

Harald Haaland Paaske

Investigating Power Side-Channel Leakage of HQC, and Countermeasures

Master's thesis in Information Security

Supervisor: Tjerand Silde

Co-supervisor: Emil August Hovd Olaisen

June 2026



Norwegian University of
Science and Technology

Harald Haaland Paaske

Investigating Power Side-Channel Leakage of HQC, and Countermeasures

Master's thesis in Information Security
Supervisor: Tjerand Silde
Co-supervisor: Emil August Hovd Olaisen
June 2026

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





Norwegian University of
Science and Technology

Investigating Power Side-Channel Leakage of HQC, and Countermeasures

Paaske, Harald Haaland

Submission date: May 2026

Main supervisor: Silde, Tjerand, NTNU

Co-supervisor: Olaisen, Emil August Hovd, NTNU

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Problem description

The project will look into the post-quantum cryptographic Key Encapsulation Mechanism (KEM) HQC, addressing what it is and how it works, as it becomes important to move towards secure post-quantum cryptographic protocols. These protocols need to be based on hard enough problems for a quantum computer to solve, and also the implementation itself needs to be secure against leakage of sensitive information. The main objective of this project is to investigate side channel attacks on HQC implementations. The literature will go through important aspects of HQC, KEMs and error correcting codes, as well as looking at the side-channel weaknesses of HQC previously proposed.

Approved: 2026-01-10 – Silde, Tjerand, NTNU (Main supervisor)

Abstract

With the rise of quantum computers, new cryptographic standards must be developed, as quantum computers pose a threat to widely used public-key cryptosystems such as RSA and Diffie–Hellman. One important part of testing new standards is side-channel analysis, where information about secret-dependent values can be extracted by analysing physical leakage such as power consumption. This thesis presents a profiling power side-channel analysis of the polynomial multiplication function `base_mul` inside HQC decapsulation, together with selected implementation-level countermeasures.

HQC has already been shown to leak side-channel information in previous work, and this thesis continues this evaluation by testing the PQ-Clean `base_mul` implementation and three alternative implementations: a masked lookup-table variant, a shuffled variant, and an alternative `base_mul2` implementation proposed by Velek et al. (ICISSP 2026).

Our results show strong leakage in the standard implementation, where the profiling model is able to recover the 64-bit secret-dependent value processed in `base_mul` with 100% accuracy when provided with 10 traces of the operation. A profiling model trained on the masked implementation still showed prediction accuracy above random guessing, while the shuffled implementation gave the most promising results, with only two outlier positions where the model was more accurate than random guessing. The `base_mul2` implementation also showed consistent leakage, with model accuracy above random guessing. Overall, the results show that the standard PQClean `base_mul` implementation leaks strongly under the tested profiling setup, while the evaluated countermeasures reduce but do not fully remove exploitable leakage.

Sammendrag

Med utviklingen av kvantedatamaskiner må også nye kryptografiske standarder utvikles. Kvantedatamaskiner utgjør en trussel mot dagens mest brukte offentlig-nøkkel-kryptosystemer som RSA og Diffie–Hellman. En viktig del av testing av nye standarder er sidekanalanalyse, hvor informasjon om hemmelige verdier kan bli funnet ved å analysere fysiske kanaler, som for eksempel strømforbruk. Denne oppgaven presenterer en profilbasert sidekanalanalyse av strømforbruk av polynommultiplikasjonsfunksjonen `base_mul` som foregår i en HQC-dekapsulering, og implementasjonsspesifikke mottiltak.

HQC har allerede blitt vist å lekke sidekanalinformasjon i tidligere arbeid, og denne oppgaven viderefører denne evalueringen ved å teste PQClean-implementasjonen av `base_mul` og tre alternative implementasjoner: en maskert variant, en stokket variant, og en alternativ `base_mul2`-implementasjon foreslått av Velek et al. (ICISSP 2026).

Resultatene våre viser sterk lekkasje i standardimplementasjonen til PQClean, hvor profileringsmodellen er i stand til å gjenopprette den 64-bit hemmelige verdien som behandles i `base_mul` med 100% nøyaktighet når den får 10 spor av operasjonen. En profileringsmodell trent på den maskerte implementasjonen viste fortsatt prediksjonsnøyaktighet over tilfeldig gjetting, mens den stokkede implementasjonen ga de mest loven- de resultatene, med kun to avvikende posisjoner hvor modellen var mer nøyaktig enn tilfeldig gjetting. `base_mul2`-implementasjonen viste også konsistent lekkasje, med modellnøyaktighet over tilfeldig gjetting. Samlet viser resultatene at standard PQClean-`base_mul`-implementasjonen lekker sterkt under det testede profileringsoppsettet, mens de evaluerte mottiltakene reduserer, men ikke fullt ut fjerner utnyttbar lekkasje.

Preface

This master's thesis was written as part of the master's programme in Information Security at the Norwegian University of Science and Technology (NTNU), Department of Information Security and Communication Technology, in spring 2026. I would like to thank my supervisors, Tjerand Silde and Emil August Hovd Olaisen, for providing me with an exciting thesis topic, as well as their guidance, feedback, and support during the thesis work.

Contents

List of Figures	xi
List of Tables	xiii
List of Symbols	xv
List of Acronyms	xvii
1 Introduction	1
1.1 Motivation	1
1.2 Main objective and research questions	3
1.3 Scope and limitations	3
1.4 Related work	4
1.4.1 Attacks against HQC	4
1.5 Our contribution	6
1.6 Thesis structure	6
1.7 Sustainability	7
2 Theory	9
2.1 Key encapsulation mechanism	9
2.2 Error-correcting codes and HQC	10
2.2.1 HQC cryptographic scheme	11
2.3 Side-channel analysis	12
2.3.1 Power analysis	12
2.3.2 Profiling attacks	13
2.3.3 Classification models	13
2.4 Countermeasures to side-channel analysis	15
2.4.1 Masking	15
2.4.2 Shuffling	16
3 Technical setup	17
3.1 PQClean’s HQC implementation	17
3.1.1 KEM functions	18

3.1.2	HQC PKE functions	18
3.1.3	Polynomial multiplication	19
3.2	ChipWhisperer and HQC technical setup	20
3.2.1	Software and hardware setup	20
3.2.2	HQC integration and firmware adaptation	22
3.2.3	Build, flashing, and capture configuration	23
3.2.4	Jupyter workflow	24
3.2.5	Code availability	24
3.2.6	KEM Demonstration Program	24
3.2.7	Implementation validation	25
4	Methodology	27
4.1	Target operation	27
4.2	Methodological strategy	28
4.3	Experimental procedure	29
4.3.1	Exploratory trace comparison	29
4.3.2	Isolating <code>base_mul</code>	30
4.3.3	Classification models	31
4.3.4	Model 1: Nearest-centroid classification model	31
4.3.5	Model 2: Logistic-regression model	38
4.3.6	Comparing models	39
4.4	Countermeasure implementations	39
4.4.1	Masked lookup-table implementation	40
4.4.2	Shuffled implementation	42
4.4.3	Alternative protected implementation	44
4.5	Evaluation	47
4.6	Methodological limitations	48
4.7	Description of use of artificial intelligence	49
5	Results	51
5.1	Exploratory trace comparison	51
5.2	Isolated <code>base_mul</code> trace inspection	51
5.3	Unprotected <code>base_mul</code> classification	53
5.4	Countermeasure results	55
5.4.1	Masked lookup-table implementation	55
5.4.2	Shuffled implementation	55
5.4.3	Alternative <code>base_mul2</code> implementation	56
5.5	Comparison of implementations	57
6	Discussion	61
6.1	Interpretation of the baseline leakage	61
6.2	Effectiveness of supervised profiling for classification	62

6.3	Why it leaks	63
6.4	Effect of countermeasures	64
6.4.1	Effects of the masked implementation	65
6.4.2	Effects of the shuffled implementation	66
6.4.3	Effects of the alternative <code>base_mul2</code> implementation	67
6.5	Comparing the implementations	68
6.6	Discussion of research questions	69
6.7	Limitations	70
6.8	Future work	71
7	Conclusions	73
	References	75

List of Figures

3.1	ChipWhisperer-HUSKY hardware setup.	21
4.1	The 16 cropping boundaries applied to a base_mul power trace. The boundaries are the beginning and end of one segment.	33
4.2	Example of the aligned cropped segments with length REPEATED_CYCLE_LEN = 434 applied to a base_mul power trace. Used to find correct segment length.	33
4.3	Example of the aligned cropped segments with length REPEATED_CYCLE_LEN = 37 applied to a base_mul2 power trace. Used to find correct segment length.	46
4.4	The 63 cropping boundaries applied to a base_mul2 power trace. The boundaries mark the beginning and end of one segment.	46
5.1	200 base_mul traces averaged sample-wise and combined to one trace. Captured with key-limb value 0x01a745a63d40c592.	52
5.2	Zoomed in plot of Figure 5.1, from sample index 3000 to 4000. Showing trace segments corresponding to nibble values 3 (left) and 6 (right). . .	53
5.3	Accuracy per position of guessed key limb from base_mul traces using the unmasked, masked, shuffled and base_mul2 implementations compared to expected random guesses. Guess based on a single trace.	58
5.4	Accuracy per position of guessed key limb from base_mul traces using the unmasked, masked, shuffled and base_mul2 implementations compared to expected random guesses. Guess based on the average of 200 traces with the same key limb.	58
5.5	Accuracy per position of guessed key bit from the base_mul2 implementation compared to expected random guesses. Guess based on a single trace.	59
5.6	Accuracy per position of guessed key bit from the base_mul2 implementation compared to expected random guesses. Guess based on the average of 200 traces with the same key limb.	59

List of Tables

3.1	ChipWhisperer trace capture configuration.	24
4.1	Cropping parameters used for trace segmentation.	34
4.2	Hyperparameter search space used for model selection.	37
4.3	ADC sample settings used for each implementation.	40
4.4	Cropping parameters used for the base_mul2 trace segmentation.	47
4.5	Datasets used in the experimental methodology.	48
5.1	Mean absolute differences between averaged trace sets.	52
5.2	Selected hyperparameter configurations for the standard base_mul profiling data.	53
5.3	Limb-recovery performance of the nearest-centroid model (NCM) and logistic-regression model (LRM). Exact recovery means that all 16 nibbles of the 64-bit limb were classified correctly. Results are computed from 400 benchmark files using seed 1337.	54
5.4	Trace cost for first exact limb recovery on 400 files.	54
5.5	Summary of full-trace classification results for the masked model.	55
5.6	Summary of full-trace classification results for the shuffled model.	56
5.7	Per-position accuracy for the shuffled full-trace model. The columns correspond to the number of averaged traces, denoted n_{avg} . Position 07 with higher accuracy is highlighted in bold.	56
5.8	Summary of full-trace classification results for the BM2 model. Reported values are from 400 files.	57

List of Symbols

a	64-bit secret-dependent input limb to <code>base_mul</code> .
a_w	four-bit nibble of a at position w .
b	64-bit public or ciphertext-dependent input limb to <code>base_mul</code> .
$\mathcal{C}$	error-correcting code used in HQC.
c	128-bit output of <code>base_mul</code> , stored as two 64-bit words.
$\hat{c}$	predicted class label output by the classifier.
δ_w	input mask applied to nibble position w in the masked implementation.
$\mathbf{e}$	error vector used during HQC encapsulation.
f	feature vector extracted from selected trace samples.
$\mathbb{F}_2$	finite field with two elements.
f'	standardized feature vector used as input to the classifier.
$\mathbf{G}$	generator matrix of the error-correcting code.
$\mathbf{h}$	public vector in the HQC public key.
$\mathbf{m}$	message encoded during HQC encapsulation.

μ_f	feature-wise mean computed from the training feature vectors.
m_w	output mask applied to nibble position w in the masked implementation.
n_{avg}	number of traces averaged before classification.
$\mathbf{r}_1$	random sparse vector used during HQC encapsulation.
$\mathbf{r}_2$	random sparse vector used during HQC encapsulation.
ρ	radius around each point of interest used during feature extraction.
$\mathbf{s}$	public key component computed from the secret vectors and $\mathbf{h}$.
σ_f	feature-wise standard deviation computed from the training feature vectors.
$\mathbf{u}$	ciphertext component in HQC.
$\mathbf{v}$	ciphertext component in HQC.
w	nibble or bit position index, depending on the experiment.
$\mathbf{w}$	intermediate value computed during HQC decapsulation before decoding.
$\mathbf{x}$	secret sparse vector in the HQC secret key.
$\mathbf{y}$	secret sparse vector in the HQC secret key.

List of Acronyms

ADC	Analog-to-Digital Converter.
API	Application Programming Interface.
BCH	Bose–Chaudhuri–Hocquenghem.
CPA	Correlation Power Analysis.
CW	ChipWhisperer.
DPA	Differential Power Analysis.
ECC	Error-Correcting Code.
HQC	Hamming Quasi-Cyclic.
KEM	Key Encapsulation Mechanism.
LRM	Logistic-Regression Model.
ML	Machine Learning.
NCM	Nearest-Centroid Model.
NIST	National Institute of Standards and Technology.
POI	Point of Interest.
PQC	Post-Quantum Cryptography.
SAM4S	Atmel SAM4S microcontroller.
SCA	Side-Channel Analysis.
SNR	Signal-to-Noise Ratio.
SPA	Simple Power Analysis.

Chapter 1

Introduction

This thesis is about post-quantum cryptography and how the newly selected HQC scheme can be attacked and protected against side-channel information such as power consumption leakage. This chapter introduces the background and motivation for the thesis, and provides the reader with a clear understanding of the goal, research questions, scope, and structure of the thesis.

1.1 Motivation

While the threat of quantum computers is nothing new, current advancements in the field of quantum computing from corporations like IBM, Microsoft, Google and others, make the need for *Post-Quantum Cryptography* (PQC) larger than before [Val24; LCC+23]. Due to this, the planned roll-out of PQC is pushed forward. For example, Google has announced a 2029 timeline for migrating to PQC in their services [AS26], and Android 17 is planned to have built-in PQC [LE26].

Quantum computers use quantum mechanics to perform computations in a way classical computers cannot. Instead of regular bits that can be 1 or 0, a quantum computer uses *qubits*, which can represent more than one state at a time. This makes it possible to construct algorithms that take advantage of this property to solve certain problems faster than classical computers. As of 2026, cryptographically relevant quantum computers capable of breaking deployed public-key cryptography are not available. This does not mean the threat is not real. With a “harvest now, decrypt later” attack, an attacker stores large amounts of encrypted data in the hope of decrypting it in the future, when quantum computers may be available and the information may still be valuable [Nat24b].

One of the most famous quantum algorithms is Shor’s algorithm [Sho94], which can be used for factoring large numbers and finding discrete logarithms. This directly affects RSA and Diffie-Hellman, which are the two main public-key cryptosystems used today [LCC+23]. This means that new cryptosystems designed to withstand

2 1. INTRODUCTION

the capability of a quantum computer must rely on other mathematical foundations to remain secure. These cryptosystems are the ones called PQC.

Mainly public-key cryptography is threatened by quantum computers. Symmetric cryptography, such as AES with sufficiently large keys, is still considered resistant to quantum attacks. The main problem is therefore how shared key material can be established securely against a quantum adversary. One important approach for this purpose is a *key encapsulation mechanism* (KEM) built upon public-key encryption, where two parties establish a shared secret that can later be used for symmetric encryption.

NIST has over previous years held a standardization process for post-quantum cryptosystems, with the goal of selecting standardized algorithms [Nat24b]. As of today, ML-KEM, formerly known as CRYSTALS-Kyber, has been selected as the primary key-encapsulation mechanism [Nat24a]. ML-KEM is a lattice-based cryptosystem based on the learning with errors problem. HQC has also been selected for standardization as an additional key-encapsulation mechanism. This is important because HQC is based on error-correcting quasi-cyclic codes, and therefore relies on a different mathematical foundation than ML-KEM [NIS25].

With the development of new cryptographic schemes like HQC, it is important to ensure that rigorous testing has been performed before the new schemes are deployed in critical systems. Deploying vulnerable cryptographic schemes can have severe consequences, as was shown by Bernstein in 2005 when he proposed a side-channel attack on the mathematically secure cryptographic standard AES [Ber05]. The attack, and many others, demonstrate that a cryptographic scheme is only as strong as its weakest link, which in this case was the implementation. Therefore, it is important not only to develop cryptographic schemes with a strong mathematical foundation, but also to secure implementations against side-channel attacks. As of early 2026, several papers have been written on the topic of side-channel analysis of HQC, showing that implementation-level leakage is an important concern for the scheme [PT19; SRSW21; GLG22; RBG+26; VRB26].

Post-quantum cryptography motivates this thesis because future migration to post-quantum standards requires implementation-level security testing. With new standards comes the need for a large amount of testing, where one important security aspect is side-channel leakage. Since several papers already address this topic, the results show that there is still work to be done before specific HQC implementations can be considered side-channel resistant in a deployment setting. This thesis therefore continues the evaluation of HQC implementation leakage by testing the PQClean `base_mul` implementation and comparing it with masked, shuffled, and alternative multiplication variants.

1.2 Main objective and research questions

Previous work has shown different parts of HQC can leak side-channel information and be vulnerable to attacks. This mainly includes the decoding step and polynomial multiplication. PQClean provides a clean and standalone implementation of HQC, where polynomial multiplication happens using a `karatsuba` algorithm with `base_mul` as the base case for multiplication [KSSW22; Wei]. During decapsulation, this algorithm processes secret-dependent values, making it a relevant target for side-channel attacks. The problem addressed in this thesis is not whether HQC polynomial multiplication can leak in principle, but to evaluate recoverable information in isolated executions of PQClean’s `base_mul`, set this as a baseline, and observe how the results change under different implementation-level countermeasures using the same general evaluation setup.

The thesis evaluates three alternative implementations: a masked lookup-table variant, a shuffled variant, and the alternative `base_mul2` implementation proposed by Velek et al. [VRB26]. These variants are tested under the same general profiling methodology to determine how the observed exploitable leakage changes across the different implementations.

Based on this main objective, the following research questions have been derived in order to guide the thesis:

RQ1: What side-channel weaknesses in HQC have been identified in previous work, and which implementation-level countermeasures have been proposed to reduce them?

RQ2: What recoverable secret-dependent information is present in isolated executions of PQClean’s `base_mul` on a ChipWhisperer-HUSKY platform?

RQ3: To what extent can supervised profiling models recover secret-dependent key-limb information from isolated `base_mul` power traces?

RQ4: How do the masked, shuffled, and alternative `base_mul2` implementations affect the observed leakage compared to the standard PQClean `base_mul` implementation under the same general profiling setup?

1.3 Scope and limitations

The scope of this thesis is limited to power side-channel analysis of the PQClean implementation of HQC. More specifically, the analysis is done as isolated experiments on the `base_mul` polynomial multiplication function, which is used as part of the `karatsuba` algorithm. The experiments measure the accuracy of trained profiling models when given unlabeled traces of the multiplication operation. The

models are not optimized for performance or accuracy, but are rather simple and meant to evaluate recoverable side-channel leakage, and how the leakage is affected by different countermeasure implementations. The masked and shuffled implementations are based on general side-channel countermeasure principles, but are not taken directly from any previous HQC side-channel countermeasure implementation. The experiments on these implementations should therefore be interpreted as first-iteration testing, rather than as proposed implementation fixes. The thesis does not demonstrate a complete end-to-end key-recovery attack against HQC decapsulation, but instead studies whether isolated 64-bit limbs can be recovered under controlled conditions.

1.4 Related work

Since HQC was proposed as a post-quantum KEM in 2016, considerable research has focused on evaluating the scheme and preparing it for real-world deployment. As described in Section 1.1, an important part of this process is securing the implementations against side-channel leakage. This includes timing attacks, simple power analysis, and profiling and machine learning based side-channel attacks [SRSW21; PT19; RBG+26]. This section focuses specifically on previously proposed side-channel research, attacks and countermeasures for HQC, and explains how this thesis relates to this work.

1.4.1 Attacks against HQC

One of the first implementation-level attacks against HQC was proposed by Paiva et al. in 2019 [PT19]. This article addresses the fact that the BCH decoder from the decryption process, is not in constant time and the processing time depends on the number of corrected errors in the ciphertext. This means with a timing attack it is possible to leak the number of errors corrected by the decoder. From this they construct a set of chosen ciphertexts with known r_1, r_2 , which are random sparse vectors used during HQC encapsulation to construct the ciphertext component $u = r_1 + hr_2$. They showed that distances related to the secret key produced measurable differences in decryption time. This information was then used to reconstruct the full decryption key.

After the paper by Paiva et al. was published, a constant-time decoding algorithm was implemented into HQC. In 2021, Schamberger et al. [SRSW21] published a paper addressing that a side-channel attack was still possible. This paper showcases an attack on the new BCH-decoder by utilizing power side-channel analysis, they could build an oracle that determines if a code has been corrected by the BCH decoder, for a chosen ciphertext. By combining this with a linear-algebra based recovery method,

this oracle allowed the authors to recover the decryption key with fewer than 10,000 measurements.

Another side-channel and chosen ciphertext attack was proposed in a paper from Goy et al. [GLG22]. They demonstrate a key-recovery electromagnetic side-channel attack with chosen ciphertext that exploits the reuse of static keys with physical access to a micro-controller. This attack exploits the diffusion property of the Reed-Muller decoding step. With this information they build an oracle that distinguishes between several decoding patterns of the Reed-Muller codes. They show how to retrieve the entire key with less than 20,000 electromagnetic attack traces. Finally they propose a masked implementation that they determine to be effective against the showcased attack.

Another highly relevant paper is a profiling side-channel attack on HQC presented by Rabas et al. in October 2025 [RBG+26]. They show the possibility of a power side-channel attack that exploits power consumption leakage during polynomial multiplication at the start of the decapsulation process. As in this thesis, they target the `base_mul` function from the “additional implementation”, which is the same as used in PQClean. With the use of a profiling attack based on Welch’s ANOVA statistical tests [Wel51], and a multilayer deep learning model for classification, they are able to recover intermediate values during HQC polynomial multiplication. The results show only 6 out of 20,000 tests produced an incorrect result, in recovering a key-bit value from a trace. Further, they propose two countermeasures, where one still shows exploitable leakage, and the other they argue to be competitive.

After the electromagnetic SCA paper by Goy et al. and the masked countermeasure to the attack, Goy et al. propose a fully masked HQC implementation of arbitrary order d [GSA+25]. The paper presents a detailed sensitivity analysis of HQC, and a fully masked HQC implementation. In this implementation, sensitive variables are split into several shares so that no individual share reveals the underlying secret. Their work presents a masked PQC implementation proven secure in the MIMO-SNI security model, which is a strong probing-based model for side-channel resistance [CS20].

In early 2026, Velek, Rabas, and Buček presented a single-trace Simple Power Analysis attack targeting polynomial multiplication in HQC, similar to Rabas et al. [VRB26]. Their attack focuses on power leakage from the `base_mul` function, which is used for multiplying two 64-bit values over $\mathbb{F}_2$. By capturing a single trace on a ChipWhisperer-Lite platform, they show that a consistent pattern is visible in the repeated structure of the trace, which is directly related to the secret-dependent values used during processing. The experiments report a 99.69% success rate over 10,000 attack attempts. The paper also presents a countermeasure named

`base_mul2` that removes the precomputed lookup-table operation entirely from the implementation. This work is closely related to this thesis because both works investigate leakage from the same `base_mul` implementation. However, this thesis uses a different ChipWhisperer setup and attack approach. In addition, this thesis investigates whether the `base_mul2` countermeasure still contains exploitable side-channel leakage, with the use of a profiling attack.

Overall, these papers show that HQC implementations can leak secret-dependent information through timing, power, and electromagnetic side channels. They exploit leakage from the cryptographic implementation rather than the mathematical foundation of HQC. These works show that implementation security must be considered in addition to the cryptographic security of the underlying scheme. Other attacks against HQC do exist and include attacks against the syndrome decoding problem and structural attacks such as DOOM [DGA+25]. Such attacks are outside the scope of this thesis, which focuses on power side-channel leakage from polynomial multiplication.

1.5 Our contribution

The contribution of this thesis is a controlled evaluation of leakage from PQClean’s `base_mul` implementation on a ChipWhisperer-HUSKY platform. While previous work has shown that HQC polynomial multiplication can leak secret-dependent information, this thesis isolates `base_mul` and collects labeled traces for known 64-bit limbs. These traces are used to test whether simple supervised profiling models can recover the four-bit nibble values processed by the function. The thesis also compares the standard implementation with masked, shuffled, and alternative `base_mul2` variants, and shows how other `base_mul` countermeasure implementations affect the exploitable leakage under the same general profiling method.

1.6 Thesis structure

Chapter 2 presents the theoretical background on KEMs, HQC, side-channel analysis, profiling attacks, and countermeasures. Chapter 3 describes PQClean and the technical setup used to run and measure HQC on the ChipWhisperer platform. Chapter 4 presents the experimental method, including trace capture, segmentation, classification, and countermeasure implementations. Chapter 5 presents the results of the experiments. Chapter 6 discusses the implications and limitations of the results, and finally Chapter 7 concludes the thesis.

1.7 Sustainability

This thesis is relevant in relation to the UN Sustainable Development Goals, in particular goal number 9: industry, innovation and infrastructure [FN-25], and more specifically 9.1 which is aimed at developing quality, reliable, sustainable and resilient infrastructure. Secure cryptographic protocols and implementations are important parts of building reliable and robust digital infrastructure. As more public and private services depend on digital systems, failures in cryptographic protection can affect privacy and security against the threat of quantum adversaries. Secure post-quantum cryptography is therefore relevant for maintaining robust digital infrastructure, especially when today's encrypted data may still be valuable when future quantum computers become more capable. With the objective of investigating implementation-level leakage in HQC, this work contributes to secure digital infrastructure by evaluating whether an implementation of a post-quantum KEM leaks secret-dependent information before such schemes are deployed more widely.

Chapter 2

Theory

This chapter provides the theoretical background for the rest of the thesis. Parts of this chapter are adapted from the report written for the course IMT4205 pre-project and have been adjusted to fit the scope, notation, and structure of this thesis [Paa25].

2.1 Key encapsulation mechanism

A *Key Encapsulation Mechanism* (KEM) is a cryptographic method used to establish a shared secret between two parties, that can be used for further symmetric cryptography [WGR22]. The main purpose of a KEM is therefore key establishment, not direct message encryption. However, a KEM is closely related to public-key encryption (PKE), and many KEMs are built on underlying public-key encryption schemes. KEMs are also related to key-exchange mechanisms such as Diffie–Hellman [DH76], but with a different structure: one party encapsulates a key using the receiver’s public key, and the receiver decapsulates it using the corresponding secret key, instead of the key being a mix of both parties private material.

A KEM consists of three main algorithms: key generation, encapsulation and decapsulation. The key generation algorithm generates a public-private key pair used for encapsulation and decapsulation:

$$(pk, sk) \xleftarrow{\$} \text{KeyGen}(1^\lambda).$$

The encapsulation algorithm takes the public key as input and outputs a shared key kept by the sender, and a ciphertext sent to the receiver:

$$(c, K) \xleftarrow{\$} \text{Encaps}(pk).$$

The decapsulation algorithm then uses the secret key of the receiver and the ciphertext to recover the same shared key as generated from encapsulation:

$$K' \leftarrow \text{Decaps}(sk, c),$$

where K' is either a key in the key space $\mathcal{K}$, or a failure symbol $\perp$.

Correctness means that an honestly generated ciphertext should decapsulate to the same key that was produced during encapsulation. A KEM is called ε -correct if

$$\Pr \left[K' \neq K : (pk, sk) \xleftarrow{\$} \text{KeyGen}(1^\lambda), (c, K) \xleftarrow{\$} \text{Encaps}(pk), K' \leftarrow \text{Decaps}(sk, c) \right] \leq \varepsilon.$$

If $\varepsilon = 0$, the KEM is called perfectly correct.

An important security property of KEMs is indistinguishability under chosen-ciphertext attack, usually denoted IND-CCA [BFG+19]. In relation to KEMs, this means that an attacker should not be able to distinguish a real decapsulated key from a randomly generated key, even if the attacker is allowed to query decapsulations of other ciphertexts. In many modern post-quantum KEMs, IND-CCA security is achieved by applying a Fujisaki–Okamoto-like transform to an underlying IND-CPA secure public-key encryption scheme [JMZ23].

2.2 Error-correcting codes and HQC

Error-correcting codes (ECC) are not exclusive to cryptography. For example, when data signals are sent over cables, WiFi or radio, there is a chance the signals get disrupted by external factors, resulting in modified data. To detect and correct such errors, error-correcting codes are used. Error-correcting codes can also be used as the mathematical basis for cryptographic schemes. The security assumption in code-based cryptography is that it is difficult to decode certain codewords where errors have deliberately been introduced [Ham50; Ber68]. One such cryptographic scheme is HQC.

HQC is a code-based post-quantum cryptographic scheme whose security is based on decoding problems for quasi-cyclic codes in the Hamming metric [ABD+18; DGA+25]. The Hamming metric measures how many bit-positions differ between two vectors. For $\mathbf{a} \in \mathbb{F}_q^n$, the *Hamming weight* is the number of non-zero positions, while for $\mathbf{a}, \mathbf{b} \in \mathbb{F}_q^n$, the *Hamming distance* is the number of positions in which they differ:

$$\begin{aligned} \text{wt}_H(\mathbf{a}) &= |\{i \in \{1, \dots, n\} \mid a_i \neq 0\}|, \\ d_H(\mathbf{a}, \mathbf{b}) &= |\{i \in \{1, \dots, n\} \mid a_i \neq b_i\}|. \end{aligned}$$

The Hamming distance is induced from the Hamming weight, that is $d_H(\mathbf{a}, \mathbf{b}) = \text{wt}_H(\mathbf{a} - \mathbf{b})$ [WGR22].

The *syndrome decoding problem* (SDP) is a fundamental problem in coding theory and the basis of most of code-based cryptography. SDP is described as follows: Let $\mathbf{H}$ be a parity-check matrix of a code, this is the matrix from which we can derive

the parity-check equations and determine if a vector $\mathbf{c}$ is a valid codeword. The codeword $\mathbf{c}$ is valid if $\mathbf{H}\mathbf{c} = \mathbf{0}$. If an error vector $\mathbf{e}$ is added to $\mathbf{c}$, the received vector can be written as $\mathbf{r} = \mathbf{c} + \mathbf{e}$, and the value $\boldsymbol{\sigma} = \mathbf{H}\mathbf{r}$ is called the syndrome. The SDP is about recovering the error vector $\mathbf{e}$ from the syndrome $\boldsymbol{\sigma}$, in order to revert the errors and obtain the original $\mathbf{c}$ [ES24]. This problem was proven to be NP-complete by Berlekamp, McEliece and van Tilborg [BMvT78].

2.2.1 HQC cryptographic scheme

HQC is a KEM built on top of a simpler PKE scheme, and therefore consists of the same three core KEM algorithms: key-generation, encapsulation, and decapsulation. The following description of the HQC KEM algorithms is based on Weger et al. [WGR22], with small adjustments made for clarity and notation consistency.

KeyGen: An error-correcting code $\mathcal{C}$ is chosen, that can efficiently decode up to a specified number of errors, using generator matrix $\mathbf{G}$. Now, choose a public vector $\mathbf{h}$, and secret sparse vectors $\mathbf{x}$ and $\mathbf{y}$ of a fixed Hamming weight. Then compute:

$$\mathbf{s} = \mathbf{x} + \mathbf{h}\mathbf{y},$$

using the polynomial/vector operations defined by the scheme.

The public key contains $\mathbf{G}$, $\mathbf{h}$, $\mathbf{s}$, and weight parameters, while the secret key contains only the secret vectors $\mathbf{x}$ and $\mathbf{y}$.

Encaps: Choose a message $\mathbf{m}$, an error vector $\mathbf{e}$, and random vectors $\mathbf{r}_1$ and $\mathbf{r}_2$ of fixed Hamming weight. Then compute:

$$\mathbf{u} = \mathbf{r}_1 + \mathbf{h}\mathbf{r}_2$$

and

$$\mathbf{v} = \mathbf{m}\mathbf{G} + \mathbf{s}\mathbf{r}_2 + \mathbf{e},$$

using the polynomial/vector operations defined by the scheme. The ciphertext is then $\mathbf{c} = (\mathbf{u}, \mathbf{v})$.

Decaps: Use the secret vector $\mathbf{y}$ to remove the secret-dependent part of the ciphertext by computing

$$\mathbf{w} = \mathbf{v} - \mathbf{u}\mathbf{y}.$$

Since the operations are over $\mathbb{F}_2$, subtraction is equivalent to addition. The secret vector $\mathbf{x}$ does not appear explicitly in this multiplication, but contributes to the remaining error term that the decoder must correct. The resulting word is then passed to the decoding algorithm of $\mathcal{C}$:

$$\mathbf{m}' = \text{Decode}(\mathbf{w}).$$

For the decryption process to recover the original message, the remaining error pattern must be within the correction capability of the code $\mathcal{C}$. The code $\mathcal{C}$ is built from error-correcting codes used by HQC, including BCH codes, and its decoding algorithm is used to recover the message during decapsulation [BR60; DGA+25].

For this thesis, the most important point is the multiplication $\mathbf{u}\mathbf{y}$ in the decapsulation step. Here, $\mathbf{u}$ comes from the ciphertext, while $\mathbf{y}$ is part of the secret key. This means that HQC decapsulation performs polynomial/vector operations involving secret-key material and ciphertext-dependent values. This makes the implementation path from HQC decapsulation to polynomial multiplication relevant for side-channel analysis.

2.3 Side-channel analysis

When targeting cryptographic schemes, either for an attack or for identifying implementation weaknesses, there are two broad approaches. Classical cryptanalysis, which targets the mathematical foundation of the scheme, and side-channel analysis (SCA), which targets measurable leakage from the hardware or software implementation. Such leakage can include execution time, power consumption, electromagnetic radiation, memory-access behaviour, and other implementation-dependent effects [Sta10]. This means that a cryptographic scheme can be mathematically secure, while a concrete implementation may still leak information about secret-dependent values.

2.3.1 Power analysis

This thesis focuses on power-based side-channel analysis. In power analysis, the attacker measures the power consumption of a device while it performs cryptographic operations and attempts to relate features in the measured traces to secret-dependent intermediate values. Two classical forms of power analysis are simple power analysis (SPA) and differential power analysis (DPA) [MOP07].

SPA relies on visible and repeatable patterns in one or a small number of traces. If the implementation has a clear relationship between the executed operations and the secret data, the attacker may be able to infer secret information directly from the trace structure. DPA is a statistical approach where many traces are collected and combined to reveal secret-dependent leakage that may not be visible in individual traces. This makes DPA effective when the leakage is weak or the measurements are noisy [Sta10].

2.3.2 Profiling attacks

A profiling attack is related to SPA in the sense that it exploits repeatable patterns in the power trace, but the recovery of secret-dependent values is not performed by visual inspection alone. Instead, the trace patterns are learned from labeled traces and used for classification.

A profiling attack is a powerful side-channel attack where the attacker first builds a leakage model using traces where the processed values are known [CRR03]. In the profiling phase, profiles are built from a collection of traces that share the same processing feature we want to exploit. This will most often be secret-dependent values. In the attack phase, the model is then applied, where traces from the same process, but with unknown values are captured and compared against the profiles to find the most similar one [MMT15]. There are three main stages in the profiling pipeline: point-of-interest selection, feature extraction, and classification [PHJB19].

Points of interest, feature extraction and preprocessing

A captured power trace contains many sample points, but many of these may be noise, unrelated operations, or samples that are not useful for classification. The goal is therefore to extract the parts of the trace that relate to the secret-dependent processing. To achieve this, the most informative samples in the trace are selected as points of interest (POIs) [MMT15; CRR03]. After the points of interest have been selected, the relevant trace samples must be converted into feature vectors that can be used by the classifier. A feature can be a single selected sample point, a small window around a selected point, or a transformed value derived from the trace. In profiling attacks, this step reduces the raw trace to a smaller representation that preserves the information needed for classification.

Selecting sufficient and informative features is important when building profiling models. Picek et al. show that feature selection can have a strong effect on the performance of profiling attacks, and that this is especially important for noisy traces or traces affected by countermeasures [PHJB19].

Preprocessing is often applied before classification to make traces more comparable. This can include aligning traces to reduce timing variation, normalizing or standardizing the selected features, or applying dimensionality-reduction methods. The purpose is not to create new leakage, but to make the existing leakage easier for the classifier to use [PHJB19].

2.3.3 Classification models

The next important step in a profiling attack pipeline is a classification model. This is how features are compared, and how similarity is determined. Many different

methods of performing classification exist, for this thesis, two models are highlighted. These are nearest-centroid and logistic regression classification.

Nearest-centroid classification

Nearest-centroid classification is a simple supervised classification method where each class with shared features is represented by a centroid. A centroid is the arithmetic mean of all feature vectors belonging to traces from the same computation (class). When classifying an unlabeled trace, its feature vector is compared against all centroids of the model, where the unlabeled trace is selected as belonging to the class where it has the shortest distance to the class centroid. Hence nearest centroid classification [HTF09].

When used in relation to profiling attacks, each class can represent a possible value for a specific operation, this can for example be 1 or 0 in a binary case, or 0-f in a 4-bit nibble case. An unseen feature vector is compared against the class centroids and classified as the one that is most similar. This makes the model useful as a simple baseline, because it directly tests whether the selected features form distinguishable class clusters. However, the method’s limitation lies in the assumption that distance to the class mean is sufficient for classification. If the class distributions overlap strongly, insufficient POI selection, or if the leakage is affected by noise, a more flexible classifier may perform better [HTF09].

Logistic regression classification

Logistic regression is another supervised classification model that uses a weighted combination of input features to estimate which class an unknown feature vector belongs to. For multiclass problems such as nibble classification, multinomial logistic regression is used to compute one score for each class and convert these scores into class probabilities using the softmax function below, where s_k is the score for class k and $\sum_j \exp(s_j)$ is the sum over the exponentiated scores for all classes [HTF09; sci26a]. The predicted class is then the class with the highest estimated probability for the unknown feature vector.

$$P(y = k \mid x) = \frac{\exp(s_k)}{\sum_j \exp(s_j)}$$

In a profiling side-channel attack, the input to the classification model is the feature vector from a trace, and the output is a predicted value of the targeted intermediate variable. Compared with nearest-centroid classification, multinomial logistic regression is more flexible because it can learn decision boundaries between closely related classes rather than only comparing distances to class averages. However,

it is a linear model and therefore limited by the information contained in the selected features. If the selected features do not clearly separate the classes, the accuracy of the model will reflect this.

2.4 Countermeasures to side-channel analysis

To counter side-channel attacks, the objective is to remove patterns in either timing of operations or in the power traces. To remove any secret-dependent patterns, we can either try to keep everything the same and constant, or introduce randomness. Generally, in countermeasures to timing attacks, the goal is to have every secret-dependent operation happen in constant time. The goal is that the execution time does not depend on the secret values being processed. For power-based attacks, it is harder to keep everything constant, as operations with different values can still affect the CPU's power consumption differently, even when the execution time is constant. Therefore randomness is often used as a countermeasure for these kinds of attacks [MOP07; Sta10]. Two common approaches are masking and shuffling.

2.4.1 Masking

Masking is a countermeasure to power-based side-channel attacks that introduces randomness into the values being processed. Instead of processing the secret-dependent value directly, the value is split into two or more shares. In a simple Boolean masking scheme, one share is the mask itself, and the other is the original value XORed with that mask. Let x be a secret-dependent value and let m be a random mask. The masked value can then be written as

$$x' = x \oplus m.$$

The original value can later be recovered by XORing with the same mask again:

$$x = x' \oplus m.$$

This works because XORing the same value twice cancels it out:

$$(x \oplus m) \oplus m = x.$$

The operation in question can then be performed on the shares, and the final result is obtained by recombining them. This result should be the same as if the computation had been performed on the original unmasked value. A masking operation is said

to be of order d when the value x is split into d shares. A higher order of d means more shares must be obtained by an attacker, which increases the level of security. The purpose is to reduce the direct relationship between the secret-dependent value and the measured leakage, since the individual shares appear randomized to the attacker [MOP07; CRR03]. One downside to this is the increase in processing time and implementation complexity, as the operation must be performed on multiple shares instead of a single value. Depending on the number of shares and the size of the operation, this can have a significant impact on performance [MOP07].

2.4.2 Shuffling

Shuffling also introduces randomness into the processing, but contrary to masking, the values themselves are not randomized. Instead, the order of operations is randomized. For example, if the same operation must be performed on each bit of a secret-dependent value, these operations can be executed in a random order and pieced back together at the end. The final result should still be the same as if processed normally [Pes16].

The purpose of shuffling is to reduce the direct relationship between a fixed point in the power trace and a specific secret-dependent operation. If the operation order changes for each execution, an attacker can no longer assume that a given sample region always corresponds to the same processed value. This can make trace alignment, averaging, and classification more difficult, because the leakage from a specific intermediate value is spread across different time positions [Pes16].

Shuffling also has its downsides. For example, shuffling requires randomness and additional control logic to decide the execution order. Parts of the values themselves can still leak, even though the order is scrambled, in some situations it can still be undesirable. Depending on the implementation, it may also introduce overhead from generating and applying the permutation.

Chapter 3

Technical setup

This chapter describes the technology used in the thesis and the process of setting up the initial experimental environment, running HQC, and capturing power traces before the experiments presented in Chapter 4. It explains how the HQC cryptographic operations were configured, executed, and measured using a ChipWhisperer-HUSKY capture board and a SAM4S target board. The purpose of this chapter is to document the experimental environment for correctness, clarity, and reproducibility for future experiments building on the work presented in this thesis.

3.1 PQClean’s HQC implementation

PQClean is an open source project that collects clean implementations of post-quantum cryptographic schemes from the NIST post-quantum cryptography project, where HQC is one of the included schemes [KSSW22; NIS25]. The goal of the project is to provide modern, clean, and standalone implementations that can be integrated into libraries such as liboqs [Ope26] and pqm4, used in benchmarking frameworks such as SUPERCOP, and serve as starting points for evaluation of implementation security [PQC25]. The implementations are written in C and are intended to be easy to test, compare, and integrate into other software projects. This makes PQClean suitable as an implementation basis in this thesis, where the source code must be inspected, compiled, and modified in order to analyze implementation-level side-channel leakage.

An important limitation is that PQClean is no longer under active maintenance, and the repository is planned to be archived as read-only in July 2026 [PQC25]. The repository now points users to other maintained sources for future implementations, including the HQC project repository for HQC. This does not affect its use in this thesis as a fixed implementation target, but it should be considered when interpreting the results as implementation-specific.

The PQClean implementation of HQC is its own standalone C implementation with support for key-generation, encapsulation, and decapsulation. The package also

supports the parameter sets **HQC-128**, **HQC-192**, and **HQC-256**, which correspond to different security levels. For tracing the implementation path from HQC decapsulation to polynomial multiplication, three files are especially important: `kem.c`, `hqc.c`, and `gf2x.c` [PQC25].

3.1.1 KEM functions

The `kem.c` file is the file that contains the three main HQC KEM functions, which are:

- `PQCLEAN_HQC128_CLEAN_crypto_kem_keypair()`
- `PQCLEAN_HQC128_CLEAN_crypto_kem_enc()`
- `PQCLEAN_HQC128_CLEAN_crypto_kem_dec()`

As their name implies, these are the functions that perform key-generation, encapsulation and decapsulation, explained in Section 2.2.1. However, the actual computation of these operations does not happen directly within these functions. The encapsulation and decapsulation functions in `kem.c` call the simpler public-key encryption variants of HQC, but with additional KEM operations. The encapsulation function prepares the message, salt, and randomness used as input to the HQC public-key encryption function. Before calling the decryption function, the decapsulation function divides the ciphertext into its components u , v , and the salt. These parsed values are then passed to the HQC public-key decryption function. The PKE encryption and decryption functions are located in the `hqc.c` file.

3.1.2 HQC PKE functions

Inside the `hqc.c` file, we find the core of the HQC scheme. Here we have the functions:

- `PQCLEAN_HQC128_CLEAN_hqc_pke_encrypt()`
- `PQCLEAN_HQC128_CLEAN_hqc_pke_decrypt()`

These are called from within the `kem.c` file. As explained in Section 2.2.1, both encryption and decryption perform polynomial multiplication over $\mathbb{F}_2$, and for this the function `PQCLEAN_HQC128_CLEAN_vect_mul()` is used, and where we find our target for this thesis.

Listing 3.1. Simplified structure of PQClean's `base_mul` implementation. The helper `equal_mask` returns an all-one mask if its inputs are equal, and zero otherwise.

```

1 base_mul(a, b) {
2     l = 0;
3     h = 0;
4
5     // Step 1: precompute one value for each possible 4-bit value
6     u[0..15] = precompute_values_from_b(b);
7
8     // Process the first nibble separately
9     nibble = get_nibble(a, 0);
10    g = 0;
11    for (i = 0; i < 16; i += 1) {
12        mask = equal_mask(nibble, i);
13        g ^= u[i] & mask;
14    }
15    l = g;
16
17    // Step 2: process the remaining 15 nibbles
18    for (i = 4; i < 64; i += 4) {
19        nibble = get_nibble(a, i);
20        g = 0;
21        for (j = 0; j < 16; j += 1){
22            mask = equal_mask(nibble, j);
23            g ^= u[j] & mask;
24        }
25
26        l ^= g << i;
27        h ^= g >> (64 - i);
28    }
29
30    // Step 3: handle the upper four bits of b
31    handle_upper_bits_of_b(a, b, &l, &h);
32
33    c[0] = l;
34    c[1] = h;
35 }
```

3.1.3 Polynomial multiplication

`gf2x.c` is responsible for performing binary polynomial multiplication over $\mathbb{F}_2$. This is an essential part of HQC and is performed both during encapsulation and decap-

sulation. During decryption, `hqc.c` calls `PQCLEAN_HQC128_CLEAN_vect_mul()` on operands derived from the secret key and the ciphertext. These are passed to the `karatsuba()` function, which is an efficient recursive divide-and-conquer algorithm that multiplies two large numbers [Wei]. The algorithm works by splitting the large numbers into smaller parts over and over, and performing multiplication on much smaller numbers. When the input cannot be reduced any further, a base multiplication function is used. In PQClean’s case, this is the function `base_mul`. The goal of this function is to multiply two 64-bit values. It computes the carry-less multiplication of a and b , and stores the 128-bit product in `c[0]` and `c[1]`. During decapsulation, this function can be called with secret-dependent input, and is therefore relevant for side-channel analysis. In fact, HQC polynomial multiplication has previously been studied as a target for side-channel attacks such as cache-timing attacks and power analysis [RBG+26; VRB26]. A simplified version of PQClean’s `base_mul` can be seen in Listing 3.1. The full `base_mul` code is available in the GitHub repository [PQC25].

The helper `equal_mask` represents the branch-free equality test used in the original implementation. It returns `0xffffffffffffffff` when the current table index matches the processed nibble value, and `0x0000000000000000` otherwise. The inner loop therefore scans all 16 table entries for each nibble, while only the matching entry contributes to `g`. Conceptually, this computes $g = u[\text{nibble}]$, but without using a secret-dependent branch or direct secret-dependent table lookup.

This table selection follows a constant-time programming style. Instead of directly indexing `u` with the secret-dependent nibble, the implementation accesses all 16 entries and uses bit masks to select the required value. This avoids secret-dependent memory addresses and is mainly intended to reduce leakage through cache-timing behavior.

3.2 ChipWhisperer and HQC technical setup

This section describes the setup used to run HQC on the ChipWhisperer platform using a 2021 M1 MacBook Pro. All experiments in this thesis were performed using the ChipWhisperer (CW) framework [New26b]. ChipWhisperer is an open-source project consisting of hardware, firmware, and software components specifically designed for lightweight side-channel analysis. The experiments used a ChipWhisperer-HUSKY capture board together with a SAM4S target board performing the cryptographic operations.

3.2.1 Software and hardware setup

The first step was to clone the ChipWhisperer repository from GitHub [New26a]. This repository also includes the `chipwhisperer-jupyter` repository, which contains

notebooks, courses, and demos for using the hardware and software [New26c].

A Python virtual environment was created in the project folder using:

```
python3 -m venv cw-env
source cw-env/bin/activate
```

Inside the virtual environment, the required packages were installed using the `requirements.txt` file included in the ChipWhisperer repository:

```
pip install -r chipwhisperer/requirements.txt
```

Additionally, the following packages were installed to support the setup on the M1 Mac:

```
brew install libusb
brew install --cask gcc-arm-embedded
```

The ChipWhisperer hardware was connected according to the setup manual included in the ChipWhisperer Husky kit, also available in the official documentation.

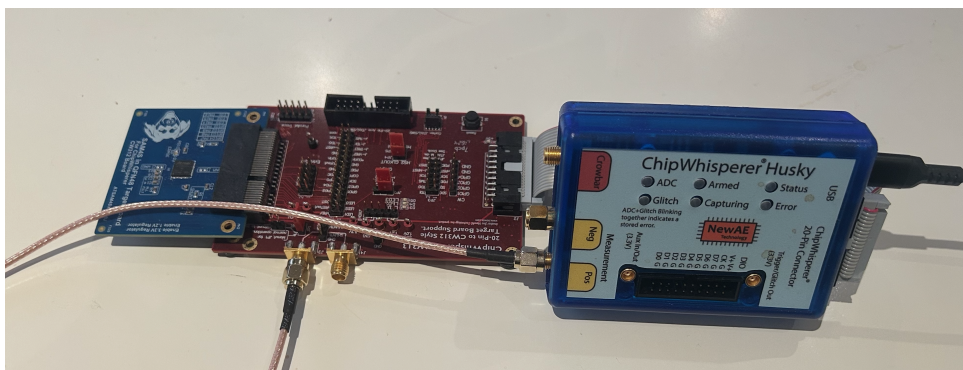


Figure 3.1. ChipWhisperer-HUSKY hardware setup.

To verify that the virtual environment and hardware were configured correctly, the setup notebooks included in the `chipwhisperer-jupyter` repository were run. Introductory AES notebooks were also used to become familiar with the tools.

```
/chipwhisperer/chipwhisperer-jupyter/0 - Introduction to Jupyter
↪ Notebooks.ipynb
/chipwhisperer/chipwhisperer-jupyter/1 - Connecting to Hardware.ipynb
```

3.2.2 HQC integration and firmware adaptation

After verifying that the hardware worked correctly with other cryptographic schemes such as AES, we could begin integrating HQC into the project. For this project we used the PQClean implementation of HQC as it is a reliable and well-structured implementation and is widely used in research [PQC25; KSSW22]. This repository was then placed inside:

```
/Users/haraldpaaske/MasterHQC/chipwhisperer/firmware/mcu/crypto/
```

This allowed HQC to be easily integrated in the same way as the existing cryptographic schemes.

The HQC implementation was then configured so that it could be built and flashed to the ChipWhisperer. This was done by creating the required Makefile integration:

- Create `firmware/mcu/crypto/PQClean/Makefile.inc`
- Add HQC source files to `SRC`
- Add HQC include paths to `EXTRAINC_DIRS`
- Add variant selection via `CRYPTO_OPTIONS = HQC_1 / HQC_3 / HQC_5`
- Edit `firmware/mcu/crypto/Makefile.crypto`
- Add a new `CRYPTO_TARGET = PQCLEAN` case
- Include `crypto/PQClean/Makefile.inc` when `CRYPTO_TARGET = PQCLEAN` is selected
- Ensure AES fallback logic does not override `PQCLEAN` or `HQC` options

A custom SimpleSerial firmware was created to execute and capture a trace of the HQC decryption algorithm on the ChipWhisperer target. This firmware followed the same structure as existing SimpleSerial targets, but replaced the cryptographic core with calls to the PQClean HQC implementation, including the internal HQC PKE decryption routine used by the KEM.

A new directory was created at: `chipwhisperer/firmware/mcu/simpleserial-PQClean/`

The core files in this directory are:

- `simpleserial-PQClean.c`, implementing the SimpleSerial command handlers
- `makefile`, based on the SimpleSerial AES makefile, but targeting PQClean/HQC
- `hqc_vectors.c` and `hqc_vectors.h`, containing fixed test vectors

The firmware uses the SimpleSerial v1.1 protocol and defines command handlers for both HQC decryption and targeted base multiplication routines. The command 'd' invokes HQC PKE decryption using fixed test vectors. The base multiplication commands, such as 'b', '2', 'm', and 's', raise and lower the ChipWhisperer trigger

pin around the computation of interest, allowing the captured traces to be aligned with the targeted operation.

Setup operations such as key generation and ciphertext creation were executed locally once using a host-side helper program. The resulting fixed secret key and ciphertext components were placed in `hqc_vectors.c` to ensure consistency across experiments.

3.2.3 Build, flashing, and capture configuration

The firmware was compiled and flashed with a Jupyter notebook. The firmware was built using:

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET" "$CRYPTO_OPTIONS" "$SS_VER"
cd ../../firmware/mcu/simpleserial-PQClean
make PLATFORM=$1 CRYPTO_TARGET=$2 CRYPTO_OPTIONS=$3 SS_VER=$4
```

This compiled the firmware and produced the firmware image `simpleserial-PQClean-CWHUSKY.hex`, which was flashed to the target using the following command from the ChipWhisperer Python API:

```
cw.program_target(scope, prog, "../../firmware/mcu/simpleserial-PQClean/sim_
↳ pleserial-PQClean-{}.hex".format(PLATFORM))
```

The main power traces were collected using a baseline set of acquisition parameters, with sample window length adjusted for some specific firmware captures. The ChipWhisperer scope was configured with the clock, ADC, gain, and trigger settings shown in Table 3.1. The firmware raises a trigger signal before the targeted computation and lowers it afterwards, allowing the captured traces to be aligned with the execution of the cryptographic operation.

The clock source “system” is to make the target clock originate from the ChipWhisperer-Husky’s internal clock generator. With a low clock frequency at 7 MHz and ADC multiplier 1, the operation takes a longer time, and the differences in the trace are easier to distinguish. The baseline capture configuration used 7500 samples per trace. For different firmware captures this was changed to fit different sample windows. The complete overview of samples per implementation can be seen in Table 4.3. Gain was set to 29 dB which is the highest value found where the ADC did not get clipped, giving the most detail without exceeding the CW’s capabilities. The trigger source “tio4” tells Husky to monitor the tio4 pin for the trigger event in the firmware. The rest of the configuration is set by the `scope.default_setup()` `cw` function.

Parameter	Value
Clock source	system
Clock frequency	7 MHz
ADC multiplier	1
Samples per trace	7500
ADC offset	0
Gain	29 dB
ADC timeout	20 s
Trigger source	tio4

Table 3.1. ChipWhisperer trace capture configuration.

3.2.4 Jupyter workflow

All interaction with the ChipWhisperer scope and target was performed through Jupyter notebooks in the `chipwhisperer-jupyter/hqc` environment. This allowed firmware compilation, flashing, capture configuration, and trace visualization to be handled in one workflow.

3.2.5 Code availability

The code, firmware modifications, notebooks, trace-processing scripts, model-training scripts, and result-processing scripts used for this thesis are available in the project repository below [Paa26]:

<https://github.com/haraldpaaske/MasterHQC>

3.2.6 KEM Demonstration Program

A minimal demonstration program, `hqc/demo_hqc.c`, was written using the `HQC` function from the API file. The program performs key generation, encapsulation, decapsulation, and shared secret verification. This was done to verify that key generation and decryption produced the same result locally and on the ChipWhisperer, and confirm that the implementation was correct.

The demo was compiled and linked against the `HQC-128` reference library:

```
clang demo_hqc.c \
-I src/common \
-I src/ref \
build-ref/src/libhqc_1_ref.a \
```

```
build-ref/lib/libfips202.a \  
-o demo_hqc
```

3.2.7 Implementation validation

The successful execution of the demo program confirmed correct HQC KEM operation on the target platform, providing a validated software baseline for future embedded and side-channel experimentation. This also confirmed that the HQC implementation executed successfully, as the same keys, ciphertexts, and message were produced by both the demo program and the implementation on the ChipWhisperer.

Chapter 4

Methodology

The methodology was designed to answer research questions RQ2, RQ3, and RQ4 from Chapter 1. The experiments evaluate recoverable secret-dependent information from isolated executions of the PQClean implementation of `base_mul` on a ChipWhisperer-HUSKY [New26b], and compare how masking, shuffling, and `base_mul2` affect the observed leakage under the same general profiling setup.

The methodology consists of three main stages. First, the experiment was narrowed to the polynomial multiplication function `base_mul` in `gf2x.c`, which has previously been targeted for side-channel analysis [VRB26; RBG+26]. Second, two profiling models were trained to classify nibble values based on cycles cropped from `base_mul` trace segments. Finally, the same general capture and profiling approach was applied to protected variants where possible, with adapted segmentation and binary classification for `base_mul2`, as proposed by Velek et al. [VRB26].

4.1 Target operation

The attack targets the `base_mul` function in `gf2x.c`, which is used during polynomial multiplication in the PQClean implementation of HQC. This function processes a 64-bit input limb four bits at a time. Each four-bit window, or nibble, determines which value is selected from a precomputed lookup-table. A nibble can take one of 16 hexadecimal values, $\{0, \dots, f\}$, corresponding to the decimal values 0 to 15. Since the selected table value depends on the processed secret-dependent limb, the operation is a candidate for power side-channel leakage.

The classification task used in this thesis is therefore defined as follows: given a trace segment corresponding to one nibble-processing cycle in `base_mul`, predict which of the 16 possible nibble values produced that segment. Correct classification of all 16 nibble positions corresponds to recovery of one 64-bit secret-dependent limb.

4.2 Methodological strategy

This section describes the strategy used for performing a power side-channel analysis of the PQClean implementation of HQC on a ChipWhisperer-HUSKY. The goal of the strategy is to establish a clear experimental path from initial leakage observation, through controlled trace collection, to classification of secret-dependent values and evaluation of countermeasures. The strategy begins after the technical setup from Section 3.2 has been completed.

The attack is inspired by the work of Velek et al., but uses a different technical setup. Velek et al. [VRB26] target the *Additional implementation* included in the NIST Round 4 submission, which is also included in PQClean [PQC25]. More specifically, they target the nested loop inside the `base_mul` function in `gf2x.c`, which is used during polynomial multiplication. Their experiments were performed using a ChipWhisperer-Lite board with an STM32F303 microcontroller, while this thesis uses a ChipWhisperer-HUSKY with a SAM4S target. Velek et al. show that this implementation can leak secret-dependent values during the decapsulation process.

The strategy for the methodology follows from this target operation. Instead of attempting to analyze a complete decapsulation trace of HQC directly, the experiments first isolates `base_mul` so that the processed values are known and segments in the trace can be labeled. This allows the potential leakage to be studied under controlled conditions before the same approach is applied to protected variants.

The methodology follows three main stages. First, the trace capture is narrowed to isolated executions of `base_mul`. This allows individual traces to be associated with known secret-dependent limb values and makes it possible to label each nibble-processing cycle. Second, the labeled traces are used to train profiling classifiers that distinguish nibble values from cropped trace segments. If the nibble values can be classified reliably, the same method can be applied to all 16 cycles in one `base_mul` trace in order to reconstruct a complete 64-bit limb.

The final stage evaluates countermeasures. In addition to testing the original PQClean polynomial multiplication for side-channel leakage, three protected variants are considered: masking, shuffling, and the alternative multiplication implementation proposed by Velek et al. [VRB26]. The same general trace-capture, segmentation, and classification strategy is applied to these variants where possible. The goal is to determine whether the countermeasures remove or reduce the exploitable relationship between the measured power trace and the processed secret-dependent values.

4.3 Experimental procedure

This section describes how the methodology was implemented experimentally, from the exploratory trace comparison to isolated `base_mul` capture, trace segmentation, classifier training, and final evaluation.

4.3.1 Exploratory trace comparison

Before isolating a specific operation, an exploratory trace comparison was performed to determine whether changing the secret-dependent limb processed by the `base_mul` function produced measurable differences in the power trace. This experiment was not intended as a formal leakage assessment, but as an initial proof of concept before moving to a more controlled analysis of `base_mul`. The capture window consisted of 11000 samples recorded from a trigger placed immediately before a selected `base_mul` call inside the HQC decryption process.

Four trace sets were collected using a fixed ciphertext `u_const` and `v_const`. Two sets were captured with the original secret key `sk_const`, and two sets where the secret-dependent key limb passed to `base_mul` was set to `0xffffffffffffffffULL`. To capture these sets, the `karatsuba` function in `gf2x.c` was modified to call `base_mul` with different parameters. The modified part of `karatsuba` is shown in Listing 4.1. Each set contained 200 traces. The traces in each set were averaged sample by sample, producing one single average trace per set. The same-condition comparisons were used as a baseline for normal measurement variation, while the different-condition comparisons were used to check whether the modified key-limb produced a larger measurable difference. The two sets captured under the modified condition were also compared to see if the modification produced a consistent difference, or just introduced more noise.

Since this experiment only considered a mean difference between averaged traces, it does not tell us anything about the values used, the result was used only to motivate the next step of the methodology. The main experiments therefore proceeds by directly targeting the `base_mul` function outside of HQC, where noise from other operations are removed.

Listing 4.1. Simplified trigger structure used for isolated `base_mul` captures.

```

1  uint64_t aa = a[0];
2  uint64_t bb = b[0];
3
4  aa = 0xffffffffffffffffULL;
5
6  trigger_high();
7  base_mul(aa, bb);
8  trigger_low();

```

4.3.2 Isolating `base_mul`

As discussed in Section 4.2, the attack is inspired by Velek et al. [VRB26], who identify the `base_mul` function used inside `vect_mul` as a vulnerable operation. To obtain accurately labeled traces, the experiment was narrowed from the decapsulation command to a single execution of `base_mul`. A new `SimpleSerial` command was added to the ChipWhisperers firmware, where the `base_mul` code from `gf2x.c` was executed inside a trigger window. The trigger was raised immediately before the lookup-table processing and lowered after the 16 nibble-processing cycles had completed. A simplified version of the firmware running `base_mul` is shown in Listing 4.2.

Listing 4.2. Simplified trigger structure used for isolated `base_mul` captures.

```

1  set_input_limb(a);
2  set_input_limb(b);
3
4  trigger_high();
5  base_mul(a, b);
6  trigger_low();
7
8  return_result(c);

```

This isolated setup made it possible to control the 64-bit input limb and associate each visible cycle in the trace with a known nibble value. The values processed by `base_mul` were set through `SimpleSerial` commands, allowing different combinations to be tested in a controlled way. The variable *a* represents the secret-dependent 64-bit key-limb being processed, while *b* represents the corresponding input limb from the other polynomial corresponding to the ciphertext. In the isolated profiling experiments, *b* was fixed to the value `0xffffffffffffffff` for the standard, masked, shuffled, and `base_mul2` implementations, while *a* was for the most part random, but known.

After isolating `base_mul`, 200 traces of the same operation with the same a value were first averaged and combined into a single trace. This was done to reduce random noise from the capture process. The averaged trace was then plotted and inspected visually to verify that the captured window contained the expected repeated structure corresponding to the nibble-processing cycles. After close inspection, the first nibble-processing event had a slightly different structure from the remaining events, while the following 15 events formed a regular repeated pattern. This observation motivated treating nibble position 0 as a separate classification task from nibble positions 1–15.

To verify that the visible structure corresponded to the implementation loops, the `base_mul` code was temporarily modified. The number of outer-loop iterations in Listing 3.1 was reduced from 64 to 32 to check whether the trace changed from 16 cycles to 8 cycles. Similarly, the inner loop was changed from 16 to 8 iterations to check whether the number of spikes inside each cycle changed. This would confirm that the visible trace structure followed the nested-loop structure of the implementation.

4.3.3 Classification models

Now that isolated `base_mul` traces can be inspected and the processed secret-dependent value is known, the next step is to infer which four bit nibble value was processed in each cycle. The classification problem could therefore be defined as the following: given a trace segment corresponding to one nibble-processing cycle in `base_mul`, predict which nibble value in $\{0, \dots, f\}$ produced that segment.

In order to accurately identify what nibble value was for any cycle, a more reliable method than manual inspection was needed. Although some cycle-patterns were consistent with the nibble values, it was generally hard to determine value from trace visually. This motivated the use of a profile-based classifier model that was trained on labeled trace segments. Two profiling classifiers were tested. Both used the same segmented traces, alignment procedure, point-of-interest (POI) selection, and feature extraction [CRR03]. The only difference was in the final classification step. The first model used nearest-centroid classification, while the second used multinomial logistic regression [HTF09].

4.3.4 Model 1: Nearest-centroid classification model

The first model for this task was a classic profiling model. The idea was to feed the model a large set of `base_mul` traces that had been carefully cut into single cycles, each with a corresponding nibble value. From this, the model builds 16 profiles based on similarities between traces created with the same nibble value. The proposed approach uses a centroid representation in a reduced feature space, which is defined by selected points of interest (POIs). In this approach, each nibble class is

represented by the mean feature vector of that class, and classification is performed by assigning an unknown trace to the class with the smallest Euclidean distance to its centroid. In order to get reliable profiles with distinct features, a large dataset had to be created in order to identify similarities and create the profiles. The classifying pipeline is inspired by the standard centroid-based classification described by Hastie et al. [HTF09].

Step 1: Dataset construction and preprocessing

First we need a large dataset with `base_mul` traces for the model to learn the patterns of which traces correspond to which nibble values. This dataset was captured using the same firmware as before, but with random secret-dependent key-limbs, so the model will be tuned on data as close to real values as possible. The profiling dataset consisted primarily of 400 random key-limb files, with 200 individually captured traces each. In addition, 16 fixed-value key-limb files with 50 traces each were captured to be sure there was at least one representation of each nibble value at every possible position. The 416 files were stored as a NPZ files at `chipwhisperer-jupyter/hqc/VERSION2/base\_mul\_traces\_random_v1`, each with shape (200, 7500), 200 traces of 7500 samples per trace. The process of capturing all 80,800 traces with the ChipWhisperer took around 13 hours. The notebook containing the Python code for capturing the traces is at `chipwhisperer-jupyter/hqc/VERSION2/capture\_traces.ipynb` [Paa26].

After all full `base_mul` traces were captured, each trace was divided into 16 segments corresponding to the 16 cycles, one dedicated first event corresponding to nibble position 0, followed by 15 repeated events corresponding to nibble positions 1 to 15. To determine suitable crop boundaries, the 200 traces from a representative key-limb file were averaged to obtain a stable reference trace. The crop parameters were tested and tuned by inspecting the trace, and aligning each cycle as seen in Figure 4.2, then the final values were chosen by visual inspection so that the first event and the repeated events were isolated as accurately as possible. This approximate method could be used because all the traces were similar in shape, and thus only choosing the crop parameters for one key could transfer to all other traces. Additionally some margin samples were included outside of the specified crop-boundaries to ensure the entire cycle corresponding to the nibble value were included in the segment, the parameter `REPEATED_EXTRACT_LEN` specified the total samples in each segment. A visualization of how the crop-alignment was chosen is shown in Figure 4.1. The values used for cropping are listed in Table 4.1. For the cycles 1 to 15, a coarse cycle grid was first defined, after which each extracted repeated segment was shifted backwards by a small fixed number of samples in order to include limited pre-cycle context. Because the first cycle was processed by a different part of the code, it does not have the same dimensions as the rest of the

cycles. For this reason, the first event was treated as a separate classification task, while the remaining 15 repeated events were grouped into a second classification task.

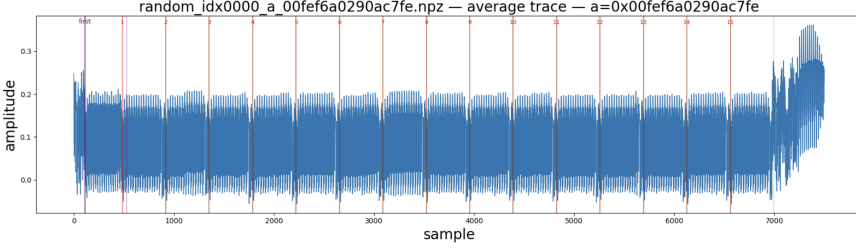


Figure 4.1. The 16 cropping boundaries applied to a `base_mul` power trace. The boundaries are the beginning and end of one segment.

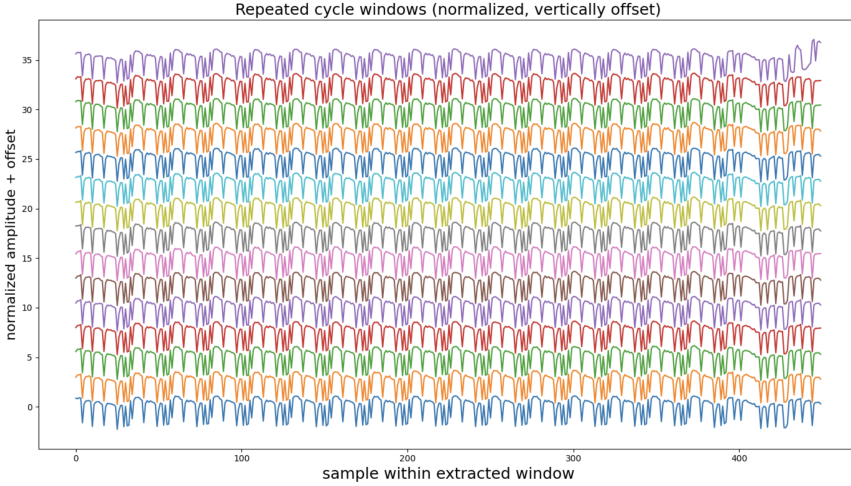


Figure 4.2. Example of the aligned cropped segments with length `REPEATED_CYCLE_LEN = 434` applied to a `base_mul` power trace. Used to find correct segment length.

Once the parameters were determined, every single trace was cropped according to the parameters defined, and saved in a folder named `profiles_randomword_v1`. The label for each cropped segment and nibble-value was obtained from the metadata associated with the source key limb. The cropped traces were sorted into two main sets `_test` and `_train`, where one was used for training the model, so it learns the decisive patterns from the trace. The second was used for testing the accuracy of the model. The split was performed at the full-file level using a 75/25 division, meaning that all traces from a given key-limb file were assigned either to

Table 4.1. Cropping parameters used for trace segmentation.

Parameter	Description	Value
FIRST_EVENT_START	Start index of the first-event crop	110
FIRST_EVENT_LEN	Length of the first-event crop	420
REPEATED_FIRST_START	Start index of the first repeated cycle	495
REPEATED_CYCLE_LEN	Length of one repeated cycle	434
REPEATED_PRE_SAMPLES	Extra samples included before each cycle	15
REPEATED_EXTRACT_LEN	Total extracted length per repeated cycle	450
N_REPEATED_CYCLES	Number of repeated cycles extracted	15

training or to testing, but never to both. Additionally the traces were sorted into `special_nibble0` and `repeated_nibble1to15` distinguished between the first nibble, which was different than the rest, because these are structurally different and are for two different models. In the repeated dataset, all repeated-position segments with the same nibble value were pooled into the same class bucket regardless of whether they originated from nibble position 1, 2, ..., or 15. The notebook containing the Python code for segmenting the traces is at `chipwhisperer-jupyter/hqc/VERSION2/split\_traces.ipynb` [Paa26].

Step 2: Alignment and point-of-interest selection

Once the test and training datasets were complete, we could begin to process the traces for the classification task. This was done through a rigorous process handling the traces, defining POIs and patterns, ultimately constructing a decision model based on the distinct features shared between traces labeled with the same nibble value.

Before addressing POIs and finding patterns, the cropped traces needed to be aligned. Even when the traces come from the same key-limb, there is some slight variation and misalignment from trace to trace. To ensure consistent alignment a template was used for reference on all traces. The template was made by averaging all cropped traces in the training data, then aligning every cropped trace to this template, then again averaging the aligned traces for a final consistent and accurate template. Next, each cropped trace was then aligned by maximizing its similarity to the template within a bounded shift interval specified by `_ALIGN_MAX_SHIFT_CANDIDATES`. Let x denote a cropped trace segment and t the alignment template. The aligned trace $\hat{x}$ is obtained by selecting the shift s within the range specified by `_ALIGN_MAX_SHIFT_CANDIDATES` that maximizes the dot-product similarity between the normalized trace and the normalized template.

$$\hat{s} = \arg \max_{s \in [-S, S]} \langle \tilde{x}_s, \tilde{t} \rangle$$

where $\tilde{x}_s$ denotes the normalized trace shifted by s , $\tilde{t}$ denotes the normalized template, and S is the maximum allowed shift.

After alignment, a global mean was subtracted from all traces in order to remove trace information shared across all traces. The goal is to obtain a residual trace that contained only the most class-dependent features for each trace, to further emphasize POIs. the residual trace is defined as:

$$r = \hat{x} - \mu$$

where r is the residual trace, $\hat{x}$ is the aligned trace, and μ is the sample-wise mean of all aligned training traces.

To reduce both the size and noise of the traces, and focus only on the most informative regions of each trace, a set of points of interest (POIs) was selected from the residual traces [RWTM24; PHJB19]. The selection was based on a signal-to-noise (SNR) like criterion inspired by SNR-based POI selection methods used in profiled side-channel analysis [RWTM24], where the variance between class means were compared to the average within-class variance:

$$\text{score}(i) = \frac{\text{Var}(\{\mu_c(i)\}_c)}{\mathbb{E}_c[\text{Var}(r_c(i))]},$$

where $\mu_c(i)$ denotes the mean residual value at sample i for class c , and $r_c(i)$ denotes the set of residual values at sample i for traces belonging to class c .

This score was used as an SNR-like ranking rule for identifying informative sample points in the trace segment.

The top- k sample positions with the highest scores were selected as POIs, subject to a minimum spacing constraint to ensure no POIs are very close together [RWTM24]. The number of POIs to be selected is denoted by k . In the implementation, possible values for k were specified by `_POI_COUNT_CANDIDATES`. A minimum spacing constraint was used to avoid selecting several highly correlated sample points from the same narrow region of the trace.

Step 3: Feature extraction

Once the POIs are selected, a feature vector can be constructed from the traces by extracting a small local neighborhood around each POI from the residual trace r . For a POI position p_j and radius ρ , the samples in the interval $p_j - \rho$ to $p_j + \rho$ were included. The local neighborhoods from all selected POIs were then concatenated into a single feature vector $\mathbf{f}$:

$$\mathbf{f} = [r(p_1 - \rho : p_1 + \rho), \dots, r(p_k - \rho : p_k + \rho)].$$

Here, $r(p_j - \rho : p_j + \rho)$ is used as interval notation and denotes the sequence of residual-trace samples from index $p_j - \rho$ to index $p_j + \rho$ around POI p_j .

This produced a reduced representation of each trace segment that preserved only the most informative local regions and a smaller size. In the python code, the radius ρ is denoted by the variable `_POI_RADIUS_CANDIDATES`.

After feature extraction, all the feature vectors were standardized using the mean and standard deviation computed from the training set:

$$\mathbf{f}' = \frac{\mathbf{f} - \boldsymbol{\mu}_f}{\boldsymbol{\sigma}_f}.$$

Here, $\boldsymbol{\mu}_f$ and $\boldsymbol{\sigma}_f$ are the feature-wise mean and standard deviation computed from the training feature vectors. The same values were used to standardize the test feature vectors.

This is to ensure that all feature dimensions contributed on a comparable scale during classification, so that features with larger numerical variation do not dominate the distance calculation simply because of their scale.

Step 4: Nearest-centroid classification

Next, a centroid-based classifier was constructed to represent the distinctive feature patterns associated with each nibble value. For each class c , consisting of traces generated by the same nibble value, the centroid was computed as the mean standardized feature vector of that class:

$$\bar{\mathbf{f}}'_c = \frac{1}{N_c} \sum_{i=1}^{N_c} \mathbf{f}'^{(c)}_i.$$

Given an unknown test feature vector $\mathbf{f}'$, classification was performed by computing its Euclidean distance to each class centroid and selecting the nearest class:

$$\hat{c} = \arg \min_c \|\mathbf{f}' - \bar{\mathbf{f}}'_c\|_2.$$

In addition to the predicted class, the full ranking of class distances was retained. This made it possible to inspect the two best candidate classes, measure the classification margin between them, and compute top-2 accuracy. The model was built with the `scikit-learn` framework, and can be found in the github repository at `Nearest_Centroid/train_model.ipynb` [sci26b; Paa26].

Step 5: Model selection and evaluation

The model parameters were selected through an iteration process using an inner validation split on the training dataset. For each classification task, the available

training traces were divided into an inner training subset and an inner validation subset. A grid search was then performed over all parameters displayed in Table 4.2, which includes further crop boundaries on the segmented traces, number of POIs, minimum POI spacing, feature radius, whether to remove the global mean, and the maximum alignment shift.

For each possible parameter combination, a model was trained as explained from step II through step IV on one part of the training data and tested on another part of the training data reserved for this purpose. The top-1 and top-2 accuracy was printed out for every model and combination of parameters. The final model selection criterion prioritized top-1 accuracy, while top-2 accuracy and mean margin were also taken into account when choosing the most accurate model.

After the best parameter configuration had been found, the final model was trained using the full training dataset and evaluated on the independent test dataset. This procedure was performed separately for the `special_nibble0` model and the `repeated_nibble1to15` model. The final trained models, together with their selected parameters, alignment templates, POIs, feature-normalization statistics, and class centroids, were then exported and saved at `VERSION2/trained_models` as compressed model bundles for later use.

Table 4.2. Hyperparameter search space used for model selection.

Parameter	Special model	Repeated model
CROP_CANDIDATES	(0,420), (0,400), (10,410)	(5,445), (0,450)
POI_COUNT_CANDIDATES	48, 64, 80	96, 128
POI_MIN_SPACING_CANDIDATES	2, 3, 4	4, 5
POI_RADIUS_CANDIDATES	2, 3	2
USE_DX_CANDIDATES	False	False
REMOVE_GLOBAL_MEAN_CANDIDATES	True	True
ALIGN_MAX_SHIFT_CANDIDATES	0	0, 6
Total combinations	54	16

Step 6: Key-limb Classification

Once the model had been tested and shown to achieve sufficient classification accuracy, it was used for full 64-bit limb prediction test. This was done by capturing and parsing a full `base_mul` trace to the model. The trace was then automatically segmented and ran through the model, resulting in a final prediction of the 64 bit key limb used to create that `base_mul` trace.

4.3.5 Model 2: Logistic-regression model

While the nearest-centroid classification model was simple and gave a reasonable result, an additional model was tried with a different classification approach in order to see if we could improve classification accuracy. For this a supervised machine learning model was used.

For this model, most of the pipeline from the Nearest-centroid classification model was reused, and only the actual classification step was changed. This made both the comparison of the accuracy of the models and development of the new model simple. This means the datasets used for training and testing both models were the same, the alignment process and POI selection was the same and the feature extraction was the same as in the Nearest-centroid classification model and explained in Section 4.3.4 part I-III.

The machine learning model worked by first processing the data as described in Section 4.3.4, but instead of classifying the segments by finding the class centroid with shortest distance, it uses a multinomial logistic-regression classifier that was trained on the extracted feature vectors [HTF09]. Also similar to the first model, the first cycle and the repeated cycles were treated as two separate tasks, giving one model for `special_nibble0` and one model for `repeated_nibble1to15`.

Multinomial logistic-regression

The logistic-regression model used the same segmentation, alignment, POI selection, feature extraction, and standardization procedure as described for the nearest-centroid model. Instead of classifying a feature vector by its distance to the nearest class centroid, this model uses multinomial logistic regression. This procedure is inspired by Hastie et al. [HTF09]. For each class c , it computes a linear score:

$$s_c(\mathbf{f}') = \mathbf{w}_c^\top \mathbf{f}' + b_c,$$

Here, $\mathbf{w}_c$ is the learned weight vector for class c , and b_c is the corresponding bias term. The scores are then converted into class probabilities using the softmax function:

$$P(y = c \mid \mathbf{f}') = \frac{\exp(s_c(\mathbf{f}'))}{\sum_{c'} \exp(s_{c'}(\mathbf{f}'))}.$$

The softmax function normalizes the class scores so that the resulting values can be interpreted as probabilities over the possible nibble classes. The predicted nibble value is then selected as the class with the highest estimated probability:

$$\hat{c} = \arg \max_c P(y = c \mid \mathbf{f}').$$

As mentioned, the full key-limb identifier consists of two separately trained models, `special_nibble0` and `repeated_nibble1to15`. Unlike the nearest-centroid model,

the logistic-regression model was not tuned using a separate grid search. Instead, the hyperparameters selected for the nearest-centroid model were reused, under the assumption that the same preprocessing, POI selection, and feature-extraction settings would also provide suitable input features for logistic regression.

The logistic-regression model did not receive a separate hyperparameter search. It reused the preprocessing parameters selected for the nearest-centroid model. The comparison therefore evaluates the effect of replacing the final classifier under the same feature-extraction pipeline, rather than fully optimizing both classifiers independently. The model was built with the `scikit-learn` framework, and can be found in the github repository at `Logistic_Regression/train_model.ipynb` [sci26a; Paa26].

After training, the models were evaluated on the independent test set using the same procedure as for Model 1. The final models were then saved as compressed bundles containing the preprocessing information together with the learned logistic regression coefficients, intercept terms, and class ordering. These saved models could then be applied to unseen `base_mul` traces by preprocessing each segment in the same way, predicting the nibble value for each sub-iteration, and combining the predicted nibbles into a full 64-bit limb.

4.3.6 Comparing models

To evaluate the two models' accuracy, a new notebook called `benchmark.ipynb` was created for the benchmark tests. This notebook performed a set of classification tests, measuring how each model scales with the number of averaged traces, how the accuracy changes for each nibble position, and how many traces are needed to recover a full key limb correctly. Although it is interesting to see how the models compare, the most important part of the benchmarking and testing is to see how reliable the leakage of sensitive information actually is, and if the leakage is sufficient enough to be applicable for attacking the cryptographic scheme.

4.4 Countermeasure implementations

Three different countermeasures were tested with the goal of mitigating side-channel leakage from the polynomial multiplication function `base_mul`. Two of these are classic side-channel countermeasures, namely masking and shuffling, while the third is an alternative multiplication implementation proposed by Velek et al. for this specific polynomial multiplication function [VRB26]. The purpose of these additional implementations was to observe whether common mitigation strategies reduce exploitable secret-dependent information, either by randomizing intermediate values in the computation, randomizing the order of computations, or removing the lookup-table structure targeted in the original attack. The masked and shuffled

implementations are developed as variants used to study how these countermeasure principles affect the observed leakage, rather than as complete or validated secure implementations.

The same general methodology was applied to the protected implementations in order to make the comparison as consistent as possible. For each implementation, a new profiling dataset was collected, segmented, labeled, split into training and test sets, and evaluated using the logistic-regression profiling pipeline described in Section 4.3.5. The number of key-limb files, traces per file, and file-level split strategy was kept the same where possible. The segmentation parameters were changed when necessary because the protected implementations had different execution lengths and trace structures. The number of captured ADC samples was also adjusted to fit each implementation, as shown in Table 4.3. The effectiveness of the countermeasure implementations was then assessed by comparing the protected variants against the unprotected `base_mul` implementation.

For simplicity, and to avoid tuning the attack specifically to each countermeasure, the hyperparameter configuration selected on the unprotected `base_mul` profiling data was reused for both the masked and shuffled implementations. This was done because the trace structures were similar and because the hyperparameter search was computationally expensive. As a result, the models trained on the protected implementations may not be fully optimized for those implementations.

Table 4.3. ADC sample settings used for each implementation.

Implementation	<code>scope.adc.samples</code>
Unprotected <code>base_mul</code>	7500
Masked <code>base_mul</code>	9000
Shuffled <code>base_mul</code>	12000
<code>base_mul2</code> (BM2)	2700

4.4.1 Masked lookup-table implementation

The first countermeasure that was tried as an attempt to mitigate the side-channel information leakage was implementation of a masked lookup-table [CS20]. This implementation still had the 16-entry loop for each nibble, but the secret-dependent lookup value was masked. The purpose of testing this implementation is to see how effective the practice of masking is at this exact stage in the HQC algorithm. For the experiment the same split-trace-ML-model pipeline is used with an attempt to identify the secret values used in the masked lookup-table `base_mul` version.

The original `base_mul`, which performs a multiplication of a and b over GF2, processes four bits (a nibble) of the 64-bit key-limb a at a time. Each nibble

determines which value from the lookup-table that is accessed. The values in the lookup-table are precomputed multiples of the ciphertext limb b . In the masked implementation, the table is first built as normal, for each cycle of accessing the lookup-table, a random input mask and output mask is applied. The input mask hides which entry in the table is accessed, and the output mask hides the actual value from the table.

A simplified version of the firmware function `base_mul_masked` is shown in Listing 4.3. Initially the table is randomised outside the capture window, this is because this is not the operation we compare to the original `base_mul`, but rather the actual query to the table. For each four-bit window w , a random value δ_w is used to permute the table indices, and a random value m_w is XORed into every table entry. Then inside the capture window (after `trigger_high()`), the secret nibble from a is XORed with δ_w before the query is performed. This means that the selected table index changes randomly, even when the underlying secret nibble is the same.

The function `accumulate_result(g, off)` represents the same shift-and-XOR accumulation used in the original implementation. The function `remove_accumulated_masks()` represents the correction step that removes the accumulated output masks after the trigger window.

The goal with this implementation is that the value accumulated during the capture is no longer the actual lookup-table value g , but rather the masked value $g \oplus m$, where m is a random mask. Finally, the mask is XORed back to recover the correct value, since $(g \oplus m) \oplus m = g$. This preserves the correctness of the computation.

Since the computational structure of the masked implementation is similar to the original, the very same procedure used in Section 4.3.5 was reused with new testing and training data. After collecting the same 416 times 200 traces from `base_mul_masked` with known key-limb values, they were aligned and split into 16 segments. As a result of the extra computation of the masking operations each segment was found to be 515 (510 for nibble 0) samples in length compared to 434 from the original. As before, the first nibble is cropped and handled separately because the structure of this segment is slightly different. Each segment was labeled with the known nibble value processed in that trace-segment, and the labeled segments were then split into training and test sets. Next was the training procedure, each training segment was aligned to a shared template, transformed into residual traces, and reduced to POI-based feature vectors. A logistic-regression classifier was trained on these feature vectors in the same way as the unmasked implementation. The classification of the masked traces were the same 16-class problem as the unmasked problem, and the results could be compared directly to see if the countermeasure

had helped.

This experiment was used to test whether the original profiling attack could detect nibble-dependent leakage after masking was introduced. If the masking worked as intended, the amount of correct predicted nibble and the measured trace should be weakened and ideally 1/16 or 6.25%, making the classifier perform close to random guessing for a 16-class problem.

Listing 4.3. Simplified masked lookup-table implementation `base_mul_masked`.

```

1  base_mul_masked(a, b) {
2      build_u_table(u, b);
3
4      // Precompute one masked table for each 4-bit window
5      for (w = 0; j < 16; w += 1) {
6          delta[w] = random_nibble();
7          mask[w] = random_word();
8
9          for (j = 0 to 16 step 1) {
10             masked_u[w][j] = u[j ^ delta[w]] ^ mask[w];
11         }
12     }
13
14     trigger_high();
15
16     // Process the 64-bit input a four bits at a time
17     for (w = 0; j < 16; w += 1) {
18         off = 4 * w;
19
20         masked_nibble = get_nibble(a, off) ^ delta[w];
21         g = masked_lookup(masked_u[w], masked_nibble);
22
23         accumulate_result(g, off);
24     }
25
26     trigger_low();
27
28     remove_accumulated_masks();
29     store_result(c);
30 }
```

4.4.2 Shuffled implementation

The second implementation that was tested used shuffled nibble processing. The function `shuffle16` implements the Durstenfeld in-place version of the Fisher–Yates

shuffle, where each iteration selects $j \in [0, i]$ and swaps the elements at positions i and j [Dur64; HGK+15]. This shuffles the order in which the 16 nibble positions are processed. The goal is to mix the cycles visible in the trace, so that the chronological order of the cycles no longer directly corresponds to the order of the nibbles in the key limb. In essence, `base_mul_shuffled` from Listing 4.4 works very much like the original implementation, but the four-bit nibble lookup happens in a random order instead of a fixed sequential order.

Listing 4.4. Simplified shuffled lookup-table implementation
`base_mul_shuffled`.

```

1  shuffle16(perm) {
2      perm = [0, 1, 2, ..., 15];
3
4      for (i = 15; i > 0; i -= 1) {
5          j = random_index(0, i);
6          swap(perm[i], perm[j]);
7      }
8  }
9
10 base_mul_shuffled(a, b) {
11     build_u_table(u, b);
12     shuffle16(perm);
13
14     trigger_high();
15
16     // Process the 16 nibbles of a in shuffled order
17     for (k = 0; j < 16; k += 1) {
18         w = perm[k];
19         off = 4 * w;
20
21         nibble = get_nibble(a, off);
22         g = lookup_constant_time(u, nibble);
23
24         accumulate_result(g, off);
25     }
26
27     trigger_low();
28     store_result(c);
29 }
```

The function `accumulate_result(g, off)` represents the same shift-and-XOR accumulation used in the original implementation, but the nibble positions are processed in the randomized order given by `perm`.

The goal with this implementation is that the order of identifiable cycles is no longer the actual order of the key-limb, so even if we could identify the correct nibble value, we do not know where in the sequence it goes. This means that although the same underlying computation is performed, the order of the key-dependent operations is randomized. The same general profiling pipeline from Section 4.3.5 could therefore be reused.

As before, this also used the same profiling pipeline. This includes collecting 400 times 200 traces from `base_mul_shuffled` with known key-limb values, aligning and splitting into 16 cycle-segments. Similar to the masked version, the additional computation made each cycle more computationally expensive and therefore required more samples. Each segment was found to be 685 (710 for nibble 0) samples in length. Everything else was also kept the same for this processing.

This experiment was used to test whether the profiling attack could still detect nibble-dependent leakage when the order of operations was randomized. Unlike masking, shuffling does not necessarily remove leakage from each individual lookup operation. A classifier may still be able to identify the nibble value processed in a given cycle. The intended protection is instead that the temporal order of the cycles no longer directly corresponds to the order of the nibbles in the key limb. Therefore, the shuffled implementation was evaluated both in terms of per-segment classification accuracy and in terms of whether the predicted values could still be placed in the correct limb positions.

4.4.3 Alternative protected implementation

Velek et al. [VRB26] also identify the `base_mul` function in PQClean’s HQC implementation as vulnerable to power analysis. This paper demonstrates a simple power analysis attack similar to the one presented in this section. In addition, they introduce a solution to the leakage they argue originates from the lookup table in the second step on the algorithm. The solution, named `base_mul2` removes the look-up table, and by this removing the leaking 16-cycle-pattern formed by the query to the look-up table. Instead of the look-up table, `base_mul2` processes each 64-bit key-limb bit by bit. For each bit of the secret-dependent value, a mask is computed from the bit value of the key-limb a and used to conditionally XOR shifted versions the ciphertext b into the result. A simplified version of the idea is shown in Listing 4.5.

Listing 4.5. Alternative multiplication implementation `base_mul2`, based on Velek et al. [VRB26].

```

1 base_mul2(a, b) {
2     mask = -(a & 1);
3     l = b & mask;
4     h = 0;
5
6     // Process the remaining 63 bits of a one bit at a time
7     for (i = 1; i < 64; i += 1) {
8         mask = -get_bit(a, i);
9         l ^= (b << i) & mask;
10        h ^= (b >> (64 - i)) & mask;
11    }
12
13    c[0] = l;
14    c[1] = h;
15 }
```

The important distinction between `base_mul2` and the original `base_mul` is that `base_mul2` removes the 16-entry lookup-table and the nested loop. This means the clear 16-cycle pattern corresponding to the 16 nibbles of the secret limb is no longer visible in the power trace. Instead, the trace now shows a longer sequence of smaller repeating patterns corresponding to each individual bit of the 64-bit limb.

Because of this change in trace structure, the same splitting and training procedure could not be applied as before. However not much change had to be done either. Instead of 16 cycles, we first tested 64 cycles as inferred by the firmware, but could not consistently identify 64 identical repeating cycles, and therefore shortened it to 63, still keeping the event-0 separate from the rest for simplicity of development. For this test, excluding one bit will still give valuable information about the potential leakage of the other 63 bits. The parameters such as `REPEATED_CYCLE_LEN` and `REPEATED_FIRST_START` were selected when the normalized cropped cycles in Figure 4.3 were aligned. This resulted in the segmentation seen in Figure 4.4. Each segment was then given a binary label, either 0 or 1, depending on the corresponding bit of the secret-dependent limb.

Additionally, because of the large difference in trace structure, parameters such as number of POIs and POI-radius were adapted to better fit with the shorter segments. For this, the grid search explained in Section 4.3.4 was performed with the trace profiles from `base_mul2`. The best configuration was then used to train the multinomial logistic-regression model.

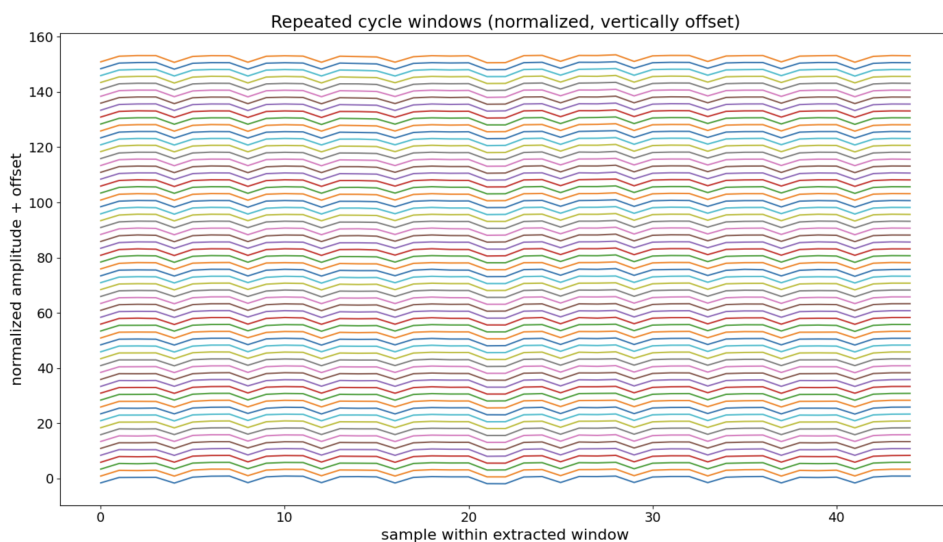


Figure 4.3. Example of the aligned cropped segments with length REPEATED_CYCLE_LEN = 37 applied to a base_mul2 power trace. Used to find correct segment length.

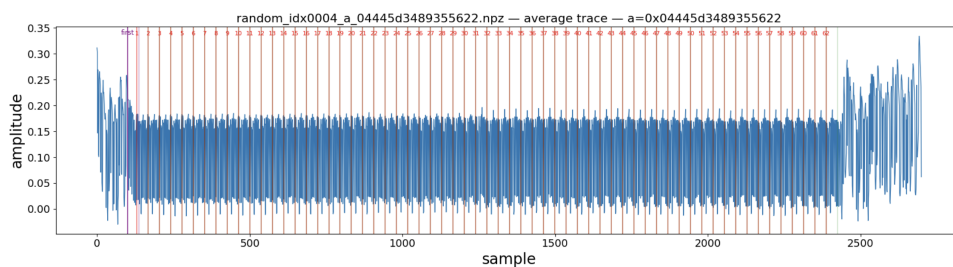


Figure 4.4. The 63 cropping boundaries applied to a `base_mul2` power trace. The boundaries mark the beginning and end of one segment.

Other than the crop parameters listed in Table 4.4 and the POI parameters, the profiling pipeline was much the same as in Section 4.3.4, which includes capture, segmentation, splitting into training and test sets. The classification model was also much the same as in Section 4.3.5 with alignment, transformation into residual traces, and reduction to POI-based feature vectors, and then a logistic-regression classifier was trained on these feature vectors. The classification problem was now binary instead of the 16-class nibble identification. As before, two separate classifiers were trained: one for the first bit event and one for the repeated bit events, this was mainly done for convenience as most of the code was reused, in addition to having no conceivable downside. This classification model trained on `base_mul2` traces will be referred to as BM2.

Table 4.4. Cropping parameters used for the `base_mul2` trace segmentation.

Parameter	Value
FIRST_EVENT_START	100
FIRST_EVENT_LEN	37
REPEATED_FIRST_START	130
REPEATED_CYCLE_LEN	37
REPEATED_PRE_SAMPLES	7
REPEATED_EXTRACT_LEN	45
N_REPEATED_CYCLES	62

4.5 Evaluation

After all models had been trained, the final step of the experiments is to perform rigorous tests and evaluation of the models. The goal of the evaluation of the models is to determine if a trained machine learning model can consistently identify secret-dependent information only from captured traces of the computation of `base_mul`, and compare this to similar models with implemented countermeasures.

The tests performed are located in the notebook `AUTOML/run_model.ipynb` [Paa26]. For each implementation, the corresponding trained models and raw trace files were loaded. For this evaluation, a completely new set of traces were generated to ensure that the models were tested on previously unseen data rather than traces used during training. For each model, a set with 400 random keys and 200 traces per key was used. In order to isolate noise, some classification tasks was given with the average of a fixed number of traces. To test the effect of averaging, a chosen number of traces, called *n_avg*, was selected from each file and averaged sample by sample into a single trace with less noise. That averaged trace was then segmented in the same way as during training. The hypothesis was that if the implementation

does not leak information, increasing n_{avg} will not increase model accuracy beyond random guessing.

For each file, the evaluation recorded whether the full prediction was correct, how many positions were classified correctly, the accuracy of the special and repeated parts, and accuracy of the top 2 candidates for each nibble. A summary containing the averages of these fields across all trace-files was also calculated. The notebook then saved both detailed result files and summary tables in .csv files. Finally the accuracies could be compared between all models in a graph.

Since the BM2 model uses a 63 bit binary classifier instead of a 16 nibble classifier, the position accuracies cannot be directly compared to the other models. For comparison purposes, the BM2 bit predictions were therefore grouped into nibble-equivalent positions. A grouped position was counted as correct only if all four corresponding bits were predicted correctly. For the last nibble, this was only three bits, since bit 63 was excluded from the model. The grouped BM2 values should therefore be interpreted as an approximate comparison to nibble-level positions, not as direct 16-class nibble classification results.

Table 4.5. Datasets used in the experimental methodology.

Dataset	Purpose	Size
Exploratory trace windows	Initial key-change comparison	4 sets $\times$ 200 traces
Unprotected profiling	Training and validation of the original implementation	$(400 \times 200 + 16 \times 50)$ traces
Protected profiling	Training and validation of countermeasure models	$(400 \times 200 + 16 \times 50)$ traces per variant
Benchmark traces	Final evaluation on unseen traces	400 files $\times$ 200 traces per variant

4.6 Methodological limitations

The isolated `base_mul` experiments provide controlled and accurately labeled traces, but they do not by themselves demonstrate recovery of full HQC secret key material from a complete decapsulation trace. Full-key recovery using this method would require reliable identification and alignment of every `base_mul` process in a much larger full HQC trace. The main evaluated recovery task in this thesis is therefore recovery of secret-dependent 64-bit limbs from isolated `base_mul` traces.

All traces were captured using the same ciphertext-limb $b = 0\text{xffffffffffffffffFULL}$. For a real attack, the attacker could influence the ciphertext component processed during decapsulation, so this does not make the setup unrealistic by itself. How-

ever, the experiments were not tested on typical ciphertext data, and therefore do not show how leakage is influenced by the ciphertext-dependent value, only the secret-dependent value.

The crop boundaries were selected by visual inspection, which may introduce experimenter bias. However, the selected parameters were then applied consistently across all files in each dataset. The repeated-cycle classifier also assumes that nibble positions 1-15 have sufficiently similar leakage structure after segmentation and alignment. This assumption was evaluated through position-wise benchmarking.

The logistic-regression model reused the preprocessing parameters selected for the nearest-centroid classifier and was therefore not independently optimized. The shuffled implementation also differs from the masked implementation in how it should be interpreted, since shuffling primarily randomizes the order of operations rather than necessarily removing leakage from each individual operation.

The goal of this methodology was not to optimize a machine-learning model, but to use the same general profiling setup to evaluate and compare recoverable secret-dependent information across the tested implementations. The classifiers were used only as analysis tools for detecting and quantifying secret-dependent information, not as a complete end-to-end key-recovery attack. More advanced architectures or broader hyperparameter searches may improve accuracy, but such optimization was outside the main scope of the methodology.

4.7 Description of use of artificial intelligence

For this thesis, artificial intelligence (AI) has been used as a support tool for both writing and debugging Python code when developing the classification models and related tests, and also for using clear and consistent language when writing the report. The tools used were ChatGPT and Codex. All code that has been generated has been carefully inspected, tested, and revised by me to ensure correctness in computation and method. When writing the report, AI has been used to format tables and equations, scan the document for errors such as typos, grammar mistakes, broken references, and inconsistencies in notation. AI has also been used to improve language by suggesting improvements to unclear sentences, which I have chosen to implement when the suggestion was evaluated to be an improvement. AI has not been used as a factual basis for the thesis. No personal or sensitive data has been uploaded to an AI tool. I take full responsibility for all academic content in this thesis and the related code on GitHub.

Chapter 5

Results

This chapter presents the results of the experiments described in Chapter 4. It follows the same sequence as the methodology chapter and presents the results for each stage of the analysis. It begins with the results of the initial exploratory experiment. Next, the chapter presents the accuracy of the two classification models and their ability to recover secret-dependent information from the standard unmasked implementation. Finally, using the most accurate model, it presents the prediction accuracies using the standard implementation and the evaluated countermeasure implementations.

5.1 Exploratory trace comparison

As described in Section 4.3, an initial exploratory comparison was performed using the first 11000 samples captured of a `base_mul` process inside an HQC decryption using a ChipWhisperer-HUSKY setup [New26b]. Four averaged trace sets were compared: two captured with the original secret key `sk_const`, and two captured under the modified condition, where the secret-dependent limb entering the `base_mul` call was replaced by `0xffffffffffffffffULL`, while keeping the ciphertext fixed. The average traces from the original condition are denoted $\bar{a}_1$ and $\bar{a}_2$, while the average traces from the modified condition are denoted $\bar{b}_1$ and $\bar{b}_2$.

Table 5.1 shows the mean absolute sample-wise differences between the averaged trace sets. The comparison shows a larger mean difference in power amplitude when original and modified key-limb conditions are compared, as opposed to comparisons within the same condition. The exploratory comparison was used to motivate further controlled experiments.

5.2 Isolated `base_mul` trace inspection

Since the exploratory comparison did show a measurable difference in `base_mul` when changing the key-limb, the next step was to inspect traces from an isolated execution

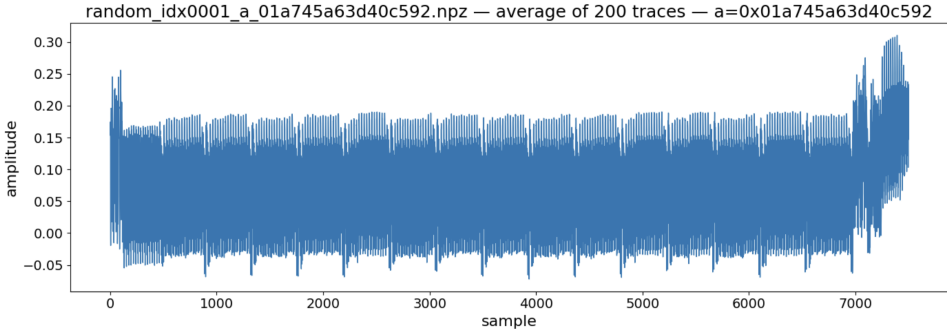
Table 5.1. Mean absolute differences between averaged trace sets.

Trace comparison	Compared	Mean $ d_{ij} $
$ \bar{a}_1 - \bar{a}_2 $	Original vs. original	0.000866
$ \bar{a}_1 - \bar{b}_1 $	Original vs. modified	0.035035
$ \bar{a}_2 - \bar{b}_1 $	Original vs. modified	0.035097
$ \bar{b}_1 - \bar{b}_2 $	Modified vs. modified	0.000926

of `base_mul` without noise from other HQC operations. As described in Section 4.3, this was done by adding a `SimpleSerial` command that executed `base_mul` inside a trigger window. Then 200 traces were captured with the same input values, and the average of these traces was plotted.

Several input combinations were tested. Figure 5.1 shows a trace where the secret-dependent input limb was set to a random value `0x01a745a63d40c592`, and the second input limb was set to `0xffffffffffffffff`. The trace shows 16 repeated regions. To validate the interpretation of this structure, the loop bounds in `base_mul` were temporarily modified. Reducing the number of outer-loop iterations reduced the number of repeated regions in the trace accordingly, while reducing the number of inner-loop iterations reduced the number of spikes inside each region.

Figure 5.2 shows a closer view of two cycles corresponding to the nibble values 3 and 6. The repeated structure is visible, and small differences can be seen, but the two cycles are not visually classifiable in a reliable way.

**Figure 5.1.** 200 `base_mul` traces averaged sample-wise and combined to one trace. Captured with key-limb value `0x01a745a63d40c592`.

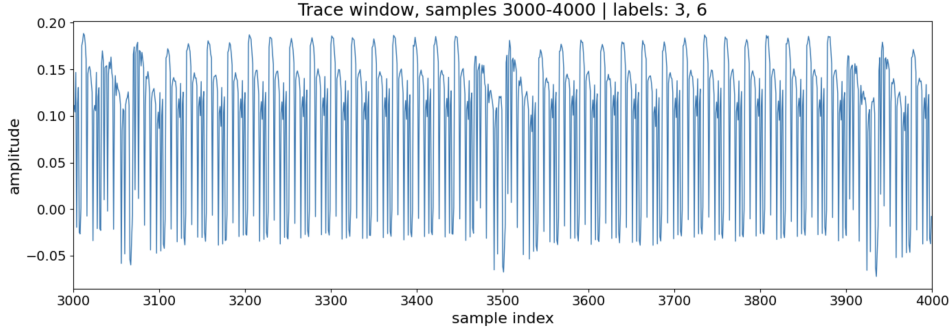


Figure 5.2. Zoomed in plot of Figure 5.1, from sample index 3000 to 4000. Showing trace segments corresponding to nibble values 3 (left) and 6 (right).

Table 5.2. Selected hyperparameter configurations for the standard `base_mul` profiling data.

Parameter	Special nibble	Repeated nibbles
Crop window	0–420	5–445
POI count	80	96
Minimum POI spacing	4	5
POI radius	3	2
First derivative	No	No
Global mean removal	Yes	Yes
Maximum alignment shift	0	6

5.3 Unprotected `base_mul` classification

The first classification experiment evaluated whether nibble values could be recovered from isolated traces of the unprotected `base_mul` implementation. Two profiling models were compared: the nearest-centroid classification model (NCM) and the logistic-regression classification model (LRM). Both models used the same segmentation, alignment, POI selection, and feature-extraction pipeline.

When training the nearest-centroid model, a grid search on a set of selected hyperparameters was performed. The parameter combination with the highest overall accuracy was also chosen for training the logistic-regression model. The parameter combination was found based on the model’s accuracy in regards to standard `base_mul` traces. The same configuration was also used to train the masked and shuffled implementations. The selected parameter configuration is shown in Table 5.2.

Table 5.3. Limb-recovery performance of the nearest-centroid model (NCM) and logistic-regression model (LRM). Exact recovery means that all 16 nibbles of the 64-bit limb were classified correctly. Results are computed from 400 benchmark files using seed 1337.

n_{avg}	Classifier	Exact limb recovery	Mean correct nibbles
1	NCM	10.8%	13.99 / 16
1	LRM	14.0%	14.12 / 16
2	NCM	58.0%	15.47 / 16
2	LRM	68.0%	15.61 / 16
5	NCM	93.5%	15.93 / 16
5	LRM	98.0%	15.98 / 16
10	NCM	99.3%	15.99 / 16
10	LRM	100.0%	16.00 / 16

Table 5.4. Trace cost for first exact limb recovery on 400 files.

Model	Files tested	Success rate	Avg. traces needed	Median	Max
NCM	400	1.00	2.58	2	15
LRM	400	1.00	2.38	2	13

Table 5.3 shows that for all tested values of n_{avg} , the logistic-regression model had higher exact limb recovery and higher mean correct nibble count than the nearest-centroid model. With a single trace, the logistic-regression model recovered the full 64-bit limb in 14% of the files, while the nearest-centroid model recovered 10.8%. When averaging five traces, the logistic-regression model reached 98% exact limb accuracy, compared to 93.5% for the nearest-centroid model. With ten averaged traces, the logistic-regression model recovered all tested limbs correctly, while the nearest-centroid model recovered 99.3%.

Table 5.4 shows the number of traces required before exact limb recovery was obtained. Both models recovered all 400 files within the maximum of 200 traces. The logistic-regression model required slightly fewer traces on average, with 2.38 traces compared to 2.58 traces for the nearest-centroid model. The median was 2 traces for both models, while the maximum was 13 traces for LRM and 15 traces for NCM. The remaining countermeasure results are reported using the logistic-regression model.

5.4 Countermeasure results

The masked and shuffled models used the feature-extraction hyperparameters selected based on the unprotected implementation using the nearest-centroid model. Only the segmentation crop parameters were adjusted to match the corresponding traces due to different total trace samples. The following results should therefore be interpreted as transfer results for the unprotected attack configuration.

5.4.1 Masked lookup-table implementation

The results from the masked implementation shown in Table 5.5, show the exact word accuracy remained 0 for all tested values of n_{avg} . Mean nibble matches increased from 2.538 with one trace to 3.885 with 200 averaged traces. The special nibble accuracy increased from 0.675 to 0.923 over the same range, while mean repeated accuracy increased from 0.124 to 0.198.

Table 5.5. Summary of full-trace classification results for the masked model.

Measure	Number of averaged traces			
	1	2	5	200
Exact word accuracy	0.000	0.000	0.000	0.000
Mean nibble matches	2.538	2.858	3.305	3.885
Special accuracy	0.675	0.780	0.865	0.923
Mean repeated accuracy	0.124	0.138	0.163	0.198
Mean top-2 accuracy	0.273	0.295	0.330	0.354

5.4.2 Shuffled implementation

Table 5.6 shows the classification results for the shuffled implementation. Exact word accuracy was 0 for all tested values of n_{avg} . Mean nibble matches ranged from 2.035 to 2.268, while mean repeated accuracy ranged from 0.092 to 0.104. The special nibble accuracy increased from 0.660 with one trace to 0.713 with 200 averaged traces.

Table 5.7 shows the per-position accuracy. Position 7 had a significantly higher accuracy than the other repeated positions, increasing from 0.428 with one trace to 0.633 with 200 averaged traces. Most other repeated positions remained between approximately 0.04 and 0.09 across the tested values of n_{avg} .

Table 5.6. Summary of full-trace classification results for the shuffled model.

Measure	Number of averaged traces			
	1	2	5	200
Exact word accuracy	0.000	0.000	0.000	0.000
Mean nibble matches	2.035	2.125	2.198	2.268
Special accuracy	0.660	0.675	0.690	0.713
Mean repeated accuracy	0.092	0.097	0.100	0.104
Mean top-2 accuracy	0.200	0.208	0.213	0.219

Table 5.7. Per-position accuracy for the shuffled full-trace model. The columns correspond to the number of averaged traces, denoted n_{avg} . Position 07 with higher accuracy is highlighted in bold.

Position	1	2	5	200
pos 00	0.660	0.675	0.690	0.713
pos 01	0.083	0.073	0.073	0.070
pos 02	0.063	0.065	0.063	0.058
pos 03	0.080	0.058	0.083	0.083
pos 04	0.050	0.078	0.083	0.058
pos 05	0.068	0.080	0.070	0.085
pos 06	0.075	0.080	0.078	0.065
pos 07	0.428	0.498	0.593	0.633
pos 08	0.065	0.063	0.060	0.083
pos 09	0.080	0.063	0.053	0.058
pos 10	0.068	0.068	0.070	0.053
pos 11	0.080	0.083	0.075	0.078
pos 12	0.065	0.080	0.070	0.075
pos 13	0.040	0.053	0.055	0.058
pos 14	0.058	0.058	0.038	0.050
pos 15	0.075	0.055	0.048	0.053

5.4.3 Alternative base_mul2 implementation

As described in Section 4.4.3 the profiling attack on this implementation uses 63 binary classification tasks, compared to the 16-nibble classification tasks of the previous implementations. Table 5.8 shows the accuracy results of the classification model given the average of 1, 2, 5 and 200 traces produced with the same 64-bit key

limb. Exact word accuracy was 0 for one, two, and five averaged traces, and 0.142 for 200 averaged traces. Mean bit matches increased from 41.883 with one trace to 61.377 with 200 averaged traces. Special bit accuracy increased from 0.723 to 1.000, while mean repeated accuracy increased from 0.664 to 0.974.

Table 5.8. Summary of full-trace classification results for the BM2 model. Reported values are from 400 files.

Measure	Number of averaged traces			
	1	2	5	200
Exact word accuracy	0.000	0.000	0.000	0.142
Mean bit matches	41.883	45.439	51.067	61.377
Special bit accuracy	0.723	0.796	0.903	1.000
Mean repeated accuracy	0.664	0.720	0.809	0.974

5.5 Comparison of implementations

The final comparison summarizes per-position accuracy for the unprotected and protected implementations. Figure 5.3 shows the result for a single trace, while Figure 5.4 shows the result when averaging 200 traces with the same input limb.

In Figure 5.3, the unprotected implementation has accuracy above 0.8 for all nibble positions. The masked and shuffled implementations have lower repeated-position accuracy than the unprotected implementation, while position 0 is higher than the other positions for both implementations. In the shuffled implementation, position 7 is also higher than the other repeated positions. The `base_mul2` values in this figure are based on grouped bit predictions, where four consecutive bit predictions are counted as one nibble-equivalent position.

In Figure 5.4, the unprotected implementation reaches 1.0 accuracy across all positions. The masked implementation shows higher per-position accuracy than in the single-trace case. The shuffled implementation remains around 0.0625 for most repeated positions, while position 0 and position 7 are even higher than in Figure 5.3. The grouped `base_mul2` values are also higher than in the single-trace case.

Figure 5.5 and Figure 5.6 show the bit-position accuracy for `base_mul2`. With only one trace, the bit-wise accuracy is consistently above 0.5, and varies across the 63 predicted bit positions. With 200 averaged traces, the classifier is consistent and accurate with 1.0 accuracy from position 9 to 53, while the early and late positions in the trace are less accurate.

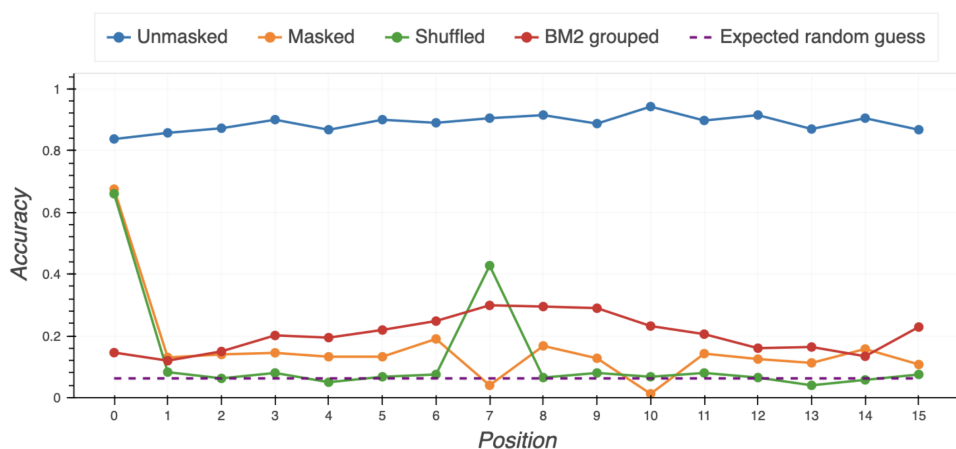


Figure 5.3. Accuracy per position of guessed key limb from `base_mul` traces using the unmasked, masked, shuffled and `base_mul2` implementations compared to expected random guesses. Guess based on a single trace.

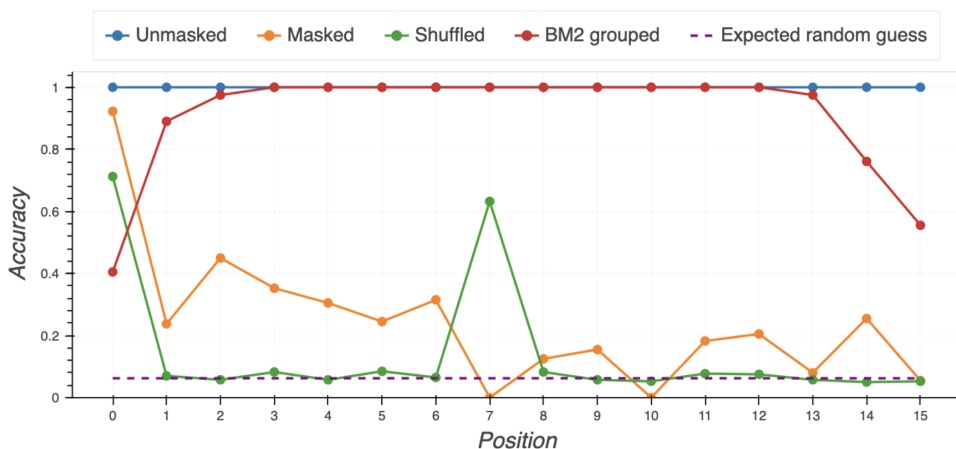


Figure 5.4. Accuracy per position of guessed key limb from `base_mul` traces using the unmasked, masked, shuffled and `base_mul2` implementations compared to expected random guesses. Guess based on the average of 200 traces with the same key limb.

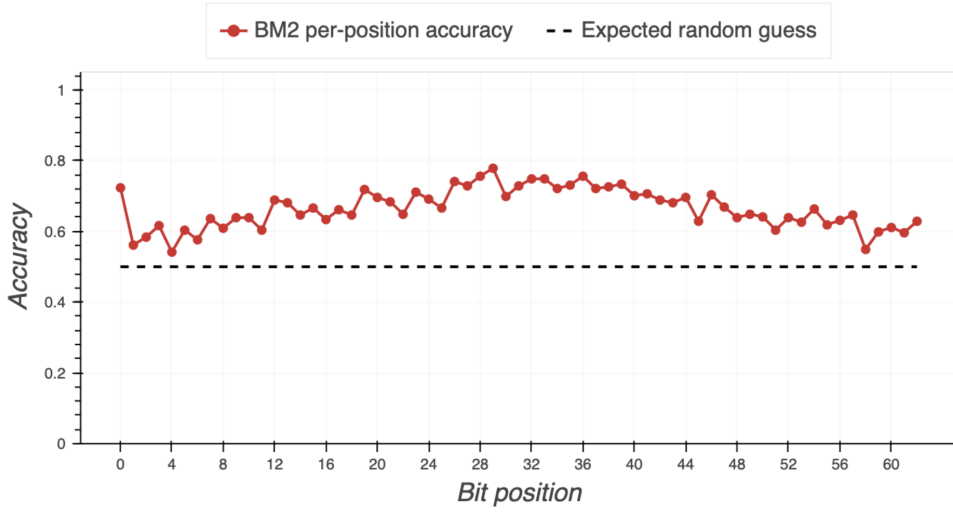


Figure 5.5. Accuracy per position of guessed key bit from the `base_mul2` implementation compared to expected random guesses. Guess based on a single trace.

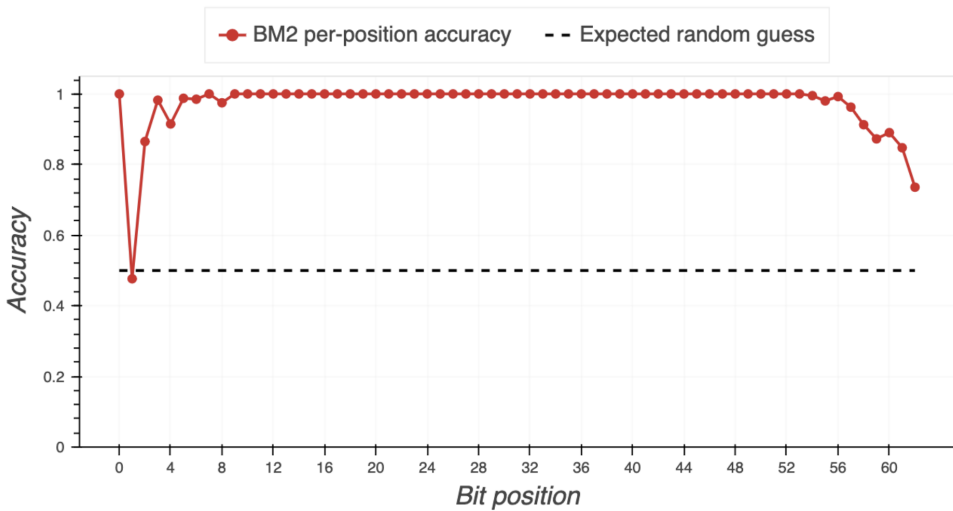


Figure 5.6. Accuracy per position of guessed key bit from the `base_mul2` implementation compared to expected random guesses. Guess based on the average of 200 traces with the same key limb.

Chapter 6

Discussion

This chapter discusses the results presented in Chapter 5 and interprets the findings in relation to the research objective stated in Chapter 1. The discussion focuses on what the results indicate about side-channel leakage in PQClean’s standard implementation of `base_mul`, and on the extent to which the evaluated countermeasures affected this leakage. The chapter also identifies limitations of the experimental method and considers the implications of the findings for future HQC implementations.

6.1 Interpretation of the baseline leakage

The exploratory comparison in Table 5.1 indicates that changing the secret-dependent key-limb entering `base_mul` produces a systematic change in the measured power trace. When comparing values where both traces have the modified key-limb, we see an almost equal difference as when original versus original is compared. This suggests that changing the key-limb input of `base_mul` in HQC, does produce a systematic observed difference and not just noise. This points to PQClean’s implementation of `base_mul` being a relevant target for further side-channel analysis, and motivates further investigative experiments by isolating `base_mul`.

The difference between traces captured with different key values motivated the choice of isolating the `base_mul` operation into an isolated firmware function. This reduced noise and allowed for more accurate captures and identification. It was also much faster to capture these traces. By isolating the operation, we could see 16 repeated processing cycles, each containing 16 inner-loop operations, similar to what is presented in the paper by Velek et al. [VRB26]. If we look at the code from Listing 3.1, we see that the cycles follow a similar structure, with 16 cycles of 16 spikes corresponding to the nested loop in `// Step 2`.

Velek et al. [VRB26] show that they are able to detect the nibble value by the highest peak in each inner cycle, but we were not able to recreate this. This could mean that the visibility and exploitability of the leakage depends on the measurement

setup, target platform, implementation details, and analysis method. Instead, the leakage had to be exploited using profiling classifiers based on points of interest across the aligned trace segments. This suggests that the implementation leaks secret-dependent information, but that the leakage in this measurement setup is more subtle than a simple peak-comparison rule.

6.2 Effectiveness of supervised profiling for classification

The results from the supervised profiling models show that the leakage observed in the exploratory experiments is exploitable in a controlled setting. Both the nearest-centroid model and the logistic-regression model were able to classify nibble values from segmented power traces of isolated `base_mul` executions. The logistic-regression classifier achieved the best performance across the measured metrics, although the retrained nearest-centroid model performed close to it. These results indicate that the current `base_mul` implementation in PQClean’s HQC implementation leaks secret-dependent information, and that key-limb values can be reconstructed nibble by nibble from power traces.

The nearest-centroid results confirm that much of the leakage is already captured by the selected POI feature representation. However, logistic regression still gives slightly better exact-limb accuracy and requires slightly fewer traces on average. This suggests that the class structure is mostly separable in the selected feature space, while the linear decision boundaries learned by logistic regression provide a small additional advantage over nearest-centroid classification. For this reason, the logistic-regression model was used as the primary classifier in the countermeasure experiments.

The exact limb recovery results in Table 5.3 show that one or two traces are not sufficient to reliably identify every key limb, even though the mean number of correctly classified nibbles is already high. Exact recovery is a stricter metric than nibble accuracy, since all 16 nibble predictions must be correct for the limb to be recovered. This explains why exact recovery is relatively low for a single trace. As shown in Table 5.4, both models recovered all 400 tested limbs within the 200-trace limit, with a median of two traces. The LRM model required 2.38 traces on average and at most 13 traces, compared to 2.58 traces on average and at most 15 traces for NCM.

The increase in accuracy relative to the number of averaged traces supports the claim that leakage exists. Since averaging more traces reduces random noise, the increase in accuracy with the number of averaged traces suggests that the classification model benefits from a clearer leakage signal. This also means that the model cannot perfectly identify the secret-dependent information from noisy traces. This does not

mean the ability to perfectly classify a full key-limb from a single `base_mul` trace captured from within an HQC decapsulation is impossible. However, the models developed for this thesis are not sufficiently accurate for a realistic single-trace HQC side-channel attack. Considering that the goal of this methodology was not to optimize a classification model, but rather determine if leakage of secret-dependent information occurs in HQC decapsulation. A more accurate method in predicting key-limbs from power traces is likely to exist, but developing such a model is outside the scope of this thesis and left for future work.

The results provide a clear answer to research question 2: PQClean’s `base_mul` implementation used in HQC does in fact exhibit measurable secret-dependent leakage on a ChipWhisperer platform [New26b]. This is shown by the exploratory experiment where traces captured with the same key-limb input produced were more similar. This is further supported by the trained profiling classification models, as they were capable of consistently identifying the nibble value used at every stage of an isolated `base_mul` execution.

It is worth clarifying that the results presented beyond the initial experiment only apply to isolated `base_mul` operations. The results do not by themselves constitute a complete key-recovery attack against a full HQC decapsulation trace. The only experiment performed on an actual HQC decryption trace was the initial experiment presented in Section 4.3.1. Therefore, we can only speculate that the observed leakage from the isolated experiments is transferable to an HQC decapsulation process. The countermeasure experiments should be interpreted with this in mind. They evaluate only how much exploitable information exists in an isolated operation. An end-to-end key-recovery attack would additionally need to identify the relevant `base_mul` executions inside the full decapsulation trace before aligning, segmenting, and applying a classification model to associate the resulting segments with the corresponding key-limb positions. This work does not demonstrate that this is possible.

6.3 Why it leaks

The observed leakage is consistent with the structure of the captured trace and the implementation code. In Section 3.1.3, the C code for the `base_mul` function was presented. In the methodology we found that the first nibble was best handled with a separate model. From Listing 3.1 lines 8–15, we see that the first nibble is handled separately and before `// Step 2` where the rest of the nibbles in the secret-dependent key limb is processed. This likely explains why the structure of nibble 0 in the trace was slightly different from the other 15. The reason for this is that the normal accumulation step would require shifting a 64-bit value by 64 bits,

which is undefined in the C programming language. Therefore, the implementation writes the first selected value to the variable `l` directly.

Next, from lines 18–27 in Listing 3.1, we see the nested loop responsible for finding the correct value in the precomputed lookup table for each nibble. The outer loop extracts one 4-bit nibble from `a`, while the inner loop scans all 16 precomputed table entries and uses a mask to select the entry corresponding to the current nibble. This corresponds with the structure of the trace, where we see 16 cycles, with 16 spikes for each cycle. The cycles correspond to the nibble processing, while the spikes correspond to the repeated inner-loop operations over the precomputed table entries. This was confirmed by temporarily modifying the bounds of both loops, and observing that the number of cycles and spikes changed accordingly. This implementation ensures the entire lookup table is scanned before selecting the correct value. Rabas et al. [RBG+26] describe this approach as an adaptation of `mul1` intended to make the HQC implementation time-constant and resistant to timing attacks. However, the observed leakage illustrates an important limitation of this countermeasure. On the ChipWhisperer, the repeated masked operations still produce data-dependent power consumption, which means a countermeasure that is designed to be effective against timing attacks can still be vulnerable in a power side-channel setting.

The exact source of the information leakage is harder to determine. The results associate the leakage with the inner loop of `// Step 2`, since the classifiers determine the correct nibble value from trace segments covering this operation. This suggests that the leakage originates from the table-selection operation. However, it does not explain how different values or low-level operations within that loop contribute to the exploitable leakage.

From the grid search of the nearest-centroid model, we see that the optimal number of POIs is between 80 and 96. This suggests that the leakage is distributed across the trace segment rather than being isolated to a single sample or peak. The success-rate of both the nearest-centroid and logistic-regression classifiers therefore supports the claim that the current `base_mul` implementation leaks secret-dependent information through data-dependent differences during the masked selection of the precomputed table value.

6.4 Effect of countermeasures

Having established the baseline leakage of the standard `base_mul` implementation under the profiling setup used in this thesis, the next step is to evaluate how the proposed countermeasures affect this leakage. This section discusses the masked, shuffled, and `base_mul2` variants individually before comparing their effectiveness. The results are then used to answer the research question 4: whether these countermeasures

reduce exploitable leakage under the same profiling methodology.

6.4.1 Effects of the masked implementation

The masked implementation targets the table-selection operation identified as the main source of exploitable leakage in the unprotected `base_mul`. As described in Section 4.4.1, the implementation applies a random input mask and output mask for each four-bit window. The goal is that the value handled during the captured part of the computation is no longer the secret-dependent value itself, but rather this value XORed with a random mask, resulting in a random value [CS20]. The random mask contribution is then removed after the trigger window, preserving the correctness of the computation.

The goal with this implementation was to weaken the correlation between the measured trace and the true nibble value, ideally making the classifier perform close to random guessing. For a 16-class problem, random guessing corresponds to $1/16 = 6.25\%$. However, as seen from Section 5.4.1, the classification model is still able to correctly guess nibble values with an accuracy above this threshold. The masked implementation therefore reduces the leakage compared to the unprotected implementation, but it does not remove the leakage completely.

We also see that the accuracy increases when more traces with the same key-limb are averaged. The hypothesis was that if the implementation does not leak information, increasing n_{avg} would not increase model accuracy, because a more accurate trace would still not give any information. The results from the masked implementation do not fully follow this hypothesis. Instead, the improvement with averaging suggests that some stable pattern related to the secret-dependent nibble value is present in the traces, even when the unmasked value itself is not directly processed.

A likely reason for this is that the implementation masks the table value at the logical level, but not the full physical computation. Inside the trigger window, the masked nibble $a_w \oplus \delta_w$, the masked lookup table, the selection logic, and the accumulator update are all still processed as part of the same trace segment. These values are randomized, but they are also related through the same mask δ_w . Because of this, the classifier may still learn patterns from the combined leakage of the operations, even if the selected table value itself is masked.

There is also clear leakage from the first nibble of the limb. This is consistent with the earlier observation that the first nibble-processing event had a slightly different structure from the remaining events, while the following 15 events formed a regular repeated pattern. As explained in Section 6.3, for position 0, the selected value is accumulated directly into the low word 1, while the later positions require shift

operations and updates to both the low and high words. Position 0 therefore has no previous accumulator state, no shift operation, and no high-word update. This gives the first position a cleaner trace pattern, which can make it easier for the classifier to identify.

The effect of this exact masked implementation is a clear improvement from the unprotected `base_mul` implementation from PQClean. However, it should not be interpreted as a leakage-secure masking scheme. The results show that the countermeasure substantially weakens the leakage, but that exploitable side-channel information still remains. This also does not rule out the possibility of a better and more specialized classifying model that could improve the guessing accuracy even further.

6.4.2 Effects of the shuffled implementation

The shuffled implementation uses another common side-channel countermeasure. Instead of randomizing the intermediate table value, it randomizes the order in which the 16 nibble positions are processed using a Fisher-Yates shuffle [Dur64; HGK+15]. The same table-selection operation is still performed for each nibble, but the position of each nibble-processing window in the trace changes. The purpose is that even if the nibble value in one cycle can be identified, the position of that nibble in the key-limb cannot be inferred directly from the trace order.

The results show that this strategy is highly effective against full-limb reconstruction in the tested setting. The exact limb accuracy remains zero for all tested values of n_{avg} , and for most repeated nibble positions, the accuracy is close to the random-guessing baseline of $1/16 = 6.25\%$. This is true even when the number of averaged traces is increased. This indicates that shuffling the processing sequence hides the original order of the key limb sufficiently for the classification model to mostly place the predicted values incorrectly.

However, the countermeasure implementation also has some weaknesses. From Section 5.4.2, we see the reported positions 0 and 7 having a significantly higher guessing accuracy than the rest. These positions also become more accurate when the number of averaged traces is increased, which suggests that some exploitable leakage remains. The exact reason for this is currently unclear, as these positions are shuffled in the same way as the other nibble positions. To determine why the accuracy of the classification model is high at these positions, further testing and investigation is needed, and this is left for future work.

Another important point is that the shuffled implementation changes how the classification results should be interpreted. The profiling model used in this thesis assumes that the chronological order of the trace segments is also the order of the

nibbles in the key-limb. This assumption holds for the unprotected and masked implementations, but not for the shuffled implementation. Here, segment 0 does not necessarily correspond to nibble position 0, segment 1 does not necessarily correspond to nibble position 1, and so on. Therefore, even if the classifier identifies the correct nibble value for a given segment, the value may still be placed in the wrong position when reconstructing the full limb.

The results therefore show that shuffling is effective against the reconstruction method used in this thesis, but they do not prove that the individual lookup operations are leakage-free. Leakage may still exist at the segment level, but this model is not designed to recover both the nibble value and the random permutation. A different model could potentially try to infer the shuffled position of each operation in addition to the processed value, but this was outside the scope of this thesis.

Overall, the shuffled implementation substantially reduces leakage exploitable by the profiling method used in this thesis. The results show that the profiling model is unable to perform full key-limb recovery from traces of this implementation. Nevertheless, it should not be considered leakage-free, as some positions remain consistently distinguishable above random guessing, and better suited classification models may exist.

6.4.3 Effects of the alternative `base_mul2` implementation

Lastly, we have the `base_mul2` implementation from Velek et al. [VRB26]. They also addressed the precomputed lookup operation, which was intended to avoid timing leakage but was shown to introduce exploitable power leakage. They presented the variant `base_mul2`, which removes this lookup table altogether. This also removes the performance improvement provided by the precomputation used in the original `base_mul`. Instead of processing the 64-bit limb as 16 four-bit windows, `base_mul2` processes the input bit by bit. This changes the visible trace structure from 16 nibble-processing cycles to one sequence of smaller bit-level operations.

To test this in a meaningful manner, the classification model was modified to perform binary classification tasks on each bit operation, instead of the 16-class nibble classification used for the previous implementations. As described in the methodology, 63 bit events were used for the final evaluation, since all 64 bit operations could not be isolated consistently. For each segment, the model therefore predicts whether the corresponding bit is 0 or 1.

The results show that the prediction ability of the profiling model is consistently above 50%, even for a single trace, which suggests that some leakage still exists in this implementation. This is also supported by the increase in accuracy when the number of averaged traces is increased. With $n_{\text{avg}} = 200$, the mean repeated bit accuracy

reaches 97.4%, while the special bit accuracy reaches 100%. This indicates that the implementation still produces bit-dependent leakage that can be exploited by the profiling model when enough traces of the same computation are averaged. However, the reported leakage is less consistent than with the standard implementation.

The first bit event again shows the strongest classification result. The exact reason for this was not determined, but it may have a similar explanation as the nibble-0 leakage in the previous implementations. This could be related to the initial state of the registers and the fact that the first operation has less previous accumulator activity than the following operations. Further testing would be required to determine whether this is a true implementation-specific leakage source or an effect of the segmentation strategy.

Due to the difference in trace structure and processing, the classification task is now binary rather than 16-class. The smaller bit-level processing windows can make manual alignment and cropping more difficult, which may affect the consistency of the profiles used for training and testing. Therefore, the reduced full-limb recovery rate may be caused by both lower leakage, and the fact that the profiling pipeline used is less suited for this implementation.

The effectiveness of `base_mul2` is therefore uncertain. It removes the lookup-table structure targeted in the original attack and makes full-limb recovery much harder with the profiling method used in this thesis. However, it does not remove all side-channel leakage. Full-limb recovery is still possible in 14.2% of cases when the model is given 200 traces of the same computation. Since the goal was not to optimize a bit-prediction model, but rather to evaluate the remaining recoverable leakage under the same general profiling approach, the increased accuracy with n_{avg} indicates that exploitable leakage remains. A more suitable and optimized classification model may therefore be able to exploit the remaining leakage more effectively.

6.5 Comparing the implementations

It is evident that the standard `base_mul` of PQClean’s HQC implementation leaks secret-dependent key-limb information through power side-channel analysis. When comparing each countermeasure to this standard implementation, it is clear that all countermeasures tested in this thesis reduce exploitable power side-channel leakage with the logistic-regression profiling model. This gives a concise answer to research question 4. However, none of the countermeasure implementations removes the leakage completely. In some cases, the accuracy of predicted values increases with the number of averaged traces, as seen for the masked implementation and `base_mul2`. In other cases, specific positions are easier to classify than others, as shown with the shuffled implementation and the first position of every implementation.

The countermeasures have important differences in how they aim to protect secret-dependent information during processing. The masked implementation randomizes the intermediate table value, but still shows residual leakage, and especially for the first nibble. The shuffled implementation does not aim to remove leakage from each individual lookup operation, but rather makes key-limb reconstruction difficult by randomizing the order of operations. This is effective against the reconstruction method used in this thesis, because the order of the cycles in the trace no longer directly corresponds to the order of the key limb. The alternative `base_mul2` implementation removes the lookup-table structure entirely, and instead performs the multiplication of a and b one bit at a time, removing the original four-bit nibble leakage target. Because of these differences, the implementations are not directly comparable only by raw classification accuracy.

It is also worth addressing that for the masked and shuffled implementations, the averaged traces are not exact repetitions of the same physical computation. Since randomness is introduced, the masks and order of computation are not necessarily the same for all 5, 10, or 200 traces. However, the actual secret value does not change. This further supports the claim that at least the masked implementation leaks secret-dependent information.

The shuffled implementation is the only tested implementation where the accuracy mostly does not increase with the number of averaged traces and remains close to the random-guessing baseline of $1/16 = 6.25\%$, with the exception of two nibble positions. This suggests that this implementation is most effective at countering the profiling-based side-channel attack used in this thesis. The alternative `base_mul2` was the only tested implementation other than the standard implementation where the classification model achieved full key-limb recovery with 200 traces, although only in 14.2% of the 400 cases tested. Based on the classification model's ability to recover the full key-limb, `base_mul2` therefore appears to be the least effective of the tested countermeasures against the current profiling pipeline.

6.6 Discussion of research questions

Now that we have interpreted the results, we can go back to the research questions from Section 1.2 and summarize the findings of this thesis.

RQ1: What side-channel weaknesses in HQC have been identified in previous work, and which implementation-level countermeasures have been proposed to reduce them?

From previous work in Section 1.4.1, we have seen that HQC has in several ways been targeted for side-channel attacks. This ranges from timing to electromagnetic and power attacks, as seen in this thesis. The targeted `base_mul` function itself has

been shown to be vulnerable, as Velek et al. [VRB26] and Rabas et al. [RBG+26] both demonstrate power side-channel attacks on this part of the decapsulation. Previous work has also proposed countermeasures to reduce this leakage, such as constant-time decoding, masking, shuffling, and alternative multiplication implementations such as `base_mul2`.

RQ2: What recoverable secret-dependent information is present in isolated executions of PQClean’s `base_mul` on a ChipWhisperer-HUSKY platform?

The exploratory trace comparison in Section 5.1 first showed that changing the secret-dependent limb processed in `base_mul` and keeping the ciphertext-dependent limb fixed produced a systematic difference in the captured power trace of an HQC decapsulation. In Section 5.2, we see that by isolating `base_mul` and inspecting the trace structure, there is a consistent repeating structure where each cycle corresponds to the processing of a nibble in the limb. By consistently being able to recover the key-limb value from a power trace only, we have shown that this value leaks through power side channels.

RQ3: To what extent can supervised profiling models recover secret-dependent key-limb information from isolated `base_mul` power traces?

From the results presented in Section 5.3, we see that the trained profiling-based classifier is consistently able to identify the secret-dependent key-limb information used in processing. Even with a single trace, the classifier is significantly more accurate than random guessing, indicating that the traces carry information about the key-limb value.

RQ4: How do the masked, shuffled, and alternative `base_mul2` implementations affect the observed leakage compared to the standard PQClean `base_mul` implementation under the same general profiling setup?

From Section 5.5, we see that all the tested countermeasures decrease the model’s prediction accuracy under the same segmentation, profiling and training setup. This indicates that the observed leakage in these implementations is less than in the standard implementation. However, further research and testing is needed before these implementations can be considered fully side-channel secure.

6.7 Limitations

The results shown in this thesis should be interpreted as evidence of leakage from within a process in the HQC decapsulation algorithm, and not as proof of a possible end-to-end key-recovery attack. To evaluate the possibility of such an attack using the findings from this thesis, one must be able to consistently locate the `base_mul`

executions, align the traces, and remove enough noise to classify the secret-dependent values. However, the ability to determine only a few bits or limbs of the secret key could still assist in a larger attack, for example by reducing the remaining search space.

The evaluation of the countermeasures is limited by the profiling method used in this thesis. The accuracies reported by the profiling model determine only how much side-channel leakage is exploitable by this kind of trained model, with the chosen segmentation, alignment, POI selection, and classifier settings. It does not show that other more specialized attack methods could not perform better in recovering the full key-limb when the countermeasures are used. This also means that the leakage from the countermeasures was limited under the current profiling setup, but we cannot prove that this is true for all setups.

For the shuffled implementation, the model still assumed that the order of the key-limb was the same as the processing cycle, which may be a reason why this implementation performed well under the profiling setup in this thesis. This could still mean that some information about the processed values or the original order can leak, but that this was not exploited by the current model. What was not tested was the model’s ability to predict the values regardless of their order, or to recover the shuffled order itself. A correct prediction of either could decrease the security of the implementation.

A central part of the results is the use of averaged traces. For an attack that follows the pipeline in this methodology, we must assume that it is possible to capture traces of similar quality as in these isolated experiments, which may or may not be the case in a full HQC decapsulation setting. The results from the averaging process should therefore mainly be interpreted as evidence that the leakage is stable across repeated measurements, and not as proof that the same recovery rate would be achievable in a practical end-to-end attack.

The isolated experiments also used a fixed ciphertext-dependent input limb for `base_mul`. This makes the measured leakage mainly related to changes in the secret-dependent value, which makes the results easier to compare. However, it also means that the measured accuracies do not show how the models would perform across randomly distributed ciphertext limbs or in a more realistic SCA scenario.

6.8 Future work

The scope of this thesis has mainly been focused on side-channel leakage in each implementation, with less focus on processing performance and implementation cost. Future work should investigate secure polynomial multiplication algorithms

that are both side-channel secure and performance-efficient. Another direction is to develop more specialized prediction models for each proposed implementation. This could, for example, include investigating whether the shuffled implementation leaks any information about the order of operations, or whether a specialized profiling model for `base_mul2` could consistently perform key-limb recovery from a single trace. Future work could also investigate why the first nibble has more leakage in all of the implementations, and seek to develop a secure countermeasure against this. An important next step is to perform similar tests as in this thesis on full HQC decapsulation traces. This would verify whether `base_mul` is locatable and exploitable in the full decapsulation setting. Additionally, none of the proposed countermeasures have been designed or tested to withstand fault injection or glitching attacks. Future work could investigate this.

Chapter 7

Conclusions

This thesis has investigated power side-channel leakage from the post-quantum KEM HQC, more specifically from polynomial multiplication during decapsulation. The implementation targeted was PQClean’s HQC implementation, where the function `base_mul` inside `karatsuba` was selected as the main target. The objective was to evaluate recoverable secret-dependent information from isolated executions of PQClean’s `base_mul`, use this as a baseline, and compare how selected implementation-level countermeasures affect the observed leakage under the same general evaluation setup.

The results from the experiments show clear leakage from the standard `base_mul` function in PQClean. A captured trace of this function shows 16 cycles corresponding to the 16 four-bit nibbles of the secret-dependent limb processed by the function. Both a nearest-centroid model and a multinomial logistic-regression model trained on labeled trace segments of individual cycles were able to classify which nibble value corresponded to a given unlabeled segment with above 87% accuracy. By classifying all 16 segments individually and combining the predicted nibbles into a full 64-bit limb, the logistic-regression model was able to predict the full limb in 400/400 cases when 10 traces of the same operation were averaged and passed to the classification model. This confirms that the leakage is exploitable by the supervised profiling pipeline used in this thesis.

After the side-channel analysis of `base_mul`, the lookup-table selection was identified as an important source of leakage. Due to the clear leakage from this operation, three other implementations of `base_mul` were presented and tested. The first used masked values during the lookup-table operation, the second used shuffling to process the nibble positions in a random order, and the third was an implementation presented by Velek et al. that removed the lookup table operation altogether. All three implementations were trained and tested in a similar way to the standard `base_mul`. The results show that all three implementations substantially reduced the prediction accuracy of the classification model. However, only the shuffled implementation showed accuracies close to what would be expected from random

guessing. In the masked and shuffled implementations, the first nibble of the limb had a significantly higher prediction rate than the rest of the nibble positions, although the reason for this was not proven. The same was observed for nibble position seven in the shuffled implementation. The model trained and tested on the alternative `base_mul2` implementation from Velek et al. showed consistent accuracy above random guessing, even with only a single trace, with the most accurately guessed positions being near the middle of the trace. Unlike the other implementations, `base_mul2` processes the 64-bit limb bit by bit, which changes the evaluation from nibble-level classification to bit-level classification. Overall, the countermeasure implementations significantly reduced the measurable leakage, but none of them can be said to completely remove leakage.

The results presented in this thesis make it possible to answer all four research questions presented in Section 1.2. For *RQ1*, previous work has shown that side-channel leakage of secret-dependent values can occur from both decoding and polynomial multiplication. Previous work has also proposed countermeasures, including modified multiplication functions and fully masked HQC implementations. For *RQ2* and *RQ3*, exploitable side-channel leakage in the PQClean implementation of `base_mul` is seen through the profiling model's ability to recover the 64-bit key limb from isolated traces, with complete recovery for all tested limbs when 10 traces were averaged. Finally, for *RQ4*, the results show that the countermeasure implementations reduced exploitable leakage with respect to the profiling model. However, none of them removed the leakage completely.

The scope of this thesis has been limited to finding exploitable side-channel leakage in an isolated execution of the `base_mul` function, which is only a small part of the HQC algorithm. The results presented are therefore not proof of a full-key recovery side-channel attack on HQC. Instead, they show that a secret-dependent operation inside HQC polynomial multiplication can leak enough information to recover 64-bit limbs under controlled conditions. Future work should therefore investigate whether the `base_mul` executions can be located and exploited inside complete decapsulation traces. In addition, the countermeasure implementations were only evaluated using the profiling model's classification accuracy. These implementations should therefore not be considered secure without further testing.

References

- [ABD+18] C. Aguilar-Melchor, O. Blazy, et al., *Efficient encryption from random quasi-cyclic codes*, 2018. [Online]. Available: <https://doi.org/10.1109/TIT.2018.2804444>.
- [AS26] H. Adkins and S. Schmieg. “Quantum frontiers may be closer than they appear”. Google, The Keyword, last visited: May 26, 2026. [Online]. Available: <https://blog.google/innovation-and-ai/technology/safety-security/cryptography-migration-timeline/>.
- [Ber05] D. J. Bernstein, “Cache-timing attacks on aes”, 2005.
- [Ber68] E. R. Berlekamp, *Algebraic Coding Theory*. McGraw-Hill, 1968, Classic text defining linear codes as vector subspaces over finite fields.
- [BFG+19] J. Brendel, M. Fischlin, et al., “Challenges in proving post-quantum key exchanges based on key encapsulation mechanisms.”, *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1356, 2019.
- [BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg, “On the inherent intractability of certain coding problems (corresp.)”, *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.
- [BR60] R. Bose and D. Ray-Chaudhuri, “On a class of error correcting binary group codes”, *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0019995860902874>.
- [CRR03] S. Chari, J. R. Rao, and P. Rohatgi, “Template attacks”, in *Cryptographic Hardware and Embedded Systems – CHES 2002*, ser. Lecture Notes in Computer Science, vol. 2523, Springer, 2003, pp. 13–28.
- [CS20] G. Cassiers and F.-X. Standaert, “Trivially and efficiently composing masked gadgets with probe isolating non-interference”, *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 2542–2555, 2020.
- [DGA+25] J.-C. Deneuville, P. Gaborit, et al., “Hamming quasi-cyclic (hqc) – specification (version 22/08/2025)”, PQC-HQC Consortium, Tech. Rep., Aug. 2025, Version dated 22 August 2025. [Online]. Available: https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf.
- [DH76] W. Diffie and M. E. Hellman, “New directions in cryptography”, *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

- [Dur64] R. Durstenfeld, “Algorithm 235: Random permutation”, *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.
- [ES24] A. Esser and P. Santini, *Not just regular decoding: Asymptotics and improvements of regular syndrome decoding attacks*, L. Reyzin and D. Stebila, Eds., Santa Barbara, CA, USA, Aug. 2024.
- [FN-25] FN-sambandet. “FNs bærekraftsmål”. Sist oppdatert 01.09.2025, last visited: May 26, 2026. [Online]. Available: <https://fn.no/om-fn/fns-baerekraftsmaal>.
- [GLG22] G. Goy, A. Loiseau, and P. Gaborit, “A new key recovery side-channel attack on hqc with chosen ciphertext”, in *Proceedings of the 4th PQC Standardization Conference*, pdf available via NIST CSRC, 2022. [Online]. Available: <https://csrc.nist.gov/csrc/media/Events/2022/fourth-pqc-standardization-conference/documents/papers/a-new-key-recovery-side-channel-attach-on-hqc-pqc2022.pdf>.
- [GSA+25] G. Goy, M. Spyropoulos, et al., “Side-channel sensitivity analysis on hqc: Towards a fully masked implementation”, Cryptology ePrint Archive, IACR, Tech. Rep. 2025/1344, 2025, Preprint, available on the IACR ePrint Archive. [Online]. Available: <https://eprint.iacr.org/2025/1344.pdf>.
- [Ham50] R. W. Hamming, “Error detecting and error correcting codes”, *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, Apr. 1950.
- [HGK+15] T. K. Hazra, R. Ghosh, et al., “File encryption using fisher-yates shuffle”, in *2015 International Conference and Workshop on Computing and Communication (IEMCON)*, 2015, pp. 1–7.
- [HTF09] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer, 2009. [Online]. Available: <https://hastie.su.domains/ElemStatLearn/main.html>.
- [JMZ23] H. Jiang, Z. Ma, and Z. Zhang, “Post-quantum Security of Key Encapsulation Mechanism Against CCA Attacks with a Single Decapsulation Query”, in *Advances in Cryptology – ASIACRYPT 2023*, Singapore: Springer, Dec. 2023, pp. 434–468.
- [KSSW22] M. J. Kannwischer, P. Schwabe, et al., “Improving software quality in cryptography standardization projects”, in *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 19–30. [Online]. Available: <https://eprint.iacr.org/2022/337>.
- [LCC+23] S. Li, Y. Chen, et al., “Post-quantum security: Opportunities and challenges”, *Sensors*, vol. 23, no. 21, 2023. [Online]. Available: <https://www.mdpi.com/1424-8220/23/21/8744>.
- [LE26] E. Lynch and D. Elliott. “Security for the quantum era: Implementing post-quantum cryptography in android”. Google Security Blog, last visited: May 26, 2026. [Online]. Available: <https://blog.google/security/security-for-the-quantum-era-implementing-post-quantum-cryptography-in-android/>.

- [MMT15] Z. Martinasek, L. Malina, and K. Trasy, “Profiling power analysis attack based on multi-layer perceptron network”, in *Computational Problems in Science and Engineering*, ser. Lecture Notes in Electrical Engineering, vol. 343, Cham: Springer, 2015, pp. 317–339.
- [MOP07] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.
- [Nat24a] National Institute of Standards and Technology, “Module-Lattice-Based Key-Encapsulation Mechanism Standard”, National Institute of Standards and Technology, Federal Information Processing Standards Publication 203, Aug. 2024. [Online]. Available: <https://csrc.nist.gov/pubs/fips/203/final>.
- [Nat24b] National Institute of Standards and Technology, “Transition to Post-Quantum Cryptography Standards”, National Institute of Standards and Technology, NIST Internal Report 8547 ipd, Nov. 2024, Initial public draft. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8547.ipd.pdf>.
- [New26a] NewAE Technology Inc., *Chipwhisperer*, Accessed: 2026-05-20, 2026. [Online]. Available: <https://github.com/newaetech/chipwhisperer>.
- [New26b] NewAE Technology Inc., *Chipwhisperer documentation*, Accessed: 2026-05-11, NewAE Technology Inc., 2026. [Online]. Available: <https://chipwhisperer.readthedocs.io/en/latest/index.html>.
- [New26c] NewAE Technology Inc., *Chipwhisperer jupyter*, Accessed: 2026-05-20, 2026. [Online]. Available: <https://github.com/newaetech/chipwhisperer-jupyter>.
- [NIS25] NIST, *Selected Algorithms - Post-Quantum Cryptography | CSRC | CSRC*, [Online; accessed 3. Nov. 2025], Sep. 2025. [Online]. Available: <https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms>.
- [Ope26] Open Quantum Safe, *liboqs: C library for quantum-safe cryptographic algorithms*, <https://github.com/open-quantum-safe/liboqs>, GitHub repository, accessed 2026-05-25, 2026.
- [Paa25] H. H. Paaske, “Investigating attacks against hqc implementations”, Norwegian University of Science and Technology (NTNU), Pre-project report (IMT4205), Dec. 2025, Unpublished student pre-project report.
- [Paa26] H. H. Paaske, *Masterhqc*, <https://github.com/haraldpaaske/MasterHQC>, GitHub repository, 2026.
- [Pes16] P. Pessl, “Analyzing the shuffling side-channel countermeasure for lattice-based signatures”, in *Progress in Cryptology – INDOCRYPT 2016*, ser. Lecture Notes in Computer Science, General reference for shuffling as a side-channel hiding countermeasure, vol. 10095, Springer, Cham, 2016, pp. 153–170. [Online]. Available: <https://eprint.iacr.org/2017/033.pdf>.
- [PHJB19] S. Picek, A. Heuser, et al., “A systematic evaluation of profiling through focused feature selection”, English, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2802–2815, 2019, Accepted author manuscript.

- [PQC25] PQCclean. “Pqclean hqc implementation”, last visited: May 11, 2026. [Online]. Available: <https://github.com/PQCclean/PQCclean>.
- [PT19] T. B. Paiva and R. Terada, “A timing attack on the hqc encryption scheme”, in *Selected Areas in Cryptography – SAC 2019: 26th International Conference, Waterloo, ON, Canada, August 12–16, 2019, Revised Selected Papers*, Waterloo, ON, Canada: Springer-Verlag, 2019, pp. 551–573. [Online]. Available: https://doi.org/10.1007/978-3-030-38471-5_22.
- [RBG+26] T. Rabas, J. Buček, et al., “Profiling side-channel attack on hqc polynomial multiplication using machine learning methods”, in *Constructive Approaches for Security Analysis and Design of Embedded Systems*, M. Rivain and P. Sasdrich, Eds., Cham: Springer Nature Switzerland, 2026, pp. 580–602.
- [RWTM24] M. G. D. Remmerswaal, L. Wu, et al., “Autopoi: Automated points of interest selection for side-channel analysis”, *Journal of Cryptographic Engineering*, vol. 14, pp. 463–474, 2024. [Online]. Available: <https://link.springer.com/article/10.1007/s13389-023-00328-y>.
- [sci26a] scikit-learn developers, *Logisticregression — scikit-learn documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html, Accessed 2026-05-04, 2026.
- [sci26b] scikit-learn developers, *Nearestcentroid — scikit-learn documentation*, <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestCentroid.html>, Accessed 2026-05-04, 2026.
- [Sho94] P. W. Shor, “Algorithms for quantum computation: Discrete logarithms and factoring”, in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, IEEE, 1994, pp. 124–134.
- [SRSW21] T. Schamberger, J. Renner, et al., “A power side-channel attack on the cca2-secure hqc kem”, in *Smart Card Research and Advanced Applications*, P.-Y. Liardet and N. Mentens, Eds., Cham: Springer International Publishing, 2021, pp. 119–134.
- [Sta10] F.-X. Standaert, “Introduction to side-channel attacks”, in *Secure Integrated Circuits and Systems*, I. M. Verbauwhede, Ed. Boston, MA: Springer US, 2010, pp. 27–42. [Online]. Available: https://doi.org/10.1007/978-0-387-71829-3_2.
- [Val24] A. Valavanidis, “Quantum computing”, *A revolutionary computing capability to sift through huge numbers of possibilities and extract potential solutions to complex problems*, vol. 1, pp. 1–25, 2024.
- [VRB26] P. Velek, T. Rabas, and J. Buček, “Simple power analysis of polynomial multiplication in hqc”, in *Proceedings of the 12th International Conference on Information Systems Security and Privacy (ICISSP 2026)*, vol. 2, SCITEPRESS, 2026, pp. 299–306.
- [Wei] E. W. Weisstein, *Karatsuba multiplication*, <https://mathworld.wolfram.com/KaratsubaMultiplication.html>, Accessed: 2026-05-12.

- [Wel51] B. L. Welch, “On the comparison of several mean values: An alternative approach”, *Biometrika*, vol. 38, no. 3/4, pp. 330–336, 1951. last visited: May 27, 2026. [Online]. Available: <http://www.jstor.org/stable/2332579>.
- [WGR22] V. Weger, N. Gassner, and J. Rosenthal, “A Survey on Code-Based Cryptography”, *arXiv*, Jan. 2022.

