unbroken despite multiple cryptanalytic results over a long period of time inspires confidence in the security of the constructions and the underlying problems. Binary Goppa codes are the conservative choice for the code family upon which the McEliece and Niederreiter schemes are constructed. In this work we investigated the recently proposed MDPC family of codes and their quasi-cyclic variants in the context of code-based cryptography. We belief to have provided convincing incentives for further consideration of QC-MDPC codes as serious competitors to binary Goppa codes in the McEliece and Niederreiter cryptoschemes by demonstrating the efficiency in multiple use-cases on various implementation targets. Our design explorations include low-cost and lightweight implementations, a hybrid encryption scheme providing IND-CCA security, and high-performance hardware accelerators. We provided and evaluated novel optimizations

for hard-decision bit-flipping MDPC decoders and were able to accelerate decoding, decrease the required decoding iterations, and significantly reduce decoding error probabilities. The optimizations apply even beyond cryptographic applications.

Furthermore, we developed side-channel attacks such as timing attacks and power analysis attacks on early FPGA and microcontroller implementations of the QC-MDPC schemes to identify which parts of the implementations have to be hardened against information leakage. Subsequently, hardened microcontroller implementations were proposed to provide constant-time operations and instruction-invariant execution flows. Continuing cryptanalysis of QC-MDPC McEliece and Niederreiter will help the scheme to further increase the confidence of the broad cryptographic community as will be discussed in the following section on future work.

Our work on hash-based signatures presented a combination of the Merkle signature scheme and Winternitz one-time signatures to achieve a quantum-resistant digital signature engine with minimal assumptions and bounded information leakage. Novel algorithmic improvements which balance leaf computations during the authentication path computation in MSS were proposed. We accelerated the overall authentication path generation and verified the reduced side-channel leakage. Our implementations on two target platforms were shown to significantly improve over previously proposed algorithms, and we showed that the leakage of the secret state is bounded throughout the scheme. The algorithmic improvements decrease the required computation time for signature creation in theory as well as in practice. Merkle signatures offer practical performance at low cost, and are among the most promising quantum-resistant digital signatures schemes due to their minimal assumptions.

11.2 Future Work

Although multiple implementations of McEliece and Niederreiter have been proposed over the last years (mostly using binary Goppa codes), hardening against power and electromagnetic analysis and especially against fault attacks still requires further investigations to provide industry grade drop-in replacements of the prevailing public-key encryption schemes RSA and ECC. An interesting question in the context of cryptosystems based on coding theory is their behavior with regard to fault attacks, since errors are inherently detected and corrected when decoding codes.

Cryptography based on QC-MDPC codes still requires more cryptanalytic results to gain further confidence in the constructions. In addition to classical cryptanalysis, e.g., ISD-like attacks (cf. Section 3.4), it appears necessary to analyze in more detail whether specific quantum algorithms can be designed to break (features of) schemes which claim quantum-resistance, although drastically more efficient quantum attacks appear to be unrealistic for McEliece and Niederreiter. The security of the prevailing RSA, ECC, and DH-based schemes disintegrates when applying Shor's quantum algorithm [Sho97], which is not applicable for McEliece and Niederreiter; only Grover's generic quantum algorithm [Gro96] applies with a limited and expected impact.

The handling of decoding errors and their probability reduction is another important research topic for QC-MDPC McEliece and Niederreiter. Our improvements already achieved a significantly lower error probability compared to the original bit-flipping decoder of Gallager [Gal63]

(cf. Section 4.5). However, it would be desirable to achieve a decoding failure probability in the range of the security parameter. The investigation of other decoder improvements, e.g., by using soft-decision decoding or by using the recently proposed worst-case decoder for MDPC codes of Chaulet et al. [CS16], could help achieving this goal. After submission of this thesis, Guo et al. [GJS16] presented a reaction attack on QC-MDPC decryption which observes and exploits decoding errors to recover the secret parity-check matrix. The attack benefits from the rather high error probability of the original Gallager bit-flipping decoder [Gal63] and degrades with a decreasing decoding failure rate.

A remaining open question is whether timing attacks can succeed to recover secret information from the timing variations of MDPC decoders. We already avoid timing variations in our implementations in Chapter 6, as we assume some form of information leakage. However, it would be interesting to investigate such an information leakage and how it could be exploited.

For future work on hash-based signatures it will be interesting to analyze the recently proposed stateless hash-based signature signature scheme SPHINCS [BHH+15] with regard to its suitability for resource constraint devices and to investigate whether similar side-channel leakage limitations from our work can be applied. The SPHINCS scheme provides a virtually unlimited number of signatures and eliminates the need for secure state handling, although [MKF+16] argue that the state handling does not pose a major thread in practice.

The NIST call for standardization of quantum-resistant public-key cryptography will likely encourage further proposals for building alternative public-key schemes and cryptanalysis thereof. McEliece and Niederreiter encryption on the basis of binary Goppa codes will almost certainly enter the competition, and a proposal of QC-MDPC McEliece and Niederreiter would be advisable due to its demonstrated practicality for embedded devices compared to binary Goppa codes. Similarly, we expect hash-based digital signatures to be among the most promising candidates of this competition.

Bibliography

[Adl79] L. Adleman, "A subexponential algorithm for the discrete logarithm problem with applications to cryptography," in *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, 1979, pp. 55–60. [23]

[AF95] Anne Canteaut and Florent Chabaud, "Improvements of the Attacks on Cryptosystems Based on Error-Correcting Codes: Research Report LIENS-95-21," 1995, ftp://ftp.ens.fr/pub/dmi/users/chabaud/CC95.ps. [27]

[AFG+08] D. Augot, M. Finiasz, P. Gaborit, S. Manuel, and N. Sendrier, "SHA-3 Proposal: FSB," 2008, https://www.rocq.inria.fr/secret/CBCrypto/fsbdoc.pdf. [136, 137, 138]

[AFS03] D. Augot, M. Finiasz, and N. Sendrier, "A Fast Provably Secure Cryptographic Hash Function," Cryptology ePrint Archive, Report 2003/230, 2003, https://eprint.iacr.org/2003/230. [137]

[AFS05] D. Augot, M. Finiasz, and N. Sendrier, "A Family of Fast Syndrome Based Cryptographic Hash Functions," in *Progress in Cryptology – Mycrypt 2005*, ser. Lecture Notes in Computer Science, E. Dawson and S. Vaudenay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3715, pp. 64–83. [137]

[AHPT11] R. Avanzi, S. Hoerder, D. Page, and M. Tunstall, "Side-channel attacks on the McEliece and Niederreiter public-key cryptosystems," *Journal of Cryptographic Engineering*, vol. 1, no. 4, pp. 271–281, 2011. [68, 90]

[AL96] N. Alon and M. Luby, "A linear time erasure-resilient code with nearly optimal recovery," *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 1732–1736, 1996. [17]

[AM89] C. M. Adams and H. Meijer, "Security-related comments regarding McEliece's public-key cryptosystem," *IEEE Transactions on Information Theory*, vol. 35, no. 2, pp. 454–455, 1989. [35]

[Atm10] Atmel, "Atmel AVR1924: XMEGA A1 Xplained Hardware User Guide," 2010, http://www.atmel.com/Images/AVR1924.zip. [94]

[Bac94] P. Bachmann, *Die analytische Zahlentheorie*, ser. Zahlentheorie. New York: Johnson, 1894, vol. Versuch einer Gesamtdarstellung dieser Wissenschaft in ihren Haupttheilen / dargest. von Paul Bachmann ; Theil 2. [35]

[Bac84] E. Bach, *Discrete logarithms and factoring*, ser. Report. Berkeley: Computer Science Division, University of California, 1984, vol. no. UCB/CSD-84-186. [1]

[BBMR14] F. P. Biasi, Barreto, Paulo S. L. M., R. Misoczki, and W. V. Ruggiero, "Scaling efficient code-based cryptosystems for embedded platforms," *Journal of Cryptographic Engineering*, vol. 4, no. 2, pp. 123–134, 2014. [3, 31, 34, 90, 103, 107, 110, 111, 112, 131]

[BCP97] W. BOSMA, J. CANNON, and C. PLAYOUST, "The Magma Algebra System I: The User Language," *Journal of Symbolic Computation*, vol. 24, no. 3-4, pp. 235–265, 1997. [83, 85]

[BCS13] D. J. Bernstein, T. Chou, and P. Schwabe, "McBits: Fast Constant-Time Code-Based Cryptography," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, ser. Lecture Notes in Computer Science, G. Bertoni and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8086, pp. 250–272. [106, 108, 154]

[BDE+11] J. Buchmann, E. Dahmen, S. Ereth, A. Hülsing, and M. Rückert, "On the Security of the Winternitz One-Time Signature Scheme," in *Progress in Cryptology – AFRICACRYPT 2011*, ser. Lecture Notes in Computer Science, A. Nitaj and D. Pointcheval, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6737, pp. 363–378. [162]

[BDH11] J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, B.-Y. Yang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 7071, pp. 117–129. [154, 162]

[BDJR97] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *38th Annual Symposium on Foundations of Computer Science*, 1997, pp. 394–403. [124]

[BDK+07] J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume, "Merkle Signatures with Virtually Unlimited Signature Capacity," in *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, J. Katz and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4521, pp. 31–45. [155, 162]

[BDL+12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012. [3, 154]

[BDPv11] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche, "The Keccak reference," 2011, http://keccak.noekeon.org/Keccak-reference-3.0.pdf. [136]

[BDS09] J. Buchmann, E. Dahmen, and M. Szydlo, "Hash-based Digital Signature Schemes," in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 35–93. [155, 160, 161, 162, 165, 166, 211]

[BEE+13] J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Pöppelmann, F. Regazzoni, F.-X. Standaert,

G. van Assche, R. van Keer, van Oldeneel tot Oldenzeel, Loïc, and I. von Maurich, "Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, S. Mangard, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7771, pp. 158–172. [4, 148, 208]

[Ber66] E. R. Berlekamp, *Nonbinary BCH decoding*, ser. Institute of Statistics mimeo series. Chapel Hill: University of North Carolina. Dept. of Statistics, 1966, vol. no. 502. [16]

[Ber97] T. Berson, "Failure of the McEliece public-key cryptosystem under message-resend and related-message attack," in *Advances in Cryptology — CRYPTO '97*, ser. Lecture Notes in Computer Science, Kaliski, BurtonS., Jr, Ed. Springer Berlin Heidelberg, 1997, vol. 1294, pp. 213–220. [33]

[BGD+06] J. Buchmann, L. C. C. García, E. Dahmen, M. Döring, and E. Klintsevich, "CMSS – An Improved Merkle Signature Scheme," in *Progress in Cryptology - INDOCRYPT 2006*, ser. Lecture Notes in Computer Science, R. Barua and T. Lange, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4329, pp. 349–363. [155]

[BGJT14] R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé, "A Heuristic Quasi-Polynomial Algorithm for Discrete Logarithm in Finite Fields of Small Characteristic," in *Advances in cryptology– EUROCRYPT 2014*, ser. LNCS sublibrary. SL 4, Security and cryptology, P. Q. Nguyen and E. Oswald, Eds. Heidelberg: Springer, 2014, vol. 8441, pp. 1–16. [2]

[BHH+15] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn, "SPHINCS: Practical Stateless Hash-Based Signatures," in *Advances in Cryptology – EUROCRYPT 2015*, ser. Lecture Notes in Computer Science, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, vol. 9056, pp. 368–397. [154, 162, 181]

[BJMM12] A. Becker, A. Joux, A. May, and A. Meurer, "Decoding Random Binary Linear Codes in $2^{n/20}$: How $1 + 1 = 0$ Improves Information Set Decoding," in *Advances in Cryptology – EUROCRYPT 2012*, ser. Lecture Notes in Computer Science, D. Pointcheval and T. Johansson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7237, pp. 520–536. [33, 34]

[BJPW13] A. Bauer, E. Jaulmes, E. Prouff, and J. Wild, "Horizontal and Vertical Side-Channel Attacks against Secure RSA Implementations," in *Topics in Cryptology – CT-RSA 2013*, ser. Lecture Notes in Computer Science, E. Dawson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7779, pp. 1–17. [69]

[BLP08] D. J. Bernstein, T. Lange, and C. Peters, "Attacking and Defending the McEliece Cryptosystem," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, J. Buchmann and J. Ding, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5299, pp. 31–46. [33, 36, 207]

[BLP11] D. J. Bernstein, T. Lange, and C. Peters, "Smaller Decoding Exponents: Ball-Collision Decoding," in *Advances in Cryptology – CRYPTO 2011*, ser. Lecture

Notes in Computer Science, P. Rogaway, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6841, pp. 743–760. [33, 35, 36, 207]

[BLPS11] D. J. Bernstein, T. Lange, C. Peters, and P. Schwabe, "Really Fast Syndrome-Based Hashing," in *Progress in Cryptology – AFRICACRYPT 2011*, ser. Lecture Notes in Computer Science, A. Nitaj and D. Pointcheval, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6737, pp. 134–152. [136, 138, 143]

[BMv78] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems (Corresp.)," *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978. [25, 33, 38, 114]

[BRC60] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960. [16]

[BS08] B. Biswas and N. Sendrier, "McEliece Cryptosystem Implementation: Theory and Practice," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, J. Buchmann and J. Ding, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5299, pp. 47–62. [36, 106, 207]

[CEvMS15] C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Differential Power Analysis of a McEliece Cryptosystem," in *Applied cryptography and network security*, ser. LNCS sublibrary. SL 4, Security and cryptology, T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, Eds. Cham: Springer, 2015, vol. 9092, pp. 538–556. [4, 51, 68, 90, 98]

[CEvMS16a] C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Horizontal and Vertical Side Channel Analysis of a McEliece Cryptosystem," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1093–1105, 2016. [4, 51, 68, 90]

[CEvMS16b] C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Masking Large Keys in Hardware: A Masked Implementation of McEliece," in *Selected areas in cryptography - SAC 2015*, ser. Lecture Notes in Computer Science, O. Dunkelman and L. Keliher, Eds. Springer, 2016, vol. 9566, pp. 293–309. [4, 51, 68, 86, 87, 90]

[CFS01] N. T. Courtois, M. Finiasz, and N. Sendrier, "How to Achieve a McEliece-Based Digital Signature Scheme," in *Advances in Cryptology — ASIACRYPT 2001*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, and C. Boyd, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 2248, pp. 157–174. [154]

[Cho16] T. Chou, "QcBits: Constant-Time Small-Key Code-Based Cryptography," in *Cryptographic Hardware and Embedded Systems – CHES 2016*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, vol. 9813, pp. 280–300. [131, 132]

[CHP12] P.-L. Cayrel, G. Hoffmann, and E. Persichetti, "Efficient Implementation of a CCA2-Secure Variant of McEliece Using Generalized Srivastava Codes," in *Public Key Cryptography – PKC 2012*, ser. Lecture Notes in Computer Science, M. Fis-

chlin, J. Buchmann, and M. Manulis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7293, pp. 138–155. [103, 107, 131, 132]

[CJ04] J.-S. Coron and A. Joux, "Cryptanalysis of a Provably Secure Cryptographic Hash Function," Cryptology ePrint Archive, Report 2004/013, 2004, https://eprint.iacr.org/2004/013. [137]

[CMNS14] P.-L. Cayrel, M. Meziani, O. Ndiaye, and Q. Santos, "Efficient Software Implementations of Code-based Hash Functions and Stream-Ciphers," *WAIFI 2014*, 2014. [137, 138]

[Coh93] H. Cohen, *A Course in Computational Algebraic Number Theory*, ser. Graduate Texts in Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, vol. 138. [23, 24]

[Cor10] I. Corporation, "Advanced Encryption Standard (AES) New Instructions Set White Paper," 2010, http://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. [173]

[CRR03] S. Chari, J. R. Rao, and P. Rohatgi, "Template Attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. Lecture Notes in Computer Science, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, vol. 2523, pp. 13–28. [175]

[CS98] A. Canteaut and N. Sendrier, "Cryptanalysis of the Original McEliece Cryptosystem," in *Advances in Cryptology — ASIACRYPT'98*, ser. Lecture Notes in Computer Science, K. Ohta and D. Pei, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, vol. 1514, pp. 187–199. [33, 34]

[CS03] R. Cramer and V. Shoup, "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack," *SIAM Journal on Computing*, vol. 33, no. 1, pp. 167–226, 2003. [119, 120, 121, 122, 124]

[CS16] J. Chaulet and N. Sendrier, "Worst case QC-MDPC decoder for McEliece cryptosystem," in *IEEE International Symposium on Information Theory (ISIT)*, IEEE, Ed. IEEE, 2016, pp. 1366–1370. [181]

[Dam90] I. B. Damgård, "A Design Principle for Hash Functions," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, ser. Lecture Notes in Computer Science, G. Brassard, Ed. New York, NY: Springer New York, 1990, vol. 435, pp. 416–427. [137, 138]

[DB14] N. S. Dattani and N. Bryans, "Quantum factorization of 56153 with only 4 qubits," *CoRR*, vol. abs/1411.6758, 2014. [2]

[DH76] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. [22]

[DJJ+06] V. S. Dimitrov, K. U. Järvinen, M. J. Jacobson, W. F. Chan, and Z. Huang, "FPGA Implementation of Point Multiplication on Koblitz Curves Using Kleinian Integers," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4249, pp. 445–459. [58, 59]

[DSS05] C. Dods, N. P. Smart, and M. Stam, "Hash Based Digital Signature Schemes," in *Cryptography and Coding*, ser. Lecture Notes in Computer Science, N. P. Smart, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3796, pp. 96–115. [158, 162, 164]

[eBA15a] eBACS, "eBACS: ECRYPT Benchmarking of Cryptographic Systems," 2015, http://bench.cr.yp.to/results-encrypt.html. [106, 108, 146]

[eBA15b] eBASH, "eBASH: ECRYPT Benchmarking of All Submitted Hashes," 2015, http://bench.cr.yp.to/ebash.html. [137]

[EGHP09] T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar, "MicroEliece: McEliece for Embedded Devices," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5747, pp. 49–64. [52, 66, 67, 103, 107, 131, 132]

[EMS14] M. Eichlseder, F. Mendel, and M. Schläffer, "Branching Heuristics in Differential Collision Search with Applications to SHA-512," Cryptology ePrint Archive: Report 2014/302, 2014, http://eprint.iacr.org/2014/302. [136]

[EOS07] D. Engelbert, R. Overbeck, and A. Schmidt, "A Summary of McEliece-Type Cryptosystems and their Security," *Journal of Mathematical Cryptology*, vol. 1, no. 2, 2007. [33, 35]

[EvMPY13] T. Eisenbarth, I. von Maurich, C. Paar, and X. Ye, "A Performance Boost for Hash-Based Signatures," in *Number Theory and Cryptography*, ser. Lecture Notes in Computer Science, M. Fischlin and S. Katzenbeisser, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8260, pp. 166–182. [4, 163]

[EvMY14] T. Eisenbarth, I. von Maurich, and X. Ye, "Faster Hash-Based Signatures with Bounded Leakage," in *Selected Areas in Cryptography – SAC 2013*, ser. Lecture Notes in Computer Science, T. Lange, K. Lauter, and P. Lisoněk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 8282, pp. 223–243. [4, 163]

[FGS07] M. Finiasz, P. Gaborit, and N. Sendrier, "Improved fast syndrome based cryptographic hash functions," *Proceedings of ECRYPT Hash Workshop*, p. 155, 2007. [137]

[Fin11] M. Finiasz, "Parallel-CFS," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, A. Biryukov, G. Gong, and D. R. Stinson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6544, pp. 159–170. [154]

[FKPR10] S. Faust, E. Kiltz, K. Pietrzak, and G. N. Rothblum, "Leakage-Resilient Signatures," in *Theory of Cryptography*, ser. Lecture Notes in Computer Science, D. Micciancio, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 5978, pp. 343–360. [164, 165]

[FL08] P.-A. Fouque and G. Leurent, "Cryptanalysis of a Hash Function Based on Quasi-cyclic Codes," in *Topics in Cryptology – CT-RSA 2008*, ser. Lecture Notes in Computer Science, T. Malkin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4964, pp. 19–35. [137]

[FO99] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, and M. Wiener, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1666, pp. 537–554. [34]

[FO13] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," *Journal of Cryptology*, vol. 26, no. 1, pp. 80–101, 2013. [34]

[FS87] A. Fiat and A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems," in *Advances in Cryptology — CRYPTO' 86*, ser. Lecture Notes in Computer Science, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, vol. 263, pp. 186–194. [154]

[FS09] M. Finiasz and N. Sendrier, "Security Bounds for the Design of Code-Based Cryptosystems," in *Advances in Cryptology – ASIACRYPT 2009*, ser. Lecture Notes in Computer Science, M. Matsui, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5912, pp. 88–105. [33]

[Gal63] R. Gallager, "Low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1963. [3, 17, 38, 39, 40, 41, 42, 43, 112, 180, 181]

[GCHB12] T. Gyorfi, O. Cret, G. Hanrot, and N. Brisebarre, "High-Throughput Hardware Architecture for the SWIFFT / SWIFFTX Hash Functions," Cryptology ePrint Archive, Report 2012/343, 2012, https://eprint.iacr.org/2012/343. [149, 150, 209]

[GDUV12] S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede, "A Speed Area Optimized Embedded Co-processor for McEliece Cryptosystem," in *2012 IEEE 23rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2012, pp. 102–108. [52, 58, 59, 66, 67]

[GHR+12] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, "Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs," Cryptology ePrint Archive: Report 2012/368, 2012, http://eprint.iacr.org/2012/368. [149, 150, 209]

[Gib95] K. Gibson, "Severely denting the Gabidulin version of the McEliece Public Key Cryptosystem," *Designs, Codes and Cryptography*, vol. 6, no. 1, pp. 37–45, 1995. [34]

[Gib96] K. Gibson, "The Security of the Gabidulin Public Key Cryptosystem," in *Advances in Cryptology — EUROCRYPT '96*, ser. Lecture Notes in Computer Science, U. Maurer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, vol. 1070, pp. 212–223. [34]

[GJS16] Q. Guo, T. Johansson, and P. Stankovski, "A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors," in *Advances in cryptology – ASIACRYPT 2016*, ser. LNCS sublibrary. SL 4, Security and cryptology, J. H. Cheon and T. Takagi, Eds. Berlin, Germany: Springer, 2016, vol. 10031, pp. 789–815. [181]

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest, "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 281–308, 1988. [118]

[Gol03] O. Goldreich, *Foundations of cryptography*, ser. Volume 2, Basic Applications. Cambridge, UK and New York: Cambridge University Press, 2003. [162]

[Gop70] V. D. Goppa, "A New Class of Linear Error Correcting Codes," *Probl. Pered. Inform.*, vol. 6, pp. 24–30, 1970. [17]

[GP08] T. Güneysu and C. Paar, "Ultra High Performance ECC over NIST Primes on Commercial FPGAs," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, ser. Lecture Notes in Computer Science, E. Oswald and P. Rohatgi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5154, pp. 62–78. [58, 59]

[GPT91] E. M. Gabidulin, A. V. Paramonov, and O. V. Tretjakov, "Ideals over a Non-Commutative Ring and their Application in Cryptology," in *Advances in Cryptology — EUROCRYPT '91*, ser. Lecture Notes in Computer Science, D. W. Davies, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, vol. 547, pp. 482–489. [34]

[Gro96] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, G. L. Miller, Ed. New York, NY: ACM Pr, 1996, pp. 212–219. [180]

[HB10] M. N. Hassan and M. Benaissa, "A scalable hardware/software co-design for elliptic curve cryptography on PicoBlaze microcontroller," in *2010 IEEE International Symposium on Circuits and Systems - ISCAS 2010*, 2010, pp. 2111–2114. [67]

[HBB13] A. Hülsing, C. Busold, and J. Buchmann, "Forward Secure Signatures on Smart Cards," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, L. R. Knudsen and H. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7707, pp. 66–80. [154, 164, 175, 209]

[Hei87] R. Heiman, "On the security of cryptosystems based on linear error-correcting codes: M.Sc. Thesis, Feinberg Graduate School, Weitzman Institute of Science, Rehovot," 1987. [33]

[Hel74] H. J. Helgert, "Alternant codes," *Information and Control*, vol. 26, no. 4, pp. 369–380, 1974. [16]

[Hel15a] Helion Technology, "RSA and Modular Exponentiation cores," 2015, http://www.heliontech.com/modexp.htm. [67]

[Hel15b] Helion Technology, "SHA-1, SHA-2 & MD5 Fast Hashing Cores for FPGA (Xilinx, Altera, Microsemi, Lattice) and ASIC," 2015, http://www.heliontech.com/fast_hash.htm. [150]

[Hey11] S. Heyse, "Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, B.-Y. Yang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 7071, pp. 143–162. [103, 107, 131, 132]

[Hey13] S. Heyse, "Post Quantum Cryptography: Implementing Alternative Public Key Schemes on Embedded Devices: Preparing for the Rise of Quantum Computers," 2013, http://www-brs.ub.ruhr-uni-bochum.de/netahtml/HSS/Diss/HeyseStefan/diss.pdf. [25]

[HG12] S. Heyse and T. Güneysu, "Towards One Cycle per Bit Asymmetric Encryption: Code-Based Cryptography on Reconfigurable Hardware," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, ser. Lecture Notes in Computer Science, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7428, pp. 340–355. [52, 58, 59]

[HMP10] S. Heyse, A. Moradi, and C. Paar, "Practical Power Analysis Attacks on Software Implementations of McEliece," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, N. Sendrier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6061, pp. 108–125. [67, 68, 69, 90]

[HP10] W. C. Huffman and V. Pless, *Fundamentals of error-correcting codes*. Cambridge, U.K. and New York: Cambridge University Press, 2010. [11, 40, 41, 42]

[HRB13] A. Hülsing, L. Rausch, and J. Buchmann, "Optimal Parameters for XMSS^{MT}," in *Security Engineering and Intelligence Informatics*, ser. Lecture Notes in Computer Science, A. Cuzzocrea, C. Kittl, D. E. Simos, E. Weippl, and L. Xu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8128, pp. 194–208. [154]

[Hül13] A. Hülsing, "W-OTS+ – Shorter Signatures for Hash-Based Signature Schemes," in *Progress in Cryptology – AFRICACRYPT 2013*, ser. Lecture Notes in Computer Science, A. Youssef, A. Nitaj, and A. E. Hassanien, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7918, pp. 173–188. [3, 154, 162]

[HvMG13] S. Heyse, I. von Maurich, and T. Güneysu, "Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, ser. Lecture Notes in Computer Science, G. Bertoni and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8086, pp. 273–292. [4, 37, 51, 90, 92, 93, 103, 107, 132]

[HZP14] S. Heyse, R. Zimmermann, and C. Paar, "Attacking Code-Based Cryptosystems with Information Set Decoding Using Special-Purpose Hardware," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, M. Mosca, Ed. Springer International Publishing, 2014, vol. 8772, pp. 126–141. [33]

[Int14] Intel, "Intel Digital Random Number Generator (DRNG) - Software Implementation Guide," 2014, https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf. [103]

[Jon12] Jonathan Ness, "Flame malware collision attack explained," 2012, http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx. [136]

[Jou14] A. Joux, "A New Index Calculus Algorithm with Complexity $$L(1/4+o(1))$$ in Small Characteristic," in *Selected Areas in Cryptography – SAC 2013*, ser. Lecture

Notes in Computer Science, T. Lange, K. Lauter, and P. Lisoněk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 8282, pp. 355–379. [2]

[KI01] K. Kobara and H. Imai, "Semantically Secure McEliece Public-Key Cryptosystems -Conversions for McEliece PKC," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, and K. Kim, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, vol. 1992, pp. 19–35. [34, 110]

[Kir11] P. Kirchner, "Improved Generalized Birthday Attack," Cryptology ePrint Archive, Report 2011/377, 2011, http://eprint.iacr.org/2011/377. [139]

[Kiz09] I. Kizhvatov, "Side channel analysis of AVR XMEGA crypto engine," in *Proceedings of the 4th Workshop on Embedded Systems Security*, D. Serpanos and W. Wolf, Eds., 2009, pp. 1–7. [175]

[KJJ99] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, and M. Wiener, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1666, pp. 388–397. [75, 93]

[KJJR11] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011. [69]

[KM15] N. Koblitz and A. J. Menezes, "A Riddle Wrapped in an Enigma," Cryptology ePrint Archive: Report 2015/1018, 2015, https://eprint.iacr.org/2015/1018. [2]

[Knu92] D. E. Knuth, "Two Notes on Notation," *The American Mathematical Monthly*, vol. 99, no. 5, p. 403, 1992. [84]

[Kob87] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, p. 203, 1987. [24]

[KY09] A. A. Kamal and A. M. Youssef, "An FPGA implementation of the NTRUEncrypt cryptosystem," in *2009 International Conference on Microelectronics - ICM*, 2009, pp. 209–212. [58, 59]

[Lam79] L. Lamport, "Constructing Digital Signatures from a One Way Function: Technical Report," 1979, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.228.2958&rep=rep1&type=pdf. [158, 162]

[Lan09] E. Landau, *Handbuch der Lehre von der Verteilung der Primzahlen.* Leipzig [u.a.]: Teubner, 1909. [35]

[LB88] P. J. Lee and E. F. Brickell, "An Observation on the Security of McEliece's Public-Key Cryptosystem," in *Advances in Cryptology — EUROCRYPT '88*, ser. Lecture Notes in Computer Science, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, and C. G. Günther, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, vol. 330, pp. 275–280. [33]

[LDW94] Y. X. Li, R. H. Deng, and X. M. Wang, "On the equivalence of McEliece's and Niederreiter's public-key cryptosystems," *IEEE Transactions on Information Theory*, vol. 40, no. 1, pp. 271–273, 1994. [24]

[Leh74] R. S. Lehman, "Factoring large integers," *Mathematics of Computation*, vol. 28, no. 126, p. 637, 1974. [23]

[Leo88] J. S. Leon, "A probabilistic algorithm for computing minimum weights of large error-correcting codes," *IEEE Transactions on Information Theory*, vol. 34, no. 5, pp. 1354–1359, 1988. [33]

[LGK10] Z. Liu, J. Großschädl, and I. Kizhvatov, "Efficient and Side-Channel Resistant RSA Implementation for 8-bit AVR Microcontrollers," *Workshop on the Security of the Internet of Things-SOCIOT*, 2010. [103, 107]

[LL93] A. K. Lenstra and H. W. Lenstra, *The development of the number field sieve.* Springer Berlin Heidelberg, 1993, vol. 1554. [23]

[LS11] J. Lee and M. Stam, "MJH: A Faster Alternative to MDC-2," in *Topics in Cryptology – CT-RSA 2011*, ser. Lecture Notes in Computer Science, A. Kiayias, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6558, pp. 213–236. [172]

[LS12] G. Landais and N. Sendrier, "Implementing CFS," in *Progress in Cryptology - INDOCRYPT 2012*, ser. Lecture Notes in Computer Science, S. Galbraith and M. Nandi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7668, pp. 474–488. [154]

[LT13] G. Landais and J.-P. Tillich, "An Efficient Attack of a McEliece Cryptosystem Variant Based on Convolutional Codes," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, P. Gaborit, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7932, pp. 102–117. [12]

[LWG14] Z. Liu, E. Wenger, and J. Großschädl, "MoTE-ECC: Energy-Scalable Elliptic Curve Cryptography for Wireless Sensor Networks," in *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, I. Boureanu, P. Owesarski, and S. Vaudenay, Eds. Cham: Springer International Publishing, 2014, vol. 8479, pp. 361–379. [103, 107]

[Mac99] Mackay, David J. C., "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, 1999. [17]

[Mas69] J. Massey, "Shift-register synthesis and BCH decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969. [16]

[Mau94] U. M. Maurer, "Towards the Equivalence of Breaking the Diffie-Hellman Protocol and Computing Discrete Logarithms," in *Advances in Cryptology - CRYPTO '94*, ser. Lecture Notes in Computer Science, Y. G. Desmedt, Ed. Berlin and London: Springer, 1994, vol. 839, pp. 271–281. [23]

[McE78] R. J. McEliece, "A Public-Key Cryptosystem Based on Algebraic Coding Theory," *JPL Deep Space Network Progress Report*, no. 42–44, pp. 114–116, 1978. [2, 5, 21, 24, 25, 33, 35, 36, 207]

[MDCE11] M. Meziani, Ö. Dagdelen, P.-L. Cayrel, and El Yousfi Alaoui, S. M., "S-FSB: An Improved Variant of the FSB Hash Family," in *Information Security and Assurance*, ser. Communications in Computer and Information Science, T.-h. Kim, H. Adeli, R. J. Robles, and M. Balitanas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 200, pp. 132–145. [137]

[Mer90] R. C. Merkle, "A Certified Digital Signature," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, ser. Lecture Notes in Computer Science, G. Brassard, Ed. New York, NY: Springer New York, 1990, vol. 435, pp. 218–238. [3, 154, 155, 156, 211]

[Mil86] V. S. Miller, "Use of Elliptic Curves in Cryptography," in *Advances in cryptology–CRYPTO '85*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed. Berlin and New York: Springer-Verlag, 1986, vol. 218, pp. 417–426. [24]

[Mis14] R. Misoczki, "Two Approaches for Achieving Efficient Code-Based Cryptosystems," 2014, https://tel.archives-ouvertes.fr/file/index/docid/931811/filename/these_archivage_3073292.pdf. [25, 39]

[MKF+16] D. McGrew, P. Kampanakis, S. Fluhrer, S.-L. Gazdag, D. Butin, and J. Buchmann, "State Management for Hash Based Signatures," Cryptology ePrint Archive, Report 2016/357, 2016, http://eprint.iacr.org/2016/357. [162, 181]

[MMO85] S. M. Matyas, C. H. Meyer, and J. Oseas, "Generating strong one-way functions with cryptographic algorithm," *IBM Technical Disclosure Bulletin*, vol. 27, no. 10A, pp. 5658–5659, 1985. [172]

[MMT11] A. May, A. Meurer, and E. Thomae, "Decoding Random Linear Codes in $\tilde{\mathcal{O}}(2^{0.054n})$," in *Advances in Cryptology – ASIACRYPT 2011*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 7073, pp. 107–124. [33]

[MN95] Mackay, David J. C. and R. M. Neal, "Good Codes based on Very Sparse Matrices," *Cryptography and Coding. 5th IMA Conference, Lecture Notes in Computer Science*, vol. 1025, pp. 100–111, 1995. [17]

[MNS13] F. Mendel, T. Nad, and M. Schläffer, "Improving Local Collisions: New Attacks on Reduced SHA-256," in *Advances in Cryptology – EUROCRYPT 2013*, ser. Lecture Notes in Computer Science, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7881, pp. 262–278. [136]

[MOP07] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards*, ser. Advances in information security. New York, N.Y.: Springer, 2007, vol. v. 31. [69, 86]

[MOV01] A. J. Menezes, Oorschot, Paul C. van, and S. A. Vanstone, *Handbook of applied cryptography*, 5th ed., ser. CRC Press series on discrete mathematics and its applications. Boca Raton: CRC Press, 2001. [172]

[MR04] S. Micali and L. Reyzin, "Physically Observable Cryptography," in *Theory of Cryptography*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, and M. Naor, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 2951, pp. 278–296. [165]

[MS86] F. J. MacWilliams and Sloane, N. J. A, *The Theory of Error-Correcting Codes*, 5th ed., ser. North-Holland mathematical library, Amsterdam, North-Holland, 1986, vol. 16. [11]

[MTSB12] R. Misoczki, J.-P. Tillich, N. Sendrier, and Barreto, Paulo S. L. M., "MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes," Cryptology ePrint Archive: Report 2012/409, 2012, https://eprint.iacr.org/2012/409. [19, 28, 43, 90]

[MTSB13] R. Misoczki, J.-P. Tillich, N. Sendrier, and Barreto, Paulo S. L. M., "MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes," in *2013 IEEE International Symposium on Information Theory (ISIT)*, 2013, pp. 2069–2073. [3, 19, 28, 34, 35, 36, 38, 40, 41, 42, 43, 90, 110, 112, 124, 207]

[Nie86] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Problems of Control and Information Theory. Problemy Upravlenija i Teorii Informacii*, no. 15, pp. 159–166, 1986. [2, 5, 21, 24, 154]

[Nig04] Nigel Boston, "Graph-Based Codes," 2004, http://www.math.wisc.edu/~boston/graphcodes.pdf. [17, 39]

[NIKM08] R. Nojima, H. Imai, K. Kobara, and K. Morozov, "Semantic Security for the McEliece Cryptosystem Without Random Oracles," *Designs, Codes and Cryptography*, vol. 49, no. 1-3, pp. 289–305, 2008. [34, 110]

[NIS99] NIST Computer Security Division, "FIPS 46-3, Data Encryption Standard (DES) (withdrawn May 19, 2005)," 1999, http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf. [141]

[NIS01] NIST Computer Security Division, "FIPS 197, Advanced Encryption Standard (AES)," 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. [123, 139, 141]

[NIS07] NIST Computer Security Division, "Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family," 2007, http://csrc.nist.gov/groups/ST/hash/documents/SHA-3_FR_Notice_Nov02_2007-morereadableversion.pdf. [136]

[NIS12] NIST, *Secure hash standard*, ser. Federal information processing standards publication. Gaithersburg, MD and Springfield, VA: Computer Systems Laboratory, National Institute of Standards and Technology, U.S. Dept. of Commerce, Technology Administration and For sale by the National Technical Information Service, 2012, vol. FIPS PUB 180-4. [136]

[NIS13] NIST Computer Security Division, "FIPS 186-4, Digital Signature Standard," 2013, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. [3, 24, 154]

[NIS14] NIST Computer Security Division, "Draft FIPS 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," 2014, http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf. [136]

[NMBB12] R. Niebuhr, M. Meziani, S. Bulygin, and J. Buchmann, "Selecting parameters for secure McEliece-based cryptosystems," *International Journal of Information Security*, vol. 11, no. 3, pp. 137–147, 2012. [36, 207]

[OB09] S. Ouzan and Y. Be'ery, "Moderate-density parity-check codes," *arXiv preprint arXiv:0911.3262*, 2009. [19]

[Ore48] Ø. Ore, *Number theory and its history*, ser. Dover science books. New York: Dover, 1988, ©1948. [23]

[OS09] R. Overbeck and N. Sendrier, "Code-based cryptography," in *Post-Quantum Cryptography*, D. J. Bernstein, J. Buchmann, and E. Dahmen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 95–145. [27]

[Pat75] N. Patterson, "The algebraic decoding of Goppa codes," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 203–207, 1975. [25, 29]

[Per13] E. Persichetti, "Secure and Anonymous Hybrid Encryption from Coding Theory," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, P. Gaborit, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7932, pp. 174–187. [35, 110, 113, 114, 118, 119, 120, 122, 123, 124, 130]

[Per14] R. Perlner, "Optimizing Information Set Decoding Algorithms to Attack Cyclosymmetric MDPC Codes," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, M. Mosca, Ed. Springer International Publishing, 2014, vol. 8772, pp. 220–228. [34, 90, 103, 131]

[Pet10] C. Peters, "Information-Set Decoding for Linear Codes over Fq," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, N. Sendrier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6061, pp. 81–94. [33]

[PG14a] T. Pöppelmann and T. Güneysu, "Area optimization of lightweight lattice-based encryption on reconfigurable hardware," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014, pp. 2796–2799. [66, 67]

[PG14b] T. Pöppelmann and T. Güneysu, "Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware," in *Selected Areas in Cryptography – SAC 2013*, ser. Lecture Notes in Computer Science, T. Lange, K. Lauter, and P. Lisoněk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 8282, pp. 68–85. [58, 59]

[Poi00] D. Pointcheval, "Chosen-Ciphertext Security for Any One-Way Cryptosystem," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, H. Imai, and Y. Zheng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, vol. 1751, pp. 129–146. [34]

[Pol78] J. M. Pollard, "Monte Carlo methods for index computation (mod p)," *Mathematics of computation*, vol. 32, no. 143, pp. 918–924, 1978. [23, 24]

[Pom96] C. Pomerance, "A tale of two sieves," *Notices Amer. Math. Soc*, 1996. [23]

[RED+08] S. Rohde, T. Eisenbarth, E. Dahmen, J. Buchmann, and C. Paar, "Fast Hash-Based Signatures on Constrained Devices," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, G. Grimaud and F.-X. Standaert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5189, pp. 104–117. [159, 164, 174, 175, 209]

[Riv92] R. Rivest, *The MD5 message-digest algorithm*, ser. Network Working Group request for comments. Cambridge, Mass.: MIT Laboratory for Computer Science, 1992, vol. 1321. [136]

[RRM12] C. Rebeiro, S. S. Roy, and D. Mukhopadhyay, "Pushing the Limits of High-Speed GF(2 m) Elliptic Curve Scalar Multiplication on FPGAs," in *Cryptographic Hardware and Embedded Systems – CHES 2012*, ser. Lecture Notes in Computer Science, E. Prouff and P. Schaumont, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7428, pp. 494–511. [58, 59]

[RS60] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [16]

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. [23]

[RSA12] RSA Laboratories, "PKCS #1 v2.2: RSA Cryptography Standard," 2012, http://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf. [3, 33, 154]

[RSVC09] M. Renauld, F.-X. Standaert, and N. Veyrat-Charvillon, "Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5747, pp. 97–111. [176]

[RVM+14] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact Ring-LWE Cryptoprocessor," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, ser. Lecture Notes in Computer Science, L. Batina and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 8731, pp. 371–391. [58, 59]

[RW11] L. Rothamel and M. Weiel, "Report Cryptography Lab SS2011 - Implementation of the RFSB hash function," 2011, http://cayrel.net/IMG/pdf/report.pdf. [138]

[Rya03] W. E. Ryan, "An Introduction to LDPC Codes," 2003, http://www.telecom.tuc.gr/~alex/papers/ryan.pdf. [17]

[RZ14] M. Repka and P. Zajac, "Overview of the Mceliece Cryptosystem and its Security," *Tatra Mountains Mathematical Publications*, vol. 60, no. 1, 2014. [25]

[Saa07] M.-J. O. Saarinen, "Linearization Attacks Against Syndrome Based Hashes," in *Progress in Cryptology – INDOCRYPT 2007*, ser. Lecture Notes in Computer Science, K. Srinathan, C. P. Rangan, and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4859, pp. 1–9. [137]

[Sen05] N. Sendrier, "Encoding information into constant weight words," in *Proceedings. International Symposium on Information Theory, 2005. ISIT 2005*, 2005, pp. 435–438. [30, 110]

[Sen11] N. Sendrier, "Decoding One Out of Many," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, B.-Y. Yang, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 7071, pp. 51–67. [34]

[Sha48] C. E. Shannon, *A mathematical theory of communication.* [S.l.]: [s.n.], 1948. [11]

[Sho97] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, 1997. [2, 23, 24, 154, 180]

[SM11] D. Suzuki and T. Matsumoto, "How to Maximize the Potential of FPGA-Based DSPs for Modular Exponentiation," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E94-A, no. 1, pp. 211–222, 2011. [58, 59]

[SMY09] F.-X. Standaert, T. G. Malkin, and M. Yung, "A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks," in *Advances in Cryptology - EUROCRYPT 2009*, ser. Lecture Notes in Computer Science, A. Joux, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5479, pp. 443–461. [175, 176]

[SPY+10] F.-X. Standaert, O. Pereira, Y. Yu, J.-J. Quisquater, M. Yung, and E. Oswald, "Leakage Resilient Cryptography in Practice," in *Towards Hardware-Intrinsic Security*, ser. Information Security and Cryptography, A.-R. Sadeghi and D. Naccache, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 99–134. [172]

[SRM12] S. Sinha Roy, C. Rebeiro, and D. Mukhopadhyay, "Generalized high speed Itoh–Tsujii multiplicative inversion architecture for FPGAs," *Integration, the VLSI Journal*, vol. 45, no. 3, pp. 307–315, 2012. [58, 59]

[SS92] V. M. Sidelnikov and S. O. Shestakov, "On insecurity of cryptosystems based on generalized Reed-Solomon codes," *Discrete Mathematics and Applications*, vol. 2, no. 4, 1992. [34]

[SSA+09] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger, "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate," in *Advances in Cryptology - CRYPTO 2009*, ser. Lecture Notes in Computer Science, S. Halevi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5677, pp. 55–69. [136]

[SSMS10] A. Shoufan, F. Strenzke, H. G. Molter, and M. Stöttinger, "A Timing Attack against Patterson Algorithm in the McEliece PKC," in *Information, Security and Cryptology – ICISC 2009*, ser. Lecture Notes in Computer Science, D. Lee and S. Hong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 5984, pp. 161–175. [68, 90]

[Ste89] J. Stern, "A method for finding codewords of small weight," in *Coding Theory and Applications*, ser. Lecture Notes in Computer Science, G. Cohen and J. Wolfmann, Eds. Berlin/Heidelberg: Springer-Verlag, 1989, vol. 388, pp. 106–113. [33]

[Ste94] J. Stern, "A new identification scheme based on syndrome decoding," in *Advances in Cryptology — CRYPTO' 93*, ser. Lecture Notes in Computer Science, D. R. Stinson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, vol. 773, pp. 13–21. [154]

[STM+08] F. Strenzke, E. Tews, H. G. Molter, R. Overbeck, and A. Shoufan, "Side Channels in the McEliece PKC," in *Post-Quantum Cryptography*, ser. Lecture Notes in

Computer Science, J. Buchmann and J. Ding, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5299, pp. 216–229. [68, 90]

[STM14] STMicroelectronics, "UM1472 User Manual - Discovery kit for STM32F407/417," 2014, http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/DM00039084.pdf. [94]

[Str10] F. Strenzke, "A Timing Attack against the Secret Permutation in the McEliece PKC," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, N. Sendrier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6061, pp. 95–107. [68, 90]

[Suz07] D. Suzuki, "How to Maximize the Potential of FPGA Resources for Modular Exponentiation," in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Lecture Notes in Computer Science, P. Paillier and I. Verbauwhede, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4727, pp. 272–288. [58]

[SvMG13] T. Schneider, I. von Maurich, and T. Güneysu, "Efficient implementation of cryptographic primitives on the GA144 multi-core architecture," in *2013 IEEE 24th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2013, pp. 67–74. [4]

[SvMGO14] T. Schneider, I. von Maurich, T. Güneysu, and D. Oswald, "Cryptographic Algorithms on the GA144 Asynchronous Multi-Core Processor," *Journal of Signal Processing Systems*, 2014. [4]

[SWM+09] A. Shoufan, T. Wink, G. Molter, S. Huss, and F. Strentzke, "A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2009, pp. 98–105. [52, 58]

[SWM+10] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert, "A Novel Cryptoprocessor Architecture for the McEliece Public-Key Cryptosystem," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1533–1546, 2010. [52, 58, 59]

[Szy04] M. Szydlo, "Merkle Tree Traversal in Log Space and Time," in *Advances in Cryptology - EUROCRYPT 2004*, ser. Lecture Notes in Computer Science, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, M. Y. Vardi, C. Cachin, and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3027, pp. 541–554. [155, 156, 211]

[Tan81] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, 1981. [18]

[TH08] S. Tillich and C. Herbst, "Attacking State-of-the-Art Software Countermeasures—A Case Study for AES," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, ser. Lecture Notes in Computer Science, E. Oswald and P. Rohatgi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 5154, pp. 228–243. [86]

[UN12] US Department of Commerce and NIST, "NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition," 10.10.2012, http://www.nist.gov/itl/csd/sha-100212.cfm. [136]

[Var97] A. Vardy, "The intractability of computing the minimum distance of a code," *IEEE Transactions on Information Theory*, vol. 43, no. 6, pp. 1757–1766, 1997. [33]

[VCMKS12] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, "Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note," in *Advances in cryptology - ASIACRYPT 2012*, ser. Lecture Notes in Computer Science, X. Wang, Ed. Heidelberg: Springer, 2012, vol. 7658, pp. 740–757. [86]

[vMG12] I. von Maurich and T. Güneysu, "Embedded Syndrome-Based Hashing," in *Progress in Cryptology - INDOCRYPT 2012*, ser. Lecture Notes in Computer Science, S. Galbraith and M. Nandi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7668, pp. 339–357. [4, 135]

[vMG14a] I. von Maurich and T. Güneysu, "Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices," in *Design Automation and Test in Europe*, 2014, pp. 1–6. [4, 51]

[vMG14b] I. von Maurich and T. Güneysu, "Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, M. Mosca, Ed. Springer International Publishing, 2014, vol. 8772, pp. 266–282. [4, 89, 131]

[vMHG16] I. von Maurich, L. Heberle, and T. Güneysu, "IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter," in *Post-quantum cryptography*, ser. LNCS sublibrary. SL 4, Security and cryptology, T. Takagi, Ed. Cham: Springer, 2016, vol. 9606, pp. 1–17. [4, 109, 131]

[vMOG15] I. von Maurich, T. Oder, and T. Güneysu, "Implementing QC-MDPC McEliece Encryption," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 3, pp. 1–27, 2015. [4, 37, 51, 89]

[vS87] J. van Lint and T. Springer, "Generalized Reed - Solomon codes from algebraic geometry," *IEEE Transactions on Information Theory*, vol. 33, no. 3, pp. 305–309, 1987. [16]

[Wag02] D. Wagner, "A Generalized Birthday Problem," in *Advances in Cryptology — CRYPTO 2002*, ser. Lecture Notes in Computer Science, M. Yung, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, vol. 2442, pp. 288–304. [154]

[Wie10] C. Wieschebrink, "Cryptanalysis of the Niederreiter Public Key Scheme Based on GRS Subcodes," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, N. Sendrier, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6061, pp. 61–72. [34]

[WOS14] C. Whitnall, E. Oswald, and F.-X. Standaert, "The Myth of Generic DPA...and the Magic of Learning," in *Topics in cryptology - CT RSA 2014*, ser. Lecture Notes in Computer Science, J. Benaloh, Ed. Cham [u.a.]: Springer, 2014, vol. 8366, pp. 183–205. [69]

[WYY05] X. Wang, Y. L. Yin, and H. Yu, "Finding Collisions in the Full SHA-1," in *Advances in Cryptology – CRYPTO 2005*, ser. Lecture Notes in Computer Science, V. Shoup, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3621, pp. 17–36. [136]

[XLF13] T. Xie, F. Liu, and D. Feng, "Fast Collision Attack on MD5," Cryptology ePrint Archive, Report 2013/170, 2013, https://eprint.iacr.org/2013/170. [136]

[XZL+12] N. Xu, J. Zhu, D. Lu, X. Zhou, X. Peng, and J. Du, "Quantum Factorization of 143 on a Dipolar-Coupling Nuclear Magnetic Resonance System [Phys. Rev. Lett. 108 , 130501 (2012)]," *Physical Review Letters*, vol. 109, no. 26, 2012. [2]

List of Figures

List of Tables

List of Algorithms

About the Author

Personal Data

Name	Ingo von Maurich
E-Mail	ingo.vonmaurich@rub.de
Place of birth	Bremen, Germany

Education

08/2011	Doctoral Candidate, Hardware Security Group, Horst Görtz Institute for IT Security, Ruhr-University Bochum
10/2006 – 07/2011	Diploma in IT Security, Ruhr-University Bochum
08/1998 – 07/2005	Abitur, Gymnasium Lilienthal

Internships/Foreign Exchange

01/2011 – 07/2011	Visiting Scholar, Florida Atlantic University, Boca Raton, USA
09/2010 – 11/2010	Internship, IT Security Advisory, KPMG AG

Professional Experience

04/2015	Advanced Security Engineer, NXP Semiconductors Germany GmbH
08/2011 – 03/2015	Research Associate, Hardware Security Group, RUB
04/2009 – 01/2011	Student Assistant, Chair for Embedded Security, RUB
10/2007 – 04/2009	Student Assistant, Chair for Software Engineering, RUB

List of Publications

Peer-Reviewed Publications in Journals

- I. von Maurich, T. Oder, and T. Güneysu, "Implementing QC-MDPC McEliece Encryption," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 3, pp. 1–27, 2015.
- C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Horizontal and Vertical Side Channel Analysis of a McEliece Cryptosystem," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1093–1105, 2016.
- T. Schneider, I. von Maurich, T. Güneysu, D. Oswald, "Cryptographic Algorithms on the GA144 Asynchronous Multi-Core Processor - Implementation and Side-Channel Analysis," in *Journal of Signal Processing Systems*, vol. 77, pp. 151-167, 2014.

Peer-Reviewed Publications in Conference Proceedings

- I. von Maurich, L. Heberle, and T. Güneysu, "IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter," in *Post-quantum cryptography*, ser. LNCS sub-library. SL 4, Security and cryptology, T. Takagi, Ed. Cham: Springer, 2016, vol. 9606, pp. 1–17.
- C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Masking Large Keys in Hardware: A Masked Implementation of McEliece," in *Selected areas in cryptography - SAC 2015*, ser. LNCS, O. Dunkelman and L. Keliher, Eds. Springer, 2016, vol. 9566, pp. 293–309.
- C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Differential Power Analysis of a McEliece Cryptosystem," in *Applied cryptography and network security*, ser. LNCS sub-library. SL 4, Security and cryptology, T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, Eds. Cham: Springer, 2015, vol. 9092, pp. 538–556.
- I. von Maurich and T. Güneysu, "Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices," in *Post-Quantum Cryptography*, ser. LNCS, M. Mosca, Ed. Springer, 2014, vol. 8772, pp. 266–282.
- I. von Maurich and T. Güneysu, "Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices," in *Design, Automation and Test in Europe*, 2014, pp. 1–6.

- S. Heyse, I. von Maurich, and T. Güneysu, "Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, ser. Lecture Notes in Computer Science, G. Bertoni, and J.-S. Coron, Eds. Springer Berlin Heidelberg, 2013, vol. 8086, pp. 273–292.

- T. Eisenbarth, I. von Maurich, and X. Ye, "Faster Hash-Based Signatures with Bounded Leakage," in *Selected Areas in Cryptography – SAC 2013*, ser. Lecture Notes in Computer Science, T. Lange, K. Lauter, and P. Lisonˇ ek, Eds. Springer Berlin Heidelberg, 2014, vol. 8282, pp. 223–243.

- T. Schneider, I. von Maurich, and T. Güneysu, "Efficient Implementation of Cryptographic Primitives on the GA144 Multi-core Architecture," in *International Conference on Application-Specific Systems, Architectures and Processors - ASAP 2013*, 2013, pp. 67-74.

- I. von Maurich and T. Güneysu, "Embedded Syndrome-Based Hashing," in *Progress in Cryptology - INDOCRYPT 2012*, ser. Lecture Notes in Computer Science, S. Galbraith, and M. Nandi, Eds. Springer Berlin Heidelberg, 2012, vol. 7668, pp. 339–357.

- J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Pöppelmann, F. Regazzoni, F.-X. Standaert, G. van Assche, R. van Keer, van Oldeneel tot Oldenzeel, and I. von Maurich, "Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices," in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, S. Mangard, Ed. Springer Berlin Heidelberg, 2013, vol. 7771, pp. 158–172.

- T. Kasper, I. von Maurich, D. Oswald and C. Paar, "Chameleon: A Versatile Emulator for Contactless Smartcards," in *Information Security and Cryptology - ICISC 2010*, ser. Lecture Notes in Computer Science, K. H. Rhee and D. Nyang, Eds. Springer Berlin Heidelberg, 2010, vol. 6829, pp. 189-206.

Book Chapters

- T. Eisenbarth, I. von Maurich, C. Paar, and X. Ye, "A Performance Boost for Hash-Based Signatures," in *Number Theory and Cryptography*, ser. Lecture Notes in Computer Science, M. Fischlin, and S. Katzenbeisser, Eds. Springer Berlin Heidelberg, 2013, vol. 8260, pp. 166–182.

Magazine Articles

- C. Paar, I. von Maurich, M. Wolf, "IT Security and Electromobility," in *ATZelektronik worldwide*, Springer Automotive Media, 2012, vol. 7, no. 4, pp. 24-29.

- C. Paar, I. von Maurich, M. Wolf, "IT Sicherheit in der Elektromobilität," in *ATZelektronik*, Springer Automotive Media, 2012, vol. 7, no. 4, pp. 274-279.

Technical Reports

- S. Heyse, I. von Maurich, A. Wild, C. Reuber, J. Rave, T. Pöppelmann, C. Paar, and T. Eisenbarth, "Evaluation of SHA-3 Candidates for 8-bit Embedded Processors," in *2nd SHA-3 Candidate Conference*, 2010.

Invited Talks

- I. von Maurich. Smaller Keys for Code-based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices, *4th Code-based Cryptography Workshop, Rocquencourt, France, June 10-12, 2013*.
- I. von Maurich. Advances in Implementations of Code-based Cryptography on Reconfigurable Devices, *HGI Kolloquium, Ruhr University Bochum, Germany, November 21, 2013*.

Participation in Selected Conferences & Workshops

- PQCrypto'16, 7th Conference on Post-Quantum Cryptography, 2016, Fukuoka, Japan
- Post-Quantum Cryptography Winter School, 2016, Fukuoka, Japan
- 32C3, 32th Chaos Communication Congress, 2015, Hamburg, Germany
- Summer School on Real-World Crypto and Privacy, 2015, Sibenik, Croatia
- RWC'15, Real World Cryptography Workshop, 2015, London, United Kingdom
- 31C3, 31th Chaos Communication Congress, 2014, Hamburg, Germany
- 2nd ETSI Quantum-Safe Crypto Workshop, 2014, Ottawa, Canada
- PQCrypto'14, 6th Conference on Post-Quantum Cryptography, 2014, Waterloo, Canada
- Post-Quantum Cryptography Summer School, 2014, Waterloo, Canada
- Design and Security of Cryptographic Algorithms and Devices for Real-World Applications, 2014, Sibenik, Croatia
- Security in Times of Surveillance, 2014, Eindhoven, Netherlands
- DATE'14, Design, Automation and Test in Europe, 2014, Dresden, Germany
- 30C3, 30th Chaos Communication Congress, 2013, Hamburg, Germany
- CHES'13, 15th Workshop on Cryptographic Hardware and Embedded Systems, 2013, Santa Barbara, USA
- CRYPTO'13, 33rd International Cryptology Conference, 2013, Santa Barbara, USA

- SAC'13, 20th Conference on Selected Areas in Cryptography, 2013, Vancouver, Canada
- CryptArchi'13, 11th International Workshop on Cryptographic Architectures Embedded in Reconfigurable Devices, 2013, Fréjus, France
- CBC'13, 4th Code-based Cryptography Workshop, 2013, Rocquencourt, France
- ASAP'13, 24th IEEE International Conference on Application-specific Systems, Architectures and Processors, 2013, Washington D.C., USA
- Crypto for 2020, 2013, Tenerife, Spain
- 29C3, 29th Chaos Communication Congress, 2012, Hamburg, Germany
- Post-Quantum Cryptography and Quantum Algorithms, 2012, Lorentz Center, Leiden, Netherlands
- CHES'12, 14th Workshop on Cryptographic Hardware and Embedded Systems, 2012, Leuven, Belgium
- Indocrypt'12, 13th International Conference on Cryptology in India, 2012, Kolkata, India
- Asiacrypt'12, 18th Annual International Conference on the Theory and Application of Cryptology and Information Security, 2012, Beijing, China
- CBC'12, 3rd Code-based Cryptography Workshop, 2012, Copenhagen, Denmark
- PQCrypto'11, 5th Conference on Post-Quantum Cryptography, 2011, Taipei, Taiwan
- Asiacrypt'11, 17th Annual International Conference on the Theory and Application of Cryptology and Information Security, 2011, Seoul, South Korea
- CHES'11, 13th Workshop on Cryptographic Hardware and Embedded Systems, 2011, Nara, Japan
- European Trusted Infrastructure Summer School, 2011, Darmstadt, Germany
- RFIDSec'11, 7th Workshop on RFID Security and Privacy, 2011, Northampton, USA
- ICISC'11, 14th Annual International Conference on Information Security and Cryptology, 2011, Seoul, South Korea
- Eurocrypt'09, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2009, Cologne, Germany