Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

J. Mathias H. Grünfeld

Side-Channel Attacks on CRYSTALS Kyber

An analysis of a post-quantum algorithm and its vulnerabilities to side-channel attacks

Master's thesis in Communication Technology and Digital Security
Supervisor: Stig-Frode Mjølsnes
Co-supervisor: Anders Paulsus, Ella Kristensen
June 2023

NTNU
Norwegian University of
Science and Technology

J. Mathias H. Grünfeld

Side-Channel Attacks on CRYSTALS Kyber

An analysis of a post-quantum algorithm and its vulnerabilities to side-channel attacks

Master's thesis in Communication Technology and Digital Security
Supervisor: Stig-Frode Mjølsnes
Co-supervisor: Anders Paulsus, Ella Kristensen
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title: Side-Channel Attacks on CRYSTALS Kyber: An analysis of a post-quantum algorithm and its vulnerabilities to side-channel attacks

Student:

Problem description:

Cryptographic systems based on the Integer Factorization problem (IFP), such as RSA, have remained a world-wide standard for public key cryptography (PKC). When implemented correctly, these cryptosystems can withstand most brute-force attacks from a classical computer. However, the inevitable arrival of full-scale quantum computers might pose a threat to the security of our standardized PKC cryptosystems.

As a result, the National Institute of Standards and Technology (NIST) has launched a competition to find new cryptographic algorithms that can withstand the power of quantum computers. One of the winners of NIST´s competition within Key Encapsulation Mechanisms (KEM) is CRYSTALS Kyber. Contrary to RSA, Kyber is a lattice-based cryptosystem, based on the learning-with-errors (LWE) problem. There are currently no known quantum algorithms which can solve the LWE-problem in polynomial time.

However, even though Kyber is designed to resist attacks from quantum computers, it is still vulnerable to side-channel attacks. A side-channel attack is a type of attack that exploits weaknesses in the implementation of a cryptographic algorithm, rather than weaknesses in the algorithm itself. A highly efficient side-channel attack is analyzing the power consumption of a micro-controller when it is performing cryptographic operations in order to retrieve a secret key.

This project aims to execute side-channel attacks on Kyber, using power analysis and a tool called ChipWhisperer. Using NIST´s own implementations of Kyber, analyses will be performed in order to evaluate the impact of side-channel attacks.

Approved on:
Main supervisor:
Co-supervisors:

Abstract

The potential rise of quantum computers pose a threat to today's encryption standards. CRYSTALS Kyber a post-quantum lattice-based Key Encapsulation Mechanism (KEM) used to exchange a set of symmetric keys used for encryption. However, even the most secure algorithms have vulnerabilities. The research questions of this thesis ask how information can be leaked from the implementation of a cryptographic component and how the leaked information can be used to exploit an algorithm.

This thesis takes a deep-dive into the Kyber algorithm, its subcomponents, its mathematics and how it is implemented in code. The thesis uses the ChipWhisperer tool in order to analyze power traces of the Kyber decapsulation phase in order to execute a side-channel attack. The thesis also explains how the attack is set up, in order to verify and continue the research of the vulnerabilities of Kyber in regard to side-channels. The goal of the attack is to extract a part of the secret key.

The attack did, however, not succeed on the given implementation of Kyber used in this thesis. Interesting findings were observed, but they were not specific enough to carry out the entire attack. The findings are evaluated and interpreted to propose potential explanations for the attack's failure. Finally, the thesis proposes areas of improvement for the attack and important aspects which require further research.

Sammendrag

Den potensielle fremveksten av kvantedatamaskiner utgjør en trussel mot dagens krypteringsstandarder. CRYSTALS Kyber er en post-kvante, gitterbasert nøkkelinnkapslingsmekanisme (Key-Encapsulation Mechanism", KEM) som brukes til å utveksle et sett med symmetriske nøkler for kryptering. Imidlertid har selv de mest sikre algoritmene sårbarheter. Forskningsspørsmålene i denne avhandlingen spør hvordan informasjon kan lekkes fra implementeringen av en kryptografisk komponent og hvordan den lekkede informasjonen kan brukes til å utnytte en algoritme.

Denne avhandlingen tar et dypdykk inn i Kyber-algoritmen, dens delkomponenter, dens matematikk og hvordan den er implementert i kode. Avhandlingen bruker ChipWhisperer-verktøyet for å analysere strømsporene til Kyber-dekapsuleringsfasen for å utføre et sidekanalangrep. Avhandlingen forklarer også hvordan angrepet er satt opp, for å verifisere og fortsette forskningen på Kybers sårbarheter med hensyn til sidekanaler. Målet med angrepet er å utvinne en del av den hemmelige nøkkelen.

Angrepet lyktes imidlertid ikke på den gitte implementeringen av Kyber som brukes i denne avhandlingen. Det ble observert interessante funn, men ikke spesifikke nok til å gjennomøre hele anrepet. Funnene evalueres og tolkes for å foreslå potensielle forklaringer på angrepets feil. Til slutt foreslår avhandlingen forbedringsområder for angrepet og viktige aspekter som krever ytterligere forskning.

Preface

This thesis concludes five years of studying at the Norwegian University of Science and Technology (NTNU). Despite surviving a pandemic, obtaining severe stress levels two months each year and enduring long, rainy winters spanning from September to May, I must thank NTNU for the education I have received. I have met friends who will last a lifetime, gained invaluable experiences and grown as a human during these five years.

I would like to thank my supervisor, Stig-Frode, for valuable insights in cryptography. I am deeply grateful to Ella and Anders for their invaluable insights and guidance on side-channel attacks and the Kyber algorithm. Their patience, trust, and support throughout the entire year have been instrumental.

Contents

List of Figures ix

List of Tables xi

Algorithms xiii

List of Acronyms xv

1 Introduction and background 1
- 1.1 Research Questions . . . 2
- 1.2 Thesis Structure . . . 2
- 1.3 Quantum Computers . . . 3
- 1.4 NIST PQC . . . 4
- 1.5 Related work and other contributions . . . 5

2 Theory 7
- 2.1 Lattice-based cryptography . . . 7
  - 2.1.1 Learning With Errors . . . 9
- 2.2 Key Encapsulation Mechanism . . . 9
- 2.3 CRYSTALS Kyber . . . 11
  - 2.3.1 The Kyber Algorithm . . . 11
  - 2.3.2 Kyber Parameters . . . 15
- 2.4 Side-Channel attacks . . . 16
  - 2.4.1 Power Analysis . . . 17
  - 2.4.2 Correlation Power Analysis . . . 19

3 Methodology 23
- 3.1 Lab Setup . . . 23
  - 3.1.1 ChipWhisperer . . . 23
  - 3.1.2 The pqm4 library . . . 24
  - 3.1.3 Applying Kyber to ChipWhisperer . . . 25
- 3.2 Setting up attack on Kyber . . . 26
  - 3.2.1 Decapsulation of ciphertexts . . . 28

3.3 Attack on Kyber 30
3.3.1 Data Collection and Hypothesis phase 30
3.3.2 Correlation Phase 36
3.3.3 Real World Scenario 37

4 Results and Analysis **39**
4.1 Results 39
4.1.1 Observed results 39
4.2 Analysis of the results and potential pitfalls 43
4.2.1 Powers of 2 43
4.2.2 Differences in implementations 43
4.2.3 Noise in traces and trigger positions 44

5 Discussion **45**
5.1 Difficulties with applying Kyber 45
5.1.1 pqm4 45
5.1.2 Makefiles and compiling 46
5.1.3 Kyber is not an integrated part of ChipWhipserer 46
5.1.4 Endianess and SimpleSerial 47
5.2 Areas of improvement and future research 48
5.2.1 Intermediate values 48
5.2.2 A different attack 49

6 Conclusion **51**

References **53**

List of Figures

2.1 A 2-dimensional lattice and a good basis 8
2.2 A 2-dimensional lattice and a bad basis 8
2.3 A sequence diagram of a KEM 11
2.4 5 incrementing loops 18
2.5 8 incrementing loops 18
2.6 Multiplication by 11 in 10 loops 18
2.7 Division by 11 in 10 loops 18
2.8 Illustration of a Vertical CPA attack 21

3.1 The ChipWhisperer-Lite capture board in black, STM32-f3 taget board in blue and UFO board in red 25
3.2 Contents of the makefile in /crypto/kyber512_clean 26
3.3 Contents of makefile in /crypto 26
3.4 Contents of makefile in hardware/victims/firmware/simpleserial-kyber512 27
3.5 Running commands to build Kyber512 27
3.6 Kyber512 running on CW 29
3.7 indcpa_dec 31
3.8 Definition of "poly"-structure in c 31
3.9 Definition of "polyvec"-structure in c 31
3.10 polyvec_basemul-function in c 32
3.11 basemul-function in c 32
3.12 Definition of basemul-function in c 32
3.13 fqmul-function in c 32
3.14 Capturing of traces in Jupyter Notebook 33
3.15 Traces with triggers around basemul_montgomery 33
3.16 SimpleSerial functions in basemul_montgomery 35
3.17 Function to intercept the intermediate ciphertext and secret key bytes . 35
3.18 The fast_pearsson function 36
3.19 Full script of the attack 36

4.1 Plot of each max PCC-value for each key guess 40
4.2 PCC-values for one correct and one wrong guess with low correlation . . 41
4.3 PCC-values for two correct guesses 42

4.4 PCC-values for one correct (2626) and one wrong (2048) guess 42

List of Tables

2.1 Parameters for Kyber 15
2.2 Kyber security levels 15

Algorithms

2.1 Kyber.CPAPKE.KeyGen() 13
2.2 Kyber.CPAPKE.Enc(pk, m, r) 13
2.3 Kyber.CPAPKE.Dec(sk, c) 13
2.4 Kyber.CCAKEM.KeyGen() 14
2.5 Kyber.CCAKEM.Enc(pk) 14
2.6 Kyber.CCAKEM.Dec(sk, c) 14

List of Acronyms

CCA Chosen Ciphertext Attack.

CPA Correlation Power Analysis.

DPA Differential Power Analysis.

ECC Eliptic Curve Cryptography.

FO Fujisaki-Okamoto.

HW Hamming Weight.

IFP Integer Factorization Problem.

IND-CCA2 Indistinguishable under Adaptive Chosen Ciphertext Attack.

KEM Key-Encapsulation Mechanism.

LWE Learning With Errors.

MLWE Module Learning With Errors.

NIST National Institute of Standards Technology.

NTNU Norwegian University of Science and Technology.

NTT Number-Theoretic Transformation.

PCC Pearson Correlation Coefficient.

PKC Public-Key Cryptosystem.

PKE Public Key Encryption.

PQC Post-Quantum Cryptgraphy.

PQM4 Post-Quantum M4.

RSA Rivest-Shamir-Adleman.

SCA Side-Channel Attack.

SPA Simple Power Analysis.

Chapter 1
Introduction and background

Cryptography plays a crucial role in secure information transmission over networks, providing confidentiality to the transmitted data. This confidentiality ensures that the information can only be accessed and read by the intended recipients. This is highly important for a variety of data types. Whether we're dealing with government secrets that demand high levels of security or casual content like internet memes, encryption is the go-to strategy for secure transmission.

One of the prevalent methods for achieving fast and secure encryption over networks is the use of a Key-Encapsulation Mechanism (KEM). KEM operates based on a Public-Key Cryptosystem (PKC), widely used in modern encryption methodologies. The concept behind the use of KEM is straightforward yet clever. The mechanism involves the generation and transfer of a symmetric key, which is then employed for symmetric key encryption.

The beauty of using a KEM in encryption lies in its inherent efficiency. By using a public-key cryptosystem for the transfer of a symmetric key, we combine the security advantages of asymmetric encryption with the speed and computational efficiency of symmetric encryption. This combination results in an encryption system that is both secure and fast.

The symmetric key used in this system is crucial because it streamlines the mathematical processing elements of the system. Symmetric key encryption algorithms are computationally less demanding, which allows for faster processing speeds. By using a KEM, we get the best of both worlds - the security of public-key cryptography and the speed of symmetric encryption. This makes KEMs an ideal choice for secure and efficient data encryption over networks.

In 1977 [DH76], the Rivest-Shamir-Adleman (RSA) public-key cryptography cryptosystem PKC was introduced. As of 2023, this cryptosystem is still widely in use. RSA is based on the hard problem of finding two prime factors of a "large"

composite number. The problem has since its introduction been a sufficiently hard problem to assume no classical computer can break the security of the cryptosystem, given the right parameters. However, the introduction of quantum computers poses a serious threat to the problem of which RSA is based.

In 2016, the National Institute of Standards Technology (NIST) introduced an open competition, calling for proposals for a new cryptographic standard, able to withstand quantum computers. The competition had two sections for which submissions could be sent: 1) key generation and KEM, 2) digital signatures. In 2022 the winners of the two sections were announced. For digital signatures, three algorithms were selected. For Key generation and KEM, only one remained the winner: CRYSTALS Kyber.

The purpose of this thesis is to explain how Kyber works, both mathematically, and in code, how it can be implemented on a micro-controller, and how the implementation of the algorithm may contain vulnerabilities. The thesis looks at how Kyber can leak information when commands are being executed, allowing an attacker to extract secret information and ultimately recover the secret key.

1.1 Research Questions

The research questions stated in the project proceeding this thesis (TTM4502) [Grü22] will be used in this thesis, with slight modifications. The research questions will be denoted as **RQ1** and **RQ2**.

RQ1: *How can ChipWhisperer be used to measure the power consumption of cryptographic algorithms?*

RQ2: *How can we use the information from 1) to construct and execute a side-channel attack on Kyber?*

RQ1 requires a deep understanding of and experience with the ChipWhisperer tool. **RQ2** uses the information from the **RQ1** in combination with a thorough understanding of the Kyber algorithm in order to examine it's vulnerable parts. **RQ2** is highly dependant on **RQ1**. In order to properly answer **RQ2**, Kyber must be examined both mathematically and in code.

1.2 Thesis Structure

The following is an explanation of the thesis structure.

Chapter 1: The background of the thesis, quantum computers and how NIST prepares for the arrival of quantum computers.

Chapter 2: The theory of lattice-based cryptography, KEM, Kyber and side-channel attacks.

Chapter 3: The methodology the thesis. How Kyber is applied to the ChipWhisperer and how the attack is executed.

Chapter 4: An analysis of the results. What worked as intended, what results were achieved and how they can be analysed.

Chapter 5: Discussion of what was good with the methodology and what could have been better. Also introduces difficulties, pitfalls, areas of improvement and future research.

Chapter 6: Conclusion and answering the research questions.

1.3 Quantum Computers

Quantum computing is a revolutionary field of computer science that utilizes the principles of quantum mechanics, a theory in physics that describes the behaviors of particles, such as atoms and subatomic particles like electrons and photons.

Quantum computers operate quite differently than classical computers. Whereas traditional computers rely on bits, which can take the value of either a 0, or a 1, quantum computers use another form of bits, namely "qubits".

Unlike classical bits, qubits can exist in multiple states at once, thanks to the property of "superposition". When a qubit is in a superposition-state, the qubit may take the value of both 0 and 1, simultaneously, until measured. [Nan20]

Another key principle of quantum computing is entanglement, wherein the state of one qubit becomes linked to the state of another, no matter the distance between them. Changes to one qubit will instantaneously affect its entangled partner, a feature that allows quantum computers to process information in complex, interconnected ways that are currently impossible for classical computers. [Nan20]

The properties of superposition and entanglement increased the computational powers of quantum computers vastly, compared to classical computers. [MN19] The properties of quantum computers, not only allow for better computational power, but also new ways to attack. Shor's Algorithm [Qua21] is an algorithm developed already in 1994 by Peter Shor. The purpose of the algorithm is to guess the prime factors of a composite integer. The algorithm consists of two parts: a classical part and a quantum part. For the algorithm to work properly, both parts need to be applied. So far, only the classical part has been able to be executed, but with quantum

computers, the algorithm can applied in its entirety, which may be the demise of the RSA PKC.

However, although quantum computers may seem superior to classical computers, there are several problems delaying the arrival of conventional quantum computers [Gia23]. On the other hand, the research on quantum computers is increasing rapidly. There are vast uncertainties regarding when a fully-functional quantum computers, capable of breaking RSA, will arrive. IBM has stated in a blog-post that they will have a quantum computer ready in 2025 [Gam22], whereas a McKinsey report for 2020 states that there could be between 2,000 and 5,000 quantum computers in the world by 2030 [HMP20]. Conclusively, there is a possibility that quantum computers powerful enough to break today's encryption standards will arrive within the next 10 years.

Even though the arrival of quantum computers will pose several problems in the future, the actions we are currently taking may increase the impact of the problems. "Harvest now, decrypt later" [Dur23] is a problematic theory which introduces the problem of that adversaries may already be "harvesting" data encrypted with non-quantum resistant schemes, which subsequently will be decrypted when conventional quantum computers arrive. This significantly underlines the importance of Post-Quantum Cryptgraphy (PQC) and the fact that it probably should be applied sooner, rather than later.

1.4 NIST PQC

Widely used public key encryption standards, e.g. RSA and Eliptic Curve Cryptography (ECC), base their security on hard mathematical problems which are considered infeasible for classical computers to crack without sufficient information about the secret components used in the encryption scheme. Given a correct implementation and use of parameters, classical computers need about 300 trillion years to brute force a the secret key of RSA2048 [Dac22].

RSA is based on the "Integer Factorization Problem (IFP)" [Len11], which states that it is hard for a computer to find two large prime factors of a composite integer. Shor's Algorithm [Qua21] with the combination of a quamtum computer is, however, able to solve the hard mathematical problems of RSA and ECC. Therefore, a need for new cryptographic standards has emerged, resulting in the NIST PQC petition.

NIST has been playing a key role in the development and standardization of quantum-resistant cryptographic algorithms, also known as PQC. This effort is of paramount importance, as quantum computing poses a significant threat to the security of current cryptographic systems. Once quantum computing reaches a level

where it can break today's public key cryptography, significant portions of our digital infrastructure could be at risk.

NIST initiated its PQC Standardization process in 2016[oSta23] to solicit, evaluate, and standardize quantum-resistant public key cryptographic algorithms. The process has involved multiple stages of evaluation, which includes public comment periods and third-party analysis of the proposed algorithms.

The goal of this project is to select one or more algorithms for each of the following cryptographic primitives: public-key encryption and key-establishment algorithms, and digital signatures. These new algorithms will be resistant to both classical and quantum computer attacks, ensuring the continued security of digital communications in a post-quantum world.

NIST issued a call for proposals in in February 2016. Researchers and experts submitted their proposals of quantum-safe encryption schemes. By the deadline, in Novemeber 2017, there was a total of 69 proposals of both digital signature schemes and Key Encapsulation Mechanisms, which were considered "complete and proper". The standardization process has continued, filtered out multiple schemes over a period of several years, as well as a total of four rounds of submissions. In 2022, one Public-key Encryption and Key-establishment algorithm remained: CRYSTALS Kyber.

1.5 Related work and other contributions

The focus on quantum-computers and how it can break today's encryption is becoming increasingly popular. The paper from [KdG21] has been a great motivation and guide for this thesis, managing to execute a side-channel attack on a specific implementation of Kyber. Highly advanced attacks, such as [Guo23] has managed to execute an attack on Kyber with a small sample of leaked information. The "pqm4"-library [KPR+] is a continuously updated library purely created for testing and benchmarking post-quantum algorithms.

Chapter 2
Theory

This chapter will give a brief introduction to lattice-based cryptography, the Learning With Errors (LWE) and Module Learning With Errors (MLWE) problems, the definition of the Kyber algorithm and a KEM, as well as an introduction to Side-Channel Attack (SCA) with specific focus on the Correlation Power Analysis (CPA) attack. Some sections contain re-used information from the project proceeding this thesis (TTM4502) [Grü22], with adjustments and additions to have the information better align with the scope of this thesis.

2.1 Lattice-based cryptography

In lattice-based cryptography, the fundamental building blocks are lattices, which are geometric structures formed by an infinite set of points in a multi-dimensional space. These lattices exhibit rich mathematical properties and offer a fertile ground for developing cryptographic schemes that resist attacks from classical and quantum adversaries.

A lattice is a set of points in an n-dimensional space, generated by n-linearly independent vectors, $v_1, \ldots, v_n \in \mathbb{R}^n$. The set vectors is called a *basis* for the lattice. Equation 2.1 shows the mathematical expressions of a lattice. [Reg06]

$$L(v_1, \ldots, v_n) := \left\{ \sum_{i=1}^{n} a_i v_i \,|\, a_i \in \mathbb{Z} \right\} \tag{2.1}$$

In other words, a lattice is the set of points generated by all integer linear combinations of the vectors in the basis.

In a 2-dimensional space, consider two linearly independent vectors, denoted as r_1 and r_2. Figure 2.1 illustrates a subset of the points that can be reached by combining these vectors in various ways. Furthermore, let c be a point in the space,

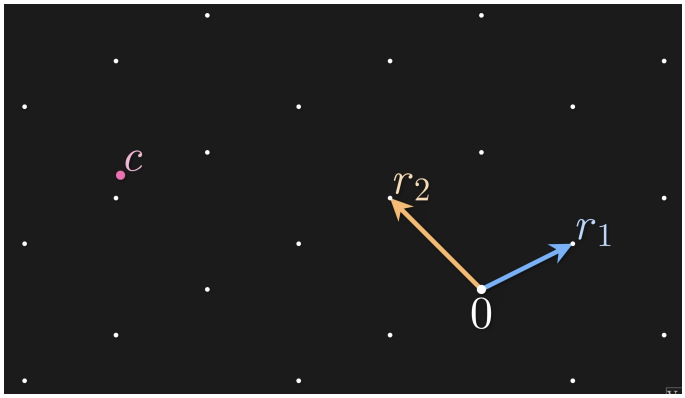


Figure 2.1: A 2-dimensional lattice and a good basis

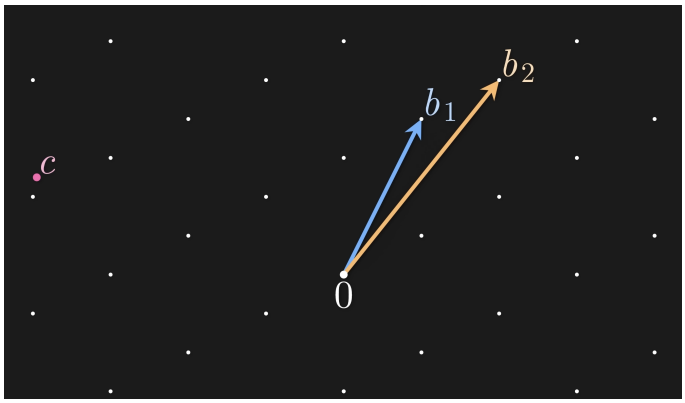


Figure 2.2: A 2-dimensional lattice and a bad basis

close to a lattice point. The key problem of lattice-based cryptography is finding the combination of vectors which reach the point closest to c. [Laa15]

For the basis of r_1 and r_2, it can easily be observed that the point closest to c can be reached by the combination $2sr_2 + 2(-r_1)$.

The basis r_1 and r_2 is not the only set of vectors which generate this lattice. Figure 2.2 illustrates the same lattice, but with the basis b_1 and b_2.

The shortest combination of b_1 and b_2 to reach the point closest to c, is already more complicated, being $8b_1 + 6(-b_2)$. For a 2-dimensional lattice, this is fairly simple. But as the dimensions increase, the problem becomes significantly harder. Furthermore, without the correct information, the person solving the problem does

not have the lattice points, only the basis of the lattice.

In a lattice-based PKC scheme, the so-called "good" vectors are usually part of a private key, while a set of "bad" vectors is used to form a public key. When a sender wishes to transmit a message, they select a point within the lattice, often adding random noise to obscure the original message. The recipient, using the private key, has the means to 'correct' this noise and recover the original message. However, for an outside observer without access to the private key, determining the original lattice point from the noisy data – essentially solving a high-dimensional 'closest point' problem – is computationally infeasible. This computational hardness forms the basis of the security in these schemes, and it is closely related to a well-known mathematical challenge known as the Learning With Errors (LWE) problem.

2.1.1 Learning With Errors

The LWE problem poses a complex challenge where we must solve a system of linear equations of the form ax = b. Here, 'a' and 'b' are known values, while 'x' is the unknown variable we aim to determine. However, the problem is compounded by the presence of a noisy component added to 'b', making the task of accurately determining 'x' significantly more difficult. Owing to this computational hardness, the LWE problem is widely considered to be resistant to both classical and quantum computing attacks. This resistant nature renders it a suitable foundation for PQC.

The Module Learning With Errors (MLWE) problem introduces a higher degree of complexity to the traditional LWE problem by incorporating the concept of module lattices. Unlike standard lattices, module lattices are defined over rings of polynomials, resulting in an intricate structure that is inherently more efficient to manipulate. Similar to the LWE problem, the MLWE problem also necessitates the solution of a system of linear equations - yet in this case, these equations are situated over a ring of polynomials, rather than standard integers.

The MLWE problem retains the presumed hardness of the LWE problem, making it a suitable foundation for secure cryptographic systems. Furthermore, its structure allows for more efficient cryptographic constructions, which is why it's used in encryption schemes like Kyber.

2.2 Key Encapsulation Mechanism

A Key Encapsulation Mechanism (KEM) takes use of a PKE scheme to transfer a key which can be used for symmetric key encryption, such as AES. The PKE-scheme is transformed into a KEM, using a Fujisaki-Okamoto (FO) transform. The details of

the FO transform will not be explained in this thesis. The FO transform does make the KEM secure against chosen ciphertext attacks (CCA), making it "CCA-secure".

Symmetric encryption is generally faster, requires less storage and often have built-in hardware-based encryption.

A KEM consists of three main functions:

1. Key Generation:

   keyGen():
   Input: None
   Output: publicKey, privateKey

   Identical to any regular PKE scheme. A public and private keypair is generated randomly.

2. Encapsulation:

   keyEncaps():
   Input: publicKey
   Output: ciphertext, symmetricKey

   Contrary to a PKE scheme, the encapsulation function does not have a message (plaintext) as input. The encapsulation function takes the public key generated from keyGen(), and randomly generates a symmetric key. The symmetric key will then be encapsulated, using the public key of the recipient.

3. Decapsulation:

   keyDecaps():
   Input: ciphertext, privateKey
   Output: symmetricKey

   The recipient will then receive the ciphertext. The ciphertext is decapsulated using the recipient's private key, and gets the symmetric key, hereby denoted as the "shared secret".

Conclusively, there are a total of 3 (4) keys used in a KEM: The private and public keys, generated by Alice. Alice keeps the private key and and issues the public key. The shared secret, of which both Alice and Bob possess a copy, used for symmetric encryption.

Figure 2.3 illustrates the flow of messages during the process of a KEM between two participants.

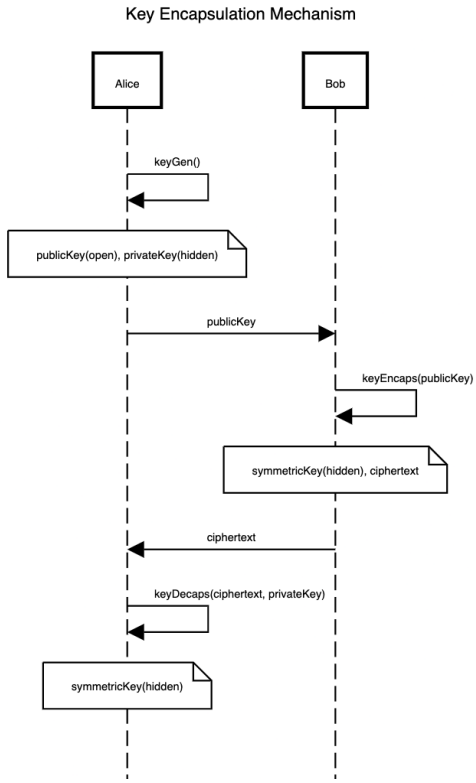


Figure 2.3: A sequence diagram of a KEM

2.3 CRYSTALS Kyber

This section will explain the Kyber algorithm Kyber has three main implementations with varying security levels and complexity: Kyber-512, -768 and -1024. The specifications from [ABD+19] will be used for this thesis. Multiple parts of the specifications, such as the several mathematical definitions, are out of scope for this thesis and will not be explained. Therefore, the algorithms contain simplifications.

2.3.1 The Kyber Algorithm

The Kyber algorithm is based on the MLWE problem. Kyber is an "Indistinguishable under Adaptive Chosen Ciphertext Attack (IND-CCA2)"-secure algorithm, meaning an attacker cannot distinguish between the encryptions of the chosen plaintexts, even when they are allowed to make adaptive chosen ciphertext queries. In other words, an attacker can request the decryption of any ciphertext of their choosing, before

and after they choose the two target plaintexts, as long as the challenge ciphertext is not queried.

The construction of Kyber follows a 2-step approach. Firstly, there is the CPA-secure PKE scheme which encrypts the 32-byte message (shared secret). This will be denoted as Kyber.CPAPKE. Secondly, there is the FO-transformed IND-CCA2-secure KEM, denoted as Kyber.CCAKEM.

[ABD+19] goes in detail of how the Kyber algorithm is constructed, using pseudocode. There will be parts of the explanation which are out of scope for this thesis. Therefore, a slightly simplified version, similar to the explanation from [KdG21] will be given.

Notation

We denote the set of 8-bit unsigned integers $\{0, ...255\}$ as $\mathcal{B}$, i.e bytes. Furthermore, the notation $\mathcal{B}^k$ denotes the array with bytes of length k.

The polynomial ring $\mathbb{Z}_q[X]/(X^n + 1)$ will be denoted as R_q. The parameters n, n' and q are fixed as $n = 256$, $n' = 9$ and $q = 3329$. Vectors are written in bold font and matrices are written with bold, capital letters. A matrix $\mathbf{A}^T$ is the transposed matrix of $\mathbf{A}$. For an $x \in \mathbb{Q}$, we use $\lceil x \rfloor$ as the rounding of x to the closest integer, where ties are rounded up.

For the explanation of Kyber, [ABD+19] firstly explains the three algorithms:

1. $Parse : \mathcal{B}^* \rightarrow R_q^n$
2. $CBD_\eta : \mathcal{B}^{64\eta} \rightarrow R_q$
3. $Decode_\ell : \mathcal{B}^{32\ell} \rightarrow R_q$

While this thesis will not delve into the details of these specific algorithms, they are fundamentally utilized within the subsequent algorithms.

Kyber.CPAPKE

Algorithm 2.1 briefly describes the process of generating a keypair. However, as previously described, this is a simplification. We can observe in line 6 and 7 that the public key is generated as $pk = \mathbf{As} + \mathbf{e}$ and the private key as $sk = \mathbf{s}$. It is important to observe the error vector, $\mathbf{e}$. This error vector introduces the MLWE problem, as it represents the random noise added.

Algorithm 2.1 Kyber.CPAPKE.KeyGen()

Output: Secret key sk
Output: Public key pk
Generate matrix $\mathbf{A} \in \mathbb{R}_q^{k \times k}$
Sample $\mathbf{s} \in \mathbb{R}_q^k$ from B_{η_1}
Sample $\mathbf{e} \in \mathbb{R}_q^k$ from B_{η_2}
$pk := \mathbf{As} + \mathbf{e}$
$sk := \mathbf{s}$

Algorithm 2.2 Kyber.CPAPKE.Enc(pk, m, r)

Input: Public key pk
Input: Message m
Input: Random coins r
Output: Ciphertext c
Generate matrix $A \in \mathbb{R}_q^{k \times k}$ in NTT domain
Sample $\mathbf{r} \in \mathbb{R}_q^k$ from B_{η_1}
Sample $\mathbf{e}_1 \in \mathbb{R}_q^k$ from B_{η_2}
Sample $\mathbf{e}_2 \in \mathbb{R}_q$ from B_{η_2}
$\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v := \mathbf{t}^T\mathbf{r} + \mathbf{e}_2 + \text{Decompress}_q(m, 1)$
$c := (\text{Compress}_q(\mathbf{u}, du)||\text{Compress}_q(v, dv))$

From 2.2 we can observe the three inputs: pk, m and r and one output, i.e. the ciphertext c. In line 11, of Algorithm 2.2, we can observe how the ciphertext is the concatenation: $c = (\text{Compress}_q(\mathbf{u}, du)||\text{Compress}_q(v, dv))$. This concatenation is equal to $c = c_1||c_1$, where $c_1 = (\text{Compress}_q(\mathbf{u}, du)$ and $c_2 = \text{Compress}_q(v, dv))$. This will later be observed in Chapter 3.

Algorithm 2.3 Kyber.CPAPKE.Dec(sk, c)

Input: Secret key sk
Input: Ciphertext c
Output: Message m
$u = \text{Decompress}_q(c1, du)$
$v = \text{Decompress}_q(c2, dv)$
$m = \text{indcpa dec}(sk, m)$ // $m = v - sk^T u$
$m = \text{Compress}_q(m, 1)$

Kyber.CCAKEM

2.4, 2.5 and 2.6 are the pseudo-codes for the three main functions of a KEM.

Algorithm 2.4 Kyber.CCAKEM.KeyGen()

Output: Secret key sk
Output: Public key pk
$z \leftarrow \mathcal{B}^{32}$
$(pk, sk') \leftarrow$ Kyber.CPAPKE.KeyGen()
$sk := (sk'||pk||H(pk)||z)$
return (pk, sk)

Algorithm 2.5 Kyber.CCAKEM.Enc(pk)

Input: Public key pk
Output: Shared Secret K
Output: Ciphertext c
$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow H(m)$
$(\overline{K}, r) \leftarrow G(m||H(pk))$
$c \leftarrow$ Kyber.CPAPKE.Enc(pk, m, r)
$K \leftarrow$ KDF$(\overline{K}||H(c))$
return $(c, \overline{K})$

Algorithm 2.6 Kyber.CCAKEM.Dec(sk, c)

Input: Ciphertext c
Input: Secret key sk
Output: Shared secret K
$pk \leftarrow sk + \frac{12kn}{8}$
$h \leftarrow sk + \frac{24kn}{8} + 32$
$z \leftarrow sk + \frac{24kn}{8} + 64$
$m' :=$ Kyber.CPAPKE.Dec(s, c)
$(\overline{K'}, r') := G(m'||h)$
$c' :=$ Kyber.CPAPKE.Enc(pk, m', r')
if $c = c'$ **then**
    $K :=$ KDF$(\overline{K'}||H(c))$
    return $\overline{K}$
else
    $K :=$ KDF$(z||H(c))$
    return K
end if

2.3.2 Kyber Parameters

Tables 2.3.2 and 2.2 give the parameters for the current versions of Kyber[Gon21]. This thesis will focus on an implementation of the Kyber512 version. For all versions of Kyber, the shared secret, ultimately used for symmetric key encryption, is 32 bytes in size. Table 2.2 states the security levels of the Kyber versions. It should be emphasized that it is not trivial to compare the security levels of quantum-resistant algorithms and non-quantum-resistant ones.

The following parameters in Table 2.3.2 are defined as:

- n: The maximum degree of a polynomial
- k: Amount of polynomials per vector
- q: Modulus for numbers/coefficients
- η_1, η_2: Maximum value of coefficients in "small" polynomials
- d_u, d_v: Control how much (u, v) get compressed
- δ: Probability of a wrong decryption result

Name	**n**	**k**	**q**	$\boldsymbol{\eta_1}$	$\boldsymbol{\eta_2}$	$\mathbf{d_u}$	$\mathbf{d_v}$	$\boldsymbol{\delta}$
Kyber512	256	2	3329	3	2	10	4	2^{-139}
Kyber768	256	3	3329	2	2	10	4	2^{-164}
Kyber1024	256	4	3329	2	2	11	5	2^{-174}

Table 2.1: Parameters for Kyber

Table 2.2 shows the different security levels of Kyber, in comparison to AES.

Version	**Security Level**	**Private Key Size**	**Public Key Size**	**Ciphertext Size**
Kyber512	AES128	1632	800	768
Kyber768	AES192	2400	1184	1088
Kyber1024	AES256	3168	1568	1568

Table 2.2: Kyber security levels

2.4 Side-Channel attacks

(This section thoughout the subsection "Vertical CPA attack" contains partly re-used information from the pre-project from TTM4502 with necessary adjustments to align better with this thesis.)

A side-channel attack (SCA) is an attack on a cryptographic system, based on additional information such as power consumption, execution time, electromagnetic radiation, sound, etc. In contrast to mathematical cryptographic attacks, instead of finding a way to break through the mathematical hard problem of the algorithm, SCA exploits vulnerabilities in hardware implementation. Knowledge about how the algorithm is, however, still crucial. NIST explains SCA as: *An attack enabled by leakage of information from a physical cryptosystem.*[oST17] SCA is a powerful tool, which, if properly executed, can crack most cryptographic systems. To further emphasize the impact of SCA, it can be observed that in the specifications [ABD+19] of Kyber, it is specified that Kyber will be vulnerable to Differential Power Analysis (DPA) (an advanced power analysis attack). The specifications further states that certain countermeasures can be implemented in order to mitigate the probabilities of a successful attack.

The idea of SCA is to attack via information leakage on a system. One of the first official SCA attacks is explained by P. Wright in [Wri87]. In 1965, MI5 tried to break a cipher used by the Egyptian Embassy in London, but failed, due to lack of computational power. Wright then placed a microphone near the rotor-cipher device in order to listen to the sounds the rotors made. The additional information from the sounds of the rotors made MI5 able to crack the ciphering mechanism and spy in the Egyptian Embassy for years. [ZF05]

FIPS 140-2 (superseded by 140-3, not yet implemented) is a standard, created by NIST, which standardizes security levels of cryptographic components. 140-2 is dated back to 2002, yet re-evaluated every fifth year. 140-2 states 11 areas and their specifications needed to achieve a certain security level. The levels span from 1 (lowest) to 4 (highest). The areas include ports and interfaces, roles, authentication, key management, etc. In order to achieve a certain security level of an area, the standard specifies what requirements are needed. Section 4.11 **"Mitigations of other attacks"** of the standard states that cryptographic components may be susceptible to attacks for which: *"testable security requirements were not available at the time this version of the standard was issued"* [oST02]. In other words, side-channel attacks were not taken into consideration when the standard was originally released. It is further stated that if components are designed to be resistant to such attacks, it should be documented, such that it can be reviewed when the mitigations are part of a standard. FIPS 140-3 has been approved and is currently under testing [oSta19],

yet it does not include direct specifications against side-channel attacks.[oST19]

Side-Channel Attacks have three separate classifications, depending on how the attack is executed. The classifications are: control over the computation process, the way of accessing the module and the method used for the analysis. [ABCS06]

For the control over the computation process, there are two main methods of attack. A *passive attack* is an attack in which the attacker does not interfere with the operation of the target, whereas in an *active attack* the attacker exerts some influence on the target. For how the module is accessed, there has been defined three approaches. An *invasive attack* is an attack in which the attacker can deconstruct the device, and e.g. add a probe on a bus to measure values. A *semi-invasive attack* is an attack in which the attacker has access to the device but does not damage the physical layer or make electrical contact other than what already exists. A *non-invasive attack* is an attack in which the attacker does not physically interfere with the device, but can still measure leaked information. This attack is, in most cases, completely undetectable. [ABCS06] The method used for analysis simply describes how to analyze the traces received when measuring power. For the attack on Kyber in this thesis, a passive attack is executed, as the algorithm itself remains unchanged. Whether the attach is an invasive or semi-invasive is not completely straightforward. Triggers are applied in the software of the code, in order to mitigate the amount of samples captured. However, the ChipWhisperer uses built-in hardware in order physically measure the power consumption.

2.4.1 Power Analysis

A power analysis attack is based on measuring the power consumption of a device when it is performing cryptographic operations. The goal of a power analysis attack is to look at one (or more) plot(s) retrieved from the measuring power consumption, analyze the plots and ultimately retrieve a secret key. The plot is made of one (or more) trace(s), which is an array of measured electrical power. Each power trace has a given amount of samples, which will be illustrated by the x-axis of the plots. For the ChipWhisperer implementation the traces are normalized around 0, making the samples span within the values of -0.5 and 0.5. Figures 2.4 and 2.4 are illustrations of work done in TTM4502 [Grü22]. The figures illustrate the pure basics of how to analyze different traces. For the simple traces in the figures, the attacker can easily observe a pattern and make reliable hypotheses of the underlying operations being executed.

The differences between Figures 2.4 and 2.5 and Figures 2.6 and 2.7 indicate the difference in required samples to observe the executed operations. Operations which are executed over several clock cycles, require more samples. This information is important for the attack on Kyber.

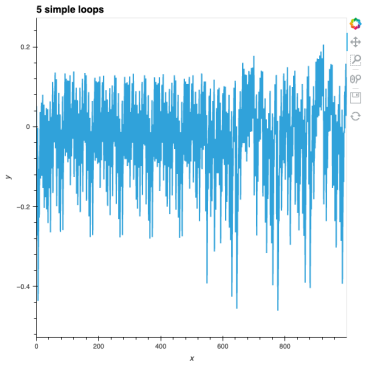


Figure 2.4: 5 incrementing loops

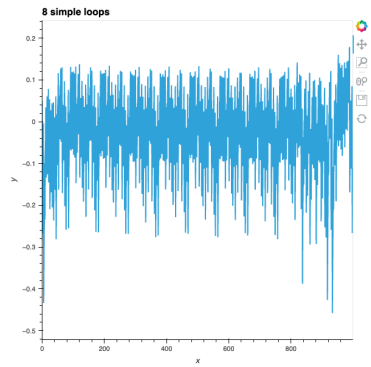


Figure 2.5: 8 incrementing loops

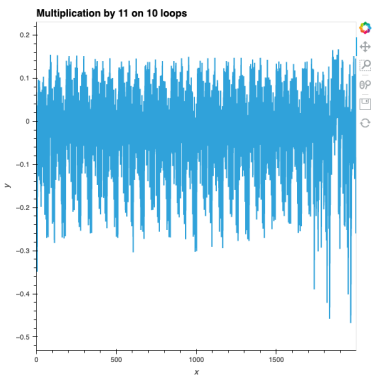


Figure 2.6: Multiplication by 11 in 10 loops

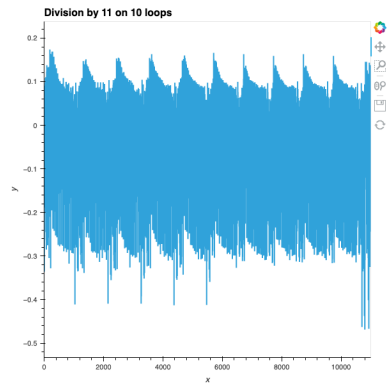


Figure 2.7: Division by 11 in 10 loops

As stated in TTM4502 [Grü22], power analysis attacks can be divided into Simple Power Analysis (SPA), Differential Power Analysis (DPA) and Correlation Power Analysis (CPA). SPA is already explained in the TTM4502, therefore it will not be explained in this thesis. The CPA attack is the attack used in this thesis, so it will be thoroughly explained. CPA and DPA are terms often used as each other. As [ABD+19] stated that Kyber is vulnerable to DPA attacks, it is safe to assume it also is vulnerable to CPA attacks. The attacks are, in detail, different types of attacks, yet the fundamentals of the attacks remain the same: gather enough data to perform a statistical attack. As they both are quite similar in nature, the terms have a tendency to get mixed up. The DPA attack works similarly to CPA, but instead of looking at the correlation of the guesses and the traces captured, the DPA attack

looks at the difference between the traces.

For this thesis, the most relevant attack on Kyber is a CPA attack. The reason for this is partly because SCA attacks are more widespread. Whenever scientist attempt to execute advanced power analysis attacks, SCA is generally used (in combination with even more advanced techniques, such as machine learning). This causes the amount of data, research papers and theory of SCA to outweigh DPA.

2.4.2 Correlation Power Analysis

For CPA attacks, multiple several traces with a large amount of samples is generally needed. TTM4502 [Grü22] went into detail on how a CPA attack could be executed on AES128. In that CPA attack, the method of operations was to encrypt multiple plaintexts with the same symmetric private key. The more traces captured, the higher the probability of the best correlating guesses actually being the right key bytes.

A commonly used hypothesis, is that the Hamming weight of an operation will correlate with samples measured on a power trace. The Hamming weight refers to the amount of 1s in a binary representation, and in a variety of digital systems, power consumption correlates with the Hamming weight of the data being processed. This is due to the power required when a binary bit transitions from 0 to 1 or 1 to 0, with more simultaneous transitions leading to higher power usage. By observing the power trace during a cryptographic operation, an attacker can estimate the Hamming weight of the data being processed, potentially revealing sensitive details. [Owe17]

A CPA attack generally consists of four steps:

1. Data Collection:

   Firstly, an attacker needs to collect traces of the device's power consumption while it's processing a cryptographic operation. The attacker would also need to know the plaintext or ciphertext associated with each trace. This causes the attacker to have both public information, as well as leaked information in the form of power traces. This thesis uses the ChipWhisperer toolkit in order to gather the power traces. For the attack posed on Kyber, the ciphertexts are pre-calculated, simulating a scenario in which an attacker has "sniffed" multiple chiphertexts. The ChipWhisperer open-source software also contains built-in functions to generate key guesses, plaintext and ciphertext.

2. Hypothesis Phase:

   The attacker generates a hypothesis about the value of a part of the secret key. Based on this hypothesis, the attacker guesses an intermediate value of the

cryptographic operation for each trace. From TTM4502, this was the S-BOX operation in AES. For this thesis, it will be observed how the multiplication of coefficients can be such a sensitive point. The function that maps the key hypothesis and known plaintext/ciphertext to the intermediate value is often called the "leakage model". A fundamental principle of this phase, is that the Hamming weight of the operation will correlate with the specific sample in which the operation takes place.

3. Correlation Calculation:

   The attacker calculates the Pearson Correlation Coefficient (PCC) between the observed power traces and the predicted power consumption from the leakage model. This is done for each time sample in the traces. The correlation is calculated separately for each key hypothesis. $\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$

   he PCC takes in two sets of data and returns a number, ρ, between -1 and 1, based on the linear correlation of the numbers. If ρ is -1 or 1, there is an accurate, linear correlation between the numbers. If $0 < \rho < 1$ or $-1 < \rho < 0$, it is possible to find a linear correlation line between the numbers. However, if $\rho = 0$, there is no correlation. In other words, given two input numbers, X and Y, we can say that if Y always increases when X increases, ρ will be 1. If Y always increases when X decreases, ρ will be -1. If ρ is between 1 and -1 but not equal to 0, there is a correlation.

4. Key Selection:

   In CPA attacks, high correlation means a good guess. The more traces are captured, the higher the probability of a good guess actually is a correct guess. The key hypothesis that results in the highest correlation with the power traces is considered the most likely correct value of the key.

5. (Repeat for All Key Parts)

Vertical CPA

As described in Section 2.4.1, a power trace is the measured power consumption of a device during the execution of operations. Often, even with ChipWhisperer, it can be difficult to completely isolate and measure the exact operation of interest in an algorithm. Typically, the attacker will have to measure the power consumption over a variety of commands, knowing the desired operation is amongst them. For this reason, "Vertical CPA attacks" are beneficial. In a vertical CPA attack, the attacker focuses on only one operation at a low level, in which both the known information (ciphertext/plaintext) is used, as well as part of the hidden key. In a vertical CPA attack, it is necessary to have the operation under attack be represented in a single sample. The reason for this is can be explained with the Hypothesis Phase: the

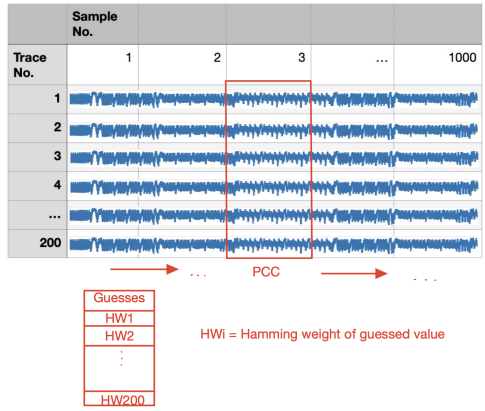


Figure 2.8: Illustration of a Vertical CPA attack

attacker needs to make a hypothesis on a single value. This value will be represented as a sample in the power trace. The attackers guess will be the Hamming weight of the known text combined with a guess for the hidden part of the operation. The theory then states that the Hamming wight of this operation will correlate with the given sample.

When capturing multiple traces, the idea of a Vertical CPA attack is to align the traces and observe the correlation on each sample of each trace. For the attack posed in this thesis, 200 traces are gathered, with 1000 samples for each trace. The Vertical CPA attack calculates the correlation between an array of guesses and the jth sample of all the traces. E.g. if a multiplication operation is executed at sample 100, the correlation between the guesses and the 100th sample from all the traces should yield a much higher correlation than any other sample number in the traces.

Figure 2.8 illustrates how a Vertical CPA attack is executed. The red box containing the Hamming weights of each guess combined with the ith ciphertext calculates the correlation of all the jth samples. The illustrations show each sample as a trace, when in reality each sample is a number.

The use of the PCC calculation only makes sense if there is an equal amount of values in both arrays. Using a Vertical CPA attack, the traces are transposed, meaning first element of the transposed array represents all the first samples of the 200 traces. The red box from Figure 2.8 is the hypothesized guess for each ciphertext. I.e. the first value of the red box (HW1) represents the Hamming weight of the first ciphertext and a fixed key byte guess. The second element (HW2) is the second ciphertext combined with the same fixed key guess used for HW1. For the attack on Kyber, it will be observed that the red box is denoted as *rst*. The red box is "slided"

along the aligned traces, calculating a correlation for each sample number.

Chapter 3
Methodology

This chapter explains the lab setup, how to capture the leaked information and ultimately, how to use the leaked information to construct and execute an attack. The attack introduced in this thesis is a semi-static setting. This includes the ciphertexts and the secret key being hard-coded into the implementation. A Python implementation of Kyber [Pop] was used to execute Algorithms 2.4 and 2.5. The reasoning for using another implementation of Kyber, is simply to generate and use real ciphertext and keys. If the versions of Kyber being run is the same, it is irrelevant what kind of implementation was used when generating the ciphertext and keys. Therefore algorithms 2.4 and 2.5 were used to generate the static secret key and ciphertexts. Furthermore, testing was done to ensure the decapsulation of the ciphertexts yielded the same results in both implementations.

3.1 Lab Setup

The lab setup includes the ChipWhisperer-Lite board, an STM32F-3 target board with a 32-bit ARM Cortex M4 Processor and a CW308 UFO board. Furthermore, the entirety of the Kyber algorithm is gathered from pqm4 library, available on GitHub. The host computer is my own personal computer, a Macbook Pro M1-pro.

3.1.1 ChipWhisperer

The ChipWhisperer kit is used to research power-analysis side channel attacks and glitching attacks. ChipWhisperer-Lite comes in two main parts: the capture board and the target board [Inc22]. The capture board is used to capture the power traces and transfer them to the host computer via a Micro-USB cable. The traces can then be reviewed and analyzed in order to execute and attack. The target board, on the other hand, is where the algorithms are implemented. In the case of this thesis, the clean implementation of Kyber512 algorithm from the Post-Quantum M4 (PQM4)-library [KPR+] will be implemented onto the target board, and the capture board will send captured power traces to the host computer.

The communication with the ChipWhisperer capture board and the host computer is done in Python. For the lab setup for this thesis, an IDE (Visual Studio Code) was used to modify makefiles used for compiling, apply the static ciphertext and private key, as well as having my own "master"-code ("egentest.c") which states what algorithms to execute. Building and compiling the project, as well as flashing it onto the target board is done via Jupyter Notebook. This allows for individual cells to be re-run when preferred.

ChipWhisperer and NewAE technology is a fully open-source system, including hardware, software, firmware and FPGA-code. NewAE has created a GitHub repository, which includes an entire start-up guide, as well as demo projects with examples on how certain side-channel attacks can be executed. In order for the capture board to communicate with the target board, the SimpleSerial protocol is used in almost all of the demo projects. This allows ASCII-characters to be sent to and from the target board, initiated by the capture board. The data can then be sent back to the host computer. The target board has a buffer of 192 bytes, meaning at most 192 bytes can be sent at once. Due to the fact that most of the demos use SimpleSerial for communication to the target board, this was the method of approach for the thesis as well. The use of SimpleSerial proved crucial for the execution of the attack, as well as for verifying the correctness of decapsulation. The small buffer-size did, however, pose problems when trying to extract the 768-byte sized ciphertexts.

The functionalities of the demos also introduces a severely important topic: the triggers. The triggers are used by the capture board in order to limit when to start and when to stop capturing the leaked power information. The triggers are defined in a "hal.h"-file from the ChipWhisperer Github repository. This allows for the pins which capture the power to initiate and end capturing. For an SCA to work, it is essential to understand the activities in the power traces. For simplicity of the thesis, we do not (and can not with the given equipment) capture the power consumption of the entire algorithm. Instead, we apply the triggers inside the code in a desired area. This allows for analysis to be more easily executed.

3.1.2 The pqm4 library

pqm4 is a crypto library for the M4 processor [KPR+]. It includes Kyber, as well as other KEMs and Digital Signatures. Furthermore, pqm4 has versions of Kyber spanning from 512 to 1024, as well as speed optimized versions, memory optimized versions, clean implementations and "m4"-implementations. The clean implementations are the ones originally proposed to NIST, without any optimizations. This thesis will focus on a combination of the implementations.

Each algorithm is composed of a series of intertwined header (.h)-files, .c-files and, in some cases, .S-files (Assembly). The algorithms from Section 2.3.1 can be

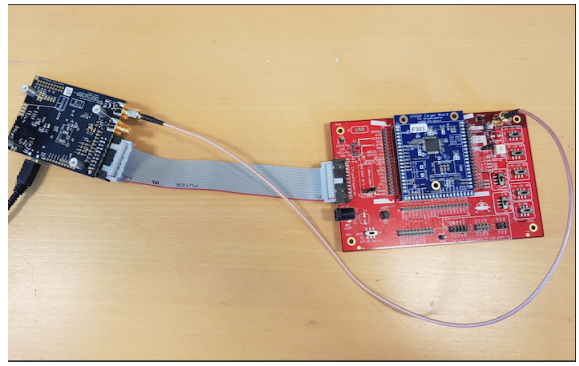


Figure 3.1: The ChipWhisperer-Lite capture board in black, STM32-f3 taget board in blue and UFO board in red

observed throughout the files in pqm4. E.g. The final key generation algorithm, 2.4, can be found in the *kem.c* file, while 2.1 is found in the *indcpa.c* file.

Within pqm4, there are several Python-files which can be used for testing and running benchmarks for the different algorithms. The Pyhton-files are created so that they are flashed over the ChipWhisperer, execute key generation, encapsulation and decapsulation. Finally the results are stored within several folders which can be used for a variety of research- and study cases. In order to compile the projects, the ARM toolchain is required [Devc].

3.1.3 Applying Kyber to ChipWhisperer

For the lab setup, the entire folder containing the algorithm, was copied to the ChipWhisperer's repository. This allows for the Kyber algorithm to easily include both the SimpleSerial-protocol, as well as the functionality needed for the triggers to be applied. For the following explanation, the Kyber512-clean implementation will be used.

As the ChipWhisperer repository has the crypto-algorithms inside the hardware/victims/firmware/crypto folder, the same will be done in this thesis. The folder containing the entire algorithm will be denoted as kyber512_clean. Furthermore, the files "fips202.c (and .h)" and "randombytes.c (and .h)" from pqm4/common must also be copied over. Within the same folder, a makefile must be created. This file explains what header-files and object-files to be compiled during the make-function. Figure 3.2 shows the contents of the makefile. The second makefile is used to link the makefile in /crypto/kyber512 with the final makefile. Figure 3.3 shows the contents of the second makefile in /crypto. The line with the headers (.h-files) may be redundant, yet it works. The final makefile is located in a folder specific

to the Kyber512 algorithm. This folder, named "simpleserial-kyber512", is located in hardware/victims/firmware. This folder is the one being used when building the project in Jupyter Notebook. Figure 3.4 shows the contents of the makefile in hardware/victims/firmware/simpleserial-kyber512. Notice how this makefile has the command "SRC += egentest.c" on line 8. The file "egentest.c" is the file which runs the algorithms and keeps track of which ciphertexts to decapsulate during the decapsulation.

When the makefiles have been created, the correct files are imported and the folders are located in the proper locations, the project can be built. This is done similarly to the other ChipWhisperer demo projects. Figure 3.5 illustrates the steps needed to build the Kyber512 algorithm from pqm4.

```
LIB=libkyber512_clean.a
HEADERS=api.h cbd.h indcpa.h kem.h ntt.h params.h poly.h polyvec.h reduce.h symmetric.h verify.h
OBJECTS=cbd.o indcpa.o kem.o ntt.o poly.o polyvec.o reduce.o symmetric-shake.o verify.o

CFLAGS=-O3 -Wall -Wextra -Wpedantic -Werror -Wmissing-prototypes -Wredundant-decls -std=c99 -I../../../common $(EXTRAFLAGS)

all: $(LIB)

%.o: %.c $(HEADERS)
    $(CC) $(CFLAGS) -c -o $@ $<

$(LIB): $(OBJECTS)
    $(AR) -r $@ $(OBJECTS)

clean:
    $(RM) $(OBJECTS)
    $(RM) $(LIB)
```

Figure 3.2: Contents of the makefile in /crypto/kyber512_clean

```
1   ########
2   CRYPTO_LIB = kyber512_clean
3
4   SRC += cbd.c fips202.c indcpa.c kem.c ntt.c poly.c polyvec.c randombytes.c reduce.c symmetric-shake.c verify.c
5
6   HEADERS=api.h cbd.h indcpa.h kem.h ntt.h params.h poly.h polyvec.h randombytes.h reduce.h symmetric.h verify.h
7
8   VPATH += :$(FIRMWAREPATH)/crypto/$(CRYPTO_LIB)
9
10  EXTRAINCDIRS += $(FIRMWAREPATH)/crypto/$(CRYPTO_LIB)
```

Figure 3.3: Contents of makefile in /crypto

3.2 Setting up attack on Kyber

The attack on the clean implementation on Kyber512 is done during the decapsulation stage. This is due to it being the only part of the algorithm using the secret key. Whereas the pre-project [Grü22] executed an attack on the S-Box substitution step in AES, this thesis also goes in depth of the decapsulation in order to find a vulnerable leakage model. Within the decapsulation stage, there is a segment executing a coefficient multiplication of two polynomials. The polynomials are modified representations of the secret key and the ciphertexts. For the attack, a

hardware > victims > firmware > simpleserial-kyber512 > makefile

```
# Target file name (without extension).
# This is the name of the compiled .hex file.
TARGET = simpleserial-kyber


# List C source files here.
# Header files (.h) are automatically pulled in.
SRC += egentest.c

CRYPTO_TARGET = NONE
#CRYPTO_OPTIONS = DES

# ----------------------------------------------------

ifeq ($(CRYPTO_TARGET),)
CRYPTO_TARGET = AVRCRYPTOLIB
endif

#ifeq ($(CRYPTO_OPTIONS),)
#CRYPTO_OPTIONS = AES128C
#endif

include ../crypto/Makefile.kyber512
#Add simpleserial project to build
include ../simpleserial/Makefile.simpleserial
FIRMWAREPATH = ../.
include $(FIRMWAREPATH)/Makefile.inc
```

Figure 3.4: Contents of makefile in hardware/victims/firmware/simpleserial-kyber512

```
In [1]: SCOPETYPE = 'OPENADC'
        PLATFORM = 'CWLITEARM'
        num_traces = 200
        import numpy as np

In [2]: CRYPTO_TARGET = 'NONE'

In [3]: %%sh -s "$PLATFORM" "$CRYPTO_TARGET"
        cd ../../hardware/victims/firmware/simpleserial-kyber512
        echo $1 $2
        gmake PLATFORM=$1 CRYPTO_TARGET=$2
        +
        Creating Extended Listing: simpleserial-kyber-CWLITEARM.lss
        arm-none-eabi-objdump -h -S -z simpleserial-kyber-CWLITEARM.elf > simpleserial-kyber-CWLITEARM.lss
        .
        Creating Symbol Table: simpleserial-kyber-CWLITEARM.sym
        arm-none-eabi-nm -n simpleserial-kyber-CWLITEARM.elf > simpleserial-kyber-CWLITEARM.sym
        SS_VER set to SS_VER_1_1
        Size after:
           text    data     bss     dec     hex filename

         166068    1772    7604  175444   2ad54 simpleserial-kyber-CWLITEARM.elf
        +--------------------------------------------------------
        + Default target does full rebuild each time.
        + Specify buildtarget == allquick == to avoid full rebuild
        +--------------------------------------------------------
        +--------------------------------------------------------
        + Built for platform CW-Lite Arm \(STM32F3\) with:
        + CRYPTO_TARGET = NONE
        + CRYPTO_OPTIONS =
        +--------------------------------------------------------
```

Figure 3.5: Running commands to build Kyber512

total of 200 ciphertexts was used, thereby using 200 power traces. Each power trace was set to capture 1000 samples. The reasoning behind using 200 traces, is because it is the same amount used in [KdG21]. This should provide a sufficient amount of samples, reducing the probability of completely random results occurring.

The paper [KdG21] has managed to execute an attack on an implementation of Kyber, which in this thesis will be labeled as the "m4"-implementation. In the mentioned paper, the focus is also on the coefficient multiplication in the decapsulation phase. This thesis will use similar logic, but on the clean implementation. [KdG21] has been used as a guideline to execute the same type of attack.

The previous section has already explained how to import Kyber into the ChipWhisperer-repository. The idea behind this was to be able to use the triggers to capture the power traces, as well as being able to use the SimpleSerial protocol for debugging. The SimpleSerial protocol proved to be crucial, not only for debugging, but for executing the attack as well.

The decapsulation of the different ciphertexts yield 200 different shared keys. However, it is not the shared keys which are under attack, it is the static private key used for decapsulation. From Table 2.2, it can be observed that the secret key used for decapsulation is 1632 bytes, all the 200 ciphertexts are 768 bytes. Lastly, the shared secret is only 32 bytes long. With the limited buffer size of 192 bytes in the SimpleSerial protocol, sending 200 ciphertexts of size 768 would be an issue. Therefore, the setup is as follows. The Python-implementation of Kyber512 [Pop] has been used to generate the secret key, as well as the ciphertexts. The secret key is implemented as a static variable in "egentest.c", with the 200 ciphertexts are added in a separate header-file, named "data.h", located in /simpleserial-kyber512. Using this lab-setup, the only necessary part from pqm4 is the decapsulation. The benefits of this are a wide and clear view of all the ciphertexts, as well as the secret key, the possibility to quickly alter amount of ciphertexts to be decapsulated and a variable flashing time, causing a faster method for testing and debugging.

3.2.1 Decapsulation of ciphertexts

As the ciphertexts are stored in a separate header-file, it can easily be included by adding the command "#include data.h" in "egentest.c". One large issue remains for the decapsulation to function properly: when capturing the power traces, how can be make the program only decapsulate one ciphertext per power trace? The initial thought was to simply add the decapsulation method into a loop, and iterate over all the ciphertexts. This would, however, cause the program to decapsulate all the ciphertexts for each power trace, which would be useless for the attack to work properly. The solution was inspired by the password bypass SPA attack from TTM4502 [Grü22]. By creating helper-functions which allow for reading and sending

of information using SimpleSerial, we can send the target board a "counter" value. When capturing the power traces, the counter will be sent from Jupyter Notebook, and read by the target board, using the *my_read* function. The target board will then decapsulate the ciphertext with the same index as the counter, causing each power trace to be the decapsulation of a single power trace. Furthermore, the use of this counter makes it also possible to store all decapsulated ciphertexts (i.e. shared secrets), and finally sending them back to Jupyter Notebook (using the regular *simpleserial_put*-function) verifying the decapsulation has been properly executed.

Following these steps, the program should properly decapsualate each ciphertext properly with the possibility to verification by sending the shared secret back to the host computer. The final step of setting up Kyber on the ChipWhisperer is to add the triggers. As the entire project is located in the ChipWhisperer repository, the functionality of the triggers is already defined in /hardware/victims/-firmware/hal/stm32f3/stm32f3_hal.c, and can easily be imported by adding the command: <#include ..hal/hal.h>, depending on the what target board is being used and the folder-structure of the "attacker". Including this file, allows for the use of <trigger_high()> and <trigger_low()> around the desired functions.

```
59  int main(void)
60    {
61
62      platform_init();
63      init_uart();
64      trigger_setup();
65
66      char input[32];
67
68      my_read(input, 32);
69      int myInt = (input[0] - '0') * 100 + (input[1] - '0') * 10 + (input[2] - '0');
70
71      PQCLEAN_KYBER512_CLEAN_crypto_kem_dec(key_a[myInt],all_ciphertexts[myInt],sk_a);
72      simpleserial_put('r',32,(unsigned char*)&key_a[myInt]);
73
74    return 1;
75    }
```

Figure 3.6: Kyber512 running on CW

Figure 3.6 represents the main-function being set up on the target board. The functions on line 62-64 are functions necessary to use the triggers. On line 66, the array for taking in the counter variable is defined. The size of this could be altered to save storage, however the amount of possible storage to be saved is negligible. The array is then filled with the input from the function on line 68. For it to properly work as intended, it needs to be converted to an integer, as done in line 69. Line 71 is the actual decapsulation. The first parameter is the pointer to the output array of shared keys. The second parameter, *all_ciphertexts* is the 200*768 array containing all the ciphertexts. The final parameter is the static private key used for decapsulation. This is predefined at the beginning of the script. Finally, the *simpleserial_put*-function on line 72 sends back each shared key, verifying the correctness of the decapsulation.

3.3 Attack on Kyber

The attack on Kyber is a vertical CPA attack, explained in Section 2.4.2. The goal of the attack is ultimately be able to extract the entire secret key. However, in order to succeed with extracting the secret key, it must be reconstructed byte by byte. The attack focuses on the extraction of the intermediate values of a set of secret key bytes, $k_0k_1k_2k_3$, which are the first four bytes of the unpacked secret key. When all unpacked bytes are extracted, the key can be packed and correctly reconstructed. Finding $k_0k_1k_2k_3$ proves the attack can be executed on the remaining bytes.

The previous section explained how to build Kyber and flash it onto the ChipWhisperer target board. Furthermore, details on how to execute one decapsulation for each ciphertext is explained. Finally how to apply the triggers were explained.

3.3.1 Data Collection and Hypothesis phase

In order to properly initiate the first phase of a CPA-attack, i.e. the data collection phase, it is necessary to go in depth into the decapsulation algorithm of Kyber. As described in Section 2.4.2, the idea of a CPA-attack is to find a low-level operation of a known value and an unknown value and measure the power consumption of this operation. For Kyber, this means the attacker needs to find such a low-level operation. The triggers could, in theory, have been put around the entire decapsulation-algorithm. This would, however, cause a large data set with a very large amount of operations, i.e. a trace with a large amount of samples. Furthermore, the amount of samples needed would far exceed the amount of possible samples for the ChipWhisperer-Lite. Therefore, it is essential to find the low-level operations within the algorithm and then apply the triggers around this operation.

The function, `PQCLEAN_KYBER512_CLEAN_crypto_kem_dec`, used in "egentest.c" is the same as Algorithm 2.6. This algorithm in turn invokes CPAPKE.Dec(s, c) (Algorithm 2.3). Simply put, the KEM decapsulation process employs the standard Kyber decryption method. Hence, the attack can be specifically targeted at Algorithm 2.3. For the successful setup of the attack, it is critical to understand how the ciphertext is decompressed and divided into the polynomial u and the vector of polynomials v. Line 6 of Algorithm 2.3 calls the `indcpa_dec(sk, m)`, with $m = v - sk^T u$. In the clean implementation, the "unpacking" of the ciphertext is done within the `indcpa_dec`-function, (shown in Figure 3.7) yet it maintains the same functionality.

The thesis will not go in depth on how the decompression is done, however, it can be observed that the function `unpack_ciphertext(&b, &v,c);` on line 319 takes in the ciphertext, c, as argument. The variables $\mathbf{b}$ and v are the respective vector of polynomials and the polynomial. It can also be observed how the secret key, sk

```
313  void PQCLEAN_KYBER512_CLEAN_indcpa_dec(uint8_t m[KYBER_INDCPA_MSGBYTES],
314                                         const uint8_t c[KYBER_INDCPA_BYTES],
315                                         const uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES]) {
316      polyvec b, skpv;
317      poly v, mp;
318
319      unpack_ciphertext(&b, &v, c);
320      unpack_sk(&skpv, sk);
321
322      PQCLEAN_KYBER512_CLEAN_polyvec_ntt(&b);
323      PQCLEAN_KYBER512_CLEAN_polyvec_basemul_acc_montgomery(&mp, &skpv, &b);
324      PQCLEAN_KYBER512_CLEAN_poly_invntt_tomont(&mp);
325
326      PQCLEAN_KYBER512_CLEAN_poly_sub(&mp, &v, &mp);
327      PQCLEAN_KYBER512_CLEAN_poly_reduce(&mp);
328
329      PQCLEAN_KYBER512_CLEAN_poly_tomsg(m, &mp);
330  }
```

Figure 3.7: indcpa_dec

```
10  typedef struct {
11      int16_t coeffs[KYBER_N];
12  } poly;
```

Figure 3.8: Definition of "poly"-structure in c

```
7   typedef struct {
8       poly vec[KYBER_K];
9   } polyvec;
```

Figure 3.9: Definition of "polyvec"-structure in c

is unpacked as well into a vector of polynomials. Table 2.2 explains the size of the polynomial and amount of polynomials per vector. In the case of Kyber512, this means that the variable b is now a vector containing two polynomials, each polynomial having 256 coefficients. The definitions of the "poly"- and "polyvec"-structures are shown in Figure 3.8 and 3.9, respectively. During this stage of the decryption, it is also important to notice how the "poly"-structure is defined as containing a list of 256 "int16_t"-values. This means each coefficient is in the structure of a 16-bit signed integer.

Line 322 in Figure 3.7 calls `polyvec_ntt(&b);`. This perform an Number-Theoretic Transformation (NTT) on each of the polynomials in the vector. Going even deeper into the algorithm, `polyvec_basemul_acc_montomery(&mp, &skpv, &b);` is called. The description of the function is as follows: "Multiply elemnts of a and b in NTT domain, accumulate into r, and multiply by $2^{-}16$", a and b is called as &skpv and &b, respectively and r being &mp. The definition of the function is shown in Figure 3.10. Furthermore, Figure 3.10 also introduces the triggers. The triggers are placed around the first `poly_basemul_montgomery(r, &a->vec[0], &b->vec[0]);`-function. It can be observed that a total of two `basemul_montgomery` functions are called. The first one calls the first elements of the vectors of the variables

a, and b, i.e. $\&skpv$ and $\&b$. The second `basemul_montgomery`-function is called within a loop. However, the iteration starts at $i = 1$ and ends with $i < KYBER\_K$. For Kyber512, $KYBER\_K$ is defined as the total number of polynomials per vector, i.e. 2.

```
142  void PQCLEAN_KYBER512_CLEAN_polyvec_basemul_acc_montgomery(poly *r, const polyvec *a, const polyvec *b) {
143      unsigned int i;
144      poly t;
145      trigger_high();
146      PQCLEAN_KYBER512_CLEAN_poly_basemul_montgomery(r, &a->vec[0], &b->vec[0]);
147      trigger_low();
148      for (i = 1; i < KYBER_K; i++) {
149          PQCLEAN_KYBER512_CLEAN_poly_basemul_montgomery(&t, &a->vec[i], &b->vec[i]);
150          PQCLEAN_KYBER512_CLEAN_poly_add(r, r, &t);
151      }
152
153      PQCLEAN_KYBER512_CLEAN_poly_reduce(r);
154  }
```

Figure 3.10: polyvec_basemul-function in c

The `basemul_montgomery` function can be seen in Figure 3.11. This function calls the `basemul`-functions a total of 64 times (KYBER_N/4). The first basemul takes in the first coefficient of a and b ($skpv$ and b) and performs the operations shown in Figure 3.12. The values of the inputs are then applied in the `fqmul`-function. This function multiplies the values together and reduces them using a Montgomery reduction. It is ultimately the multiplication within the `fqmul`-function the attacker can utilize.

```
223  void PQCLEAN_KYBER512_CLEAN_poly_basemul_montgomery(poly *r, const poly *a, const poly *b) {
224      size_t i;
225      for (i = 0; i < KYBER_N / 4; i++) {
226          PQCLEAN_KYBER512_CLEAN_basemul(&r->coeffs[4 * i], &a->coeffs[4 * i], &b->coeffs[4 * i], PQCLEAN_KYBER512_CLEAN_zetas[64 + i]);
227          PQCLEAN_KYBER512_CLEAN_basemul(&r->coeffs[4 * i + 2], &a->coeffs[4 * i + 2], &b->coeffs[4 * i + 2], -PQCLEAN_KYBER512_CLEAN_zetas[64 + i]);
228      }
229  }
```

Figure 3.11: basemul-function in c

```
176  void PQCLEAN_KYBER512_CLEAN_basemul(int16_t r[2], const int16_t a[2], const int16_t b[2], int16_t zeta) {
177      r[0]  = fqmul(a[1], b[1]);
178      r[0]  = fqmul(r[0], zeta);
179      r[0] += fqmul(a[0], b[0]);
180      r[1]  = fqmul(a[0], b[1]);
181      r[1] += fqmul(a[1], b[0]);
182  }
```

Figure 3.12: Definition of basemul-function in c

```
71   static int16_t fqmul(int16_t a, int16_t b) {
72       return PQCLEAN_KYBER512_CLEAN_montgomery_reduce( (int32_t)a * b);
73   }
```

Figure 3.13: fqmul-function in c

The work behind the first phase of the CPA-attack is now done, and the actual capturing of traces can begin. The only final modifications which do not interfere with the execution of the algorithm, is to apply `simpleserial_put`-functions in order to debug. Section 3.2 elaborated on how a counter was used in order to capture the power traces individually, for each ciphertext. Figure 3.14 illustrates how the traces are captured, using the counter.

```
#Capture Traces
from tqdm import trange

traces = []
for i in trange(num_traces, desc = "Capturing traces"):


    if i < 10:
        char = "00"+str(i)+'\n'
    elif 9 < i < 100:
        char = "0"+str(i)+'\n'
    else:
        char = str(i)+'\n'

    reset_target(scope)
    scope.arm()
    target.flush()
    target.write(char)
    #time.sleep(0.2)
    ret = scope.capture()
    if ret:
        print('Timeout happened during acquisition')
    traces.append(scope.get_last_trace())
```

Figure 3.14: Capturing of traces in Jupyter Notebook

In the script, i is incremented from 0 to 199. The value of i is then stored as a character. The `reset_target(scope`-function is simply a pre-defined function that adds some waiting time and switches the IO-pins off and on. Then, the scope is armed and the buffer of the target is reset. The character is then written to the target board, which reads the character and defines it as the "myInt"-variable, shown in Figure 3.6. With the triggers applied around the first `basemul_montgomery`-function, the plot of all the 200 traces can be seen in Figure 3.15

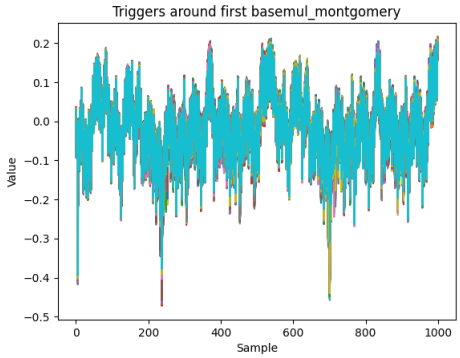


Figure 3.15: Traces with triggers around basemul_montgomery

The attack posed in this thesis focuses on the same parts of the decapsulation as the attack, i.e. the `basemul` functions. However, the attack from [KdG21] is done on the "m4"-implementation, which has implemented Assembly code for several operations, including the two `basemul`-operations. [KdG21] states that finding the first four bytes ($k_0k_1k_2k_3$) with high probability can be done by attacking the first `basemul`. The steps are (direct quote from [KdG21]):

1. Make a guess for k_2k_3 (2^{16} possibilities) and compute the result $rst = [rst_0, ..., rst_{200}$ where rst_i is the Hamming weight of the operation `smultt` (...) using the ith ciphertext.

2. Compute Pearson correlation coefficient between T_i and rst for all i and keep the biggest value in absolute $PCC_{k_2k_3}$.

3. Repeat step 1-2 for all possibilities of k_2k_3 and keep a sample $S = \{k_2k_3$ such that $PCC_{k_2k_3} > x\}$. For example $x = 0.6$.

[KdG21] has previously defined T_j^i as: "Let T_j^i denote the ith point of the jth trace and T_i be the vector such that $T_i = [T_i^1, ..., T_i^{200}]$". This thesis understands this a minor typo, with the intended notation as: "Let T_i^j denote (...)", simply swapping the i and the j. Therefore, this thesis will use the definition: "Let T_i^j denote the ith point of the jth trace and T_i be the vector such that $T_i = [T_i^1, ..., T_i^{200}]$". In other words, T_i is simply the array of traces, transposed, indicating a vertical CPA attack, as explained in Section 2.4.2.

The `smultt` operation is an Assembly instruction, which takes two 32bit integers and multiplies the lower halves of them. I.e. a simple multiplication of two 16-bit integers.
The statement "...using the ith ciphertext" is interpreted as the unpacked and NTT-transformed ciphertext bytes, denoted as $b_0b_1b_2b_3$. From the explanation of the decapsulation previously in this section, the ciphertext is unpacked into a vector of two polynomials and NTT-transformed. This means the ciphertext needs to be modified substantially before the correlation can be calculated. In the "m4"-implementation, the ciphertext is firstly unpacked in the regular c-code and NTT-transformed in Assembly.

From the attacker's point of view, the same functions must be applied to the intercepted ciphertext used in the c-code in order to get the proper correlation. This does add an additional piece of information needed in order to execute the attack; the "zetas". The zetas are values used to perform the NTT-transformation. This problem will be discussed further in Chapter 5.

In order to overcome this problem, there are several solutions. Either, the functions needed to unpack and transform the ciphertext can be recreated, using the same zetas as the implementation. Another solution, which was used for this thesis, is to use the SimpleSerial protocol to send the modified ciphertext bytes directly to the host machine in Jupyter Notebook. SimpleSerial allows for the transmission of both the ciphertext bytes, as well as the unpacked secret key bytes. The unpacked secret key bytes, are not known to the attacker and is only used for testing purposes in this thesis. Adding the `simpleserial_put` commands in an "if"-statement, conditioning if $i = 0$, will give the attacker the first bytes of both the unpacked ciphertext and the unpacked secret key. Figure 3.16 shows how the SimpleSerial functions may be applied.

```
223  void PQCLEAN_KYBER512_CLEAN_poly_basemul_montgomery(poly *r, const poly *a, const poly *b) {
224      size_t i;
225      for (i = 0; i < KYBER_N / 4; i++) {
226          // START own code
227          if (i==0){
228              simpleserial_put("r", 2, &a->coeffs[4 * i]);
229              simpleserial_put("r", 2, &b->coeffs[4 * i]);
230          }
231          // END own code
232          PQCLEAN_KYBER512_CLEAN_basemul(&r->coeffs[4 * i], &a->coeffs[4 * i], &b->coeffs[4 * i], PQCLEAN_KYBER512_CLEAN_zetas[64 + i]);
233          PQCLEAN_KYBER512_CLEAN_basemul(&r->coeffs[4 * i + 2], &a->coeffs[4 * i + 2], &b->coeffs[4 * i + 2], -PQCLEAN_KYBER512_CLEAN_zetas[64 + i]);
234      }
235  }
```

Figure 3.16: SimpleSerial functions in basemul_montgomery

When reading the outputs of `simpleserial_put`, a function similar to how the traces are captured, needs to be applied in Jupyter Notebook. Figure 3.17 illustrates the use of the counter in order to get the first ciphertext and secret key bytes from the respective ciphertext decapsulation. For each output (decapsulation of a ciphertext), the values of the secret key bytes remain static, further verifying the correctness of the function.

```
hypbytes=[]
for i in range(5):

    if i < 10:
        char = "00"+str(i)+'\n'
    elif 9 < i < 100:
        char = "0"+str(i)+'\n'
    else:
        char = str(i)+'\n'

    reset_target(scope)
    scope.arm()
    target.flush()
    target.write(char)
    time.sleep(0.2)
    output=target.read()
    hypbytes.append(output)
    print(str(i),  ": \n",output)
```

```
0 :
 r750A420A
rD4FED4FA
```

Figure 3.17: Function to intercept the intermediate ciphertext and secret key bytes

3.3.2 Correlation Phase

Given the attacker has successfully extracted the correctly unpacked and transformed ciphertext bytes, phase 3 i.e. the correlation phase can begin. For this phase, the steps from [KdG21] can, more or less, be used.
Firstly, the built-in Pearson correlation coefficient function from numpy (`numpy.corrcoef(x,y)`), which takes in two arrays of equal size, is fairly slow. Therefore a simplified version of the calculation is defined, which can be seen in Figure 3.18. Tests have been executed to verify the correctness of this function. The only "limitation" of using this function is that the arrays it takes in must be a "numpy-array". This is due to how it uses `np.sum(x)`, which requires the use of a numpy-array. The "problem" is easily solved by converting regular python arrays to numpy arrays, using the function `array = numpy.array(array)`.

```
def fast_pearson(x, y):
    n = len(x)
    sum_x = np.sum(x)
    sum_y = np.sum(y)
    sum_x_sq = np.sum(x ** 2)
    sum_y_sq = np.sum(y ** 2)
    psum = np.sum(x * y)
    num = psum - (sum_x * sum_y / n)
    den = np.sqrt((sum_x_sq - sum_x ** 2 / n) * (sum_y_sq - sum_y ** 2 / n))
    if den == 0: return 0
    return num / den
```

Figure 3.18: The fast_pearsson function

```
from tqdm import trange
import numpy as np

testarr=[[]for i in range(2**16)]
pcck2k3=[]

threshhold=0.9
for i in trange(2**16): #Iterating over all possible values for k2k3
    rst=[]
    pccmax=[]

    for k in range(num_traces):
        rst.append(bin(i*cipherbyte[k].count("1")) # Hamming weight of each guess multiplied by the ith ciphertext
    rst=np.array(rst)

    for tnum in range(scope.adc.samples):
        pcc = (fast_pearson(ti[tnum], rst)) #PCC of the transposed traces and rst
        if pcc > threshhold: # For debugging when testing
            print("KeyGuess "+str(i) + " bigger than " +str(threshhold)+", "+"ti[tnum] "+str(tnum)+". "+"Value: "+str(p
        pccmax.append(abs(pcc))
        testarr[i].append(pcc) # To plot correlation of each guess

    pcck2k3.append(max(pccmax))

print(max(pcck2k3))
print(np.mean(pcck2k3))
```

Figure 3.19: Full script of the attack

Figure 3.19 gives a suggested method to execute steps 1-3 from [KdG21]. In the first loop, i is iterated over all possible key guesses. A threshold is set for testing. An *rst*-array is generated for each guess, giving a total of 65536 *rst*-arrays. With 1000 samples for each trace, there is a total of 1000 transposed traces. In other words, the PCC is calculated 1000 times for each guess, adding up to a total of $65536 * 1000 = 65536000$ calculations. Running a single attack on a two-byte value takes about 12 minutes.

When steps 1-3 from [KdG21] has been executed, there should remain a sample of possible k_2k_3 guesses. The continuation of the attack is to fix each k_2k_3 from the sample and repeat steps 1-3, using a function which combines the secret key values, with a guess on k_0k_1. In other words, for n possible values of k_2k_3, the remaining steps need to be executed n times. With an execution time of approximately 12 minutes, this gives a total execution time of n*12 minutes.

To combine the k_2k_3 and k_0k_1, the attack from [KdG21] uses the `pkhtb`-function. This is an Assembly function which combines a halfword from one register with a halfword from another register [Deva]. The clean implementation does not use neither the `smultt` operation, nor the `pkhtb` operation. Therefore, the attack on the clean implementation focuses on the first operation (line 177) in the `basemul` as the `smultt` operation in steps 1-3 and the third (line 179) operation as the `pkhtb` operation.

3.3.3 Real World Scenario

In a real world scenario, the situation will be quite different for this phase. There would be no SimpleSerial and trigger functionality. The scenario could be e.g. an IOT-device which has a built-in Kyber512 cryptosystem. To initiate communication with a client, the shared keys must be exchanged as used a symmetric key encryption. If the IOT-device has a static secret key, which is rarely refreshed, the ground terms for the attack is accepted. The attacker must execute an invasive side channel attack, in order to measure the power consumption of the device during decapsulation.

Furthermore, the attacker needs to have a tool for capturing the ciphertexts, sent over e.g. the MQTT-protocol. Given that the attacker is able to measure the power traces and "sniff" out the ciphertexts, there are still several problems. Firstly, the attacker needs to know exactly what implementation and security level of Kyber the IOT-device is using. The `basemul`-functions will work mostly similarly in the different implementations, however, when capturing a power trace of the entire decapsulation, knowing exactly where in the trace to observe the `basemul`-operations will be helpful and reduce the time of the attack significantly. Secondly, it is crucial to pack the secret key correctly, based on the values of k, which vary, depending on the version of Kyber.

Chapter 4
Results and Analysis

The study in this thesis has focused on exploring side-channel attacks on the clean implementation of Kyber512 from the pqm4 library. Chapter 3 has explained how to set up Kyber on the ChipWhisperer and how to use the information from [KdG21] to suggest an attack on the clean implementation on Kyber. This chapter will explain the results observed compared to the anticipated results, the limitations of the attack and ultimately the limitations of the difficulties and limitations of using ChipWhisperer Lite in combination with Kyber.

4.1 Results

The attack on the clean implementation of Kyber yielded interesting results. The goal of the attack would be to get a small sample of numbers with significant max correlation, i.e. a "spike" with high correlation on a small range of samples, and generally low correlation elsewhere. It would be anticipated that wrong guesses would have generally low correlation and no "spikes".

The continuation of the attack would proceed to take the sample of bytes, e.g. 10 probable bytes, fix each one of them and continue the attack. The final part of the attack would then take approximately 10*12 minutes to extract the first four bytes ($k_0k_1k_2k_3$) of the secret key. Again, due to how the bytes in this intermediate operation is unpacked, as can be seen in Figure 3.7, it would be reasonable to assume that all the bytes of a polynomial or both polynomials of the vector must be extracted, in order to re-pack the key into its original form.

4.1.1 Observed results

Total PCC-values

The attack in Figure 3.19 does yield interesting results, yet it is not able to extract a small enough sample to continue the attack. A plot of all the values in the $pcck_2k_3$-

array can be seen in Figure 4.1. In the plot, the y-axis is the max, in absolute, PCC-value and the x-axis is each key guess in decimal.

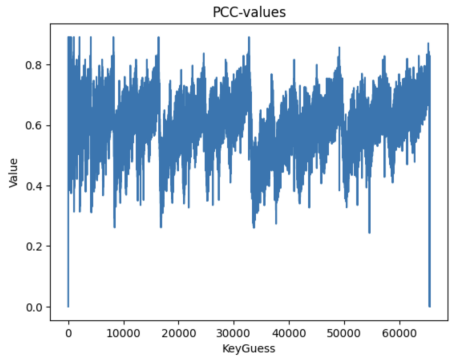


Figure 4.1: Plot of each max PCC-value for each key guess

There are several interesting observations with this plot. Firstly saw-tooth pattern can be observed. For key guesses below 100, the correlations are fairly high, indicating a good guess. The highest spikes are when the guesses are of powers of two, which can be most easily seen at key guess 8192. After this guess, the PCC-value drops substantially, and initiates the saw-tooth pattern. The pattern does very much indicate that there is a repeating pattern, with a phase of approximately 16384, or 2^14. The reasoning behind this is currently unknown for this thesis. However there are several hypotheses.

Firstly, the unpacking of the secret key may be a reason for the pattern. When reproducing the unpacking-function and applying it to the used secret key, the values of all the coefficients for both polynomials are below the value 3329. 3329 is the value of the modulus used for coefficients, so it does certainly make sense for the values to be below the modulus value. Furthermore, the values used as inputs in the `basemul`-functions are all "int_16"-values, stating they are signed 16-bit integers. For a signed 16-bit integers, the range of values span from -32768 to 32767. Both of these assumptions give an indication of why the saw-tooth pattern spikes at certain values and plummets right after. However, this would rather imply that the saw-tooth pattern would either have a phase of 3329 or 32768, it does not explain why the phase is of value 16384.

The values of the negative numbers of the signed 16-bit values is also worth a discussion. In the attack, i is incremented from 0 to $2^{16} - 1$, i.e. 0 to 65535. i is then multiplied with the intermediate ciphertext bytes, converted to binary and counting the amount of "1"s in the binary string. The multiplication does yield a 32-bit signed integer, which is also the data type of the value used within the `fqmul`-operation.

However, the Hamming weight of a signed 16-bit integer does span over the exact same range as the Hamming weight of an unsigned 16-bit integer. It does not make a difference on the Hamming weight of the calculated product, whether the values of the factors are negative or positive.

PCC-values of single guesses

Whereas Figure 4.1 is a plot of all the max-PCC values for each guess in absolute, it is also interesting to observe the PCC-values for a single guess. Assuming Little-Endian notation the first values going into the first `basemul`-operation would be the strings: "FAD4FED4" for the ciphertext and "0A420A75" for the secret key. The third operation in the `basemul`-function, is `r[0] = fqmul(a[0], b[0])`, stating that the two last bytes of each string is to be multiplied and Montgomery reduced. In other words, somewhere in the power traces there should be a multiplication of the values "FAD4" and "0A42" (2626_{10}). The fifth line in the `basemul`-function uses the same value for `b[0]`, but uses the final two bytes of the secret key, `a[1]` $= 0A75 = 2677_{10}$. This would imply that if we are using the first two bytes of each unpacked ciphertext as input values in the attack, we should observe significant values for the guesses "2626" and "2677".

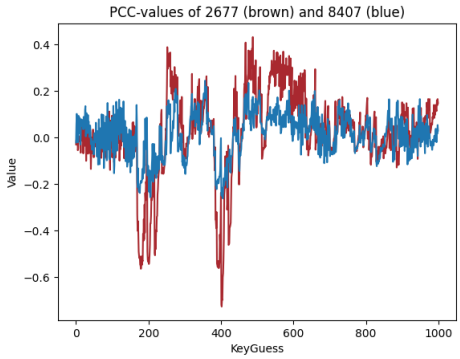


Figure 4.2: PCC-values for one correct and one wrong guess with low correlation

Figure 4.2 show the PCC-values of two guesses. The brown plot is the correlation when the key guess has the value 2677. The blue plot is a wrong guess and has the value 8407. It can be observed that the wrong guess has minor spikes around the same samples as the correct guess. Even though the PCC-values of the wrong key guess follows a similar, yet minor, pattern as the correct guess, the absolute value of the max PCC-value of the wrong guess is below 0.3, indicating it is a wrong guess.

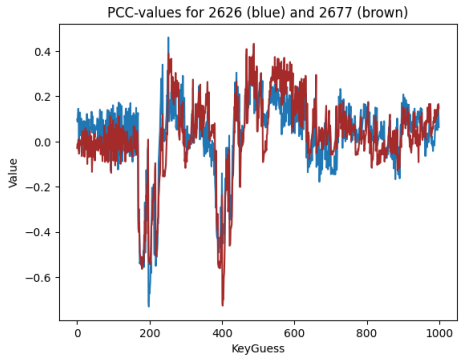


Figure 4.3: PCC-values for two correct guesses

Figure 4.3 is the plot of the PCC-values of the two correct guesses. The hypothesis does match with the observed values. In the blue plot, at sample 200, the correlation of the guess 2626 ($0A42_{16}$) has its maximum (minimum, rather yet the absolute of the value is what is desired), implying the operation `a[0] * b[0]` or $FAD4_{16} * 0A42_{16} = 64212_{10} * 2626_{10}$ is observed at this sample. This is further emphasized with the brown plot. The brown plot is the value of `a[1] * b[0]` or $FAD4_{16} * 0A75_{16} = 64212_{10} * 2677_{10}$. It can be clearly observed that the correct guess in the blue plot appears before the the correct guess in the brown, which matches with the code.

The problem, however, is the large amount of "false positives" which appear and even have higher correlation than the correct guesses. Figure 4.4 show how even though 2626 is the correct guess, the PCC values of a wrong guess, 2048, has higher correlation in multiple occurrences.

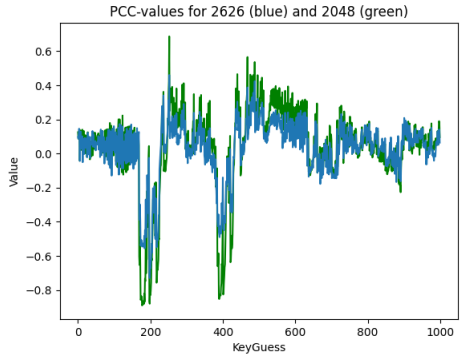


Figure 4.4: PCC-values for one correct (2626) and one wrong (2048) guess

4.2 Analysis of the results and potential pitfalls

4.2.1 Powers of 2

Further analysis states that key guesses of a power of two has the highest correlation. Going back to the original attack, the array *rst* is simply the Hamming weight of the product of the guess and the intermediate ciphertext byte. If the key guess is a power of 2, the Hamming weight will remain the same as the Hamming weight of the ciphertext byte. E.g. the value $FAD4_{16} = 64212_{10}$ is represented in binary as 1111101011010100_2, which has a hamming weight of 10. The product $64212_{10} * 2626_{10} = 131506176_{10}$ is represented in binary as $11111010110101000000000000_2$ which also has a Hamming weight of 10. On the other hand, the product $FAD4_{16} * 0A42_{16} = 64212_{10} * 2626_{10} = 168620712_{10} = 10100000110011110010101010000_2$ which is the correct guess multiplied by the intercepted ciphertext bytes has a Hamming weight of 12. In other words, it would appear as that values which have a Hamming weight equal to the ciphertext bytes have a higher correlation than values equal to the product of the key guess and the ciphertext bytes. Conclusively, it seems the attack does not find the product of the ciphertext and secret key, but rather the ciphertext itself.

Verifying this is rather difficult. Each *rst* contains 200 values for each ciphertext. Some key guesses will give the same Hamming weight as the ciphertext for some of the ciphertexts, but not necessarily for all. However, whenever the guess is a power of 2, every single value in the *rst* array has a Hamming weight equal to the Hamming weight of the ciphertext. It must be emphasized that this is only an assumption to the reasoning behind the high correlation of key guesses of powers of 2.

Ultimately, the attack generates far too many "false positives" in order to proceed with finding the correct byte values. With 65536 key guesses and a threshold of 0.75, there are still over 500 potential candidates. An idea could be to remove all the candidates if they are a power of 2. This would simply remove a maximum of 16 values. Furthermore, it would also remove candidates for the key which, in a real world attack, could as well have been correct guesses. The problem lies in that there exists a large amount of values between 0 and 655355 which yield a Hamming weight equal to the Hamming weight of the ciphertext.

4.2.2 Differences in implementations

The attack has been executed on the clean implementation of Kyber512 from the pqm4-library. How the code is executed, down to a bit-wise level, on the target might be problematic. Whereas a single MUL operation on the Arm Cortex M4 processor takes one cycle to execute [Devb], other factors can cause the operation to require more cycles. The technical reference manual states that more complex

operations such as a signed or unsigned Multiply-Accumulate (MLA, MLS) take 2 cycles [Devb]. This can cause the multiplication operation under attack in this this thesis to be executed over multiple samples for a power trace, indicating a wrong result. The attack from [KdG21] was executed on the `SMULTT` operation, which only uses one clock cycle. The multiplications on the 16-bit signed values in the clean implementation should be possible to execute in a `SMULL` operation, using 1 cycle, yet there may be different factors, e.g. compiler functionality, which may cause problems.

4.2.3 Noise in traces and trigger positions

The triggers have been strategically placed to minimize the amount of possible noise in the trace. However, there is always a possibility of noise appearing in the capture board, which may cause problems for the attack. If this were to be the case, there would not have been such a strong correlation on certain guesses and simply a slightly lesser correlation on other guesses. If all the traces contained vast amounts of noise, it would be safe to assume there would be no coherent correlations. However, slight amounts of noise can cause issues when executing a CPA-attack.

The trigger placements, themselves, may also be problematic. In this attack, the triggers are placed around a `basemul`-function measuring the power consumption of that execution. In the attack of this thesis, the main focus is on an even deeper level than where the triggers are positioned, i.e. the multiplication in the `fqmul`-operation. An idea would therefore be to apply the triggers around this multiplication. However, this function is being continuously called throughout the entire decapsulation, meaning when the `fqmul`-operation is called from the `basemul`-function, the `fqmul`-function has already been executed a vast amount of times, called from other functions. This would rise a need for a much larger sample size, increasing the execution time of the attack severely.

Chapter 5
Discussion

The research field of side-channel attacks on a cryptographic implementation is a deep-dive into a world of micro-processors, c-coding, electrical physics, advanced mathematics and a general ability to combine all these aspects. When the cryptographic algorithm is an un-standardized algorithm, currently undergoing testing and a standardization, the field is quite narrow. An SCA, itself, is an old concept; seek leaking information from a system and use that information to exploit the system. Applying the concepts of an SCA to a new cryptographic algorithm in order to seek its vulnerabilities may have several outcomes. Posed attacks may be successful and vulnerable parts of the algorithm can be proven. On the other hand, researcher may try continuously to attack different parts of the algorithm, and conclude that it is difficult to extract any hidden information from said parts. This thesis has landed somewhere in between. The `basemul`-operation may certainly be a vulnerable part of the algorithm. Interesting results are found, yet the results have not been good enough to retrieve the secret key. This chapter will be a discussion and self-evaluation of the experiment and methodology.

5.1 Difficulties with applying Kyber

A large part of the thesis has been managing to apply Kyber to the ChipWhisperer. Although seemingly easy, this task requires experience with makefiles and flashing. An ignorant assumption was that this was as easy as in TTM4502 [Grü22], where the makefiles were already written and the demos guide the user seamlessly along multiple SCAs. This section explains a handful of difficulties researchers may encounter when attempting the same experiment.

5.1.1 pqm4

As explained in detail in Chapter 3, the process of applying Kyber is not onto the ChipWhisperer is not straightforward. The pqm4 library contains a huge amount of files, some being exclusive to each implementation of Kyber, and other files in

the library, being global files, used in most implementations, e.g. the *randombytes* files. This causes a need for understanding the hierarchy of the pqm4 library, as well as understanding what files are relevant to the exact implementation. Furthermore, the pqm4-library is mainly used for testing. The integrated files used in the library uses standardized functions to call the different algorithms. It is therefore essential to understand how to extract the correct calls for the desired functions in order to properly execute them in your own code.

5.1.2 Makefiles and compiling

Makefiles may seem fairly simple, yet they can cause a lot of distress. It is highly recommended to have prior experience with makefiles when going into this project. A makefile will compile several files defined in the makefile. Different ways to compile the files can also be applied by adding different "flags" in the makefile. When starting with minuscule experience with makefiles, compiling the entire project into a hex-file for it to be flashed onto the micro-controller turned out to be bigger of a problem than first anticipated.

In the tutorials from ChipWhisperer, the makefiles were already made, and simply had to be called by executing a command, also already made. Again, when going into a project of this scale, it would be highly recommended to understand the logic and interactions of a makefile, in order to have a better understanding of how to compile the project.

Whereas compiling the entire project is in fact a problem in itself, so is the sheer size of the project. When compiling the 200 ciphertexts, the Kyber512 algorithm, and the storage for the 200 shared secrets, needed for debugging, the total size of the hex file is approximately 165kB. Even though this does not seem like a lot, when flashing it onto the target, this can take up to 4 minutes. When making small changes to the c-code trying with a large amount of test, the 4 minutes needed to flash makes testing quite time consuming.

5.1.3 Kyber is not an integrated part of ChipWhipserer

ChipWhisperer has great tutorials for the projects they have implemented. The entire idea of the ChipWhisperer is to research on effects of side-channel attacks. It is assumed the user of a ChipWhisperer has a general understanding of the algorithms used. However experience with and understanding of the algorithm are not in any way needed to execute the attack. ChipWhisperer has made it easy enough to execute an advanced attack on e.g. AES, that only a few commands and a minor time investment is needed. It is highly recommended to understand the underlying theory of the ChipWhisperer and the guides they provide before using the same concepts and ideas on another algorithm.

For this thesis, a general understanding of both ChipWhisperer and side-channel attacks in general was researched on in the pre-project from TTM4502 [Grü22]. However, the pre-project merely scratched the surface of the the underlying integrated functions in the ChipWhisperer library. As described in Chapter 3 the ChipWhisperer-library allows for an integrated function when captures traces, allowing for both sending of an encryption key, as well as plaintext while capturing the trace. For encryption schemes using small keys and plaintexts this works fairly fine. However, with the 1632 byte secret key and the 768 byte long ciphertexts, the buffer size of the SimpleSerial protocol is not sufficient for the integrated commands in ChipWhisperer. There are other ways of solving this issue than the use of the counter explained in Chapter 3, yet the problem remains the same: how can the attacker capture the traces of single decapsulations for different ciphertexts? Even if NewAE-Technology, the producers of the ChipWhisperer, were to make guides on Kyber, the buffer size for small versions of ChipWhisperer, e.g. the ChipWhisperer Lite, will still have the small buffer.

ChipWhisperer has somewhat addressed this issue with the attacks on RSA. For their own implementations of attacking RSA, they consider an RSA key of a far smaller key size than in standardized RSA [Inc18]. In hindsight, the same idea could be applied in this thesis. Instead of focusing on a full-scale implementation of Kyber, the attack could be on a self-made implementation of "Baby-Kyber", e.g. equal to [Gon21]. This could also give interesting results, yet it was not the intention of this thesis.

5.1.4 Endianess and SimpleSerial

Even though SimpleSerial has been a key component of this thesis it has also caused some confusion. The outputs from 3.17 are in one order, while the values used in in the attack is in reversed order. This is due to how the bytes are being reversed in certain situations when using SimpleSerial. This has, naturally, caused confusion. Firstly, when addressing the issue of whether the decapsulation is done correctly, SimpleSerial was used to send the shared secrets for each decapsulation. Verifying the answers, showed that the decapsulation was in fact done correctly. The shared secrets matched with the anticipated results.

However, further testing indicated that this is not always the case. The bytes in 3.17 are in fact in reversed order, or Little-Endian represented. The reason for this is not entirely known. One hypothesis could be that SimpleSerial treats different data types differently. The shared keys are stored as the data type "unsigned char". This is because this is what was used in the original implementation from pqm4. One would therefore reasonably assume that data sent as the data type "int16_t" also would be in correct order. A lot of testing was therefore done with values in reversed order. Is

this, however, a problem when the *rst*-array is an array of Hamming weights? Yes, e.g. the ciphertext "D4FA" multiplied by a key guess will, in most cases, yield a different Hamming Weight, than the ciphertext "FAD4" multiplied with the same guess. Verifying that the outputs of Figure 3.17 was done by defining two known values with a known product and sending the values over SimpleSerial. The exact reason to why it is reversed is not known. It could be parameters defined in the c-code or it could be how SimpleSerial interprets different types. Nevertheless, for an attacker, it is important to have this in knowledge when sending information via SimpleSerial.

5.2 Areas of improvement and future research

The focus are for this thesis has been the clean implementation of Kyber from the pqm4 library. More specifically, the attack has been on coefficient multiplication during decapsulation. For this thesis, there are several areas which need more understanding to be able to completely analyze the attack. This section poses to iterate through the weakest points of the attack and discuss how they can be improved.

5.2.1 Intermediate values

Finding ciphertext bytes

During the research for this thesis, a challenge has been to calculate the *rst*-array. As it should simply be the Hamming weight of the multiplication of two 16-bit integers, finding the value of the first factor, i.e. the ciphertext bytes has not been straightforward.

In this thesis, the final solution was using SimpleSerial to extract the values of the bytes. In a real-world scenario, this would, in most cases, not be practically possible. If an attacker would have the ability to read intermediate values directly from the source code, the better approach would simply be to read the secret key bytes. Therefore, for an attacker to have knowledge of the intermediate ciphertext bytes, both the `unpack_ciphertext`-function and the `polyvec_ntt` function must be recreated. `unpack_ciphertext` is mostly straightforward, as it uses one decompress function for the vector, and one for the polynomial. Knowing simply how these functions operate, which is public information, would allow to unpack the ciphertext.

Then, the ciphertext is NTT-transformed, using the `polyvec_ntt`-function. For the NTT to take place, a set of "zeta"-values need to be generated. Generating the matrix of zetas is not trivial, however for each version of Kyber, the zetas remain the same. They are static and can therefore be hard-coded into the implementation in order to save execution time. However, from the attacker's point of view, it is

essential to know exactly when the zeta values are being used. E.g. in the clean implementation used in this thesis, all coding is done in in c. The zeta values used in the decapsulation are applied before a "Barret reduction". In the "m4"-implementation, used in [KdG21], the NTT-transformation is done at a different stage in the decapsulation. The zeta-values, by themselves, should remain static for each implementation, yet when processed at different stages of the algorithm, the values of the zetas may be different and/or rearranged. This emphasizes even further the importance of understanding the exact decapsulation algorithm in order to execute the attack.

Optimizing the range of guesses

In the attack posed in this thesis, the number of guesses range from 0-65535. One question which may arise from this is whether that span is necessary. When the secret key is unpacked, certain operations are performed to generate a vector of polynomials from the 1632 sized byte string. As previously stated, the coefficients for each polynomial are reduced modulo 3329. Is there therefore a possibility that if the attack, in reality, poses to find the coefficients of the polynomials, would the range of guesses go from 0-3329? When attempting to re-create the unpacking of the key, all the values remained under 3329. However, in order to verify the correctness of the unpacked key, it would have to be re-packed and matched with the original key. This thesis did not enter that area, but should be researched on further, as it could potentially reduce the execution time of the attack significantly.

The modulus of the coefficients also raise the question: are the intercepted ciphertext bytes intercepted correctly? The intercepted hex value "FAD4" has the decimal value of 64212. This value is far beyond the the modulus of 3329. In other words, seemingly one of the factors used to calculate the *rst* array in the attack may have been interpreted wrongly.

5.2.2 A different attack

The attack posed in this thesis has been a vertical CPA attack. Although it being a strong attack, the "horizontal" CPA attack could be considered. This attack has not been within the scope of this thesis, yet it could generate interesting results. Whereas a vertical CPA attack focus on single executions, a horizontal CPA attack focuses on operations in which the secret key is used in multiple occasions [CFG+10].

Completely different attacks, such as the one posed in [Guo23] seems to yield significant results. This attack presents advanced techniques, modeling the traces as low-density parity-check code and extracting the secret key using an average of 12 traces.

Chapter 6
Conclusion

This thesis has researched the effects of a CPA attack on the clean implementation of CRYSTALS Kyber from the pqm4 library. The ChipWhisperer tool has been used to gather power traces from the algorithm during the decapsulation phase. Ultimately, the attack did not succeed the way it was initially intended to. However, interesting results have been documented and can provide a relevant foundation for further research.

In the context of the attack posed in this thesis, the recommendation when implementing Kyber for security reasons is to frequently refresh the secret key.

Chapter 2 explained the definition of Kyber, both mathematically and algorithmic, the theory of power traces and how they can be utilized to execute a CPA. This theory has been the basis of Chapter 3.

The thesis has explained, in detail, how the Kyber algorithm can be applied on the ChipWhisperer tool, hopefully lowering the entry barrier for researchers wanting to contribute to the field. Even though the ChipWhisperer tool makes attacks on certain algorithms trivial, it is highly recommended for researchers to possess a deep understanding of c-code, the desired algorithm under attack and the ChipWhisperer library.

The research questions were:

RQ1: *How can ChipWhisperer be used to measure the power consumption of Kyber?*

This was partly explained in TTM4502 [Grü22], however, using an algorithm which is not an integrated part of the ChipWhisperer library complicated this research question. Ultimately, this was done successfully, as described in Chapter 3.

RQ2: *How can we use the information from 1) to construct and execute a side-channel attack on Kyber?*

A side-channel attack on Kyber was constructed and executed on Kyber. The intended results were unfortunately not achieved, yet interesting results were found.

It is difficult to conclude why the attack did not succeed. Chapter 5 introduced possible suggestions to why the attack failed, which may be foundation for future work. On the other hand, there is a possibility that the clean implementation of Kyber512 requires a different attack.

References

[ABCS06] R. Anderson, M. Bond, *et al.*, «Cryptographic processors-a survey», *Proceedings of the IEEE*, vol. 94, pp. 357–369, Mar. 2006.

[ABD+19] R. Avanzi, J. Bos, *et al.*, «Crystals-kyber algorithm specifications and supporting documentation», *NIST PQC Round*, vol. 2, no. 4, pp. 1–43, Jan. 2019.

[CFG+10] C. Clavier, B. Feix, *et al.*, «Horizontal correlation analysis on exponentiation», IACR Cryptology ePrint Archive, Tech. Rep. 2010/394, 2010, ffhal-02486982f. [Online]. Available: https://eprint.iacr.org/2010/394.

[Dac22] Q. Dace Krauthamer. «Q-day: The problem with legacy public key encryption». (Jul. 2022), [Online]. Available: https://www.helpnetsecurity.com/2022/07/15/legacy-public-key-encryption-problem/.

[Deva] A. Developer, «Arm compiler armasm reference guide version 6.00», Tech. Rep. [Online]. Available: https://developer.arm.com/documentation/dui0802/a/A32-and-T32-Instructions/PKHBT-and-PKHTB.

[Devb] A. Developer, «Arm cortex-m4 processor technical reference manual revision r0p0», Tech. Rep. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions.

[Devc] A. Developer, «Gnu arm embedded toolchain downloads», Tech. Rep. [Online]. Available: https://developer.arm.com/downloads/-/gnu-rm.

[DH76] W. Diffie and M. E. Hellman, «New directions in cryptography», *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[Dur23] J. Duran. «Harvest now, decrypt later? the truth behind this quantum theory». Last modified: February 07, 2023. (Feb. 2023).

[Gam22] J. Gambetta. «Quantum-centric supercomputing: The next wave of computing». (2022), [Online]. Available: https://research.ibm.com/blog/next-wave-quantum-centric-supercomputing.

[Gia23] K. Gian. «Why don't we have quantum computers yet?» (Apr. 2023), [Online]. Available: https://tomrocksmaths.com/2023/04/19/why-dont-we-have-quantum-computers-already/.

[Gon21] R. Gonzales. «Kyber - how does it work?» (2021).

[Grü22] M. Grünfeld, «Side-channel attacks on cryptographic components», Department of Information Security, Communication Technology, NTNU – Norwegian University of Science, and Technology, Project report in TTM4502, Dec. 2022.

[Guo23] Guo, Qian and Nabokov, Denis and Nilsson, Alexander and Johansson, Thomas, *SCA-LDPC: A Code-Based Framework for Key-Recovery Side-Channel Attacks on Post-Quantum Encryption Schemes*, eng, Preprint, Mar. 2023. [Online]. Available: %7Bhttps://eprint.iacr.org/2023/294%7D.

[HMP20] E. Hazan, A. Menard, and M. Patel, «The next tech-revolution: Quantum computing», Mar. 2020.

[Inc18] N. T. Inc. «V3:tutorial b11 breaking rsa». (2018), [Online]. Available: https://wiki.newae.com/V3:Tutorial_B11_Breaking_RSA.

[Inc22] N. T. Inc. «Chipwhisperer-lite». (2022), [Online]. Available: https://rtfm.newae.com/Capture/ChipWhisperer-Lite/.

[KdG21] A. Karlov and N. L. de Guertechin, *Power analysis attack on kyber*, Cryptology ePrint Archive, Paper 2021/1311, https://eprint.iacr.org/2021/1311, Sep. 2021. [Online]. Available: https://eprint.iacr.org/2021/1311.

[KPR+] M. J. Kannwischer, R. Petri, *et al.*, *PQM4: Post-quantum crypto library for the ARM Cortex-M4*, https://github.com/mupq/pqm4.

[Len11] A. K. Lenstra, «Integer factoring», in *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg and S. Jajodia, Eds. Boston, MA: Springer US, 2011, pp. 611–618. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5_455.

[Laa15] T. Laarhoven, *Lattice cryptography and lattice cryptanalysis*, Lecture for Cryptography I, Technische Universiteit Eindhoven, Aug. 2015. [Online]. Available: https://thijs.com/docs/lec1.pdf.

[MN19] A. Matuschak and M. A. Nielsen. «Quantum computing for the very curious». Lat updated: March 18, 2019. (2019), [Online]. Available: https://quantum.country/qcvc.

[Nan20] G. Nannicini, «An introduction to quantum computing, without the physics», *SIAM Review*, vol. 62, no. 4, pp. 936–981, 2020. [Online]. Available: https://doi.org/10.1137/18M1170650.

[oST02] N. I. of Standards and Technology, «Security requirements for cryptographic modules», U.S. Department of Commerce, Washington, D.C., Tech. Rep., Feb. 2002.

[oST17] N. I. of Standards and Technology, «Nist sp 800-63-3 digital identity guidelines», U.S. Department of Commerce, Washington, D.C., Tech. Rep. NIST Special Publication (NIST SP) 800-63-3, Inlcudes updates as of 03-02-2020, Feb. 2017.

[oST19] N. I. of Standards and Technology, «Security requirements for cryptographic modules», U.S. Department of Commerce, Washington, D.C., Tech. Rep., Mar. 2019.

[oSta19] N. I. of Standards Technology. «Announcing approval and issuance of fips 140-3, security requirements for cryptographic modules». (May 2019), [Online]. Available: https://www.nist.gov/news-events/news/2019/05/announcing-approval-and-issuance-fips-140-3-security-requirements.

[oSta23] N. I. of Standards Technology. «Post-quantum cryptography». Created January 03, 2017, Updated June 14, 2023. (2023), [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography.

[Owe17] W. J. B. D. C. Owen Lo, *Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa)*, Journal of Cyber Security Technology, 1:2, 88-107, 2017. [Online]. Available: https://doi.org/10.1080/23742917.2016.1231523.

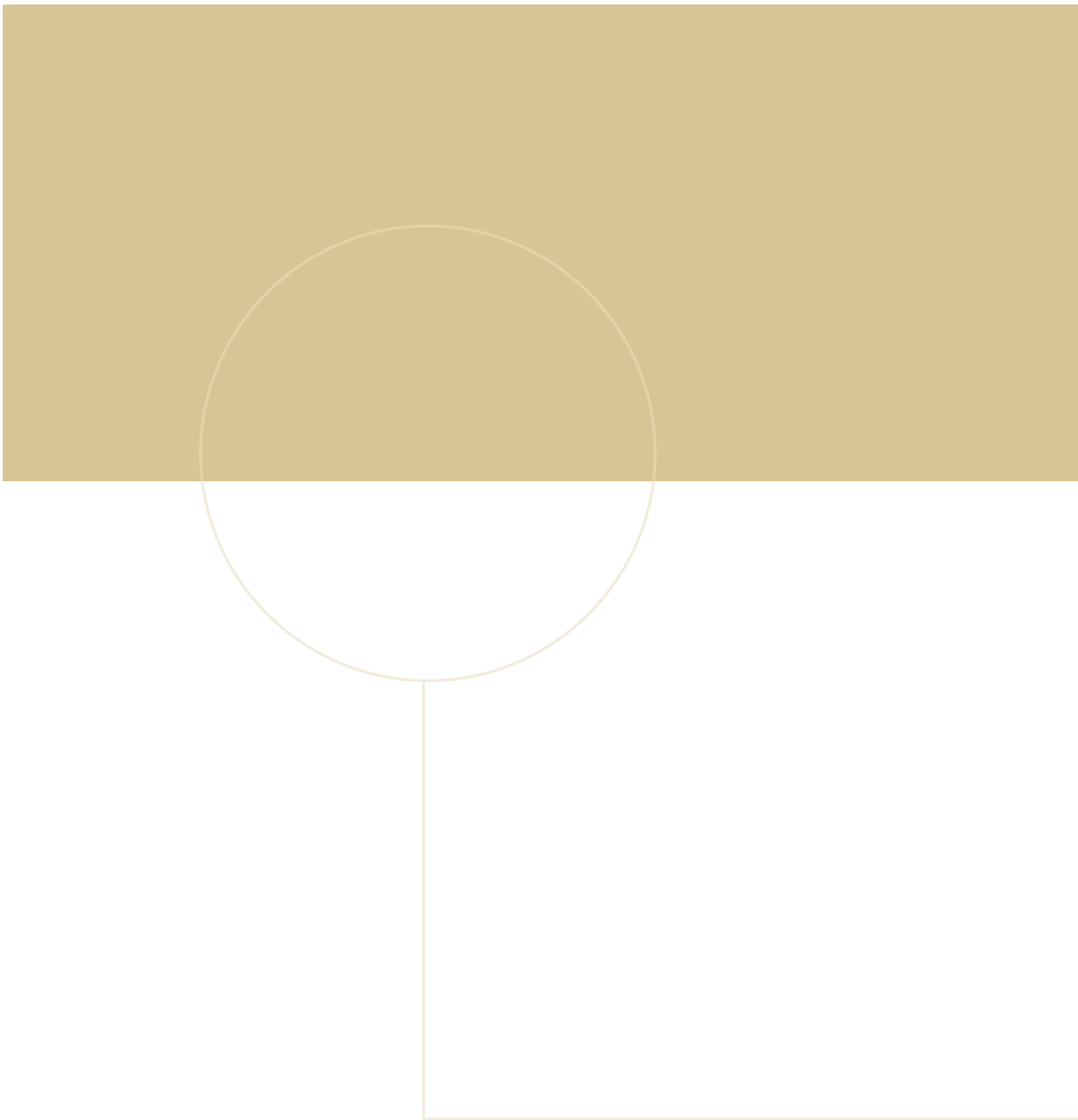
[Pop] G. Pope. «Crystals-kyber python implementation». (), [Online]. Available: https://github.com/GiacomoPope/kyber-py.

[Qua21] I. Quantum. (2021), [Online]. Available: https://quantum-computing.ibm.com/composer/docs/iqx/guide/shors-algorithm.

[Reg06] O. Regev, «Lattice-based cryptography», in *Advances in Cryptology - CRYPTO 2006*, C. Dwork, Ed., ser. Lecture Notes in Computer Science, vol. 4117, Berlin, Heidelberg: Springer, Aug. 2006.

[Wri87] P. Wright, Spycatcher: The Candid Autobiography of a Senior Intelligence Officer, 1987.

[ZF05] Y. Zhou and D. Feng, *Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing*, zyb@is.iscas.ac.cn 13083 received 27 Oct 2005, Oct. 2005. [Online]. Available: http://eprint.iacr.org/2005/388.



NTNU

Norwegian University of
Science and Technology