A Plaintext-checking Oracle for Decapsulation Leakage in the NIST HQC FPGA Reference Implementation

Benjamin Halasz
benjamin_halasz@icloud.com

Department of Electrical and Information Technology
Lund University

Supervisor: Qian Guo

Examiner: Thomas Johansson

December 15, 2025

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

In this thesis, we evaluate the side-channel security of the NIST HQC post-quantum cryptosystem by analysing the official FPGA decapsulation implementation. A hardware wrapper was created to interact with the HQC decapsulation modules, enabling ciphertext decapsulation with precise power trace capturing using a ChipWhisperer CW305 and Husky setup. A dataset with 20k power traces was generated from decapsulating two different ciphertext groups, having $m = 0$ and $m = rand()$. Statistical leakage assessment revealed significant leakage about the message in both the SHAKE pseudorandom number generator and the decoder used in the HQC decoding stage.

A few machine-learning models were trained and evaluated on their distinguishing capabilities of determining the group of the message decapsulated from a single power trace. The best-performing model, a small convolutional neural network, achieved a validation accuracy of 97.2%. Using this oracle accuracy, a key-recovery simulation was performed, which showed that approximately 736 decapsulation traces were enough to recover the secret-key with a probability of 72.3%.

These results demonstrated that the 2021 NIST HQC FPGA implementation contains exploitable side-channel leakage and that both the SHAKE part and the decoding part require stronger countermeasures.

Keywords

Side-channel attack, Public key encryption, HQC, Hardware, FPGA, Deep learning

Acknowledgements

During my master's thesis there have been a few people supporting me. Firstly I would like to thank my supervisor Qian Guo for guiding me through the project by answering difficult questions that have come up throughout the work. I would also like to thank my parents, friends, and roommates for taking supporting roles in driving me forward

Popular Science Summary

As the digital world expands, so does the need for secure communication. Today everything from private messages to online banking uses encryption to protect data from getting into the wrong hands. But there is a problem. In the future it may become possible to break today's secure cryptosystems using quantum computers. Therefore it is of importance that new cryptosystems are developed that can withstand quantum computing. One of these algorithms is called HQC, Hamming Quasi-Cyclic, which has been selected as one of the standard cryptosystems used by the U.S National Institute of Standards and Technology, NIST.

Even if HQC is a mathematically secure cryptosystem, the physical implementation running on a chip or CPU can leak information from different sources such as power consumption. This happens when the chip in question performs different operations creating small variations in, for example power consumption. These types of attacks using power consumption or electromagnetic radiation are called side-channel attacks. By measuring the power consumption during decryption, an attacker can sometimes get information about sensitive data such as the secret-key.

This thesis investigates whether the official NIST HQC FPGA implementation from 2021 is vulnerable to such attacks. A FPGA board was used to run the implementation which was measured with a measurement device. Tens of thousands of power traces were created by measuring the power consumption when running the HQC decapsulation.

The power traces were used to train machine-learning models to distinguish messages based on the power consumption leakage. A convolutional neural network was trained to distinguish these groups, achieving an accuracy of over 97%. If this is combined with known attack methods, it becomes possible to recover HQC secret-keys with fewer than one thousand power traces.

The conclusion is that the NIST HQC FPGA implementation from 2021 is vulnerable to side-channel attacks showing the need for stronger implementation designs capable of withstanding these kinds of attacks. This would make the cryptosystem not only secure in theory but also in real-life scenarios ensuring secure digital communication.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

Public key encryption was first introduced in 1976 and changed secure communication drastically. Suddenly two parties could establish a secure channel without prior key exchange (or meeting in person). Since then it has become a necessity of digital security by protecting sensitive data from being leaked. However, advancements in quantum computing pose a significant threat to traditional encryption schemes. Algorithms such as RSA which are used today, could possibly be broken by a quantum computer running Shor's algorithm in the future. To make sure that the threat from quantum computers against secure communication is mitigated, the National Institute for Standards and Technology, NIST, chose two new cryptosystems, Kyber and Hamming Quasi-Cyclic encryption scheme, HQC. While much research has been done on attacks against Kyber, research is still needed on HQC. Side-channel attacks, SCAs, have been performed on software implementations on NIST HQC [2] but it has not been tested whether hardware implementations are vulnerable to SCA. This is what this thesis aims to do.

1.2 Goals and contributions

The goal of the master thesis was to investigate if the 2021 FPGA implementation of NIST HQC [1] was vulnerable to side-channel attacks using the power traces generated from the decapsulation process. The goals for the master's thesis therefore could be formulated as

1. Does the NIST submitted FPGA HQC decapsulation implementation leak information through its power consumption when run?
2. Does the hardware implementation allow for the completion of a PC oracle based on the power consumption?
3. Can a key-recovery attack be performed on the implementation submitted to NIST?

These questions were asked to figure out if the implementation was in need of a more robust approach able to protect sensitive data, or if the implementation was secure.

To answer the questions above, some advancements had to be made.

1. A general low-level wrapper for the hardware FPGA implementation was created to be able to use the decapsulation module being attacked.
2. A high-level integration using the ChipWhisperer API was created to communicate with the wrapper.
3. Data creation and analysis scripts were created.
4. Machine-learning models were designed and tested.

These different advancements made it possible to

1. Do leakage analysis to determine if information was leaking from decapsulation and suggest improvements to lessen it.
2. Use FPGAs to decapsulate ciphertexts.
3. Create a binary classification model able to distinguish messages.
4. Test different machine-learning models on power traces to see if an oracle could be created.
5. Simulate attacks against the NIST HQC FPGA implementation using real oracle accuracy.

1.3 Related work

The main article being used for this master thesis was "OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates"[2] which provided an attack against HQC using SCA. This used the power consumption from a CPU when decapsulating ciphertext to acquire leakage data used to create an oracle determining whether the ciphertext resulted in a decoding success or failure. This was later used to attack the software implementation running on the CPU by using specifically created ciphertexts to perform a key-recovery attack, analysing the oracle's decoding result when given the power trace generated during decapsulation. This can be used to see how the secret-key interacted with the ciphertext which provided information that could statistically reveal the structure of the secret-key.

The HQC authors also had to publish a hardware FPGA implementation as part of the NIST HQC submission when it was announced that HQC was one of the finalists. This had all of the components needed for HQCs KEM method which involved key-gen, encapsulation and decapsulation [1].

In 2022, a paper titled Curse of Re-encryption: A Generic Power/EM Analysis on Post-Quantum KEMs put forward that, because of the re-encryption used in many KEMs, including HQC, to achieve IND-CCA, they can be vulnerable to side-channel attacks. This paper discusses the use of the pseudorandom function, PRF, to observe different power traces from the pseudorandom function, in HQCs case the SHAKE algorithm. This can be used to train machine-learning models, namely deep learning models, able to predict if m was sent or not from the power

traces generated when decapsulating [14]. Two deep learning models were also proposed as ways of creating the power trace oracle. In Qian's article about the offline template attack, the power-trace leakage from decoding failures was used to create a decoding-failure oracle [2]. This has been shown to work on software implementations of HQC but also needs to be tested on hardware implementations. This is what this master's thesis aims to test by showing that a distinguishing oracle can be created using power-trace data to train a model to distinguish between $m = 0$ and $m = rand()$.

Chapter 2
Theory

2.1 Mathematical notations

In the current section, the background behind the HQC encryption scheme will be discussed. To be able to do this, some mathematical notations are first introduced.

2.1.1 Finite fields

When working with cryptography in real systems, the standard is that everything is in bits. A vector v can therefore be represented as $v_i \in v$, $v_i \in \mathbb{F}_2$ where $\mathbb{F}_{\not 2}$ is the binary finite field. A standard measurement of these vectors is the Hamming weight represented as:

$$w_H = \sum_{i=0}^{n-1} v_i \tag{2.1}$$

where n is the length of the binary vector.

A field is a mathematical set where addition, subtraction, multiplication and inverses are defined.

Specifically this is true when we have the binary finite field, $\mathbb{F}_2$.

2.1.2 HQC polynomial ring

All of the operations in a binary finite field are performed under modulo two because of the binary finite property of only having the elements $0, 1$ in it.

When using the encryption scheme in HQC we are working with a polynomial ring, $\mathcal{R} = \mathbb{F}_2[x]/(X^n - 1)$. In the ring each polynomial can be represented as a vector of n bits, each element representing a coefficient which can either be one or zero.

An example of this can be the polynomial $g = X^3 + X \in \mathbb{F}_2[x]/(X^4 - 1)$ which in vector representation would look like $[0, 1, 0, 1]$

To be able to describe the size of a polynomial we use Hamming weights, $w_H(g)$, where the Hamming weight of a polynomial is the number of non-zero bits in g. For example the polynomial above has the Hamming weight, $w_H(g) = 2$

2.2 Side-channel attacks

A side-channel attack, SCA, can be described as 'physical attacks on cryptographic devices take advantage of implementation-specific characteristics to recover the secret parameters involved in the computation' [7]. This can be done in many different ways by using physically recordable phenomena such as power consumption or electromagnetic radiation from a chip when doing computations to acquire information about secret-keys [7]. This can be categorized into two groups, the first being active or passive and the second invasive or non-invasive. The group active vs passive can be distinguished if the attacker is just observing the process (passive) or if the attacker is using customized data such as specific ciphertexts when attacking the process (active). The group invasive vs non-invasive is distinguished whether the attacker is satisfied with using externally provided information such as power consumption (non-invasive), or if data from inner processes is needed to attack such as reading a data bus (invasive) [7].

The attack described in Guo and Dongs paper "OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates" can be classified as a non-invasive, active attack where power consumption is used and also specific fabricated ciphertexts are used. This makes it easier to perform than invasive attacks, but not as easy as just observing random decapsulations.

2.2.1 Side-channel attack countermeasures

Masking is one of the most widely used countermeasures against SCAs where power consumption or electromagnetic radiation is used to extract information. The goal with masking is to make all computations involving sensitive data such as the secret-key mathematically randomized. This is done to prevent the power consumption, electromagnetic radiation or other real world data from correlating to the secret- key. Masking works by introducing a random share to the secret-key vector by doing

$$x = x_1 \oplus x_2 \tag{2.2}$$

where x_1 is sampled uniformly and x_2 is calculated by $x_2 = x_1 \oplus x$ where x is the true secret value [18]. The values of x_2 are then used in computation before being transformed back to the original form. This makes computations not leak as much by having intermediate values in the computation instead of the secrets. Masking must be applied to every operation involving secret-dependent data. If just one operation remains unmasked, the countermeasure can fail due to leakage from that operation. Equation 2.2 is a first order mask which hides linear leakage. Higher-order leakage can still occur, even when masking is applied. Although mitigating it requires more complex operations to create additional intermediate values, the core concept remains the same; the attack simply relies on a higher-order combination of leakage using a different equation

Hiding is another countermeasure against SCAs and works by reducing the information in the power traces or electromagnetic radiation by adding noise and making different steps constant time. This is done by the use of dummy operations

that have no effect on the real computation, introduced jitter or random delays. This makes the data harder to distinguish making the important computations with the sensitive data harder to observe.

The last countermeasure that will be presented is shuffling. Shuffling uses randomization of which operations to do next [23]. This makes the power traces differ because the operations are done in a random order.

There are more ways to countermeasure SCA, but because of the scope of the thesis, these will not be discussed.

2.3 HQC cryptosystem

In HQC, coding theory is used to make decryption hard if the secret-key is not known. This is done by assuming that decoding linear codes with noise is hard. Reed-Muller, RM, and Reed-Solomon, RS, codes are used to achieve this. To be able to explain these codes and why they make the decryption problem mathematically hard, some definitions are needed.

2.3.1 Code theory definitions

In HQC's documentation the following definitions are presented to be able to discuss RM and RS [1]. If more in-depth explanation is needed it can be studied in the official HQC documentation.

Definition 1 (Linear Code)**.** *A linear code $\mathcal{C}$ of length n and dimension k (denoted $[n, k]$) is a subspace of $\mathcal{R}$ of dimension k. Elements of $\mathcal{C}$ are referred to as codewords.*

Definition 2 (Generator Matrix)**.** *Given an $[n, k]$ code $\mathcal{C}$, a matrix $\mathbf{G} \in \mathbb{F}_2^{k\times n}$ is a generator matrix for $\mathcal{C}$ if:*

$$\mathcal{C} = \{\mathbf{mG}, \forall \mathbf{m} \in \mathbb{F}_2^k\} \tag{2.3}$$

Definition 3 (Parity-Check Matrix)**.** *Given an $[n, k]$ code $\mathcal{C}$, a matrix $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$ is a parity-check matrix for $\mathcal{C}$ if $\mathbf{H}$ is a generator matrix of the dual code $\mathcal{C}^\perp$*

$$\mathcal{C} = \{\mathbf{v} \in \mathbb{F}_2^n \text{ such that } \mathbf{Hv}^T = \mathbf{0}\} \text{ or equivalently } \mathcal{C}^\perp = \{\mathbf{uH}, \forall \mathbf{u} \in \mathbb{F}_{\not\varepsilon}^{\aleph-\urcorner}\} \tag{2.4}$$

Definition 4 (Minimum distance)**.** *Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathcal{R}$ and let w be a norm on $\mathcal{R}$. The minimum distance of $\mathcal{C}$ is:*

$$d = \min_{\substack{\mathbf{u},\mathbf{v}\in C \\ \mathbf{u}\neq\mathbf{v}}} \omega(\mathbf{u} - \mathbf{v}) \tag{2.5}$$

Definition 5 (Syndrome)**.** *Let $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$ be a parity-check matrix of some $[n, k]$ code $\mathcal{C}$, and $\mathbf{v} \in \mathbb{F}_2^n$ be a word. The syndrome of $\mathbf{v}$ is $\mathbf{Hv}^T$, and one has $\mathbf{v} \in \mathcal{C} \Leftrightarrow \mathbf{Hv}^T = \mathbf{0}$*

We can then denote a code of length n and dimension k with minimum distance d as $[n, k, d]$, which can correct up to $\Delta = [\frac{d-1}{2}]$ errors.

2.3.2 Modified Reed-Solomon codes

Reed–Solomon codes are used for error correction in many different applications such as communication systems [20], and to make decryption of HQC a hard problem to solve [1]. Reed-Solomon is a linear code. In HQC-128 it is more specifically a linear code with $n = 46$ and $k = 16$ which makes the minimum distance $d = 15$. This is not the same as the normal Reed-Solomon codes using $n = 255$ and $k = 225$ [1] which does not affect the error-correction capabilities but makes the codes more suitable for the HQC-128 security sizes because the sizes of the input and output vectors correspond to the standard values, for example n. In the Shortened Reed-Solomon code HQC-128 uses, the field being worked in is $\mathbb{F}_{2^m}$ where $m = 8$ which is represented as bytes. When encoding a message, it is therefore a message of 16 bytes that is transformed to 46 bytes. The encoding is done with the help of a RS generator polynomial described by

$$\begin{aligned} g(x) = 89 &+ 69x + 153x^2 + 116x^3 + 176x^4 + 117x^5 + 111x^6 + 75x^7 + 73x^8 \\ &+ 233x^9 + 242x^{10} + 233x^{11} + 65x^{12} + 210x^{13} + 21x^{14} + 139x^{15} \\ &+ 103x^{16} + 173x^{17} + 67x^{18} + 118x^{19} + 105x^{20} + 210x^{21} \\ &+ 174x^{22} + 110x^{23} + 74x^{24} + 69x^{25} + 228x^{26} + 82x^{27} \\ &+ 255x^{28} + 181x^{29} + x^{30} \end{aligned} \tag{2.6}$$

when HQC-128 is used [1].

To encode a message, m, with the help of the generator polynomial the message is interpreted as a polynomial over $\mathbb{F}_{2^8}$ which can be described as

$$m(x) = m_0 + m_1 x + m_2 x^2 + ... + m_{k-1} x^{k-1} \tag{2.7}$$

where $k = 16$ in HQC-128 and m_i is the different bytes in the message.

This is then multiplied by x^{n-k} and the remainder of $g(x)$ is taken.

$$b(x) = x^{n-k} m(x) mod(g(x)) \tag{2.8}$$

where $n = 46$ in HQC-128

which then gives the encoded symbols by adding the reminder to $x^{n-k}m(x)$ giving

$$c(x) = b(x) + x^{n-k} m(x) \tag{2.9}$$

This makes the first 30 bytes parity bits, while the last 16 bytes are the symbols. This makes the error correction capabilities of the Shortened Reed-Solomon decoder be 16 bytes out of 46 bytes.

To decode a message a more complex system is employed where the received messages can be described as

$$r(x) = c(x) + e(x) \tag{2.10}$$

where r is the received message which is a combination of the real message, c, and an error polynomial, e. The objective of the decoding is to calculate e so that c can be computed by

$$c(x) = r(x) - e(x) \tag{2.11}$$

To locate the errors, the syndromes described in definition 5 is used to calculate

$$S_i = r(\alpha^i) \tag{2.12}$$

where α is a primitive element of the field.

If all S_i are zero, it means no errors need to be corrected; otherwise an error-location polynomial is created, which has its roots at the locations of the errors. This makes it possible to calculate the error values, which then makes it possible to compute equation 2.11. The message can then be retrieved from the last k bytes of the message c [1]. To understand how the error-correction works it can be further studied in [22] but is not important for this thesis and will therefore not be presented. What is important is that this method can correct up to $d = 15$ errors but fails to correct most errors if more than 15 errors occur.

2.3.3 Modified Reed-Muller codes

Reed-Muller codes are another family of linear error-correcting codes used in the HQC cryptosystem. The specific Reed-Muller code used in HQC is RM(1,7), which is a first-order Boolean function with 8 dimensions, described by

$$f(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = a_0 + a_1x_1 + a_2x_2 + ... + a_7x_7 \tag{2.13}$$

where $a_i \in \mathbb{F}_2$ represent an 8-bit message, on all $2^7 = 128$ possible input vectors $(x_1, ..., x_7) \in \mathbb{F}_2^7$. This produces a binary codeword of length 128 from a message of length 8 bits.

This makes RM(1,7) be a linear code of $[128, 8]$, which has a $d = 64$ [1]

HQC uses concatenated Reed–Muller codes with a multiplicity of 3 for HQC-128, meaning that each codeword created from a message is concatenated three times, described by

$$c = (c_0 = RM(m))|(c_1 = RM(m))|(c_2 = RM(m)) \tag{2.14}$$

This is done to strengthen the error-correction capabilities of the decoding process. This means that the modified Reed-Muller code used is a linear code of $[384, 8]$ with $d = 192$.

The decoding is done using the Fast Hadamard transform with slightly modified input because of the use of concatenated Reed-Muller codes which uses a mapping described by $F : \mathbb{F}_2^3 \to \{-3, -1, 1, 3\}$ described by

$$F(x_1, x_2, x_3) = (-1)^{x_1} + (-1)^{x_2} + (-1)^{x_3} \tag{2.15}$$

where $x_i \in \mathbb{F}_2$.

The Fast Hadamard transform is not relevant to understand beyond the fact that it can correct up to 64 bit errors out of the 128 bits.

2.3.4 Concatenated Modified Reed-Muller and Reed-Solomon codes

When Modified Reed-Muller and Shortened Reed-Solomon codes are concatenated a new linear code can be created by having a 16 byte message be turned into 46 bytes with the Shortened Reed-Solomon encoder, this can then be passed into the Modified Reed-Muller code being transformed into 46 blocks of 384 bits, equalling to $17,664$ bits which is the same as $n_1 n_2$ in table 2.1. This is called having the Shortened Reed–Solomon code as the outer code and that the Modified Reed-Muller code is the inner code.

When decoding 17664 bits are then split into 46 blocks with 384 bits that are then transformed to 46 bytes which is then transformed to the 16-byte message if not more than 15 errors are present in the 46 Reed-Solomon bytes.

2.3.5 Hamming Quasi-Cyclic Public key encryption

In 2016, the HQC cryptosystem for public key encryption, PKE, was proposed for the first time [10]. This uses the difficulty of decoding random quasi-cyclic codes with added randomness to it. The idea came from another paper by Alekhnovich in 2003 but was altered to use the random quasi-cyclic codes instead of random linear codes [10].

The Hamming Quasi-Cyclic cryptosystem needs to define a couple of parameters which can be seen in table 2.1. This paper only studies HQC-128 and therefore only the values for this instance are described whereas HQC-192 and HQC-256 also exist for better security using different parameters.

The six first parameters in table 2.1 are for Reed-Solomon and Reed-Muller and was discussed in the previous sections.

Instance	n_1	k_1	d_{RS}	Mult.	n_2	d_{RM}	$n_1 n_2$	n	w	$w_r = w_e$
hqc-128	46	16	15	3	384	192	17664	17669	66	75

Table 2.1: The parameters and values for HQC-128. The six first parameters are for the Reed-Solomon and Reed-Muller codes while the last four is four the the polynomials.

To construct a secret-key, n from the table 2.1 is used to create an irreducible polynomial which is used to create the polynomial ring described in section 2.1.2. From this three polynomials, $h, x, y \in \mathcal{R}$, with $w_H(x) = w_H(y) = w$ where w can be found in table 2.1 are sampled from the polynomial ring. The secret-key then becomes the pair (x, y) and the public key becomes $(h, s = x + hy)$.

To encrypt data three new polynomials are sampled from the polynomial ring, $r_1, r_2, e \in \mathcal{R}$. These are all sampled so that $w_H(r_1) = w_H(r_2) = w_r$ and $w_H(e) = w_e$. To encrypt data these polynomial are then used with the public key to encrypt data by creating the ciphertext (u, v) as

$$u = r_1 + hr_2 \tag{2.16}$$

$$v = m\mathbf{G} + sr_2 + e \tag{2.17}$$

where $\mathbf{G}$ is a generator matrix created by using the concatenated Shortened Reed-Solomon and Modified Reed-Muller codes to encode the message. It is worth noting that this is not a real matrix, but it can be described as one because of its linear properties.

The last step of having a complete public key encryption scheme is the decryption which is done in the HQC cryptosystem as

$$\begin{aligned} v - uy &= m\mathbf{G} + sr_2 + e - (r_1 + hr_2)y \\ &= m\mathbf{G} + (x + hy)r_2 + e - (r_1 + hr_2)y \\ &= m\mathbf{G} + e + r_2x + r_1y \end{aligned}$$

After this computation has been done the decoder connected to $\mathbf{G}$ is used to decode the message. This can be done as long as each block in $m\mathbf{G} + e + r_2x + r_1y$ does not break the error correcting capabilities of the concatenated Modified Reed-Muller and Shortened Reed-Solomon decoder. If the decoder is not able to decode it properly, it will result in a decoding failure which will create a decryption failure.

2.4 Hamming Quasi-Cyclic Key Encapsulation Mechanism

The HQC Key encapsulation mechanism, KEM, is similar to the PKE with the exception of removing the choice of the message and making the process IND-CCA secure by performing re-encryption in the decapsulation function to check the ciphertext's authenticity. The KEM process involves three different steps. Key generation, encapsulation and decapsulation, which can be seen in figures 2.1, 2.2 and 2.3

As can be seen in figure 2.1, the KEM process starts with the creation of an encapsulation key; public key, and a decapsulation key; secret-key. This is done by creating a seed used to generate the keys.

The second part of the HQC KEM can be seen in figure 2.2. It can be seen in the figure that a message of length k bytes and a salt is sampled. These, together with the hashed public key, are concatenated and hashed with another hash function to generate a shared-secret, K, and a seed, θ. The seed is used with the sampled message and public key to encrypt the message, and then concatenated with the salt to create the HQC KEM ciphertext.

The last part of the HQC KEM is the decapsulation. This is done by splitting the data and decrypting the ciphertext that was sent without the salt. The process then computes the shared-secret, K' , and seed, θ, and the ciphertext again using the salt and decrypted message. This is done to be able to check if the computed ciphertext is the same as the ciphertext received.If not, then the shared-secret becomes trash to make the KEM IND-CCA secure."

2.4.1 OT-PCA attack

In OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates a key-recovery attack is described against HQC using side-channel data to construct a decoding oracle to extract information

HQC-KEM.Keygen()

Input: None.

Output: Encapsulation key $\mathsf{ek}_{\mathsf{KEM}}$, decapsulation key $\mathsf{dk}_{\mathsf{KEM}}$.

▷ Sample $\mathsf{seed}_{\mathsf{KEM}}$

1. $\mathsf{seed}_{\mathsf{KEM}} \leftarrow\$ \mathcal{B}^{|\mathsf{seed}|}$

▷ Compute $\mathsf{seed}_{\mathsf{PKE}}$ and randomness σ

2. $\mathsf{ctx}_{\mathsf{KEM}} \leftarrow \mathsf{XOF.Init}(\mathsf{seed}_{\mathsf{KEM}})$
3. $(\mathsf{ctx}_{\mathsf{KEM}}, \mathsf{seed}_{\mathsf{PKE}}) \leftarrow \mathsf{XOF.GetBytes}(\mathsf{ctx}_{\mathsf{KEM}}, |\mathsf{seed}|)$
4. $(\mathsf{ctx}_{\mathsf{KEM}}, \sigma) \leftarrow \mathsf{XOF.GetBytes}(\mathsf{ctx}_{\mathsf{KEM}}, |k|)$

▷ Compute HQC-PKE keypair

5. $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}}) \leftarrow \mathsf{HQC\text{-}PKE.Keygen}(\mathsf{seed}_{\mathsf{PKE}})$

▷ Compute HQC-KEM keypair

6. $\mathsf{ek}_{\mathsf{KEM}} \leftarrow \mathsf{ek}_{\mathsf{PKE}}$
7. $\mathsf{dk}_{\mathsf{KEM}} \leftarrow (\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{PKE}}, \sigma, \mathsf{seed}_{\mathsf{KEM}})$
8. **return** $(\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{KEM}})$

Figure 2.1: HQC KEM keygen description [1]

HQC-KEM.Encaps($\mathsf{ek}_{\mathsf{KEM}}$)

Input: Encapsulation key $\mathsf{ek}_{\mathsf{KEM}}$.

Output: Shared secret key K, ciphertext $\mathbf{c}_{\mathsf{KEM}}$.

▷ Sample message $\mathbf{m}$ and salt

1. $\mathbf{m} \leftarrow\$ \mathcal{B}^{|k|}$
2. $\mathsf{salt} \leftarrow\$ \mathcal{B}^{|\mathsf{salt}|}$

▷ Compute shared key K and ciphertext $\mathbf{c}_{\mathsf{KEM}}$

3. $(K, \theta) \leftarrow \mathsf{G}(\mathsf{H}(\mathsf{ek}_{\mathsf{KEM}}) \| \mathbf{m} \| \mathsf{salt})$
4. $\mathbf{c}_{\mathsf{PKE}} \leftarrow \mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{KEM}}, \mathbf{m}, \theta)$
5. $\mathbf{c}_{\mathsf{KEM}} \leftarrow (\mathbf{c}_{\mathsf{PKE}}, \mathsf{salt})$
6. **return** $(K, \mathbf{c}_{\mathsf{KEM}})$

Figure 2.2: HQC KEM encapsulation description [1]

HQC-KEM.Decaps($\mathsf{dk}_{\mathsf{KEM}}, \mathbf{c}_{\mathsf{KEM}}$)

Input: Decapsulation key $\mathsf{dk}_{\mathsf{KEM}}$, ciphertext $\mathbf{c}_{\mathsf{KEM}}$.
Output: Shared secret key K'.

▷ Parse decapsulation key $\mathsf{dk}_{\mathsf{KEM}}$
1. $\mathsf{ek}_{\mathsf{KEM}} \leftarrow \mathsf{dk}_{\mathsf{KEM}}[0 : |\mathsf{ek}_{\mathsf{KEM}}|]$
2. $\mathsf{dk}_{\mathsf{PKE}} \leftarrow \mathsf{dk}_{\mathsf{KEM}}[|\mathsf{ek}_{\mathsf{KEM}}| : |\mathsf{ek}_{\mathsf{KEM}}| + |\mathsf{dk}_{\mathsf{PKE}}|]$
3. $\sigma \leftarrow \mathsf{dk}_{\mathsf{KEM}}[|\mathsf{ek}_{\mathsf{KEM}}| + |\mathsf{dk}_{\mathsf{PKE}}| : |\mathsf{ek}_{\mathsf{KEM}}| + |\mathsf{dk}_{\mathsf{PKE}}| + |\sigma|]$

▷ Parse ciphertext $\mathbf{c}_{\mathsf{KEM}}$
4. $\mathbf{c}_{\mathsf{PKE}} \leftarrow \mathbf{c}_{\mathsf{KEM}}[0 : |\mathbf{c}_{\mathsf{PKE}}|]$
5. $\mathsf{salt} \leftarrow \mathbf{c}_{\mathsf{KEM}}[|\mathbf{c}_{\mathsf{PKE}}| : |\mathbf{c}_{\mathsf{PKE}}| + |\mathsf{salt}|]$

▷ Compute message $\mathbf{m}'$
6. $\mathbf{m}' \leftarrow \mathsf{HQC\text{-}PKE.Decrypt}(\mathsf{dk}_{\mathsf{PKE}}, \mathbf{c}_{\mathsf{PKE}})$

▷ Compute shared key K' and ciphertext $\mathbf{c}'_{\mathsf{KEM}}$
7. $(K', \theta') \leftarrow \mathsf{G}(\mathsf{H}(\mathsf{ek}_{\mathsf{KEM}}) \| \mathbf{m}' \| \mathsf{salt})$
8. $\mathbf{c}'_{\mathsf{PKE}} \leftarrow \mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{KEM}}, \mathbf{m}', \theta')$
9. $\mathbf{c}'_{\mathsf{KEM}} \leftarrow (\mathbf{c}'_{\mathsf{PKE}}, \mathsf{salt})$

▷ Compute rejection key $\bar{K}$
10. $\bar{K} \leftarrow \mathsf{J}(\mathsf{H}(\mathsf{ek}_{\mathsf{KEM}}) \| \sigma \| \mathbf{c}_{\mathsf{KEM}})$
11. **if** $\mathbf{m}' = \perp$ **or if** $\mathbf{c}'_{\mathsf{KEM}} \neq \mathbf{c}_{\mathsf{KEM}}$
12. $\quad K' \leftarrow \bar{K}$
13. **endif**

14. **return** K'

Figure 2.3: HQC KEM decapsulation description [1]

on the properties of the secret-key. This attack can be performed in the following manner:

The first step of this process is to design malicious ciphertexts using the public key (h,s). This is done by creating ciphertexts using $m = 0$, $r_1 = X^k$ and $r_2 = 0$ along with $m = 0$, $r_1 = 0$ and $r_2 = X^k$. This creates the ciphertexts as

1. $(X^k, m\mathbf{G} + e = (X^k, e)$
2. $(hX^k, m\mathbf{G} + sX^k + e) = (hX^k, sX^k + e)$

which when the decryption is executed results in

$$v - uy = e + X^k y \tag{2.18}$$

$$v - uy = sX^k + e - hX^k y = (x + hy)X^k + e - hX^k y = e + X^k x \tag{2.19}$$

The decryption result $v - uy$ needs to be decoded. The decoder can as seen in section 2.3.2 correct up to 15 errors for HQC-128 and mostly fails when 16 or more errors are inserted.

This can be done by creating $\frac{\delta_{RS}-1}{2} = 15$ blocks of errors containing only ones, which guarantees a decoding failure of that block in the Reed-Muller decoder. This results in a Shortened Reed-Solomon symbol being wrong. If done correctly this should not result in a decoding failure because Shortened Reed-Solomon decoder can correct 15 errors or less.

The first block of the error vector is the only block in the error vector that should not be used as a corruption block used to produce a guaranteed failure. This is instead used to be the deciding block if the decoding is supposed to fail or not. This is done by maximizing the Shannons entropy of a Reed-Muller block being decoded correctly or not. The Reed-Muller decoding problem being optimized is the decoding of an arbitrary 46 byte block from a secret-key. The Reed-Muller decoding problem can be described as

$$decode(e_{block} + secret\text{-}key_{block}) \tag{2.20}$$

If this problem is optimized to maximize entropy it will result in the decoding failing about half of the times and succeeding half of the time. The block created are called error patterns.

The next step of the attack is to create the decoding oracle. This is achieved by decapsulating ciphertexts described above to get the power traces. These are then used to train a machine-learning model to identify whether the power trace is from a decapsulation where a decoding failure occurred or not. The decoding results are gathered by using the software implementation of the Reed-Muller decoder on the decryption result from the ciphertext, secret-key pair to acquire the decoding result to be able to train the model with supervised learning, where the correct answer is known. The power trace is therefore the input, and the decoding failure or success is the output from the model.

The offline templates for each error pattern is created to determine the probabilities of different bits being one in the secret-key block being targeted, dependent

on k in equation 2. These are created by sampling 10^6 different sparse vectors and generating a mapping for each sampled vector and its result when decoded with the chosen error. This is then saved in a lookup table.

To finalize the attack the error patterns are used to create different ciphertexts that are sent to the hardware being attacked with 46 rotations of k to obtain information on x and y in the secret-key. This is done for a few error patterns, depending on the quality of the decoding oracle, until enough power traces have been collected.

The power traces are then evaluated by the decoding oracle returning either failure or success. This is used together with the error templates to calculate the probability of a position being 0 or 1. These are later sorted and used with Gaussian elimination to calculate each bit. A more in-depth explanation can be found in [2] if needed. For this thesis it is only needed to understand the large picture therefore it will not be discussed further.

2.5 Statistics

2.5.1 Welch t-test and p-value

The Welch t-test is a statistical method of determining whether the means of two independent groups differ.

If we have two groups X and Y defined by:

$$X \sim \mathcal{N}(\mu_1, \sigma_1^2) \tag{2.21}$$

$$Y \sim \mathcal{N}(\mu_2, \sigma_2^2) \tag{2.22}$$

The Welch t-test can be calculated as

$$t = \frac{\bar{X} - \bar{Y}}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}} \tag{2.23}$$

where s_1 and s_2 are the sample variances from the different groups. N_1 and N_2 are the number of samples in the group. Lastly $\bar{X}$ and $\bar{Y}$ is the mean of the group.

This gives a large t-value when the mean difference is large and the sample variances are small.

When using the t-test, the sign of t is not of interest which is why the absolute value of t is used.

One value associated with the t-value is the p-value. The p-value is the probability of rejecting the null hypothesis, H_0, and can be calculated as described in [6] by integrating

$$f(t, v) = \frac{\Gamma(\frac{v+1}{2})}{\sqrt{\pi v}\Gamma(\frac{v}{2})}(1 + \frac{t^2}{v})^{-\frac{v+1}{2}} \tag{2.24}$$

as

$$p = 2 \int_{|t|}^{\infty} f(t, v)dt \tag{2.25}$$

where v is the degree of freedom of associated with the t-test described in [6].

The p-value describes the probability of the null hypothesis being true based on the data given [6]. This means that if the null hypothesis is $H_0 : \bar{X} = \bar{Y}$ a low value is the same as the data of the groups given has a low probability of having the same mean value.

A simple t-value to use as a threshold is 4.5, because it results in a p-value of $p < 0.00001$ under the null hypothesis [6].

2.5.2 Shannon entropy

Entropy is a way of determining the amount of randomness in a discrete random variable, X. This is done by calculating the entropy as

$$H(X) = -\sum_{x \in X} p(x)log(p(x)) \tag{2.26}$$

where x is each element in X and $p(x)$ is the probability of getting x if sampling X

If the logarithm chosen is log_2, called Shannon entropy, then we can create an optimization problem to maximize the entropy. This is done by trying to make X be uniformly distributed. If the entropy is maximized successfully then the result is a uniform distribution.

2.6 Hardware notations

2.6.1 Field programmable gate arrays

An FPGA is a integrated circuit, IC, with the special function of being able to be restructured after leaving the factory. An FPGA is a semi-ready silicon chip made to be programmed electronically to be used in a system or digital circuit [11]. This creates the possibility of testing different IC structures by programming the FPGA to be restructured in the same way as a real integrated circuit. An FPGA can be viewed as three parts. The first is the configurable logic blocks, CLB, which also are called logic blocks. The logic blocks have the function of being programmed to different logical layouts which makes up the IC structure being tested.

The second part is the programmable interconnections. These are what connects different logic blocks and program the logic blocks. Lastly are the I/O blocks which is used for input and output from the FPGA logic blocks and can be programmed in different ways. These parts together can be seen in figure 2.4 where the I/O blocks are located close to the edge of the FPGA while the logic blocks are found in a grid with intercommunications between them.

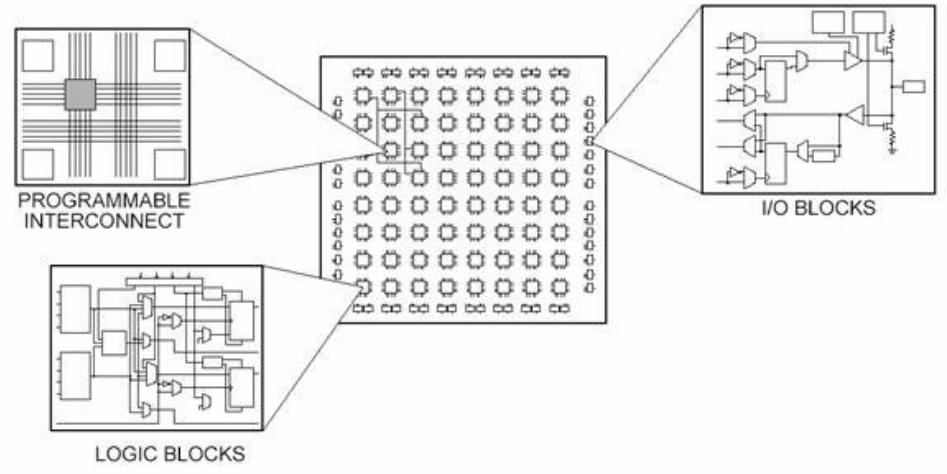

Figure 2.4: Illustration of FPGA structure taken from [12].

2.6.2 Final state machines

A finite state machine, FSM, is a computational model used to describe in what order different modules in a system should be executed. This is done by having a finite number of states that are run in order so that it can be guaranteed that a resource is ready before being used.

A FSM can be divided into three parts which can be described as

1. States - specific values representing what the system is doing currently.
2. Transitions - rules for how the system should move from one state to another which can depend on either only the state or the state and input.
3. Outputs - the signals generated by the different states, which can depend on the input and the state.

There are two common types of FSMs, Moore- and Mealy machines. The difference between Moore- and Mealy machines are how they output data. In Moore machines the output is only dependent on the state which it is in. Meanwhile Mealy machines are dependent on the state and the input to its output. This makes Moore machines more stable while Mealy machines react faster to input changes but are more complex because of the instability it introduces.

An example of a final state machine can be seen below in Listing 2.1 where we have three states, idle, read and write. The transition between the different states are done in the always block starting on row 16. This rotates through the states starting on idle, continuing to the read state where it remains until done is set high, after which it moves to the write state and then back to the idle state. This prevents the read and write from happening at the same time which makes the process lose speed but gain stability.

```
1 typedef enum logic [1:0] {
```

```
2       IDLE,
3       READ,
4       WRITE
5   } state_t;
6
7   state_t state, next_state;
8
9   always @(posedge clk or negedge rst_n) begin
10      if (!rst_n)
11          state <= IDLE;
12      else
13          state <= next_state;
14  end
15
16  always @(*) begin
17      case (state)
18          IDLE:   next_state = start ? READ  : IDLE;
19          READ:   next_state = done  ? WRITE : READ;
20          WRITE:  next_state = done  ? IDLE  : WRITE;
21          default: next_state = IDLE;
22      endcase
23  end
```

Listing 2.1: A simple Moore machine

This can be important if data is transferred from one system to another, where the process needs to wait for the data to be finished sending or computation is needed before another system can start.

2.6.3 Little-endian vs Big-endian

When packing bytes into larger arrays such as 32-bit or 64-bit arrays there are two primary ways of storing bytes; little and big-endian [8]. These are crucial when working with different processes such as when using different processing structures and communication protocols. Most processors, such as processor architectures, use little-endian, while most network protocols use big-endian [8].

The difference is how bytes are stored in vectors, and is better described with a figure, see figure 2.5.

When looking at the transformation in the figure above it can be seen that the storage of bytes only varies by how each byte is placed in the array. In little-endian, the least significant byte is placed in the lowest numbered address. In big-endian, the most significant byte is instead placed in the lowest numbered address which makes it appear as if nothing is changed.

2.6.4 8N1 UART

To send data between devices, different communication protocols can be used, such as TCP, USB, and UART. They all have the function of sending bytes from one device to another over some hardware interface. For this thesis UART will be used and therefore explained. The Universal Asynchronous Receiver and Transmitter,

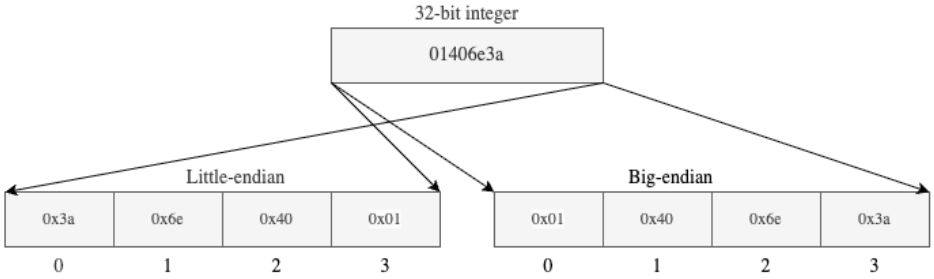

Figure 2.5: Endian differences when packed into a 32-bit array from a 32-bit integer.

UART, protocol is an asynchronous, serial communication protocol which allows two devices to share data without using the same clock by agreeing on a BAUD rate (bits/second). When transmitting or receiving data, this is done over two wires, one for transmit and one for receive.

The UART configuration that was used in this thesis is 8N1 where one packet can be seen as in figure 2.6

UART uses little-endian bit ordering, which makes the least significant bit, LSB, sent first, making the byte appear reversed.

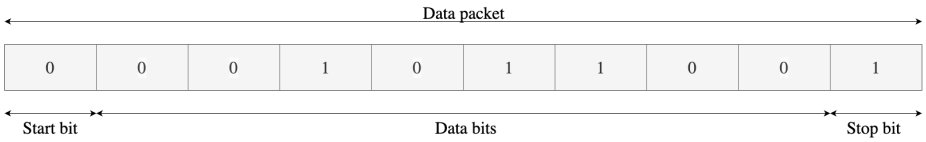

Figure 2.6: 8N1 UART where the start bit always is 0 and stop bit 1. Idle state is high (1). In the example above, the value 0x34 is being sent.

2.6.5 Shared Memory Interface

The shared memory interface being used is a slave–master structure where the slave provides the data while the master accesses it. This is done to make communication between modules as smooth as possible. An example of this can be seen below where the slave handles the data for the master. The master sends the address which it wants to read from and then a signal indicating that it wants the data which makes the slave present the data asked for, in the example below it presents a byte .

```
1     input  wire [12:0] ct_addr
2     input  wire        ct_ce
3     output reg  [7:0]  ct_data
```

```
4
5       reg [7:0]  ct_mem [0:CT_BYTES-1];
6
7       always @(posedge clk) begin
8           if (ct_ce)
9               ct_data <= ct_mem[ct_addr];
10      end
```

2.7 Machine-learning models

Machine-learning models can be used to classify data by taking an input and giving an output. The use of such models has had a great impact on side-channel attacks against different cryptosystems, helping to create different oracles needed for different attacks.

2.7.1 Normalization

When using machine-learning models normalization is often needed to achieve any accuracy gain better than the model simply guessing [19]. This can be done in many ways where one of the most common ways of normalizing data is standardization, which can be described as

$$v_{scaled} = \frac{v - \mu}{\sigma_A} \tag{2.27}$$

where v is a vector, μ is the mean and σ is the standard deviation. Described by the following equations

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} v_i \tag{2.28}$$

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(v_i - \mu)^2} \tag{2.29}$$

The standardization makes each vector have $\mu = 0$ and $\sigma = 1$, which makes outliers in the data stand out more by removing noise shared throughout the whole vector. This results in the machine-learning model not having to learn small differences often found in power traces from hardware implementation where difference in power at a specific point is very subtle.

2.7.2 Logistic regression

One of the most used machine-learning models is logistic regression taking an input, $x \in \mathbb{R}$ and giving $y \in Y$ where $Y = \{0, 1\}$. This can be described as

$$\hat{y} = \begin{cases} 1, & \text{if } \sigma(\mathbf{w}^\top \mathbf{x} + b) \geq 0.5, \\ 0, & \text{otherwise.} \end{cases} \tag{2.30}$$

where $\hat{y}$ is the predicted y value, $\mathbf{w}$ is the weight vector, b is the bias and x is the input vector.

The function σ is the sigmoid function, which maps real-valued input to the open interval between zero and one, $\sigma : \mathbb{R} \to (0, 1)$ which is described as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.31}$$

In logistic regression, $\sigma(\mathbf{w}^\top \mathbf{x} + b)$ represents the estimated probability that the input $\mathbf{x}$ belongs to a class. If this is used together with a threshold of 0.5 as in equation 2.30.

This can be interpreted with the probability function P as

$$P(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b) \tag{2.32}$$

which gives the probability of $y = 1$ if $\mathbf{x}$ is the input. Because of the binary classification the probability of having $y = 0$ then becomes $P(y = 0|\mathbf{x}) = 1 - P(y = 1|\mathbf{x})$.

To get the logistic regression to work, tuning of the weight vector, $\mathbf{w}$, and the bias, b, need to be tuned for the dataset $D = \{\mathbf{x}_i, y_i\}$ where $0 \leq i$, with training data from the classes it should be able to distinguish between.

Therefore we want to make the difference between $\hat{y}_i$ and y_i as small as possible. This is done by using a cost function to measure this difference.

The normal cost function used is the cross-entropy loss function which can be described as

$$\mathcal{L}(\mathbf{w}, b) = -\frac{1}{N} \sum_{i=1}^{N} (y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)) \tag{2.33}$$

When $\mathcal{L}$ becomes smaller it means the predicted values are getting closer to the real classification. This is then used with an optimization method to minimize $\mathcal{L}$. One optimization method is gradient descent which can be described as

$$\begin{cases} \mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b) \\ b \leftarrow b - \eta \frac{\partial \mathcal{L}(\mathbf{w}, b)}{\partial b} \end{cases} \tag{2.34}$$

This is done iteratively where η is the learning rate which describes how large the steps in gradient descent should be.

When this is done the weights, $\mathbf{w}$, and the bias, b, are optimized to classify the data given as well as possible. If there is enough difference between the classes a higher accuracy will be achieved, where 1 is the best (all data classified correctly), and 0.5 is the worst, where it predicts the right answer 50% of the time which means no distinction can be made.

2.7.3 Multilayer perceptron

Another machine-learning model often used in SCA is the multilayer perceptron which is part of the artificial neural network family, making it able to model both

linear and non-linear relationships between the input and output. This makes the multilayer perceptron, MLP, more versatile than the logistic regressor.

The MLP can be seen as three parts:

1. The input layer which receives the input data
2. Hidden layers are where the network learns to represent the data
3. Output layer where the final prediction is done

These parts can be seen in figure 2.7 where the data comes to the input layer to then be passed down through the hidden layers to the output layer. Each neuron has an input of data, which is then multiplied by the neuron's weights. The values are then summed and sent through a non-linear function θ.

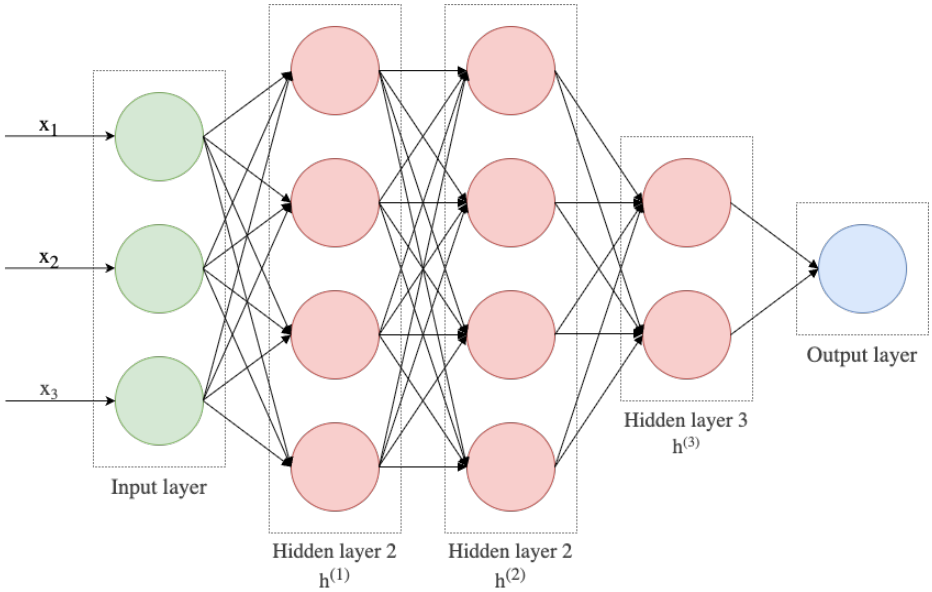

Figure 2.7: Overview of neural network structure of a multilayer perceptron with fully connected layers.

This can be described as

$$h = \theta(\mathbf{w}^T\mathbf{x} + b) \tag{2.35}$$

where h is the output from a neuron and $\mathbf{w}$ is the weight vector for the neuron,$\mathbf{x}$ is the input vector and b is the bias.

h can be described as

$$h = \theta(\sum_i w_i x_i + b) \tag{2.36}$$

To describe this for a whole layer the notation can be described as

$$h^{(l)} = \theta(\mathbf{W}^l h^{(l-1)} + b^{(l)}) \tag{2.37}$$

where $\mathbf{W}$ is a matrix with all of the weights in a layer, b is the bias vector for the layer.

For the output layer, a sigmoid function like the one in equation 2.31 is used to obtain a probability instead of logits.

When doing the binary classification the same loss function can be used as in the logistic regression along with the gradient descent method. Described in equation 2.33 and equation 2.34 which is done iteratively in batches which is then used to optimize the weights in the different layers.

The most important thing in the neural network is the introduction of the non-linear function θ. This introduces non-linearity in the system, making the transformation not necessarily linear.

There are many different non-linear functions that can be used but the most popular one is the ReLu function described as

$$\theta(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases} \tag{2.38}$$

This function is computationally efficient and the most popular non-linear function for neural networks.

With this structure it is possible to predict both linear and non-linear correlations in the data provided.

2.7.4 Convolutional neural networks

A Convolutional neural networks, CNNs, is another machine-learning model often used in side-channel attacks that has been shown to have high potential. CNNs differ from fully connected networks by exploiting the spatial locality present in side-channel traces [15]. Instead of treating all time samples as equally related, CNNs learn small local patterns that correspond to leakage points.

A CNN layer uses a set of kernels (also called filters) that slide over the input. Each kernel is a small vector of trainable weights that is applied across the entire trace, which is known as weight sharing. For a 1 dimensional input trace, the convolution operation is given by

$$h(t) = \theta(\sum_{i=0}^{k-1} \mathbf{w}_i \mathbf{x}_{i+t} + b) \tag{2.39}$$

where θ is the non-linear function, $\mathbf{w}_i$ is the kernel at position i, $\mathbf{x}_{i+t}$ is the input at position $i + t$ and b is the bias.

To describe this it is better to use an example that can be seen in figure 2.8. In this example, we observed that each layer takes a vector as input and produces a weighted sum over the neighbourhood surrounding $x + i + 1$. This makes the output of a layer consist of features of different parts of the vector.

Stride and padding are two important concepts in CNNs. The stride controls how far the kernel moves each time it is applied. With stride = 1, the kernel

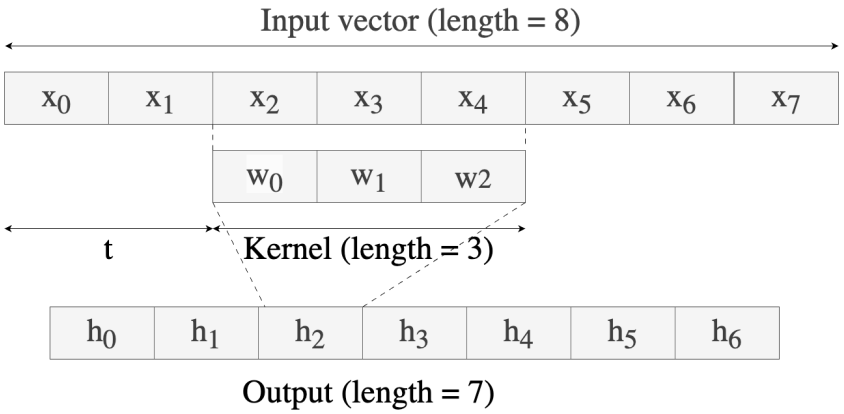

Figure 2.8: Example of how CNN layer works when stride = 1, padding = 0, kernel size = 3.

shifts by one sample after each convolution, producing an output at every possible position. With stride = s, the kernel moves s samples at a time, which reduces the number of positions where the kernel is applied and therefore decreases the size of the output vector.

Padding is used to counter this reduction by adding zeros to the beginning and end of the input vector. If p zeros are added on each side, the effective input length becomes $n+2p$ for an original trace of length n. This allows the convolution to operate near the borders of the trace and can preserve the original input size. For a 1D convolution with input length n, kernel size k, stride s, and padding p, the output length is given by the equation below [16]

$$L = \left\lfloor \frac{n + 2p - k}{s} \right\rfloor + 1. \tag{2.40}$$

Increasing the stride therefore reduces the output size, while padding can be used to keep it.

Instead of having one kernel, a CNN layer typically applies many kernels in parallel. Each kernel learns a different set of weights, making the layer extract multiple types of local features from the trace. The output of the layer is therefore not a single vector but a collection of vectors, each one created by a kernel.

These vectors form the input to the next convolutional layer. Conceptually, each vector from the previous layer becomes a channel in a multi-channel input. The kernels in the next layer operate across all these channels and produce a new set of vectors. Figure2.9 illustrates this process: the first layer uses three kernels, producing three vectors, and the second layer uses two kernels, resulting in six vectors in total.

After the final convolutional layers, the resulting multi-channel representation is flattened by concatenating all vectors into one single vector. This vector is then passed to one or more fully connected layers which perform the final classification.

As can be seen in equation 2.39 each layer has a non-linear function to introduce complexity into the network, where it is recommended to use ReLU [15].

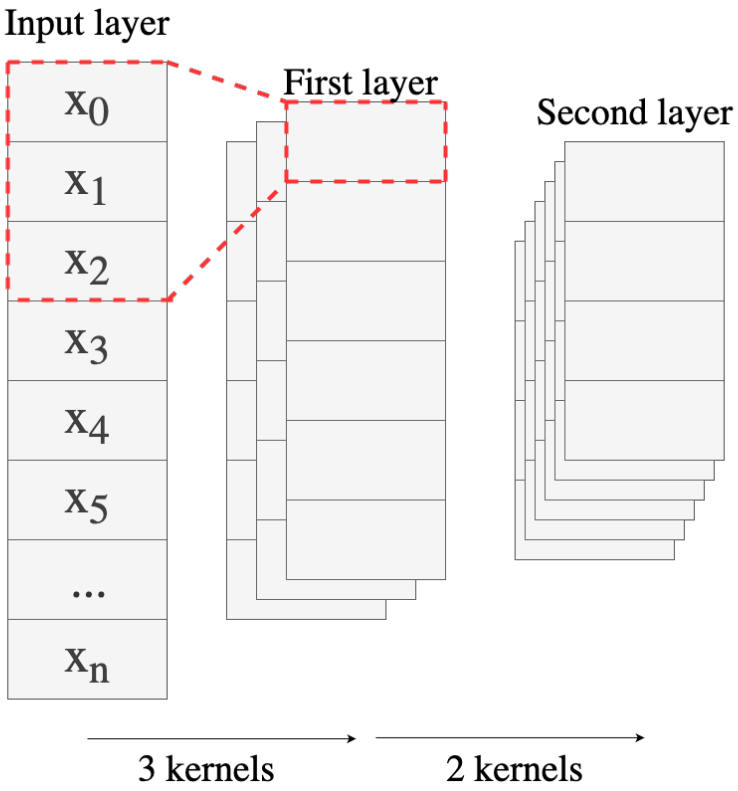

Figure 2.9: Example of how the number of kernels affects the matrix.

CNNs, just like MLPs, uses gradient descent to train the weights using back propagation.

There are many more concepts to CNNs such as pooling but because these are not used in this paper, they will not be discussed. These concepts can be studied further in papers focusing on CNNs, such as [16]

Chapter 3
Method

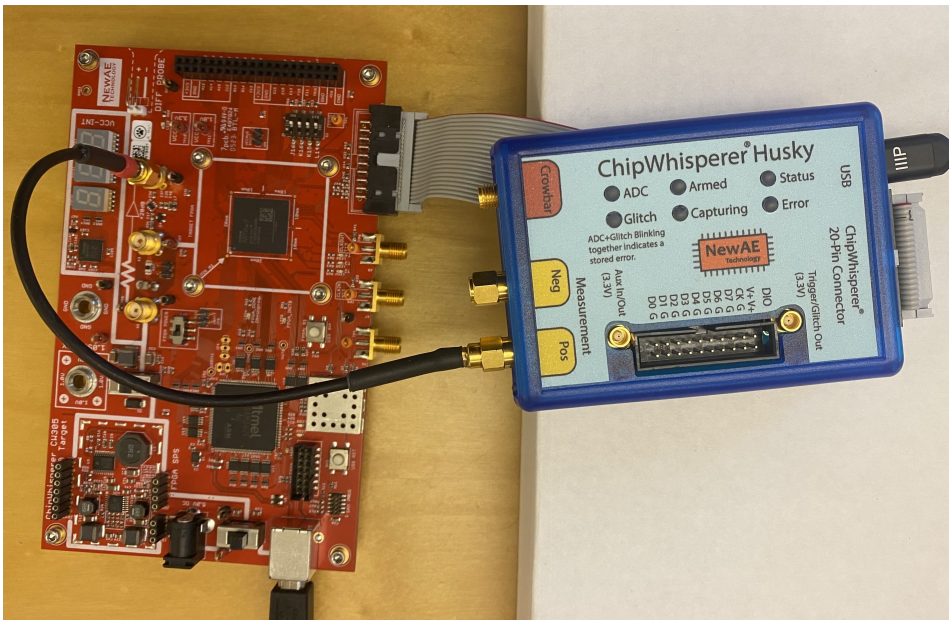

Figure 3.1: The CW305 board being used as the attack target and the Husky capturing the power traces

3.1 Hardware overview

When testing the suggested NIST HQC FPGA implementation two things were needed, a bitstream synthesized by a program in this case Xilinx Vivado, and hardware. The hardware was chosen with the knowledge that we are trying to perform SCA on HQC's hardware implementation [1]. This made ChipWhisperer components from NewAE Technology a natural choice because of their ability to easily connect measuring devices to the FPGA by design. The board that was chosen was the ChipWhisperer CW305, which had an SMA connector, X4, with a

20 dB low-noise amplifier that made it easy to measure the power consumption. To measure it, an ADC (ChipWhisperer Husky), analogue-to-digital converter, was connected with the positive connector leaving the negative connector grounded. The Husky, in addition to being an ADC, also supported the FPGA with a clock making them synchronized.

3.1.1 ChipWhisperer CW305

The ChipWhisperer CW305 is a FPGA board used for SCA testing designed to easily receive data, glitch different aspects such as voltage, and measure real-world data such as electromagnetic radiation and power consumption from the FPGA running on the board. The CW305 also features USB interface, external PLL for clocking the FPGA and much more [17]. Even though the CW305 features a USB interface, this was not used in the design of the hardware wrapper to make it more versatile and adaptable to different FPGA boards, avoiding binding the wrapper to the ChipWhisperer platform. The board also included an easy way to connect to the ChipWhisperer Husky with a 20-pin layout connecting to the FPGA for easy transfer of information using serial communication like UART. The FPGA is an Artix-7 FTG256 which has a supported toolchain in Xilinx Vivado making it easy to synthesize.

3.1.2 ADC setup

The ADC used was the ChipWhisperer Husky to measure the power consumption. Because of the use of the same clock, the Husky was set to capture the power traces with four times sampling rate. Therefore the ADC was running on 200 MHz while the clock was running on 50 MHz which made the power traces capture four sample points for each clock cycle. This was done to capture more in-depth information in the power consumption that would not appear with an ADC multiplier of one. The Husky also has the ability to amplify the signals to better use the whole range of the 12-bit capturing range. In this report the used ADC dB gain was 14 which was optimized to avoid ADC clipping while still using most of the 12-bit range.

3.1.3 Overlook of HQC hardware implementation

The hardware implementation created for the NIST submission had three different modules, key-gen, encapsulation and decapsulation. These together created a valid KEM. The decapsulation module was the only module used in this thesis. To be able to decapsulate, the wrapper had to be designed to work together with the module so that it operated correctly. The module for decapsulation required wiring from the top module and can be seen in table 3.1. The clock was supplied from an external source via the N14 pin on the CW305 located in the 20-pin connector connected to the Husky. The reset was kept low except when initiating the bitstream on the CW305. The start was triggered every time the data had been loaded onto the CW305 board.

The module also had three similar input/output sources for the shared secret, ciphertext and secret-key being used in the process. All three of these used a shared

memory interface discussed in the theory in section 2.6.5 using the master, slave dynamic, where the decapsulation module was the master and the other modules were the slave. This was done to guarantee that data was presented correctly as requested by the decapsulation module The last wiring for the module to work as it was expected to do was the ap_return which is a status bus returning zero if decapsulation succeeded and one otherwise.

The decapsulation module follows the KEM description described in section 2.4.

Name	In/Out	Wiring
ap_clk	IN	STD_LOGIC
ap_rst	IN	STD_LOGIC
ap_start	IN	STD_LOGIC
ap_done	OUT	STD_LOGIC
ap_idle	OUT	STD_LOGIC
ap_ready	OUT	STD_LOGIC
ss_V_address0	OUT	STD_LOGIC_VECTOR (2 downto 0)
ss_V_ce0	OUT	STD_LOGIC
ss_V_we0	OUT	STD_LOGIC
ss_V_d0	OUT	STD_LOGIC_VECTOR (63 downto 0)
ss_V_q0	IN	STD_LOGIC_VECTOR (63 downto 0)
ct_V_address0	OUT	STD_LOGIC_VECTOR (12 downto 0)
ct_V_ce0	OUT	STD_LOGIC
ct_V_q0	IN	STD_LOGIC_VECTOR (7 downto 0)
ct_V_address1	OUT	STD_LOGIC_VECTOR (12 downto 0)
ct_V_ce1	OUT	STD_LOGIC
ct_V_q1	IN	STD_LOGIC_VECTOR (7 downto 0)
sk_V_address0	OUT	STD_LOGIC_VECTOR (8 downto 0)
sk_V_ce0	OUT	STD_LOGIC
sk_V_q0	IN	STD_LOGIC_VECTOR (63 downto 0)
ap_return	OUT	STD_LOGIC_VECTOR (31 downto 0)

Table 3.1: Wiring table for the top module for decapsulation in NIST HQC submission [1].

3.1.4 Data transfer PC $\leftrightarrow$ FPGA

To transfer data, the UART protocol was chosen as the way of communicating between the PC and FPGA while running the different tests. This was implemented by Ben Marshall using a simple 8N1 UART setup [5]. This was chosen to make it easy to use on many different FPGAs without much friction, only requiring changes to the constraint file used to connect the receive (rx) and transmit (tx) ports. It was also chosen because of the testing done on Artix-7 FPGA boards.

UART was used to send and receive data from the Husky, which had the role of being the connector between the PC and FPGA. Between the PC and Husky, a USB-C cable was used to send data, also via UART. To send and receive data from the FPGA the 20-pin connector on the CW305 was connected to the 20-pin connector on the Husky where tio1 and tio2 were used as shown in 3.2. To send the data from the PC NewAEs Python API was used to send the data via UART to the Husky which routed the data to the FPGA. The ChipWhisperer API is open-sourced and makes it easier to send data with implemented function calls enabling communications without having to do manual work on the computer side by doing serial communication via ports.

Figure 3.2: Data transfer using UART to connect PC to FPGA via Husky.

3.1.5 Final state machine implementations

When designing the hardware wrapper FSMs were crucial in the process to control the dataflow. Moore machines were chosen as the design to follow to make the wrapper more stable and avoid timing issues such as speeding up the process by for example doing computation at the same time as loading data which can make invalid data being used. FSMs can be found in all modules of the hardware implementation where they were used to in quite a simple manner often having the structure of the example in section 2.6.2. Another reason why Moore machines were chosen was the need to avoid polluting the power traces from the decapsulation where it was important that only the decapsulation was running.

3.1.6 Wrapper implementation

In the NIST HQC Hardware submission a key-gen, encapsulation and decapsulation module was provided. This was used to perform the computations needed, but these did not contain any top module to receive or store data, send the output or start the process. This was implemented in Verilog and SystemVerilog to be able to send and receive data from the CW305 board. This consisted of a few different modules, shown below in figure 3.3

The decaps module expected two different formats of the data being provided as can be seen in table 3.1, where ciphertext is expected as bytes and the secret-key is expected as 64-bit vectors. The data-loader therefore had to store the

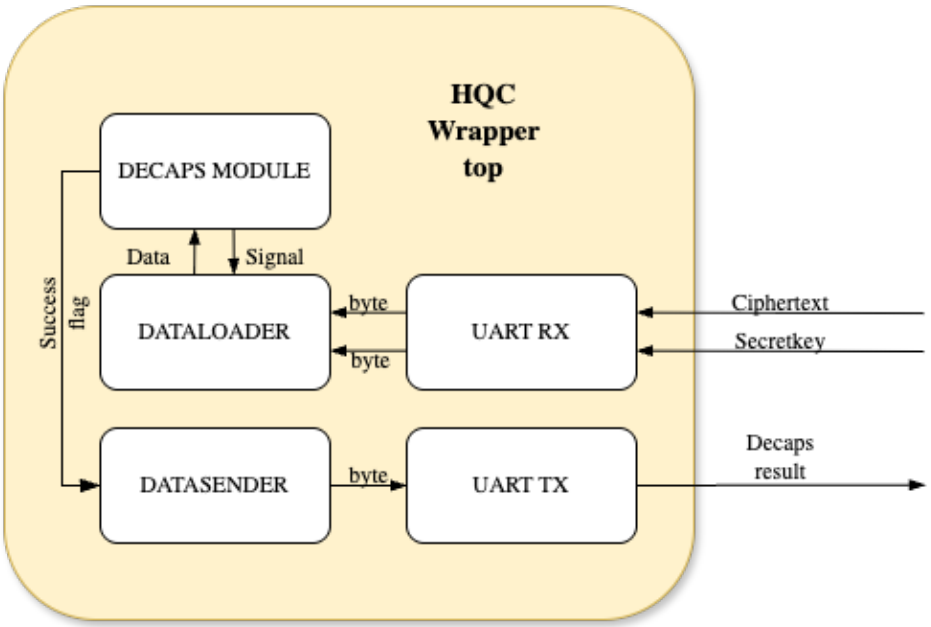

Figure 3.3: Overview of top wrapper and of the dataflow

ciphertexts and the secret-key differently. The ciphertext was stored as it was sent byte-wise in a large register that stored bytes specified in the PQC HQC docs [1], in this case 4481. The secret-key was supposed to be stored as a 64-bit array. This required the data-loader to transform data from bytes (8-bit) to 64-bit words. This was done using little-endian storage discussed in section 2.6.3 which was the same endianness used in the decapsulation module. In the documentation of HQC it is specified that the secret-key should be 2289 bytes [1]. Because this cannot be equally divided by 8, the secret-key was padded with 7 bytes of zeroes to fill the last 64-bit block which resulted in the bytes being sent to the CW305 become 2296 bytes. Both the ciphertext and secret-key were stored in registers that were slaved to the decapsulation module.

The data-sender was used mostly for debug purposes to determine if the decapsulation failed or succeeded which made it possible to spot if any data was invalid. The data-sender was only a wrapper for the UART tx module. It sent back single bytes of information being either a success flag, 'Y' (0x59), or a failure flag, 'N' (0x4E), to indicate a decapsulation outcome after the decapsulation had been finished.

A trigger was also implemented by having a wire connected to decapsulation start wire to signal when the power traces should be captured. This was later connected to the T14 pin located in the 20-pin connector, which was routed to the tio4 pin on the Husky. This made it easy to capture the decapsulation process without having to determine when it started by analysing the trace.

To be able to run the wrapper on an FPGA, a bitstream was needed. This was created with the Xilinx Vivado 2025.1 platform which had the ability to transform

the hardware description language into an IC-structure that could be programmed onto the FPGA using synthesis comparable to a compiler. When synthesizing the project, all modules of the decapsulation were selected, using the performance specific decapsulation module and its dependencies. The top wrapper built over it was also selected and synthesized. This made the cw305_top.v the top module of our project as can be seen in `https://github.com/Equasa/HQC-FPGA-TOP`

3.1.7 Running the decapsulation module

While the decapsulation module was running, the top wrapper was designed to generate as little noise as possible. This was done by loading all of the data before starting the process instead of continuous loading, where the latter could result in leakage not created from the module. A FSM was used to control that only the decapsulation module was running when getting traces. It was started by signalling ap_start found in table 3.1 and then was presented with data from the master–slave relationship between the data-loader module and decapsulation module, where the latter was the master. The FSM waited until it received a signal from the wire ap_done, and then continued to the next state.

3.1.8 FPGA settings

The FPGA was set to receive its clock not from the internal PLL, but instead from the PLL inside the Husky. This signal was received through the 20-pin connector used to connect the Husky to the CW305. The TIO_HS2 pin was used for the clock because of the high speed capabilities. This was done to make the ADC and FPGA run at the same frequency, which made it easier to capture the traces. The CW305 was run using a 50 MHz clock frequency.

3.2 Software overview

To communicate with the hardware, a script using the ChipWhisperer open-sourced Python API was used. This made it easier to configure the different settings that needed to be matched with the hardware implementation running on the CW305 board. There was also a need for training data to later be used for the machine-learning models. This was done by using the standard KEM encapsulation and a modified KEM encapsulation function. When power traces had been collected a analysis script was used to acquire deeper knowledge about which data points should be used for the different machine-learning models.

3.2.1 Data generation

The objective of the experiment was to establish if there was a difference between $m = 0$ and $m \neq 0$. Therefore, the normal key-gen from the software implementation of NIST PQC HQC was used to create the key-pair. The encapsulation function was also used to generate ciphertext with $m = rand()$ for half of the key-pairs. For the other half, a modified key-gen function was used where the message was pinned to zero, $m = 0$. To get a good training set, 20k samples were created

where 10k were from each group. Every sample used a different key to minimize the risk of the training data being biased by a specific key pair A script was created to generate these and save them in a binary file, saving the ciphertext, secret-key, shared secret-key, and public-key. This was done to make the data more versatile and debug friendly. If more data had been generated, the decision to save only the secret-key and ciphertext would have been taken, saving up to 25% memory. To access this data, a data-loader was created to process the binary into bytes in memory.

3.2.2 Power-trace generation

The power-traces were generated by using the training set described in section 3.2.1. These were then sent to the CW305 using the Python API with serial tx on the Husky to send data with the 'write' function with the 'write' function in the API via the SimpleSerial class made for serial communication, in this case the UART protocol.

Before sending data, the Husky was set up with settings for capturing, receiving and sending data. The settings can be seen in table 3.2. The tio1 pin was in contrast to the hardware side, set to be rx instead of tx. This was done to match tx to rx and rx to tx between the devices so data could be sent properly. The clock frequency and baud used was the same as on the CW305 board to make the communication protocol work as expected. To capture traces with good quality, the trigger setting was set to start the capturing when tio4 went high which was the same behaviour created from the hardware wrapper.

The decibel gain used to amplify the already amplified low-noise signal was set to 14 by fine tuning the gain to use the highest gain possible without having the power-traces being clipped. This process was done with a script testing different decibel gains until 14 dB was found. The ADC was also set to capture 60k samples with a ADC multiplier of four which meant that the 15k first clock cycles were captured 4 times for each cycle.

Each of the 20k generated ciphertext-secret-key pairs were used as input data to the wrapper, running sequentially until all pairs had been decapsulated. To mitigate the risk of heat differences between the groups, each sample was alternated between the groups so that heat differences did not arise from the grouping. The data was saved in batches with the decryption success/failure flag and the label ($m = 0$ or $m \neq 0$). The software implementation waited for the success flag before resetting and starting a new key pair decapsulation process by sending new data and arming the trigger for the next capture. This whole process can be seen in figure 3.4

3.2.3 Data analysis

To determine if there was any statistically significant difference between the groups, the Welch t-test was used described in section 2.5.1 where a t-value greater than 4.5 was used as a threshold to determine if there was side-channel leakage or if it was not statistically probable that there was an actual mean difference between the groups. The reason for choosing this t-value can be found in section 2.5.1. This

Setting	Value
clkgen_src	system
clkgen_freq	50 Mhz
hs2	clkgen
tio1	serial_rx
tio2	serial_tx
triggers	tio4
adc samples	40000
adc offset	0
adc_mul	4
db gain	14
baud	115200

Table 3.2: Husky settings when running capture script

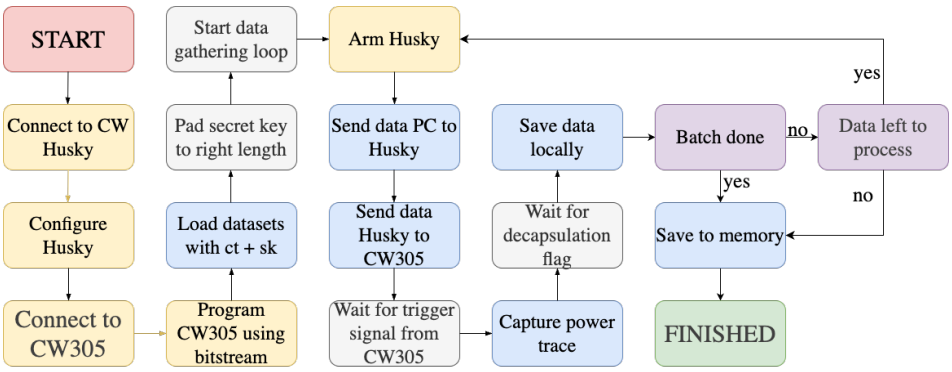

Figure 3.4: Overview of software implementation and dataflow. Red and green symbolises the start and stop while yellow is settings, blue is for data, purple is logic and grey is other.

was done on the raw power traces pointwise to determine if there was an actual statistical difference at any single point or if too little information was leaking through the power consumption traces.

Other tests, such as mean difference, were used to obtain a better understanding of where data was leaking to try to understand when different steps in the decapsulation process was happening. This was done by taking the mean power trace of each group and calculating the difference between groups. A pointwise standard deviation was also calculated to get a more robust understanding of the t-test and the quality of the data.

3.2.4 Data pruning

When pruning the data given, the t-value and p-values were used. A mask was created to filter out all data points where the p-value was $p_{value} \geq 0.01$, to remove all data points that did not reject the null hypothesis of $H_0 : \bar{X} = \bar{Y}$. This was done to remove all data where it was not statistically probable that there was an actual mean difference which meant that the data point was not as usable in the classification problem.

3.2.5 Machine-learning models

A couple of machine-learning models were used to try and create a distinguishing oracle to distinguish between the classes $m = 0$ and $m = rand()$.

The first one tested was the logistic regression model using gradient descent with SAGA optimization provided in the sklearn.linear_model package. This was combined with regularization to try to mitigate overfitting using the $L2$ norm giving the logistic regressor the parameters described in table 3.3.

Setting	Value
Optimizer	SAGA
Max_iter	500
C	0.005
Penalty	$L2$

Table 3.3: Husky settings when running capture script

The whole dataset, containing 20k ciphertext–power-trace samples was used. The dataset was split into a training set and a validation set being split at 16k and 4k. After splitting the two sets normalization was done on both separately to avoid leaking information about the validation set into the training set.

The second model that was tested was a multilayer perceptron described in section 2.7.3. The design of the MLP model can be seen in table 3.4 and used optimizers and loss functions described in section 3.5. This was trained on a few different versions of the traces where the data pruning was used and also the raw data only having normalization performed on it was used like when using the logistic regressor. The model described in table 3.4 was designed by having fully

connected layers using ReLu as the non-linear function, θ. The output of the MLP is two neurons where each neuron represents the probability of the power trace corresponding to a given class. This is later used with an argmax function to get the most probable option which is either 0 or 1 in this case. The hyperparameters used were set by first using standard settings from the paper used to attack SHAKE [14] which was then optimized for our specific traces.

A few versions of convolutional neural networks were also tested where the best performing one is described in table 3.6. Using the same optimizers, loss functions and hyperparameters as the MLP. The only data pruning used for the CNNs input data was normalization making the important data stand out more, lowering the amplitude making the subtle differences in each time point stand out more. The model from [14] was modified to have an output of 2 instead of 1 so that the CrossEntropyLoss function could be used. The power trace samples were cut to only use the first 40k sample points in the power trace.

Layer	**Type**	**Output Size / Details**
Input	-	(batch, power_trace_length)
1	Linear	power_trace_length $\rightarrow$ 512
	ReLU	–
2	Linear	512 $\rightarrow$ 256
	ReLU	–
3	Linear	256 $\rightarrow$ 128
	ReLU	–
4	Linear	128 $\rightarrow$ 64
	ReLU	–
Output	Linear	64 $\rightarrow$ 2
	Loss function	CrossEntropyLoss

Table 3.4: MLP architecture used in the experiments

Setting	**Value**
Loss function	CrossEntropyLoss
Optimizer	Adam
Learning rate	5×10^{-4}
Weight decay	1×10^{-4}
Epochs	100
Batch size	64

Table 3.5: Training settings for the MLP model

Layer	Type	Output Size / Details
Input	-	(batch, 1, input_len)
1	Conv1D	$1 \to 16$, kernel size 7, padding 3
	ReLU	–
2	Conv1D	$16 \to 32$, kernel size 7, padding 3
	ReLU	–
3	Conv1D	$32 \to 32$, kernel size 7, padding 3
	ReLU	–
	AvgPool1D	kernel size 4 (length becomes input_len/4)
Flatten	-	(batch, $32 \cdot$ input_len/4)
FC1	Linear	$(32 \cdot \text{input\_len}/4) \to 64$
	ReLU	–
Output	Linear	$64 \to 2$
	Loss function	CrossEntropyLoss

Table 3.6: CNN architecture used in the experiments

When training the different models, a MacBook Air laptop with a 10-core M3 CPU, 10-core built-in GPU and 24 GB of RAM was used running each model at different times training the models.

3.3 Secret-key full recovery attack on HQC

The attack against the NIST HQC FPGA implementation was simulated using the code from [2], where the maximum validation accuracy of the best model was used as the accuracy of the simulated oracle used to distinguish between decoding success and failure. Error templates were generated, the best group was identified, and this group, combined with the simulated oracle, was then used to recover the secret-key described in section 2.4.1.

To evaluate how many decapsulation power traces are required to achieve a secret-key-recovery probability greater than 50% using the OT-PCA attack described in Section 2.4.1, two approaches were used. The first approach was straightforward: the simulated success rates reported in [2] were directly consulted. The second approach replicated those results by running the authors' publicly provided simulation code.

Chapter 4
Results

4.1 Traces

In figure 4.1, the mean trace from the two groups and the mean difference can be seen. The difference between the two groups is subtle, where the maximum mean difference between the two groups was 0.004 which is only 1% of the maximum trace amplitude. However, compared to the absolute mean of the mean-difference curve, 0.004 is 40 times larger than the absolute mean of the mean difference (0.000106).

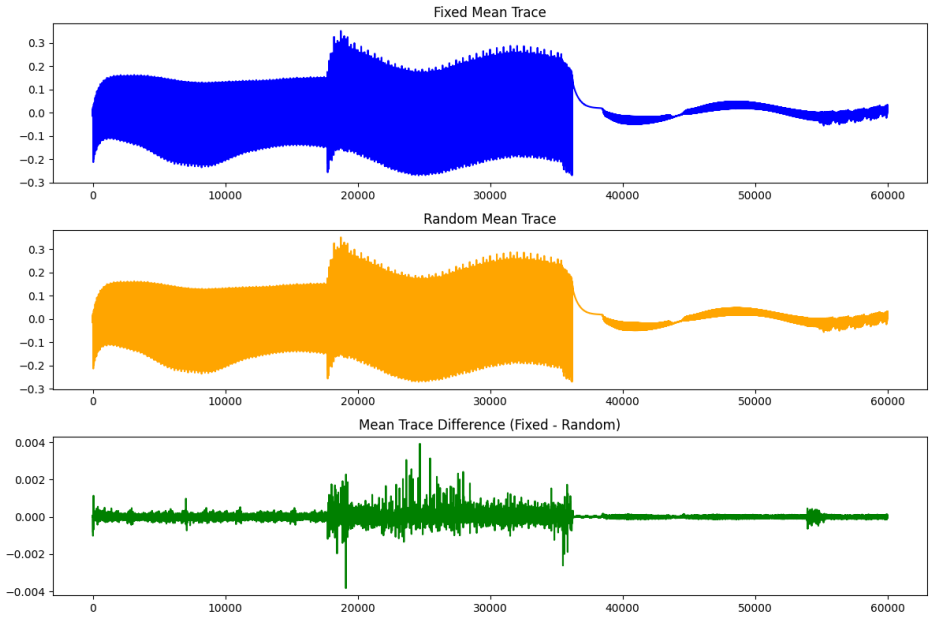

Figure 4.1: The mean traces from each group and the mean difference between the groups.

It can be seen in figure 4.2 that most significant mean differences appear around samples 17–37 thousand where there is a lot of spikes in mean difference.

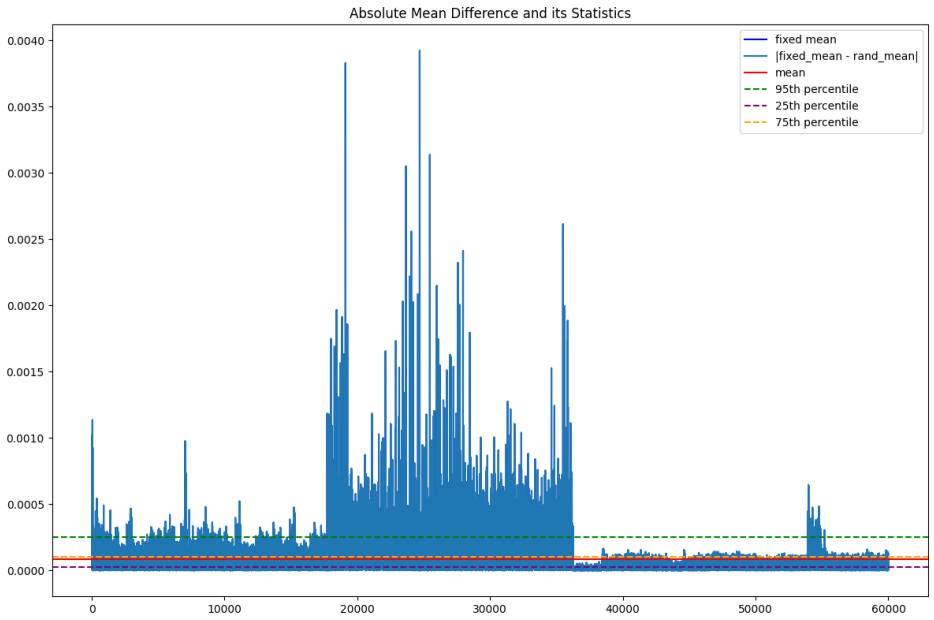

Figure 4.2: The absolute mean difference of groups $m = 0$ and $m = rand()$. Plotted together with percentiles for clarity of the distribution.

The t-test in figure 4.4 shows leakage across almost the entire power trace, where nearly 10% of all points exhibit a significant mean difference. The region between samples 17–35 thousand contains more concentrated high t-test values, indicating stronger leakage in that part of the trace. There are three spikes corresponding to the mean-difference peaks in the power consumption seen in figure 4.2 and figure 4.3. These spikes in the t-test are correlated with the SHAKE algorithm being used in the KEM three times to generate pseudorandom numbers as can be seen in Listing A.

4.2 Machine-learning models

When training the different models described in section 3.2.5, a range of accuracies was recorded. These can be seen in table 4.1 where the tiny CNN described in table 3.6 was the best-performing convolutional neural network found.

4.3 Simulations

When running simulations, a group of 8 error patterns was generated together with error templates created for an oracle with 97.2% accuracy. This resulted in a simulated probability of recovering the secret-key, shown in table 4.2.

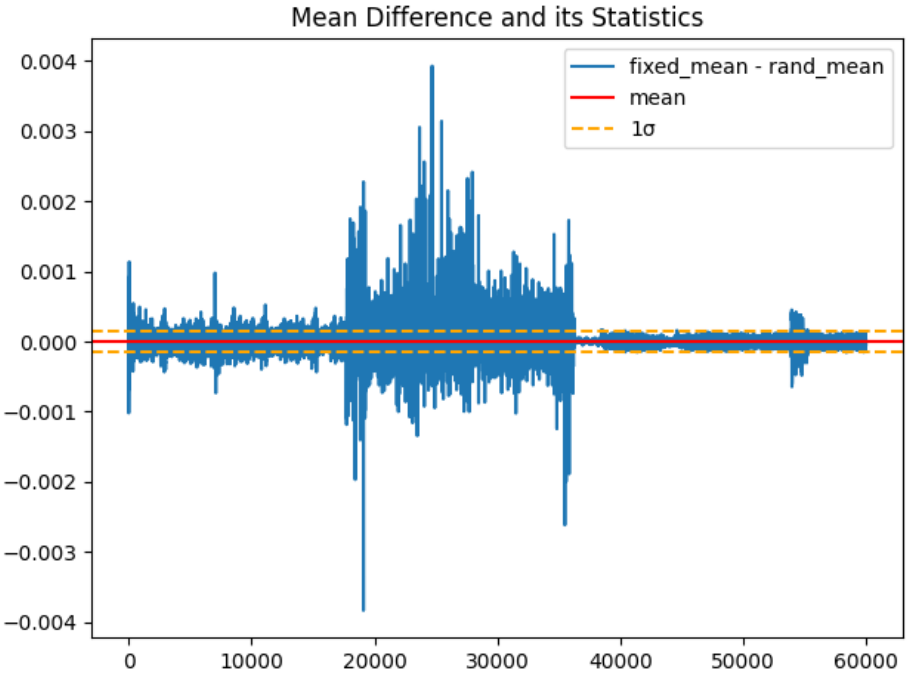

Figure 4.3: The mean difference of groups $m = 0$ and $m = rand()$. The mean of the mean difference is zero and one standard deviation has been plotted to show the spread.

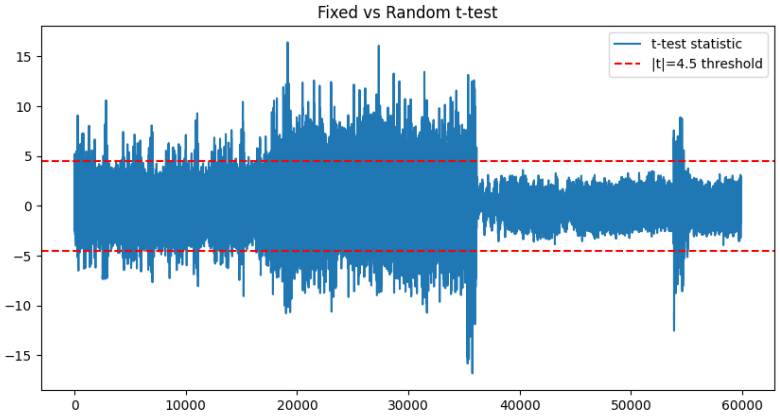

Figure 4.4: The t-test between the groups $m = 0$ vs $m = rand()$ to see if pointwise difference in mean exists in the power traces. The threshold was set to 4.5. 4143 of 60k points (6.905%) had $|t| \geq 4.5$

Model	Accuracy
Logistic regression	0.854
Multilayer perceptron	0.945
Tiny CNN	0.972

Table 4.1: Validation accuracy for logistic regression, MLP and CNN where Tiny CNN can be found in table 3.6. The validation set used 4k samples, with equal amount having $m = 0$ and $m = rand()$

Oracle accuracy	Number of error patterns	Probability of recovery
97.2%	8	72.3%

Table 4.2: Probability of recovering a secret-key with the use of 8 number of patterns using 736 decapsulation power traces with a oracle having 97.2% accuracy.

4.4 Implementation results

When running the wrapper together with the software implementation to gather training data, gathering times were recorded to document the performance of the wrapper. These can be seen in table 4.3 where the the 20k samples were chosen as a benchmark because that was how many traces were used during training.

Parameter	Value
BAUD rate	115200 bit/s
Bytes	6777 B
Avg. transfer time / sample	0.59 s
Avg. sample time	0.66 s
Time for 20k samples	3.67 h

Table 4.3: Measured performance of the wrapper during data collection.

Chapter 5
Discussion

Because of the situation where an FPGA implementation is being evaluated, the thesis can be used both by the creators of the implementation and by attackers trying to acquire sensitive information. This can be done by using the leakage information and the framework either to develop countermeasures or to exploit the vulnerability to obtain information about the secret-keys.

5.1 Design improvements

It can be seen in figure 4.4 that there is leakage from many parts of the decapsulation process. This means that it is not only the SHAKE algorithm leaking information described in [14] but also that the decryption part is leaking information. This likely comes from the decoder leaking information about the message, $m = 0$ vs $m = rand()$. This is expected because of the close connection between the message and the decoder where it is directly interacting with the message.

This means that SCA countermeasures are needed not only in the SHAKE part of the algorithm but also in the decoding components, Shortened Reed-Solomon and Modified Reed-Muller decoders. This could potentially be achieved with the help of masking, hiding and shuffling to make the signal more noisy, moving around different operations and computing on intermediate values as done in masking. This is something that should be studied further to see if the different modules can become resistant against SCA with the usage of power traces.

5.2 Attack surface

The t-test in figure 4.4 is showing that at almost one tenth of all data points in the power trace from the decapsulation module are leaking data when comparing the message groups $m = 0$ and $m = rand()$. This was expected because of the computational differences from $m = 0$ and $m = rand()$ in the SHAKE algorithm [14]. It can be seen in figure 4.4 that we have three of these peeks which is exactly what is expected if looking at listing A.

What was not expected is the leakage from the decryption part between 0–17k in the t-test where we have leakage, is real leakage that can be exploited. This comes from the part of the decapsulation where the message is decoded with the usage of a concatenated Shortened Reed-Solomon and a Modified Reed decoder.

This is expected because of it being deeply connected to the message, m. The leakage from this part can also be used to improve the oracle trying to determine the decoding result. The best model achieved managed to have a validation accuracy of 97.2% which is high enough to be used to perform the attack described in section 2.4.1 presented by Guo and Dong. This implies that the NIST HQC FPGA implementation is, with high probability, not secure against side-channel attacks using power traces from decapsulation. It can also be seen that even if no leakage is present from the part where SHAKE is performed an oracle, although not as good, can still be created enabling the possibility of an attack against the implementation even though the SHAKE implementation is masked. There is also a strong leakage point after around 55k sample points. This is probably from some process after the decapsulation process has ended. This could be the module reseting itself after computation is done or writing to the shared secret memory outside of the module. Because of the uncertainty from what this process was it was not included when training the models in order to avoid including leakage not supposed to be observed when performing the attack.

The simulated values from the tests where an oracle with an accuracy of 97.2% was used it was determined that 736 decapsulation observations of power traces were needed to acquire the secret-key with a probability of achieving the result 72.3%. This shows that it is with high probability possible to attack the NIST HQC FPGA implementation using the oracle and offline template attack described in [2]. This matches the simulated values in [2], meaning that our simulations are correct.

It can also be seen in table 4.1 that the model being used is of high importance when trying to create a classification model. Some configurations of the CNNs were not able to learn the patterns from the power traces. This can for example be because of average pooling destroying the very small differences between the groups making it useless. The bad accuracy could also stem from the model being too large, making it train on noise, learning bad patterns and therefore not achieving high accuracy in the training set nor in the validation set.

A real attack using the specifically created ciphertext would be the next to study to see if it possible to establish a oracle able to determine the difference between decoding failures and successes based on the power traces. If the attack could be realized with a high success rate, the implementation would not be suitable for real-world usage because of the vulnerability creating a attack surface that can be exploited to decapsulate arbitrary ciphertext getting the shared-secret used to send other packets more or less being to read all traffic sent if intercepted.

It would also be interesting to see if electromagnetic radiation could be used to create a distinguishing oracle. This would have great implications because of possibility of using it from a distance different from the power traces needing to be connected between the power source and the FPGA running the decapsulation.

5.3 Wrapper improvements

The wrapper was designed with the intention of being usable on a wide range of FPGAs. Therefore it was decided that the UART protocol would be used to

transmit and receive data between the PC and FPGA instead of the USB protocol designed on the ChipWhisperer platform discussed in section 3.1.1. This used a baud rate of 115200 bits/second, making the transfer of a whole training sample to the FPGA take 0.59 seconds. This made data transfer account for about 90% of the total time required to send data and decapsulate it on the FPGA. This made data gathering an extensive process and rendered the implementation inefficient. This could have been improved by using the USB interface with a 480 Mb/s transfer speed [3]. This would have been necessary if for example 10^6 samples were needed making the data gathering take around 183 hours. This makes the wrapper implementation unsuitable for real-world usage outside of testing which it was not designed for.

An interesting and necessary next step to being able to use the wrapper in real-time systems would be to implement a protocol with higher transfer speed than UART for a general FPGA making transfer time shorter and therefore reducing transfer time which would make it viable in real-time systems that need to decapsulate many requests in comparison to this wrapper only used to produce training data for the machine-learning models.

5.4 Future work

To guarantee that an actual attack can be performed, further testing is required. This includes generating malicious ciphertexts, capturing the power traces, and creating a new oracle for the specific malicious ciphertexts. Then a script to perform the final steps described in [2] to determine which bits are 1 needs to be created. “A study to test the different components of the NIST HQC FPGA implementation may also be needed to see if they are working as they are supposed to do. These tests and the attack could be carried out relatively easily by using the framework created in the thesis and could be expanded to work for other components.

Chapter 6
Conclusion

In this thesis, a wrapper for the NIST HQC FPGA decapsulation implementation has been implemented using UART to transfer data from the PC to the FPGA. This has been run on a ChipWhisperer CW305 with an Artix-7 FPGA board communicating with the PC via a ChipWhisperer Husky. A software implementation has been implemented using the ChipWhisperer API in Python making it easy to send a ciphertext, secret-key pair to the FPGA and capturing the power trace generated from the decapsulation.

An oracle able to distinguish if $m = 0$ or $m = rand()$ was sent with an accuracy of 97.2%. The oracle was constructed from 20k generated power traces each from a different secret-key, achieving the accuracy using a convolutional neural network. This proved that the implementation was leaking heavily.

Analysis also showed that there is leakage not only from the SHAKE algorithm discussed in [14] but also from the decryption part of the module where the only probable cause is from the decoder.

Lastly, the attack described in [2] was simulated giving a probability of recovering the correct secret-key of 72.3% from only 736 power traces collected from decapsulations.

References

[1] Carlos Aguilar Melchor *et al.*, *Hamming Quasi-Cyclic (HQC): Fourth-round version, Updated version 19/02/2025*, PQC-HQC.org, February 19, 2025, 52 pp. [Online]. Available: `https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf`

[2] Haiyue Dong and Qian Guo. *OT-PCA: New Key-Recovery Plaintext-Checking Oracle Based Side-Channel Attacks on HQC with Offline Templates.* IACR Transactions on Cryptographic Hardware and Embedded Systems, Vol. 2025, No. 1, pp. 251–274, 2025. DOI: 10.46586/tches.v2025.i1.251-274.

[3] NewAE *CW305 Artix FPGA Target* [Online]. Available: `https://chipwhisperer.readthedocs.io/en/latest/Targets/CW305%20Artix%20FPGA.html`

[4] NewAE *ChipWhisperer Husky* [Online]. Available: `https://chipwhisperer.readthedocs.io/en/latest/Capture/ChipWhisperer-Husky.html`

[5] Ben Marshall. *UART Implementation* [Online]. Available: `https://github.com/ben-marshall/uart`

[6] Thomas Schneider and Amir Moradi. *Leakage Assessment Methodology. Journal of Cryptographic Engineering*, 2016 [Online] Available: `https://eprint.iacr.org/2015/207.pdf`.

[7] François-Xavier Standaert. *Introduction to Side-Channel Attacks.* Lecture notes, UCLouvain, 2004, 52 pp. [Online]. Available: `https://perso.uclouvain.be/fstandae/PUBLIS/42.pdf`

[8] "Endianness" Wikipedia. [Online]. Available: `https://en.wikipedia.org/wiki/Endianness` (accessed Nov. 3, 2025).

[9] Umakanta Nanda and Sushant Kumar Pattnaik. *Universal Asynchronous Receiver and Transmitter (UART).* In *Proceedings of the 2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS-2016)*, Coimbatore, India, Jan. 22–23, 2016. DOI: 10.1109/ICACCS.2016.7586376. [Online]. Available: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7586376`

[10] Carlos Aguilar Melchor, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. *Efficient Encryption from Random Quasi-Cyclic Codes.* arXiv preprint *arXiv:1612.05572 [cs.CR]*, submitted on Dec. 16, 2016. DOI: 10.48550/arXiv.1612.05572. [Online]. Available: `https://arxiv.org/abs/1612.05572`.

[11] Muhammed Fahri Ünlerşen. *Field Programmable Gate Array (FPGA).* In *Programmable Smart Microcontroller Cards*, Chapter 12, pp. 179–207, November 2021. The International Society for Research in Education and Science (ISRES), 2021. [Online]. Available: `https://www.isres.org/programmable-smart-microcontroller-cards.html`

[12] National Instruments Corp., "FPGA Fundamentals: Basics of Field-Programmable Gate Arrays," Feb. 14, 2025. [Online]. Available: `https://www.ni.com/en/shop/electronic-test-instrumentation/add-ons-for-electronic-test-and-instrumentation/what-is-labview-fpga-module/fpga-fundamentals.html`

[13] Guillaume Goy, Alexandre Loiseau, and Philippe Gaborit. *A New Key Recovery Side-Channel Attack on HQC with Chosen Ciphertext.* In *Proceedings of the 13th International Conference on Post-Quantum Cryptography (PQCrypto 2022)*, September 2022. [Online]. Available: `https://cea.hal.science/cea-03823234/document`

[14] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. *Curse of Re-encryption: A Generic Power/EM Analysis on Post-Quantum KEMs.* IACR Transactions on Cryptographic Hardware and Embedded Systems, Vol. 2022, No. 1, pp. 296–322, 2022. [Online]. Available: `https://tches.iacr.org/index.php/TCHES/article/view/9298`

[15] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli & Cécile Dumas. *Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database.* IACR Cryptology ePrint Archive, Report 2018/053, 2018. [Online]. Available: `https://eprint.iacr.org/2018/053.pdf` (accessed Nov. 17, 2025).

[16] S. Albawi, T. A. Mohammed, and S. Al-Zawi. *Understanding of Convolutional Neural Network (CNN): A Review.* In *2017 International Conference on Engineering and Technology (ICET)*, Antalya, Turkey, Aug. 21–23, 2017. DOI: 10.1109/ICEngTechnol.2017.8308186. [Online]. Available: `https://ieeexplore.ieee.org/document/8308186`.

[17] NewAE *CW305 Artix FPGA target datasheet.* [Online]. Available:`https://media.newae.com/datasheets/NAE-CW305_datasheet.pdf`

[18] Thomas De Cnudde, Maik Ender, and Amir Moradi. *Hardware Masking, Revisited. Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, vol. 2018, no. 2, pp. 123–148, 2018. DOI: 10.13154/tches.v2018.i2.123-148 . [Online]. Available: `https://tches.iacr.org/index.php/TCHES/article/view/877/829`

[19] Salvador García, Julián Luengo, and Francisco Herrera. *Data Preprocessing in Data Mining.* Series: Intelligent Systems Reference Library (ISRL, Vol. 72). Springer International Publishing Switzerland, 2015. DOI: 10.1007/978-3-319-10247-4. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-319-10247-4.pdf

[20] P. Thomadakis and A. Argyriou, "Reed-Solomon and Concatenated Codes with Applications in Space Communication," University of Thessaly, Aug. 13 2016. [Online]. Available: https://arxiv.org/pdf/1608.03961

[21] U. P. Singh, "Error Detection and Correction Using Reed-Solomon Codes," Central University of Jammu, Mar. 2013. [Online]. Available: https://www.researchgate.net/publication/305641094_Error_Detection_and_Correction_Using_Reed_Solomon_Codes/link/5aec83e7aca2727bc00483d1/download?_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uIn19

[22] Shu Lin and Daniel J. Costello. *Error Control Coding*, 2nd ed. Prentice Hall, Englewood Cliffs, NJ, 2004.

[23] Linus Mainka and Kostas Papagiannopoulos. *Combined Masking and Shuffling for Side-Channel Secure Ascon on RISC-V.* IACR Cryptology ePrint Archive, Report 2025/564. [Online]. Available: https://eprint.iacr.org/2025/564.pdf.

Appendix A
Extra material

```
[language=C,breaklines=true,breakatwhitespace=true,
caption={Decapsulation function of the HQC\_KEM IND\_CAA2 scheme (internal version)}]
/**
 * @brief Decapsulation function of the HQC_KEM IND_CAA2 scheme, internal version
 *
 * The internal version has a large signature that includes many arrays that can be shared by multiple
 * functions to reduce hardware usage, as crypto_kem_all_functions_hls does. In that wrapper, arrays
 * are only defined once and shared between keygen, encaps, and decaps internal functions, giving a
 * realistic estimation of implementing the three operations together in hardware.
 *
 * The HLS version (standalone) defines its own arrays and calls this internal function. It yields
 * realistic cost for standalone hardware implementation.
 *
 * The API version wraps the HLS function but exposes standard C data types rather than HLS-specific types.
 *
 * @param[out] ss  Shared secret
 * @param[in]  ct  Ciphertext
 * @param[in]  sk  Secret key
 * @param[inout] state_seedexpander Internal seed expander state
 * @param[inout] Multiple arrays used internally and shared across HLS modules
 *
 * @returns 0 if decapsulation is successful, 1 otherwise
 */
int crypto_kem_dec_internal(
    ap_uint64 ss[SHARED_SECRET_64BIT_SIZE],
    ap_uint8  ct[CRYPTO_CIPHERTEXTBYTESIZE],
```

```
    ap_uint64 sk[KEY_MAX_64BIT_SIZE],
    ap_uint64 state_seedexpander[PRNG_STATE_64BIT_SIZE],

    // Temporary arrays used by KEM
    ap_uint8  mc[VEC_K_BYTESIZE + VEC_N_BYTESIZE +
    ↪ VEC_N1N2_BYTESIZE],
    ap_uint64 theta[SHAKE256_512_64BIT_SIZE],
    ap_uint64 d[SHAKE256_512_64BIT_SIZE],

    // Temporary arrays used by HQC
    vector_byte_type  store_A[VEC_N_MULTIPLIERWORDSIZE *
    ↪ MULTIPLIER_BYTESIZE],
    vector_byte_type  store_B[VEC_N_BYTESIZE],
    vector_byte_type  store_C[VEC_K_BYTESIZE],
    vector_word_type
    ↪ h_s[VEC_N_VECTORWORDSIZE_FOR_MULTIPLIER + 1],
    vector_index_type random_vector_A[PARAM_OMEGA_R],
    vector_index_type random_vector_B[PARAM_OMEGA_R],
    ap_uint64         store_seeds[KEM_SEED_64BIT_SIZE]
) {
    static ap_uint8 uv2[VEC_K_BYTESIZE + VEC_N_BYTESIZE +
    ↪ VEC_N1N2_BYTESIZE];

    uint8_t i;

    bool cmp_d_d2 = false;
    bool cmp_u_u2 = false;
    bool cmp_v_v2 = false;
    bool result;

    const ap_uint64 masks[2] = { 0xffffffffffffffffULL, 0x0 };

    // Retrieve u, v, and d from ciphertext
    hqc_ciphertext_from_string_64(
        &mc[VEC_K_BYTESIZE],
        &mc[VEC_K_BYTESIZE + VEC_N_BYTESIZE],
        d,
        ct
    );

    // Decrypt
    hqc_pke_decrypt(
        mc,
        &mc[VEC_K_BYTESIZE],
        &mc[VEC_K_BYTESIZE + VEC_N_BYTESIZE],
        sk,
        state_seedexpander,
        store_A,
        store_B,
        h_s,
        random_vector_A,
```

```
        store_seeds
    );

    vect_copyresize(store_C, VEC_K_BYTESIZE, mc,
    ↪ VEC_K_BYTESIZE);

    // Compute theta
    shake256_512_ds(theta, mc, VEC_K_BYTESIZE, G_FCT_DOMAIN);

    // Encrypt m'
    hqc_pke_encrypt(
        &uv2[VEC_K_BYTESIZE],
        &uv2[VEC_K_BYTESIZE + VEC_N_BYTESIZE],
        store_C,
        theta,
        &sk[KEM_SEED_64BIT_SIZE],
        state_seedexpander,
        store_A,
        store_B,
        h_s,
        random_vector_A,
        random_vector_B,
        store_seeds
    );

    // Compute d'
    shake256_512_ds(theta, mc, VEC_K_BYTESIZE, H_FCT_DOMAIN);

    // Compute shared secret
    shake256_512_ds(
        ss,
        mc,
        VEC_K_BYTESIZE + VEC_N_BYTESIZE + VEC_N1N2_BYTESIZE,
        K_FCT_DOMAIN
    );

    // Compare u,u' ; v,v' ; d,d'
    cmp_u_u2 = vect_compare(
        &mc[VEC_K_BYTESIZE],
        &uv2[VEC_K_BYTESIZE],
        VEC_N_BYTESIZE
    );

    cmp_v_v2 = vect_compare(
        &mc[VEC_K_BYTESIZE + VEC_N_BYTESIZE],
        &uv2[VEC_K_BYTESIZE + VEC_N_BYTESIZE],
        VEC_N1N2_BYTESIZE
    );

    cmp_d_d2 = vect_compare_64(d, theta,
    ↪ SHAKE256_512_64BIT_SIZE);
```

```
123
124     // Abort if mismatch
125     result = cmp_u_u2 | cmp_v_v2 | cmp_d_d2;
126
127 control_loop:
128     for (i = 0; i < SHARED_SECRET_64BIT_SIZE; i++) {
129         ss[i] = masks[result] & ss[i];
130     }
131
132     return result & 1;
133 }
```

```
1  import time, os
2  import chipwhisperer as cw
3  import numpy as np
4  import json
5  from data_loader import load_dataset
6
7  def send_blob(tgt: cw.targets.SimpleSerial, data, chunk=512,
       ↪ pause=0.003):
8      """Send large blob of bytes to target in chunks."""
9      off = 0
10     while off < len(data):
11         end = min(off + chunk, len(data))
12         tgt.write(data[off:end])
13         off = end
14         time.sleep(pause)
15
16 transfor_success = {"Y": 1, "N": 0, "": -1}
17
18
19 ### Setup CW305 + Husky ###
20 scope = cw.scope()
21 scope.clock.clkgen_src = "system"
22 scope.clock.clkgen_freq = 50e6
23 scope.io.hs2 = "clkgen"
24 scope.io.tio1 = "serial_rx"
25 scope.io.tio2 = "serial_tx"
26 scope.trigger.triggers = "tio4"
27 scope.adc.samples = 40000
28 scope.adc.offset = 0
29 scope.clock.adc_mul = 4
30 scope.gain.db = 14
31
32
33
34 BITFILE = "cw305_top.bit"
35 if not os.path.exists(BITFILE):
36     raise FileNotFoundError(BITFILE)
37
38 target = cw.target(scope, cw.targets.SimpleSerial)
```

```
target.baud = 115200
fpga = cw.target(None, cw.targets.CW305, bsfile=BITFILE,
    ↪ force=True)

# Waiting for complete setup
time.sleep(4)
scope.clock.reset_adc()
assert scope.clock.adc_locked
print(scope.adc.basic_mode)


# Parameters
CT_BYTES = 4481
SK_BYTES = 2296

# Load datasets
data_m0 =
    ↪ load_dataset("../hqc-data-generator/data/dataset_m0_10000.bin")
data_mrand =
    ↪ load_dataset("../hqc-data-generator/data/dataset_mrand_10000.bin")

# Build vectors
fixed_vectors = [(sk + b"\x00" * 7, ct) for (_, sk, ct, _)
    ↪ in data_m0]
rand_vectors  = [(sk + b"\x00" * 7, ct) for (_, sk, ct, _)
    ↪ in data_mrand]

fixed_vectors = fixed_vectors[1000:]
rand_vectors = rand_vectors[1000:]

# Interleave fixed & random
test_vectors = []
labels = []
for f, r in zip(fixed_vectors, rand_vectors):
    test_vectors.append(f)
    labels.append("fixed")
    test_vectors.append(r)
    labels.append("random")

results = []
traces = []
batch_labels = []

# Run captures
time0 = time.time()
batch_size = 2000
batch_idx = 1

for idx, (sk, ct) in enumerate(test_vectors):
    print(f"\n=== Running test {idx} ({labels[idx]}) ===")
    assert len(ct) == CT_BYTES and len(sk) == SK_BYTES
```

```
85
86      scope.arm()
87      send_blob(target, ct)
88      send_blob(target, sk)
89
90      if scope.capture():
91          print("Capture timed out!")
92          trace = None
93      else:
94          trace = scope.get_last_trace().astype(np.float32)
95          traces.append(trace)
96
97      if scope.adc.errors != False:
98          print(scope.adc.errors)
99
100     result = ""
101     t0 = time.time()
102     while time.time() - t0 < 60.0:
103         s = target.read(1, timeout=20)
104         if not s:
105             print("miss")
106             continue
107         print("rx:", repr(s))
108         if s in ("Y", "N"):
109             result = s
110             break
111
112     results.append(transfor_success[result])
113     batch_labels.append(labels[idx])
114     time.sleep(0.1)
115
116     # Save every batch_size traces
117     if (idx + 1) % batch_size == 0:
118         out_name =
    ↪ f"./data/new-data/test1/captures_batch{batch_idx}.npz"
119         np.savez(out_name,
120                  traces=np.array(traces, dtype=np.float32),
121                  results=np.array(results, dtype=np.int8),
122                  labels=np.array(batch_labels))
123         with open(f"results_batch{batch_idx}.json", "w") as f:
124             json.dump({"results": results, "labels":
    ↪ batch_labels}, f, indent=2)
125         print(f"Saved batch {batch_idx} -> {out_name}")
126         traces = []
127         results = []
128         batch_labels = []
129         batch_idx += 1
130
131 # Save leftover traces if not a full batch
132 if traces:
133     out_name =
```

```
    ↪ f"./data/new-data/test1/captures_batch{batch_idx}.npz"
134 np.savez(out_name,
135          traces=np.array(traces, dtype=np.float32),
136          results=np.array(results, dtype=np.int8),
137          labels=np.array(batch_labels))
138 with open(f"results_batch{batch_idx}.json", "w") as f:
139     json.dump({"results": results, "labels":
    ↪ batch_labels}, f, indent=2)
140 print(f"Saved␣final␣partial␣batch␣{batch_idx}␣->␣
    ↪ {out_name}")
```

Listing A.1: Data capture script used for generating the power-traces