Power analysis on FPGA implementation of Classic McEliece

Andreas Johansson
an6883jo-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Qian Guo

Examiner: Thomas Johansson

October 25, 2021

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In this work, the hardware implementation of Classic McEliece has been assessed for side-channel leakage through a power analysis. The official, unprotected, decryption procedure of Classic McEliece was implemented on a Xilinx Atix7 field-programmable gate array and incorporated into the ChipWhisperer framework. Traces captured during decryption was assessed for information leakage and it was concluded that the implementation leaks information at multiple points. A procedure for a partial message-recovery on Classic McEliece was suggested where a neural network was employed to predict the distribution of bit values in the plaintext. The suggested attack procedure managed to predict if the Hamming weight of the first half plaintext bits was greater than 32 or not with an accuracy of 78 %. During the attack, only a single trace was used for predicting the Hamming weight. The suggested attack procedure targets the last step of decryption where plaintext bits are recovered. More specifically, the suggested attack procedure exploits leakage from the incremental storage of plaintext bits in a shift register.

Acknowledgments

I would like to thank my supervisor Qian Guo for supporting me with good feedback and interesting discussions during this thesis. Many of the questions you asked made me think one step further which in turn pushed this work forward. I also would like to thank the Department of Electrical and Information Technology in Lund for the opportunity to finish my master's degree. Finally, I would like to thank my wife and kids for their endless support, encouragement, patience, and understanding during the last years.

Popular science summary

So, you came up with a strong password used when signing into government websites. Well, then you don't need to worry that anyone else can get access to your data, or should you worry? In this work, it is shown that strong passwords alone do not guarantee that personal data will be kept confidential.

More and more of our daily activities such as work, social interaction, and contact with authorizes are performed with an ever-increasing number of electronic devices around us all connected to the Internet. Often we transmit sensitive data, like bank account numbers, private messages over some social platform, or images from our home security camera over the Internet. Many people do not concern themselves about security when they transmit sensitive data over the Internet, and why should they? After all, they used a strong password to authorize themselves before sending any sensitive information over the Internet.

National authorities also handle a lot of information that needs to be kept confidential, both for keeping the integrity of citizens and for national safety. Authorities might be more aware of how to keep sensitive data secure while it sent over the Internet. But, both authorizes and ordinary people rely on that their sensitive data is encrypted before it is sent over the Internet. This is possible since serval cryptosystems have been developed throughout the years where a private key is used to encrypt sensitive data. In an ideal cryptosystem, only the holder of the private key should be able to retrieve the original data that was encrypted.

However, now and then some cryptosystems are reported as broken as someone has figured out how to get their hands on sensitive encrypted data without having access to the private key. Typically, cryptographic systems rely on that some mathematical problems that take a long time to solve without knowledge of the private key. When a cryptosystem is broken, someone has typically found a flaw in the used mathematical problem that allows them to solve the problem in a short time. And by solving the problem the encrypted data can be decrypted without access to the private key.

A big concern for many of today's cryptosystems is the increased research and development of quantum computers. If or when a sufficiently powerful quantum computer becomes a reality, many of the mathematical problems used in today's cryptosystems will be easily solved. This is a well-known fact in the research community, and in an attempt to fuel the development of new quantum computer resistant cryptosystems, the US agency National Institute of Standards and

Technology launched a competition for finding new cryptosystems. Currently, the competition is in its last round and one of the finalists is called Classic McEliece.

Since Classic McEliece made it to the final there is a high hope that this cryptosystem is quantum computer resistant. However, as a cryptosystem is implemented on an electronic device another possible threat opens up. Since 1996 it has been known that by measuring the power consumption of an electronic device it is sometimes possible to retrieve the private key of a cryptosystem. Thereby, an attacker can bypass the tedious work of solving the underlying mathematical problem of the cryptosystem. Therefore, it is important to assess if potential future cryptosystems can be broken by observing the power consumption of the device where the cryptosystem is implemented.

Since Classic McEliece is a possible future standard cryptosystem there is an interest to evaluate this system in multiple ways. In this thesis, the power consumption of Classic McEliece was measured while decryptions were executed. It turned out that a straightforward implementation of Classic McEliece suffers from a lot of information leakage that potentially could be exploited by an attacker, even without a quantum computer.

Table of Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Thesis goal and contributions . . . 2
- 1.3 Related work . . . 2
- 1.4 Scope . . . 3
- 1.5 Outline . . . 3

2 Theory **5**
- 2.1 Coding theory . . . 5
- 2.2 McEliece cryptosystem . . . 7
- 2.3 Niederreiter cryptosystem . . . 8
- 2.4 Classic McEliece . . . 8
- 2.5 Field programmable gate array . . . 9
- 2.6 Side-channel attack . . . 10
- 2.7 Leakage assessment . . . 13

3 Classic McEliece for ChipWhisperer platform **15**
- 3.1 Implementation of hardware modules . . . 15
- 3.2 Software modules . . . 25
- 3.3 Partial message-recovery attack on Classic McEliece . . . 30

4 Result **35**
- 4.1 Trace capture metrics . . . 35
- 4.2 Leakage evaluation . . . 36
- 4.3 Partial message-recovery attack . . . 40

5 Discussion **45**
- 5.1 From an evaluators point of view . . . 45
- 5.2 From an attackers point of view . . . 46

6 Conclusion **49**

References **51**

A McEliece Python module 53

B Extension of CW305 API 57

C Trace capture script 61

D Probability of HW in a window 65

List of Figures

2.1 Simplified internals of an FPGA consisting of configurable logic block (CLB)s, input-output block (IOB)s, and programmable interconnect. 9
2.2 A fully connected neural network with a single hidden layer and a single output, $j = 3$ and $i_k = \{3, 4, 4, 1\}$. 11

3.1 Simplified block diagram of CW, consisting of an analog-to-digital converter (ADC) and amplifier, connected to a target device. 16
3.2 System overview consisting of a PC, CW, and the add-on board CW305. 16
3.3 Overview of FPGA implementations consisting of three modules. . . 17
3.4 Overview of decryption core used in Classic McEliece. 17
3.5 Implemented decryption core interface. 19
3.6 Organization of bits in the signal *poly_g*. 20
3.7 Organization of bits in the secret support memory. 20
3.8 Organization of bits of the ciphertext. 20
3.9 Block diagram of the interface between PC and FPGA of CW305 . . 21
3.10 USB payload. 22
3.11 USB advanced encryption standard (AES) address format. 22
3.12 USB address format fo implemented decryption core. 22
3.13 Overview of the decryption module consisting of the decryption core, three memories, and a multiplexer. 24
3.14 Memory for storing secret Goppa polynomial. 24
3.15 Memory for storing ciphertext. 25
3.16 Memory for storing secret support. 25
3.17 Procedure for trace capturing. 29
3.18 Simplified block diagram of the last decryption step. 31
3.19 Pipeline structure used in the hardware decryption implementation during plaintext recovery. 32
3.20 Sweeping Hamming weight (HW) window across a plaintext. The HW of the plaintext inside the windows was calculated for each window position. 32

4.1 Mean measured voltage of 1000 traces using a fixed, randomly selected, ciphertext. 37

4.2 Distribution of measured voltage of 1000 decryption cycles using random ciphertexts. 38
4.3 Fixed vs. random ciphertext leakage assessment with set sizes $n_0 = n_1 = 2984$. 38
4.4 Leakage assessment where traces were sorted into sets based on HW of the first half of plaintexts. 39
4.5 Leakage assessment of the last decryption step where traces were sorted into sets based on HW of the first half of plaintexts. 39
4.6 Leakage assessment using samples with different offsets from the rising edge of decryption clock, (a) 0 offset, (b) 1/6 clk offset, (c) 2/6 clk offset, (d) 3/6 clk offset, (e) 4/6 clk offset and (f) 5/6 clk offset. . . 40
4.7 Leakage assessment of last decryption step where the shift register was removed from the hardware. 41
4.8 Prediction accuracy using samples with different offsets from the rising edge of decryption clock. 41
4.9 Prediction accuracy of HW windows belonging to the class with HW $>$ 32. The graph also shows the true fraction of traces belonging to the other class with HW $\leq$ 32. 42
4.10 Prediction accuracy of HW windows belonging to the class with HW $<$ 32. The graph also shows the true fraction of traces belonging to the other class with HW $\geq$ 32. 42
4.11 Prediction accuracy using different HW window sizes. 43
4.12 Comparison of prediction accuracy when removing the shift register from the decryption pipeline. 43

D.1 Probability mass function of $\mathbf{D}$ where $n = 3488$ and $t = 64$. 66
D.2 Difference between the expected value of the two classes for different HW window sizes. 66

List of Tables

3.1 Crypto and design parameters of decryption core. 18
3.2 Crypto and design parameters used when building the decryption core. 19
3.3 Clock cycles for each of the four constant steps of decryption. 21
3.4 Register addresses. 23
3.5 Neural network architecture used to classify traces based on HW windows. 33

4.1 FPGA resource requirements and utilization. 35
4.2 Execution time for key generation. 36
4.3 Execution time for trace capturing. 36
4.4 By using a principal component analysis (PCA) dimensionality was reduced to 547 features leading to a neural network with 449634 parameters. 41

List of Acronyms

ADAM adaptive moment estimation

ADC analog-to-digital converter

AES advanced encryption standard

API application programming interface

ASIC application-specific integrated circuit

BM Berlekamp-Massey

CCA chosen-ciphertext attack

CLB configurable logic block

CNN convolutional neural network

CW ChipWhisperer

DPA differential power analysis

ELP error locater polynomial

EM electromagnetic

FFT fast Fourier transform

FPGA field-programmable gate array

HW Hamming weight

IC integrated circuit

IND-CCA2 indistinguishably under adaptive chosen ciphertext attack

IOB input-output block

KEM key encapsulation mechanism

MLP multiple layer perceptron

NIST national institute of standards and technology

OW-CPA one-way chosen-plaintext attack

PCA principal component analysis

PCB printed circuit board

PKC public-key cryptosystem

PQC post-quantum cryptography

ReLU rectified linear unit

SCA side-channel attack

SPA simple power analysis

XGCD extended greatest common divider

Chapter 1
Introduction

This chapter presents the background, goals, and contributions of this work. Related works are also presented in this chapter.

1.1 Background

To meet the security threat of quantum computers, the US agency national institute of standards and technology (NIST) announced a competition for post-quantum cryptography (PQC) [13]. The goal of the competition is to find new algorithms which are believed to be secure even after large quantum computers become available. The competition consists of four rounds and in each round, some of the suggested algorithms are selected to advance to the next round. Currently, the competition has entered its last round. Four finalists for public-key encryption and three for digital signature algorithms remain in the competition.

One of the finalists in the NIST PQC competition is Classic McEliece which is a code-based cryptosystem derived from Niederreiter's cryptosystem [1]. It belongs to the family of code-based cryptography and the underlying security has been investigated for more than 40 years. The name of the proposed cryptosystem was chosen to honor Robert J. McEliece, the inventor of the first cryptosystem based on coding theory. However, despite its name Classic McEliece is based on the public-key cryptosystem (PKC) suggested by Niederreiter which is a variant of the PKC initially proposed by McEliece. The two cryptosystems introduced by McEliece and Niederreiter are equal from a security perspective as if one mange to break one of them, the other can also be broken.

One drawback with McEliece PKC, Niederreiter PKC, and Classic McEliece is the huge size of both public and private keys. This makes software implementations in embedded systems unsuitable since keys would occupy a large fraction of the available on-chip memory. However, in the case Classic McEliece is selected by NIST as the new standard the algorithm will likely be made available to embedded systems as hardware implementations. Therefore, it is of practical significance that side-channel leakage of such hardware implementations is investigated. Previously one publication regarding a side-channel attack (SCA) on hardware implementations of Classic McEliece has been found where a message-recovery attack was suggested [9]. However, it is still unclear if a key-recovery attack could be conducted which could be a much stronger attack.

1.2 Thesis goal and contributions

The goal of this project was to investigate side-channel leakage of the official field-programmable gate array (FPGA) implementation of Classic McEliece through power analysis. Based on this the following research questions were formed.

1. Does the hardware implementation of Classic McEliece leak side-channel information through its power consumption during decryption?
2. If so, how can this be used during a message-recovery attack?

To answer stated questions, the project was divided into the following sub-goals

- Port the official FPGA decryption implementation of Classic McEliece to the ChipWhisperer platform.
- Acquire power traces from the FPGA as decryption is executed and search for leakage points.
- Investigate possible procedures that could be used to perform a message-recovery attack.

Contributions of this work are

- a detailed analysis of the official Classic McEliece hardware implementation.
- integration of Classic McEliece to the ChipWhisperer framework.
- leakage assessment during decryption procedure.
- a suggested procedure for a partial message-recovery attack.

1.3 Related work

The first SCA on McEliece was reported in 2008 [20]. the authors of this paper showed that a straightforward software implementation of the McEliece cryptosystem may leak information in several side channels. In the paper, a successful timing attack was performed which recovered the error vector that was added to the plaintext during encryption. This was achieved by applying bit-flips to the ciphertext and measuring the execution time of the decoding algorithm. More specifically, the attack measured the execution time of the extended greatest common divider (XGCD) during error vector reconstruction. However, since the attack only retrieved the error vector an attack had to be relaunched for every ciphertext as the error vector is randomly generated for each message. The authors of the paper also suggested a power analysis attack that could be performed during the key generation procedure of McEliece.

One of the first SCA on hardware implantation of McEliece was investigated in 2010 [18]. In this work, it was suggested that the random error vector could be retrieved by measuring execution times during decryption. The attack procedure was similar to [20]. But, instead of measuring execution time during error vector reconstruction, the time during error locater polynomial (ELP) calculation was measured. The side channel found in this work was caused by a break condition

during the calculation of ELP. However, the attack was only simulated for an FPGA and no real measurements were performed.

Probably the first power analysis attack on a software implementation of McEliece was performed during 2010 where simple power analysis (SPA) was used [7]. The attack was launched against different implementations suitable for 8-bit micro-controllers. In this work, the secret key was recovered by observing power consumption during decryption of ciphertext with Hamming weight (HW) equal to one, i.e. HW=1.

The first attack with real measurements on a hardware implementation of McEliece was reported in [12]. In this paper, the same side-channel leakage as in [18] was exploited. But, instead of measuring execution time, the power consumption during ELP calculation was measured. By tracing the power consumption, the number of iterations during XGCD was estimated. The target of the attack was an FPGA implementation of McEliece. During the attack, a single bit-flip was introduced in the ciphertext and after repeated bit-flips, the error vector was retrieved. The error vector was then used to recover the plaintext message.

In [15] differential power analysis (DPA) was employed instead of SPA. In this paper, a software implementation of McEliece was targeted. More specifically, the authors attacked different versions of the decryption procedure which employed a permutation of the ciphertext as the first step of decoding. These decoders are considered less secure. But, they are considered more suitable in constrained hardware like embedded systems. During the attack, the secret permutation matrix was recovered.

Another plaintext recovery attack was presented in 2020 [9]. In this paper, the work done in [18] was transformed to attacking the Niederreiter cryptosystem, i.e. the same system that is used in Classic McEliece. The attack was performed on the FPGA reference implementation of Classic McEliece. Side-channel information was acquired by measuring electromagnetic (EM) leakage and the authors managed to recover plaintexts of encrypted messages.

1.4 Scope

This thesis mainly focuses on the decryption primitive used in Classic McEliece. There exist multiple methods for side-channel leakage assessment, but in this thesis, only one method was used. The same goes for side-channel attack methods where this thesis mainly focuses on using a neural network to predict sensitive information in the FPGA during decryption. Apart from the aforementioned limitations, this thesis mainly focuses on a message-recovery attack. But, some parts are also relevant for a key-recovery attack.

1.5 Outline

This thesis is organized as follows. In chapter 2, a theoretical background of Classic McEliece is given along with an introduction to side-channel attacks, leakage assessment, FPGA's, and neural networks. In chapter 3, the implemented hardware of the Niederreiter decryption core is presented. The interface between the

decryption core and the CW is explained. This chapter also gives detailed information on how keys were generated, how traces were captured, and how leakage assessment was performed. Furthermore, in chapter 3, a procedure is presented for a partial message-recovery attack. Results of leakage assessment and performance of suggested attack procedure are presented in chapter 4. Finally, in chapter 5, the results are discussed and the work is concluded in chapter 6.

Chapter 2
Theory

In this chapter, a brief introduction to coding theory is presented. The three cryptosystems McEliece PKC, Niederreiter PKC, and Classic McEliece are introduced and explained. Further, a brief introduction to FPGAs is given as well as an introduction to SCA. At the end of the chapter, Welch's t-test is introduced and its use for leakage assessment is discussed.

2.1 Coding theory

The purpose of this section is to briefly introduce finite fields and linear codes which are used extensively in this thesis. For more in-depth information on these topics, the reader is referred to [8] which is used as a reference for this section.

2.1.1 Finite fields

A field $\mathbb{F}$ is a set of elements together with two binary operations, called addition and multiplication, for which the result is an element in $\mathbb{F}$. Performing addition or multiplication between elements a, b, and c of $\mathbb{F}$ should satisfy

- Associativity, $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- Commutativity, $a + b = b + a$ and $a \cdot b = b \cdot a$
- Distributivity, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

and there should for every element a in $\mathbb{F}$

- exist an element, denoted 0, in $\mathbb{F}$ such that $a + 0 = a$ (additive identity)
- exist an element, denoted 1, in $\mathbb{F}$ such that $a \cdot 1 = a$ (multiplicative identity)
- exist an element, denoted $-a$, in $\mathbb{F}$ such that $a+(-a) = 0$ (additive inverses)
- exist an element, denoted a^{-1}, in $\mathbb{F}$ such that $a \cdot a^{-1} = 1$ (multiplicative inverses)

A finite field is a field where the number of elements is finite. The order or size of a finite field is determined by the number of elements in it. A finite field of order q only exists if $q = p^k$ where p is a prime number and is k a positive integer. An example of a finite field is integers modulo a prime number p, commonly denoted

$\mathbb{Z}/p\mathbb{Z}$. When q is a prime number, i.e. when $k = 1$, the finite field is called a prime field denoted $\mathbb{F}_p$, when $k > 1$, the field is called an extension field denoted $\mathbb{F}_{p^k}$. Moreover, a finite field is commonly called a Galois field. The notation $\mathrm{GF}(p^k)$ is commonly used in literature to denote a finite field of order $q = p^k$. In this thesis, only fields with $p = 2$ are used.

A polynomial $f(x)$ with coefficients in $\mathbb{F}_q$, i.e.

$$f(x) = \sum_{i=0}^{n} c_i \cdot x^i \tag{2.1}$$

where all c_i are elements in $\mathbb{F}_q$, is called a polynomial over $\mathbb{F}_q$. The degree of $f(x)$ is given by the largest $j \leq n$ for which $c_j \neq 0$ and if $c_j = 1$, the polynomial is called a monic polynomial. The set of polynomials over $\mathbb{F}_q$ is denoted $\mathbb{F}_q[x]$. However, the set $\mathbb{F}_q[x]$ is not finite.

An irreducible polynomial over $\mathbb{F}_q$ is a non-constant polynomial that cannot be constructed as a product of two or more non-constant polynomials from $\mathbb{F}_q[x]$. Given an irreducible polynomial $f(x)$ over $\mathbb{F}_q$, a finite field of polynomials can be formed by taking the polynomials modulo $f(x)$. This is commonly written $\mathbb{F}_q[x]/f(x)\mathbb{F}_q[x]$, which can be compared to the finite field $\mathbb{Z}/p\mathbb{Z}$ given previously, $\mathbb{F}_q[x]$ constitutes the elements in the set like $\mathbb{Z}$ does and $f(x)$ plays the same role as the prime p does. When working with the set $\mathbb{F}_q[x]/f(x)\mathbb{F}_q[x]$, coefficients of polynomials are module q and the polynomials are modulo $f(x)$. In the following chapters, the notation $\mathrm{GF}(p^k)$ will be used to denote the set $\mathbb{F}_q[x]$, i.e. the set of polynomials over $\mathbb{F}_q$ where $q = p^k$.

2.1.2 Linear code

An n-dimensional vector space over a finite field $\mathbb{F}_q$ consists of a set of q^n vectors. Each vector can be represented by a sequence of n elements $(a_{n-1}, \cdots a_0)$ where each a_i is an element in $\mathbb{F}_q$. The n-dimensional vector space over $\mathbb{F}_q$ is denoted $\mathbb{F}_q^n$

A $[n, k]$ linear code $\mathcal{C}$ is a subset of $\mathbb{F}_q^n$, consisting of q^k vectors. Each vector in $\mathcal{C}$ is called a codeword $\mathbf{c}$. Thus, code $\mathcal{C}$ has q^k codewords. The parameter n is called the code length and k is called the code dimension. In a linear $[n, k]$ code, n-dimensional codewords are constructed from k-dimensional words by encoding the words. A linear code is commonly represented by a generator matrix $\mathbf{G}$ or a parity check matrix $\mathbf{H}$.

A generator matrix $\mathbf{G}$ is any $k \times n$ matrix whose rows form a basis for the code, vector space, $\mathcal{C}$. In general, there exist many $\mathbf{G}$ for a given code and a generator matrix of the form $[\mathbf{I}_k \mid \mathbf{A}]$, where $\mathbf{I}_k$ is the $k \times k$ identity matrix, is called to be in systematic form.

A parity check matrix $\mathbf{H}$ of the code $\mathcal{C}$ is an $(n - k) \times n$ matrix defined by

$$\mathbf{H} = \left\{\mathbf{c} \mid \mathbf{H}\mathbf{c}^{\mathrm{T}} = \mathbf{0}\right\} \tag{2.2}$$

where $\mathbf{c}$ is a codeword of $\mathcal{C}$. In general, there exist many $\mathbf{H}$ for a given $\mathcal{C}$ and a parity check matrix of the form $[\mathbf{B} \mid \mathbf{I}_{n-k}]$, where $\mathbf{I}_{n-k}$ is the $(n - k) \times (n - k)$ identity matrix, is called to be in systematic form.

If a code $\mathcal{C}$ is given by either $\mathbf{H}$ or $\mathbf{G}$ in systematic form, i.e. $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{A}]$ or $\mathbf{H} = [\mathbf{B} \mid \mathbf{I}_{n-k}]$, the other can be derived in systematic form by the relation $\mathbf{B} = -\mathbf{A}^{\mathrm{T}}$.

To encode a k-dimensional vector $\mathbf{x}$ using a code $\mathcal{C}$, the vector is mapped to a codeword $\mathbf{c}$ as $\mathbf{c} = \mathbf{x}\mathbf{G}$. Keep in mind that adding and multiplying is done over $\mathbb{F}_q^n$. Since $\mathbf{c}$ has a higher dimension than $\mathbf{x}$, redundancy can be added to the codeword which later can be used to detect or correct possible errors introduced while transmitting the codeword. The number of errors that can be detected or corrected depends on the employed code $\mathcal{C}$.

The syndrome $\mathbf{s}$ of a codeword $\mathbf{c}$ for given $\mathcal{C}$ is defined by

$$\mathbf{s} = \mathbf{H}\mathbf{c}^{\mathrm{T}}. \tag{2.3}$$

Thus, every valid codeword of $\mathcal{C}$ will have a syndrome equal to $\mathbf{0}$. This can be used to detect and correct errors in a received codeword.

There exist many types of codes $\mathcal{C}$ but in this thesis, only Goppa codes are considered as this is the type of code used in Classic McEliece. The reason for using Goppa codes is that they have god error-correcting capability and it is hard to distinguish the parity check matrix of a Goppa code from a random matrix.

2.2 McEliece cryptosystem

In 1978, McEliece proposed the first public-key encryption scheme based on coding theory [11]. Today, this encryption scheme is known as the McEliece PKC. The idea behind the proposed cryptosystem was the existence of fast decoding algorithms for general Goppa code while no decoding algorithm existed for general linear codes.

McEliece cryptosystem is defined by two parameters, code length n, and the number of added errors t. To construct the secret key, a random irreducible polynomial over $\mathrm{GF}(2^m)$ is chosen with $n = 2^m$. To each irreducible polynomial over $\mathrm{GF}(2^m)$, there exist a Goppa code of length $n = 2^m$ and dimension $k \geq n - mt$ capable of correcting up to t errors. The generator matrix $\mathbf{G}$ of size $k \times n$ for the Goppa code is then derived. Next, a random dense non-nonsingular scramble matrix $\mathbf{S}$ of size $k \times k$ is constructed as well as a random permutation matrix $\mathbf{P}$ of size $n \times n$. The three matrices $\mathbf{G}$, $\mathbf{S}$, and $\mathbf{P}$ are the secret key of the cryptosystem. The public key is an obfuscated generator matrix $\mathbf{G}'$, constructed from the private key as $\mathbf{G}' = \mathbf{S}\mathbf{G}\mathbf{P}$.

Data to be encrypted is first split into k-bit words $\mathbf{u}_i$. The ciphertext $\mathbf{c}_i$ for each block $\mathbf{u}_i$ is generated by $\mathbf{c}_i = \mathbf{u}_i\mathbf{G}' + \mathbf{e}$ where $\mathbf{e}$ is a random n-bit word with Hamming weight$= t$.

During decryption, the secret key is first used to undo the permutation of the ciphertext to form $\mathbf{c}_i' = \mathbf{c}_i\mathbf{P}^{-1}$ which is a codeword of the chosen Goppa code. Next, a decoding algorithm in conjunction with the secret generator matrix $\mathbf{G}$ is used to recover the scrambled data $\mathbf{u}_i\mathbf{S}$. Finally, the secret key is used to unscramble the data as $\mathbf{u}_i = \mathbf{u}_i\mathbf{S}\mathbf{S}^{-1}$.

The security of McEliece PKC relies on the hardness of decoding a general linear code and distinguishing the public-key matrix from a random matrix.

A drawback of the McEliece cryptosystem is the large key sizes. The public key $\mathbf{G}'$ can be transformed to a systematic form, so there is no need to store the identity matrix part of the generator matrix. But still, the matrix will be of size $k \times (n-k)$.

2.3 Niederreiter cryptosystem

In 1986, Niederreiter proposed a code-based cryptosystem in which a parity check matrix $\mathbf{H}_{\text{pub}}$ was used as the public key [14]. Niederreiter suggested using Reed-Solomon codes but later this turned out to be insecure. Niederreiter also mentioned that Goppa codes could be used. Later, it was shown that by using Goppa codes the security of the Niederreiter cryptosystem is equivalent to the security of the McEliece cryptosystem.

To construct keys Niederreiter suggested to selects a linear $[n, k]$ code $\mathcal{C}$ capable of correcting t errors. Then, the parity check matrix $\mathbf{H}$ of $\mathcal{C}$ is derived. The public key $\mathbf{H}_{\text{pub}}$ is obtained by $\mathbf{H}_{\text{pub}} = \mathbf{MHP}$, where $\mathbf{M}$ is a random non-singular matrix of size $(n-k) \times (n-k)$ and $\mathbf{P}$ a random permutation matrix of size $n \times n$. The matrices $\mathbf{M}$, $\mathbf{H}$, and $\mathbf{P}$ are kept as the secret key.

During encryption, data is encoded to n-bit long blocks $\mathbf{u}_i$ with weight t. The ciphertext $\mathbf{c}_i$ of each block is then obtained by $\mathbf{c}_i = \mathbf{H}_{\text{pub}}\mathbf{u}_i$. Since the $\mathbf{u}_i$ is multiplied by a parity check matrix $\mathbf{c}_i$ can be viewed as the syndrome of the erroneous codeword $\mathbf{u}_i$ in $\mathcal{C}$.

In decryption, a ciphertext $\mathbf{c}_i$ is first multiplied by the inverse of $\mathbf{M}$ to get $\mathbf{M}^{-1}\mathbf{u}_i = \mathbf{M}^{-1}\mathbf{MHP}\mathbf{u}_i = \mathbf{HP}\mathbf{u}_i^T = \mathbf{H}(\mathbf{u}_i\mathbf{P}^T)^T$. Since $\mathbf{u}_i\mathbf{P}^T$ is of weight t, a syndrome decoding algorithm can be used to retrieve $\mathbf{u}_i\mathbf{P}^T$ from $\mathbf{HP}\mathbf{u}_i^T$. Lastly, the plaintext, i.e. the error causing the syndrome $\mathbf{u}_i$, is retrieved by multiplying $\mathbf{u}_i\mathbf{P}^T$ with the inverse of $\mathbf{P}$.

Just as the McEliece cryptosystem, the security of the Niederreiter cryptosystem relies on the hardness of decoding a random linear code and distinguishing the public-key matrix from a random matrix.

2.4 Classic McEliece

Classic McEliece is one of the finalists in the PQC competition. The name, Classic McEliece, of the proposed cryptosystem, is to honor the inventor of the first cryptosystem based on coding theory Robert J. McEliece. Despite the name, Classic McEliece is built on top of the Niederreiter cryptosystem.

Niederreiter, as well as the original McEliece, was designed to be secure against a one-way chosen-plaintext attack (OW-CPA). This means that it is infeasible to recover the complete plaintext of a ciphertext given that the public key is available to an attacker. Classic McEliece is presented as a key encapsulation mechanism (KEM) indistinguishably under adaptive chosen ciphertext attack (IND-CCA2) which is an interactive form of chosen-ciphertext attack (CCA). A KEM is used to establish symmetric keys in a secure way using a public-key cryptosystem. Public-key schemes are in general inefficient when it comes to encrypting large amounts of data. Thus, PKC's are mostly used to transmit short messages, like a symmetric

sessions key which can be used to efficiently encrypt/decrypt large amounts of data [19].

Under the settings of IND-CCA2, an attacker can initially decrypt arbitrarily chosen ciphertexts. Eventually, the attacker sends two plaintexts m_0 and m_1 to the cryptosystem. The cryptosystem then arbitrarily selects $b = 0$ or $b = 1$ and encrypts a message m_b and returns the ciphertext c_b. The attacker can then again decrypt arbitrarily chosen ciphertexts, except the received ciphertext c_b. For a system to be IND-CCA2 secure, an attacker should not be able to determine b at a higher probability that could be achieved by randomly guessing the value of b.

As this thesis focus on the Niederreiter decryption algorithm used in Classic McEliece, the way IND-CCA2 is achieved will not be discussed further. However, details can be found in [1]. The submission of Classic McEliece consists of both a software and a hardware implementation but this thesis solely focuses on the FPGA hardware implementation.

2.5 Field programmable gate array

An FPGA is an integrated circuit used to implement digital designs. The key benefit of using an FPGA is that the design implemented on it can be reconfigured. Thus, FPGAs are commonly used during prototyping. An FPGA is also rather cheap compared to manufacturing an application-specific integrated circuit (ASIC). Thus for low-volume production, it is more economical to use an FPGA over an ASIC.

Figure 2.1 shows a simplified picture of the internals of an FPGA. It mainly consists of three parts; configurable logic block (CLB), programmable interconnect, and input-output block (IOB). A CLB consists of different logic resources like flip-flops, look-up tables, and shift registers. CLBs are used to implement logical functions of a digital design. IOBs are the interface between logic inside the FPGA and circuits outside of the FPGA. Commonly IOBs can be configured according to different logic standards depending on application needs. The actual connection between the FPGA and other parts of a circuit is accomplished by using copper traces on a printed circuit board (PCB) onto which electronic components are soldered. The programmable interconnects, routed between CLBs and IOBs, are used to connect logic functions and to connect external signals to the internal logic of the FPGA through IOBs.

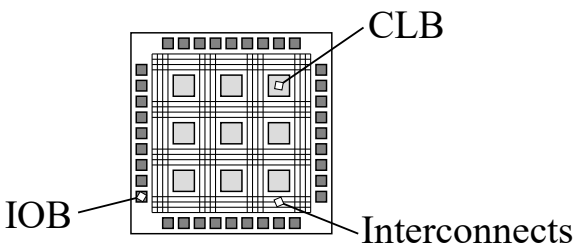

Figure 2.1: Simplified internals of an FPGA consisting of CLBs, IOBs, and programmable interconnect.

As logic functions inside the FPGA are built out of transistors, the power consumption of an FPGA can be divided into static and dynamic power consumption. Static power consumption occurs all the time and is caused by leakage in transistors. Static power highly depends on temperature, manufacture variations, and the logic state of the design, i.e. number of signals assigned a high or low value. Dynamic power consumption mainly occurs when values stored in transistors changes value. Thus, this part of the power consumption is highly dependent on the processed data. According to experiments done in [17], the static power was estimated to constitute 5-20% of the total power consumption.

2.6 Side-channel attack

During SCAs information about a cryptographic device is gathered while the device performs some cryptographic operations, like decryption of ciphertext. The goal of an attacker is to get information about the internal state of a cryptographic algorithm such that an attacker can either retrieve an encrypted message or acquire the secret key used by the cryptographic algorithm.

An SCA can be classified as either passive or active [10]. In the former, an attacker can manipulate inputs and outputs of a device as well as the environment, such as temperature or supply voltage. Typically, the attacker strives to create conditions where the device starts to operate abnormally and might expose sensitive information. In the case of a passive attack, the device is operated as intended and the attacker can just observe the behavior of a device while it performs cryptographic operations.

SCAs are also classified as either invasive or non-invasive [10]. During an invasive attack, there are no limits on what can be done to retrieve sensitive information. For example, a cryptographic chip, i.e. an integrated circuit (IC), can be de-packaged and passivation layers removed such that the bare IC becomes accessible for measurements in a probe station. In the case of a non-invasive attack, only the intended interface of a cryptographic device is accessible for measurements.

The channel used to gather information about a cryptographic device can be of many types. Lately, timing, power, and electromagnetic side channels have gained a lot of interest. A timing attack consists of observing the execution time of an algorithm and then inferring something about secrets based on the measured time. A power attack is based on observing the power consumption of a device and then inferring something about the internal state of the device based on measured power consumption. During an EM attack, the emission of EM radiation, typically caused by transistor switching, is measured. For the rest of this thesis, power will be employed as the side-channel of interest. Other side channels could be used similarly. But, with a different measurement technique.

A power attack is typically performed by placing a resistor in series with the power line of the target device. Then, the power consumption can be indirectly recorded by measuring the voltage across the resistor and making use of Ohm's law. The data obtained by measuring the power consumption during a cryptographic operation is commonly called a trace. To successfully conduct an attack many traces are required with different inputs patterns to the device. A critical step

before performing any analysis of captured traces is to make sure that they are well aligned. Such that power consumption of the same operation, with different inputs, can be compared.

There exist many methods to conduct side-channels attacks. During a SPA captured traces are visually inspected to find obvious patterns of the consumption which are directly related to secrets of the device. There exist statistical methods like DPA during which a hypothetical model of a device's power consumption is compared to the measured power. The power model should be a function that computes an intermediate value of the cryptographic algorithm based on a small part of the secret key and some other input, like plaintext, ciphertext, or a previous intermediate value. The benefit of DPA is that it typically does not require detailed information about the cryptographic device. But, a large number of traces are needed to evaluate the dependency between processed data and power consumption.

2.6.1 Neural network

In recent years, deep learning algorithms such as multiple layer perceptron (MLP) network or convolutional neural network (CNN), have been used during SCA with performance comparable to existing state-of-the-art SCAs [4].

An MLP consists of j layers. The first layer is called the input layer, the last is called the output layer and the layers in between are called hidden layers. Each layer consists of i_k, $k = 0, 1, \ldots j$ nodes called neurons which are connected to neurons in the previous layer in a certain way. In this thesis, only fully connected layers are considered where each neuron is connected to all neurons in the previous layer as shown in figure 2.2.

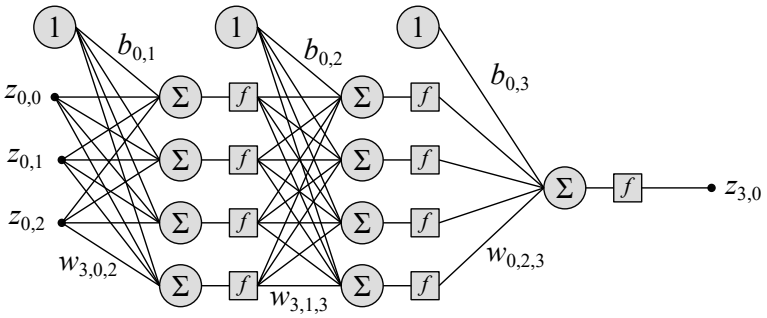

Figure 2.2: A fully connected neural network with a single hidden layer and a single output, $j = 3$ and $i_k = \{3, 4, 4, 1\}$.

The output $z_{m,n}$ of neuron m in layer n is described by

$$z_{m,n} = f\left(b_{m,n} + \sum_{p=0}^{i_{n-1}-1} z_{p,n-1} w_{m,n,p}\right) \tag{2.4}$$

where $w_{m,n,p}$ is the weight of the connection between neuron m in layer n and neuron p in layer $n-1$, $b_{m,n}$ is the bias input to neuron m in layer n, f is a

function commonly called activation function, $z(0, q)$, $q = 0, \ldots, i_0$ are input the to the network and $z(j, r)$, $r = 0, \cdots, i_j$ are the output of the network.

Before training a neural network, all biases and weights are initialized according to some scheme, like picking values from a standard normal distribution. Training of the network is then performed by calculating output values when labeled input vectors are propagated through the network. The error made by the network is calculated by comparing the predicted labels $\hat{y}_i$ with the known labels y_i using a chosen loss function L. Starting at the output layer and going towards the input layer, the loss contribution of each neuron is evaluated. Subsequently, an optimization algorithm is used to tune all biases and weights to reduce the loss, i.e. making better predictions. Then, a new set of inputs are propagated through the network, the loss is calculated and parameters are tuned. The number of training samples used during each pass is called the batch size and when all training samples have been propagated an epoch is completed. The complete procedure, except initialization of biases and weight, is then repeated for a selected number of epochs.

Different loss functions L are used depending on the prediction task. If a network should perform a binary classification task, the binary cross-entropy loss function

$$L = -\frac{1}{N_B} \sum_{i=1}^{N_B} \Big(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i) \Big) \tag{2.5}$$

is commonly used where y_i is the true class, $\hat{y}_i$ is the predicted class of the network, and N_B is the batch size. To use the binary cross-entropy loss, the activation function of the output layer must generate values between 0 and 1. A common activation function used for this purpose is the sigmoid function

$$f_{sigmoid}(x) = \frac{1}{1 + e^{-x}}. \tag{2.6}$$

For hidden layers, the rectified linear unit (ReLU) activation function

$$f_{relu}(x) = \max(0, x) \tag{2.7}$$

is commonly used as it is fast to compute.

There exist several optimization algorithms to use for updating parameters after each batch of training. A widely used optimizer is the adaptive moment estimation (ADAM) which is an extension of stochastic gradient descent. It is used to estimate how biases and weights should be adjusted to reduce the loss L. An important parameter of ADAM is the learning rate which controls how far to move in the direction of minimizing the loss function; selecting a too high value might lead to missing a local minimum but picking a too low value requires more training rounds and thus longer computation time.

All in all, a neural network has many parameters that need to be tuned to achieve good performance. Typically, one makes use of a tuning algorithm that tests combinations of different parameters. The amount of combinations that can be tested depends on available computational power and time.

2.7 Leakage assessment

To successfully conduct an SCA against a cryptographic device, the device must of course leak sensitive information in some side-channel. During the NIST non-invasive attack testing workshop in 2011, Welch's t-test was suggested to be used for leakage assessment [6]. This method has been used in several works to determine if and where a cryptographic algorithm leaks sensitive information [3][5]. The purpose of using Welch's t-test is to determine if samples picked from two sets are from the same population or not. When performing leakage assessment traces are split into two sets depending on some feature. The test is then used to determine if traces of the two sets have the same mean, i.e. they come from the same population. If traces from the two sets do not have equal mean the device is considered to leakage information.

Welch's t-test can be conducted in many ways. The test can be either univariate or multivariate. In the former, tests are performed individually at each sample point whereas. In the multivariate case, leakage from different sample points is first weighted with some function and then the t-test is performed on the weighted result. In [16], the authors suggest using a univariate test for hardware implementations as computations with secret shares are conducted in parallel and the leakage at a given point is the sum of all individual computations.

The t-test can be either specific or non-specific. In the former case, the classification of sets is based on some intermediate value during the cryptographic process. However, as there might be many distinct intermediate values many tests need to be performed before one can conclude if samples are from the same population or not. In the case of a non-specific test, the classification of sets is instead based on the input pattern to the cryptographic device. The fixed-vs-random test is an example of a non-specific test where the first set consists of traces from a fixed input pattern and the second set consists of traces from random input patterns.

To perform a univariate t-test, captured traces are first assigned to one of two sets $\mathcal{Q}_0$ or $\mathcal{Q}_1$. The test statistic t is then calculated as

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}} \tag{2.8}$$

where μ_i, s_i and n_i are the mean, standard deviation, and size of each set $\mathcal{Q}_i$, $i = 0, 1$. Next, the degree of freedom v is calculated as

$$v = \frac{\left(\frac{s_0^2}{n_0} + \frac{s_1^2}{n1}\right)^2}{\frac{\left(\frac{s_0^2}{n_0}\right)^2}{n_0 - 1} + \frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1}}. \tag{2.9}$$

However, in case $s_0 \approx s_1$ and $n_0 \approx n_1$ the degree of freedom can be estimated as

$$v \approx n_0 + n_1. \tag{2.10}$$

The test statistic t and degree of freedom v is then used to construct a Student's t-distribution

$$f(t, v) = \frac{\Gamma(\frac{v+1}{2})}{\sqrt{\pi v}\Gamma(\frac{v}{2})}\left(1 + \frac{t^2}{v}\right)^{-\frac{v+1}{2}} \tag{2.11}$$

where Γ is the gamma function. The p-value of the two-tailored Welsh's t-test is given by

$$p = 2 \int_{|t|}^{\infty} f(t, v) \mathrm{d}t \tag{2.12}$$

where a small p gives evidence to reject the null hypotheses, i.e. the two sets $\mathcal{Q}_0$ and $\mathcal{Q}_1$ do not have the same mean value.

In literature related to side-channel attacks, a threshold of $|t| > 4.5$ is commonly used to decide if the null hypothesis should be rejected [16]. The rationale behind this is that

$$p = 2 \int_{|t|}^{\infty} f(|t| \geq 4.5, v) \mathrm{d}t < 0.00001 \tag{2.13}$$

if $v > 1000$ which gives a confidence level of 99.999% to reject the null hypothesis. Thus, the computation time of Welsh's t-test can be reduced by performing the test based on the test statistic t in equation 2.8 as long as $s_0 \approx s_1$ and $n_0 \approx n_1 \approx 500$. In this case, the null hypothesis is rejected if $|t| > 4.5$ at a confidence level of 99.999 %.

Chapter 3

Classic McEliece for ChipWhisperer platform

3.1 Implementation of hardware modules

In this section, a high-level overview of the hardware system is given. The official hardware implementation of Classic McEliece is investigated in terms of operation, execution time, and signal interface. A description of how the hardware decryption core of Classic McEliece can be integrated into the CW system is also given in this section.

3.1.1 Overview

To collect traces an ADC is needed and commonly oscilloscopes are used for this purpose. Additionally, an amplifier might be used to increase the magnitude of measured traces. The reason for using an amplifier stems from the way power consumption is measured. Typically, a resistor is placed in series with the power line to a device. According to Ohm's law, a larger resistor will have a larger voltage drop compared to a small resistor for a given current. However, a large voltage drop across the resistor is unwanted as this might cause the device to malfunction due to insufficient supply voltage. Thus, commonly a small resistor in combination with an amplifier is used. Instead of using a standalone oscilloscope and amplifier, this project makes use of CW, a commercially available low-cost solution for SCA. A benefit of this is that the results of this work can be made available to a broader community.

At its core, CW consists of an amplifier and a 10-bit ADC as shown in figure 3.1. One of the key features of CW is that the ADC sampling can be driven by the same clock as used by the cryptographic target device. Furthermore, the ADC sampling can be triggered by a signal from the target device. Thus, sample points get well-aligned with operations performed by the target device. Since capturing multiple traces can be triggered and driven by the same clock, all traces will be well-aligned which eliminates the need for resynchronizing traces.

Besides features of sample synchronized traces, the CW also comes with a USB interface which can be used both for controlling CW and transferring captured traces to a PC. The CW is controlled through an open-source application programming interface (API) written in Python.

The company suppling CW has also developed several add-on boards which

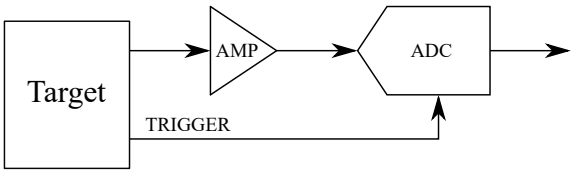

Figure 3.1: Simplified block diagram of CW, consisting of an ADC and amplifier, connected to a target device.

are equipped with different micro-controllers or FPGAs. All add-on boards are specially designed to ease the process of performing SCA. Most of the add-on boards can be programmed through CW to test different cryptographic algorithms. As this work focus on SCA of a hardware implementation, the add-on board CW305 from NewAE was chosen as a platform for development. The CW305 is a more complex add-on board compared to other micro-controller add-on boards from NewAE. The CW305 contains an Artix7 FPGA, USB transceiver, power supply filtering, and an amplifier for measuring the power consumption of the FPGA. It can be controlled through the same Python API as CW and some open-source implementations are available, such as a 128-bit AES core.

Figure 3.2 shows a high-level block diagram of how the CW, CW305, and a PC are connected to form the complete SCA system. The system operates as follows. First, the PC connects to CW. Then, the PC programs the CW305 and loads the secret key and the ciphertext. Then, the PC issues a start command and CW305 starts to decrypt the ciphertext. At the same time, the CW starts to capture a power trace of the CW305 FPGA. Lastly, the PC reads the captured trace from CW and performs further operations to carry out an SCA.

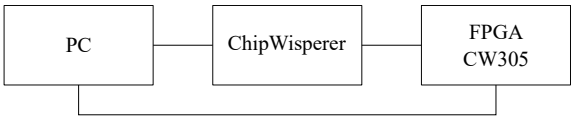

Figure 3.2: System overview consisting of a PC, CW, and the add-on board CW305.

To get a fully functional system a couple of tasks were conducted. The reference decryption implementation of Classic McEliece was ported to a Xilinx Artix7 FPGA and an interface between the FPGA and PC was developed. For the latter, an open-source implementation of a 128-bit AES was used as a starting point. Figure 3.3 shows a block diagram of the developed FPGA implementation consisting of three blocks. The *USB Interface* communicates with the USB transceiver of CW305 which in turn communicates with a PC. The USB transceiver issues either a read or write to the *USB Interface* which decodes where data should be read or written and generates an address accordingly. Actual data is sent in chunks of 8 bits between the transceiver and interface. Subsequently, the *Decryption Register* controls when and where data should be stored or read depending on signals from the *USB Interface*. The *Decryption Register* also contains logic for cross-clock

domain synchronization. Finally, the *Decryption Module* contains the reference decryption implementation of Classic McEliece, i.e. the Niederreiter decryption core. The *Decryption Module* also contains memories for storing the secret keys, ciphertext, and recovered plaintext.

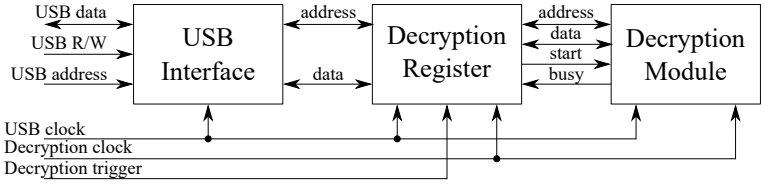

Figure 3.3: Overview of FPGA implementations consisting of three modules.

3.1.2 Decryption core

The submission of Classic McEliece to the NIST PQC competition consists of both a software and hardware implementation. However, the hardware implementation only consists of modules for key generation, encryption, and decryption. Details of these modules are given in [22]. The three modules are configured by a set of system parameters where the three most important are the field size m, code length n, and the number of correctable errors t. As this thesis focus on decryption, the key generation and encryption modules will not be discussed further. Information about these modules is given in [22] and [21].

In figure 3.4 a block diagram of the decryption core is given. It consists of the four blocks *Additive FFT*, *Double Syndrome* constructor, *Berlekamp-Massey* decoder, and *Error Locater*.

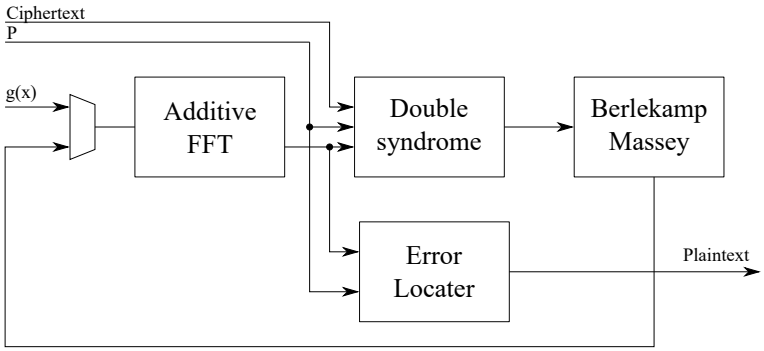

Figure 3.4: Overview of decryption core used in Classic McEliece.

The decryption algorithm consists of five steps. First, the *Additive FFT* polynomial evaluator is used to evaluate the secret Goppa polynomial $g(x)$ for all elements in the field $\mathrm{GF}(2^m)$. The output of this step only depends on the secret Goppa polynomial and the execution time is constant. In the second step, a double-size syndrome $\mathbf{H}^{(2)}$ is calculated as $\mathbf{S}^{(2)} = \mathbf{H}^{(2)} \times (\mathbf{c} \mid \mathbf{0})$ where $\mathbf{c}$ is the

Parameter	Description
m	Field size, GF(2^m)
n	Code length
t	Number of correctable errors
Block	Design parameter for *Double Syndrome*
Sec	Design parameter for *Additive FFT*
Factor	Design parameter for *Additive FFT*
MulSecBM	Design parameter for *Berlekamp-Massey*
MulSecBMStep	Design parameter for *Berlekamp-Massey*

Table 3.1: Crypto and design parameters of decryption core.

input ciphertext. The parity check matrix $\mathbf{H}$ is first constructed by reading the secret support $P = \{\alpha_0, \alpha_1, \ldots \alpha_{n-1}\}$ from external memory. Support points are read one at a time but α_i is only read if the corresponding bit in the ciphertext is a high, i.e. $c_i = 1$. Thus, the execution time of the *Double Syndrome* depends on the number of 1's in the ciphertext. During step three, the $\mathbf{S}^{(2)}$ is read by a constant-time Berlekamp-Massey (BM) decoder to produce an error locater polynomial (ELP). At step four, the same *Additive FFT* module as in step one is reused to evaluate the ELP at all elements of GF(2^m). Lastly, in step five, bits of the plaintext are constructed by making use of the secret support P and the evaluated ELP.

The authors of [22] made the Niederreiter hardware modules available as an open-source build system. To create hardware modules, the build system was installed on a Linux computer along with SageMath[1]. The Niederreiter build system consists of multiple SageMath and Python scripts that based on user-selected design parameters generate Verilog[2] source files. In the build system, SageMath is mainly used to carry out calculations over finite fields.

To build source files for the decryption module, all seven design parameters in table 3.1 must be set. The three first parameters m, n, and t are related to setting the security level of the system. According to [1] using the parameter set `kem/mceliece348864` with $m = 12$, $n = 3488$ and $t = 64$ corresponds to a security level equivalent to exhaustive key search of AES-128. Other parameters of table 3.1 are related to optimization of the hardware implementation where a user can make a trade-off between usage of logic resources (area) and speed (number of required clock cycles). Table 3.2 shows all chosen parameters used in this work. For further information on design parameters, the reader is referred to [22].

[1] SageMath is an open source mathematics software built on top of many other open source packages. The key feature of SageMath is that it gives access to all of these packages through a common Python based language

[2] Verilog is a hardware description language used to synthesize hardware in FPGAs or ASICs.

Parameter	Used value
m	12
n	3488
t	64
Block	20
Sec	4
Factor	0
MulSecBM	20
MulSecBMStep	20

Table 3.2: Crypto and design parameters used when building the decryption core.

3.1.3 Analysis of reference implementation

Figure 3.5 shows the hardware interface of the generated decryption core by using parameters of table 3.2. The interface consists of five input and four output signals. The operation of the core is driven by a clock signal *clk*. Decryption is started by setting *start* to a logic high level for one clock cycle. The secret Goppa polynomial is supplied through the signal *poly_g* as a binary vector. For the secret support P, a memory interface is used consisting of the signals *P_rd_en*, *P_rd_addr*, and *P_out*. When *P_rd_en* is set to a logic high, the memory row pointed by *P_rd_addr* is expected to be available at *P_out* after one clock cycle. When decryption is finished, the signal *done* is set high for one clock cycle and the recovered plaintext becomes available through the *error_recovered* signal as a binary vector.

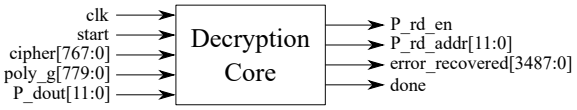

Figure 3.5: Implemented decryption core interface.

The secret Goppa polynomial should be of degree $t = 64$ over the field $\mathrm{GF}(2^m) = \mathrm{GF}(2^{12})$. Therefore, $t+1 = 64+1 = 65$ coefficients g_i are needed to represent this polynomial. Since the polynomial is over $\mathrm{GF}(2^{12})$, 12 bits are needed to encode each coefficient a_{i_j} of g_i. Thus, the width of *poly_g* is $(t+1)\cdot m = (64+1)\cdot 12 = 780$ bits. Figure 3.6 shows how bits of *poly_g* are organized where each g_i is the 12-bit coefficient of x^i of the Goppa polynomial. The decryption core expects to read a monic polynomial. Thus, the x^{64}-coefficient g_{64} must be equal to 1 as shown in figure 3.6.

The secret support P is accessed by the decryption core through a memory interface. The secret support consists of $n = 3488$ distinct points of the field $\mathrm{GF}(2^m) = \mathrm{GF}(2^{12})$. Thus, each point is encoded by 12 bits. The decryption core expects that each point is stored as a row in a memory with the first point at the first row as depicted in figure 3.7. Bits of each point should be arranged in the

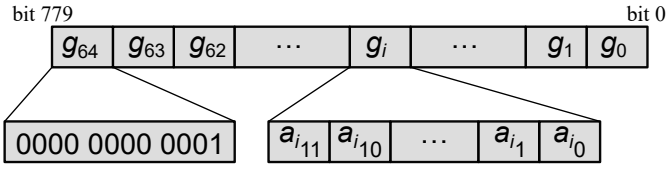

Figure 3.6: Organization of bits in the signal *poly_g*.

same way as the secret polynomial, i.e. the least significant bit at the lowest bit position in the memory.

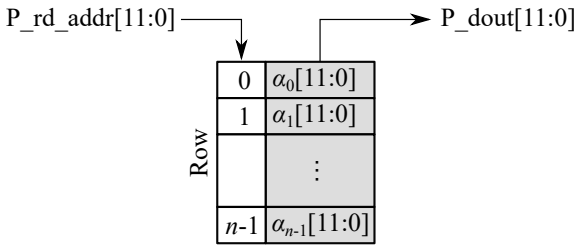

Figure 3.7: Organization of bits in the secret support memory.

Bit orientation of *error_recovered* is not important as long as the original plaintext and the recovered plaintext are interpreted in the same way. The ciphertext $\mathbf{c}$ is generated by multiplying the plaintext $\mathbf{p}$ with the public key $\mathbf{c} = \mathbf{H}_{\text{pub}}\mathbf{p}$, whereas $\mathbf{H}_{\text{pub}}$ is a binary $mt \times n$ matrix and the plaintext $\mathbf{p}$ is an n-bit column vector. Thus, $\mathbf{c}$ will be a column vector of length $mt = 12 \cdot 64 = 768$. The decryption expects that these bits are organized as shown in figure 3.8, where the lowest bit corresponds to the multiplication between the first row of $\mathbf{H}_{pub}$ and $\mathbf{p}$. All multiplications are carried out in GF(2). Thus, the result is either 0 or 1.

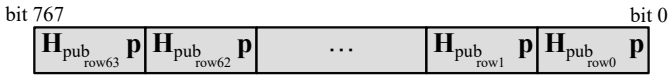

Figure 3.8: Organization of bits of the ciphertext.

After source files of the decryption core were generated, the design was inspected and simulated in Vivado[3]. In particular, the execution time in terms of clocks cycles was studied. It was concluded that for all decryption steps except the double syndrome calculation, the execution times were fixed and did not depend on inputs. Table 3.3 shows the number of clock cycles needed for each of the four constant decryption steps. It should be noted that even though the number of clocks cycles in table 3.3 does not depend on inputs, they do depend on design parameters. Thus, values given in the table are only valid when the design parameters of table 3.2 are used.

[3]Vivado is a tool used to design and implement digital circuits in Xilinx's FPGAs

Step	Clock cycles
Goppa polynomial evaluation (additive FFT)	1095
BM decoder	1921
ELP evaluation (additive FFT)	1095
Error locater	3498

Table 3.3: Clock cycles for each of the four constant steps of decryption.

The number of clock cycles for the double syndrome calculation was derived as

$$\textit{Double syndrome clks} = 14 + \lceil \#(\textit{ones in ciphertext})/20 \rceil \cdot 130 + A \tag{3.1}$$

through analysis of the source code, where A is the number of bits in the ciphertext until 20 1's are found counting from most towards the least significant bit of the ciphertext. Thus, the execution time of the double syndrome as well as the whole decryption depends on how bits are distributed in the ciphertext. This in turn depends on the plaintext and public key used during encryption. The validity of expressions 3.1 was also verified by simulations in Vivado.

3.1.4 Interfacing the decryption core

As the decryption core should communicate with a PC, a serial-to-parallel interface was needed in between due to the wide signals of the decryption core. Since the CW305 board includes a USB transceiver, this interface was used for communication between the decryption core and PC. Furthermore, a hardware driver for sending and receiving data between the FPGA and USB transceiver of the CW305 was available as part of a 128-bit AES implementation [4]. Figure 3.9 shows a simplified block diagram of parts used for USB communication. The PC either send

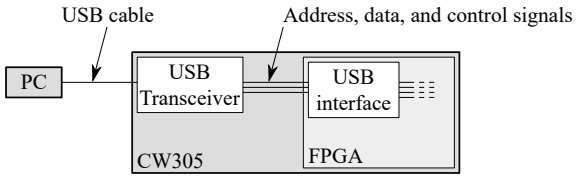

Figure 3.9: Block diagram of the interface between PC and FPGA of CW305

a read or write request to the USB transceiver. In the case of a write operation, the address to write and data are decoded from the payload. The address and data along with the control signal (read-write signal and USB clock) are routed to the USB interface implemented in the FPGA, which stores data in a register pointed by the address. During a read request, the USB interface fetches data

[4] `https://github.com/newaetech/chipwhisperer`

from a pointed register and sends it to the USB transceiver, which in turn sends data to the PC using USB.

Figure 3.10 shows the payload of a USB transaction. The part *dlen* specifies how many bytes to either write or read at the register pointed by *reg*. In the case of a read operation, the payload only consists of the first two fields *dlen* and *addr*. Both *dlen* and *addr* are 32 bits wide and transmitted as the least significant byte first. However, due to hardware limitations, the accessible address space is limited to 21-bits. The value specified in *addr* decides where the first byte, $byte_0$, should be stored. Following bytes are stored at ascending address locations. Thus, it is important to have an agreement between software and hardware on how registers are accessed.

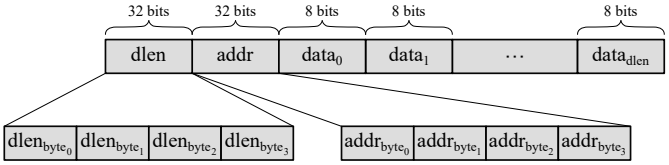

Figure 3.10: USB payload.

The AES example code provided by NewAE divides the *addr* part of the payload into two parts as shown in figure 3.11. The first part *reg_ addr* is used to

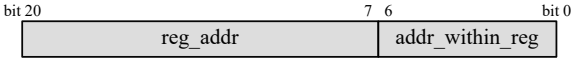

Figure 3.11: USB AES address format.

access different registers in the AES example FPGA implementation according to the upper part of table 3.4. The lowest 8 bits *addr_ within_ reg* are always written as zeros as the USB transceiver generates this part. For example, if $dlen = 3$ and $addr = 01_{16} = 00000001_2$, the USB transceiver decodes the address as 1000000_2 for the first byte of the payload, 1000001_2 for the second byte, and 1000010_2. Thus, the maximum size of a register is $2^7 = 128$ bytes. This is too small for the implemented decryption core as the largest key, the secret support has a size of 41856 bits. Thus the address format was changed to allow a maximum register size of $2^{13} = 8192$ bytes.

Figure 3.12 shows the new address format used for communication between the PC and decryption core. The first field selects which memory to interact with. The second field specifies which bank within the memory to access. The last field is used by the USB transceiver to access individual bytes within a bank. The reason

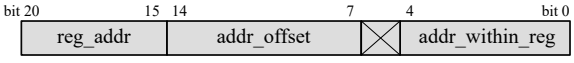

Figure 3.12: USB address format fo implemented decryption core.

of selecting this address format is based on requirements of writing to the largest memory, the secret support, in the implemented design. If the number of bytes in

Name	Register address	Comment
REG_CLKSETTINGS	0x00	
REG_USER_LED	0x01	
REG_CRYPT_TYPE	0x02	
REG_CRYPT_REV	0x03	
REG_IDENTIFY	0x04	
REG_CRYPT_GO	0x05	
REG_CRYPT_TEXTIN	0x06	Removed in this work
REG_CRYPT_CIPHERIN	0x07	Removed in this work
REG_CRYPT_TEXTOUT	0x08	Removed in this work
REG_CRYPT_CIPHEROUT	0x09	Removed in this work
REG_CRYPT_KEY	0x0A	Removed in this work
REG_BUILDTIME	0x0B	
REG_P_MATRIX_IN	0x0C	Added in this work
REG_POLY_G_IN	0x0D	Added in this work
REG_CIPHER_IN	0x0E	Added in this work
REG_REC_ERR_OUT	0x0F	Added in this work

Table 3.4: Register addresses.

each transfer is limited to 32 bytes, a total of $(2 \cdot 3488)/32 = 218$ transactions are needed to write the secret support. Thus, $\lceil \log_2 218 \rceil = 8$ bits are needed for base register address offset, and $\lceil \log_2 32 \rceil = 5$ bits are needed to address memory rows within each transaction. Conceptually, each memory is divided into banks of 32 bytes and during a USB operation, a complete bank is written. Additionally, four new registers were added and five removed as depicted in table 3.4 to better suit the implemented decryption core.

3.1.5 Storing decryption inputs

To store the ciphertext and secret key in the FPGA, three memories were implemented as shown in figure 3.13. Furthermore, the memories were implemented by using available RAM blocks in the FPGA as using distributed RAM (flip-flops and look-up tables) would result in a high resources utilization of the FPGA. This in turn would make it harder and maybe impossible for the synthesis tool to implement the complete design. Figure 3.14 shows the memory used for storing the secret Goppa polynomial. It consists of four dual-port memories which are only partially used. Port one of the memories is of size 8 bit $\times$ 32 but only the first 16 rows are used. The reason for this is that the decryption core needs to read all bits of secret polynomial during one clock cycle and the core does not have the functionality to address the memories. Therefore, port 2 of the memories is of size 256 bit $\times$ 2, but the address is fixed to row 0. Thus, each dual-port memory can store 256 bits and since the secret polynomial consists of 780 bits, four dual-port memories were connected in parallel.

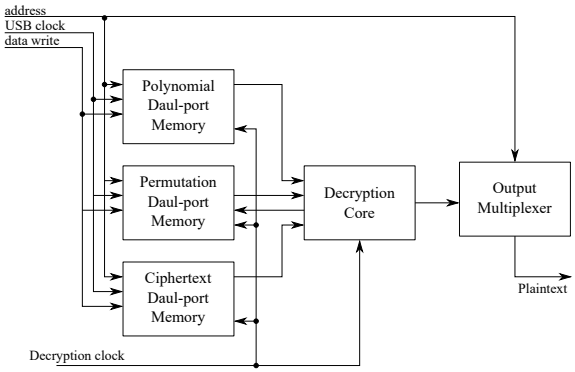

Figure 3.13: Overview of the decryption module consisting of the decryption core, three memories, and a multiplexer.

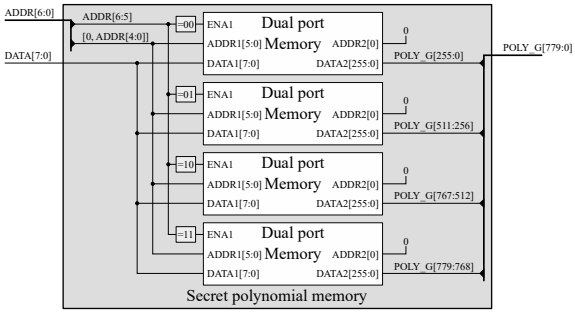

Figure 3.14: Memory for storing secret Goppa polynomial.

The complete ciphertext should also be available for the decryption core to read during a single clock cycle. Thus, the ciphertext was stored in a memory similar to the secret Goppa polynomial. Figure 3.15 shows the implemented memory which in comparison to the Goppa polynomial memory only contains three dual-port memories since the cipher text only consists of 768 bits.

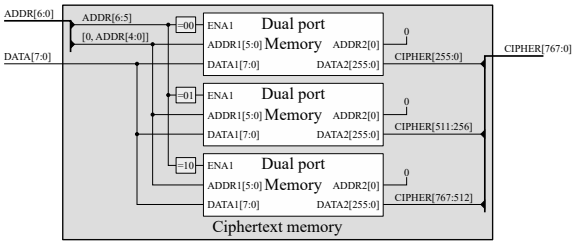

Figure 3.15: Memory for storing ciphertext.

Storing the secret support was done similarly, but since the decryption core utilizes a memory interface for reading the support row by row, the address of port 2 is connected to the decryption core rather than to a fixed value. Since data at port 1 is written 8 bits at a time and the decryption core expects to read 12 bits, 2 dual-port memories were connected in parallel as shown in figure 3.16. The first memory stored the 8 lowest bits of each field point and the second memory stored the upper 4 bits.

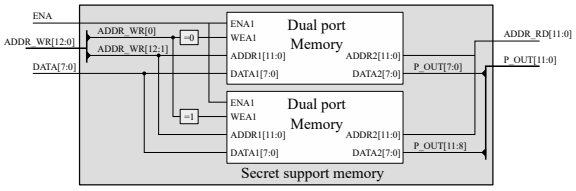

Figure 3.16: Memory for storing secret support.

Once decryption is done, the recovered error is stored in 3488 flip-flops inside the decryption core. A 3488-to-8 bit multiplexer was then used to read the recovered plaintext by 8 bits at a time.

3.2 Software modules

In this section, four different software modules developed during this project are presented. First, a module for generating the secret and public keys is explained. A module for collecting traces is introduced and also a module for performing leakage assessment is given. Last, the idea of a partial message-recovery attack is presented.

3.2.1 Classic McEliece Python module

To perform decryption, a plaintext must first be encrypted. To support this, a McEliece Python module was developed. The module contains functions for generating private and public keys as well as generating random plaintexts and encrypting plaintexts. The source code of this module is available in Appendix A.

Key generation starts with finding a random irreducible polynomial $g(x)$ of degree t. This was done similarly as in [21]. First, a random polynomial $h(x)$ over GF(2^m) with $deg(h) = t$ was selected. This polynomial was used to create the field GF(2^m)$[x]/h(x)$ from which a random element $r(x)$ of degree $t-1$ was selected. The monic polynomial of least degree that fulfills $g(r(x)) = 0$ will be a minimal, i.e. irreducible, polynomial with coefficients in GF(2^m). Since $r(x)$ is of degree $t-1$, the minimal polynomial $g(x)$ will be of degree t if it exists. To find the minimal polynomial $g(x)$ of $r(x)$ suppose

$$g(x) = g_0 + g_1 x^1 + \dots g_{t-1} x^{t-1} + x^t \tag{3.2}$$

and

$$r(x) = r_0 + r_1 x^1 + \cdots + r_{t-1} x^{t-1} \tag{3.3}$$

where $g(x)$ is in monic form since $g_t = 1$. Then, the minimal polynomial of $r(x)$ must fulfill

$$g(r(x)) = g_0 + g_1 r(x) + g_2 r^2(x) + \cdots + g_{t-1} r^{t-1}(x) + r^t(x) = 0. \tag{3.4}$$

From equation 3.4 a system of t linear equations can be formed as

$$\begin{cases} (r)_{t-1}g_1 + (r^2)_{t-1}g_2 + \dots + (r^{t-1})_{t-1}g_{t-1} = (r^t)_{t-1} \\ (r)_{t-2}g_1 + (r^2)_{t-2}g_2 + \dots + (r^{t-1})_{t-2}g_{t-1} = (r^t)_{t-2} \\ \vdots \quad\quad \vdots \quad\quad \ddots \quad\quad \vdots \quad\quad \vdots \\ (r)_1 g_1 + (r^2)_1 g_2 + \dots + (r^{t-1})_1 g_{t-1} = (r^t)_1 \\ g_0 + (r)_0 g_1 + (r^2)_0 g_2 + \dots + (r^{t-1})_0 g_{t-1} = (r^t)_0 \end{cases} \tag{3.5}$$

where $(r^i)_j$ is the x^j coefficient after calculating $r^i(x)$ over GF(2^m)$[x]/h(x)$. By treating g_i as variables, the minimal polynomial can be obtained by solving the system. This can be done by forming the augmented matrix as

$$\left[\begin{array}{ccccc|c} 0 & (r)_{t-1} & (r^2)_{t-1} & \dots & (r^{t-1})_{t-1} & (r^t)_{t-1} \\ 0 & (r)_{t-2} & (r^2)_{t-2} & \dots & (r^{t-1})_{t-2} & (r^t)_{t-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & (r)_1 & (r^2)_1 & \dots & (r^{t-1})_1 & (r^t)_1 \\ 1 & (r)_0 & (r^2)_0 & \dots & (r^{t-1})_0 & (r^t)_0 \end{array}\right]. \tag{3.6}$$

Then, by bringing the augmented matrix to reduced row-echelon form, i.e.

$$\left[\begin{array}{cccccc|c} 1 & 0 & 0 & \dots & 0 & 0 & g_0 \\ 0 & 1 & 0 & \dots & 0 & 0 & g_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & g_{t-2} \\ 0 & 0 & 0 & \dots & 0 & 1 & g_{t-1} \end{array}\right] \tag{3.7}$$

the coefficients of $g(x)$ can be found by extracting the last column of the matrix in 3.7 and by setting $g_t = 1$.

The next step in the key generation was to construct the secret support P consisting of n distinct elements $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}) \in \mathrm{GF}(2^m)$. This was done by randomly selecting n distinct integers in the range $[0, 2^m - 1)$ and interpreting the integers as elements in $\mathrm{GF}(2^m)$.

Next, the public key $\mathbf{H}_{\mathrm{pub}}$ was constructed. First, 2^m integers were generated where each integer $k_i = i, i = 0, 1, \ldots, 2^m - 1$ was binary encoded with m-bits as $k_i = 2^{m-1}k_{i_{m-1}} + 2^{m-2}k_{i_{m-2}} + \cdots + 2^1 k_{i_1} + 2^0 k_{i_0}$, where $k_{i_{m-1}}$ was the most significant bit. The integers k_i were then used to create a list of all elements a_i of $\mathrm{GF}(2^m)$ as $a_i = k_{i_0}x^{m-1} + k_{i_1}x^{m-2} + \cdots + k_{i_{2^m-1}}x^0$. This means that each k_i was interpreted in reversed bit order. The reason for this is to follow how the hardware decryption implementation interprets integers of the secret support P. After this, all elements a_i were evaluated by the minimal polynomial $g(x)$ and the matrix

$$\mathbf{Z} = \begin{bmatrix} \frac{1}{g(a_0)} & 0 & \ldots & 0 & 0 \\ 0 & \frac{1}{g(a_1)} & \ldots & 0 & 0 \\ 0 & 0 & \frac{1}{g(a_3)} & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & 0\frac{1}{g(a_{m-1)}} \end{bmatrix} \tag{3.8}$$

was formed. If any $g(a_i) = 0$ the process was aborted and a new private key was generated. Then, another matrix was generated as

$$\mathbf{Y} = \begin{bmatrix} 1 & 1 & \ldots & 1 \\ a_0 & a_1 & \ldots & a_{n-1} \\ a_0^2 & a_1^2 & \ldots & a_{n-1}^2 \\ \vdots & \vdots & \ddots & \vdots \\ a_0^{t-1} & a_1^{t-1} & \ldots & a_{n-1}^{t-1} \end{bmatrix}. \tag{3.9}$$

A $2^m \times 2^m$ permutation matrix $\mathbf{P}_{\mathrm{perm}}$ was constructed as

$$\mathbf{P}_{\mathrm{perm}_{i,j}} = \begin{cases} 1 & , \text{if } i = P_j \\ 0 & , \text{otherwise} \end{cases}, \tag{3.10}$$

i.e. in each column j of $\mathbf{P}_{\mathrm{perm}}$ only the row given by element j in the secret support P was set to 1. Finally, the public key was computed as

$$\mathbf{H}_{\mathrm{pub}} = \mathbf{YZP}_{\mathrm{perm}} \tag{3.11}$$

Now, each element $H_{\mathrm{pub}_{i,j}}$ of $\mathbf{H}_{\mathrm{pub}}$ was an element in the field $\mathrm{GF}(2^m)$ and thus each element was represented by m-bits. The binary matrix was then constructed

by picking the first n columns of $\mathbf{H}_{\text{pub}}$ and stacking the bits as

$$\mathbf{H}_{\text{pub}_{\text{bin}}} = \begin{bmatrix} H_{\text{pub}_{0,0_0}} & H_{\text{pub}_{0,1_0}} & \cdots & H_{\text{pub}_{0,n-1_0}} \\ H_{\text{pub}_{0,0_1}} & H_{\text{pub}_{0,1_1}} & \cdots & H_{\text{pub}_{0,n-1_1}} \\ \vdots & \vdots & \ddots & \vdots \\ H_{\text{pub}_{0,0_{m-1}}} & H_{\text{pub}_{0,1_{m-1}}} & \cdots & H_{\text{pub}_{0,n-1_{m-1}}} \\ H_{\text{pub}_{1,0_1}} & H_{\text{pub}_{1,1_1}} & \cdots & H_{\text{pub}_{1,n-1_1}} \\ \vdots & \vdots & \ddots & \vdots \\ H_{\text{pub}_{t-1,0_1}} & H_{\text{pub}_{t-1,1_1}} & \cdots & H_{\text{pub}_{t-1,n-1_1}} \\ \vdots & \vdots & \ddots & \vdots \\ H_{\text{pub}_{t-1,0_{m-1}}} & H_{\text{pub}_{t-1,1_{m-1}}} & \cdots & H_{\text{pub}_{t-1,n-1_{m-1}}} \end{bmatrix}. \tag{3.12}$$

Finally, $\mathbf{H}_{\text{pub}_{\text{bin}}}$ was brought to systematic form such that the first mt columns were the identity matrix, i.e. $\mathbf{H}_{\text{pub}_{\text{sys}}} = [\mathbb{I} \mid \mathbf{K}_{\text{pub}}]$. If $\mathbf{H}_{\text{pub}_{\text{bin}}}$ could not be brought to systematic form the whole key generation process was restarted with the generation of a new private key. According to [21], the success probability of getting a public key that can be brought into systematic form is 29%. To encrypt a message the plaintext p is multiplied with $\mathbf{H}_{\text{pub}_{\text{sys}}}$ over the field GF(2), i.e. addition is performed by using the xor operator.

The implemented McEliece module also contains some support functions for storing and loading keys as well as functions for formatting the private key and ciphertext such that they can be loaded to the CW305 FPGA.

3.2.2 Trace capture

To integrate trace capturing of Classic McEliece decryption into the CW framework, the source code of the existing Python API for the AES implementation was extended. Functions for sending and receiving data to the CW305 FPGA were added according to the specification given under section 3.1.4. The added functions are found in Appendix B.

To capture traces during the decryption procedure another Python script was written which can be found in Appendix C. Figure 3.17 shows the procedure used to capture traces. First, the script connects the PC to the CW and CW305 through USB. The FPGA of CW305 was programmed with the hardware description according to section 3.1. The CW was then configured with parameters related to sampling. The sampling clock was configured to be derived from the decryption clock used by the FPGA. The sample trigger was set to be the same as the signal that started the decryption procedure. The gain $A = [0, 56]$ dB of the amplifier in front of the ADC was configured as well as the number of samples $n_s = (0, 24400]$ to capture for each trace.

The script then created a McEliece object from the module described in section 3.2.1 with $m = 12$, $t = 64$, and $n = 3488$. Public and private keys were either created or loaded. Next, the FPGA supply voltage and operating frequency were set. The secret key was also transferred to the FPGA.

Then, the number of batches and traces in each batch were configured. The

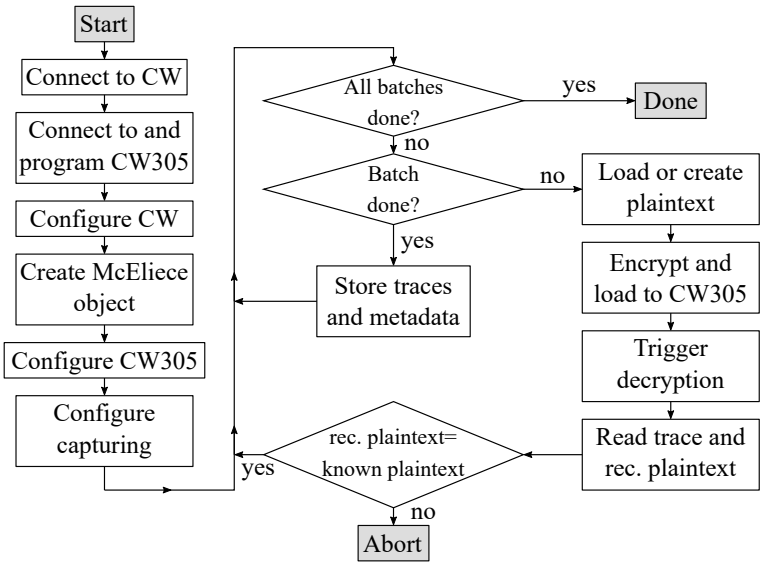

Figure 3.17: Procedure for trace capturing.

reason for dividing captured traces into batches was to reduce the amount of RAM that was needed to store variables before they were written to the hard drive.

Then, the type of plaintext generation was specified. The plaintext could be of three types. The first option was to solely generate random plaintext for each trace. The second option was to, with equal probability, select either a fix or random plaintext. The last option was to only use a fixed plaintext.

Since the CW is only capable of collecting a maximum of 244400 samples during a trace and the decryption procedure takes around 10200 clock cycles, it was only possible to have a maximum sampling rate of two times the decryption clock. To increase the maximum sampling rate another option was introduced. These options specified if the trace should be of the whole decryption procedure or only the last step, the error locater step in figure 3.4. Thus, by only sampling the last step, the sampling rate could be increased to six times the decryption clock frequency. The reason for only implementing this feature for the last step was that during the last step, the plaintext was recovered bit-by-bit and it was believed that this could be used for a plaintext-recovery attack.

Once all settings were configured, the trace capture began. For the number of specified batches and traces in each batch, a plaintext was generated depending on the selected option. The ciphertext was generated by encrypting the plaintext and then transferred to the FPGA. A start signal for decryption, as well as trace capture, was generated. Once the decryption was done, the recovered plaintext was read from the FPGA and the trace was transferred from CW to the PC. As a sanity check, the recovered plaintext was compared with the one used for generating the ciphertext. If they were unequal the capture procedure was aborted. Once a full batch of traces had been collected, all traces, as well as corresponding plaintexts, and ciphertexts were written to the hard drive.

3.2.3 Evaluation of side-channel leakage

After capturing traces, a leakage assessment was performed on collected traces according to section 2.7. As summing up a large number of values can cause numerical instability, the procedure suggested by [16] was used. First, each trace was assigned to one out of two sets, $\mathcal{Q}_i$ $i = 0, 1$, by using a function $f(plaintext)$ which outputs a 0 or 1 depending on some feature of the plaintext. Then for each set, three variables were initialized as $n_i = 0$, $\mu_i = 0, 0, \ldots, 0$ and $CS_i = 0, 0, \ldots, 0$. The first was used for counting the number of traces in each set. The second was for estimating the mean of each set. The last was for calculating the central sum which later was used to estimate the variance. Note that the two latter variables μ_i and CS_i had the same number of elements as sample points in traces. For each trace in a set $\mathcal{Q}_\rangle$ the three variables were updated as

$$n_i^+ = n_i + 1 \tag{3.13}$$

$$\mu_i^+ = \mu_i + \frac{trace - \mu_i}{n_i} \tag{3.14}$$

$$CS_i^+ = CS_i + \frac{(trace - \mu_i)^2 (n_i - 1)}{n_i} \tag{3.15}$$

where $i \in \{0, 1\}$ was assigned a value depending on the function $f(plaintext)$. After going through all traces the variance was estimated as

$$\sigma_i^2 = \frac{CS_i}{n_i} \tag{3.16}$$

and the t-statistic was calculated as given in equation 2.8.

If traces were captured for the whole decryption, the samples corresponding to the double size syndrome calculation were removed before calculating the t-statistic. The reason for this is that the double syndrome was of non-constant time. Thus, if this part was kept, subsequent samples in traces would not correspond to the same operation. Equation 3.1 was used to determine which samples in a trace that corresponds to the double syndrome.

Depending on how the selection function $f(plaintext)$ was specified, the sizes of $\mathcal{Q}_0$ and $\mathcal{Q}_1$ could become quite unequal. This, in turn, would make the simplification given equation 2.10 invalid. Therefore, after traces had been assigned to sets, the largest set was shrunk to the size of the smallest set by randomly picking traces.

3.3 Partial message-recovery attack on Classic McEliece

The hardware decryption implementation was thoroughly studied to identify possible weaknesses that could be exploited during a message-recovery attack. The last stage, step five, seemed like a promising starting point as the plaintext was reconstructed bit-by-bit during this stage. Figure 3.18 shows a simplified block diagram of how plaintext bits were reconstructed. The first element, P_OUT, of the secret support was read from the memory. The seven most significant bits of P_OUT were used as an address to read a block of 32 field elements from the ELP

evaluation performed during step four. Based on bit four of P_OUT, either the upper or lower 16 field elements were stored in a register. From the 16 stored field elements, either the upper or lower 8 field elements were stored in another register depending on bit 3 of P_OUT. This procedure is repeated until a single field element was stored in a register. If this single element was equal to all zeros the first, most significant, plaintext bit was set to a 1 otherwise it was set to a 0. The value of the plaintext bit was subsequently stored in a shift register. This whole procedure was then repeated 3488 times to reconstruct all plaintext bits. During each cycle, the recovered plaintext, stored in the shift register, was updated with a new bit at the least significant bit position.

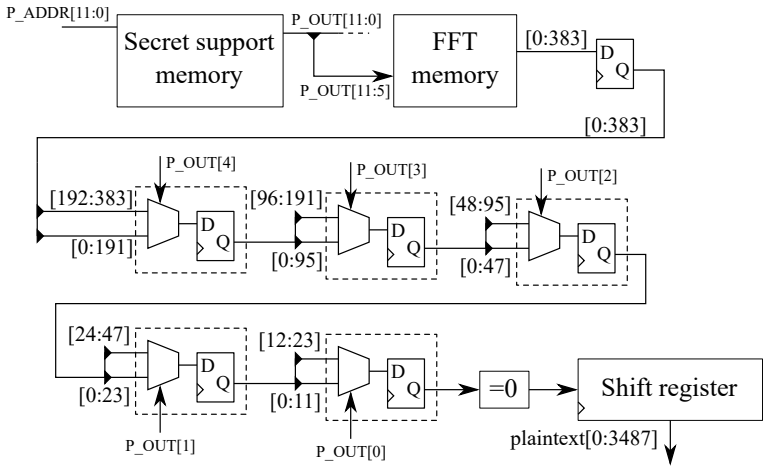

Figure 3.18: Simplified block diagram of the last decryption step.

However, the procedure for recovering plaintext bits was not done in strictly sequential order as a pipelined design was employed. Figure 3.19 shows the 8-stage pipeline used to recover plaintext bits. Thus, during each clock cycle, eight plaintext bits are evaluated in the different stages of the pipeline. This might make it harder to perform a message-recovery attack as power consumption during a single clock cycle will not solely depend on the value of one-single plaintext bit.

To retrieve information that could be used during a message-recovery attack, captured traces were divided into two classes based on the HW of the corresponding plaintext. For each trace, the HW of the plaintext was evaluated inside a window, HW window. The window was then moved one-bit position and the HW inside the window was recorded. This procedure was repeated and when a window extended beyond the last bit of the plaintext it was wrapped around to the beginning of the plaintext. Figure 3.20 shows how the HW window was swept across the plaintext. The window width was set to $m/2 = 3488/2 = 1744$ and a total of $m = 3488$ windows was used.

Thus, for each trace i a total of 3488 HWs were calculated as

$$HW_{ij} = \begin{cases} \sum_{k=j}^{j+1743} p_{ik} & , \text{ if } j <= 1744 \\ \sum_{k=0}^{j+1743 \bmod 3488} p_{ik} + \sum_{k=j}^{3487} p_{ik} & , \text{ if } j > 1744 \end{cases} \tag{3.17}$$

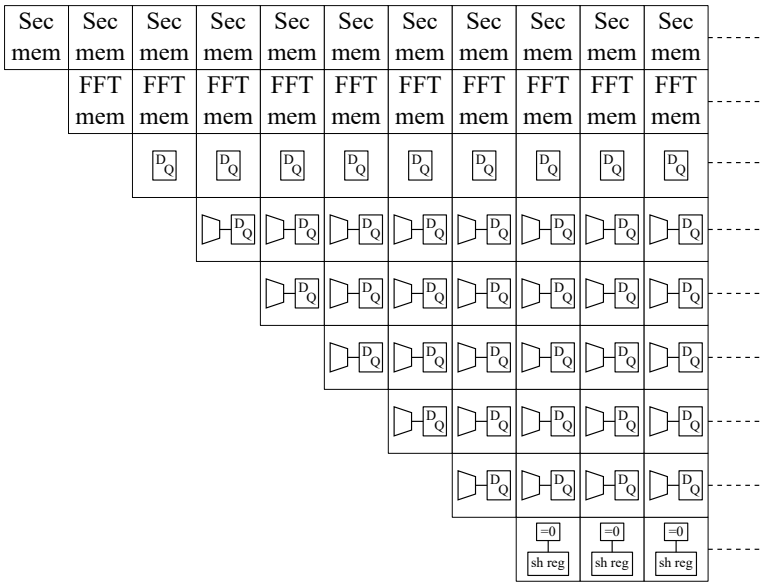

Figure 3.19: Pipeline structure used in the hardware decryption implementation during plaintext recovery.

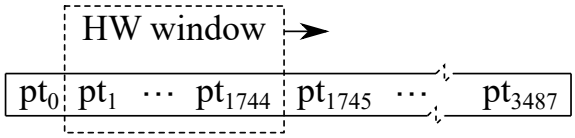

Figure 3.20: Sweeping HW window across a plaintext. The HW of the plaintext inside the windows was calculated for each window position.

Layer type	(input, output) shape	# Parameters
Dense	$(v,\ v)$	$v \cdot v + v$
Dense	$(v, v/2)$	$v \cdot v/2 + v/2$
Dense, Sigmoid	$(v/2, 1)$	$v/2 + 1$

Table 3.5: Neural network architecture used to classify traces based on HW windows.

where p_{ik} was bit k in the plaintext of trace i and j was the HW window number. For each HW window number j, all traces were divided into two classes where the first contained traces with $H_{ij} > 32$ and the second consisted of traces with $H_{ij} <= 32$. A PCA was performed on traces to reduce the dimensionality of captured traces. The number of new features to keep after the PCA was adjusted such that 85 % of the variance was kept. The new features were then used to train 3488 neural networks, one for each HW window number. Table 3.5 shows the neural network architecture used to predict which class a trace belonged to. The parameter v in table 3.5 was the number of features after a PCA had been performed. During training, a total of 80000 traces were used and each network was trained for 12 epochs.

Another set of 3488 neural networks for binary classification of $HW_{ij} < 32$ was conducted in the same way as for the classification of $HW_{ij} > 32$ where the same traces and PCA were used.

Chapter 4
Result

4.1 Trace capture metrics

Table 4.1 shows the FPGA resource requirements of the implemented hardware system. Most resources are used by the decryption core but storing inputs, i.e. secret key and ciphertext require roughly 23 % of available 36 KB RAM blocks. The memories for the ciphertext and secret Goppa polynomial were not fully utilized. A high number of memories was needed because signals needed to be supplied in parallel to the decryption core. The memory used for storing the secret permutation was better utilized. This was possible since the core used a memory interface for reading the secret support.

Part	LUTs	Flip-flops	Muxes	RAM 18KB	RAM 36 KB
Decryption core	12953	22972	16	4	19
Ciphertext memory	2	0	0	0	12
Secret Goppa memory	2	0	0	0	16
Support memory	4	0	0	0	3
Error rec	646	0	672	0	0
USB interface	29	65	0	0	0
Total	13938	23038	688	7	47
Utilization	22%	18%	2%	3%	35%

Table 4.1: FPGA resource requirements and utilization.

The execution times for the implemented key generation Python module are given in table 4.2. The costliest operations were the calculation of $\mathbf{Y}$ and the matrix multiplication $\mathbf{H}_{\text{pup}} = \mathbf{YZP}_{\text{perm}}$. Execution times were measured on a virtual machine running Ubuntu OS with 11 GB RAM and 2 CPU cores. The host of the virtual machine was a 2.7 GHz Intel i7 CPU with 16 GB RAM. On average, three attempts were needed to get a public key that was in systematic form. Therefore, approximately 13 minutes was needed to successfully generate keys. The generated keys were stored as Python Numpy arrays and the total key size, i.e. both public and private key, was 2.59 MB.

Key generation step	Execution time
Minimal polynomial $g(x)$	1.9 s
Secret support P	0.007 s
Matrix $\mathbf{Y}$	93.1 s
Matrix $\mathbf{Z}$	2.7 s
Matrix $\mathbf{P}_{\text{perm}}$	0.15 s
Matrix $\mathbf{H}_{\text{pup}}$	150 s
Matrix $\mathbf{H}_{\text{pup}_{\text{bin}}}$	13.1 s
Total	261 s

Table 4.2: Execution time for key generation.

Table 4.3 shows the execution time for setting up and capturing traces. The three top lines were only executed once. So, for a large number of traces, the average time needed to capture a trace was 0.31 s. Execution times were measured on the same virtual machine as times given in table 4.2. The decryption frequency was set to 5 MHz and a total of 20988 points were sampled during each trace capture. This resulted in traces of 168 KB each.

Trace capture step	Execution time
Connect to CW	3.0 s
Program FPGA	7.1 s
Load keys to FPGA	0.13 s
Generate random plaintext	0.005 s
Encrypt and load to FPGA	0.074 s
Decryption/trace capture	0.12 s
Read trace	$5 \cdot 10^{-6}$ s
Read recovered plaintext	0.10 s
Verify recovered plaintext	0.007 s
Total	11 s
Total capture loop	0.31 s

Table 4.3: Execution time for trace capturing.

4.2 Leakage evaluation

Figure 4.1 shows the mean of 1000 traces captured during decryption of a fixed randomly selected ciphertext. Decryption was driven by a 5 MHz clock and traces were sampled at a rate of 10 MHz. To avoid saturation, i.e. peak clipping, in the ADC a total amplification of 45 dB was used. Voltage was the actual quantity measured by the ADC. But, assuming the voltage was measured over an ideal resistor the captured trace corresponds to the power consumption of the FPGA.

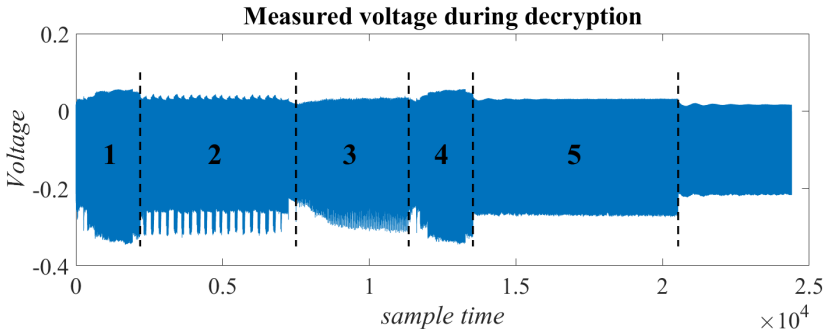

Figure 4.1: Mean measured voltage of 1000 traces using a fixed, randomly selected, ciphertext.

In Figure 4.1, the five steps of decryption are clearly visible. Step one and four have a similar shape which is probably due to that the same additive FFT block was used during both steps. During step two, 20 negative peaks can be seen which corresponds to the block-wise computation of the double syndrome. The ciphertext used as input consisted of 384 ones which were divided into blocks of size of 20. This resulted in 20 blocks that agree well with the observed trace. However, one should keep in mind that the maximum block size is a design parameter, see table 3.2, which can vary between implementations. During step three, the BM decoder gradually reads vectors of the double syndrome and stores them in a shift register. The recovered plaintext was successively calculated bit-by-bit during step five. However, as the decryption implementation utilized a pipelined design during step five, no visual correlation could bee seen between the plaintext and the power consumption.

Figure 4.2 visualizes the measured voltage distribution of 1000 decryption cycles using random ciphertexts. Decryption was driven by a 5 MHz clock, sampling was performed at 10 MHz and a gain of 45 dB was used. The distribution shows two clusters. The clusters centered around -0.26 V, corresponds to samples taken at the rising edge of the decryption clock. The cluster around 0.03 V, corresponds to samples taken at the falling edge of the decryption clock. As all values in the decryption module were updated on the rising clock edge, the left cluster should have a stronger correlation with the dynamic power consumption. One can also observe that the left cluster seems to have a larger variance compared to the right cluster. However, the distribution does not tell if the larger variance was due to the randomly selected inputs, i.e ciphertexts, or the way the hardware was designed.

Figure 4.3 shows the absolute value of Welch's t-statistic. The first set consisted of traces captured with a fixed ciphertext and the second set consisted of traces captured with random ciphertexts. Decryption was performed at 5 MHz and traces were sampled at 10 MHz with a gain of 45 dB. Before calculating the t-statistic the size of each set was truncated to $n_0 = n_1 = 2984$. As decryption step 2, the double syndrome calculation was of non-constant time this part was removed from traces before evaluating the t-statistic. The graph in figure 4.3 clearly shows that leakage was present during steps 3 and 4. Some leakage was also

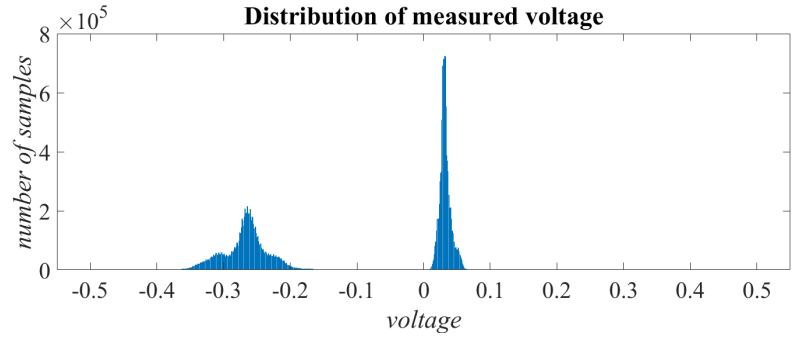

Figure 4.2: Distribution of measured voltage of 1000 decryption cycles using random ciphertexts.

present during step 5. No leakage was detected during step 1. This was expected as the execution of step 1 depended on the secret key which was the same for all traces.

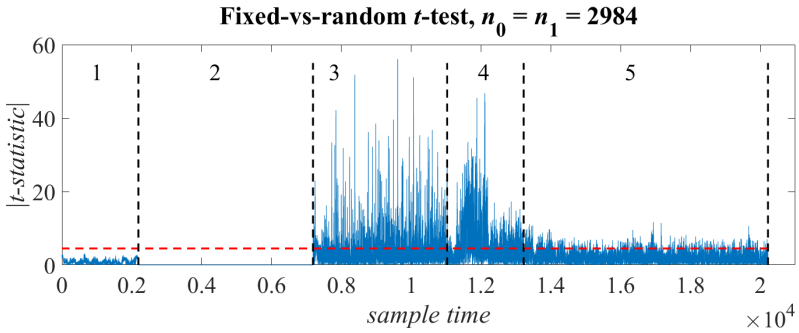

Figure 4.3: Fixed vs. random ciphertext leakage assessment with set sizes $n_0 = n_1 = 2984$.

For the t-statistic in fig 4.4, traces were assigned to one of two sets depending on the HW of the first half of the corresponding plaintext. The first set contained traces where the HW $>$ 32 in the first half of the plaintext. The second set contained all traces where the HW $<=$ 32. As before, the second decryption step was removed before calculating the t-statistic. Traces were captured at a rate twice the decryption frequency, .i.e. 10 MHz, and a gain of 45 dB was used

Figure 4.5 shows the t-statistic where only the last part of decryption was captured. Traces were sorted into classes based on HW like before. In comparison to figure 4.4 trace in 4.5 were sampled at 30 MHz. Thus, six samples during each decryption clock cycle were captured. Furthermore, the gain was increased to 50 dB. The t-statistic in figure 4.5 clearly shows that leakage is present when traces are categorized based on HW.

Figure 4.6 shows t-statistics using only one of the six sample points for each decryption clock, i.e. figure 4.6 shows the six individual sample offsets of figure 4.5.

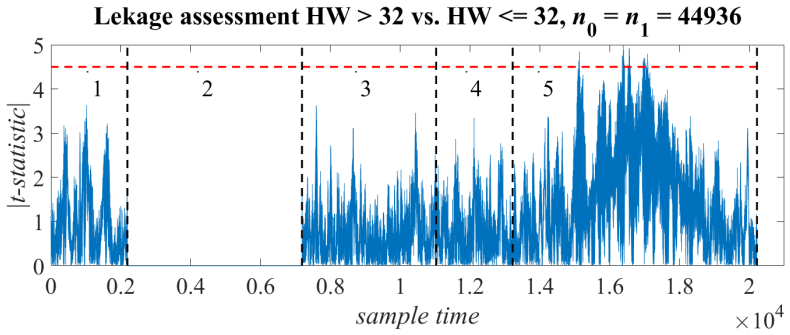

Figure 4.4: Leakage assessment where traces were sorted into sets based on HW of the first half of plaintexts.

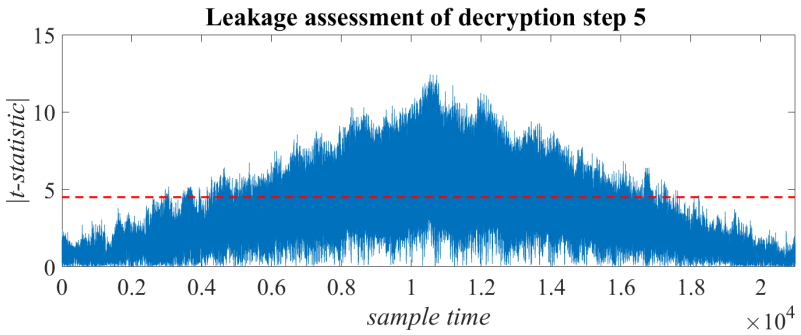

Figure 4.5: Leakage assessment of the last decryption step where traces were sorted into sets based on HW of the first half of plaintexts.

As seen, leakage seems to be present at 0, 3/6, and 4/6 offsets, i.e. most leakage appears to happen during the rising and falling edge of the decryption clock.

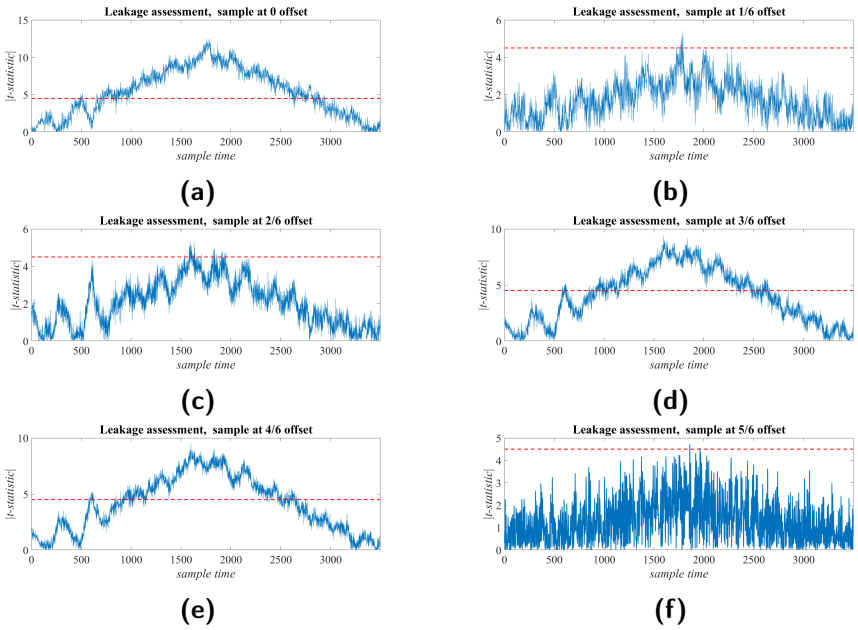

Figure 4.6: Leakage assessment using samples with different offsets from the rising edge of decryption clock, (a) 0 offset, (b) 1/6 clk offset, (c) 2/6 clk offset, (d) 3/6 clk offset, (e) 4/6 clk offset and (f) 5/6 clk offset.

Figure 4.7 shows the t-statistic of a modified version of the last decryption step. In the modified hardware the shift register was removed from the pipeline in figure 3.19. Traces were samples at 30 MHz with a gain of 50 dB. During the calculation of t-statistic traces were sorted into sets based on HW like before.

4.3 Partial message-recovery attack

Figure 4.8 shows the accuracy of predicting which class the first HW window, HW_{i0}, belonged to for 80000 traces. Different sample offsets were tested. The two single offsets that gave the best result were 0 and 3/6, i.e. the samples captured at the rising and falling edge of the decryption clock. Furthermore, using the two best offsets during training increased the accuracy even further. Using all offsets did not lead to higher accuracy when a maximum of 50 epochs was used.

When performing a PCA using the two best offsets dimensionally was reduced from $2 \cdot 3498$ to 547 while keeping 85 % of the variance. The final architecture of the neural network is given in table 4.4.

Figure 4.9 shows the prediction accuracy of classifying if traces belonged to plaintexts with HW ≥ 32 within a window. The accuracy was calculated by using

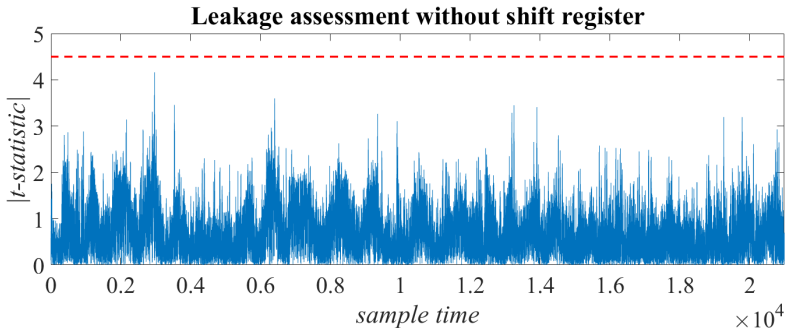

Figure 4.7: Leakage assessment of last decryption step where the shift register was removed from the hardware.

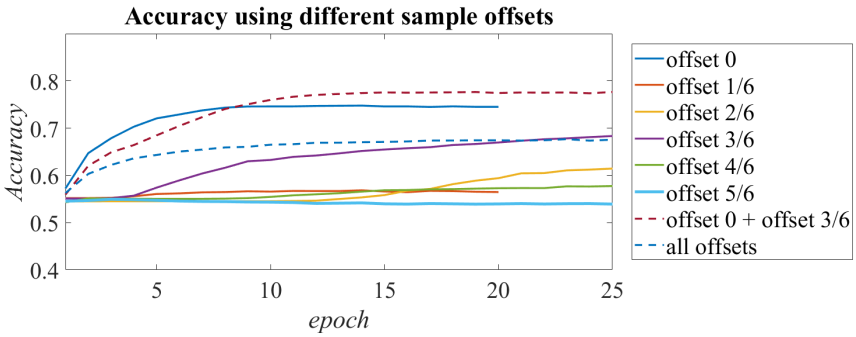

Figure 4.8: Prediction accuracy using samples with different offsets from the rising edge of decryption clock.

Layer type	(input, output) shape	# Parameters
Dense, ReLU	(547, 547)	299756
Dense, ReLU	(547, 273)	149604
Dense, sigmoid	(273, 1)	274

Table 4.4: By using a PCA dimensionality was reduced to 547 features leading to a neural network with 449634 parameters.

a test set of 5000 traces. On average, the accuracy was 78 %. This should be compared to the case when traces are always classified as belonging to the ≤ 32-class in which case the accuracy was 55 % due to the distribution of HW.

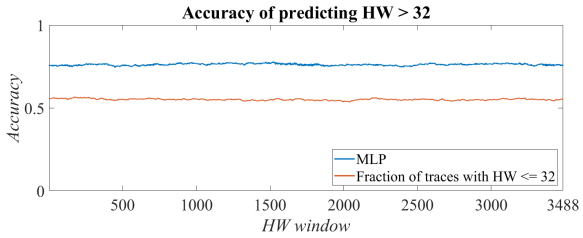

Figure 4.9: Prediction accuracy of HW windows belonging to the class with HW > 32. The graph also shows the true fraction of traces belonging to the other class with HW ≤ 32.

Figure 4.10 shows accuracy for the other set of 3488 neural networks used to predict if the HW was less than 32. The accuracy was similar to 4.9 with an average accuracy of 78 %.

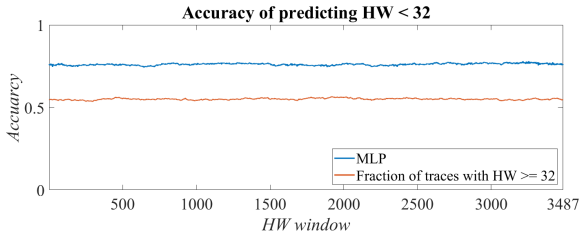

Figure 4.10: Prediction accuracy of HW windows belonging to the class with HW < 32. The graph also shows the true fraction of traces belonging to the other class with HW ≥ 32.

Figure 4.11 shows the prediction accuracy of using different HW window sizes. For each window size x, the MLP was trained to classify if $HW_{i0} > \frac{64}{3488}x$, i.e. if the HW of x first bits of the plaintexts was greater than the average for a given window size x. Figure 4.11 also shows the accuracy when traces always were classified as belonging to the largest of the two sets, i.e. either $HW_{i0} > \frac{64}{3488}x$ or $HW_{i0} \leq \frac{64}{3488}x$.

In figure 4.12, the accuracy of predicting $HW_{i0} > 32$ is shown for both the previously used decryption hardware and modified hardware without shift register in the last decryption step. A new PCA was performed on traces from the modified hardware before training the neural network but the same MLP architecture was used.

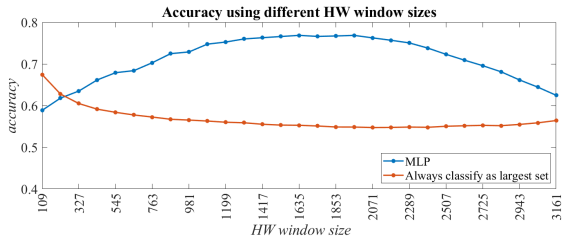

Figure 4.11: Prediction accuracy using different HW window sizes.

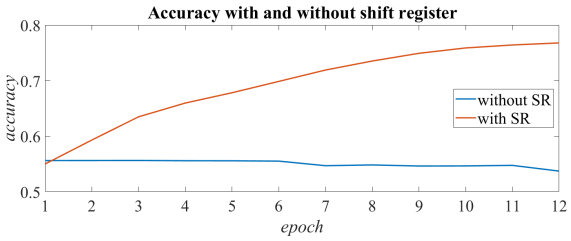

Figure 4.12: Comparison of prediction accuracy when removing the shift register from the decryption pipeline.

Chapter 5
Discussion

The outcome of this thesis is twofold. Some parts are most relevant to an evaluator of the cryptosystem who can modify the system. Other parts are more relevant to an attacker who only can use the system as it is. Therefore the discussion is split into 2 sections.

5.1 From an evaluators point of view

The implemented hardware requires approximately a fifth of the available resources of the CW305 FPGA. Thus there is room to implement more functionality in the FPGA to speed up trace capturing. Inspecting table 4.3 reveals that reading back the recovered plaintext makes up a third of the time needed to capture a trace. However, this is an important step since it is needed to verify that decryption works as intended. Since there is plenty of resources available in the FPGA another approach could have been used. Initially, the capture software could generate a random plaintext and send it to the FPGA. After decryption had finished, a hash function implemented on the FPGA could be used to generate a new plaintext from the recovered plaintext. Then, by also implementing encryption in the FPGA a new ciphertext could be generated. Once a fixed number of decryptions had been performed, the PC could read the last recovered plaintext. Then, by using the initial plaintext along with the same hash function the PC could calculate the expected value of the last recovered plaintext and compare it to the last recovered plaintext. This would essentially reduce the capture time by almost a third.

Another way to speed up trace capturing is to increase the decryption frequency. The sampling frequency could be fixed to two times the decryption frequency as a higher sample rate did not provide more information. Since the CW has a maximum sampling frequency of 96 MHz the decryption frequency could be increased from 5 MHz to 48 MHz which would reduce the trace capture time by almost a third. Thus combining the two suggestions would lead to a trace capture time of approximately 0.1 s. However, when the system was evaluated at higher decryption frequencies, severe distortion, ringing, was observed in captured traces. It is not clear if the distortion was caused by the CW, FPGA, or a combination of both.

Another suggested improvement is based on the observations in figures 4.2 and 4.5. As most leakage seems to happen during the rising edge of the decryption

clock it make sense to capture this part as accurately as possible. However, as seen in figure 4.2 the distribution of samples at the rising decryption clock edge is restricted to the range -0.37 V to -0.17 V. Thus only 20 % of the ADC dynamic range is used. Thus, by modifying the CW hardware it would be possible to increase the dynamic range of captured traces by almost five times. This in turn would reduce the noise in traces caused by quantization.

5.2 From an attackers point of view

The fixed-vs-random leakage assessment in figure 4.3 clearly shows that leakage was present during all steps of decryption except for the first step. However, this was expected since the execution of the first step only depends on, same for all traces, minimal polynomial $g(x)$. With the suggested approach to split traces into sets depending on HW inside windows of the plaintext, only leakage seems to be present during the last decryption step. But one should be aware that using another trace partitioning might lead to different results

As shown in figure 4.8, using more than two samples within each decryption clock cycle does not seem to provide more information. The reason why two samples were better than one is probably due to the way memory cells inside the FPGA is built. Unfortunately, Xilinx does not tell which flip-flop architecture they use in their FPGAs. However, many common flip-flop architectures operate on both the rising and falling edge of the driving clock. For some architectures, data is sampled on one clock edge and propagated through the flip-flop at the other edge. For other architectures, transistors inside the flip-flop are charged to a logic high value at one clock edge and then, possibly, discharged on the other clock edge depending on data present at the flip-flop input [2]. Thus, when targeting leakage of memory cells built out of flip-flops in an FPGA it makes sense to sample at twice the decryption frequency. In case the sample clock is not synchronous to the decryption clock a higher sample rate is probably needed to get equally good measurements.

From figures 4.7 and 4.12 it seems clear that the main leakage point for the purposed attack scenario is the shift register in the pipeline of the last decryption step. As the shift register is the main contributor of leakage, the magnitude of power leaked should be proportional to the number of bits inside the shift register that changes value from zero to one or the other way around. Since the plaintext consists of 64 ones and $3488 - 64$ zeros, most of the ones will be led and tailed by a zero. Thus, using the name HW window might be a bit misleading as what was measured were the sum of 0-to-1 and 1-to-0 transitions. But as mentioned, the low fraction of ones in the plaintext means that the number of transitions will be, approximately, twice the HW of plaintext bits within a given window.

The suggested approach to label traces before training the neural network seems to perform well. It is stressed that predictions were made on single traces, i.e. no averaging of traces was performed. For a message-recovery attack, this is a reasonable setting as a message is typically only transmitted once. However, the information gathered from the neural network in this work is not sufficient to recover the plaintext from a ciphertext. However, the retrieved information

tells an attacker how bits, i.e. zeros and ones, of the plaintext are distributed. Additionally, as seen in figure 4.11 it should be possible to use multiple HW window sizes. By using more HW windows sizes it should be possible to get even more detailed information on the distribution of plaintext bits.

As observed in figure 4.11, the prediction accuracy was rather low for small and large HW window sizes. The reason for this is not obvious. One possible explanation of this is given in Appendix D. For small or large windows the expected number of ones in the two prediction classes are closer together compared to when a window size of 1744 is used. This might explain why prediction is more accurate with window sizes around 1744. This would also mean that prediction accuracy should be low when it comes to predicting the class of a trace where the true HW is close to the threshold used to divide the two classes. However, from this work, it is not possible to confirm this. Another reason why the network has low accuracy for small or large window sizes could be due to a low number of training samples in one of the prediction classes. For example, when using an HW window size of 109, traces are classified depending on the HW of the first 109 plaintext bits. If the HW is above 2, a trace is assigned to set $\mathcal{Q}_{\text{high}}$ otherwise it is assigned to $\mathcal{Q}_{\text{low}}$. Thus, the first set only contains plaintexts with HW$= 0, 1, 2$ but the other set contains plaintext with HW$= 3, 4, \ldots, 63, 64$. For this case, as shown in Appendix D, the probability that a randomly generated plaintext belongs to $\mathcal{Q}_{\text{low}}$ is 68%. This can be compared to 55 % when using a window size of 1744. Thus, in the latter case sizes of the two classes become more equal.

Chapter 6
Conclusion

In this thesis, the Niederreiter decryption core used by Classic McEliece, one of the NIST PQC competition finalists, has been implemented on an Artix 7 FPGA along with an interface for communication over USB with the CW. A Python module for generating keys and encrypting messages has been developed and the CW API has been extended to handle communication with the decryption core.

A fixed-vs-random leakage assessment has been performed on captured traces. This showed that the hardware implementation suffers many leakage points that could, potentially, be exploited during a side-channel attack. Performing a leakage assessment where traces were assigned to a set depending on the HW of a subpart of plaintext bits showed a large number of leakage points during the last step of decryption.

Training of neural networks showed that a sampling frequency greater than two times the decryption frequency did not lead to higher prediction accuracy. By using a sampling frequency of two times the decryption frequency, employing a PCA to reduce dimensionally and training 3488 rather simple neural networks, the HW of subparts of plaintexts could be predicted to be either greater or lower than 32 with an accuracy of 78 % using only a single trace for prediction.

References

[1] Albrecht, M. R., Bernstein, D. J., Chou, T., et al. Classic mceliece: conservative code-based cryptography. `https://classic.mceliece.org/nist/mceliece-20201010.pdf`, 2020.

[2] Alioto, M., Consoli, E., and Palumbo, G. *Flip-Flop Optimized Design*, pages 81–117. Springer International Publishing, Cham, 2015.

[3] Balasch, J., Gierlichs, B., Grosso, V., et al. On the cost of lazy engineering for masked software implementations. In Joye, M. and Moradi, A., editors, *Smart Card Research and Advanced Applications*, pages 64–81, Cham, 2015. Springer International Publishing.

[4] Benadjila, R., Prouff, E., Strullu, R., et al. Deep learning for side-channel analysis and introduction to ascad database. *Journal of Cryptographic Engineering*, 10, 2020.

[5] Bilgin, B., Gierlichs, B., Nikova, S., et al. Higher-order threshold implementations. In Sarkar, P. and Iwata, T., editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 326–343. Springer Berlin Heidelberg, 2014.

[6] Goodwill, G., Jun, B., Jaffe, J., and Rohatgi, P. A testing methodology for sidechannel resistance validation. In *NIST non-invasive attack testing workshop*, 2011.

[7] Heyse, S., Moradi, A., and Paar, C. Practical power analysis attacks on software implementations of mceliece. In Sendrier, N., editor, *Post-Quantum Cryptography*, pages 108–125, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Huffman, W. C. and Pless, V. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, Cambridge, 2003.

[9] Lahr, N., Niederhagen, R., Petri, R., and Samardjiska, S. Side channel information set decoding using iterative chunking - plaintext recovery from the "classic mceliece" hardware reference implementation. In *ASIACRYPT*, 2020.

[10] Mangard, S., Oswald, E., and Popp, T. *Power analysis attacks Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[11] McEliece, R. J. A public-key cryptosystem based on algebraic coding theory. Report, NASA, 1978.

[12] Molter, H. G., Stöttinger, M., Shoufan, A., and Strenzke, F. A simple power analysis attack on a mceliece cryptoprocessor. *Journal of Cryptographic Engineering*, 1(1):29–36, 2011.

[13] National Institute of Standards and Technology. Post-quantum cryptography standardization. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`, 2017.

[14] Niederreiter, H. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15(2):159–166, 1986.

[15] Petrvalsky, M., Richmond, T., Drutarovsky, M., et al. Differential power analysis attack on the secure bit permutation in the mceliece cryptosystem. In *2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 132–137, 2016.

[16] Schneider, T. and Moradi, A. Leakage assessment methodology. *Journal of Cryptographic Engineering*, 6(2):85–99, 2016.

[17] Shang, L., Kaviani, A. S., and Bathala, K. Dynamic power consumption in virtex^{TM}-ii fpga family. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, FPGA '02, page 157–164, New York, NY, USA, 2002. Association for Computing Machinery.

[18] Shoufan, A., Strenzke, F., Molter, H. G., and Stöttinger, M. A timing attack against patterson algorithm in the mceliece pkc. In Lee, D. and Hong, S., editors, *Information, Security and Cryptology – ICISC 2009*, pages 161–175, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[19] Smart, N. P. *Cryptography: an introduction*, volume 3. McGraw-Hill, New York, 2003.

[20] Strenzke, F., Tews, E., Molter, H. G., et al. Side channels in the mceliece pkc. In Buchmann, J. and Ding, J., editors, *Post-Quantum Cryptography*, pages 216–229, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[21] Wang, W., Szefer, J., and Niederhagen, R. Fpga-based key generator for the niederreiter cryptosystem using binary goppa codes. In Fischer, W. and Homma, N., editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 253–274, Cham, 2017. Springer International Publishing.

[22] Wang, W., Szefer, J., and Niederhagen, R. Fpga-based niederreiter cryptosystem using binary goppa codes. In Lange, T. and Steinwandt, R., editors, *Post-Quantum Cryptography*, pages 77–98, Cham, 2018. Springer International Publishing.

Appendix A

McEliece Python module

```
import numpy as np
import random as rd
import galois
import bitarray as ba

class mceliece():

    def __init__(self, m, n, t):
        """Creates a new McEliece object
        Parameters:
            m (int): Size of binary field
            n (int): Code length (length of plaintext)
            t (int): number of errors"""
        self.__m = m
        self.__n = n
        self.__t = t
        self.__irr_poly = galois.irreducible_poly(2, m, method="smallest")
        self.__GF = galois.GF(2**m,irreducible_poly=self.__irr_poly)
        self.__min_poly = self.__GF(0)
        self.__P = []
        self.__H = self.__GF.Zeros([self.__t, self.__n])

    def rand_irr_poly(self, r = None):
        # Get the first lexicographic irreducible polynomial of GF(2)
        p_mod = galois.irreducible_poly(2, self.__t, method="smallest");
        p_mod2 = galois.Poly(p_mod.coeffs, field=self.__GF)

        if (r is None):
            r = galois.Poly.Random(self.__t-1, field=self.__GF)

        ri = galois.Poly(1,field=self.__GF)
        coeff = self.__GF.Zeros([self.__t, self.__t+1])

        for i in range(self.__t+1):
            for j in range(self.__t):
                if ri.degree >= j:
                    coeff[j,i] = ri.coeffs[::-1][j]
            ri = (ri * r) % p_mod2

        coeff_gs = coeff.row_reduce()
        min_poly = galois.Poly(np.concatenate(([1], [0]*(self.__t))),
            field=self.__GF)
        min_poly = min_poly + galois.Poly(coeff_gs[:,-1][::-1],field=self.__GF)

        return min_poly

    def convert(self,beta, num):
        ret = self.__GF(0)
        for i in range(len(beta)):
            if(num >> i) % 2:
                ret += beta[i]
        return ret

    def parity_check_matrix(self):
        basis = self.__GF(2**(np.arange(self.__m)[::-1]))
        alpha = []
        for i in range(2**self.__m):
            alpha.append(self.__GF(self.convert(basis, i)))
```

```
        poly_eval = []

        for i in range(2**self.__m):
            poly_eval.append(self.__min_poly(alpha[i]))
            if(poly_eval[i] == self.__GF.Zeros(1)):
                return False

        Y = self.__GF.Zeros([self.__t, 2**self.__m])
        for i in range(2**self.__m):
            Y[0,i] = self.__GF.Ones(1)
            for j in range(1, self.__t):
                Y[j, i] = Y[j-1, i] * alpha[i]

        Z = self.__GF.Zeros([2**self.__m, 2**self.__m])
        for i in range(2**self.__m):
            Z[i, i] = poly_eval[i]**(-1)


        P_trans = self.__GF.Zeros([2**self.__m, 2**self.__m])
        for i in range(2**self.__m):
            row = self.__P[i]
            P_trans[row, i] = 1

        H_tmp = np.dot(np.dot(Y, Z), P_trans)

        # Transform the parity check matrix to systematic form
        GF2 = galois.GF(2)
        H_bits_tmp = GF2(self.unpack_bits(np.array(H_tmp[:,0:self.__n])))
        self.__H = H_bits_tmp.row_reduce()

        # Check that the parity check matrix is in systematic form
        iden_matrix = GF2.Identity(self.__m*self.__t)
        #iden_matrix = np.identity(self.m*self.t)
        if (self.__H[:self.__t*self.__m,:self.__t*self.__m]
                != iden_matrix).any():
            print("Error, could not bring H to systematic form!" +
                " Trying new keys.")
            return False
        return True

    def random_elements(self):
        P = rd.sample(range(2**self.__m), 2**self.__m)
        return P

    def unpack_bits(self, x):
        num_bits = self.__m
        xshape = list(x.shape)
        x = x.reshape([-1,1], order='F')
        mask = 2**np.arange(num_bits, dtype=x.dtype).reshape([1, num_bits])
        return np.transpose((x & mask).astype(bool).astype(int)).reshape(
            [xshape[0]*num_bits, xshape[1]],order='F')

    def gen_private_key(self):
        self.__min_poly = self.rand_irr_poly()
        self.__P = self.random_elements()

    def generate_keys(self):
        while(1):
            self.gen_private_key()
            if(self.parity_check_matrix()):
                return
            print("Testing new key")

    def generate_random_plaintext(self):
        plaintext = [1]*self.__t + [0]*(self.__n - self.__t)
        rd.shuffle(plaintext)
        return plaintext

    def encrypt(self, plaintext):
        return np.dot(np.array(self.__H), plaintext) % 2

    def ciphertext_as_bytearray(self, ciphertext):
        if((len(ciphertext) % 8) != 0):
            ciphertext = np.append([0]*(len(ciphertext) % 8), ciphertext)

        ciphertext = ciphertext.reshape([-1,8])
        weights = 2**np.arange(8)[::-1]
        ciphertext = np.dot(ciphertext, weights)
        ciphertext = np.array(ciphertext, dtype=np.uint8)
        return bytearray(ciphertext[::-1])
```

```
def support_as_bytearray(self):
    return bytearray(np.array(self.__alpha))

def min_poly_as_bytearray(self):
    bytes_per_coeff = int(np.ceil(self.__m/8))
    min_poly_coeff = np.zeros([bytes_per_coeff*(self.__t+1)],
        dtype=np.uint8)
    for i in reversed(range(self.__t+1)):
        tmp_coeff = bytearray(np.array([self.__min_poly.coeffs[i]]))
        for j in range(len(tmp_coeff)):
            min_poly_coeff[(self.__t - i)*bytes_per_coeff + j] = (
                tmp_coeff[j])
    return bytearray(min_poly_coeff)

def save_H(self, path):
    np.save(path, np.array(self.__H))

def load_H(self, path):
    GF2 = galois.GF(2)
    self.__H = GF2(np.load(path))

def save_min_poly(self, path):
    np.save(path, np.array(self.__min_poly.coeffs))

def load_min_poly(self, path):
    coeffs = np.load(path)
    self.__min_poly = galois.Poly(coeffs, field=self.__GF)

def save_P(self, path):
    np.save(path, np.array(self.__P))

def load_P(self, path):
    self.__P = list(np.load(path))
```

Appendix B

Extension of CW305 API

```
import logging
import time
import random
from datetime import datetime
import os.path
import re
import io
from ._base import TargetTemplate
from chipwhisperer.hardware.naeusb.naeusb import NAEUSB, packuint32
from chipwhisperer.hardware.naeusb.pll_cdce906 import PLLCDCE906
from chipwhisperer.hardware.naeusb.fpga import FPGA
from chipwhisperer.common.utils import util
from chipwhisperer.common.utils.util import camel_case_deprecated
from chipwhisperer.common.utils.util import fw_ver_required

from bitarray import bitarray
from .CW305 import CW305
from .CW305 import CW305_USB
import numpy as np

class CW305MC(CW305):
    _name = "ChipWhisperer CW305 (Artix-7)"
    BATCHRUN_START = 0x1
    BATCHRUN_RANDOM_KEY = 0x2
    BATCHRUN_RANDOM_PT = 0x4

    def __init__(self):
        import chipwhisperer as cw

        TargetTemplate.__init__(self)
        self._naeusb = NAEUSB()
        self.pll = PLLCDCE906(self._naeusb, ref_freq = 12.0E6)
        self.fpga = FPGA(self._naeusb)

        self.hw = None
        self.oa = None

        self._woffset_sam3U = 0x000
        self.default_verilog_defines = 'cw305_defines.v'
        self.default_verilog_defines_full_path = os.path.dirname(cw.__file__) +
            '/../../hardware/victims/cw305_artixtarget/fpga/common/' +
            self.default_verilog_defines
        self.bytecount_size = 7 # pBYTECNT_SIZE in Verilog
        self.registers = 16 # number of registers we expect to find
        self.address_bits = 8 # (maximum) number of addresses in each register

        self.m = 12 # size of field, GF(2^m)
        self.t = 64 # weight of error vector
        self.n = 3488 # code length

        self.poly_g_path = ""
        self.p_matrix_path = ""
        self.H_pub_path = ""

        self._clksleeptime = 1
        self._clkusbautooff = True

        self.last_key = bytearray([0]*16)
        self.p_matrix = []
```

```
        self.poly_g = []
        self.H_pub = []
        self.last_cipher = []
        self.last_error = []

        self.error = bitarray()

        self.target_name = 'McEliece'

    def chunk_array(self, arr, n):
        chunk_size = [n]*(len(arr)//n) + [len(arr)%n]*(len(arr)%n != 0)

        tmp_list = []
        start=0
        stop = 0
        for i in range(len(chunk_size)):
            stop += chunk_size[i]
            tmp_list.append(arr[start:stop])
            start += chunk_size[i]
        return tmp_list

    def pack_bits_to_bytearray(self, arr, bits_per_element):
        fmt = "{0:0" + str(bits_per_element) + "b}"
        bit_str = ""
        for i in range(len(arr)):
            bit_str += fmt.format(arr[i])

        # Zero pad if needed
        bit_str = "0"*(len(bit_str)%8) + bit_str

        # construct bytes of the string starting with LSB first
        byte_lst = []
        for i in reversed(range(0, len(bit_str), 8)):
            byte_lst.append(int(bit_str[i:i+8],2))

        # Convert to bytearray
        return bytearray(byte_lst)

    def pack_alpha_bits_to_bytearray(self, arr):
        # Create format for converting ints to bit string
        fmt = "{0:0" + str(16) + "b}"
        byte_lst = []
        for i in range(len(arr)):
            tmp_str = fmt.format(arr[i])
            byte_lst.append(int(tmp_str[8:16],2))
            byte_lst.append(int(tmp_str[0:8],2))

        return bytearray(byte_lst)


    def load_support_from_bytearray(self, alpha):
        packed_bits = self.pack_alpha_bits_to_bytearray(alpha)
        self.p_matrix = self.chunk_array(packed_bits, 32)

        for i in range(0, len(self.p_matrix)):
             self.fpga_write(addr=self.REG_P_MATRIX_IN, offset= i,
                data=self.p_matrix[i])

    def load_poly_g_from_bytearray(self, min_poly):
        packed_bits = self.pack_bits_to_bytearray(min_poly, self.m)
        self.poly_g = self.chunk_array(packed_bits, 32)

        for i in range(0, len(self.poly_g)):
            self.fpga_write(addr=self.REG_POLY_G_IN, offset=i,
                data=self.poly_g[i])

    def load_cipher_from_bytearray(self, cipher):
        packed_bits = self.pack_bits_to_bytearray(cipher, 1)
        self.last_cipher = self.chunk_array(packed_bits, 32)

        for i in range(0, len(self.last_cipher)):
            self.fpga_write(addr=self.REG_CIPHER_IN, offset=i, data=self.
                last_cipher[i])

    def read_error(self):
        self.last_error = []

        for i in range(0, 14):
            if (i  < 13):
                self.last_error.append(bytearray(
```

```
                self.fpga_read(addr=self.REG_REC_ERR_OUT,
                offset=i, readlen=32)[::-1]))
        else:
            self.last_error.append(bytearray(
                self.fpga_read(addr=self.REG_REC_ERR_OUT,
                offset=i, readlen=20)[::-1]))

def fpga_write(self, addr, offset, data):
    addr = ((addr << self.address_bits) + offset) << self.bytecount_size
    return self._naeusb.cmdWriteMem(addr, data)

def fpga_read(self, addr, offset, readlen):
    addr = ((addr << self.address_bits) + offset) << self.bytecount_size
    data = self._naeusb.cmdReadMem(addr, readlen)
    return data

def is_done(self):
    result = self.fpga_read(addr= self.REG_CRYPT_GO, offset=0,
        readlen=1)[0]
    if result == 0x01:
        return False
    else:
        self.fpga_write(addr=self.REG_USER_LED, offset=0, data=[0])
        return True
```

Appendix C

Trace capture script

```
import chipwhisperer as cw
import pandas as pd
import random as rd
import pickle
from bitarray import bitarray
import numpy as np
import mceliece as mceliece
import json
import feature_extract_helper as fh
import time

#############################################################
# Configure trace capturing
#############################################################

# Configure number of traces to capture. Total numer of
# traces is (number_of_batch - batch_start_num) * nT
number_of_batch = 100     # Number of batches to capture
batch_start_num = 50      # At which batch number to start
nT = 1000                 # Number of traces in each batch

# Configure type of plaintext
# 0 : only fixed plaintext
# 1 : fixed or random with equal probability
# 2 : only random plaintext
plaintext_type =  2

# Configure which part to capture
# If set to false capture starts at the beginning of decryption
# If set to true capture starts at beginning of step five, error locater
only_capture_last_part = True

sample_points = 24400
if only_capture_last_part:
    sample_points = 3498*6

#Path to trace and data storage. For each batch four files
#are stored, traces, plaintexts, ciphertexts and type.
#Files are stored as
#<trace_path> + <pre_str> + "_traces_batch" + {batch numer} +
#    <post_str> + ".npy"
#<trace_path> + <pre_str> + "_plaintexts_batch" + {batch numer} +
#    <post_str> + ".npy"
#<trace_path> + <pre_str> + "_ciphertexts_batch" + {batch numer} +
#    <post_str> + ".npy"
#<trace_path> + <pre_str> + "_type_batch" + {batch numer} +
#    <post_str> + ".npy"
trace_path = "/home/traces/"
pre_str = "capture_id_0"
post_str = ""

#############################################################
# Connect to ChipWhisperer
#############################################################
scope = cw.scope()

#############################################################
# Connect to CW305 target and program FPGA
#############################################################
```

```
target = cw.target(
    scope,
    cw.targets.CW305MC,
    force=True,
    bsfile='/home/mc/hw/cw305_top.bit',
    defines_files=['/home/hw/cw305_defines.v'],
    slurp=True)

################################################################
# Confiure ChipWhisperer
################################################################
decimate = 1

scope.gain.db = 30

scope.adc.samples = sample_points
scope.adc.decimate = decimate
scope.adc.offset = 0
scope.adc.presamples=0
scope.adc.basic_mode = "rising_edge"

scope.clock.adc_src = "clkgen_x1"
scope.clock.adc_phase = 0

scope.clock.clkgen_src = "extclk"
scope.clock.clkgen_mul = 6
scope.clock.clkgen_div = 1

scope.trigger.triggers = "tio4"

scope.io.tio1 = "serial_rx"
scope.io.tio2 = "serial_tx"
scope.io.hs2 = "disabled"

################################################################
# Create McEliece object and load/generate keys
# generate keys
################################################################
m=12          # Field size
n=3488        # Code length
t=64          # Number of correctable errors

mc = mceliece.mceliece(m,n,t)

# This part is for using previously generated keys
key_path = "/home/mc/key/"
mc.load_H(key_path + "key_id_0_H.npy")
mc.load_P(key_path + "key_id_0_P.npy")
mc.load_min_poly(key_path + "key_id_0_min_poly.npy")

fixed_pt = mc.generate_random_plaintext()

################################################################
# Configure CW305
################################################################
target.vccint_set(1.0)
crypto_freq = 5e6

target.pll.pll_enable_set(True)
target.pll.pll_outenable_set(False, 0)
target.pll.pll_outenable_set(True, 1)
target.pll.pll_outenable_set(False, 2)

target.pll.pll_outfreq_set(crypto_freq, 1)

target.clkusbautooff = True
target.clksleeptime = np.ceil((sample_points/crypto_freq)*1000 +
    1).astype("int")

# Load private key to FPGA
target.load_support_from_bytearray(mc.get_P())
target.load_poly_g_from_bytearray(mc.get_min_poly().coeffs)

scope.clock.reset_dcms()
scope.clock.reset_adc()
assert (scope.clock.adc_locked), "ADC failed to lock"
time.sleep(0.5)
print(scope.clock.adc_freq)

################################################################
```

```
# Main trace capture loop
#################################################################
for batch_num in range(batch_start_num, number_of_batch):
    print("Batch: " + str(batch_num + 1))

    traces = []
    plaintexts = []
    ciphertexts = []
    pt_types = []

    for i in range(nT):
        operation_ok = True
        pt = 0
        pt_type = 0 # 0 fixed, 1 random

        if plaintext_type ==  0:
            pt = fixed_pt
            pt_type = 0
        elif plaintext_type == 1:
            if (rd.randint(0,9) < 5):
                pt = fixed_pt
                pt_type = 0
            else:
                pt = mc.generate_random_plaintext()
                pt_type = 1
        else:
            pt = mc.generate_random_plaintext()
            pt_type = 1

        ct = mc.encrypt(pt)
        target.load_cipher_from_bytearray(ct)

        if only_capture_last_part:
            double_syndrome_clk = fh.calculate_double_syndrome_clks(ct)
            scope.adc.offset = (1095 + double_syndrome_clk + 1921 +
                1095) * scope.clock.clkgen_mul
        else:
            scope.adc.offset = 0

        scope.arm()
        target.go()
        ret = scope.capture()

        timeout = 0
        while (not target.is_done()):
            ii += 1
            time.sleep(0.05)
            if timeout > 100:
                print("Target did not finish operation")
                operation_ok = False

        if ret:
            print("Timeout happened during capture")
            operation_ok = False

        trace = scope.get_last_trace()
        target.read_error()

        rec_pt = bytearray()
        for word  in reversed(target.last_error):
            rec_pt.extend(word)

        for j in range(len(rec_pt)):
            tmp_byte = np.dot(pt[j*8 : j*8 + 8], 2**(np.arange(8)[::-1]))
            if not tmp_byte == rec_pt[j]:
                print("Recovered plaintext doesn't match known plaintext!")
                operation_ok = False

        if operation_ok == False :
            print("An error occoured, aborting operation at trace number " +
                str(i))
            break

        traces.append(trace)
        plaintexts.append(pt)
        ciphertexts.append(ct)
        pt_types.append(pt_type)

    batch_str = "_batch" + str(batch_num)
    np.save(trace_path + pre_str + "_traces" + batch_str + post_str +
```

```
        ".npy", np.array(traces), allow_pickle=False)
    np.save(trace_path + pre_str + "_plaintexts" + batch_str + post_str +
        ".npy", np.array(plaintexts, dtype="uint8"), allow_pickle=False)
    np.save(trace_path + pre_str + "_ciphertexts" + batch_str + post_str +
        ".npy", np.array(ciphertexts, dtype="uint8"), allow_pickle=False)
    np.save(trace_path + pre_str + "_error_types" + batch_str + post_str +
        ".npy", np.array(pt_types, dtype="uint8"), allow_pickle=False)

capture_settings = {
    "batches" : number_of_batch,
    "samples" : scope.adc.samples,
    "type␣of␣pt" : plaintext_type,
    "gain" : scope.gain.db,
    "trigger" : scope.adc.basic_mode,
    "presamples" : scope.adc.presamples,
    "decimate" : scope.adc.decimate,
    "sampling_freq" : scope.clock.adc_freq,
    "adc_src" : scope.clock.adc_src,
    "clkgen_src" : scope.clock.clkgen_src,
    "adc_phase" : scope.clock.adc_phase,
    "clkgen_div" : scope.clock.clkgen_div,
    "clkgen_mul" : scope.clock.clkgen_mul
}

setting_file = open(trace_path + pre_str + "_settings.json", "w")
json.dump(capture_settings, setting_file)
setting_file.close()

print("Done!")
```

Appendix D

Probability of HW in a window

Let $\mathbf{D}$ be a random binary vector of length n where t bits have value 1. Let $\mathbf{X}$ denote the sub-vector consisting of the x first bits of $\mathbf{D}$. Furthermore, let p denote the number of 1's in $\mathbf{X}$. Then, the number of possible sub-vectors $\mathbf{X}$ is

$$\binom{p}{x} = \frac{(p+x-1)!}{p!(x-1)!} = \frac{\Gamma(p+x)}{\Gamma(p+1)\Gamma(x)} \tag{D.1}$$

where Γ is the gamma function. Given that the first x bits of $\mathbf{D}$ have p 1's, the number of possible sub-vectors of the other part of $\mathbf{D}$ is

$$\binom{t-p}{n-x} = \frac{(t-p+n-x-1)!}{(t-p)!(n-x-1)!} = \frac{\Gamma(t-p+n-x)}{\Gamma(t-p+1)\Gamma(n-x)}. \tag{D.2}$$

Thus, the total number of possible vectors $\mathbf{D}$ of length n with t 1's where the first x bits contains p 1's is

$$\frac{\Gamma(p+x)}{\Gamma(p+1)\Gamma(x)} \frac{\Gamma(t-p+n-x)}{\Gamma(t-p+1)\Gamma(n-x)}. \tag{D.3}$$

The total number of possible vectors D of length n with t 1's is given by

$$\binom{t}{n} = \frac{(t+n-1)!}{t!(n-1)!} = \frac{\Gamma(t+n)}{\Gamma(t+1)\Gamma(n)}. \tag{D.4}$$

The probability that a randomly generated vector $\mathbf{D}$ of length n with t 1's has exactly p 1's in the first x bits is then given by

$$P(p,x,t,n) = \frac{\Gamma(p+x)}{\Gamma(p+1)\Gamma(x)}\cdot \tag{D.5}$$

$$\frac{\Gamma(t-p+n-x)}{\Gamma(t-p+1)\Gamma(n-x)}\cdot \tag{D.6}$$

$$\frac{\Gamma(t+1)\Gamma(n)}{\Gamma(t+n)}. \tag{D.7}$$

However, as the value of the gamma function rapidly grows with a larger argument is often necessary to approximate the gamma function with the log-gamma

function $\ln\Gamma(\cdot)$. Thus, $p = \ln(P)$ can be approximated by

$$p = \ln\Gamma(p+x) - \ln\Gamma(p+1) - \ln\Gamma(x) + \tag{D.8}$$
$$\ln\Gamma(t-p+n-x) - \ln\Gamma(t-p+1)\ln\Gamma(n-x) + \tag{D.9}$$
$$\ln\Gamma(t+1) + \ln\Gamma(n) - \ln\Gamma(t+n) \tag{D.10}$$

and then P can be approximated by

$$P = e^p. \tag{D.11}$$

In figure D.1 the probability mass function of $\mathbf{D}$ is shown for the three variable sets $(p = 58, x = 3161)$, $(p = 32, x = 1744)$ and $(p = 2, x = 109)$, in all three cases $n = 3488$ and $t = 64$.

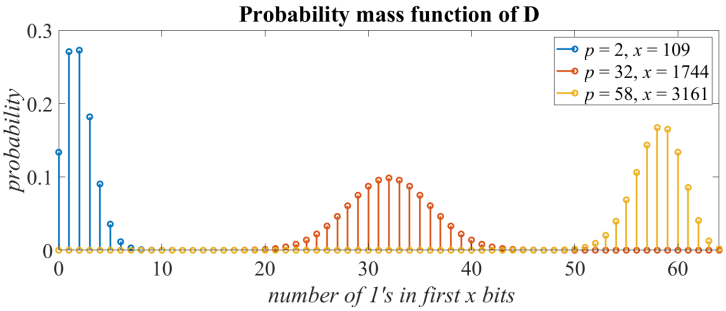

Figure D.1: Probability mass function of $\mathbf{D}$ where $n = 3488$ and $t = 64$.

Now, the same settings as used during classification in figure 4.11 can be employed where $n = 3488$ and $t = 64$. The first class consists of plaintexts having $2y$, $y = 1, 2, \ldots, 29$ or fever 1's in first $x = 109y$ bits, i.e. x denotes the HW window size. Then, the probability mass function of each class and its complement class can be derived from equation D.10. Figure D.2 shows the difference between the expected value of the two classes for different HW window sizes.

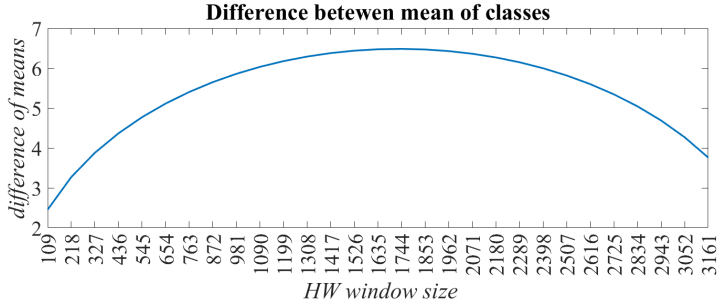

Figure D.2: Difference between the expected value of the two classes for different HW window sizes.