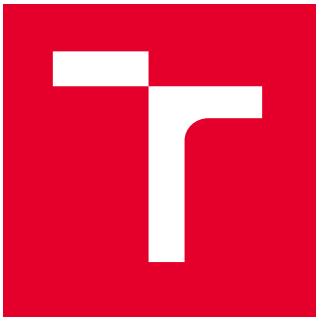


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

SIDE-CHANNEL ANALYSIS OF THE ML-DSA ALGORITHM USING TECHNIQUES INSPIRED BY MARVIN AND MINERVA

ANALÝZA POSTRANNÍCH KANÁLŮ ALGORITMU ML-DSA S VYUŽITÍM TECHNIK INSPIROVANÝCH ÚTOKY MARVIN A MINERVA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Robin Vala**
AUTOR PRÁCE

SUPERVISOR — **M.Sc. Sara Ricci, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2026

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Robin Vala ***ID:*** 241124

Year of study: 2 ***Academic year:*** 2025/26

TITLE OF THESIS:

Side-Channel Analysis of the ML-DSA Algorithm Using Techniques Inspired by Marvin and Minerva

INSTRUCTION:

This project focuses on analysing side-channel vulnerabilities in post-quantum ML-DSA digital signature scheme. The student will explore the adaptation of testing methods and strategies inspired by the Marvin and Minerva attacks. These techniques will be used to identify and measure timing-based side-channel leakage in ML-DSA implementations. Where leakage is found, the student will assess the impact and propose possible mitigation strategies.

RECOMMENDED LITERATURE:

[1] National Institute of Standards and Technology (NIST). Module-Lattice-Based Digital Signature Standard. Gaithersburg, MD: U.S. Department of Commerce, 2024. Available from: https://csrc.nist.gov/pubs/fips/204/final

[2] KARIO, H. Everlasting ROBOT: The Marvin Attack. Computer Security – ESORICS 2023. Lecture Notes in Computer Science. 2024, s. 243-262. ISBN 978-3-031-51478-4. Available from: https://doi.org/10.1007/978-3-031-51479-1_13.

Date of project specification: 9.2.2026 ***Deadline for submission:*** 26.5.2026

Supervisor: M.Sc. Sara Ricci, Ph.D.
Consultant: M.Sc. Alicja Kario

prof. Ing. Jan Hajný, Ph.D.
Chair of study program board

WARNING:

The author of the Master's Thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.

ABSTRACT

This thesis deals with timing side-channel analysis of the Module-Lattice Digital Signature Algorithm (ML-DSA). The aim is to examine whether numerical properties of intermediate values used during signing exhibit measurable timing dependencies. The methodology is inspired by the Marvin and Minerva timing attacks.

A dedicated toolkit was implemented for this purpose. It supports generation of keys and message datasets, timing measurement of several ML-DSA implementations, reconstruction of selected intermediate values, feature extraction, and statistical evaluation using the *tlsfuzzer* tool.

The thesis evaluates ML-DSA signing using feature-based and class-based experiments on random and crafted message datasets. The measurements cover both the complete signing operation and partial signing without the final signature encoding step. Random-message measurements did not show stable timing dependence. In contrast, crafted-message experiments revealed statistically significant timing differences in selected configurations. The results show that the observed dependencies depend on the implementation, processor, message dataset, and the measurement boundary.

KEYWORDS

ML-DSA, Post-quantum cryptography, Timing side-channel analysis, Marvin, Minerva, Lattice-based cryptography

ABSTRAKT

Tato práce se zabývá analýzou časových postranních kanálů algoritmu Module-Lattice Digital Signature Algorithm (ML-DSA). Cílem je prozkoumat, zda numerické vlastnosti mezilehlých hodnot používaných během podepisování vykazují měřitelné časové závislosti. Metodologie je inspirována časovými útoky Marvin a Minerva.

Pro tento účel byl implementován samostatný nástroj. Podporuje generování klíčů a datových sad zpráv, měření času několika implementací ML-DSA, rekonstrukci vybraných mezilehlých hodnot, extrakci charakteristik a statistické vyhodnocení pomocí nástroje *tlsfuzzer*.

Práce hodnotí proces podepisování algoritmu ML-DSA na základě experimentů zaměřených na vlastnosti a třídy, a to na sadách náhodných i speciálně vytvořených zpráv. Měření zahrnují jak kompletní operaci podepisování, tak částečné podepisování bez závěrečného kroku kódování podpisu. Měření na sadách náhodných zpráv neprokázala stabilní časovou závislost. Oproti tomu experimenty se speciálně vytvořenými zprávami odhalily statisticky významné časové rozdíly ve vybraných konfiguracích. Výsledky ukazují, že pozorované závislosti závisí na implementaci, procesoru, datové sadě zpráv a hranici měření.

KLÍČOVÁ SLOVA

ML-DSA, Postkvantová kryptografie, Analýza časových postranních kanálů, Marvin, Minerva, Kryptografie založená na mřížkách

Typeset by the `thesis` package, version 4.13; `https://latex.fekt.vut.cz/`

ROZŠÍŘENÝ ABSTRAKT

Digitální podpisová schémata založená na mřížkových konstrukcích představují klíčovou součást postkvantové kryptografie. Jejich bezpečnost je teoreticky dobře podložena, avšak praktické implementace mohou být ohroženy útoky postranními kanály, zejména časovými rozdíly vznikajícími při podepisování. Tyto rozdíly mohou vést k odhalení informací o zpracovávaných datech či o soukromém klíči. Algoritmus Module-Lattice Digital Signature Algorithm (ML-DSA), který byl standardizován NIST jako nástupce schématu CRYSTALS-Dilithium, se stal předmětem intenzivní analýzy. Praktické časové chování konkrétních implementací však stále vyžaduje experimentální ověření, zejména s ohledem na závislosti vznikající při zpracování mezilehlých hodnot. Moderní útoky Marvin a Minerva, ačkoli cílí na jiné kryptografické konstrukce, ukazují, že i velmi slabé časové odchylky lze za vhodných podmínek odhalit pomocí pečlivé statistické analýzy.

Tato práce si proto kladla za cíl vytvořit experimentální nástroj, který umožní zkoumat časové charakteristiky implementací ML-DSA a vyhodnocovat možné úniky pomocí metod inspirovaných uvedenými útoky. Součástí tohoto nástroje jsou skripty, které generují klíče a zprávy, měří dobu výpočtu podpisových operací, rekonstruují mezilehlé hodnoty algoritmu a připravují data pro následné statistické vyhodnocení.

Pro přípravu vstupních dat byl vytvořen skript, který pomocí příkazového nástroje OpenSSL automatizuje generování veřejných a soukromých klíčů pro všechny tři bezpečnostní úrovně ML-DSA a připravuje sadu zpráv určených k podepisování. Kromě náhodných zpráv jsou generovány také řízené datové sady, ve kterých jsou zprávy vybírány tak, aby pro daný klíč a deterministický režim podepisování došlo k přijetí podpisu již v první iteraci algoritmu. Všechna měření nad těmito daty mají tedy stejný počet iterací hlavní podpisové smyčky, čímž se omezuje jeden z nejvýraznějších zdrojů časové variability.

Měření podpisových operací probíhá tak, že nástroj načte soukromý klíč a pro každou zprávu provede podpis v deterministickém režimu. To umožňuje opakovat výpočet a rekonstruovat mezilehlé hodnoty použité při podpisu. Čas jedné podpisové operace je zaznamenán pro každou zpracovanou zprávu. Tento přístup byl aplikován na více implementací ML-DSA, konkrétně *dilithium-py*, *OpenSSL*, *Go* a *Nettle*, pro které byly vytvořeny samostatné měřicí programy. V rámci experimentů jsou využívána jak kompletní měření celého podpisového procesu, tak i částečná měření, u nichž je časování ukončeno před finální fází kódování podpisu. Oddělení těchto dvou režimů umožňuje porovnat měření celého podpisu s kontrolovanějšími experimenty a lépe posoudit, ve které části algoritmu se časová závislost projevuje.

Na měření navazuje rekonstrukce mezilehlých hodnot ML-DSA. Díky deterministickému režimu algoritmu a znalosti soukromého klíče lze pro každý zaznamenaný podpis znovu vypočítat vybrané interní hodnoty použité při podepisování. Tyto hodnoty jsou následně převedeny na numerické charakteristiky, jako je Hammingova váha, bitová velikost koeficientů, počet nulových hodnot nebo počet koeficientů vyžadujících redukci. Výsledkem je datová sada, ve které je každému časovému měření přiřazena množina charakteristik odvozených z odpovídající podpisové operace.

Získaná data jsou poté vyhodnocena pomocí statistického modulu nástroje *tlsfuzzer*, který byl využit také při analýzách útoků Marvin a Minerva. Použité neparametrické metody zahrnují sign test, Wilcoxonův párový test, testy pro blokově uspořádaná data, jako je Friedmanův nebo Skillings–Mackův test, a bootstrapové odhady intervalů spolehlivosti.

Použití neparametrických metod je při tomto měření zásadní, protože časová data často nesplňují předpoklady klasických parametrických testů. Navíc jsou tato data zatížena šumem a mohou obsahovat odlehlé hodnoty. Zvolený přístup tak umožňuje při dostatečném počtu měření a vhodně kontrolovaných podmínkách detekovat i velmi slabé závislosti mezi hodnotami zpracovávanými algoritmem a dobou jeho běhu, aniž by bylo nutné předpokládat konkrétní rozdělení dat.

Praktická analýza byla provedena pro všechny tři bezpečnostní varianty ML-DSA a pro více implementací. Při měření celého podpisového procesu na náhodných datech nebyla pozorována stabilní ani reprodukovatelná časová závislost na sledovaných charakteristikách rekonstruovaných mezilehlých hodnot. V tomto nastavení se do naměřeného času promítá odmítací vzorkování, finální kódování podpisu i další části algoritmu, což může slabší efekty překrýt. Výsledek proto naznačuje, že při běžném měření celého podpisu na náhodných zprávách jsou případné závislosti obtížně rozlišitelné.

Při použití řízených datových sad, ve kterých je omezen vliv odmítacího vzorkování, se v některých konfiguracích podařilo detekovat statisticky významné závislosti mezi dobou běhu a vybranými charakteristikami mezilehlých hodnot. V měřeních celého podpisu byly nejsilnější signály u implementací *Go*, *OpenSSL* a *Nettle* spojeny především s charakteristikami odvozenými z hodnot w_0. Po vyloučení finálního kódování podpisu však tyto signály pozorovány nebyly. To naznačuje, že zjištěná časová závislost souvisela spíše s kódováním výsledného podpisu než s předchozí částí podpisového výpočtu.

Částečná měření, ve kterých bylo časování ukončeno před finálním kódováním podpisu, umožnila samostatně vyhodnotit část podpisového výpočtu před vytvořením výsledné reprezentace podpisu. U implementace *Go* se v některých konfiguracích objevily statisticky významné výsledky pro charakteristiky související se

součiny $c \cdot s_1$ a $c \cdot s_2$. Tyto výsledky naznačují, že po vyloučení kódování podpisu se mohou projevit i jiné zdroje časové závislosti než ty pozorované při měření celého podpisu. Interpretaci těchto výsledků však omezuje skutečnost, že se je nepodařilo reprodukovat na všech testovaných procesorech. Implementace *dilithium-py* vykazovala širší časové závislosti napříč několika rekonstruovanými mezilehlými hodnotami, což odpovídá jejím odlišným implementačním vlastnostem a vyšší úrovni abstrakce.

Samostatně byly provedeny také třídové experimenty, ve kterých byly porovnávány naměřené časy podpisových operací provedených s různými klíči. U některých konfigurací se objevily statisticky významné rozdíly mezi třídami, jejich interpretace je však omezená. V části těchto experimentů byly použity klíče s různými hodnotami parametru ρ, a tedy i s různými maticemi A. Pozorované rozdíly proto nelze přímo přiřadit konkrétní tajné složce klíče ani konkrétní části podpisového algoritmu. Doplňkové experimenty s pevnou hodnotou ρ tuto nejednoznačnost částečně omezují, při použitém množství dat však nevedly k jednoznačnému závěru.

Celkově výsledky ukazují, že časové chování implementací ML-DSA závisí na kombinaci faktorů, mezi které patří konkrétní implementace, procesor, datová sada a hranice měření. Přestože práce neprokazuje prakticky zneužitelný časový útok, ukazuje, že vhodně navržená statistická analýza může odhalit slabší časové závislosti, které nejsou při běžném měření celého podpisového procesu snadno pozorovatelné.

Výstupem práce je experimentální nástroj, který pokrývá všechny kroky potřebné pro časovou analýzu ML-DSA a umožňuje studium časového chování. Tento nástroj vytváří základ pro další výzkum v oblasti postranních kanálů postkvantových podpisových schémat. Do budoucna se nabízí zejména rozšíření o další charakteristiky mezilehlých hodnot, zapojení dalších implementací a podrobnější analýza vlivu konkrétních procesorů.

VALA, Robin. *Side-Channel Analysis of the ML-DSA Algorithm Using Techniques Inspired by Marvin and Minerva*. Master's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2026. Advised by M.Sc. Sara Ricci, Ph.D.

Author's Declaration

Author:	Bc. Robin Vala
Author's ID:	241124
Paper type:	Master's Thesis
Academic year:	2025/26
Topic:	Side-Channel Analysis of the ML-DSA Algorithm Using Techniques Inspired by Marvin and Minerva

I declare that I have written my thesis independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis. I have not violated the principles and recommendations of the Brno University of Technology regarding the use of generative artificial intelligence.

As the author, I furthermore declare that, with respect to the creation of this thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank my advisor, M.Sc. Sara Ricci, Ph.D. for her guidance, feedback, and support during the preparation of this thesis.

I am especially grateful to M.Sc. Alicja Kario for her guidance throughout this thesis. Her explanations helped me understand difficult parts of the topic, and her feedback shaped the work from the initial ideas to the final evaluation. I greatly appreciate the time, patience, and effort she devoted to this thesis.

Contents

Introduction **16**

1 Preliminaries **18**
1.1 ML-DSA . . . 18
1.2 Side-channel attacks and timing leakage . . . 22
1.3 The Marvin attack . . . 25
1.4 The Minerva attack . . . 29
1.5 Statistical methods in Marvin and Minerva . . . 33
1.6 Implementation background . . . 37

2 Timing side-channel analysis of ML-DSA **40**
2.1 Measurement methodology . . . 41
2.1.1 Timing harness . . . 41
2.2 Data preparation . . . 46
2.2.1 Key generation . . . 47
2.2.2 Message generation . . . 47
2.2.3 Measurement scheduling . . . 48
2.2.4 Special dataset generation . . . 49
2.3 Reconstruction of intermediate values . . . 50
2.3.1 Reconstruction approach . . . 50
2.3.2 Selected intermediate values . . . 52
2.3.3 Feature extraction . . . 54
2.4 Statistical analysis . . . 56
2.4.1 Data representation . . . 57
2.4.2 Analysis procedure . . . 57
2.4.3 Sensitivity analysis via synthetic side-channels . . . 58

3 Experimental results and discussion **60**
3.1 Full-signing measurements . . . 61
3.1.1 Random messages . . . 61
3.1.2 Crafted dataset . . . 64
3.2 Partial-signing measurements . . . 68
3.2.1 Go . . . 69
3.2.2 OpenSSL and Nettle . . . 77
3.2.3 dilithium-py . . . 78
3.3 Key comparison experiments . . . 80
3.3.1 Random-message key comparison . . . 81

3.3.2 Crafted-message key comparison 83
3.3.3 Fixed-ρ key comparison 84
3.4 Discussion of results 87
3.4.1 Observed timing correlations 87
3.4.2 Weak or inconclusive signals 89
3.4.3 Negative results 90
3.4.4 Responsible disclosure 90

Conclusion **91**

Bibliography **93**

Symbols and abbreviations **97**

A Contents of the electronic attachment **98**

List of Figures

1.1 Side-channel and fault attacks on cryptographic implementations. . . 23
1.2 Structure of PKCS#1 v1.5 encryption block. 25
1.3 Timing evaluation in the Marvin attack. 27
1.4 Timing evaluation in the Minerva attack. 32
2.1 Measurement boundary for the full-signing harness. 44
2.2 Measurement boundary excluding signature encoding. 45
2.3 Generation of clean and early-rejection datasets. 50
2.4 Simplified dependency diagram of ML-DSA intermediate values. . . . 53
3.1 Confidence intervals for median of differences between `bit-size-w0` classes for *Go* ML-DSA-44, measured with the crafted dataset using the full-signing harness. 66
3.2 Confidence intervals for median of differences between `hw-w0` classes for *Go* ML-DSA-44, measured with the crafted dataset using the full-signing harness. 67
3.3 Confidence intervals for median of differences between `c-s1-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness. 71
3.4 Confidence intervals for median of differences between `c-s2-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness. 71
3.5 Confidence intervals for median of differences between `c-s1-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness on AMD EPYC 7742. 74
3.6 Confidence intervals for median of differences between `c-s2-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness on AMD EPYC 7742. 74
3.7 Confidence intervals for median of differences between `y-n-need-reduction` classes for *dilithium-py* ML-DSA-44, measured with the crafted dataset using the partial-signing harness. 80
3.8 Confidence intervals for median of differences between secret-key classes for *Nettle* ML-DSA-87 using random messages. 82
3.9 Confidence intervals for median of differences between secret-key classes for *OpenSSL* ML-DSA-44 using the crafted dataset. 84

List of Tables

1.1 ML-DSA parameter sets comparison. 18
2.1 Naming convention for timing harnesses in the `mldsa-toolkit` repository. 43
2.2 Data preparation scripts used in the `mldsa-toolkit` repository. . . . 46
2.3 Expected distribution of the number of rejected attempts before successful signing in ML-DSA [4]. 49
2.4 Extracted features from reconstructed intermediate values. 55
3.1 Evaluated ML-DSA implementations and their versions. 60
3.2 Selected timing analysis results for *OpenSSL* ML-DSA-65 with random messages. 61
3.3 Sensitivity analysis for `y-n-need-reduction` in the *OpenSSL* ML-DSA-65 random-message experiment. 62
3.4 Selected timing analysis results for *OpenSSL* ML-DSA-44 and ML-DSA-87 with random messages. 63
3.5 Summary of timing analysis results for the crafted dataset (full signing). 65
3.6 Selected timing analysis results for *dilithium-py* ML-DSA-44 with the crafted dataset. 67
3.7 Selected timing analysis results for the *Go* implementation in the partial-signing setting on Intel Xeon E5-2630 v3. 70
3.8 Representative timing analysis results for the *Go* implementation on negative platforms (partial-signing setting). 73
3.9 Timing analysis results for the *Go* implementation on Intel Xeon Gold 6438N (DOIT disabled vs. enabled). 76
3.10 Selected timing analysis results for *OpenSSL* and *Nettle* in the partial-signing setting. 77
3.11 Selected timing analysis results for *dilithium-py* in the partial-signing setting. 79
3.12 Summary of random-message key-comparison experiments. 81
3.13 Fixed-ρ key-comparison results using 250 000 crafted messages per key. 86

List of Algorithms

1 ML-DSA key generation 19
2 ML-DSA signing 20
3 ML-DSA verification 21

List of Listings

2.1 Timing measurement structure used in *OpenSSL*. 42
2.2 Simplified cycle counter implementation for x86_64. 42
2.3 Timing measurement used in *dilithium-py* harness. 43
2.4 Timing measurement used in the *Go* harness. 43
2.5 Key generation using *OpenSSL*. 47
2.6 Preparation of the ML-DSA reconstruction context in `extract.py`. . 51
2.7 Reconstruction of selected ML-DSA intermediate values. 52
2.8 Feature extraction for the reconstructed value y. 56
3.1 Fixed value of ρ used in the modified *OpenSSL* key-generation experiment. 85

Introduction

The move towards post-quantum cryptography has accelerated in recent years as advances in quantum computing begin to challenge the assumptions underlying today's public-key systems. The shift is motivated by long-term confidentiality requirements and by the risk that adversaries may collect encrypted data today and decrypt them in the future once quantum capabilities mature [1]. In response, the National Institute of Standards and Technology (NIST) [2] initiated a multi-year evaluation of post-quantum algorithms. In 2022, this process resulted in the selection of four primary algorithms, three signature schemes and one key encapsulation mechanism, with the majority based on lattice constructions. CRYSTALS-Dilithium [3], now standardized as the Module-Lattice Digital Signature Algorithm (ML-DSA) [4], was selected as the primary lattice-based signature mechanism intended for deployment in cryptographic systems.

As these schemes move towards practical deployment, their security must extend beyond the mathematical properties of the underlying hardness assumptions. Side-channel attacks [5, 6, 7] have demonstrated that cryptographic implementations may unintentionally leak sensitive information through observable effects such as execution time, power consumption, or microarchitectural behavior. Even very small timing variations can become observable through statistical techniques, which makes implementation-level analysis necessary for post-quantum schemes as well.

This thesis studies ML-DSA from the perspective of timing side-channel analysis. It builds on methodologies implemented in the Marvin and Minerva toolkits [8, 9], which were developed for the analysis of timing leakage in RSA and the Elliptic Curve Digital Signature Algorithm (ECDSA). Although the algebraic structure of ML-DSA differs significantly from these classical schemes, the same idea can be applied to ML-DSA by relating timing measurements to numerical properties of selected internal values and evaluating them statistically.

The main goal of this work is to determine whether the execution time of ML-DSA signing shows measurable dependence on selected intermediate values, especially those related to secret-dependent computations. Since ML-DSA does not expose a single obvious leakage target, this requires adapting existing timing-analysis ideas to a lattice-based signature scheme, choosing suitable measurement boundaries, selecting relevant intermediate values, and deriving numerical features that can be compared with signing time. The objective is not to break ML-DSA, but to study under which conditions these dependencies become observable.

The thesis is structured as follows. Chapter 1 introduces the background needed for the rest of the work. It introduces ML-DSA, side-channel attacks, the Marvin and Minerva attacks and their analysis toolkits, and the statistical methods and

implementations that are used later in the work.

Chapter 2 then describes how these ideas are adapted to ML-DSA. It presents the developed *mldsa-toolkit*, including the measurements setup, input data preparation, reconstruction of signing intermediates, feature extraction, and the way statistical tests are applied to the measurements.

Chapter 3 presents the experimental evaluation. The results are discussed according to the measurement boundary and experiment type, starting with full-signature measurements, continuing with partial-signing measurements without final signature encoding, and ending with key-comparison experiments. The chapter concludes by comparing the observed effects and discussing what can and cannot be inferred from the measurements.

During the preparation of this thesis, I used ChatGPT in thinking mode, DeepL, and Grammarly for language-related consultations, mainly for correcting grammatical errors, improving phrasing, and finding suitable wording or synonyms. After consulting these tools, I reviewed and edited the content myself, and I take full responsibility for the final version of the thesis.

1 Preliminaries

1.1 ML-DSA

ML-DSA is a post-quantum signature scheme standardized by NIST in FIPS 204 [4]. The scheme belongs to the family of lattice-based cryptography and is derived from CRYSTALS-Dilithium [3]. The security of ML-DSA relies on the Module Learning With Errors problem (MLWE) and the Module Short Integer Solution problem (MSIS). These problems are believed to remain resistant to quantum adversaries, which makes the scheme a suitable replacement for classical schemes like RSA and ECDSA.

At a high level, ML-DSA follows the Fiat–Shamir with aborts paradigm. The signer samples a short random vector, computes a commitment, derives a challenge using a hash function, and then produces a response. To ensure security, the algorithm applies rejection sampling, meaning that some signing attempts are discarded if certain bounds are not satisfied and the process is repeated.

ML-DSA defines three parameter sets

- ML-DSA-44 targeting NIST security category 2,
- ML-DSA-65 targeting NIST security category 3,
- ML-DSA-87 targeting NIST security category 5.

All parameter sets operate over the polynomial ring

$$R_q = \mathbb{Z}_q[X]/(X^{256} + 1),$$

with modulus $q = 8\,380\,417$ and polynomial degree $n = 256$.

The parameter sets differ in the dimensions (k, ℓ) of matrix A, the collision strength of $\tilde{c}$ denoted as λ, the coefficient range of the signing vector y given by γ_1, the low-order rounding bound γ_2, the number of nonzero coefficients τ in the challenge polynomial, and the maximum number ω of 1's allowed in the hint vector h. A comparison is provided in Table 1.1, the complete specification is listed in FIPS 204 [4].

Table 1.1: ML-DSA parameter sets comparison.

Parameter set	(k, ℓ)	λ	γ_1	γ_2	τ	ω	Security level
ML-DSA-44	$(4, 4)$	128	2^{17}	$(q-1)/88$	39	80	Category 2
ML-DSA-65	$(6, 5)$	192	2^{19}	$(q-1)/32$	49	55	Category 3
ML-DSA-87	$(8, 7)$	256	2^{19}	$(q-1)/32$	60	75	Category 5

The following sections define algorithms for key generation, signing and verification.

Key generation

ML-DSA key generation produces the public and private keys used in signing and verification. It is shown in Algorithm 1. The function starts by creating a 32-byte seed ξ using a random byte generator. This seed is then expanded into three internal values ρ, ρ' and K. The value ρ is used to reconstruct the public matrix A. The value ρ' determines the sampling of the short vectors s_1 and s_2. After these vectors are generated, the algorithm computes $t = A \cdot s_1 + s_2$ and splits the result into high and low parts. The high part t_1 is included in the public key together with ρ. The algorithm also computes the hash $\mathrm{tr} = H(pk, 64)$, which is stored in the private key to support the signing process. The private key stores the seeds and secret values used during signing. The function outputs both keys derived from the initial seed.

Algorithm 1. ML-DSA key generation

Output: Public key pk
Output: Private key sk
$\xi \leftarrow \textsc{RandomBytes}(32)$
$(\rho, \rho', K) \leftarrow \textsc{ExpandSeed}(\xi)$
$A \leftarrow \textsc{ReconstructMatrix}(\rho)$
$(s_1, s_2) \leftarrow \textsc{SampleShortVector}(\rho')$
$t \leftarrow A \cdot s_1 + s_2$
$(t_1, t_0) \leftarrow \textsc{Split}(t)$
$\mathrm{pk} \leftarrow (\rho, t_1)$
$\mathrm{tr} \leftarrow H(\mathrm{pk}, 64)$
$\mathrm{sk} \leftarrow (\rho, K, \mathrm{tr}, s_1, s_2, t_0)$
return $(\mathrm{pk}, \mathrm{sk})$

Signing

The signing function produces a signature for a given message using the private key. It follows the structure outlined in Algorithm 2. Two operational modes are supported. In the deterministic mode, all signatures are derived from the secret key and the message. In the hedged variant, each signing operation additionally incorporates fresh randomness.

The signer begins by extracting the internal values from private key and reconstructs the matrix A from public seed ρ. The message is combined with the hash of public key to obtain the message representative μ. A dedicated seed ρ'' is then computed to provide pseudorandomness during the signing attempt.

In the rejection loop, first the vector y is sampled from ρ'' and is used to compute the intermediate vector w. The high bits w_1 are extracted and used as an input for

the computation of the commitment hash $\tilde{c}$. This value is used to derive the challenge polynomial c. The vector z is obtained as a combination of y and the product of challenge polynomial c and the secret vector s_1. Lastly the hint h is created from internal values derived from c, w, s_2 and t_0.

Each candidate pair (z, h) is checked against the required bounds. If the attempt violates any constraint, a new value of y is sampled and the process repeats. Once a valid pair is produced, the signature is formed as $(z, h, \tilde{c})$.

Algorithm 2. ML-DSA signing

Input: Secret key sk
Input: Message m
Output: Signature $\sigma = (z, h, \tilde{c})$
rnd $\leftarrow$ RandomBytes(32) ▷ for deterministic mode use rnd $= \{0\}^{32}$
Extract $(\rho, K, \mathrm{tr}, s_1, s_2, t_0)$ from sk
$A \leftarrow$ ReconstructMatrix(ρ)
$\mu \leftarrow H(\mathrm{tr} \,\|\, m, 64)$
$\rho'' \leftarrow H(K \,\|\, \mathrm{rnd} \,\|\, \mu, 64)$
$\kappa \leftarrow 0$
$(z, h) \leftarrow \perp$
while $(z, h) = \perp$ **do**
  $y \leftarrow$ SampleShortVector(ρ'', κ)
  $w \leftarrow A \cdot y$
  $w_1 \leftarrow$ HighBits(w)
  $\tilde{c} \leftarrow H(\mu \,\|\, w_1, \lambda/4)$
  $c \leftarrow$ DeriveChallenge($\tilde{c}$)
  $z \leftarrow y + cs_1$
  if *InvalidZ(z)* **then**
    continue
  $h \leftarrow$ ComputeHint(c, w, s_2, t_0)
  if *TooManyNonzeros(h, ω)* **then**
    $(z, h) \leftarrow \perp$
    continue
  $\kappa \leftarrow \kappa + \ell$
return $\sigma = (z, h, \tilde{c})$

Verification

Algorithm 3 describes the steps of a verifier who is checking the validity of a signature. The verifier begins by reconstructing the matrix A from the public seed ρ. The value z must satisfy bounds and the hint h must be properly byte-encoded. Any violation leads to rejection.

After these preliminary checks, the verifier recomputes the public key hash tr and uses it together with the message to form the message representative μ. The challenge polynomial c is reconstructed from the commitment hash included in the signature. The verifier then computes the value $w' = A \cdot z - c \cdot t_1$ and applies the hint to recover w'_1.

The commitment hash is recomputed from μ and w'_1. The signature is accepted only when the recomputed value matches the commitment hash carried in the signature. This confirms that the response is consistent with the public key and that the signer used a valid private key when producing the signature.

Algorithm 3. ML-DSA verification

Input: Public key $\text{pk} = (\rho, t_1)$
Input: Message m
Input: Signature $\sigma = (z, h, \tilde{c})$
Output: Accept or Reject
$A \leftarrow \textsc{ReconstructMatrix}(\rho)$
if *InvalidZ(z)* **then**
  Reject;
if *InvalidHint(h, ω)* **then**
  Reject;
$\text{tr} \leftarrow H(\text{pk}, 64)$
$\mu \leftarrow H(\text{tr} \,\|\, m, 64)$
$c \leftarrow \textsc{DeriveChallenge}(\tilde{c})$
$w' \leftarrow Az - c\,t_1$
$w'_1 \leftarrow \textsc{UseHint}(h, w')$
$\tilde{c}' \leftarrow H(\mu \,\|\, w'_1, \lambda/4)$
return $\tilde{c}' = \tilde{c}$

The three algorithms together describe the complete operation of ML-DSA. The focus of this thesis is on the signing procedure and the intermediate values produced during its execution, since these values can have an impact on the timing behavior. The practical part examines these intermediate values and evaluates whether such timing dependencies can be observed in practice, with particular attention to the deterministic signing mode.

1.2 Side-channel attacks and timing leakage

Cryptographic algorithms are designed to resist mathematical attacks under well-defined hardness assumptions. Real implementations may, however, reveal information through effects that are outside the abstract algorithm. These effects include physical properties of the device, such as power consumption or electromagnetic radiation, and microarchitectural behavior, such as cache activity. This unintended leakage is referred to as a side channel. A side-channel attack exploits the implementation of a cryptographic construction rather than the construction itself.

Side-channel attacks can use different sources of information. Some attacks observe physical properties of the device during computation. Power analysis [10] uses variations in power consumption caused by the switching activity of electronic circuits. Simple power analysis may reveal information directly from visible patterns in a power trace. Differential power analysis can detect weaker data-dependent effects by applying statistical processing to many measurements. Electromagnetic analysis [11] measures radiation emitted by the device during computation and uses data-dependent variations in this signal to infer information about internal computations. Acoustic attacks [12] show that leakage is not limited to directly measured electrical signals, since sound produced by electronic components may also carry information about the computation.

Other side-channel attacks exploit effects inside modern processors. Timing attacks [5, 6] use differences in the duration of operations and are especially relevant when execution time depends on secret data or on values derived from it. Cache attacks [13] observe memory access patterns through timing differences between cached and non-cached accesses. This can reveal which memory locations were accessed by another computation, which is dangerous when memory accesses depend on secret values. Speculative execution attacks, such as Spectre [14], exploit microarchitectural side effects caused by instructions executed speculatively. Even if the architectural state of the processor is later restored, traces left in caches or other microarchitectural state can still leak information.

Fault attacks [15] form a related class of implementation attacks, where the adversary intentionally induces errors in the computation and studies the behavior. These faults may affect registers, memory, or intermediate values, and the resulting incorrect outputs can reveal information about secret data or internal state.

The listed examples prove that an implementation may leak information even when the cryptographic scheme is mathematically secure. Side-channel attacks are mainly passive, because the adversary observes infomation leaked by a normal execution. Fault attacks are active, because the adversary interferes with the computation. Figure 1.1 gives a simple overview of these two classes of implementation attacks.

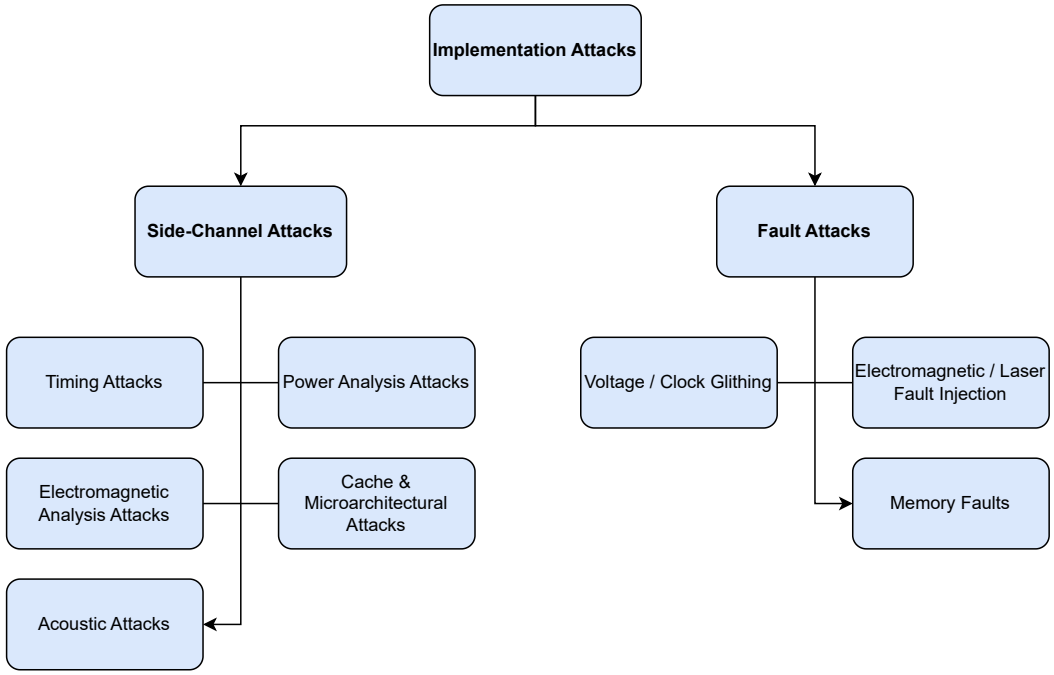


Fig. 1.1: Side-channel and fault attacks on cryptographic implementations.

Timing side-channels

This thesis focuses on timing side channels, first introduced by Kocher [5]. A timing side channel arises when the execution time of a cryptographic operation depends on secret data. These differences may be small, but if they are systematic, repeated measurements can make them statistically detectable.

Unlike many physical side channels, timing leakage can sometimes be observed remotely, for example through the response time of a network service. This makes it relevant for protocols such as Transport Layer Security (TLS). Remote measurements are affected by network noise, but practical attacks have shown that useful timing information can still be extracted in such settings [16].

Timing variability can originate from several sources. Data-dependent branches may cause different execution paths to be taken. Memory accesses may introduce variation through cache effects, because cached data can be accessed faster than data fetched from main memory. Some arithmetic operations may also exhibit input-dependent latency on certain architectures.

The implementation does not behave in constant time when such dependencies exist. Constant-time implementations aim to make secret-dependent values independent of observable timing behavior, typically by avoiding secret-dependent control flow and memory access patterns. Achieving this property is difficult in practice, because compiler optimizations may transform code in unexpected ways

and microarchitectural effects may introduce timing variation that is not visible at the source-code level. For this reason, code inspection alone is not always sufficient to assess timing behavior. Measurement-based analysis is needed to complement implementation review and to test whether execution time depends on secret or secret-derived values [16].

Real-world attacks confirm that this leakage remains relevant. The Marvin attack [6] demonstrated that several TLS implementations of RSA decryption still exhibited timing differences that enabled Bleichenbacher-style attacks. Similarly, the Minerva attack [7] showed that timing measurements can reveal information about secret-dependent intermediate values.

Relevance to ML-DSA

Timing side channels are particularly relevant for modern lattice-based schemes such as ML-DSA. This signing algorithm consists of multiple operations on vectors of polynomials, including the Number Theoretic Transform (NTT), modular reductions, and decomposition into high and low bits. Although these operations are designed to be efficient, their implementation may exhibit input-dependent behavior due to memory access patterns or arithmetic operations.

In addition, the signing algorithm relies on rejection sampling. The number of iterations required to produce a valid signature depends on intermediate values computed during signing. This behavior is an inherent part of the algorithm, but it may introduce variability in execution time that may be observable depending on the implementation.

The ML-DSA specification [4] itself treats side channels as relevant implementation concerns. This is reflected in the signing modes defined by the standard. The default mode is *hedged*, where fresh randomness is incorporated during signing to reduce the risks associated with side-channel leakage. A *deterministic* variant is also permitted, but the standard notes that deterministic signing may leak information about the message through timing side channels.

The standard further emphasizes that intermediate values produced during key generation and signing may contain sensitive information, even when they are not part of the final public signature. Such values must be destroyed when they are no longer needed, and internal values from unsuccessful signing iterations must not be released because they may leak information about the private key.

For timing analysis, ML-DSA therefore has to be considered as an implemented signing procedure, not only as an abstract mathematical scheme. The relevant question is whether the execution time of an implementation correlates with properties

of intermediate values produced during signing, and how this behavior is affected by rejection sampling, signing mode, and implementation-level choices.

1.3 The Marvin attack

The Marvin attack [6] is a modern timing variant of the classical Bleichenbacher attack against RSA PKCS#1 v1.5 padding. It shows that RSA decryption can remain vulnerable even when implementations no longer expose explicit padding errors. Marvin uses small differences in execution time instead of relying on error messages. These differences form a timing oracle that can be used in a Bleichenbacher-style attack and, for vulnerable implementations, can lead to ciphertext decryption.

The connection to Bleichenbacher is important because Marvin uses the same oracle-based attack structure. The difference is that the oracle is not exposed through error messages, but inferred from timing measurements. The original attack and its oracle model are therefore recalled first.

The Bleichenbacher attack [17] is a chosen-ciphertext attack against RSA encryption with PKCS#1 v1.5 padding. It relies on the fact that an implementation may reveal whether a decrypted plaintext satisfies the required padding format, thereby providing an oracle that distinguishes between conforming and non-conforming inputs.

The PKCS#1 v1.5 encryption scheme [18] uses a structured padding format, shown in Figure 1.2. A valid plaintext begins with a fixed prefix (`0x00`, `0x02`), followed by a padding string of nonzero bytes, denoted as `PS`, a delimiter byte equal to zero, and the actual data, denoted as `D`.

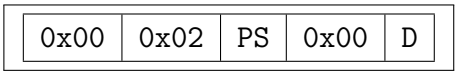


Fig. 1.2: Structure of PKCS#1 v1.5 encryption block.

During decryption, the implementation checks whether this structure is satisfied. If the outcome of this check is exposed, even indirectly, it provides a binary oracle that indicates whether the plaintext is correctly padded. This oracle forms the basis of the attack.

The attack uses this oracle adaptively. Starting from a given ciphertext, the attacker crafts modified ciphertexts and submits them to the decryption routine. Based on the responses, the attacker refines the set of possible plaintext values.

The effectiveness of this approach relies on the multiplicative property of RSA. Given a ciphertext $c = m^e \bmod n$, an attacker can construct modified ciphertexts of the form $c' = c \cdot s^e \bmod n$, which correspond to a plaintext $m' = m \cdot s \bmod n$

for a chosen value s. This allows the attacker to transform the unknown plaintext in a controlled way and observe how the oracle responds to these modified inputs. By combining these responses across multiple queries, the attacker can progressively narrow down the set of possible plaintext values until the plaintext is uniquely determined.

In the original attack, this oracle is explicit, for example through distinct error messages. Many implementations were later modified to return uniform errors in order to hide this information. However, uniform error reporting does not eliminate the oracle if the internal processing of inputs remains input-dependent. Differences during padding validation or related processing may still affect execution time. If these timing differences are observable, they provide an implicit oracle that distinguishes between classes of inputs. This is the setting exploited by Marvin, where the attacker does not observe a direct valid or invalid response, but instead distinguishes inputs using timing differences that reflect whether the underlying plaintext satisfies specific padding conditions.

The Marvin paper showed that this type of timing leakage was practically detectable in several real-world implementations. The attack was demonstrated against multiple cryptographic libraries and protocols, including TLS. The evaluation covered widely used libraries such as OpenSSL, Bouncy Castle and libgcrypt, as well as TLS stacks such as GnuTLS and the Linux kernel [19].

The analysis [6] connected the observed leakage to input-dependent behavior inside RSA decryption implementations. The relevant sources included padding validation, control flow, and handling of intermediate values. In several evaluated implementations, part of the leakage was linked to general-purpose multi-precision arithmetic. For example, OpenSSL and NSS used integer representations with a variable number of machine words and normalized these representations by removing leading zero words. Values obtained from RSA decryption could therefore be processed with different internal lengths. Later operations, including conversion to byte strings for padding validation, could then take different amounts of time depending on the magnitude of the plaintext. This created measurable timing differences between input classes.

The paper also shows how this leakage can be detected experimentally. The analysis is based on repeated timing measurements and non-parametric statistical tests, including the sign test and the Wilcoxon signed-rank test. These methods were able to detect timing differences on the order of tens of nanoseconds. The results also show that the reliability of the measurements is important, since a small but stable timing difference can be more useful than a larger effect hidden by noise.

Evaluation of timing leakage

The Marvin toolkit [8] provides tools for testing whether an RSA PKCS#1 v1.5 implementation exposes a timing oracle. The toolkit measures decryption times for ciphertexts with controlled PKCS#1 v1.5 structure and checks whether different ciphertext classes are processed in measurably different amounts of time. It generates the ciphertext classes, measures the decryption behavior of the implementation under test, associates each timing measurement with the corresponding class, and applies statistical tests. A statistically significant timing difference between classes indicates that the implementation is vulnerable to the Marvin attack. The evaluation can be divided into three phases: measurement, extraction, and analysis, as illustrated in Figure 1.3.

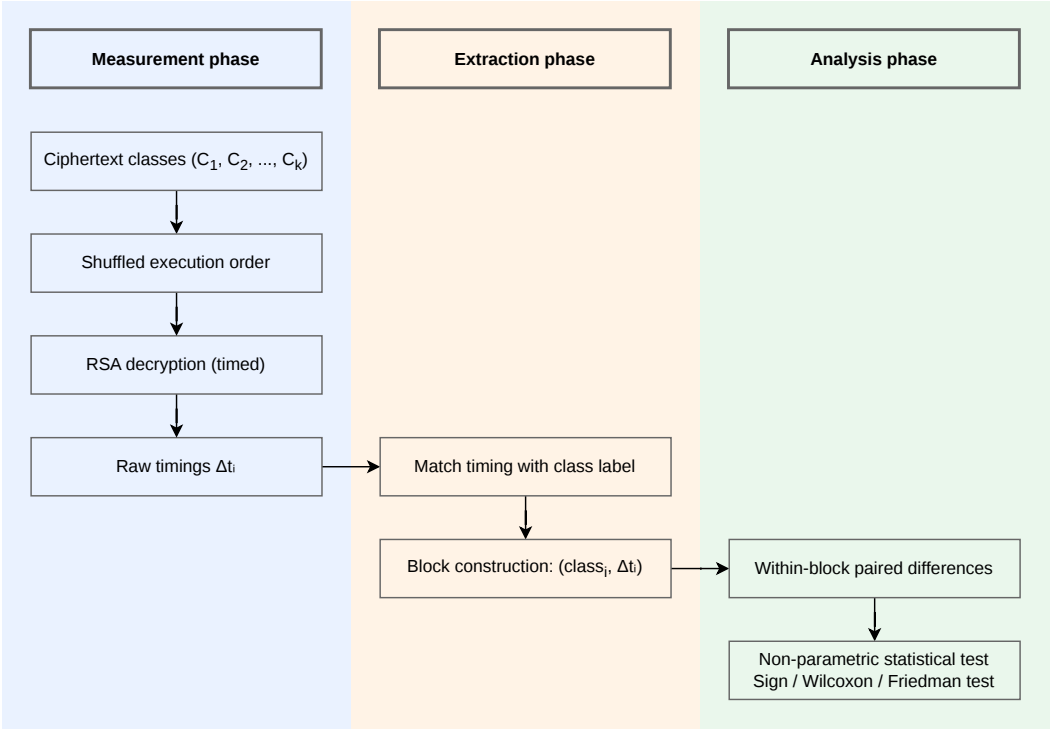


Fig. 1.3: Timing evaluation in the Marvin attack.

In the first phase, a dataset of ciphertexts is constructed and divided into classes $(C_1, C_2, \ldots, C_k)$, where each class corresponds to a specific PKCS#1 v1.5 padding structure. These classes represent different cases that the RSA decryption routine must handle during padding validation. They include correctly formatted ciphertexts as well as malformed ones, such as those with invalid header bytes, incorrect placement of the delimiter, zero bytes in the padding string, or completely invalid structure. Each class therefore reflects a different behavior of the padding verification logic.

Once the ciphertexts are generated, they are arranged into a single shuffled sequence to ensure that inputs from different classes are interleaved during execution. The order and class labels are recorded in a preparation log, which is later used to associate timing measurements with the corresponding inputs.

Each ciphertext in the shuffled sequence is processed exactly once by the RSA decryption routine. For every decryption, timestamps are taken immediately before and after the operation. The difference between these timestamps is recorded as the execution time Δt_i for the given ciphertext. This produces a sequence of raw timing values $(\Delta t_1, \Delta t_2, \ldots, \Delta t_n)$ corresponding to the shuffled inputs. At this stage, the timing data is unlabeled with respect to the ciphertext classes and reflects only the observed execution times of the decryption routine.

In the second phase, the timing values are matched with their ciphertext classes using the preparation log (i.e. each timing value Δt_i is paired with its class label C_j).

In the third phase, the labeled measurements are grouped into small consecutive blocks that reflect the order of execution. Each block contains a mixture of ciphertexts from different classes that were evaluated under similar system conditions. Within each block, the timing values of different classes are compared, and pairwise differences are computed to capture relative timing behavior.

The set of within-block differences obtained across all blocks forms the input to the statistical evaluation. Non-parametric tests such as the sign test and Wilcoxon signed-rank test are used to assess whether the observed differences are consistently biased in one direction. When multiple classes are considered, the Friedman test evaluates whether there is a systematic difference in execution time across all classes.

A statistically significant result indicates that execution time depends on the ciphertext class, which reveals input-dependent behavior in the decryption routine.

Mitigation strategies

The main conclusion of the Marvin paper [6] is that RSA PKCS#1 v1.5 decryption is difficult to implement safely against timing attacks. The preferred mitigation is to avoid this padding mode where possible, rather than attempting to repair individual sources of leakage. This can be done by using protocols and APIs that rely on more modern mechanisms, such as RSA-OAEP, or by avoiding RSA-based key transport entirely.

For systems that still need to support PKCS#1 v1.5 decryption, the paper proposes implicit rejection. With this countermeasure, invalid padding does not produce a distinguishable error output. The implementation instead derives a pseudorandom replacement message from the private key and the ciphertext. The goal is to make valid and invalid ciphertexts indistinguishable at the API level. However, this protection is effective only if the whole decryption and rejection procedure avoids secret-dependent timing behavior.

The implementation requirements are therefore broader than the padding check itself. Operations that process the decrypted value, such as unblinding, conversion between integer and byte-string representations, and padding validation, must avoid input-dependent control flow, memory access, and variable-size arithmetic effects. The Marvin results show that general-purpose multi-precision arithmetic can be problematic in this setting, especially when integer representations depend on the magnitude of secret-derived values.

Blinding remains an important countermeasure, but it does not by itself remove all leakage. It can hide some input relationships during RSA exponentiation, but later processing may still operate on secret-derived plaintext values. For this reason, the paper treats blinding as a useful component of a defense, not as a complete mitigation.

1.4 The Minerva attack

Minerva [7] is a timing side-channel attack that targets implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA). It exploits the fact that the execution time of the signing operation can depend on the bit-length of the ephemeral nonce k used in each signature. Partial information about the nonce can reveal information about the private key. When such leakage is accumulated over many signatures, it can be used to recover the private key under suitable conditions. Unlike Bleichenbacher-style attacks, this approach does not rely on a binary oracle but on statistical correlation between execution time and nonce-dependent features.

The attack focuses on the implementation of scalar multiplication $[k]G$. In some implementations, the scalar multiplication algorithm processes the scalar only up to its most significant nonzero bit. The number of loop iterations then depends on the bit-length of k. As a result, signatures created with shorter nonces require fewer iterations and may be faster to compute. This creates a correlation between runtime and the bit-length of the nonce.

Although the timing differences are small, they can be consistent across executions. This makes the dependence detectable through statistical analysis when enough measurements are available. In addition to bit-length, some implementations may exhibit weaker dependence on other properties of the nonce, such as its Hamming weight, although this is not the primary source of leakage exploited in Minerva.

Other implementation choices can also contribute to the leakage, such as incomplete elliptic curve formulas that may introduce data-dependent behavior when intermediate values encounter exceptional cases. This may cause the execution time to depend on secret nonce-derived values.

The impact of the leakage is amplified by the structure of ECDSA, where the nonce is linearly related to the private key. Partial information about multiple nonces can be combined into a hidden number problem, which can be solved using lattice-based techniques. With a sufficient number of observations, this enables recovery of the private key.

The paper also shows that deterministic nonces, for example as specified in RFC 6979 [20], do not mitigate this form of leakage. Deterministic nonce generation removes the need for external randomness during signing, but it does not remove the dependence of scalar multiplication on the nonce value. Moreover, when the same message is signed repeatedly, the same nonce is generated, which allows timing measurements to be averaged and the noise to be reduced.

Impact of Minerva-type leakage

The Minerva study [7] evaluated the attack on several software libraries, including `libgcrypt`, `MatrixSSL`, `SunEC`, `Crypto++`, and `wolfSSL`, as well as on a certified smartcard platform. In these implementations, the bit-length of the nonce produced measurable differences in execution time.

The observed leakage was sufficient for practical key recovery. In the reported experiments, private keys were recovered from approximately 1,200 signatures in software implementations such as `libgcrypt`, and from around 2,100 signatures for the evaluated smartcard platform.

The study further showed that the leakage is not limited to timing measurements. In the smartcard experiments, the same dependence on nonce properties was observed in power traces, indicating that the source of the leakage lies in data-dependent computation rather than in a specific measurement channel.

The presence of this leakage in certified hardware devices is particularly notable. The study suggests that such implementations may include both performance-optimized and side-channel-resistant variants of elliptic curve operations. In some cases, an optimized variant without sufficient side-channel protections may be used for secret-dependent computations, leading to observable timing differences.

To mitigate this form of leakage, the Minerva study recommends several implementation-level countermeasures. In particular, scalar multiplication should use complete addition formulas that behave consistently for all inputs, including exceptional cases. Additionally, fixing the bit-length of nonces, for example by adding a multiple of the group order, removes the dependency between execution time and nonce length. Another approach is to use longer nonces, reducing the usefulness of partial information about the most significant bits.

Detection of Minerva-type leakage

The Minerva toolkit [9] provides tools for testing whether a signing implementation exposes timing leakage related to the ephemeral nonce. The toolkit measures signing times and relates them to numerical features derived from nonce-related internal values, most notably the bit length of the nonce.

This differs from the Marvin evaluation. Marvin compares timing behavior across predefined input classes, whereas Minerva studies whether execution time changes with numerical properties of internal values. The toolkit generates signatures, collects timing measurements, derives the relevant features, and applies statistical analysis. Similarly to Marvin, the evaluation can be divided into three phases, namely measurement, extraction, and analysis, as illustrated in Figure 1.4.

In the first phase, a dataset of messages $(m_1, m_2, \ldots, m_n)$ is prepared and signed using a fixed private key. Each message produces a corresponding signature (r_i, s_i). For each signing operation, timestamps are recorded immediately before and after execution, and their difference is stored as the execution time Δt_i. This produces a sequence of raw timing values $(\Delta t_1, \Delta t_2, \ldots, \Delta t_n)$ corresponding to the processed messages.

At this stage, the timing data is unlabeled with respect to the nonce-dependent properties and reflects only the observed execution time of the signing routine.

In the second stage, the nonce k_i used during each signing operation is reconstructed offline from the signature (r_i, s_i) and the known private key. From the

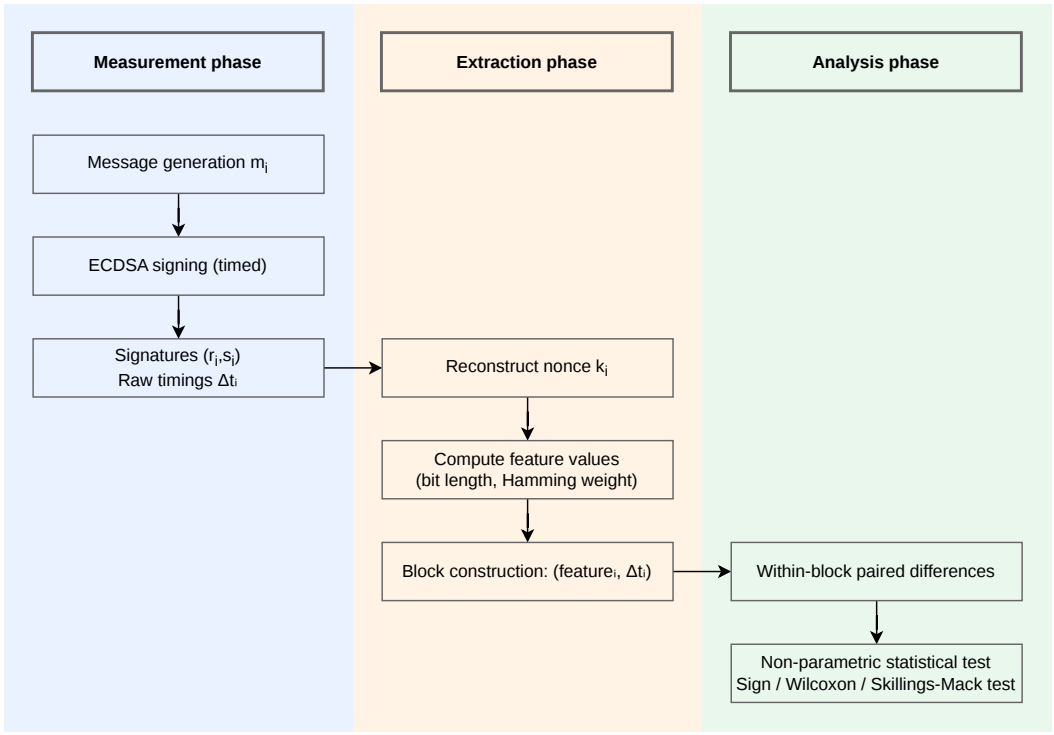


Fig. 1.4: Timing evaluation in the Minerva attack.

reconstructed nonce, a set of features is derived that may influence execution time. These features include the bit-length of k_i, the bit-length of its modular inverse, and their Hamming weights. Each timing value Δt_i is then paired with the feature value v_i derived from the same signing operation, which produces labeled data of the form $(v_i, \Delta t_i)$.

In the third stage, the labeled measurements are grouped into small consecutive blocks that reflect the order of execution. Each block contains measurements obtained under similar system conditions. Within each block, timing values are compared across different feature values, and pairwise differences are computed to capture relative timing behavior.

The set of within-block differences obtained across all blocks form the input to the statistical evaluation. Non-parametric tests such as the sign test and Wilcoxon signed-rank test are used to assess whether the observed differences are consistently biased in one direction. When multiple feature groups are considered, the Skillings–Mack test is used to evaluate whether execution time depends on the selected feature.

A statistically significant result indicates that the execution time differs with respect to the analyzed nonce-derived feature, which suggests feature-dependent tming behavior in the signing routine.

1.5 Statistical methods in Marvin and Minerva

The Marvin and Minerva toolkits use statistical methods to determine whether execution time depends on selected properties or processed data. In Marvin, this corresponds to predefined classes of RSA ciphertexts or plaintext structures. In Minerva, this correspond to features derived from internal values, most importantly the bit-length of the ECDSA nonce. Although the source of the grouping differs, both analyses aim to detect whether one group of measurements is consistently faster or slower than another.

The main difficulty is that the timing differences are small compared to measurement noise. Individual measurements are therefore not meaningful on their own. Moreover, timing measurements are often not independent. Consecutive measurements may be affected in similar ways by system load, CPU state, cache effects, scheduling, and network conditions.. Directly comparing absolute timing values can therefore lead to misleading results.

To reduce this problem, the analysis compares related measurements obtained under similar conditions. Measurements are organized into small blocks or rounds that preserve the order of execution. Comparisons are then performed within each block, so that shared noise affects all compared measurements in approximately the same way. This shifts the analysis from absolute timing values to relative timing differences.

The statistical evaluation is based on hypothesis testing. The null hypothesis assumes that no timing dependency is present and that any observed differences are caused by noise. Each statistical test produces a p-value, which expresses how likely the observed differences are under the null hypothesis. A result is considered statistically significant when the p-value falls below a selected threshold. In Marvin and Minerva timing experiments, strict threshold of $\alpha = 10^{-5}$ is used to reduce the probability of false positives [21, 22].

The analysis relies on non-parametric statistical tests as they operate on signs or ranks of measurements and do not require the timing data to follow a specific probability distribution. this makes them suitable for timing experiments, where the noise is often non-stationary, correlated, and affected by outliers.

In addition to the statistical methodology itself, the experimental setup must be carefully designed to avoid introducing artificial biases. In particular, measurements should be performed in a randomized order to prevent correlations between input classes and the internal state of the system. Inputs are prepared in advance and randomized independently of the measurement code, so that the execution does not depend on the class being measured.

Another important factor is the consistency of the implementation. The detectability of timing leakage depends not only on the magnitude of the timing differences, but also on the variability of the measurements. Smaller timing differences can be easier to detect in implementations with stable execution times, while larger differences may remain undetectable in highly noisy environments [6].

Finally, the evaluation may include control inputs that are not expected to produce timing differences. These inputs serve as a sanity check for the measurement setup. If the statistical analysis reports a significant difference for such inputs, the result may indicate an experimental artifact rather than leakage from the implementation.

The following sections describe the statistical tests used in the Marvin and Minerva analyses.

Sign test

The sign test [21, 22] is a non-parametric paired test that evaluates whether two related timing sequences differ systematically. For each pair of measurements, the difference is computed and only its sign is retained, indicating which value is larger. Ties are discarded. The test then counts the number of positive and negative outcomes across all pairs.

Under the null hypothesis, positive and negative differences are equally likely, and the number of positive outcomes follows a binomial distribution with parameter $p = 0.5$. The p-value is computed based on the observed imbalance between positive and negative signs.

Since the test considers only the direction of differences and not their magnitude, it is robust against outliers and large variations in timing values. A statistically significant result indicates that one group of measurements is consistently faster or slower than the other.

Wilcoxon signed-rank test

The Wilcoxon signed-rank test [21, 22] extends the sign test by incorporating information about the magnitude of the differences. For each pair of measurements, the difference is computed, and its absolute value is ranked. Each rank is then assigned the sign of the original difference, and the sum of these signed ranks is used to compute a p-value.

Under the null hypothesis, positive and negative differences of similar magnitude are expected to balance out. A consistent bias in the signed ranks indicates that one group of measurements is systematically larger or smaller than the other.

Unlike the sign test, the Wilcoxon test takes into account the relative size of the differences, which generally increases its sensitivity to timing shifts. However, this improvement relies on the assumption that the distribution of pairwise differences is approximately symmetric. If this assumption is violated, the sign test may provide more robust results.

Friedman test

The Friedman test [21, 22] is a non-parametric test for comparing more than two related groups of measurements. Within each block, the timing values are ranked, and these ranks are then aggregated across all blocks for each group. The test evaluates whether the observed rank patterns differ systematically between groups.

Under the null hypothesis, all groups are expected to have similar rank distributions. A statistically significant result indicates that at least one group is consistently faster or slower than the others.

By evaluating all groups simultaneously, the Friedman test can be more sensitive than pairwise tests when multiple input classes or feature values are present.

Skillings–Mack test

The Skillings–Mack test [23] is a generalization of the Friedman test that allows for missing or unbalanced observations. As in the Friedman test, timing values are ranked within each block, and these ranks are aggregated across blocks for each group.

In contrast to the Friedman test, which requires a complete and balanced dataset, the Skillings–Mack test can handle blocks where some measurements are missing. This is achieved by adjusting and weighting the within-block ranks so that comparisons remain consistent across blocks with different numbers of observations.

Under the null hypothesis, all groups are expected to have similar aggregated rank values. A statistically significant result indicates that at least one group is systematically faster or slower than the others.

This makes the Skillings–Mack test particularly suitable for timing analysis scenarios where data may be incomplete or unevenly distributed across groups.

Bootstrapping

Bootstrapping [21, 22] is a resampling method used to estimate the magnitude and variability of timing differences. Unlike the previously described statistical tests, which determine whether a timing dependency exists, bootstrapping is used to quantify the size of the effect and provide a confidence interval.

The method operates on paired differences between related measurements. From the observed dataset, new samples are repeatedly generated by sampling with replacement. For each resampled dataset, an effect size is computed using a robust estimator such as the median, trimmed mean, or trimean. The distribution of these resampled effect values is then used to construct a confidence interval for the timing difference.

The use of robust estimators is important in timing analysis, as measurement data often contains outliers and does not follow a normal distribution. By relying on medians or trimmed means, the analysis reduces the influence of extreme values and provides a more reliable estimate of the timing behavior.

The resulting confidence interval provides a direct interpretation of the observed effect. If the interval does not include zero, it indicates that the timing difference is consistently positive or negative, confirming the presence of input-dependent behavior. Conversely, if the interval is narrow and centered around zero, it provides evidence that no measurable timing difference is present within the resolution of the experiment.

Bootstrapping is used together with hypothesis testing. A statistically significant p-value indicates the presence of a timing dependency, and the bootstrapped confidence interval quantifies its magnitude.

The outcome of these tests determines whether execution time exhibits a systematic dependence on the selected grouping criterion. In the Marvin setting, this corresponds to distinguishing between different classes of inputs, while in the Minerva setting it reflects dependence on properties of internal values. A statistically significant result indicates the presence of timing-dependent behavior that may serve as a basis for further analysis.

Application to ML-DSA

The methodology described above can be applied beyond the original Marvin and Minerva settings. In this thesis, it is adapted to the analysis of ML-DSA implementations, where the structure of the computation differs significantly from both RSA and ECDSA.

Unlike Marvin, which compares execution time across different classess of inputs, and Minerva, which studies the dependence of execution time on properties of the ephemeral nonce, ML-DSA does not provide natural separation into input classes or a single dominant source of leakage. Instead, the analysis must determine whether execution time depends on properties of multiple intermediate values computed during the signing procedure.

Nevertheless, both approaches provide useful guidance for this setting. From the Marvin analysis, the idea of comparing related groups of measurements can be reused when evaluating timing differences across secret keys.

The Minerva approach is more directly applicable. In particular, the idea of relating execution time to properties of secret-dependent intermediate values can be adapted to ML-DSA. However, this requires adapting the notion of intermediate values from scalar quantities to structured objects such as vectors of polynomials.

This difference has two main consequences. First, the analysis operates on a larger scale, as each signing operation involves many intermediate values rather than a single scalar. Second, the relevant properties are more complex, since they are derived from high-dimensional data rather than from a single value.

Despite these differences, the underlying statistical methodology remains unchanged. The same non-parametric tests can be used to evaluate whether observed timing differences are statistically significant and whether they indicate timing-dependent behavior in ML-DSA implementations.

1.6 Implementation background

This chapter describes the libraries and software components used in the practical part of the thesis and their roles in the analysis.

The *dilithium-py* library is used as a reference implementation for reconstruction of intermediate values during the extraction phase. *OpenSSL*, the *Go* implementation, and the *Nettle* library serve as target implementations for timing analysis.

For each implementation, dedicated measurement harnesses record execution time of the ML-DSA signing operation over many inputs. In selected cases, minor modifications are applied to isolate specific parts of the signing procedure.

The *tlsfuzzer* is used for extraction and statistical analysis. It processes the collected timing data, associates it with reconstructed values, and evaluates whether execution time depends on selected properties of internal computations.

dilithium-py

Dilithium-py [24] is a Python implementation of CRYSTALS-Dilithium and ML-DSA as defined in FIPS 204. Its structure follows the specification, which makes it useful for inspecting individual steps of the signing procedure. The implementation is not optimized for performance, but this is acceptable for its role in this thesis.

In this thesis, *dilithium-py* is used to reconstruct selected intermediate values of the ML-DSA signing algorithm. Given a message, secret key, and signature, the implementation can reproduce the computations needed to derive values such as the

masking vector, intermediate products, and decomposition results. These values are then used for feature extraction and comparison with externally measured signing times.

The library is also useful because it supports deterministic signing. This makes the reconstructed values reproducible for the same input data, which is necessary when matching them with timing measurements.

OpenSSL

OpenSSL [25] is a widely used open-source cryptographic library that provides implementations of many cryptographic primitives and protocols, including TLS. Recent versions include support for post-quantum algorithms.

The ML-DSA implementation in OpenSSL is integrated into the library's provider framework. It is accessed through the EVP interface, which provides a high-level API for cryptographic operations. In this interface, ML-DSA signing is invoked through the general OpenSSL signing API rather than by calling implementation-specific signing code directly.

Go implementation

Go [26] is an open-source programming language developed at Google, designed for efficient systems programming and distributed with a standard library that includes cryptographic functionality. It includes an implementation of ML-DSA in the `crypto/internal/fips140/mldsa` package [27]. This implementation forms part of the language's internal infrastructure and follows the ML-DSA specification.

The package is internal and is not exposed as a public API for general use. As a result, using it for signing requires more direct interaction with the implementation than in libraries that provide a high-level signing interface.

Nettle implementation

Nettle [28] is a low-level cryptographic library written in C, designed to provide basic cryptographic primitives with a simple and flexible interface. In this thesis, a development version of Nettle [29] is evaluated, which contains experimental support for ML-DSA. At the time of the experiments, this implementation supported the ML-DSA-65 and ML-DSA-87 parameter sets.

Since the ML-DSA support is not part of an official release, access to the signing functionality requires a custom test setup.

tlsfuzzer

The tlsfuzzer [22, 30] is a Python test suite for TLS implementation testing. It provides tools for constructing protocol test cases, executing them against a target implementation, and processing the resulting data. In addition to protocol testing functionality, the project includes utilities for timing-data extraction and statistical analysis.

In this thesis, two components of tlsfuzzer are particularly relevant, the extraction script and the analysis script.

The extraction script is responsible for parsing measurement logs and associating each timing sample with the corresponding experimental input. In this work, the script is extended to support reconstruction of intermediate values and to align these values with the measured execution times. It processes raw timing data, organizes it into structured datasets, and prepares them for statistical evaluation.

The analysis script performs statistical testing on the prepared datasets. It groups measurements according to selected feature classes and applies statistical tests to determine whether execution time depends on the chosen features. The implemented tests are described in section 1.5.

2 Timing side-channel analysis of ML-DSA

This chapter describes the `mldsa-toolkit`, which was developed to support timing side-channel analysis of ML-DSA implementations. The toolkit is inspired by Marvin and Minerva and follows similar structure of preparing inputs, collecting timing measurements, and evaluating the measured data statistically.

The toolkit supports two types of experiments. The first type is feature-based analysis, following Minerva. In this setting, input data are generated first, including key pairs and message datasets. The signing operation is then measured, and each recorded runtime is stored together with the produced signature. Selected intermediate values of the signing algorithm are reconstructed afterwards, and numerical features are derived from them. These features are analyzed using the *tlsfuzzer* analysis tool to determine whether execution time correlates with properties of internal signing values.

The second type is class-based analysis, following Marvin. In this setting, measurements are grouped according to predefined classes, which in this thesis correspond to different secret keys. The statistical analysis then evaluates whether execution time differs systematically between these classes, which would indicate key-dependent timing behavior.

To enable reconstruction of the signing procedure, the toolkit works with the deterministic variant of ML-DSA signing. In this setting, the value *rnd* is fixed to $[\mathbf{0}] \times \mathbf{32}$ bytes. As a result, the randomness used during signing becomes reproducible for a given message and secret key. This makes it possible to reconstruct selected intermediate values associated with the accepted signing attempt from the message, secret key, and produced signature.

The code developed for this thesis is available in a GitHub repository [31]. The repository contains the measurement harnesses for the evaluated implementations, scripts for dataset generation and scheduling, reconstruction and feature extraction code, and helper tools used to automate the experiments.

The remainder of this chapter is organized as follows. Section 2.1 describes how timing measurements of the signing operation are collected. Section 2.2 explains how keys and message datasets used in the experiments are generated. Section 2.3 presents the reconstruction of intermediate values and the extraction of numerical features. Finally, Section 2.4 describes the statistical procedures used to evaluate the collected data.

2.1 Measurement methodology

This section describes how timing measurements of the ML-DSA signing operation are collected. The goal is to obtain precise runtime values for a large number of signatures, which are later used to evaluate whether execution time correlates with reconstructed internal values of the algorithm.

2.1.1 Timing harness

To collect timing measurements, dedicated timing harnesses were developed for the evaluated ML-DSA implementations. Each harness repeatedly invokes the signing operation and records its execution time. For every processed message, the harness stores both the measured runtime and the produced signature. During reconstruction, this ordering is used to match each timing sample with intermediate values derived from the same signature.

Separate harnesses were developed for *OpenSSL*, *dilithium-py*, *Go* and *Nettle*. Since these implementations are written in different programming languages, each harness uses the timing mechanism available in that environment.

The harnesses process messages and secret keys from input files using fixed-size records. In experiments with a single secret key, the key is loaded once before the measurement loop and then used for all messages in the input file. In experiments with multiple secret keys, each iteration reads one secret key and one message, performs one deterministic signing operation, and writes one timing value and one signature. This preserves sample alignment, where the i-th timing value corresponds to the i-th signature, the i-th message, and, in the multi-key setting, the i-th secret key. The harnesses check that complete records are read from the input files, since truncated keys or messages would break the alignment required by reconstruction.

The *OpenSSL* and *Nettle* harnesses are implemented in C and measure execution time using architecture-specific clock cycle counters. On `x86_64` systems this is implemented using the `RDTSC` and `RDTSCP` instructions together with a `CPUID` serialization barrier. This provides cycle-level timing measurements with low overhead. The timer implementation also supports additional architectures such as `aarch64`, `s390x`, and `ppc64`, where equivalent hardware counters are used.

The measurement structure used in the *OpenSSL* harness is illustrated in Listing 2.1. This listing shows the *OpenSSL* signing call `EVP_PKEY_sign()`, which is surrounded by calls to the timing functions. The cycle counter is read immediately before invoking the signing function and again immediately after the call returns. The difference between these two values represents the number of clock cycles spent in the measured signing operation.

Listing 2.1: Timing measurement structure used in *OpenSSL*.

```
time_before = get_time_before();
r_ret = EVP_PKEY_sign(ctx, sig, &sig_len, msg, msg_len);
time_after = get_time_after();
```

The *Nettle* harness follows the same measurement structure, but uses the signing interface provided by the *Nettle* implementation.

The functions `get_time_before()` and `get_time_after()` read the clock cycle counter using architecture-specific instructions. For clarity, Listing 2.2 shows a simplified version of the implementation used on `x86_64` systems.

Listing 2.2: Simplified cycle counter implementation for x86_64.

```
uint64_t get_time_before() {
  uint32_t time_before_high = 0, time_before_low = 0;
  asm volatile (
       "CPUID\n\t"
       "RDTSC\n\t"
       "mov %%edx, %0\n\t"
       "mov %%eax, %1\n\t" : "=r" (time_before_high),
       "=r" (time_before_low)::
       "%rax", "%rbx", "%rcx", "%rdx");
  return (uint64_t)time_before_high<<32| time_before_low;
}
uint64_t get_time_after() {
  uint32_t time_after_high = 0, time_after_low = 0;
  asm volatile (
       "RDTSCP\n\t"
       "mov %%edx, %0\n\t"
       "mov %%eax, %1\n\t"
       "CPUID\n\t" : "=r" (time_after_high),
       "=r" (time_after_low)::
       "%rax", "%rbx", "%rcx", "%rdx");
  return (uint64_t)time_after_high<<32 | time_after_low;
}
```

The Python harness used for the *dilithium-py* implementation relies on the `time.monotonic_ns()` function provided by the Python standard library. This function returns a monotonic clock value with nanosecond resolution. The signing operation is measured as shown in Listing 2.3.

Listing 2.3: Timing measurement used in *dilithium-py* harness.

```
time_before = time.monotonic_ns()
sig = scheme.sign(sk, data, deterministic=True)
time_after = time.monotonic_ns()
```

The *Go* harness uses the `time.Now()` function to obtain a timestamp before the signing operation. After the signing routine completes, the elapsed time is computed using `time.Since(start)` and converted to nanoseconds. The measurement structure used in the *Go* harness is shown in Listing 2.4.

Listing 2.4: Timing measurement used in the *Go* harness.

```
start := time.Now()
sig, err = SignDeterministic(privKeyPtr, msgBuf, "")
diff := time.Since(start).Nanoseconds()
```

Several variants of the timing harness were implemented to evaluate different parts of the signing procedure. The variants cover full signing, signing without the final signature encoding step, and key comparison experiments across multiple secret keys.

The harness source files are stored in the `mldsa-toolkit` repository [31]. The *OpenSSL* harnesses are located in `openssl/`, the *dilithium-py* harnesses in `dilithium-py/`, the *Go* harnesses in `golang/`, and the *Nettle* harnesses in `nettle/`. The same file naming convention is used across implementations, as summarized in Table 2.1.

File name	Measurement boundary	Analysis type
`timing.*`	Full signing	Feature-based
`timing_no_encode.*`	Without signature encoding	Feature-based
`key_timing.*`	Full signing	Class-based
`key_timing_no_encode.*`	Without signature encoding	Class-based

Table 2.1: Naming convention for timing harnesses in the `mldsa-toolkit` repository.

Full-signing harness

In the full signing harness, the timer starts immediately before the signing function is invoked and stops immediately after the function returns. The measured interval therefore includes both the internal signing computations and the final encoding of the signature, as illustrated in Figure 2.1.

This variant measures the signing operation as exposed by the evaluated implementation. It does not separate individual steps of the algorithm, but instead captures the full runtime observed by a caller of the signing function. This makes it useful as a baseline experiment, because any timing dependence visible in this setting would also be present in the complete signing interface.

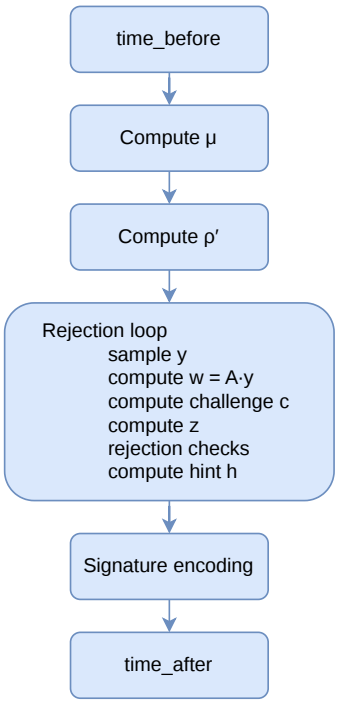


Fig. 2.1: Measurement boundary for the full-signing harness.

Signing without signature encoding

The second variant excludes the final signature encoding step. In this configuration, the timer starts before the signing function is invoked and stops once the signature components have been computed, but before they are encoded into the final byte string, as illustrated in Figure 2.2. The purpose of this variant is to reduce timing variation introduced by serialization and focus the measurement on the internal computations of the signing algorithm.

For the *dilithium-py* implementation, this boundary is implemented by adding timestamps directly to the implementation code. The start timestamp is recorded at the beginning of `sign()`, and the stop timestamp is recorded in `_sign_internal()` immediately before `_pack_sig()` is called. The function then continues normally and returns the encoded signature, so the signature can still be written to the output file.

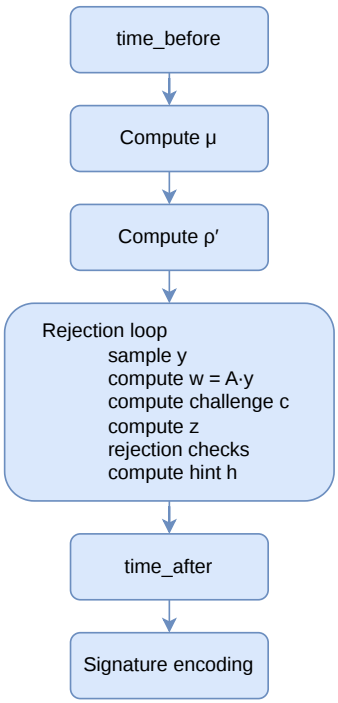


Fig. 2.2: Measurement boundary excluding signature encoding.

The outputs use the same format as the full signing harness. The same measurement boundary is applied to the other implementations by placing the second timestamp before the final signature encoding step.

Key comparison harness

The final harness variant is used to collect timing measurements for class-based key comparison, where each key defines one class. The measurements are later analyzed to dtermine whether signing time differs across keys, which would suggest potential key-dependent behavior.

The harness processes two input files. One file contains the sequence of secret keys, and the other contains the messages used for signing. For each measurement, the harness reads one key-message pair and performs one deterministic signing operation.

To avoid bias caused by a fixed key order, the input sequence is randomized. The experiment is organized into rounds, where each key is used exactly once per round. Within each round, the order of keys is randomly shuffled. As a result, each key appears at different positions in the measurement sequence, which reduces the influence of systematics effects such as cache state or operating-system scheduling.

2.2 Data preparation

Before timing measurements can be performed, the input data must be prepared. This includes generating secret keys, creating message datasets, and defining the order in which signing operations are executed during measurement. The point of this preparation step is to ensure that timing samples can later be matched with the data needed for reconstruction and statistical analysis.

The data preparation scripts are stored in the `scripts/` directory of the `mldsa-toolkit` repository [31]. Table 2.2 summarizes the scripts used in this work.

Script	Purpose
`generate_data.py`	Generates random messages and one key pair for each ML-DSA parameter set.
`generate_keys.py`	Generates multiple key pairs for a selected parameter set.
`generate_schedule.py`	Generates randomized key schedules for random-message key-comparison experiments.
`generate_schedule_keys.py`	Generates aligned key and message schedules for crafted per-key datasets.
`generate_crafted_dataset.py`	Generates crafted messages for each key with controlled rejection behavior.

Table 2.2: Data preparation scripts used in the `mldsa-toolkit` repository.

2.2.1 Key generation

The harnesses support keys for all three ML-DSA parameter sets, namely ML-DSA-44, ML-DSA-65, and ML-DSA-87. Keys are generated with the *OpenSSL* command-line interface and stored in PEM format, since this format is supported by most of the evaluated implementations. The exception is the *Go* implementation, which uses ML-DSA private keys derived from 32-byte seeds. For *Go* harness, the generated keys are therefore converted to the seed format.

Two independent scripts are used for key generation. The first one (`generate_data.py`) prepares the default directory structure for single-key experiments. For each selected parameter set, it creates `sk.pem` and `pk.pem`, which are then used when many messages are signed under one fixed secret key.

The second script (`generate_keys.py`) prepares sets of keys for key-comparison experiments. It creates a selected number of independent key pairs for one ML-DSA parameter set and stores them in indexed directories such as `key_00`, `key_01`, and so on. It also writes a `legend.csv` file that maps each numeric key identifier to the directory containing the key pair. This mapping is later used when measurements are grouped by key during class-based analysis.

Listing 2.5 shows the *OpenSSL* commands used by the scripts to create a secret key and derive the public key.

Listing 2.5: Key generation using *OpenSSL*.

```
subprocess.run(
    ["openssl", "genpkey", "-algorithm", SCHEMES[name],
    "-out", sk_path, "-outform", "PEM"],
    check=True
)
subprocess.run(
    ["openssl", "pkey", "-in", sk_path, "-pubout",
    "-out", pk_path, "-outform", "PEM"],
    check=True
)
```

The first command generates a secret key for the selected ML-DSA parameter set, and the second command derives the public key from it.

2.2.2 Message generation

The experiments use large datasets of input messages for the signing procedure. Messages are generated as random byte strings using the operating system randomness source. Each message has a fixed length of 32 bytes and is stored in a

contiguous binary file. During measurement, the harness reads one message at a time and processes each message once. This gives each timing sample a distinct input message.

The dataset size depends on the experiment. Most measurements use hundreds of thousands to several million messages. In our experiments, approximately 2.5 million samples per parameter set were sufficient to obtain stable results with the applied statistical tests. Smaller datasets of around one million messages were usually sufficient for ML-DSA-65 and ML-DSA-87, while ML-DSA-44 required larger datasets to achieve comparable stability.

2.2.3 Measurement scheduling

During measurement, the prepared datasets are consumed in the order defined by each experiment. For single-key experiments, messages are processed sequentially and do not require scheduling. This setting is used when timing behavior is evaluated over many different messages under one fixed secret key.

Key-comparison experiments require an explicit schedule. In these experiments, each secret key represents one class, and the goal is to test whether signing time differs between these classes. The schedule follows the round-based structure described in Section 2.1. Each round contains one signing operation for every key, and the key order is randomly shuffled within the round.

The schedule is generated before the measurement run. For ordinary key-comparison experiments, the scheduling script (`generate_schedule.py`) reads the generated key directories, extracts the secret key material, and writes it to `schedule.bin` in the order in which the harness should use the keys. Depending on the implementation, this file contains either expanded secret-key bytes obtained from `sk.pem` or 32-byte seeds used by the *Go* harness. The script also writes `log.csv`, which records the randomized key order in each round. This log is later used to assign each timing sample to the correct key class.

A second scheduling script (`generate_schedule_keys.py`) is used for experiments where messages are selected separately for each key, such as clean or early-rejection datasets. In this case, the script reads the selected messages from per-key CSV files and writes the messages and key schedule to `messages.bin` and `schedule.bin` in the same randomized order. This preserves the correspondence between scheduled keys, selected messages, produced signatures, and timing values without the need to repeat the classification step during measurement.

2.2.4 Special dataset generation

Besides random message datasets, specialized datasets are constructed using the `generate_crafted_dataset.py` script to control the rejection behavior of the signing procedure. The main purpose is to obtain measurements in which signing follows the same rejection pattern, especially cases where signing succeeds on the first attempt.

The rejection behavior of ML-DSA is described in FIPS 204. Table 2.3 summarizes the expected distribution of the number of rejected attempts before a successful signature for the three ML-DSA parameter sets, based on the values reported in the standard. The column "0" corresponds to signatures that succeed on the first attempt.

Scheme	Expected	0	1	2	3	4	5	> 5
ML-DSA-44	3.25	0.23	0.18	0.14	0.11	0.08	0.06	0.20
ML-DSA-65	4.10	0.19	0.16	0.12	0.10	0.08	0.07	0.28
ML-DSA-87	3.85	0.25	0.19	0.14	0.11	0.08	0.06	0.17

Table 2.3: Expected distribution of the number of rejected attempts before successful signing in ML-DSA [4].

A significant fraction of executions completes without any rejection. According to Table 2.3, this fraction is approximately 19–25%, depending on the parameter set. These executions form the basis of the *clean datasets* used in this work.

To construct the dataset, candidate messages are tested using an instrumented signing implementation. The instrumentation records the number of signing attempts and the rejection checks that cause failed attempts, especially the checks involving z and r_0. Messages are then classified according to their rejection behavior. For *clean datasets*, only messages for which signing succeeds on the first attempt are selected and stored.

The overall generation procedure is illustrated in Figure 2.3. Random messages are first passed to the instrumented signing implementation. The collected rejection statistics are then used to classify each message. Messages with the required behavior are written to the selected dataset.

The datasets are used as inputs to the timing harness during measurement. The same idea is used for both single-key experiments and key-comparison experiments. In the multi-key setting, candidate messages are classified separately for each key, because rejection behavior depends on both the message and the secret key.

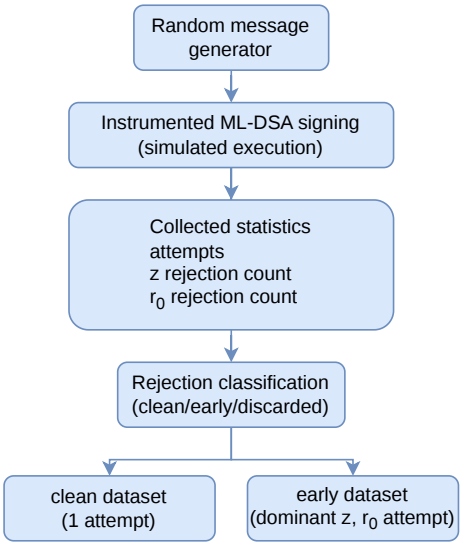


Fig. 2.3: Generation of clean and early-rejection datasets.

The same procedure can also construct datasets with specific early-rejection patterns. These datasets are useful for isolating parts of the rejection process, but they are not used in the experiments of this work. The main analysis focuses on clean executions, which remove timing variability caused by repeated signing attempts.

2.3 Reconstruction of intermediate values

To analyze timing side-channel leakage in the ML-DSA signing procedure, measured execution times must me related to properties of intermediate values used during signing. Since these values are not exposed by the external signing interface, they are reconstructed offline from the data recorded during measurement. This section describes this reconstruction step and the extraction of numerical features used for statistical analysis.

2.3.1 Reconstruction approach

The reconstruction is performed offline using the message, secret key, and signature associated with each measurement. The corresponding timing value is kept together with the extracted features for later statistical analysis. The implementation is part of the `extract.py` script from the *tlsfuzzer* repository. For ML-DSA, the script

uses a modified version of *dilithium-py*, which provides the polynomial and NTT operations needed to reproduce selected parts of the signing computation.

Before processing signatures, the reconstruction code loads the ML-DSA secret key and identifies the parameter set from the key size. The secret key is then unpacked into its components, including ρ, K, tr, s_1, s_2, and t_0. Values that are reused across signatures are precomputed once, these are s_1, s_2, and t_0 in the NTT domain and the public matrix A expanded from ρ. This preparation is shown in Listing 2.6.

Listing 2.6: Preparation of the ML-DSA reconstruction context in `extract.py`.

```
rho, k, tr, s1, s2, t0 = scheme._unpack_sk(sk)
ctx = {
    "scheme": scheme,
    "sk": sk,
    "rho": rho,
    "k": k,
    "tr": tr,
    "s1": s1,
    "s2": s2,
    "s1_hat": s1.to_ntt(),
    "s2_hat": s2.to_ntt(),
    "t0_hat": t0.to_ntt(),
    "A_hat": scheme._expand_matrix_from_seed(rho),
    "q": s1.parent.ring.q,
}
```

For each signature, the reconstruction code reads the message and timing value from the input files. The signature is unpacked to obtain $\tilde{c}$, z, and h. The message representative μ is recomputed from tr and the formatted message. Since the measurements use deterministic signing with $rnd = [0] \times 32$, the value ρ' can also be recomputed from K, rnd, and μ.

The reconstruction does not replay rejection sampling or recover rejected attempts. It only reconstructs values associated with the accepted signing attempt represented by the final signature. After recovering the challenge polynomial c from $\tilde{c}$, the value $c \cdot s_1$ is recomputed and the masking vector is obtained from the relation $y = z - c \cdot s_1$. The reconstructed y is then used to recompute further values, including $w = A \cdot y$, $c \cdot s_2$, and the low bits w_0. Listing 2.7 shows the reconstruction of $c \cdot s_1$, y, and w.

Listing 2.7: Reconstruction of selected ML-DSA intermediate values.

```
c_s1_hat = s1_hat.scale(c_hat)
c_s1 = c_s1_hat.from_ntt()

y = z - c_s1
y_hat = y.to_ntt()

w_hat = A_hat @ y_hat
w = w_hat.from_ntt()
```

This gives a partial reconstruction of the selected intermediate values from the successful signing attempt. The purpose is not to recompute the original execution trace, but to obtain only the necessary values from which numerical features can be derived, associated with the execution time, and used during the statistical analysis.

2.3.2 Selected intermediate values

The ML-DSA signing algorithm performs computations over secret key components, the challenge polynomial, and the masking vector. Values produced by these computations are relevant for timing analysis because their coefficient representation may influence the execution time of arithmetic operations.

The values analyzed in this work are selected based on prior study of side-channel leakage in Dilithium [32]. This study identifies which intermediate values are sensitive in the context of leakage. Since ML-DSA follows the same algorithmic structure as Dilithium, the same types of intermediate values are considered here.

Figure 2.4 is adapted from this prior work and shows a simplified dependency view of the signing computation. The diagram omits some details, such as byte encodings, norm checks, and the fixed deterministic value rnd, but captures the dependencies between the values used in the reconstruction. The red nodes mark secret or secret-dependent values whose leakage would be security-relevant, while the blue nodes show public, or supporting values involved in the signing computation.

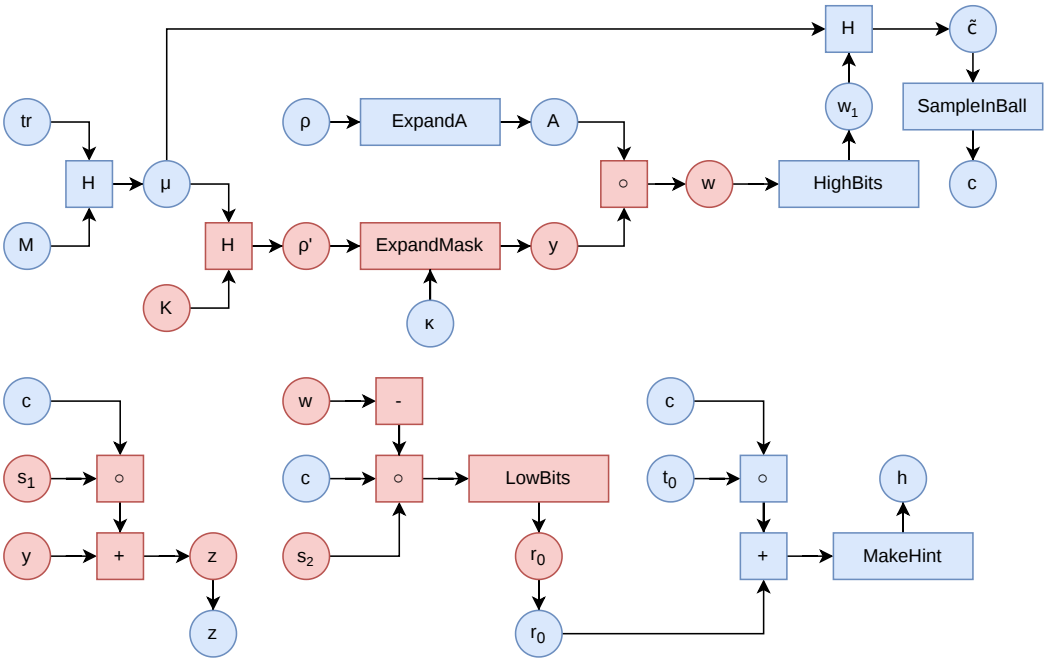


Fig. 2.4: Simplified dependency diagram of ML-DSA intermediate values.

In this work the reconstruction focuses mainly on values computed from the masking vector, the public matrix multiplication, and products involving the challenge polynomial and secret key components.

The reconstruction focuses on the following values and derived representations.

- ρ', the seed used to expand the masking vector y
- y, the masking vector sampled during signing
- y_{NTT}, the NTT representation of y
- w, the vector produced by the matrix-vector multiplication
- w_{NTT}, the NTT representation of w
- $c \cdot s_1$, the product of the challenge polynomial and the secret key vector s_1
- $(c \cdot s_1)_{\mathrm{NTT}}$, the NTT representation of $c \cdot s_1$
- $c \cdot s_2$, the product of the challenge polynomial and the secret key vector s_2
- $(c \cdot s_2)_{\mathrm{NTT}}$, the NTT representation of $c \cdot s_2$
- w_0, the low bits of w obtained during decomposition

These values appear in polynomial multiplication, NTT transformations, and decomposition steps. In these operations, execution time may depend on coefficient properties such as magnitude or representation, for example through modular reduction, conditional adjustments, or arithmetic instructions operating on coefficient values.

Only values from the successful signing attempt are reconstructed. Intermediate values from rejected attempts are not considered, because the final signature only

exposes the accepted attempt. Reconstructing rejected attempts would require additional instrumentation of the signing implementation and is outside the scope of this work.

Some values are intentionally excluded from feature extraction. In particular, z is not used as an analyzed feature because it is directly included in the final signature. The value r_0 is also not used as a separate target feature. Instead, the analysis additionally focuses on w_0, which is not originally computed in ML-DSA.

2.3.3 Feature extraction

The reconstructed intermediate values are not used directly in the statistical analysis. Instead, they are first mapped to numerical features that describe properties of their coefficients and can be compared with the measured execution time.

The main features are the Hamming weight and the sum of coefficient bit-lengths. These features capture simple size and bit-level properties of the reconstructed values. Additional count-based features include the number of zero coefficients and the number of coefficients whose stored value differs from its centered representative modulo q. The complete set of extracted features is summarized in Table 2.4.

Some features are related to the decomposition step of ML-DSA. Coefficients are classified with respect to the low-bit decomposition threshold. This gives features counting coefficients within the low-bit range and coefficients above the selected cutoff.

The extracted features are computed from the reconstructed intermediate values produced by the extraction code. Hamming weight and bit-size features are computed directly from this representation. For selected count-based features, coefficients are additionally interpreted modulo q and mapped to the centered interval, so that the counts are evaluated consistently across equivalent coefficient representations.

The same type of feature is computed both in the coefficient domain and, where applicable, in the NTT domain. This allows the analysis to include values before and after the NTT transformation.

Table 2.4: Extracted features from reconstructed intermediate values.

Intermediate value	Feature	Description
y	hw-y	Hamming weight of coefficients
	bit-size-y	Sum of bit-lengths of coefficients
	y-n-zero	Number of zero coefficients
	y-n-need-reduction	Number of coefficients requiring centered reduction
	y-n-fully-low-bit	Coefficients within low-bit range
	y-n-above-low-bit-cutoff	Coefficients above decomposition threshold
y_{NTT}	ntt-hw-y	Hamming weight in NTT domain
	ntt-bit-size-y	Bit-size in NTT domain
w	hw-w	Hamming weight of coefficients
	bit-size-w	Sum of bit-lengths of coefficients
	w-n-zero	Number of zero coefficients
	w-n-need-reduction	Number of coefficients requiring centered reduction
	w-n-fully-low-bit	Coefficients within low-bit range
	w-n-above-low-bit-cutoff	Coefficients above decomposition threshold
w_{NTT}	ntt-hw-w	Hamming weight in NTT domain
	ntt-bit-size-w	Bit-size in NTT domain
$c \cdot s_1$	hw-c-s1	Hamming weight
	bit-size-c-s1	Bit-size
	c-s1-n-zero	Number of zero coefficients
	c-s1-n-need-reduction	Number of coefficients requiring centered reduction
$(c \cdot s_1)_{\text{NTT}}$	ntt-hw-c-s1	Hamming weight in NTT domain
	ntt-bit-size-c-s1	Bit-size in NTT domain
$c \cdot s_2$	hw-c-s2	Hamming weight
	bit-size-c-s2	Bit-size
	c-s2-n-zero	Number of zero coefficients
	c-s2-n-need-reduction	Number of coefficients requiring centered reduction
$(c \cdot s_2)_{\text{NTT}}$	ntt-hw-c-s2	Hamming weight in NTT domain
	ntt-bit-size-c-s2	Bit-size in NTT domain
w_0	hw-w0	Hamming weight
	bit-size-w0	Bit-size
ρ'	hw-rho-prime	Hamming weight of bytes
	bit-size-rho-prime	Bit-size of bytes
	rho-prime-n-zero	Number of zero bytes

Listing 2.8 shows an example of feature extraction for the reconstructed value y. The code iterates over all coefficients of the reconstructed vector and computes the total Hamming weight and the sum of coefficient bit-lengths. Similar pattern is used for other reconstructed polynomial vectors as well.

Listing 2.8: Feature extraction for the reconstructed value y.

```
y_hw = 0
y_bits = 0

for c0 in self._iter_vector_coeffs(y):
    y_hw += bit_count(c0)
    y_bits += bit_length(c0)

values["hw-y"] = y_hw
values["bit-size-y"] = y_bits
```

The extracted features are written in the format expected by the *tlsfuzzer* analysis tools. Each feature is stored in a separate `measurements-*.csv` file together with the associated timing values. For implementations that report timing in clock cycles, the extraction script also converts the raw cycle counts to seconds using the processor frequency supplied for the experiment.

The `extract.py` script also supports class-based datasets used in key-comparison experiments. In that mode, no intermediate values are reconstructed. Instead, timing measurements are grouped by the key used during signing and arranged according to the schedule generated for the experiment.

2.4 Statistical analysis

The statistical analysis in this work follows the methodology described in Section 1.5. The same general idea is used for the ML-DSA experiments, but the grouping criterion depends on the type of dataset being analyzed.

In the feature-based setting, timing measurements are grouped according to numerical features reconstructed from intermediate signing values. This follows the Minerva-style analysis, where the goal is to determine whether execution time depends on properties of internal values.

In the class-based setting, timing measurements are grouped according to predefined classes. In this work, these classes correspond to different secret keys. This follows the Marvin-style analysis, where the goal is to determine whether execution time differs systematically between input classes.

Both settings are analyzed using the `analysis.py` script from the *tlsfuzzer* tool. The statistical tests themselves are not implemented as part of this thesis. They are used here to evaluate the datasets produced by the measurement, scheduling, and reconstruction steps described in the previous sections.

2.4.1 Data representation

The analysis operates on blocked datasets that preserve the measurement order. Each block corresponds to a group of measurements collected close together in time, typically one measurement round as described in Section 2.2.3. This allows comparisons to be performed within blocks, which reduces the influence of slow changes in system state, load, or scheduling.

In the feature-based setting, the extracted data is written in the long-format representation expected by the `-Hamming-weight` mode of `analysis.py`. Despite the name of this option, it is used in this work for all reconstructed numerical features. Each record contains a block identifier, a group identifier, and a timing value. The group identifier corresponds to the reconstructed feature value, and the timing value corresponds to the measured execution time. For the j-th measurement in the i-th block, this can be viewed as a pair $(t_{i,j}, x_{i,j})$, where $t_{i,j}$ is the measured execution time and $x_{i,j}$ is the feature value.

In the class-based setting, observations are grouped by discrete classes. Each class corresponds to one secret key. Each block contains one measurement for each key class, and the analysis compares timing values across these classes within the same block.

2.4.2 Analysis procedure

Before the statistical tests are applied, the input dataset is converted into the binary format used by `analysis.py`. For feature-based datasets, this conversion processes the long-format `measurements.csv` file containing records of the form `(block, group, value)`. For class-based datasets, the input is a `timing.csv` file in which columns correspond to classes and rows correspond to measurement rounds.

The first step is a global blocked test. For feature-based datasets, the Skillings–Mack test is used. This test is suitable because feature values may be unevenly distributed across blocks. For class-based datasets, the Friedman test is used instead, since each block contains one measurement for each key class. These tests provide the main indication of whether the timing data depends on the selected grouping criterion.

The script then constructs auxiliary paired datasets from the same measurements. In the class-based setting, pairwise comparisons are made between key classes. In the feature-based setting, selected feature-value groups are compared using two-column datasets derived from the blocked measurements. These datasets are analyzed using the sign test, and the Wilcoxon signed-rank test.

For feature-based datasets, the script also constructs a *slope* dataset. Within each block, observations are ordered by their feature values, and neighboring lower–higher pairs are written to a separate two-column dataset. The resulting paired differences indicate whether execution time tends to increase or decrease as the feature value grows. The slope dataset is analyzed using the same paired tests.

The script also computes bootstrapped confidence intervals from paired differences. These intervals estimate the magnitude of the observed timing effect. In the slope analysis, the estimates can be interpreted as timing change per unit increase of the feature. The confidence intervals are used together with p-values to distinguish stable effects from measurement noise.

The statistical output therefore consists of several complementary parts. The first part is the global test result, either Skillings–Mack or Friedman. The second part contains paired test results between groups. For feature-based datasets, the output also includes slope-analysis results and bootstrapped confidence intervals describing the direction and magnitude of the observed effect. These outputs are used together to decide whether a feature or class shows measurable timing dependence.

A result is considered statistically significant when the p-value falls below the selected threshold. In this work, the threshold is set to $\alpha = 10^{-5}$.

2.4.3 Sensitivity analysis via synthetic side-channels

The sensitivity analysis uses functionality already present in the *tlsfuzzer* analysis tool. It is used to estimate what size of timing effect can be detected under the noise conditions of the collected ML-DSA measurements.

The analysis is performed by injecting artificial timing dependencies into measured feature-based datasets and then running the Skillings–Mack test again. Since the injected dependency has a known magnitude, the resulting p-values indicate whether the analysis can detect timing effects of that size.

Two types of synthetic side-channel are used. The first adds a constant timing offset to all measurements belonging to a selected group. This simulates a case where one feature-value group consistently takes longer to process. The injected offset is varied over fixed magnitudes, such as 10 ns, 1 ns, and 0.1 ns.

The second adds a timing offset proportional to the feature value. For a feature value x, the modified timing includes an additional term proportional to $x - \text{median}(x)$. This models a value-dependent timing effect, such as a small timing increase per additional unit of the analyzed feature. The injected magnitude is varied over fixed values down to 0.01 ns per unit.

For each injected side-channel, the modified dataset is analyzed using the same global Skillings–Mack test as the original feature-based dataset. The resulting p-values are compared across injected magnitudes. This provides an estimate of the detection threshold of the analysis for each dataset.

3 Experimental results and discussion

This chapter reports the results of the timing analysis performed on four ML-DSA implementations: *dilithium-py*, *Go*, *OpenSSL*, and *Nettle*. The experiments cover random and crafted message datasets, full-signing and partial-signing measurements, and both feature-based and class-based analysis. The evaluated versions are listed in Table 3.1.

Table 3.1: Evaluated ML-DSA implementations and their versions.

Implementation	Version
dilithium-py	1.4.0
Go	1.26
OpenSSL	3.6.1
Nettle	development version

All measurements were performed on dedicated machines running *RHEL 9.7*. Several processor models were used during the evaluation, and the relevant hardware platform is reported with the corresponding experiment. To reduce measurement noise, one physical CPU core was isolated from the rest of the system using the `tuned` *cpu-partitioning* profile. The timing harnesses were pinned to the isolated core using explicit CPU affinity. All experiments use the setup and statistical procedures described in Section 2.4.

The experiments were carried out iteratively. We first measured the complete signing operation on random message datasets, but these measurements did not show clear timing dependencies. In parallel, we performed class-based experiments, which provided some additional observations about where timing differences could appear. These observations guided the next step of the evaluation and led us back to feature-based analysis, this time using crafted datasets and narrower measurement boundaries. For clarity, we present the results by experiment type rather than chronologically.

The chapter is organized as follows. Section 3.1 presents full-signing measurements for both random and crafted datasets. Section 3.2 evaluates partial-signing measurements, where the final signature encoding step is excluded from the measured interval. Section 3.3 presents class-based experiments that compare signing times across different secret keys. Finally, Section 3.4 discusses the results and summarizes their implications for the interpretation of the timing leakage in the ML-DSA implementations.

3.1 Full-signing measurements

The first set of experiments focuses on the execution time of the complete ML-DSA signing operation. The measured interval includes message processing, the internal signing computation, rejection sampling when it occurs, and the final signature encoding step.

This is the broadest measurement boundary in this work and corresponds to a black-box view of the signing interface. The measurement is taken around the externally exposed signing operation, without separating individual steps inside the implementation. It is useful as a baseline, but it cannot identify which part of the signing procedure is responsible for an observed timing effect.

For each implementation, two types of input datasets are considered. The first uses randomly generated messages and includes the normal rejection behavior of the signing procedure. The second uses a controlled dataset containing only messages for which signing succeeds on the first attempt.

3.1.1 Random messages

The first full-signing experiment uses randomly generated messages. For each evaluated implementation and parameter set, 10^6 signature operations were measured on an `x86_64` system equipped with an Intel Xeon Silver 4116 CPU. This setting includes the normal rejection behavior of ML-DSA.

Eleven runs were performed in total across the evaluated implementations and parameter sets. No consistent timing dependence was detected in this setting. Table 3.2 shows one representative result for *OpenSSL* with the ML-DSA-65 parameter set.

Table 3.2: Selected timing analysis results for *OpenSSL* ML-DSA-65 with random messages.

Feature	Skillings–Mack p-value	Median Estimate [10^{-9}s/unit]	Median 95% CI [10^{-9} s/unit]
`bit-size-c-s1`	8.47×10^{-1}	2.86	$[-20.05, 27.68]$
`bit-size-c-s2`	4.22×10^{-2}	-8.59	$[-35.32, 19.09]$
`bit-size-y`	9.60×10^{-1}	15.27	$[-10.50, 43.91]$
`hw-c-s1`	9.36×10^{-1}	-18.13	$[-42.00, 6.68]$
`hw-c-s2`	7.86×10^{-1}	-11.46	$[-35.32, 13.36]$
`w-n-need-reduction`	2.94×10^{-2}	-11.46	$[-37.23, 17.18]$
`y-n-need-reduction`	7.16×10^{-1}	-3.82	$[-31.50, 23.87]$

The representative results do not show statistically significant timing dependence. The smallest p-values are obtained for `bit-size-c-s2` and `w-n-need-reduction`, but both remain above the threshold $\alpha = 10^{-5}$.

The median estimates also do not provide evidence of a stable timing effect. For every selected feature, the 95% confidence interval crosses zero. This means that the data do not support a consistent timing increase or decrease as the feature value grows. For example, `y-n-need-reduction` has a median estimate of -3.82×10^{-9} s per unit, but its confidence interval ranges from -31.50 to 23.87 ns per unit. Such a wide interval includes both negative and positive effects, so the feature does not show a reliable timing slope.

The sensitivity analysis for `y-n-need-reduction` is summarized in Table 3.3. The original Skillings–Mack p-value is 7.16×10^{-1}. Adding a constant artificial offset of 10 ns, 1 ns, or 0.1 ns to one selected feature-value group does not substantially change the result. This indicates that this type of isolated group offset is not detected in this random-message setting.

Table 3.3: Sensitivity analysis for `y-n-need-reduction` in the *OpenSSL* ML-DSA-65 random-message experiment.

Injected effect	Magnitude	Skillings–Mack p-value
None	–	7.16×10^{-1}
Constant group offset	10 ns	7.20×10^{-1}
Constant group offset	1 ns	7.17×10^{-1}
Constant group offset	0.1 ns	7.17×10^{-1}
Value-dependent offset	10 ns/unit	1.01×10^{-96}
Value-dependent offset	1 ns/unit	7.64×10^{-1}
Value-dependent offset	0.1 ns/unit	7.40×10^{-1}
Value-dependent offset	0.01 ns/unit	7.18×10^{-1}

A systematic value-dependent offset is easier to detect. When an artificial dependency of 10 ns per unit is added, the Skillings–Mack p-value drops to 1.01×10^{-96}. However, injected effects of 1 ns, 0.1 ns, and 0.01 ns per unit are not detected. This suggests that, for this feature and measurement configuration, the analysis would detect a large systematic timing effect, but not effects around 1 ns per unit or smaller.

This agrees with the confidence interval. Since the interval for `y-n-need-reduction` spans from -31.50 to 23.87 ns per unit, a 1 ns per-unit effect would be hidden inside the uncertainty of the estimate. It would therefore not be reliably distinguishable from measurement variability in this configuration.

The remaining features follow the same pattern. Their p-values are above the significance threshold, their confidence intervals cross zero, and the estimated directions are inconsistent. The additional slope analysis was also reviewed, but it did not change this interpretation. Overall, this representative random-message experiment is treated as a negative result. This does not prove that no timing dependence exists, it only shows that no dependence was detectable at the sensitivity achieved by this measurement setup.

To check whether the *OpenSSL* ML-DSA-65 result is specific to one parameter set, Table 3.4 reports the same selected features for the remaining *OpenSSL* parameter sets.

Table 3.4: Selected timing analysis results for *OpenSSL* ML-DSA-44 and ML-DSA-87 with random messages.

Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
44	`bit-size-c-s1`	7.95×10^{-1}	15.27	$[-0.95, 38.18]$
44	`bit-size-c-s2`	4.07×10^{-1}	0.95	$[-17.18, 16.23]$
44	`bit-size-y`	6.27×10^{-1}	-21.95	$[-39.16, -1.91]$
44	`hw-c-s1`	9.78×10^{-3}	3.82	$[-18.14, 21.96]$
44	`hw-c-s2`	5.22×10^{-1}	2.86	$[-15.27, 23.87]$
44	`w-n-need-reduction`	4.95×10^{-1}	4.77	$[-19.09, 27.68]$
44	`y-n-need-reduction`	1.66×10^{-1}	-9.54	$[-31.50, 11.46]$
87	`bit-size-c-s1`	3.10×10^{-1}	2.86	$[-14.32, 25.77]$
87	`bit-size-c-s2`	1.32×10^{-1}	-5.73	$[-25.77, 13.36]$
87	`bit-size-y`	4.66×10^{-1}	-15.27	$[-33.41, 4.77]$
87	`hw-c-s1`	9.50×10^{-1}	-10.50	$[-27.68, 7.64]$
87	`hw-c-s2`	6.13×10^{-1}	-7.64	$[-26.73, 10.50]$
87	`w-n-need-reduction`	2.81×10^{-1}	-6.68	$[-25.77, 15.27]$
87	`y-n-need-reduction`	4.48×10^{-1}	8.59	$[-10.50, 30.55]$

The results for ML-DSA-44 and ML-DSA-87 are consistent with the ML-DSA-65 example. None of the selected features reaches the significance threshold $\alpha = 10^{-5}$. The smallest p-value is obtained for `hw-c-s1` in ML-DSA-44, but it still remains above the threshold used in this work.

The median estimates also do not provide stable evidence of timing dependence. Most confidence intervals cross zero. The only exception shown here is `bit-size-y` for ML-DSA-44, where the median 95% confidence interval lies below zero. Since this interval is not accompanied by a significant Skillings–Mack result, it is not treated as evidence of timing dependence.

The same pattern was observed in the remaining random-message runs across the evaluated implementations and parameter sets. The random-message whole-signature setting is therefore treated as a negative result, and the next section considers a more controlled dataset.

3.1.2 Crafted dataset

The second full-signing experiment uses crafted message datasets. A fixed secret key is used, and messages are selected such that signing succeeds on the first attempt. The measurements were performed on an `x86_64` system equipped with an Intel Xeon Gold 6330 processor. The dataset contains 2 591 625 messages for ML-DSA-44, 2 700 953 messages for ML-DSA-65, and 2 415 233 messages for ML-DSA-87.

Compared to the random-message experiment, this setting removes variation caused by different numbers of rejection iterations. All measured signatures are accepted on the first attempt, while the measurement boundary still includes the complete signing operation and the final signature encoding step.

In total, eleven runs were performed across the evaluated implementations and parameter sets. In contrast to the random-message setting, several crafted-dataset runs produced statistically significant results. For *Go*, *OpenSSL*, and *Nettle*, the strongest signals were mainly associated with features derived from w_0, especially `hw-w0` and `bit-size-w0`. The *dilithium-py* implementation behaved differently and is discussed separately below.

Table 3.5 summarizes selected w_0-based results for *Go*, *OpenSSL*, and *Nettle*. The table reports the Skillings–Mack p-value, median estimate, and the median 95% confidence interval for `hw-w0` and `bit-size-w0`.

The results show a clear w_0-related signal for *Go* and *OpenSSL*. For both implementations, the selected w_0-based features produce very small Skillings–Mack p-values across all three parameter sets. These results therefore provide strong statistical evidence that execution time depends on the w_0-based feature values in the crafted full-signing setting.

The *OpenSSL* results are the most stable. For all reported parameter sets and both selected w_0-based features, the median estimates are positive and the 95% confidence intervals remain above zero. The estimated effects are typically on the order of 1 to 3 ns per feature unit. This means that increasing the w_0-based feature is consistently associated with longer execution time in these measurements.

The size of the *OpenSSL* effect is also relevant for interpreting the experiment. In the random-message full-signing setting, the sensitivity analysis showed that effects around 1 ns per feature unit were not reliably detected for the analyzed feature. In the crafted-dataset setting, the *OpenSSL* confidence intervals are consistently in

Table 3.5: Summary of timing analysis results for the crafted dataset (full signing).

Impl.	Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
Go	44	`bit-size-w0`	0.0	2.00	$[1.0, 3.0]$
Go	44	`hw-w0`	5.86×10^{-71}	1.00	$[0.0, 2.0]$
Go	65	`bit-size-w0`	1.85×10^{-42}	2.00	$[0.0, 3.0]$
Go	65	`hw-w0`	2.28×10^{-95}	3.00	$[1.0, 4.0]$
Go	87	`bit-size-w0`	2.70×10^{-31}	1.00	$[-1.0, 3.0]$
Go	87	`hw-w0`	1.97×10^{-56}	2.00	$[0.0, 4.0]$
OpenSSL	44	`bit-size-w0`	0.0	2.86	$[1.91, 2.86]$
OpenSSL	44	`hw-w0`	1.10×10^{-269}	0.95	$[0.95, 1.91]$
OpenSSL	65	`bit-size-w0`	6.65×10^{-191}	0.95	$[0.95, 2.86]$
OpenSSL	65	`hw-w0`	6.65×10^{-191}	1.90	$[0.95, 2.86]$
OpenSSL	87	`bit-size-w0`	2.91×10^{-76}	2.86	$[0.95, 3.82]$
OpenSSL	87	`hw-w0`	7.42×10^{-143}	1.91	$[0.95, 3.82]$
Nettle	65	`bit-size-w0`	6.76×10^{-19}	0.95	$[-0.95, 2.86]$
Nettle	65	`hw-w0`	9.06×10^{-35}	2.86	$[0.95, 4.77]$
Nettle	87	`bit-size-w0`	7.14×10^{-3}	3.82	$[0.0, 7.64]$
Nettle	87	`hw-w0`	6.21×10^{-13}	1.91	$[-0.95, 4.77]$

approximately the 1–3 ns range and remain above zero. This indicates that controlling the rejection behavior improved the sensitivity of the measurement enough to expose timing effects that were not visible in the random-message setting.

The concentration of the signal in w_0-based features is notable, but it should be interpreted carefully. The value w_0 is not directly exposed by the signing interface and is reconstructed here as an analysis target derived from the intermediate value w. The observed dependence therefore does not by itself identify the operation responsible for the timing effect. It may be related to how the implementation processes values derived from w, to decomposition-related operations, or to the final signature encoding step. This question is examined further in the partial-signing measurements, where the final encoding step is excluded in order to narrow the possible source of the signal.

The *Go* results also show very small p-values, but the confidence intervals are less stable. Some intervals are positive, while others touch or cross zero. This means that the Skillings–Mack test detects timing dependence between execution time and the feature values, but the median estimate does not always give a stable

direction of the effect. The *Go* signal is therefore statistically strong, but closer to the measurement noise floor than the *OpenSSL* signal.

The *Nettle* results are weaker. For ML-DSA-65, `hw-w0` reaches the significance threshold and has a positive confidence interval, while `bit-size-w0` is significant but its confidence interval crosses zero. For ML-DSA-87, `hw-w0` remains significant according to the Skillings–Mack test, but the confidence interval again crosses zero. The `bit-size-w0` result for ML-DSA-87 does not reach the threshold $\alpha = 10^{-5}$. Thus, *Nettle* shows evidence of w_0-related timing dependence in some cases, but the effect is weaker and less stable than for *Go* and *OpenSSL*.

As a concrete example, Figures 3.1 and 3.2 show the confidence-interval plots for *Go* ML-DSA-44. For `bit-size-w0`, the median estimate is 2.0×10^{-9} s per unit, with a 95% confidence interval from 1.0 to 3.0 ns per unit. For `hw-w0`, the median estimate is 1.0×10^{-9} s per unit, with a 95% confidence interval from 0.0 to 2.0 ns per unit. These plots illustrate that the crafted full-signing measurement detects a w_0-related timing effect in *Go*, although the confidence interval for `hw-w0` touches zero and is therefore less stable than the `bit-size-w0` result.

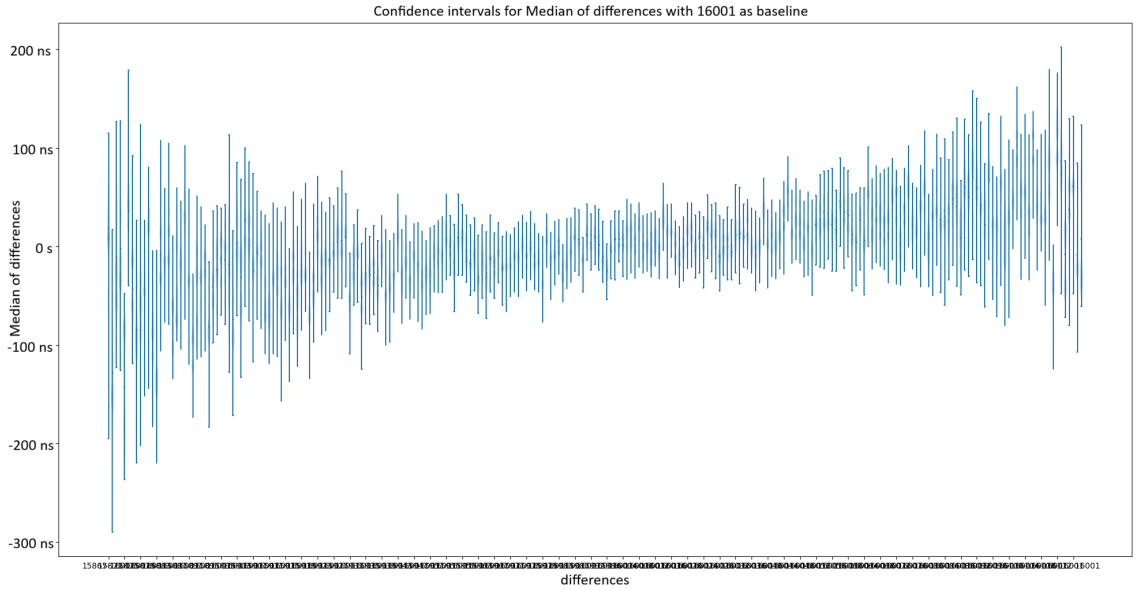


Fig. 3.1: Confidence intervals for median of differences between `bit-size-w0` classes for *Go* ML-DSA-44, measured with the crafted dataset using the full-signing harness.

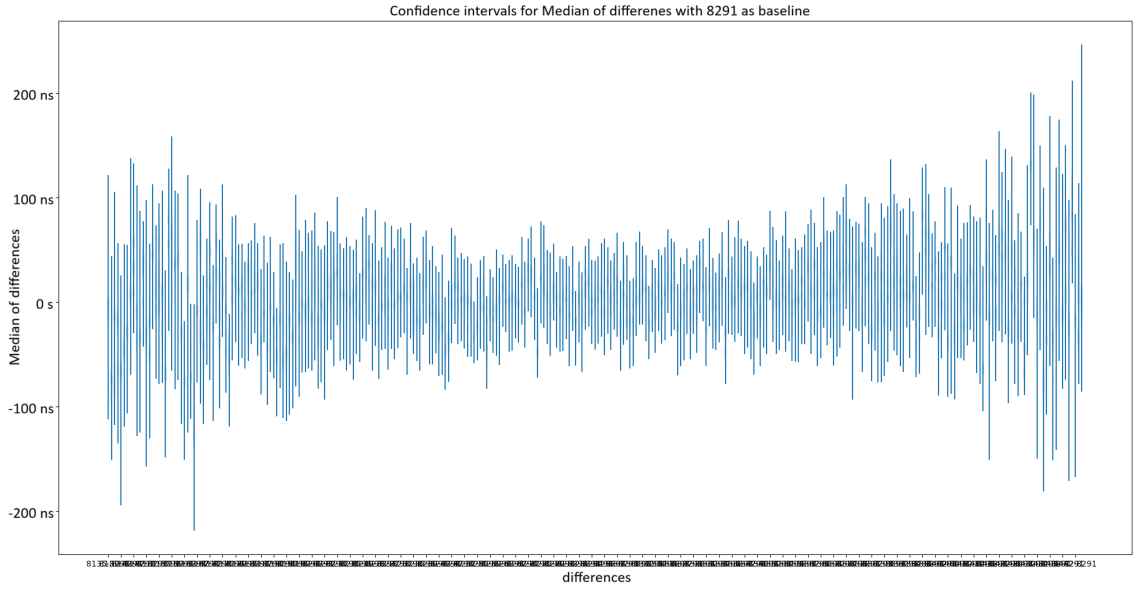


Fig. 3.2: Confidence intervals for median of differences between `hw-w0` classes for *Go* ML-DSA-44, measured with the crafted dataset using the full-signing harness.

The *dilithium-py* implementation shows a different pattern from *Go*, *OpenSSL*, and *Nettle*. In this implementation, the significant results are not concentrated in w_0-based features. Instead, timing dependence is detected for features derived from several other reconstructed intermediate values, including $c{\cdot}s_1$, $c{\cdot}s_2$, and the masking vector y. Table 3.6 shows representative results for the ML-DSA-44 parameter set.

Table 3.6: Selected timing analysis results for *dilithium-py* ML-DSA-44 with the crafted dataset.

Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
`bit-size-c-s1`	0.0	96	$[-92.0, 251.0]$
`bit-size-y`	3.10×10^{-36}	287	$[110.0, 436.0]$
`c-s1-n-need-reduction`	0.0	-234	$[-422.0, -64.0]$
`c-s2-n-need-reduction`	2.75×10^{-13}	-170	$[-376.0, 6.5]$
`hw-c-s1`	2.94×10^{-298}	-217	$[-386.5, -50.0]$
`hw-y`	2.37×10^{-19}	165.5	$[7.0, 339.0]$
`w-n-above-low-bit-cutoff`	2.97×10^{-11}	229	$[-17.0, 533.0]$
`w-n-fully-low-bit`	6.64×10^{-10}	77	$[-169.0, 344.0]$
`y-n-above-low-bit-cutoff`	7.51×10^{-68}	161	$[-23.0, 368.0]$
`y-n-fully-low-bit`	2.42×10^{-69}	0	$[-172.0, 200.0]$
`y-n-need-reduction`	1.25×10^{-88}	236	$[24.0, 425.0]$

A striking difference is the magnitude of the estimated effects. In *Go* and *OpenSSL*, the w_0-based timing effects are typically on the order of 1–3 ns per feature unit. In contrast, several *dilithium-py* features have median estimates in the range of hundreds of nanoseconds per unit, such as `bit-size-y`, `hw-c-s1`, and `y-n-need-reduction`. This much larger scale helps explain why significant p-values appear across many different feature families rather than being concentrated in one narrow part of the computation.

The confidence intervals show that the direction and stability of the effect differ between features. Some features have median confidence intervals that do not cross zero, such as `c-s1-n-need-reduction`, `hw-c-s1`, `hw-y`, and `y-n-need-reduction`. These features provide both a very small p-value and a stable estimated timing effect. Other features, such as `bit-size-c-s1`, `w-n-fully-low-bit`, and `y-n-fully-low-bit`, have very small p-values but median confidence intervals that cross zero. For these features, the Skillings–Mack test detects timing dependence, but the median estimate does not give a stable direction.

This behavior is expected for *dilithium-py*. The implementation is written in Python and is not designed as a constant-time cryptographic implementation. Its arithmetic uses high-level Python objects and standard language operations, which do not provide constant-time guarantees. The observed timing dependence should therefore be interpreted as broader timing variability of the high-level implementation rather than as evidence of one isolated algorithmic bottleneck.

The measurements in this subsection still cover the complete signing operation, including signature encoding. Therefore, these results show that the full-signing runtime can depend on features derived from reconstructed intermediate values, but they do not identify which exact operation is responsible for the observed dependence. The next experiment narrows the measurement boundary by excluding the final signature encoding step.

3.2 Partial-signing measurements

The full-signing experiments with the crafted dataset showd statistically significant timing dependence, most visibly for features derived from w_0. However, those measurements included the final signature encoding step. To examine whether the observed signal remains visible before this step, we use the partial-signing harness described in Section 2.1. The harness starts the timer before the signing operation and stops it immediately before the final signature encoding step. The measured interval therefore covers the internal signing computation up to the point where the signature components have been constructed, but excludes their serialization into the final signature byte string.

The same crafted datasets as in the full-signing experiment are used here. For each parameter set, the messages are selected so that signing succeeds on the first attempt. This keeps the rejection behavior fixed across measurements.

The purpose of this experiment is to determine whether the timing dependence observed in th full-signing measurements persists after the final encoding step is removed from the measured interval. The results differ between implementations. For *OpenSSL* and *Nettle*, the w_0-related signal observed in the full-signing setting is no longer detected in the partial-signing measurements. This suggests that the earlier signal was associated with the final signature encoding step rather than with the measured pre-encoding part of signing.

For *Go*, excluding the encoding step reveals a different pattern. The w_0-related signal is no longer present. Instead, on selected processors, statistically significant dependence appears for features derived from $c \cdot s_1$ and $c \cdot s_2$. The *dilithium-py* implementation continues to show timing dependence across a broader set of features.

The following sections discuss the *Go* results, the *OpenSSL* and *Nettle* results, and the *dilithium-py* results separately.

3.2.1 Go

The strongest *Go* results were obtained on the Intel Xeon E5-2630 v3 platform. Table 3.7 summarizes selected results for this platform. For each feature, the table reports the Skillings–Mack p-value, median estimate, and confidence interval of the median timing difference.

The results show low p-values for several features derived from $c \cdot s_1$ and $c \cdot s_2$. The strongest evidence is observed for the reduction-count features, especially `c-s1-n-need-reduction` and `c-s2-n-need-reduction`. For example, in ML-DSA-44, these features produce p-values of 2.11×10^{-69} and 1.05×10^{-91}, respectively. Hamming-weight and bit-size features also produce low p-values, although the strength differs between parameter sets.

The estimated effects are small, typically on the order of a few nanoseconds per feature unit or less. Several confidence intervals touch zero or cross it, so the median estimate is not equally stable for all features. This is especially visible for the reduction-count features. For example, `c-s1-n-need-reduction` in ML-DSA-44 has a reported median estimate of 0 ns per unit and a confidence interval of $[-1.0, 0.0]$ ns per unit, while still producing a very small Skillings–Mack p-value. It indicates that the Skillings–Mack test detects a consistent dependence, while the median effect is very small and close to the reporting resolution of the timing differences.

Table 3.7: Selected timing analysis results for the *Go* implementation in the partial-signing setting on Intel Xeon E5-2630 v3.

Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
44	`bit-size-c-s1`	5.69×10^{-10}	-1	$[-2.0, 0.0]$
44	`bit-size-c-s2`	5.00×10^{-23}	1	$[0.0, 1.0]$
44	`c-s1-n-need-reduction`	2.11×10^{-69}	0	$[-1.0, 0.0]$
44	`c-s2-n-need-reduction`	1.05×10^{-91}	1	$[0.0, 1.0]$
44	`hw-c-s1`	4.72×10^{-16}	-1	$[-1.0, 0.0]$
44	`hw-c-s2`	9.45×10^{-16}	1	$[0.0, 2.0]$
65	`bit-size-c-s2`	1.47×10^{-9}	1	$[0.0, 2.0]$
65	`c-s2-n-need-reduction`	1.34×10^{-68}	1	$[-1.0, 2.0]$
65	`hw-c-s2`	6.40×10^{-15}	1	$[0.0, 2.0]$
87	`bit-size-c-s1`	3.45×10^{-13}	-1	$[-2.0, 0.0]$
87	`bit-size-c-s2`	2.34×10^{-15}	2	$[1.0, 3.0]$
87	`c-s1-n-need-reduction`	4.24×10^{-66}	0	$[-2.0, 1.0]$
87	`c-s2-n-need-reduction`	3.41×10^{-78}	2	$[0.0, 3.0]$
87	`hw-c-s1`	1.31×10^{-10}	-2	$[-3.0, 0.0]$
87	`hw-c-s2`	1.86×10^{-16}	2	$[0.0, 3.0]$

For ML-DSA-44, the selected $c \cdot s_1$-related features tend to have negative median estimates or confidence intervals touching zero from below, while the selected $c \cdot s_2$-related features tend to have positive estimates or intervals touching zero from above. Thus, increasing the reconstructed features of $c \cdot s_1$ is associated with slightly shorter execution time, while increasing the corresponding features of $c \cdot s_2$ is associated with slightly longer execution time.

Figures 3.3 and 3.4 show confidence-interval plots for `c-s1-n-need-reduction` and `c-s2-n-need-reduction` in the ML-DSA-44 parameter set. These two plots illustrate the opposite directions observed in the table.

These results show that, on the Intel Xeon E5-2630 v3 platform, the *Go* partial-signing measurements detect timing dependence related to products involving the challenge polynomial and secret key components. To check whether this behavior also appears on other hardware, we repeated the same experiment on different platforms.

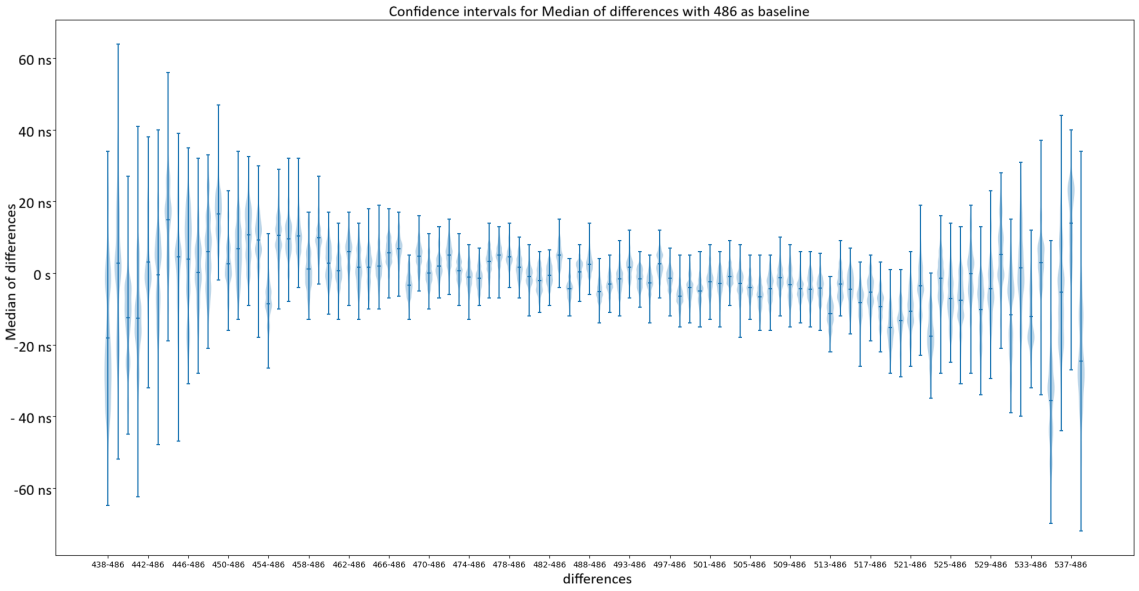


Fig. 3.3: Confidence intervals for median of differences between `c-s1-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness.

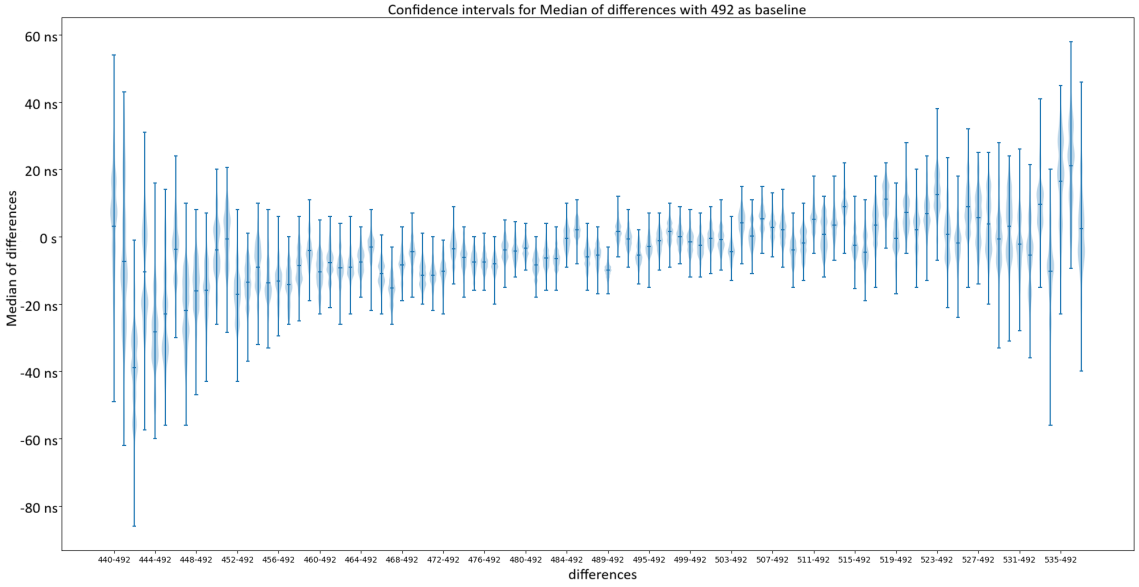


Fig. 3.4: Confidence intervals for median of differences between `c-s2-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness.

The repeated measurements show that this behavior is not uniform across all tested processors. Similar statistically significant dependence for features related to $c \cdot s_1$ and $c \cdot s_2$ was also observed on the Intel Xeon Silver 4216 platform.

In contrast, the same experiment on the Intel Xeon Silver 4116 platform did not show statistically significant dependence. Measurements on the AMD EPYC 7742 platform produced only isolated lower p-values, without a stable pattern across parameter sets or features. This shows that the *Go* result is not determined only by the source-level implementation, but also depends on the hardware platform. Table 3.8 summarizes representative results from the Intel Xeon Silver 4116 and AMD EPYC 7742 platforms, where the *Go* partial-signing experiment did not reproduce the timing dependence observed on Intel Xeon E5-2630 v3.

The values remain above the significance threshold $\alpha = 10^{-5}$. The AMD EPYC 7742 results contain a few smaller p-values, such as 2.4×10^{-4} for `c-s2-n-need-reduction` in ML-DSA-44, but this value does not reach the threshold used in this work and does not appear consistently across parameter sets.

The AMD results also differ in the reported median estimates and confidence intervals. For many ML-DSA-44 and ML-DSA-65 features, the median estimate is reported as 0 and the confidence interval as $[0.0, 0.0]$ ns per unit. This does not indicate a detected timing effect. Rather, it means that, at the reporting precision of the analysis, the median timing differences do not show a measurable deviation from zero.

The smaller p-values in the AMD results are isolated. The lowest one, 2.4×10^{-4} for `c-s2-n-need-reduction` in ML-DSA-44, has a median estimate of 0 and does not repeat across parameter sets. It is therefore not treated as a reproduced signal.

Figures 3.5 and 3.6 show the confidence-interval plots for the AMD EPYC 7742 platform for the same two ML-DSA-44 features that produced strong results on the Intel Xeon E5-2630 v3 platform. A comparison with Figures 3.3 and 3.4 shows the difference in signal quality. On the Intel Xeon E5-2630 v3 platform, the selected features show a visible direction in the confidence intervals. On the AMD EPYC 7742 platform, the intervals are instead centered around zero or collapse to zero at the reported precision.

The AMD results therefore do not reproduce the behavior observed on the Intel Xeon E5-2630 v3 platform. The confidence intervals do not show the same stable separation pattern, and the p-values do not form a consistent feature family across parameter sets. The difference between the tested processors shows that the detected *Go* timing dependence is platform-dependent. The exact cause is not isolated in this work.

Table 3.8: Representative timing analysis results for the *Go* implementation on negative platforms (partial-signing setting).

Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
Intel Xeon Silver 4116				
44	`bit-size-c-s1`	0.051	-1	$[-2.0, 0.0]$
44	`bit-size-c-s2`	0.225	-1	$[-2.0, 1.0]$
44	`c-s1-n-need-reduction`	0.528	0	$[-2.0, 1.0]$
44	`c-s2-n-need-reduction`	0.605	-1	$[-2.0, 1.0]$
44	`hw-c-s1`	0.168	1	$[0.0, 2.0]$
44	`hw-c-s2`	0.692	-1	$[-2.0, 1.0]$
65	`bit-size-c-s2`	0.851	0	$[-1.0, 1.0]$
65	`c-s2-n-need-reduction`	0.360	0	$[-1.0, 2.0]$
65	`hw-c-s2`	0.228	1	$[0.0, 3.0]$
87	`bit-size-c-s1`	0.362	0	$[-1.0, 2.0]$
87	`bit-size-c-s2`	0.855	-1	$[-3.0, 0.0]$
87	`c-s1-n-need-reduction`	0.180	-1	$[-3.0, 1.0]$
87	`c-s2-n-need-reduction`	0.038	-1	$[-2.0, 1.0]$
87	`hw-c-s1`	0.647	0	$[-1.0, 2.0]$
87	`hw-c-s2`	0.594	0	$[-1.0, 2.0]$
AMD EPYC 7742				
44	`bit-size-c-s1`	0.871	0	$[0.0, 0.0]$
44	`bit-size-c-s2`	0.267	0	$[0.0, 0.0]$
44	`c-s1-n-need-reduction`	0.560	0	$[0.0, 0.0]$
44	`c-s2-n-need-reduction`	2.4×10^{-4}	0	$[0.0, 0.0]$
44	`hw-c-s1`	0.937	0	$[0.0, 0.0]$
44	`hw-c-s2`	0.418	0	$[0.0, 0.0]$
65	`bit-size-c-s2`	0.036	0	$[0.0, 0.0]$
65	`c-s2-n-need-reduction`	0.443	0	$[0.0, 0.0]$
65	`hw-c-s2`	0.701	0	$[0.0, 0.0]$
87	`bit-size-c-s1`	0.099	1	$[0.0, 9.0]$
87	`bit-size-c-s2`	0.894	0	$[-1.0, 0.0]$
87	`c-s1-n-need-reduction`	0.493	0	$[0.0, 9.0]$
87	`c-s2-n-need-reduction`	0.858	0	$[0.0, 9.0]$
87	`hw-c-s1`	0.521	0	$[-1.0, 0.0]$
87	`hw-c-s2`	0.382	0	$[-1.0, 0.0]$

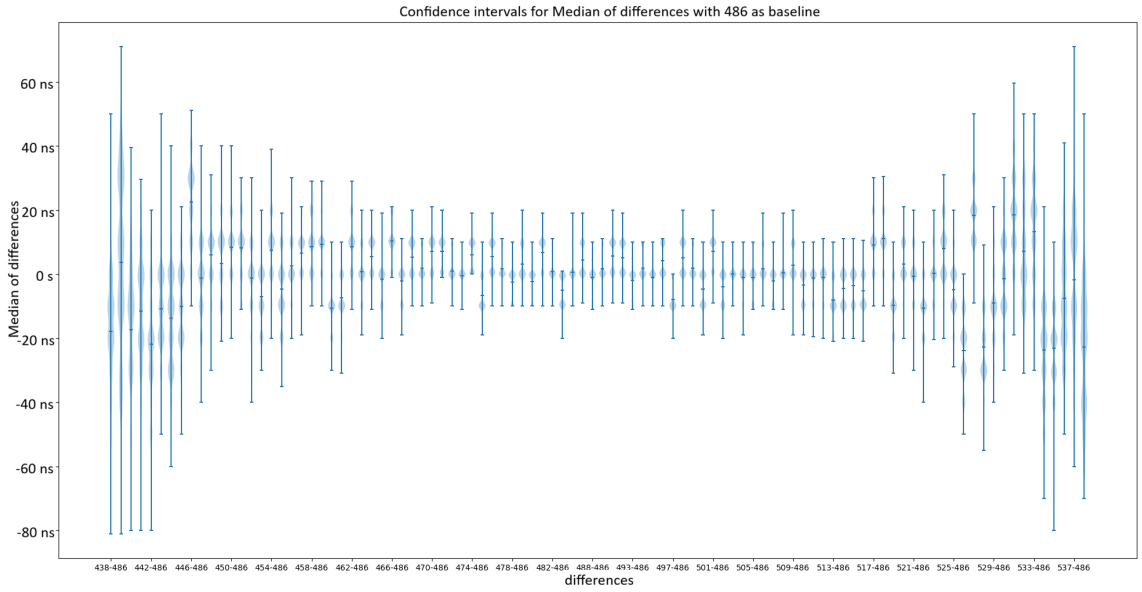


Fig. 3.5: Confidence intervals for median of differences between `c-s1-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness on AMD EPYC 7742.

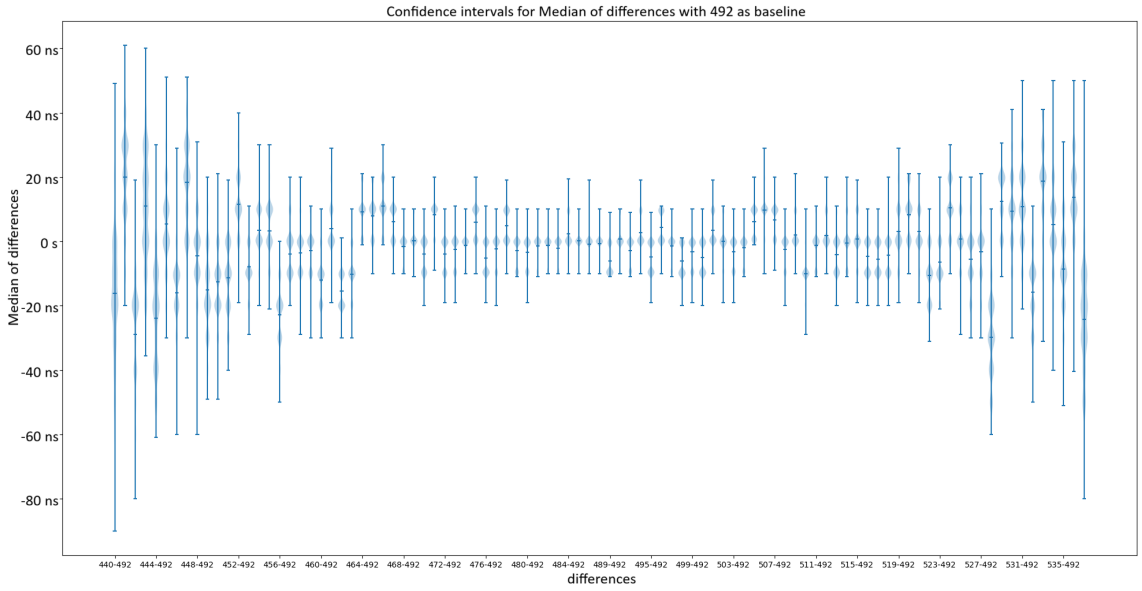


Fig. 3.6: Confidence intervals for median of differences between `c-s2-n-need-reduction` classes for *Go* ML-DSA-44, measured with the crafted dataset using the partial-signing harness on AMD EPYC 7742.

The *Go* partial-signing experiment was also repeated on an Intel Xeon Gold 6438N platform with Intel Data Operand Independent Timing mode (DOITM) [33]. DOITM is a processor feature intended to reduce timing differences caused by the values of data operands. When enabled, supported instructions should execute with timing that does not depend on the particular data values they process, provided the code is already written in a constant-time. Intel documents DOITM only for a specific subset of instructions and also notes that enabling it globally is not recommended because of possible performance impact. The experiment was therefore run in two configurations, with DOIT disabled and with DOIT enabled. Table 3.9 summarizes the results for both configurations.

The Intel Xeon Gold 6438N results do not show a consistent feature-dependent timing pattern in either configuration. With DOIT disabled, all selected p-values remain above the significance threshold $\alpha = 10^{-5}$. The same is true with DOIT enabled. A few p-values in the DOIT-enabled run are lower than the rest, such as 0.034 and 0.020 for ML-DSA-87 reduction-count features, but they remain above the threshold and do not repeat across parameter sets or feature families.

The confidence intervals lead to the same interpretation. In both configurations, the intervals either cross zero, touch zero, or remain very close to zero. They therefore do not show a stable positive or negative timing effect for the selected features.

Enabling DOIT did not change the outcome of this experiment. The selected features remain non-significant with DOIT disabled and with DOIT enabled, and the confidence intervals do not show a stable separation from zero in either case. This suggests that the tested *Go* code path was not affected by the DOIT setting on this platform. The relevant operations may already have sufficiently uniform timing on the Intel Xeon Gold 6438N, or they may not fall into the instruction cases where DOIT changes timing behavior.

For `c-s1-n-need-reduction` in ML-DSA-44 with DOIT enabled, the sensitivity analysis gives additional context. The original Skillings–Mack p-value is 0.824, and the median estimate is 0.0 ns per unit with a 95% confidence interval from -1.0 to 1.0 ns per unit. This interval crosses zero, so the measured data do not indicate a stable timing effect for this feature.

After introducing a systematic artificial side-channel, the analysis detects injected effects down to 0.1 ns per unit. A 10 ns per-unit effect and a 1 ns per-unit effect both produce p-values reported as 0.0. A 0.1 ns per-unit effect produces a p-value of 8.63×10^{-6}, which is below the threshold $\alpha = 10^{-5}$. A 0.01 ns per-unit effect is not detected, with a p-value of 0.744.

This sensitivity result shows that the dataset and analysis procedure were able to detect small artificial value-dependent effects. The original measurements, however, do not show such a signal for the tested feature on this platform. The difference

between the Intel Xeon E5-2630 v3 and Intel Xeon Gold 6438N results remains a platform-dependent observation. Identifying the exact cause requires further investigation and is outside the scope of this work.

Table 3.9: Timing analysis results for the *Go* implementation on Intel Xeon Gold 6438N (DOIT disabled vs. enabled).

Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
Data Operand Independent Timing (DOIT) Disabled				
44	`bit-size-c-s1`	0.089	0	$[-1.0, 0.0]$
44	`bit-size-c-s2`	0.362	0	$[0.0, 1.0]$
44	`c-s1-n-need-reduction`	0.639	0	$[-2.0, 0.0]$
44	`c-s2-n-need-reduction`	0.926	1	$[1.0, 2.0]$
44	`hw-c-s1`	0.089	0	$[-1.0, 1.0]$
44	`hw-c-s2`	0.966	0	$[0.0, 1.0]$
65	`bit-size-c-s2`	0.269	-1	$[-2.0, 1.0]$
65	`c-s2-n-need-reduction`	0.296	0	$[-2.0, 1.0]$
65	`hw-c-s2`	0.336	-1	$[-2.0, 0.0]$
87	`bit-size-c-s1`	0.469	0	$[-1.0, 2.0]$
87	`bit-size-c-s2`	0.582	-1	$[-3.0, 0.0]$
87	`c-s1-n-need-reduction`	0.650	0	$[-2.0, 1.0]$
87	`c-s2-n-need-reduction`	0.174	1	$[-1.0, 2.0]$
87	`hw-c-s1`	0.351	1	$[-1.0, 3.0]$
87	`hw-c-s2`	0.574	0	$[-2.0, 2.0]$
Data Operand Independent Timing (DOIT) Enabled				
44	`bit-size-c-s1`	0.448	0	$[-1.0, 1.0]$
44	`bit-size-c-s2`	0.309	0	$[0.0, 1.0]$
44	`c-s1-n-need-reduction`	0.824	0	$[-1.0, 1.0]$
44	`c-s2-n-need-reduction`	0.439	0	$[-1.0, 1.0]$
44	`hw-c-s1`	0.740	0	$[-1.0, 1.0]$
44	`hw-c-s2`	0.052	-1	$[-1.0, 0.0]$
65	`bit-size-c-s2`	0.650	-1	$[-2.0, 1.0]$
65	`c-s2-n-need-reduction`	0.056	1	$[0.0, 3.0]$
65	`hw-c-s2`	0.604	1	$[0.0, 2.0]$
87	`bit-size-c-s1`	0.833	0	$[-2.0, 1.0]$
87	`bit-size-c-s2`	0.674	-1	$[-3.0, 1.0]$
87	`c-s1-n-need-reduction`	0.034	1	$[-1.0, 3.0]$
87	`c-s2-n-need-reduction`	0.020	1	$[-1.0, 2.0]$
87	`hw-c-s1`	0.470	0	$[-2.0, 1.0]$
87	`hw-c-s2`	0.966	0	$[-2.0, 2.0]$

3.2.2 OpenSSL and Nettle

The same partial-signing experiment was performed for *OpenSSL* and *Nettle*. In contrast to the full-signing measurements with the crafted dataset, the previously observed w_0-related timing dependence was no longer detected after excluding the signature encoding step. Table 3.10 summarizes representative results for the features most relevant to this comparison.

Table 3.10: Selected timing analysis results for *OpenSSL* and *Nettle* in the partial-signing setting.

Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
OpenSSL				
44	`bit-size-w0`	0.427	0	$[0.0, 0.0]$
44	`hw-w0`	0.895	0	$[0.0, 0.0]$
44	`c-s1-n-need-reduction`	0.564	0	$[0.0, 1.25]$
44	`c-s2-n-need-reduction`	0.175	0	$[0.0, 1.25]$
65	`bit-size-w0`	7.71×10^{-3}	0	$[-1.25, 0]$
65	`hw-w0`	0.637	0	$[0.0, 1.25]$
65	`c-s1-n-need-reduction`	0.982	0	$[-1.25, 1.25]$
65	`c-s2-n-need-reduction`	0.409	0	$[0.0, 1.25]$
87	`bit-size-w0`	0.527	0	$[0.0, 1.25]$
87	`hw-w0`	0.148	0	$[0.0, 1.25]$
87	`c-s1-n-need-reduction`	0.204	0	$[-1.25, 0.0]$
87	`c-s2-n-need-reduction`	0.846	0	$[0.0, 1.25]$
Nettle				
65	`bit-size-w0`	0.594	1.25	$[-2.5, 3.75]$
65	`hw-w0`	0.635	1.25	$[-1.25, 3.75]$
65	`c-s1-n-need-reduction`	0.700	-2.50	$[-5.0, 1.25]$
65	`c-s2-n-need-reduction`	0.156	0	$[-3.75, 3.75]$
87	`bit-size-w0`	0.619	1.25	$[-1.25, 2.5]$
87	`hw-w0`	0.059	0	$[-2.5, 2.5]$
87	`c-s1-n-need-reduction`	0.240	0	$[-2.5, 2.5]$
87	`c-s2-n-need-reduction`	0.135	-1.25	$[-3.75, 1.25]$

The table includes the w_0-related features that were significant in the full-signing setting, together with the reduction-count features `c-s1-n-need-reduction` and `c-s2-n-need-reduction`, which produced strong results for the *Go* implementation

in the partial-signing setting. For each feature, the table reports the Skillings–Mack p-value, the median estimate, and the corresponding 95% confidence interval.

These results differ from the full-signing crafted-dataset measurements, where w_0-related features produced strong signals for *OpenSSL* and *Nettle*. After the encoding step is excluded, those signals are no longer detected. The reduction-count features that produced strong results for *Go* in the partial-signing setting also do not show corresponding dependence in *OpenSSL* or *Nettle*.

Since the main difference between the full-signing and partial-signing measurements is the exclusion of the final signature encoding step, these results suggest that the earlier w_0-related signal was associated with the part of the signing path removed by the partial-signing harness. This does not isolate the exact encoding operation responsible for the effect, but it narrows the observed dependence to the part of the final encoding step, which operates on values included in the public signature.

3.2.3 dilithium-py

The *dilithium-py* implementation is evaluated separately from the optimized implementations. It provides a useful comparison point for the statistical analysis.

The measurements are performed on 10^6 messages for each parameter set using the crafted dataset and the partial-signing harness. Unlike *OpenSSL* and *Nettle*, *dilithium-py* continues to show statistically significant timing dependence after the encoding step is excluded. The detected dependence is not centered around w_0-related features, but appears across features derived from several reconstructed intermediate values, including $c \cdot s_1$, $c \cdot s_2$, and the masking vector y.

Table 3.11 summarizes selected results across all three ML-DSA parameter sets. The table reports the Skillings–Mack p-value, the median estimate, and the corresponding 95% confidence interval.

The table shows that the partial-signing measurements for *dilithium-py* remain strongly feature-dependent. Very small Skillings–Mack p-values appear for all three parameter sets, especially for features derived from $c \cdot s_1$, reduction counts, and y.

The magnitude of the estimated effects is much larger than in the optimized implementations. While the *Go* and *OpenSSL* partial-signing results are mostly on the order of a few nanoseconds per feature unit, several *dilithium-py* estimates remain in the hundreds of nanoseconds per unit. For example, in ML-DSA-44, `bit-size-y` has a median estimate of 548×10^{-9} s per unit, and `c-s2-n-need-reduction` has a median estimate of 769×10^{-9} s per unit.

Table 3.11: Selected timing analysis results for *dilithium-py* in the partial-signing setting.

Param.	Feature	Skillings–Mack p-value	Median Estimate [10^{-9} s/unit]	Median 95% CI [10^{-9} s/unit]
44	`bit-size-c-s1`	8.28×10^{-62}	-470	$[-850, -81]$
44	`bit-size-y`	3.74×10^{-27}	548	$[150, 923]$
44	`c-s1-n-need-reduction`	2.59×10^{-273}	-530	$[-924, -18]$
44	`c-s2-n-need-reduction`	1.00×10^{-26}	769	$[339, 1168]$
44	`hw-c-s1`	2.87×10^{-32}	-284	$[-646, 87.5]$
44	`y-n-need-reduction`	1.60×10^{-77}	295	$[-150, 709]$
65	`bit-size-c-s1`	1.29×10^{-117}	222	$[-316, 739]$
65	`bit-size-y`	1.15×10^{-12}	239	$[-335, 775]$
65	`c-s1-n-need-reduction`	0.0	-287	$[-854, 317]$
65	`y-n-need-reduction`	6.64×10^{-64}	745	$[127, 1352]$
65	`hw-c-s1`	2.21×10^{-47}	-85	$[-643, 420]$
87	`bit-size-c-s1`	0.0	262	$[-641.5, 1149]$
87	`bit-size-y`	8.67×10^{-8}	-372	$[-1324.5, 559]$
87	`c-s1-n-need-reduction`	0.0	421	$[-591, 1479]$
87	`c-s2-n-need-reduction`	2.90×10^{-20}	691	$[-413, 1713.5]$
87	`hw-c-s1`	0.0	-415	$[-1387, 540]$
87	`y-n-need-reduction`	8.77×10^{-20}	831	$[-242, 1819]$

In several cases, the median 95% confidence intervals stay away from zero. For ML-DSA-44, this is the case for `bit-size-c-s1`, `bit-size-y`, `c-s1-n-need-reduction`, and `c-s2-n-need-reduction`. These features therefore provide both a very small global p-value and a stable estimated timing effect.

Figure 3.7 shows a representative confidence-interval plot for `y-n-need-reduction`. This feature is selected because it illustrates that timing dependence in *dilithium-py* is not limited to features derived from $c \cdot s_1$ or $c \cdot s_2$, but also appears for the masking vector y. In this case, the p-value indicates strong statistical evidence of dependence, while the median confidence interval intersects zero, showing that the median estimate alone does not fully describe the detected dependence.

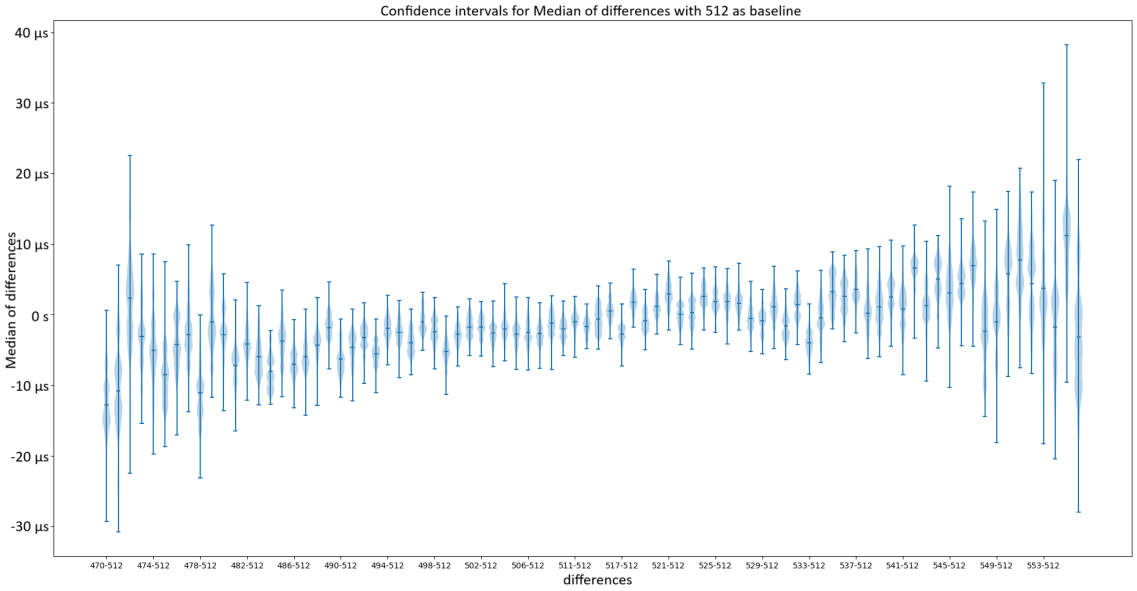


Fig. 3.7: Confidence intervals for median of differences between `y-n-need-reduction` classes for *dilithium-py* ML-DSA-44, measured with the crafted dataset using the partial-signing harness.

Compared with the full-signing crafted-dataset results, excluding the final encoding step does not remove the timing dependence in *dilithium-py*. The significant features remain spread across several reconstructed values, including $c \cdot s_1$, $c \cdot s_2$, and y. This differs from *OpenSSL* and *Nettle*, where the w_0-related signal disappeared after the encoding step was excluded.

3.3 Key comparison experiments

The previous experiments analyze timing dependence with respect to features reconstructed from individual signatures. This section uses a complementary class-based approach. Instead of grouping samples by reconstructed feature values, samples are grouped by the secret key used for signing. Each secret key therefore represents one timing class.

This follows the class-based methodology used in the Marvin analysis. The goal is to determine whether different secret keys produce consistently different signing times under the same measurement conditions. This result would indicate key-dependent timing behavior.

To reduce bias from temporal effects, the measurements are organized into rounds. In each round, all selected keys are used exactly once, and the key order is randomized. The statistical analysis then compares samples within the same

round, so that measurements collected under similar system conditions are evaluated against each other.

3.3.1 Random-message key comparison

The first key-comparison experiment uses ten independently generated secret keys and randomly generated messages. For each implementation and parameter set, 10^6 messages are used and divided across the ten key classes. The measurements are scheduled in randomized rounds, with each key used once per round. The experiment was performed on an Intel(R) Xeon(R) E5-2630 v3 processor.

The same messages and key sets are reused across implementations whenever supported by the implementation interface. The keys are generated independently, so each key class may differ in several components, including the secret vectors and ρ, which determines the public matrix A. A detected difference between key classes therefore shows key-dependent timing behavior, but does not identify which key component is responsible.

Table 3.12 summarizes the random-message key-comparison results. For each implementation and parameter set, the table reports the global Friedman p-value, the key pair identified as most different by the analysis, median estimate, and the confidence interval of the median timing difference for that pair.

Table 3.12: Summary of random-message key-comparison experiments.

Impl.	Param.	Friedman p-value	Key Pair (ID)	Median Estimate [10^{-6} s]	Median 95% CI [10^{-6} s]
Go	44	1.58×10^{-5}	01–09	0.222	$[0.113, 0.333]$
	65	0.186	04–05	0.163	$[0.048, 0.283]$
	87	0.006	00–03	0.124	$[0.064, 0.188]$
OpenSSL	44	9.23×10^{-116}	01–06	0.139	$[0.120, 0.163]$
	65	6.04×10^{-40}	07–09	-0.119	$[-0.141, -0.091]$
	87	1.72×10^{-160}	00–01	0.201	$[0.176, 0.218]$
Nettle	65	2.05×10^{-48}	01–02	-1.564	$[-1.843, -1.323]$
	87	2.22×10^{-80}	04–05	-1.238	$[-1.406, -1.104]$

OpenSSL and *Nettle* show very small Friedman p-values for the reported parameter sets, and the confidence intervals for the most different key pairs are separated from zero. The *Go* results are weaker. ML-DSA-44 gives a p-value close to the threshold, while ML-DSA-65 and ML-DSA-87 remain above it.

As a representative example of a significant class-based result, *Nettle* is presented with the ML-DSA-87 parameter set. The Friedman test reports p-value of 2.22×10^{-80}, indicating statistically significant differences between key classes. The smallest pairwise sign-test p-value is 4.11×10^{-38}, and the median pairwise p-value is 5.10×10^{-5}.

For the most different key pair identified by the analysis, key 04 and key 05, the median timing difference is -1.24×10^{-6} s, with a 95% confidence interval of $[-1.41, -1.10] \times 10^{-6}$ s. Since the confidence interval is separated from zero, this pair shows a measurable median timing difference between the two key classes.

Figure 3.8 shows the confidence intervals. Several key classes have intervals separated from zero relative to the baseline class, which is consistent with the global class-based test.

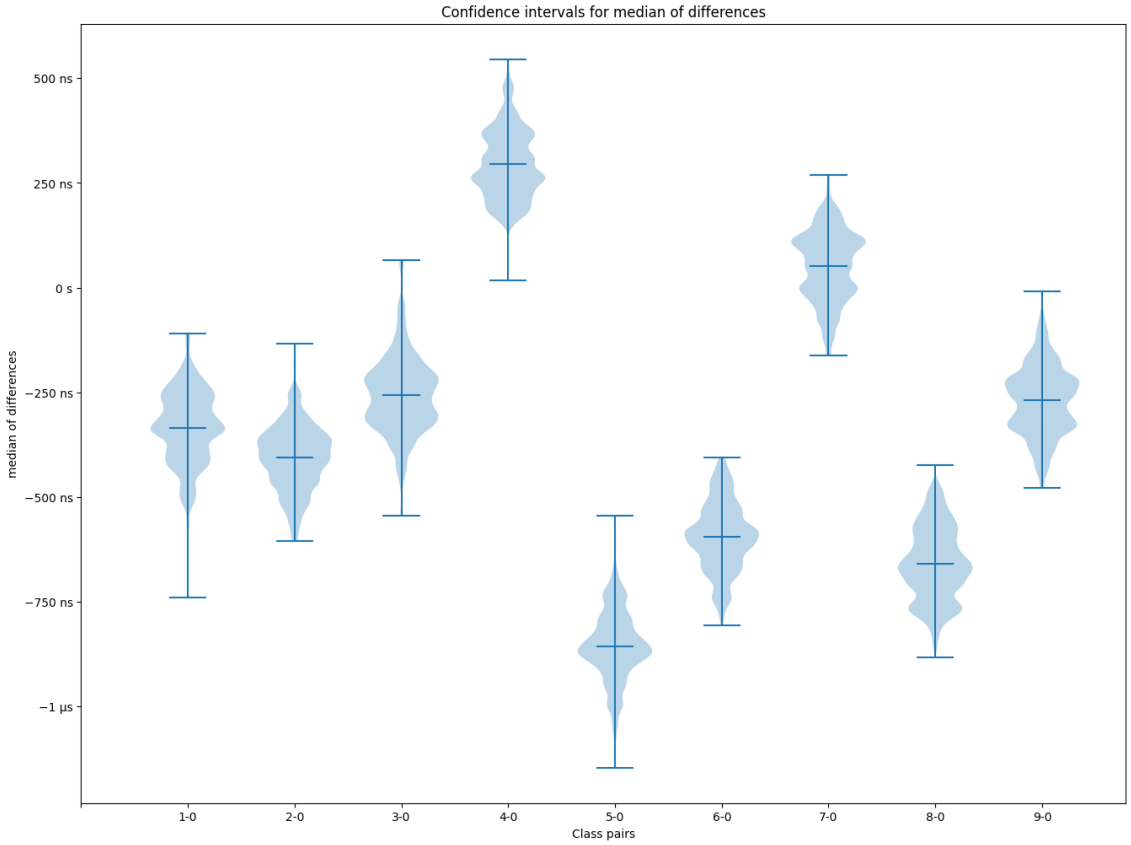


Fig. 3.8: Confidence intervals for median of differences between secret-key classes for *Nettle* ML-DSA-87 using random messages.

These results show that class-based timing differences can be detected with independently generated keys. However, the random-message setting still includes variation from rejection sampling, and the keys differ in several components at the same time. To narrow the interpretation, the next experiment uses crafted message datasets, where signing succeeds on the first attempt.

3.3.2 Crafted-message key comparison

The random-message key comparison shows detectable timing differences between key classes in several configurations. However, the interpretation is affected by three factors. First, normal signing includes rejection sampling, so different measurements may involve different numbers of signing attempts. Second, the measurements are performed on the full-signing operation, which includes the signature encoding. Third, independently generated keys differ in all key components. To reduce the first two sources of variability, we also perform a key-comparison experiment using crafted messages, and partial-signing harness.

This allows key-class differences to be evaluated under a more controlled execution path, where rejection behavior is fixed and the final encoding step is excluded. The measurement order follows the same randomized-round structure as before, with each key used once per round.

The experiment was performed for multiple implementations and parameter sets. Since the purpose of this subsection is to illustrate the remaining key-class effect under the crafted-message setting, we present one representative case in detail.

For *OpenSSL* ML-DSA-44, the class-based analysis reports a Friedman test p-value of 0.0, indicating statistically significant differences between key classes. The most different pair identified by the analysis is key 004 and key 005. For this pair, the median timing difference is estimated as -1.63×10^{-7} s, with a 95% confidence interval of $[-1.66, -1.59] \times 10^{-7}$ s. Since the confidence interval is separated from zero, the timing difference between these two key classes is also visible in the median estimate.

Figure 3.9 shows the confidence intervals of median timing differences between key classes. Several intervals are separated from the zero line, which is consistent with the Friedman test result and shows that some key classes have measurable timing differences relative to the baseline class.

The results are important as they show that key-class timing differences can remain detectable even when signing succeeds on the first attempt and the final signature encoding step is excluded. However, the keys are still independently generated, so the experiment does not identify which key component or internal operation is

responsible for the difference. The fixed-ρ experiment in the next subsection is used as an additional diagnostic step to reduce this ambiguity.

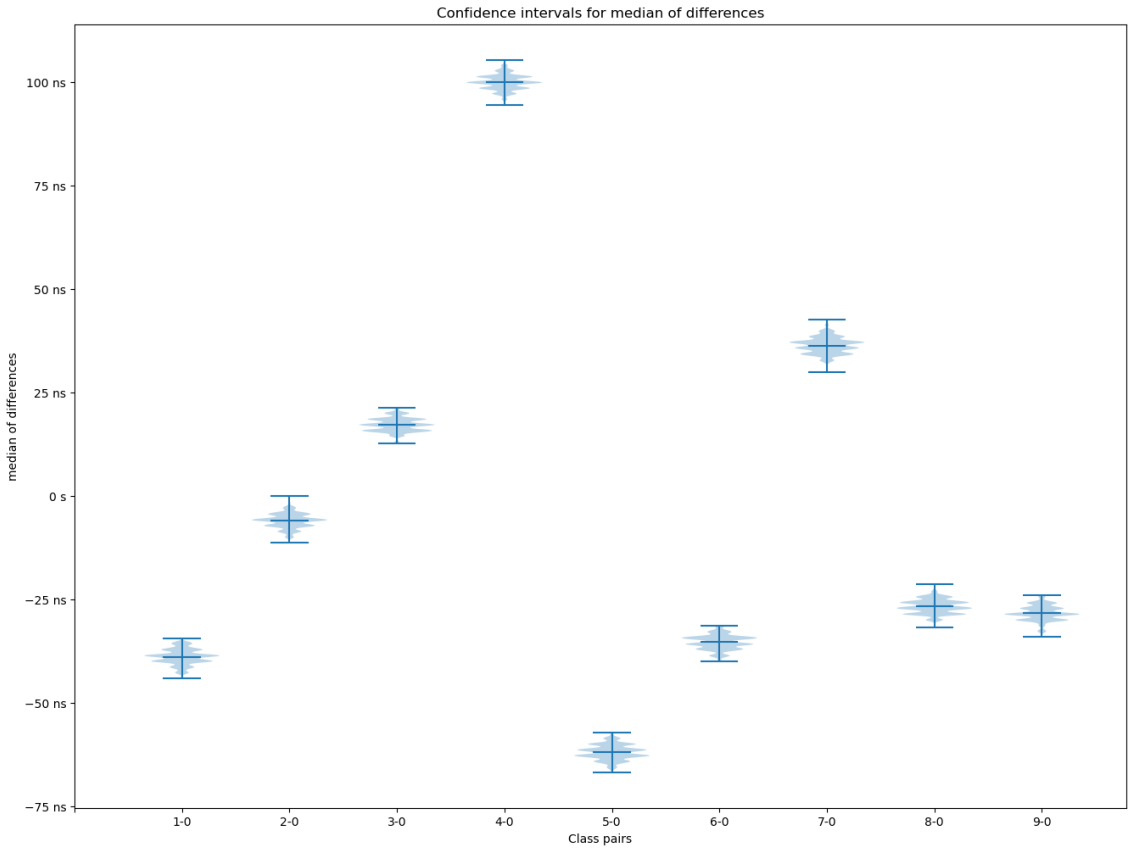


Fig. 3.9: Confidence intervals for median of differences between secret-key classes for *OpenSSL* ML-DSA-44 using the crafted dataset.

3.3.3 Fixed-ρ key comparison

To investigate whether variation in the public matrix A contributes to the observed key-class differences, we modified the *OpenSSL* key-generation procedure so that ρ is fixed across generated keys. Since ρ determines the expansion of A, this keeps the matrix identical for all key classes while allowing the remaining key material to vary.

This modification is not intended to represent normal key generation. It is used as a diagnostic setup to reduce variation caused by the public matrix and focus on the remaining key-dependent values. Listing 3.1 shows the fixed value used for ρ in the modified *OpenSSL* key-generation procedure.

Listing 3.1: Fixed value of ρ used in the modified *OpenSSL* key-generation experiment.

```
static const uint8_t fixed_rho[ML_DSA_RHO_BYTES] = {
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10,
    0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,
    0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f, 0x20
};
```

The fixed-ρ experiment was performed only for implementations where the modified key material could be loaded directly. This requires support for externally generated PEM keys. Therefore, the experiment was applied to *OpenSSL* and *Nettle*, but not to the *Go* implementation, whose ML-DSA interface accepts only seed-based keys.

We first repeated the fixed-ρ key-comparison experiment using random messages and the partial-signing harness. In this setting, key-class differences were harder to detect and required substantially larger datasets than in the ordinary independently generated key setting.

For *OpenSSL* ML-DSA-44, statistically significant key-class differences were detected with 5×10^6 random-message measurements. The Friedman test produced a p-value of 1.12×10^{-11}. The most different key pair was key 02 and key 06, with a median timing difference of 2.29×10^{-8} s and a 95% confidence interval of $[1.34, 3.05] \times 10^{-8}$ s.

For ML-DSA-87, using 10^7 random-message measurements produced a Friedman p-value of 1.08×10^{-9}. The most different key pair was key 07 and key 09, with a median timing difference of 2.48×10^{-8} s and a 95% confidence interval of $[1.05, 3.63] \times 10^{-8}$ s. For ML-DSA-65, also using 10^7 measurements, the Friedman p-value was 3.47×10^{-2}, so this run is not treated as a stable detection despite the most different pair having a median confidence interval separated from zero.

These results indicate that fixed-ρ key-class differences can still become detectable with random messages, but they require substantially larger datasets and the evidence is less stable. The experiment therefore suggests that fixing the public matrix A reduces the strength or detectability of the key-class differences observed with independently generated keys.

We then repeated the fixed-ρ experiment using the crafted-message dataset. In this setting, signing does not encounter rejections and the measurement excludes the final signature encoding step. The crafted-message fixed-ρ experiment was evaluated with 250 000 messages per key.

Table 3.13 summarizes the results for *OpenSSL* and *Nettle*. *Nettle* does not support the ML-DSA-44 parameter set, so only ML-DSA-65 and ML-DSA-87 are reported.

Table 3.13: Fixed-ρ key-comparison results using 250 000 crafted messages per key.

Impl.	Param.	Friedman p-value	Key Pair (ID)	Median Estimate [10^{-9} s]	Median 95% CI [10^{-9} s]
OpenSSL	44	0.503	004–008	-7	$[-12.0, -2.0]$
	65	0.157	000–002	-11	$[-22.0, 0.0]$
	87	0.286	002–007	-19	$[-35.0, -1.0]$
Nettle	65	0.278	004–006	-17	$[-30.0, -9.0]$
	87	0.143	002–003	20	$[9.0, 35.0]$

None of the fixed-ρ crafted-message experiments reaches the significance threshold $\alpha = 10^{-5}$ in the global Friedman test. These runs therefore do not provide evidence of statistically significant key-class timing dependence at the global class level. Some worst-pair confidence intervals are separated from zero. For example, *Nettle* ML-DSA-87 reports a median confidence interval of $[9.0, 35.0] \times 10^{-9}$ s for the most different key pair. However, it is not accompanied by a significant global Friedman test result.

The crafted-message fixed-ρ results should be interpreted together with the random-message fixed-ρ experiments. The random-message experiments showed that key-class differences can still become detectable when substantially larger datasets are used. However, with crafted messages and 250 000 measurements per key, the global class-based test remains non-significant for all tested configurations.

The fixed-ρ experiments therefore do not rule out smaller key-dependent timing effects. They show that, once the matrix A is held constant, key-class differences are harder to detect under the available measurement budget. A stronger conclusion would require larger crafted datasets and additional controlled experiments, which is outside the scope of this thesis. The fixed-ρ analysis is therefore treated as a diagnostic experiment.

3.4 Discussion of results

The experiments show that the detectability of timing dependence in ML-DSA signing depends strongly on the measurement setting. In the random-message full-signing setting, no consistent timing dependence on reconstructed features was detected. When the execution path was controlled or the measurement boundary was narrowed, statistically significant dependence appeared in some implementations and for several features.

We performed more than 300 tests across different implementations, parameter sets, measurement harnesses, processors, and reconstructed features. Because of this scale, the previous section presented only representative results. This discussion below focuses on the main observations and their interpretation.

The main point is that a statistical result cannot be interpreted independently of the measurement configuration. A detected signal may reflect data-dependent behavior in the measured code path, but its relevance depends on where the signal appears, whether it is reproduced across platforms, and whether it affects operations that are expected to be protected against timing leakage. Conversely, the absence of a signal does not prove constant-time behavior. It only shows that no dependence was detected under the tested conditions, dataset sizes, and selected feature set.

The following subsections discuss the main observations in more detail.

3.4.1 Observed timing correlations

The strongest feature-dependent results appeared in experiments using crafted message datasets. In these datasets, all signatures succeed on the first attempt, so the analysis is not dominated by differences in the number of rejection iterations. This makes the observed timing relationships easier to interpret than in the random-message full-signing setting.

In the full-signing setting, where the final signature encoding step is included, several implementations showed statistically significant dependence on features derived from w_0, especially `hw-w0` and `bit-size-w0`. This behavior was most visible for *Go* and *OpenSSL*, while the evidence for *Nettle* was weaker. These results showed that the complete signing path can contain measurable data-dependent timing variation when the rejection behavior is controlled.

The interpretation changes when the measurement boundary is narrowed. In the partial-signing experiments, the final signature encoding step is excluded. Under this boundary, the w_0-related signal observed for *OpenSSL* and *Nettle* in the full-signing setting was no longer detected. This suggests that the earlier w_0-related dependence

in these implementations was associated with the part of the signing path removed by the partial-signing harness, more precisely the signature encoding step.

For the *dilithium-py* implementation, statistically significant dependence was observed across a broader set of features. This included features derived from $c \cdot s_1$, $c \cdot s_2$, and the masking vector y. The lowest p-values were often associated with features describing coefficient representations, such as total bit-size or the number of coefficients requiring centered reduction. Since *dilithium-py* is a high-level reference implementation and is not designed for constant-time behavior, these results are mainly useful as a reference point showing that the analysis can detect feature-dependent timing variation when it is broadly present.

For the *Go* implementation, the partial-signing measurements produced statistically significant results only on selected processors. On the Intel Xeon E5-2630 v3 platform, low p-values were observed for features derived from $c \cdot s_1$ and $c \cdot s_2$, especially for features related to coefficient reduction. Similar behavior was also observed on the Intel Xeon Silver 4216 platform.

On the Intel Xeon Silver 4116 platform, the same features did not show statistically significant dependence. The AMD EPYC 7742 measurements also did not show the same behavior.

These results were discussed with the *Go* developers. As part of this discussion, the relevant *Go* 1.26 amd64 code paths were inspected, including arithmetic routines, decomposition, hint generation, NTT and inverse NTT operations, sampling, and rejection checks. This inspection did not identify data-dependent branches or obvious variable-time instructions in the relevant routines, apart from rejection-related early exits and sampling behavior. Since rejection sampling is removed in the crafted partial-signing experiments, this did not explain the observed signal.

Following this discussion, the experiment was also repeated on an Intel Xeon Gold 6438N platform with Intel DOIT support. This platform was tested both with DOIT disabled and with DOIT enabled. In neither configuration did the selected features show the same timing pattern as on the Intel Xeon E5-2630 v3 platform. Enabling DOIT also did not change the interpretation of the measurements, since the selected features remained non-significant in both configurations.

These observations suggest that the *Go* result is not solely a property of the source-level implementation. The observed timing dependence is therefore interpreted as platform-dependent behavior whose exact cause remains unresolved. It is not attributed here to a specific instruction sequence, *Go* implementation bug, or processor feature.

The results also show why the measurement boundary matters. Full-signing timing can reveal effects in the complete signing path, including encoding, but it does not identify which part of the signing operation is responsible. Partial measurements provide a more focused view of the signing computation before signature encoding, but the *Go* results show that such measurements must still be interpreted together with the hardware platform on which they were collected.

3.4.2 Weak or inconclusive signals

Several results in the experimental evaluation require careful interpretation. The statistical tests can indicate that execution time depends on a selected feature, but they do not by themselves identify the exact operation responsible for the timing dependence. The interpretation therefore depends on the analyzed feature and on the measurement boundary.

One source of ambiguity is the measurement boundary. The crafted full-signing experiments detected w_0-related timing dependence, but the measured interval still included the final signature encoding step. The later partial-signing experiments showed that this signal disappeared when encoding was excluded. This suggests that the signal was not located in the measured pre-encoding part of signing, but the exact operation responsible for it was not isolated.

A second source of ambiguity comes from the key-comparison experiments. These experiments show that timing differences between key classes can be statistically detectable. In the random-message setting, several implementations produced significant class-based results, and the confidence intervals for the most different key pairs were separated from zero. However, independently generated keys differ in several components at the same time, including the secret vectors and ρ, which determines the public matrix A. The random-message key-comparison results therefore show key-class timing dependence, but they do not identify which key component is responsible.

To reduce this ambiguity, the fixed-ρ experiment keeps A identical across key classes while allowing the remaining key material to vary. This setup is not intended to represent normal key generation. It is used as a diagnostic experiment that reduces variation caused by the public matrix and focuses the comparison on the remaining key-dependent values.

The results were mixed. With random messages, key-class differences were still detectable for some parameter sets, but only with substantially larger datasets. In the crafted-message setting, the fixed-ρ experiments with 250 000 messages per key did not reach the global significance threshold. Some worst-pair confidence intervals were separated from zero, but these pairwise results were not supported by a sig-

nificant global test. The crafted-message fixed-ρ results are therefore inconclusive rather than negative in a strong sense.

These results do not rule out smaller key-dependent timing effects. They show that once ρ is fixed and the matrix A is held constant, any remaining key-class differences are harder to detect with the available dataset sizes. Larger crafted-message datasets would be needed to determine whether such differences persist. The fixed-ρ analysis should therefore be interpreted as a diagnostic experiment rather than a complete conclusion about key-dependent leakage.

3.4.3 Negative results

A substantial part of the evaluation produced negative results, especially in the random-message full-signing setting. These results are important because they describe what was detectable through the normal signing path under the applied measurement methodology.

In this setting, the complete signing operation was measured using random messages. The measured interval included rejection sampling, the internal signing computation, and final signature encoding. Across the evaluated implementations and parameter sets, no consistent timing dependence on the analyzed features was detected.

This should not be interpreted as proof that the implementations are constant time. It shows only that, under random inputs and full-signing measurements, the selected feature dependencies were not visible with the tested dataset sizes and analysis procedure. If such a dependence had been detected in this setting, it would have been directly relevant, because it would appear in the complete signing operation exposed by the implementation.

The result is still meaningful. The timing correlations detected later in controlled experiments are not trivially visible in end-to-end signing measurements with random inputs. This supports the use of crafted datasets and narrower measurement boundaries to study weaker timing relationships that may be hidden in the complete signing runtime.

3.4.4 Responsible disclosure

Before publication of this thesis, we reported the leakage described in Section 3.4.1 to the *Go* developers. We then discussed the observation with them and performed additional tests based on their feedback. We later asked whether the methodology and results could be included in this thesis. Since the immediate publication risk was considered low, the developers did not object to publishing the results.

Conclusion

The goal of this thesis was to determine whether the execution time of ML-DSA signing shows measurable dependence on features derived from selected intermediate values, and whether such dependencies could form a timing side channel. To perform this analysis, we developed a toolkit for generating inputs, measuring execution time, reconstructing intermediate values, and extracting features from them. The toolkit produces data in a format suitable for statistical evaluation using *tlsfuzzer*. This allows the extracted features to be compared with observed execution times, following the idea of the Marvin and Minerva analyses in the setting of a post-quantum signature scheme.

The experimental evaluation showed that the detectability of timing dependence depends strongly on the measurement setting. In the random-message full-signing setting, no consistent timing dependence on the reconstructed features was detected across the evaluated implementations. This does not prove that the implementations are constant-time. Rather, it shows that, for the tested dataset sizes and measurement configuration, the timing dependencies were not visible at the level of the complete signing operation.

The evaluation then moved to more controlled settings. When the crafted dataset was used while still measuring the complete signing operation, statistically significant signals appeared for features related to the low-bit component w_0. However, these measurements still included the final signature encoding step, so the observed dependence could not be attributed directly to the pre-encoding part of signing. After the measurement boundary was narrowed to exclude the encoding step, the w_0-related signals no longer dominated the results. Instead, timing dependence appeared in selected cases for features derived from intermediate values such as $c \cdot s_1$, $c \cdot s_2$, and the masking vector y. The most relevant features described bit-size, Hamming weight, and the number of coefficients requiring reduction.

The strongest feature-based signals were observed for the *dilithium-py* reference implementation and for the *Go* implementation on selected processors. The behavior observed for *dilithium-py* is consistent with its role as a high-level reference implementation, rather than an implementation designed for constant-time execution. The behavior observed for *Go* was more subtle. Timing dependence was detected on some Intel processors, but the same signal was not observed on other tested platforms, including AMD EPYC and an Intel Xeon Gold system tested both with and without DOIT enabled. Discussions with *Go* developers and inspection of the relevant code paths did not identify an obvious variable-time operation. The *Go* results are therefore best interpreted as platform-dependent timing behavior whose cause remains unresolved.

For *OpenSSL* and *Nettle*, the partial-signing measurements did not confirm the feature-dependent signals observed in the full-signing crafted setting. In particular, the earlier w_0-related signals disappeared after the signature encoding step was excluded. This suggests that the full-signing signals were associated with the encoding phase, rather than with the pre-encoding part of signing.

The class-based key-comparison experiments provided a complementary view. With independently generated keys, timing differences between key classes were statistically detectable in several configurations. However, these differences cannot be attributed to a specific secret-dependent operation, because independently generated keys differ in several components. This also includes ρ, which determines the public matrix A. To further investigate, we introduced a diagnostic fixed-ρ experiment, where A was kept constant across key classes. The available fixed-ρ results suggest that key-class differences become harder to detect when the matrix is fixed. However, larger datasets would be needed to draw a stronger conclusion.

The results show that timing dependence in ML-DSA signing is subtle and highly context-dependent. It may remain undetectable in end-to-end measurements, but become observable under more controlled conditions that isolate selected parts of the signing procedure. These findings do not indicate a practical attack. Rather, they show that timing-dependent behavior can be detected statistically under selected measurement conditions.

The developed tool provides a foundation for future work. The feature set can be extended with additional properties of ML-DSA intermediate values. The evaluation can also be expanded to more implementations and processor families to better understand the role of platform-specific effects. Finally, larger crafted datasets could be used to further investigate the fixed-ρ key-comparison results.

This work will continue as part of the QARC European project. In this context, the tool developed in this thesis can support further analysis of post-quantum cryptographic implementations.

Bibliography

[1] NIST. *What Is Post-Quantum Cryptography?* Online. 2025. Available from: https://www.nist.gov/cybersecurity/what-post-quantum-cryptography. [accessed 2025-12-02].

[2] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates.* Online. 2022. Available from: https://csrc.nist.gov/news/2022/pqc-candidates-to-be-standardized-and-round-4. [accessed 2025-12-04].

[3] DUCAS, L.; KILTZ, E.; LEPOINT, T.; LYUBASHEVSKY, V.; SCHWABE, P. et al. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. Online. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2018, p. 238-268. ISSN 2569-2925. Available from: https://doi.org/10.46586/tches.v2018.i1.238-268. [accessed 2025-12-03].

[4] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Module-Lattice-Based Digital Signature Standard.* Online. 2024. Available from: https://doi.org/10.6028/nist.fips.204. [accessed 2025-07-19].

[5] KOCHER, P. C. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems.* Online. Lecture Notes in Computer Science. 1996, p. 104-113. ISBN 9783540615125. ISSN 0302-9743. Available from: https://doi.org/10.1007/3-540-68697-5_9. [accessed 2025-10-25].

[6] KARIO, H. *Everlasting ROBOT: The Marvin Attack.* Online. Computer Security – ESORICS 2023. Lecture Notes in Computer Science. 2024, pp. 243-262. ISBN 978-3-031-51478-4. Available from: https://doi.org/10.1007/978-3-031-51479-1_13. [accessed 2025-09-27].

[7] JANCAR, J.; SEDLÁČEK, V.; ŠVENDA, P. and SÝS, M. *Minerva: The curse of ECDSA nonces.* Online. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2020, p. 281-308. ISSN 2569-2925. Available from: https://doi.org/10.46586/tches.v2020.i4.281-308. [accessed 2025-09-29].

[8] KARIO, A. *Marvin-toolkit.* Online. GitHub. Available from: https://github.com/tomato42/marvin-toolkit. [accessed 2025-09-28].

[9] PANTELAKIS, G. *Minerva-toolkit.* Online. GitHub. Available from: https://github.com/GeorgePantelakis/minerva-toolkit. [accessed 2025-10-15].

[10] KOCHER, P.; JAFFE, J. and JUN, B. *Differential Power Analysis*. Online. Lecture Notes in Computer Science. 1999, p. 388-397. ISBN 9783540663478. ISSN 0302-9743. Available from: https://doi.org/10.1007/3-540-48405-1_25. [accessed 2026-02-18].

[11] GANDOLFI, K.; MOURTEL, C. and OLIVIER, F. *Electromagnetic Analysis: Concrete Results*. Online. Lecture Notes in Computer Science. 2001, p. 251-261. ISBN 9783540425212. ISSN 0302-9743. Available from: https://doi.org/10.1007/3-540-44709-1_21. [accessed 2026-02-18].

[12] GENKIN, D.; SHAMIR, A. and TROMER, E. RSA Key *Extraction via Low-Bandwidth Acoustic Cryptanalysis.* Online. Lecture Notes in Computer Science. 2014, s. 444-461. ISBN 9783662443705. ISSN 0302-9743. Available from: https://doi.org/10.1007/978-3-662-44371-2_25. [accessed 2026-02-18].

[13] OSVIK, D. A.; SHAMIR, A. and TROMER, E. *Cache Attacks and Countermeasures: The Case of AES*. Online. Lecture Notes in Computer Science. 2006, p. 1-20. ISBN 9783540310334. ISSN 0302-9743. Available from: https://doi.org/10.1007/11605805_1. [accessed 2026-02-18].

[14] KOCHER, P.; HORN, J.; FOGH, A.; GENKIN, D.; GRUSS, D. et al. *Spectre Attacks: Exploiting Speculative Execution*. Online. 2019 IEEE Symposium on Security and Privacy (SP). 2019, p. 1-19. Available from: https://doi.org/10.1109/sp.2019.00002. [accessed 2026-02-18].

[15] BONEH, D.; DEMILLO, R. A. and LIPTON, R. J. *On the Importance of Checking Cryptographic Protocols for Faults.* Online. Lecture Notes in Computer Science. 1997, p. 37-51. ISBN 9783540629757. ISSN 0302-9743. Available from: https://doi.org/10.1007/3-540-69053-0_4. [accessed 2026-02-18].

[16] JANCAR, J.; FOURNÉ, M.; BRAGA, D. De Almeida; SABT, M.; SCHWABE, P. et al. *"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks.* Online. 2022 IEEE Symposium on Security and Privacy (SP). 2022, p. 632-649. Available from: https://doi.org/10.1109/sp46214.2022.9833713. [accessed 2025-10-25].

[17] BLEICHENBACHER, D. *Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1.* Online. Lecture Notes in Computer Science. 1998, p. 1-12. ISBN 9783540648925. ISSN 0302-9743. Available from: https://doi.org/10.1007/bfb0055716. [accessed 2025-12-06].

[18] KALISKI, B. *PKCS #1: RSA Encryption Version 1.5.* Online. 1998. Available from: https://doi.org/10.17487/rfc2313. [accessed 2025-12-06].

[19] KARIO, H. *The Marvin Attack*. Online. April 2025. Available from: https://people.redhat.com/~hkario/marvin/. [accessed 2026-04-23].

[20] PORNIN, T. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. Online. 2013. Available from: https://doi.org/10.17487/rfc6979. [accessed 2026-04-24].

[21] KARIO, H. *Out of the Box Texting*. Online. Cryptology ePrint Archive, Paper 2023/1441. Available from: https://ia.cr/2023/1441. [accessed 2025-09-28].

[22] RED HAT. *tlsfuzzer*. Online. Read the Docs. Available from: https://tlsfuzzer.readthedocs.io/en/latest/. [accessed 2025-11-09].

[23] CHATFIELD, M. and MANDER, A. *The Skillings–Mack Test (Friedman Test when There are Missing Data)*. Online. In: The Stata Journal: Promoting communications on statistics and Stata. SAGE Publications, 2009, p. 299-305. ISSN 1536-867X. Available from: https://doi.org/10.1177/1536867x0900900208. [accessed 2025-11-17].

[24] PYPI. Dilithium-py. *CRYSTALS-Dilithium Python Implementation.* Online. Available from: https://pypi.org/project/dilithium-py/. [accessed 2025-11-17].

[25] OPENSSL. *TLS/SSL and crypto library*. Online. Available from: https://github.com/openssl/openssl. [accessed 2026-04-12].

[26] GOLANG. *The Go programming language*. Online. Available from: https://github.com/golang/go. [accessed 2026-04-12].

[27] GO. *Mldsa package - crypto/internal/fips140/mldsa - Go Packages*. Online. Available from: https://pkg.go.dev/crypto/internal/fips140/mldsa. [accessed 2026-04-12].

[28] NETTLE. *Nettle - a low-level cryptographic library*. Online. Available from: https://www.lysator.liu.se/~nisse/nettle/. [accessed 2026-04-12].

[29] Daiki Ueno. *nettle*. Online. Available from: https://www.lysator.liu.se/~nisse/nettle/. [accessed 2026-04-12].

[30] TLSFUZZER. *tlsfuzzer*. Online. GitHub. Available from: https://github.com/tlsfuzzer/tlsfuzzer/tree/master. [accessed 2025-10-15].

[31] VALA, R. *mldsa-toolkit*. Online. GitHub. Available from: https://github.com/rvala02/mldsa-toolkit/tree/main. [accessed 2025-12-01].

[32] AZOUAOUI, M.; BRONCHAIN, O.; CASSIERS, G.; HOFFMANN, C.; KUZOVKOVA, Y. et al. Protecting Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations. Online. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2023, p. 58-79. ISSN 2569-2925. Available from: https://doi.org/10.46586/tches.v2023.i4.58-79. [accessed 2026-03-14].

[33] INTEL. *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance.* Online. 2025. Available from: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html. [accessed 2026-05-10].

[34] KARIO, A. *mlkem-sct-toolkit*. Online. GitHub. Available from: https://github.com/tlsfuzzer/mlkem-sct-toolkit. [accessed 2026-05-12].

Symbols and abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
DOIT	Data Operand Independent Timing
ECDSA	Eliptic Curve Digital Signature Algorithm
ML-DSA	Module-Lattice Digital Signature Algorithm
MLWE	Module Learning With Errors
MSIS	Module Short Integer Solution
NTT	Number Theoretic Transform
TLS	Transport Layer Security

A Contents of the electronic attachment

The electronic attachment contains the `mldsa-toolkit` repository with the source code used in the practical part of this thesis. It includes the code relevant to the experiments described in this work. The public repository [31] may contain additional harnesses, experimental code, or later changes that are not part of the submitted attachment.

The toolkit contains measurement harnesses for the evaluated implementations, scripts for dataset preparation and scheduling, and script for reconstruction of intermediate values and feature extraction. Its structure follows the approach used in timing-analysis toolkits such as Marvin and Minerva. The `mlkem-sct-toolkit` [34], developed for timing side-channel analysis of ML-KEM, provided a useful reference for the design of the measurement harnesses.

The extraction script is `extract.py`, which comes from the *tlsfuzzer* [30] project. This file is not my original work and is distributed under the GNU General Public License version 2. The license text is included in the attachment. The file is included here because it was extended in this work with ML-DSA reconstruction and feature-extraction code.

Most of the measurement harnesses and helper scripts were implemented as part of this thesis. Some files were based on external contributions. In particular, the Nettle `timing.c` and `key_timing.c` harnesses were contributed externally. The initial Go `key_timing.go` harness was also not written as part of this thesis, but it was extended and adapted for the experiments described here.

```
mldsa-toolkit
├── dilithium-py/ ..... measurement harnesses for dilithium-py
│   ├── key_timing_no_encode.py
│   ├── key_timing.py
│   ├── README.md
│   ├── timing_no_encode.py
│   └── timing.py
├── golang/ ..... measurement harnesses for the Go implementation
│   ├── key_timing.go
│   ├── key_timing_no_encode.go
│   ├── README.md
│   ├── timing.go
│   └── timing_no_encode.go
├── nettle/ ..... measurement harnesses for the Nettle implementation
│   ├── key_timing.c
│   ├── key_timing_no_encode.c
│   ├── README.md
│   ├── timing.c
│   └── timing_no_encode.c
├── openssl/ ..... measurement harnesses for the OpenSSL implementation
│   ├── key_timing.c
│   ├── key_timing_no_encode.c
│   ├── README.md
│   ├── timing.c
│   └── timing_no_encode.c
├── scripts/ ..... data generation and scheduling scripts
│   ├── create_folders.py ... creates folders and copies measurement files into them
│   ├── generate_data.py ..... generates random messages and single-key datasets
│   ├── generate_crafted_dataset.py ..... generates specially crafted messages
│   ├── generate_keys.py ..... generates multiple key pairs
│   ├── generate_schedule_keys.py . generates aligned schedules for crafted datasets
│   └── generate_schedule.py ..... generates randomized key schedules
├── setup.sh ..... helper setup script
├── extract.py ..... reconstruction and feature extraction tool
├── README.md ..... repository documentation
└── LICENSE ..... license text for the tlsfuzzer GPLv2 code
```