BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2024 Vladyslav Shapoval, BA

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

LATTICE-BASED CRYPTOGRAPHY ON CONSTRAINED DEVISES

KRYPTOGRAFIE ZALOŽENÁ NA MŘÍŽKÁCH NA OMEZENÝCH ZAŘÍZENÍCH

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR Vladyslav Shapoval, BA

AUTOR PRÁCE

SUPERVISOR M.Sc. Sara Ricci, Ph.D.

VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Vladyslav Shapoval, BA **ID:** 236349

Year of study: 2 ***Academic year:*** 2023/24

TITLE OF THESIS:

Lattice-Based Cryptography on Constrained Devises

INSTRUCTION:

The topic is focused on the development of a lattice-based signature on constrained devices such as microcontrollers. The student will investigate novel implementation techniques for high performance and resource efficiency of the needed building blocks on embedded microcontrollers. Moreover, the implemented signature will be deployed in a real-world scenario (agreed with the supervisor) including performance measurement and brief instructions for installation.

RECOMMENDED LITERATURE:

1] Pöppelmann T. Efficient implementation of ideal lattice-based cryptography. it-Information Technology. 2017 Dec 20;59(6):305-9.

2] Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schwabe P, Seiler G, Stehlé D. Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems. 2018 Feb 14:238-68.

Date of project specification: 5.2.2024 ***Deadline for submission:*** 21.5.2024

Supervisor: M.Sc. Sara Ricci, Ph.D.

doc. 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 master's thesis presents a modified software implementation of the module-lattice-based signature scheme Dilithium and its distributed variant DS2 for the ARM Cortex-M4 microcontroller. Dilithium is a part of the CRYSTALS suite and was selected by the NIST as a new post-quantum signature standard. This work is focused on reducing the memory footprint of both algorithms in order to make them more applicable to a wider spectrum of microcontrollers and constrained devices. Both signatures were optimized to run on the STM32 Cortex-M4 microcontroller. On one hand, Dilithium signature presented an already optimized implementation that can run on a microcontroller. Therefore, we focused on adding hardware acceleration support for AES for the generation of pseudo-random numbers during the generation of the signature. On the other hand, DS2 signature is more memory demanding and we proposed two microcontroller-tailored optimization approaches. These optimizations aim to reduce memory consumption while maintaining security strength. Experimental results and security analysis demonstrate the efficacy and practicality of our solutions. As a result of our work, we successfully developed new versions of both Dilithium and DS2 with memory consumption reduced by more than 50% and 90%, respectively, compared to the original.

KEYWORDS

Dilithium, Post-quantum cryptography, LWE, microcontrollers, threshold signature, quantum-resistant, RAM-optimization

ABSTRAKT

Tato diplomová práce prezentuje modifikovanou softwarovou implementaci podpisového schématu založeného na modulové mřížce Dilithium a jeho distribuované varianty DS2 pro mikrokontrolér ARM Cortex-M4. Dilithium je součástí sady CRYSTALS a byl vybrán NIST jako nový postkvantový podpisový standard. Tato práce se zaměřuje na snížení paměťové náročnosti obou algoritmů, aby byly více aplikovatelné na širší spektrum mikrokontrolérů a omezených zařízení. Oba podpisy byly optimalizovány pro běh na mikrokontroléru STM32 Cortex-M4. Na jedné straně Dilithium podpis prezentoval již optimalizovanou implementaci, která může běžet na mikrokontroléru. Proto jsme se zaměřili na přidání hardwarové akcelerace pro AES pro generování pseudonáhodných čísel během generování podpisu. Na druhé straně je podpis DS2 více paměťově náročný a navrhli jsme dva optimalizační přístupy přizpůsobené mikrokontroléru. Tyto optimalizace mají za cíl snížit spotřebu paměti při zachování bezpečnostní síly. Experimentální výsledky a bezpečnostní analýza demonstrují účinnost a praktičnost našich řešení. V důsledku naší práce jsme úspěšně vyvinuli nové verze jak Dilithium, tak DS2 s paměťovou spotřebou sníženou o více než 50% a 90%, respektive, ve srovnání s originálem.

KLÍČOVÁ SLOVA

Dilithium, Postkvantová kryptografie, LWE, mikrokontroléry, prahový podpis, kvantově odolné, optimalizace RAM

Typeset by the thesis package, version 4.09; https://latex.fekt.vut.cz/

ROZŠÍŘENÝ ABSTRAKT

V této práci se zabýváme problematikou využití postkvantových kryptografických algoritmů v praxi v zařízeních s omezenými výpočetními zdroji. Jako základ jsme se rozhodli vzít algoritmus Dilithium, který byl nedávno schválen NIST jako nový standard pro postkvantový digitální podpis (link), a také jeho distribuovanou modifikaci DS2, což je (n, n) multi-podpisové schéma. Dodaný zdrojový kód pro implementace Dilithium a DS2 jsme analyzovali a na jejich základě vytvořili upravené verze vhodné pro praktické použití v zařízeních s omezenou pamětí, jako jsou mikrokontroléry. Nové verze Dilithium a DS2 byly testovány na mikrokontroléru STM32WB55 s jádrem Cortex-M4. Na závěr uvádíme výsledky našich experimentálních měření. Nové verze Dilithium a DS2 byly testovány na mikrokontroléru, postaveném na jádře Cortex-M4. Zejména byla vytvořena praktická ukázka aplikace algoritmu DS2 na malých fyzických zařízeních založených na mikrokontrolu STM32WB55 a využívajících standard 802.15.4 pro bezdrátovou komunikaci.

První kapitola je věnována teoretickému základu nezbytnému pro pochopení principu fungování jak veškeré postkvantové kryptografie postavené na algebraických svazech, tak i algoritmu Dilithium. Zejména se zaměřujeme na definování problému "Learning with Errors" (LWE) a jeho odvozenin: Ring-LWE a Module-LWE. Poté uvádíme stručný popis algoritmů Dilithium a DS2, stejně jako dalších technologií, jako je šifrovací algoritmus AES a hashovací algoritmus SHAKE.

Druhá kapitola podrobně popisuje změny, které jsme provedli v obou algoritmech, abychom je mohli provozovat na mikrokontroléru. Naším hlavním příspěvkem k této práci je zavedení dvou hlavních změn. První je „komprese“ polynomických matic používaných ve výpočtech Dilithium a DS2. Generování pseudonáhodných matic je založeno na náhodném seedu a SHAKE hashovacím algoritmu. Vzhledem k tomu, že v každém okamžiku můžeme pracovat pouze s jedním blokem matice, nemá smysl uchovávat v paměti celou matici, ale spíše si požadovaný blok vygenerovat podle potřeby. Podařilo se nám tedy snížit spotřebu paměti pro Dilithium o 55% a pro DS2 o 80%. Druhá změna se týká výhradně DS2. Skládá se z přidání výkonného centrálního uzlu (Central Node), se kterým budou komunikovat všichni účastníci algoritmu. Algoritmus pro generování podpisu DS2 jsme zároveň rozdělili na dvě části: kritickou část, která zahrnuje výpočty s tajným klíčem a která musí být vždy prováděna přímo na zařízení s mikrokontrolérem (Secure Node); a nekritickou část, která nevyžaduje tajný klíč a vypočítává se na základě parametrů známých všem účastníkům. Tato nekritická část je outsourcována pro výpočet do centrálního uzlu, což výrazně urychluje výpočet podpisu a eliminuje omezení počtu účastníků, dříve diktované omezeními paměti mikrokontroléru. Na konci kapitoly uvádíme bezpečnostní analýzu upraveného algoritmu, abychom dokázali, že nedovolujeme

úniky tajných informací.

Poslední kapitola je věnována analýze výpočetní náročnosti původního a modifikovaného algoritmu Dilithium a DS2, a to jak z hlediska paměti, tak i doby výpočtu. Teoretické výsledky porovnáváme s reálnými měřeními výkonu. Nakonec se nám podařilo snížit spotřebu paměti DS2 o 90% oproti původní implementaci. Ukázalo se však, že upravené algoritmy jsou 3-4krát pomalejší než originály, v závislosti na počtu odmitnutí při generování podpisu. Za hlavní faktory tak výrazného poklesu výkonu jsme označili nedostatečně rychlou implementaci algoritmu SHAKE a optimalizované použití výpočtů s pohyblivou řádovou čárkou. Na závěr jsme prezentovali výsledky experimentu, ve kterém se nám podařilo zlepšit výkon Dilithia nahrazením softwarové implementace algoritmu SHAKE hardwarovou implementací hash algoritmu založeného na AES-CTR-256, který urychlil algoritmu pětkrát ve srovnání s předchozí verzí.

SHAPOVAL, Vladyslav. *Lattice-Based Cryptography on Constrained Devises*. Master's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2024. Advised by M.Sc. Sara Ricci, Ph.D

Author’s Declaration

Author:	BA. Vladyslav Shapoval
Author’s ID:	236349
Paper type:	Master’s Thesis
Academic year:	2023/24
Topic:	Lattice-Based Cryptography on Constrained Devises

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, 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 the advisor of my thesis, M.Sc. Sara Ricci Ph.D for patience, professionalism, consultation and guidance during the work.

Contents

Introduction **15**

Aim of the thesis **16**

1 Introduction into Lattice-based cryptography **17**
1.1 Basic definitions 17
1.2 Learning With Errors 20
1.2.1 Ring Learning With Errors 21
1.2.2 Module Learning With Errors 21
1.3 Polynomial Arithmetic 22
1.3.1 Multiplication of polynomials 22
1.3.2 Montgomery multiplication 25
1.3.3 Discrete Gaussian Sampling 26
1.4 Dilithium 26
1.5 DS2 28
1.6 Advanced Encryption Standard 30
1.7 SHAKE 30
1.8 STM32WB55xx description 31

2 Dilithium and DS2 signatures proposed optimization on restricted devises **32**
2.1 Preliminaries 32
2.1.1 Dilithium Reference implementation overview 32
2.1.2 DS2 Reference implementation overview 34
2.2 Reduction of memory consumption 34
2.2.1 Compression Optimization 34
2.2.2 Outsource Optimization 37
2.3 DS2 Security analysis 40

3 Implementation of Dilithium and DS2 signatures **43**
3.1 Testing Methodology 43
3.1.1 Choice of microcontroller for testing 43
3.2 Dilithium memory consumption analysis 43
3.3 Dilithium time complexity analysis 45
3.4 DS2 Memory consumption analysis 49
3.5 DS2 Time complexity analysis 50
3.6 Addition of hardware support for AES to Dilithium 53

Conclusion **56**

Bibliography **58**

Symbols and abbreviations **63**

List of appendices **64**

A Content of the Dilithium reference implementation package **65**

B Content of the DS2 reference implementation package **67**

C Content of the electronic attachment **68**

List of Figures

1.1 2D Lattice with it’s basis, fundamental region and non-basis vectors . 18
1.2 Example of the 1 and 2 successive minimums 19
2.1 Outsource optimization: Architecture. 39
2.2 Outsource optimization: communication flow between SE and CE. . . 42
3.1 Difference in signature execution time between standard and new Dilithium. 48
3.2 Comparison between theoretical and real execution time of the algorithm of matrix multiplication. 53
3.3 Difference in signature execution time between std-Dilithium and modified Dilithium with AES acceleration. 55

List of Tables

2.1 Parameters for Dilithium mode 2 33
3.1 Microcontrolles Overview 44
3.2 Comparison of memory consumption between the standard and new versions of Dilithium in KB. 45
3.3 Performance test of Key generation and Verification procedures for standard and new implementations of Dilithium 47
3.4 Performance test of Signature procedure for std-Dilithium and new-Dilithium 48
3.5 Memory consumption of different DS2 variants. 50
3.6 Comparison between theoretical execution time of the slow and fast algorithms of matrix multiplication. 52
3.7 Comparison between theoretical and real execution time of the slow algorithm of matrix multiplication. 52
3.8 Performance test of Signature procedure with AES acceleration for std-Dilithium and new-Dilithium 54
3.9 Comparison of memory consumption for std-Dilithium and new-Dilithium 55

Listings

Introduction

In recent years, we can see a rapid development of the quantum computers. This technology has unlimited potential for application across various fields and the further development of mankind. However, it also carries risks. Currently, at the time of writing the work, most of the used asymmetric cryptography algorithms for key exchange, encryption, and digital signature are based on Discrete Logarithm Problem (DLP) and Integer Factorization (IF) problem. At the same time, algorithms for quantum computers, such as Shor's algorithm [1] have already been developed, capable of quickly solving these mathematical problems. This means that the further development of quantum computing threatens almost all asymmetric cryptography used at the moment. Therefore, leading experts in the field of information security and mathematics are actively developing new cryptographic primitives based on other computational hard problems, which are theoretically proven to be be equally difficult for both classical and quantum computers. The most promising and most developed are problems based on algebraic lattices and Learning With Errors (LWE) problem. In 2017, as response to NIST call [2], was submitted "Cryptographic Suite for Algebraic Lattices" (CRYSTALS) that encompasses two cryptographic primitives based on Module-LWE problem: Kyber [5], an key-encapsulation mechanism (KEM); and Dilithium [6], a digital signature algorithm. Later in 2023 both Kyber and Dilithium were stadardized by the NIST [3, 4].

In parallel with the rapid advancement of quantum computing, we are seeing a rapid growth in the Internet of Things (IoT) industry. The number of IoT devices is growing almost exponentially every year, and with it the number of cyber attacks on IoT infrastructure is also growing. According to SAM Seamless Network report [7], more than 1 billion IoT attacks took place in 2021. IoT devices need security no less than conventional computer networks, but almost all post-quantum cryptography algorithms, including those in CRYSTALS suite, require enormous computing power and resources that are often not available to IoT devices. In order to address this gap, work is already underway to optimize these algorithms for microcontrollers. For instance, Botros *et al.* [8] developed one such optimization for the Kyber algorithm on Cortex-M4 microcontrollers. Building upon these ideas, this thesis try to continue the ideas from [8] and apply them to Dilithium and its distributed variant DS2 proposed by Damgaard *et al.* [9].

Aim of the thesis

The thesis is focused on studying the CRYSTALS Dilithium and its distributed variant DS2 n-out-of-n signature schemes and developing their modified versions, applicable for constrained devices such as microcontrollers. The goal is to study in practice basic and fundamental concepts of post-quantum cryptography and implementation techniques for maximizing the performance and efficiency on microcontrollers. The implemented signature schemes will be deployed and tested in a real application scenario.

1 Introduction into Lattice-based cryptography

1.1 Basic definitions

A lattice [10] is an infinite set of points in n-dimensional space with a periodic structure or more formally, given n-linearly independent vectors $b_1, ..., b_n \in \mathbb{R}^n$, the lattice generated by them is the set of vectors

$$L(b_1, ..., b_n) = \{\sum_{i=1}^{n} x_i b_i : x_i \in \mathbb{Z}\} \tag{1.1}$$

The vectors $b_1, ..., b_n$ are known as a basis of the lattice. Equivalently, if we define B as the m × n matrix whose columns are $b_1, b_2, ..., b_n$, then the lattice generated by B is

$$L(B) = L(b_1, ..., b_n) = \{Bx : x \in \mathbb{Z}^n\} \tag{1.2}$$

We say that the rank of the lattice is n and its dimension is m. If n = m, the lattice is called a full-rank lattice [11].

The span of a lattice L(B) is the linear space spanned by its vectors [11]

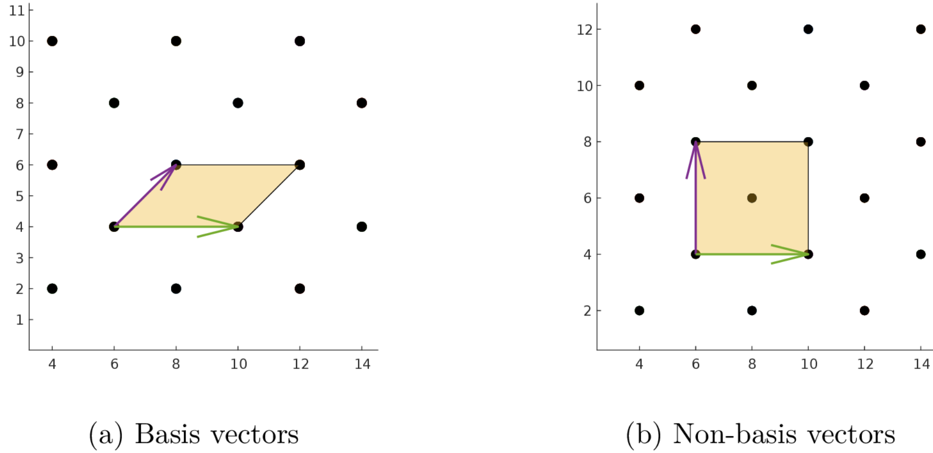
$$span(L(B)) = span(B) = \{By : y \in \mathbb{R}^n\} \tag{1.3}$$

Two lattices have equivalent bases if the basis B_1 of the first lattice can be represented as the multiplication of basis B_2 with a unimodular matrix U: $B_1 = B_2U$. A matrix $U \in \mathbb{Z}^{n\times n}$ is called unimodular if $\det(U) = \pm 1$. Multiplication by a unimodular matrix can also be represented as a permutation or linear combination of the basis vectors $b_1, ..., b_n$ or a combination of both.

A cyclic lattice is a special type of lattice that has a basis matrix B composed of vectors $b_1, ..., b_n$ where each vector equals the previous vector rotated down by 1 element [12, 13]. This property allows us to represent the lattice as a polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, which is used in R-LWE schemes.

Fundamental region is an n-dimentional convex shape, also defined as "fundamental parallelepiped" [11].

$$P(B) = \{Bx : x \in \mathbb{R}^n, \forall i : 0 \le x_i < 1\} \tag{1.4}$$



(a) Basis vectors

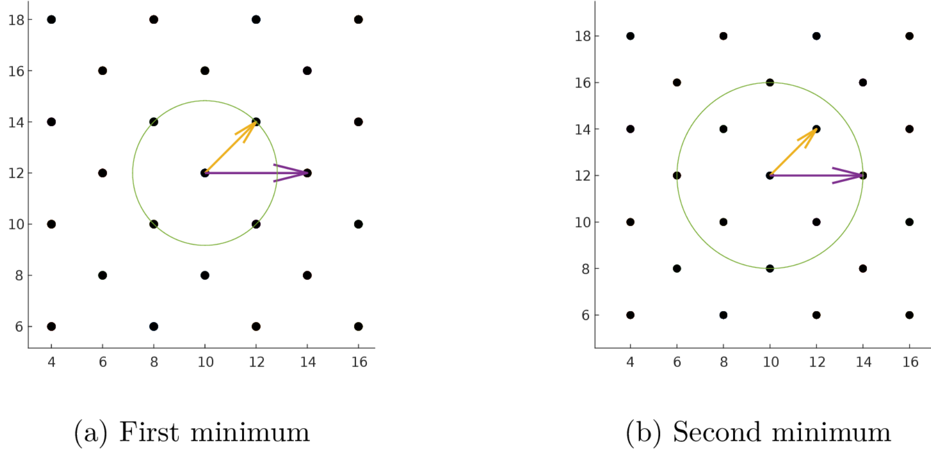
(b) Non-basis vectors

Fig. 1.1: 2D Lattice with it's basis, fundamental region and non-basis vectors

From the definition of the lattice (Equations 1.1, 1.2), it follows that each point of the lattice can be obtained via a linear combination of the basis vectors. The same lattice can be described by several bases, but not any random set of vectors can form a basis. For a given lattice Λ, a set of linearly independent vectors $b_1, ..., b_n$ forms a basis if the fundamental parallelepiped, built from those vectors, does not contain any points of the lattice inside its volume $P(B) \cap \Lambda = \{0\}$. An example of a lattice with its basis and fundamental region is shown in Figure 1.1a. An example of non-basis vectors is shown in Figure 1.1b.

A lattice has three fundamental parameters used in computational problems:

- First is the length of the shortest nonzero vector in the lattice. The length of the vectors is computed as the Euclidean norm $\ell_2 = \|x\|_2 = \sqrt{\sum x_i^2}$.
- The second is the volume of the fundamental parallelepiped, which is equal to the determinant of the lattice. The determinant of the lattice can be calculated with the Gram-Schmidt orthogonalization process.
- The third is the i-th successive minimum. According to Micciancio in [14], the i-th successive minimum $(\lambda_1, ..., \lambda_n)$ of a lattice for every $i = 1, ..., n$ can be defined as the smallest positive real r such that the ball of radius r centered at the origin contains at least i linearly independent vectors $B(0, r) = \{x : \|x\| \leq r\}$. As a special case, the first minimum λ_1 equals the length of the shortest vector (Figure 1.2).



(a) First minimum

(b) Second minimum

Fig. 1.2: Example of the 1 and 2 successive minimums

Some general Lattice problems used in cryptographic primitives are:

- **Shortest-Vector Problem (SVP)**: Given a lattice basis B, to find a shortest nonzero lattice vector $v \in L(B)$ such that $||v|| = \Lambda_1(L(B))$ [15]. This definition can be generalized to γ-approximate SVP problem, asks to find a nonzero lattice vector with euclidean norm at most $||v|| \leq \gamma * \Lambda_1(L(B))$ for $\gamma \geq 1$.
- **Closest-Vector Problem (CVP)**: Given a lattice basis B and target vector t, find the lattice vector $v \in L(B)$ such that the distance to the target $||v - t||$ is minimized [15]. For CVP also exists γ-approximate version, which asks to find vector $v \in L(B)$ such that $||v - t|| \leq \gamma * dist(t, L(B))$ for $\gamma \geq 1$, where $dist(t, L(B))$ is the distance of t to lattice.
- **γ-approximate Shortest In-dependent Vector Problem ($SIVP_\gamma$)**: For $\gamma \geq 1$, given a basis B of an n-dimensional lattice, asks to find linearly independent vectors $v_1, ...v_n \in L(B)$ such that $max_i||v_i|| \leq \gamma * \Lambda_n(L(B))$ [15].

Ajtai [16] shows that the SVP problem is NP-hard for randomized reductions. Van Emde Boas [17] proves that CVP is also NP-hard, and Micciancio [18] shows efficient reductions between SVP, CVP, and SIVP problems. Also, Micciancio and Voulgaris [19] introduce deterministic algorithms to solve CVP and SVP in $n^{O(n)}$ time, and based on this work, Aggarwal *et al.* [20] propose a randomized version of an algorithm for solving SVP, which runs in $2^{O(n)}$ time. At the time of the publication of the thesis, there is no known classical or "quantum" algorithm that can solve SVP, CVP, SIVP, or their variations in polynomial time.

1.2 Learning With Errors

Learning With Errors (LWE) problem was suggested by Regev in [21]. The LWE is a generalized variant of the "learning from parity with error" problem, which asks to find an unknown vector $s \in \mathbb{Z}_2^n$ given a list of equations with errors, such that each equation is correct with probability $1 - \epsilon$:

$$\begin{aligned} \langle s, a_1 \rangle &\approx_\epsilon b_1 \ (mod\ 2) \\ \langle s, a_2 \rangle &\approx_\epsilon b_2 \ (mod\ 2) \\ &\dots \\ \langle s, a_n \rangle &\approx_\epsilon b_n \ (mod\ 2) \end{aligned} \tag{1.5}$$

Where a_i are horizontal vectors and $\langle s, a_i \rangle$ is an inner product of vectors $\sum_j^n s_j * a_{ij}$ modulo 2.

In case of $\epsilon = 0$, System 1.5 is easily solvable by Gaussian elimination algorithm in polynomial time with $O(n)$ equations. However, if $\epsilon > 0$ (error vector e is introduced), then problem become exponentially more difficult because Gaussian elimination uses linear combinations of n equations, and with each step, it amplifies the error to some gigantic values, leaving no information from original equations. In order to eliminate the amplified error, the whole algorithm must be repeated several times, yielding an algorithm that runs in $2^{O(n)}$ time.

Learning With Errors is a generalized version of "learning from parity with error" problem:

$$\begin{aligned} \langle s, a_1 \rangle &\approx_\chi b_1 \ (mod\ p) \\ \langle s, a_2 \rangle &\approx_\chi b_2 \ (mod\ p) \\ &\dots \\ \langle s, a_n \rangle &\approx_\chi b_n \ (mod\ p) \end{aligned} \tag{1.6}$$

Where all calculations are done in the ring modulo some prime integer q, $s \in \mathbb{Z}_q^n$, and error e_i in each equation $b_i = \langle s, a_i \rangle + e_i \ (mod\ q)$ is chosen independently according to probability distribution $\chi : \mathbb{Z}_q \to \mathbb{R}^+$. The maximum likelihood algorithm described for System 1.5 solves System 1.6 in $2^{O(n\ log\ n)}$ time, while best-known algorithm solves LWE in $2^{O(n)}$.

Regev in [21] also shows that an algorithm, which can efficiently solve the LWE problem, if it exists, will be able to efficiently solve the decision version of the SVP and SIVP problems. This can be interpreted as saying that the LWE problem is at least as hard as the decision SVP or SIVP problems if χ is a discrete Gaussian distribution with standard deviation $\sigma = \alpha * q$ for some fixed real number $0 < \alpha < 1$.

1.2.1 Ring Learning With Errors

The main problem of crypto-schemes based on the LWE problem, like the 1-bit encryption scheme from [21], is the size of public and secret keys. To mitigate this problem, the Ring-LWE problem was introduced. From number theory, we know that there exists a canonical embedding from the fractional ideal of the number field to the ideal lattice [22]. Also, we know that every number field can be represented as the field of polynomials in the form $\mathbb{Q}[x]/f_\alpha(x)$, where $f_\alpha(x)$ is a monic irreducible polynomial that has the algebraic integer α as its root. From this follows that the cyclic ideal lattice can be represented as the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ containing all polynomials over the field $\mathbb{Z}_q$.

With all stated above, the Ring-LWE problem can be defined as follows: *Let n be the power of 2 and $q = 1 \ (mod\ 2n)$. Given m samples of the form $\{(a_i,\ b_i = (a_i \cdot s) + e_i),\ i \in [1...m]\}$, find s, where $s \in R_q$ is a fixed secret polynomial, $a_i \in R_q$ is uniformly chosen polynomial, and e_i is an error polynomial chosen independently from spherical Gaussian distribution over R_q.*

Needs to be mentioned that instead of "short" vectors, Ring-LWE problem additionally introduces the concept of a "small" polynomial with respect to the infinity norm. The infinity norm is simply the largest integer coefficient of a polynomial $||f(x)||_\infty = c$.

Thinking about lattice problems in terms of polynomials gives several advantages. The first one is that the embedding of the ring $\mathbb{Z}_q[x]/(x^n + 1)$ to the ideal lattice means that there is a possibility to use it in cyclic form, as was mentioned before. This allows representing all polynomials a_i by a single set of coefficients, which greatly reduces memory consumption and the size of the public keys. The second advantage is that switching to polynomial arithmetic allows replacing the operation of inner product of vectors with polynomial multiplication, for which there exist tested efficient implementations based on Number Theoretic Transform.

The proof of the hardness of Ring-LWE, established in [23] and based on the proof of hardness for LWE from [21], suggests that Ring-LWE is as hard as worst-case SIVP problems on ideal lattices.

1.2.2 Module Learning With Errors

Ring-LWE has great potential for efficient implementation, but the internal structure of the ideal cyclotomic lattice used in its basis can impose some problems. For example, some lattice problems, like the GapSVP problem, which are hard on general lattices, are proved to be potentially easier on a quantum computer [24]. It is not proven that the Ring-LWE problem is solvable, but the fact that it is somewhat easier is still an undesirable side-effect of using ideal lattices. To mitigate potential

risks, in [25], the Module Learning With Errors problem was proposed, which must provide better security than Ring-LWE but still remain more efficient than LWE.

The Module-LWE problem is defined in almost the same way as Ring-LWE. The key difference is that instead of a single ring element, we now work with modules - a set of several elements of the same ring. The number of elements inside the module is the module rank. In this context, the Module-LWE problem can be defined as follows: *Let n be the power of 2 and $q = 1 \ (mod\ 2n)$ and d be the module rank. Given m samples of the form $\{(a_i,\ b_i = \langle a_i \cdot s \rangle + e_i),\ i \in [1...m]\}$, find s, where $s \in (R_q)^d$ is a fixed secret module over ring R_q (vector of polynomials length d), $a_i \in (R_q)^d$ is uniformly chosen module, and e_i is an error module, chosen independently from spherical Gaussian distribution over R_q.* Addition and multiplication of modules $\langle a_i \cdot s \rangle$ is defined in the same way as addition and inner product of the vectors, with respect that elements of those vectors are polynomials.

Module-LWE can be interpreted as a generalized version of the Ring-LWE problem. More specifically, Ring-LWE is a Module-LWE with module rank 1. Module lattices have more complicated algebraic structures than ideal lattices and thus can provide a higher level of security but still be more efficient from a computational perspective than LWE. Thus, Module-LWE can provide the trade-off between speed and security, depending on the chosen parameters.

1.3 Polynomial Arithmetic

1.3.1 Multiplication of polynomials

As stated before, both Ring-LWE and Module-LWE cryptosystems use the multiplication of polynomials in a ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. The simplest algorithm to compute product $a * b = c,\ a, b, c\ \in\ R_q$ is schoolbook multiplication of polynomials [26]:

$$a * b = \left[\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \right] \ mod\ (x^n + 1) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (-1)^{\lfloor \frac{i+j}{n} \rfloor} a_i b_j x^{i+j \ mod\ n} \tag{1.7}$$

All coefficients in Equation 1.7 and all following equations are implicitly reduced modulo q. The problem is that the schoolbook algorithm requires n^2 modular multiplication and $(n - 1)^2$ modular additions and subtractions.

To mitigate this issue, in real-world implementations, polynomials are "converted" to the "frequency" domain using the Number Theoretic Transform (NTT) and multiplied point-wise in only n multiplications [26, 27, 28, 29]. This method exploits the convolution property of Fourier transformation, and technically speaking, NTT is just a Fast Fourier Transform (FFT) defined over the ring $\mathbb{Z}_q$. The key

difference between FFT and NTT lies in their respective definitions of the n-th root of unity. In FFT, the n-th root of unity ω is a complex number $\omega = e^{2\pi i/n}$. In NTT, the n-th root of unity is an integer that corresponds to the following equations:

$$\begin{cases} \omega^n = 1 \ (mod\ q) \\ \omega^m \neq 1 \ (mod\ q), \quad 0 < m < n \end{cases} \tag{1.8}$$

Then NTT transformation is defined as follows: Let a be the polynomial $a = a_{n-1}x^{n-1} + \cdots a_1 x + a_0$. Then NTT transformed polynomial $\bar{a} = NTT(a)$ is defined as

$$\bar{a}_i = \sum_{j=0}^{n-1} a_j w^{ij} \ (mod\ q),\ i = 0, 1 \cdots n-1 \tag{1.9}$$

and the inverse transformation $a = INTT(\bar{a})$ is defined as

$$a_i = n^{-1} \sum_{j=0}^{n-1} \bar{a}_j w^{-ij} \ (mod\ q),\ i = 0, 1 \cdots n-1 \tag{1.10}$$

The only setback of this method is that normally the convolution of polynomials in the NTT domain yields an output length of $2n$, so both polynomials must be padded with zeros, and the result requires additional reduction modulo $(x^n + 1)$. To offset this, usually, *negatively wrapped convolution* is used, which implicitly applies reduction modulo $(x^n + 1)$ to the result. *Negatively wrapped convolution* $c = a \odot b$ is defined as follows:

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \tag{1.11}$$

To avoid the need to pad polynomials with zeros, it is required to compute an additional parameter ϕ such that $\phi^2 = \omega; (mod; q)$. Then all coefficients in polynomials a and b are multiplied by powers of ϕ: $a' = a_{n-1}\phi^{n-1} + ... + a_1\phi + a_0$, $b' = b_{n-1}\phi^{n-1} + ... + b_1\phi + b_0$. The coefficients of the result polynomial c then must be multiplied by the powers of ϕ^{-1}. With all stated above, the multiplication of polynomials is computed as follows:

$$\begin{aligned} c' &= INTT(NTT(a') \odot NTT(b')) \\ c &= c'_{n-1}\phi^{-(n-1)} + ... + c'_1\phi^{-1} + c'_0 \end{aligned} \tag{1.12}$$

The important part is to efficiently implement NTT and INTT transformations. Fortunately, the algorithm for the computation of FFT that runs in $O(n \log n)$ time can also be applied to NTT [26, 27, 28, 29]. Algorithm 1 shows one of the possible implementations of NTT.

Algorithm 1 Fast Iterative Decimation-in-Time Number Theoretic Transform with Cooley-Tukey butterfly described in [26]

Input: $a \in R_q$ of length n in bit-reversed order of coefficients

Output: $NTT(a) \in R_q$ of length n in straight order of coefficients

```
 1: N := n
 2: m := 2
 3: while m ≤ N do
 4:     s := 0
 5:     while s < N do
 6:         for i = 0 to m/2 − 1 do
 7:             N := i · n/m
 8:             k := s + i
 9:             l := s + i + m/2
10:             c := a[k]
11:             d := a[l]
12:             a[k] := c + ω^N d mod q
13:             a[l] := c − ω^N d mod q
14:         end for
15:         s := s + m
16:     end while
17:     m := m · 2
18: end while
19: return a
```

1.3.2 Montgomery multiplication

Almost all programming languages have a built-in operation for the remainder after division, which is generally used to implement modular reduction of the coefficients in polynomials. Unfortunately, the division operation is difficult to implement on hardware, as it is not heavily optimized like multiplication and, most importantly, it is not computed in constant time. If such an operation were conducted only once, then there would be no problems with that. However, as shown in the previous section, multiplication of polynomials in the ring $\mathbb{Z}_q[x]/(x^n+1)$ requires performing a lot of modulo exponentiation operations. This makes the naive approach undesirable. Instead of using built-in operations for multiplication and division remainder in real Ring-LWE implementations, Montgomery's reduction technique is employed to calculate modulo exponentiation [28, 30, 31].

The idea behind Montgomery reduction is to compute the q-residue of the product of two integers whose q-residues are given, and from it, get the classical product itself. In a sense, we temporarily switch the modulus from q to some power of 2, which allows us to replace the expensive division operation by binary shifts to the left. For this, some additional parameters must be precomputed. Namely, we compute $R = 2^k,\ 2^{k-1} <= q < 2^k$. Also, R must be relatively prime to q, but since in our case q is prime, this requirement is automatically satisfied. After that, with the help of the extended Euclidean algorithm, we calculate q' and R^{-1} such that:

$$R \cdot R^{-1} - q \cdot q' = 1 \tag{1.13}$$

The q-residue of an integer x is defined as follows

$$\bar{\text{x}} = x \cdot R\ (mod\ q) \tag{1.14}$$

The Montgomery product of two n-residue values is defined as

$$\bar{\text{z}} = \bar{\text{x}} \cdot \bar{\text{y}} \cdot R^{-1}\ (mod\ q) \tag{1.15}$$

Therefore, the Montgomery modular multiplication algorithm is defined as follows

Algorithm 2 Montgomery Multiplication [30]

Input: $\bar{\text{x}}$, $\bar{\text{y}}$ - integers in q-residue form

1: $t = \bar{\text{x}} \cdot \bar{\text{y}}$
2: $m = t \cdot q'\ (mod\ R)$
3: $u = (t + m \cdot q)/R$
4: if $u \geq q$ then return $u - q$ else return u

1.3.3 Discrete Gaussian Sampling

Sampling from a Gaussian distribution is commonly used to generate random errors in lattice-based cryptosystems. However, implementing this distribution in both hardware and software is challenging. Additionally, it is impossible to use a perfect distribution on any machine running in finite time. Therefore, when developing cryptosystems, one has to use discrete approximations. These approximations should be as close as possible to the real distribution so that security proofs hold. This means that simple rounding of samples from a continuous distribution does not work. The continuous Gaussian distribution is defined as:

$$Pr(X = x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-c)^2/(2\sigma^2)} \tag{1.16}$$

Where X on R is a random variable, $x \in \mathbb{R}$, $c \in \mathbb{R}$ is a center of the distribution.

The discrete form is defined as follows [26] : *Let S be the Gaussian parameter, such that*

$$S = \sum_{k=-\infty}^{\infty} e^{-(k-c)^2/(2\sigma^2)} = 1 + 2\sum_{k=1}^{\infty} e^{-(k-c)^2/(2\sigma^2)} \tag{1.17}$$

Then for random variable X on $\mathbb{Z}$ *and* $x \in \mathbb{Z}$

$$Pr(X = x) = \frac{1}{S}e^{-(x-c)^2/(2\sigma^2)} \tag{1.18}$$

There exists several algorithms for sampling from Discrete Gaussian Distribution. The most common one is *rejection sampling*. The idea is to choose uniformly at random $u \in \{-\tau\sigma, ..., \tau\sigma\}$, where τ is a tail-cut parameter and to accept it with a probability proportional to $e^{-x^2/(2\sigma^2)}$ [26]. Overview of other algorithms used for sampling also can be found in [26].

1.4 Dilithium

Dilithium is a Module-LWE signature algorithm based on the "Fiat-Shamir with Aborts" (FSwA) scheme. It is part of the CRYSTALS (Cryptographic Suite for Algebraic Lattices) suite, which was submitted to NIST's call for post-quantum cryptographic standards. This section is an excerpt from the work [6], where the Dilithium specifications were first described. Documentation and the reference implementation for Dilithium are available at [32].

Key Generation

Algorithm 3 generates matrix A of polynomials over ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. Prime number $q = 2^{23} - 2^{13} + 1$, $n = 256$. This Algorithm samples random secret

Algorithm 3 KeyGen

1: $A \leftarrow R_q^{k \times l}$
2: $(s_1, s_2) \leftarrow S_\eta^l \times S_\eta^k$
3: $t := As_1 + s_2$
4: **return** $(pk = (A, t), sk = (A, t, s_1, s_2))$

Algorithm 4 Sign(sk, M)

1: $z := \perp$
2: **while** $z := \perp$ **do**
3: $\quad y \leftarrow S_{\eta-1}^l$
4: $\quad w_1 := \text{HighBits}(Ay, 2\gamma_2)$
5: $\quad c \in B_{60} := H(M||w_1)$
6: $\quad z := y + cs_1$
7: $\quad$ **if** $||z||_\infty \geq (\gamma_1 - \beta)$ **or** $||\text{LowBits}(Ay - cs_2, 2\gamma_2)||_\infty \geq (\gamma_2 - \beta)$ **then**
8: $\quad\quad z := \perp$
9: $\quad$ **end if**
10: **end while**
11: **return** $\sigma = (z, h, c)$

Algorithm 5 Verify(pk, M, z, c)

1: $w_1' := \text{HighBits}(Az - ct, 2\gamma_2)$
2: **return** $[||z||_\infty < (\gamma_1 - \beta)]$ **and** $[c = H(M||w_1')]$

key vectors s_1 and s_2 with elements from the same ring like in A. Each element of these vectors is an polynomial of ring R_q with small coefficients – of size at most η. All operations on vectors and matrix A are defined over ring R_q.

Signature

Algorithm 4 generates a masking vector of polynomials y with coefficients less than $\gamma 1$. The signer then computes Ay and sets w_1 to be the "high-order" bits of the coefficients in this vector. The output of special hash function c is a polynomial in R_q with exactly 60 ± 1's and the rest 0's in order to get small norm. The potential signature is then computed as $z = y + cs_1$. After that rejection sampling is applied to z in order to remove dependency of z on secret key. The parameter $\beta \leq 60\eta$ is set to be the maximum possible coefficient of cs_i. If any coefficient of z is larger than $\gamma_1 - \beta$ or the low-order bits of $Az - ct$ is greater than $\gamma_2 - \beta$, then we reject z and restart signing procedure. The first check is necessary for security, while the second is necessary for both security and correctness.

Verification

The Verification Algorithm 5 first computes w_1' to be the high-order bits of $Az - ct$, and then accepts if all the coefficients of z are less than $\gamma_1 - \beta$ and if c is the hash of the message and w_1'. This statement is true due to the fact that

$$\begin{aligned} HighBits(Ay, 2\gamma_2) = HighBits(Ay - cs_2, 2\gamma_2) \\ ||LowBits(Ay - cs_2, 2\gamma_2)||_\infty < (\gamma_2 - \beta) \end{aligned} \tag{1.19}$$

And since we know that the coefficients of cs_2 are smaller than β adding cs_2 is not enough to cause any carries by increasing any low-order coefficient to have magnitude at least γ_2. Thus System 1.19 is true and the signature verifies correctly.

1.5 DS2

DS2 is a two-round n-out-of-n signature scheme, proposed by Damgaard *et al.*[9], with low round complexity derived from the Fiat–Shamir with Aborts paradigm [36]. DS2 signature is a distributed variant of the Dilithium signature [6], with its security proof based on the hardness of Module Short Integer Solution (MSIS) and Module Learning with Errors (MLWE) problems. The first practical implementation of the DS2 algorithm was developed by Dobias *et al.* [35]. The scheme was executed on a Linux environment hosted on a Raspberry Pi.

Key Generation

Algorithm 6 `KeyGen`(1^κ)

1: $\rho_n \leftarrow \{0,1\}^{256}$
2: Send out ρ_n and receive $\rho_i; i \in [n-1]$
3: $\rho = \mathrm{H}(\rho_1|\rho_2|...\rho_n)$
4: $\bar{\mathbf{A}} := [\mathbf{A}|\mathbf{I}] \in R_q^{k\times(\ell+k)}$; $\mathbf{A} \leftarrow R_q^{k\times\ell} := \mathtt{Sam}(\rho)$
5: $\mathbf{t}_n := \bar{\mathbf{A}}\mathbf{s}_n; \mathbf{s}_n \leftarrow S_\eta^{\ell+k}$
6: $(\mathbf{t}_{n_1}, \mathbf{t}_{n_0}) := \mathtt{Power2Round}(\mathbf{t_n}, d)$; $g_n := \mathrm{H}_2(\mathbf{t}_{n_1})$
7: Send out g_n and receive $g_i; i \in [n-1]$
8: Send out $\mathbf{t}_{n_1}$, receive $\mathbf{t}_{i_1}$ and check $g_i = \mathrm{H}_2(\mathbf{t}_{i_1}); i \in [n-1]$
9: $\mathbf{t}_1 := \sum_{i\in[n]} \mathbf{t}_{i_1}$
10: $tr \in \{0,1\}^{384} := \mathtt{CRH}(\rho||\mathbf{t}_1)$
11: **return** $pk = (\rho, \mathbf{t}_1),\ sk = (\rho, tr, \mathbf{s}_n, \mathbf{t}_{n_0})$

Algorithms 6, 7 and 8 show all phases of DS2 signature generation and verification as implemented in [35]. The key generation phase (Algorithm 6) allows to generate the

secret key *sk* of each authorized signer and the common public key *pk*. The matrix $\mathbf{A}$ is sampled uniformly from the seed ρ, ensuring that it remains indistinguishable from a uniformly distributed random matrix. In the computation of the combined public key $\mathbf{t}_1$, `power2round` rounding is used to reduce the size of the transmitted data.

Signature

Algorithm 7 $\texttt{Sign}(pk, sk_n, \mu \in M)$

1: $ck := \mathrm{H}_3(\mu, pk)$
2: $\mathbf{w}_n := \bar{\mathbf{A}}\mathbf{y}_n; \mathbf{y}_n \leftarrow D_s^{\ell+k}$
3: $com_n := \texttt{Commit}_{ck}(\mathbf{w}, \mathbf{r_n}); \mathbf{r}_n \leftarrow D(S_r)$
4: Send out com_n and receive $com_i; i \in [n-1]$
5: $c := \mathrm{H}_0(com, \mu, tr); com = \sum_{i\in[n]} com_i$
6: $\mathbf{z}_n := \begin{pmatrix} \mathbf{z}_{n_1} \\ \mathbf{z}_{n_2} \end{pmatrix} := c\mathbf{s}_n + \mathbf{y}_n$
7: Rejection sampling on $(c\mathbf{s}_n, \mathbf{z}_n)$, with probability $min(1, D_s^{l+k}(\mathbf{z}_n)/(M \cdot D_s^{l+k}(\mathbf{z}_{c\mathbf{s}_n,n})))$ continue, otherwise goto 4
8: $\mathbf{z}_{n_2} = \mathbf{z}_{n_2} - c\mathbf{t}_{n_0}$
9: Send out $(\mathbf{z}_n, \mathbf{r}_n)$ and receive $(\mathbf{z}_i, \mathbf{r}_i); i \in [n-1]$
10: **for** $i \in [n-1]$ **do**
11: $\quad \mathbf{w}_i := \bar{\mathbf{A}}\mathbf{z}_i - c\mathbf{t}_{i_1} \cdot 2^d$
12: $\quad$ **if** $||z_i||_2 > B$ or $\texttt{Open}_{ck}(com_i, \mathbf{r}_i, \mathbf{w}_i) \neq 1$ **then** abort
13: $\quad$ **end if**
14: $\quad \mathbf{z} := \sum_{i\in[n]} \mathbf{z}_i; \mathbf{r} := \sum_{i\in[n]} \mathbf{r}_i$
15: **end for**
16: **return** $\Sigma = (c, \mathbf{z}, \mathbf{r})$

During the signing phase (Algorithm 7), n parties collaborate to generate a valid signature. Signature phase just like key generation very similar the one showed in the Algorithm 4. The key difference is an application of the commitment scheme proposed by Damgaard *et al.* [9]. The commitment scheme is designed to ensure that each participant generates the value of vector $\mathbf{w}_n$ based on the public key and random values, and does not select its value based on information received from other participants in such a way as to compromise the security of the signature. The scheme itself involves hiding vector $\mathbf{w}_n$ by adding it to the result of multiplying two large matrices *ck* and *r*. The values of the matrices coefficients are selected in such a way as to preserve the homomorphic property of the proposed scheme, which

allows adding all the commitments from each participant. The sum of each commitment is used to create a challenge polynomial c, which later will allow to check all commitments in single pass at the verification stage (Algorithm 8). But before that, each participant must check "open" commitments from other participants during signature stage. If any participant detects that some commitment opening fails, the signature generation is aborted.

Verification

Algorithm 8 $\texttt{Verify}(pk, \Sigma, \mu \in M)$

1: $ck' := \mathrm{H}_3(\mu, pk)$
2: $\mathbf{w}' := \bar{\mathbf{A}}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
3: $c' := \mathrm{H}_0(\texttt{Commit}_{ck'}(\mathbf{w}', \mathbf{r}), \mu, \texttt{CRH}(pk))$
4: **if** $||\mathbf{z}||_2 \leq \sqrt{n}B$ and $c = c'$ **then return** 1
5: **else return** 0
6: **end if**

In the verification phase, verifier reconstructs **w'**, which, as mentioned before, thanks to homomorphic property by addition of the scheme, represents a sum of all $\mathbf{w}_i$ vectors from each participant. The **w'** vector allows verifier to recreate the sum of commitments and recreate challenge polynomial c, without knowledge of $\mathbf{w}_i$.

1.6 Advanced Encryption Standard

The Advanced Encryption Standard (AES) is an encryption standard established by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting. It is based on a design principle known as a substitution–permutation network and is efficient in both software and hardware. AES has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits [33]. At the time of writing this work, there are already microcontrollers on the market with hardware support for AES, which are of particular interest to us, since they may allow us to compensate for the decrease in performance caused by the changes we made to Dilithium and DS2 algorithms in order to make them work on microcontrollers.

1.7 SHAKE

SHAKE-256 is an extendable-output function (XOF) in the SHA-3 family. Unlike traditional cryptographic hash functions that produce a fixed-length digest, SHAKE-

256 can generate outputs of any desired length. It uses a unique method known as sponge construction, which allows it to absorb any amount of data and squeeze any amount of data [34]. In the context of Dilithium, SHAKE-256 is used as a pseudorandom function with respect to all previous inputs to expand the matrix and the masking vectors, and to sample the secret polynomials with special properties.

1.8 STM32WB55xx description

The STM32WB55 is a multiprotocol wireless low-power microcontroller with an embedded low-power radio compliant with Bluetooth Low Energy 5.3 and IEEE 802.15.4-2011. The controller contains a dedicated Arm Cortex®-M0+ for performing all the real-time low layer operations and an Arm Cortex®-M4 CPU with an FPU, adaptive real-time accelerator, and a frequency of up to 64 MHz. The datasheet with detailed descriptions can be found at [37].

For our purposes, important points include that the STM32WB55 has 256KB SRAM1, consisting of 192KB of actual RAM memory and 64KB of hardware memory parity check. Additionally, the STM32WB55 provides access to a True Random number generator and a hardware accelerator for AES-256, which will be used to speed up and optimize Dilithium later in Section 3.6. Furthermore, its embedded wireless capabilities and hardware security features, such as memory protection, will be useful in implementing use-case demonstrations in the future.

2 Dilithium and DS2 signatures proposed optimization on restricted devises

2.1 Preliminaries

2.1.1 Dilithium Reference implementation overview

As starting point of work, we decided to investigate the reference implementation of the Dilithium from it's creators. The source code is provided in the form of *.zip package, available at link [32]. The structure of the package is shown in the Appendix A. The package contains 4 directories:

- **Supporting documentation**: contains essential explanatory materials.
- **Reference implementation**: the primary implementation in C.
- **KAT**: signature tests results and examples.
- **Additional implementations**: reference implementation, written in combination of C and Assembler x86-64 that uses AVX2 command-set for speed optimization on PC.

For our purposes, we were interested in the pure C implementation. The source files are sorted into sub-packages, each with a self-sufficient copy of the source code of Dilithium, but with different settings and security parameters preinstalled. In this way, the source code can be extracted from the package and compiled without additional efforts on the part of the user. The main difference between sub-packages lies in **config.h** and **params.h** files. In **config.h**, a namespace for all functions is defined, Dilithium mode is specified, and it is determined whether Dilithium uses AES or SHAKE for secret generation and whether a randomized or deterministic signature will be generated. Based on the definitions in **config.h**, **params.h** defines security and computation parameters like dimensions of the matrix and vectors, size of public and secret keys, size of the signature, length of bit-packing, and security parameters η, τ, $\gamma1$, $\gamma2$, β, ω. Remaining files are almost identical across all sub-packages. Due to the nature of constrained devices, from all the available variants, we choose to proceed with Dilithium mode 2 that supports AES and randomized signing. Parameters for Dilithium mode 2 described in Table 2.1.

Therefore, we investigated the source code of the chosen Dilithium sub-package. We found several notable things. First of all, The Dilithium implementation does not rely on external libraries for the implementation of AES or SHAKE. Instead, the authors created their own custom implementations for both algorithms, tailoring them to the specific needs of Dilithium. The only external libraries used are those for random number generation. In this regard, the implementation relies on libraries

Parameter	Value	Description
N	256	Number of coefficients in single polynomial
L	4	Length of vector s_1
K	4	Length of vector s_2
Q	8380417	Modulo of ring
η	2	Maximum possible coefficient in s_1 and s_2
γ_1	2^{17}	Maximum possible coefficient in y
γ_2	$(Q-1)/88$	Maximum possible coefficient in w_0
β	78	Maximum possible coefficient in cs_i
ω	80	Maximum number of 1's in h
τ	39	Security parameter

Table 2.1: Parameters for Dilithium mode 2

provided by the operating system , with support for both Linux and Windows environments. The implementation also includes a table of precomputed powers of the chosen root of unity (ω') already multiplied by powers of the 2-th root of unity (ϕ'). This design choice enables the NTT transformation without requiring the multiplication of input polynomials by the powers of ϕ', as indicated in Equation 1.12. The most notable thing is that matrix A and the vectors of polynomials are allocated on the stack. This implies that microcontrollers must not only have sufficient RAM to store all the data but also that linker scripts need to be edited to ensure the linker allocates enough memory space for the stack. We created a function call map, which can be found inside electronic attachments. This map illustrates the chains of function calls from left to right, providing insights into the approximation of stack memory allocation.

All important variables and data buffers in standard Dilithium are allocated in the stack right after the call to respective functions for key generation, signature, or verification. Memory for those variables is allocated until the exit from the function, even if the variable is used only once. The first workaround for this problem could be the usage of the heap instead of the stack. In this case, memory is allocated only when the variable is needed; after that, memory can be freed. Unfortunately, this approach has two flaws. First, standard C libraries for most microcontrollers do not support heap management. Heap management will require either an RTOS or a custom allocator. The second and more serious issue is that even if we dispose of variables after their usage, still, we would have to allocate a very big chunk of memory in a short time interval.

2.1.2 DS2 Reference implementation overview

After analyzing the code, we came to the conclusion that Dobias *et al.* [35] build their implementation of DS2 upon the same implementation of Dilithium that we described in the Section 2.1.1, only expanding for the possibility of use by several participants at the same time. Therefore, we will omit a detailed description of the code and only refer to the Appendix B for details. Provided implementation of DS2 signature scheme, taking into account the chosen parameters $k = \ell = 2$ and the number of parties $n = 10$, requires a constant allocation of at least 176,256 bytes of SRAM to store data related to parties and resultant signatures. Additionally, it necessitates a minimum of 445,744 bytes of both heap and stack allocated memory for temporary parameters and data buffers. Consequently, a microcontroller would require a minimum of 640 KB of SRAM to effectively execute this DS2 implementation. Moreover, this implementation relies on the transmission of substantial data volumes over the network, amounting to hundreds of kilobytes. Such extensive data transfer could potentially introduce bottlenecks, particularly when utilized in conjunction with low-powered WPAN standards such as 802.15.4. It is worth noting that only a limited selection of microcontrollers available on the market currently meet these memory and networking requirements and the microcontroller we use, STM32WB55RGV6 does not fall into this category. However, after making a number of changes to the implementation we managed to successfully launch and test DS2 scheme on it.

2.2 Reduction of memory consumption

2.2.1 Compression Optimization

This section introduces memory and communication optimizations aimed at enabling the implementation of the DS2 scheme to run on a microcontroller. It is noteworthy that without these optimizations, the signature would not be executable. First of all, we applied the same approach that was previously used for Dilithium. Namely, generation of matrices and vectors one block at a time, where each block is represented by a single polynomial. Note that polynomials are the entries of matrices and vectors, and any calculations can be split to manage “only” two polynomials at once. Therefore, we limit the system to generate the blocks needed at any given moment. Accordingly, the multiplication of a matrix with a vector becomes a block-by-block product of two polynomials at a time. However, this make the computation more time consuming since the same block need to be generated multiple times. Let A and B be $m \times n$ and $n \times h$ matrices, respectively. Algorithm 9 shows the computation

Algorithm 9 $\texttt{Mult}(n, m, h, (Gen_A, s_A), (Gen_B, s_B))$

1: **For** $k \in [0, n-1]$ do
2: **For** $j \in [0, h-1]$ do
3: $B[k][j] := Gen_B(s_B, j, k)$
4: **For** $i \in [0, m-1]$ do
5: $A[i][k] := Gen_A(s_A, i, k)$
6: $Res[i][j] += A[i][k] \cdot B[k][j]$
7: **Return** Res

of the product $Res = A \cdot B$, where Gen_A and Gen_B are the algorithms for generating each matrix with respective seeds s_a and s_b. Note that in case of a matrix-vector multiplication h will be equal to 1 in Algorithm 9. This approach also gives us an additional positive effect in the form that we can transmit random seeds from which these matrices are generated, instead of the matrices themselves. Thus, we reduce the volume of traffic by orders of magnitude and remove the bottleneck,caused by the slow communication protocol 802.15.4.

The aforementioned optimizations are applied as follows:

- **Algorithm 10, Line 2; Algorithm 11, Line 4; Algorithm 13, Line 5 and Algorithm 15, Line 4**: We apply similar method to both Dilithium and DS2. Both pairs ($\mathbf{t_n}$, $\mathbf{w_n}$) are computed by multiplying matrix $\bar{\mathbf{A}} \leftarrow R_q^{k \times \ell + k}$ with a randomly generated secret vector, respectively $\mathbf{s}_n := [\hat{\mathbf{s}}_n | \bar{\mathbf{s}}_n] \leftarrow S_\eta^{\ell+k}$ and $\mathbf{y}_n := [\hat{\mathbf{y}}_n | \bar{\mathbf{y}}_n] \leftarrow D_s^{\ell+k}$. Note that $\bar{\mathbf{A}} := [\mathbf{A}|\mathbf{I}]$, where $\mathbf{A} \leftarrow R_q^{k \times \ell}$ and $\bar{\mathbf{I}}$ is the $k \times k$ identity matrix. In case of Dilithium $\bar{\mathbf{y}}$ is not used, hence it is not generated. Using this approach matrices $\mathbf{A}$, $\mathbf{s}_n$ and $\mathbf{y}_n$ are represented in constant memory size of 1 KB each, independent of k and ℓ.
- **Algorithm 11, Line 7 and 8**: Operations of generation of vectors $\hat{\mathbf{y}}_n, \hat{\mathbf{s}}_n, \bar{\mathbf{s}}_n$, addition, subtraction and multiplication on the challenge polynomial c are done block-by-block.
- **Algorithm 12, Line 1**: Operations of generation of matrix A, subtraction and multiplication on the challenge polynomial c are done block-by-block. Vectors z and $\bar{\mathbf{t}}_n$ are both results of previous calculation and therefore are kept in the memory in the full size.
- **Algorithm 10, Line 3**: Matrix A and vectors $\mathbf{s}_n$ are represented by their

Algorithm 10 KeyGen(1^κ) - Dilithium

1: $\rho, \rho_s \leftarrow \{0,1\}^{256} := \texttt{SHAKE-256}(1^\kappa)$
2: $\mathbf{t_n} := \texttt{Mult}(l, k, 1, (\texttt{SHAKE-256}, \rho), (\texttt{SHAKE-256}, \rho_s)) + \bar{\mathbf{s}}$; $\bar{\mathbf{s}} := \texttt{SHAKE-256}(\rho_s)$
3: **return** $(pk = (\rho, \mathbf{t_n}), sk = (\rho, \rho_s, \mathbf{t_n}))$

Algorithm 11 Sign(sk, M) - Dilithium

1: $z := \perp$
2: **while** $z := \perp$ **do**
3: $\rho_y \leftarrow \{0,1\}^{256} := \texttt{SHAKE-256}(1^\kappa)$
4: $\mathbf{w_n} := \texttt{Mult}(l, k, 1, (\texttt{SHAKE-256}, \rho), (\texttt{SHAKE-256}, \rho_y))$
5: $w_1 := \text{HighBits}(\mathbf{w_n}, 2\gamma_2)$
6: $c \in B_{60} := H(M||w_1)$
7: $z := \hat{\mathbf{y}}_n + c\hat{\mathbf{s}}_n$; $\hat{\mathbf{y}}_n := \texttt{SHAKE-256}(\rho_y)$, $\hat{\mathbf{s}}_n := \texttt{SHAKE-256}(\rho_s)$
8: $\mathbf{v} := \mathbf{w_n} - c\bar{\mathbf{s}}_n$; $\bar{\mathbf{s}}_n := \texttt{SHAKE-256}(\rho_s)$
9: **if** $||z||_\infty \geq (\gamma_1 - \beta)$ **or** $||\text{LowBits}(\mathbf{v}, 2\gamma_2)||_\infty \geq (\gamma_2 - \beta)$ **then**
10: $z := \perp$
11: **end if**
12: **end while**
13: **return** $\sigma = (z, h, c)$

seed ρ, ρ_s. This allowed us to reduce size of the secret key to 2.5KB.

- **Algorithm 15, Lines 1 and 2**: The commitment key ck and the random parameter r_n in the original DS2 scheme occupy at least 138 KB each. Therefore, we represent these parameters by their respective seed values ρ_{r_n} and ρ_{ck}, which are 64 bytes each. The ρ_{ck} is the short hash derived from the message and the parameter tr.
- **Algorithm 14 and Algorithm 15, Line 5**: The Commitment key ck and the random parameter $\mathbf{r}_n$ are generated from the respective seeds one block at a time. The computation of the commitment is also done block by block. This approach reduces the buffer sizes for both ck and $\mathbf{r}_n$ from 138 KB to 1 KB each. Note that ck is generated uniformly and $\mathbf{r}_n$ is generated normally and, therefore, their generation takes different amount of time.
- **Algorithm 15, Line 20 and Algorithm 16**: The Open algorithm uses the same optimization function as the Commit and follows the same logic of block-by-block computation from the given seeds.
- **Algorithm 15, Lines 9-12**: The challenge polynomial is generated from the seed ρ_c, which is 64 bytes long. The seed is generated from the calculated commitment com in the form of a matrix of polynomials. Instead of sending the

Algorithm 12 Verify(pk, M, z, c) - Dilithium

1: $\mathbf{w}' := Az - c\mathbf{t_n}$; $A := \texttt{SHAKE-256}(\rho)$
2: $w'_1 := \text{HighBits}(\mathbf{w}', 2\gamma_2)$
3: **return** $[||z||_\infty < (\gamma_1 - \beta)]$ **and** $[c = H(M||w'_1)]$

Algorithm 13 `KeyGen`(1^κ) `- DS2`

1: $\rho_n, \rho_{s_n} \leftarrow \{0,1\}^{256} :=$ SHAKE-256(1^κ)
2: Send out ρ_n
3: Receive $\rho_i; i \in [n-1]$
4: $\rho :=$ SHAKE-256$(\rho_1, \rho_2, ..., \rho_n)$
5: $\mathbf{t}_n :=$ Mult$(l, k, 1, ($SHAKE-256$, \rho), ($SHAKE-256$, \rho_{s_n})) + \bar{\mathbf{s}}_n$; $\bar{\mathbf{s}}_n :=$ last k bits of SHAKE-256(ρ_{s_n})
6: $(\mathbf{t}_{n_1}, \mathbf{t}_{n_0}) :=$ Power2Round$(\mathbf{t}_n, d)$
7: $g_n :=$ SHAKE-256$(\mathbf{t}_{n_1})$
8: Send out g_n
9: Receive $g_i; i \in [n-1]$
10: Send out $\mathbf{t}_{n_1}$
11: Receive $\mathbf{t}_{i_1}$ and check $g_i =$ SHAKE-256$(\mathbf{t}_{i_1}); i \in [n-1]$
12: $\mathbf{t}_1 := \sum_{i\in[n]} \mathbf{t}_{i_1}$
13: $tr \in \{0,1\}^{512} :=$ SHAKE-256$(\rho, \mathbf{t}_1)$
14: **return** $pk = (\rho, \mathbf{t}_1, tr)$, $sk_n = (\rho_{s_n}, \mathbf{t}_{n_0})$

whole matrix to the SE node, the seed is sent in a single packet. Accordingly, the SE node can regenerate the challenge polynomial c from the seed.

- **Algorithm 15 - Lines 16-17**: the algorithm sends the seeds ρ_{r_i} instead of the whole matrix r_i, which allows eliminating a bottle neck caused by the slow communication speed (on average it takes 5 s to send 138 KB matrix at 250 Kbps).
- **Algorithm 15, Lines 21 and 22**: The signature contains all received seeds ρ_{r_i} and ρ_c, instead of the resulting parameter r. This allows for a reduction in the signature size from 143 KB to 5 KB. Therefore, the following computation is omitted $\mathbf{r} := \sum_{i\in[n]} \mathbf{r}_i$.

2.2.2 Outsource Optimization

Another optimization technique aimed to further reduce memory footprint and removing the limit of maximal number of participating parties. Our system consists of two parts: a security-critical part and a non-security-critical part. The security-

Algorithm 14 `Commit`$(w_n, \rho_{r_n}, \rho_{ck})$ `- DS2`

1: $com_n :=$ Mult$(68, k, k, ($SHAKE-256$, \rho_{r_n}), ($SHAKE-256$, \rho_{ck}))$
2: $com_n += w_n$
3: **return** com_n

Algorithm 15 $\texttt{Sign}(pk, sk_n, \mu \in M)$ - DS2

1: $\rho_{ck} \in \{0,1\}^{512} := \texttt{SHAKE-256}(\mu, tr)$ ▷ SE/CE
2: $\rho_{r_n}, \rho_{y_n} \leftarrow \{0,1\}^{512} := \texttt{SHAKE-256}(1^\kappa)$ ▷ SE
3: $\mathbf{w}_n := \texttt{Mult}(l, k, 1, (\texttt{SHAKE-256}, \rho), (\texttt{SHAKE-256}, \rho_{y_n})) + \bar{\mathbf{y}}_n;$
4: $\bar{\mathbf{y}}_n :=$ last k bits of $\texttt{SHAKE-256}\rho_{y_n})$ ▷ SE
5: $com_n := \texttt{Commit}(\mathbf{w}_n, \rho_{r_n}, \rho_{ck});$ ▷ SE
6: Send out com_n ▷ SE
7: Receive $com_i; i \in [n-1]$ ▷ CE
8: $com := \sum_{i\in[n]} com_i$ ▷ CE
9: $\rho_c \in \{0,1\}^{512} := \texttt{SHAKE-256}(com, \mu, tr);$ ▷ CE
10: Send out ρ_c ▷ CE
11: Receive ρ_c ▷ SE
12: $c := \texttt{SHAKE-256}(\rho_c)$ ▷ SE/CE
13: $\mathbf{z}_n := \begin{pmatrix} \mathbf{z}_{n_1} \\ \mathbf{z}_{n_2} \end{pmatrix} := c\mathbf{s}_n + \mathbf{y}_n;\ \mathbf{s}_n := \texttt{SHAKE-256}(\rho_{s_n})$ ▷ SE
14: Rejection sampling on $(c\mathbf{s}_n, \mathbf{z}_n)$, with probability $min(1, D_s^{l+k}(\mathbf{z}_n)/(M \cdot D_s^{l+k}(\mathbf{z}_{c\mathbf{s}_n,n})))$ continue, otherwise go to Line 4 ▷ SE
15: $\mathbf{z}_{n_2} = \mathbf{z}_{n_2} - c\mathbf{t}_{n_0}$ ▷ SE
16: Send out $(\mathbf{z}_n, \rho_{r_n})$ ▷ SE
17: Receive $(\mathbf{z}_i, \rho_{r_i}); i \in [n-1]$ ▷ CE
18: **For** $i \in [n-1]$ do
19: $\quad \mathbf{w}_i := \bar{\mathbf{A}}\mathbf{z}_i - c\mathbf{t}_{i_1} \cdot 2^d$ ▷ CE
20: $\quad$ **If** $||z_i||_2 > B$ or $\texttt{Open}(\rho_{ck}, com_i, \rho_{r_i}, \mathbf{w}_i) \neq 1$ **then** abort ▷ CE
21: $\quad \mathbf{z} := \sum_{i\in[n]} \mathbf{z}_i$ ▷ CE
22: **return** $\Sigma = (\rho_c, \mathbf{z}, \rho_{r_1}, \rho_{r_2}, ..., \rho_{r_n})$

critical part, namely the Secure Element (SE), performs operations requiring secret knowledge and must be executed on the microcontroller itself. Conversely, the non-critical part, referred to as the Central Element (CE), contains operations that do not require secret knowledge and can be performed either on another microcontroller or on a host computer connected to the CE.

The original DS2 scheme, proposed by Damgaard *et al.* [9] and implemented by Dobias *et al.* [35], employs a full-mesh topology for communication among parties. While robust, this architecture mandates each party to store additional information about all other parties and the random values generated by them. To mitigate this issue, the CE aggregates data from all parties and partially offloads the nodes, relieving them of the burden to perform non-security-critical complex calculations. Consequently, we effectively change the network topology from "full-mesh" to "star"

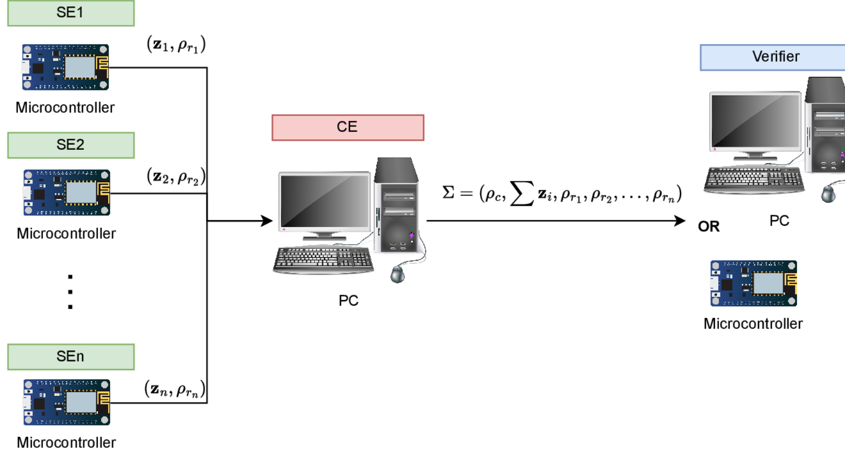


Fig. 2.1: Outsource optimization: Architecture.

configuration.

Figure 2.1 depicts the system architecture with the addition of a CE. Each secure element SE_i computes part of the signature with the help of the CE. Then CE generates the final signature Σ from the partial information. Note that the Verifier is not part of the signing process and can be represented by either a PC or microcontroller. Accordingly, the compression optimizations (see Subsection 2.2.1) are only necessary in the latter case.

Figure 2.2 sketches the communication flow between a SE and the CE. It is worth noting that Algorithm 13 for key generation remains based on "full-mesh" communication. In fact, the introduction of the CE would not release the SE of the memory-demanding computations. Specifically, in Algorithm 13, Line 5 requires knowledge of the secret value s_{ρ_n}, thereby requiring execution within the SE. Moreover, the introduction of a CE and, consequently, a "star" network, may introduce security concerns, as discussed in Section 2.3. On the contrary, in Algorithm 15, the SE is relieved of matrix-matrix multiplications, specifically Lines 4 and 20, that are outsourced to the CE. Finally, Algorithm 17 is solely involved in the verification of the signature.

Algorithm 16 $\texttt{Open}(\rho_{ck}, com_i, \rho_{r_n}, \mathbf{w}_n)$ - DS2

1: $com_n := \texttt{Mult}(69, k, k, (\texttt{SHAKE-256}, \rho_{r_n}), (\texttt{SHAKE-256}, \rho_{ck}))$
2: $com_n += w_n$
3: **If** $com_i == com_n$ **return** 1
4: **Else return** 0

Algorithm 17 `Verify`$(pk, \Sigma, \mu \in M)$ `- DS2`

1: $c :=$ `SHAKE-256`(ρ_c)
2: **For** $i \in [n-1]$ do
3: $\quad \mathbf{r}'_i :=$ `SHAKE-256`$(\rho_{r_i}) \leftarrow D(S_r, \rho_{r_i})$
4: $\quad \mathbf{r}' := \sum_{i \in [n]} \mathbf{r}'_i$
5: $ck :=$ `SHAKE-256`(μ, pk)
6: $\mathbf{w}' := \bar{\mathbf{A}}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
7: $c' :=$ `SHAKE-256`(`Commit`$_{ck'}(\mathbf{w}', \mathbf{r}'), \mu,$ `SHAKE-256`$(\rho, \mathbf{t}_1)$)
8: **If** $||\mathbf{z}||_2 \leq \sqrt{n}B$ and $c = c'$ **return** 1
9: **Else return** 0

2.3 DS2 Security analysis

In order to prove our optimized DS2 scheme is complete and secure, we prove that the `Verify` algorithm is sound and correct, and therefore, only valid signatures generated by authorized parties will be always verified correctly, and invalid signatures will always fail verification. Then, we prove that `KeyGen` and `Sign` algorithms are secure and do not leak any information about the individual parties' secret keys.

Theorem 1 (verification Soundness and Correctness). *The verification process in Algorithm 17 (*`Verify`*) is correct and sound.*

Proof. For the signature to be accepted, we need to show that the newly computed c' is equal to c, that is `SHAKE-256`$(\rho_c) =$ `SHAKE-256`(com, μ, tr) is equal to `SHAKE-256`(`Commit`$_{ck'}(\mathbf{w}', \mathbf{r}'), \mu,$ `SHAKE-256`$(\rho, \mathbf{t}_1)$). If this equality holds, it means that the signatures $\mathbf{z_n}$ were generated by the authorized parties knowing the shares s_n of the common secret key. Since tr is equal to `SHAKE-256` $(\rho, \mathbf{t}_1))$, this can be done by proving that com is equal to `Commit`$_{ck'}$ $(\mathbf{w}', \mathbf{r}')$. Note that

$$\mathbf{r}' = \sum_{i \in [n]} \mathbf{r}'_i = \sum_{i \in [n]} \texttt{SHAKE-256}(\rho_{r_i}) = \sum_{i \in [n]} \mathbf{r_i} = \mathbf{r}$$

Therefore, the commitments equality can be proven as follows:

$$\begin{aligned}
\texttt{Commit}_{ck'}(\mathbf{w}', \mathbf{r}) &= \texttt{Commit}_{ck'}(\bar{\mathbf{A}}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d, \mathbf{r}) \\
&= \texttt{Commit}_{ck'}(\sum_{i \in [n]} \bar{\mathbf{A}}\mathbf{z_i} - c\mathbf{t}_{i1} \cdot 2^d, \sum_{i \in [n]} \mathbf{r}_i) \\
&= \sum_{i \in [n]} \texttt{Commit}_{ck'}(\bar{\mathbf{A}}z_i - c\mathbf{t}_{i1} \cdot 2^d, \mathbf{r_i}) \\
&= \sum_{i \in [n]} \texttt{Commit}(\mathbf{w_i}, \mathbf{r_i}) = \sum_{i \in [n]} com_i = com
\end{aligned}$$

Note that it is straightforward that $||\mathbf{z}||_2 \leq \sqrt{n}B$ since the signers only output a signature shares that respect $||\mathbf{z}_i||_2 \leq B$. We refer to [9], Lemma 2 for more details. □

Theorem 2 (Key Generation Security)**.** *The key generation process in Algorithm 13 (`KeyGen`) is secure.*

Proof. The Algorithm 13 does not leak any information about the individual parties' secret keys as we do not modify the communication model of the original DS2 scheme. In fact, we kept a "full mesh" communication to avoid any possible misbehaving of malicious CE. In the algorithm, ρ represents a committed random value agreed between involved parties. Therefore, outsourcing its computation to a CE could cause a security risk. For example, a malicious CE would be able to pilot the computation of matrix $\bar{\mathbf{A}}$ by sending its own ρ' instead of a fairly computed value $\rho = \texttt{SHAKE-256}(\rho_1, \rho_2, \cdots, \rho_n)$ to each SE. □

Theorem 3 (Signing Security)**.** *The signing process in Algorithm 15 (`Sign`) is secure.*

Proof. It is worth noting that a malicious CE can break the correctness of the verification process in such a way that the signature generated by authorized parties will fail verification. It can be done easily by spoofing the transmitted data between signers. However, the Algorithm 15 does not leak any information about the individual parties' secret keys to any misbehaving CE. In the algorithm, the signer uses the secret key s_n to compute Proof of Knowledge (PK): $\mathbf{z}_n = \begin{pmatrix} \mathbf{z}_{n_1} \\ \mathbf{z}_{n_2} \end{pmatrix} = c\mathbf{s}_n + \mathbf{y}_n$, where $c = \texttt{SHAKE-256}(\rho_c)$, and ρ_c is the challenge computed by CE. This PK does not reveal any information about $\mathbf{s}_n$ even if the CE would have complete control over the challenge c, which it does not have since the output of the hash function $\texttt{SHAKE-256}(\rho_c)$ is unpredictable by CE. □

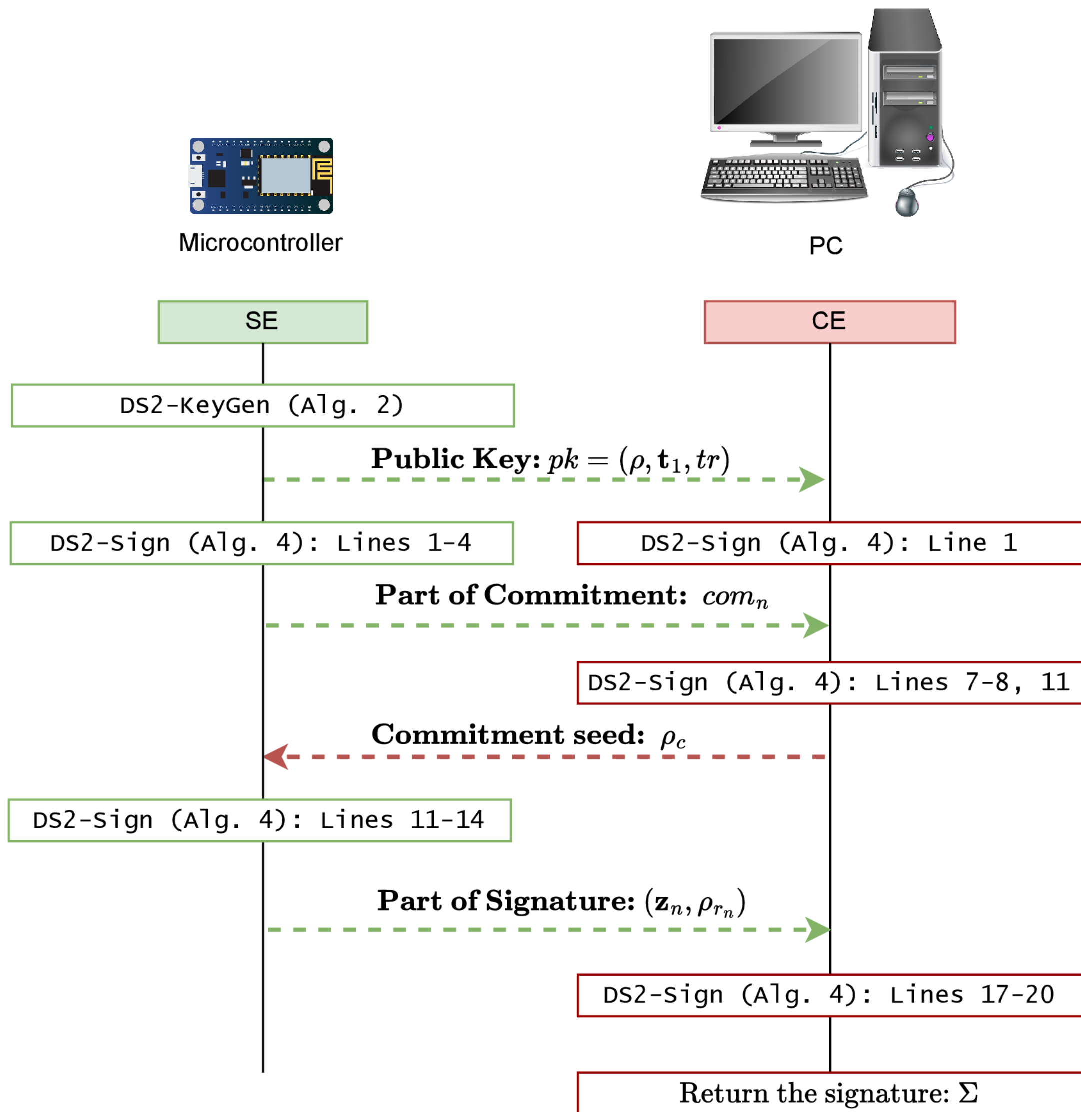


Fig. 2.2: Outsource optimization: communication flow between SE and CE.

3 Implementation of Dilithium and DS2 signatures

In this chapter we present theoretic analysis and test results of our implementation of Dilithium and DS2. Our implementation is open-source and available on the public GitLab repository[1].

3.1 Testing Methodology

From now on, for simplicity, we will reference in the text the unmodified versions of Dilithium and DS2 as **"fast"** or **the "standard"** versions and the modified version as **"slow"** or **"new"** versions. For all tests and test results described below, we used the same MCU STM32WB55RGV6 at 64MHz (interrupts were not disabled). As the metric for performance measurement, the number of CPU cycles was chosen. Logged min and max values of performance measurements for all procedures for both standard and new variants in the worst-case scenario do not deviate more than 5%, thus only max values will be shown in test results. The chosen test message for signature was 256 bytes in length, consisting solely of bytes with a value of 0xAA.

3.1.1 Choice of microcontroller for testing

While RAM size is a critical factor in the choice of microcontrollers, it's not the sole determinant. Important criteria were also the presence of hardware encryption support (AES) and a low-power and low-speed data transfer protocol in order to bring the conditions for using modified protocols closer to real ones. The study identifies a set of high-performance microcontrollers suitable for networking and security applications. More details about the chosen microcontrollers can be found in the Table 3.1, which shows a comparison of microcontrollers' specifications.

Based on the totality of all the above requirements and characteristics form the Table 3.1, we came to the conclusion that the optimal choice would be STM32WB55RGV6 microcontroller, which met all the criteria and was available to us.

3.2 Dilithium memory consumption analysis

As will be demonstrated in the Table 3.2, Dilithium was already quite well optimized for memory consumption in its original implementation and was easy to run on all microcontrollers listed in the Table 3.1. Therefore, in the case of Dilithium, we were

[1] https://gitlab.com/brno-axe/pqc/ds2ram

Table 3.1: Microcontrolles Overview

Name	Developer	Core	SRAM	Protocol	AES	Freq.
STM32WB55xx	STM	Cortex-M4	192KB	802.15.4	+	64MHz
STM32G0C1xx	STM	Cortex-M0+	128KB	None	+	128MHz
STM32H563xx	STM	Cortex-M33	640KB	None	+	250MHz
STM32L476xx	STM	Cortex-M4	128KB	None	−	80MHz
ESP32-S2 (S3)	Espressif	Xtensa LX7	340KB	Wi-Fi	+	240MHz
LPC51U68	NXP	Cortex-M0+	96KB	None	−	96MHz

interested in finding out the minimum possible values, which determined our choice to focus on the Dilithium mode-2-AES with deterministic signature implementation in order to obtain the lowest possible memory consumption, but still comply with the latest standard ML-DSA [3]. The goal was to make Dilithium applicable to the widest possible number of constrained devices. The majority of the microcontrollers on the market at the time of this thesis publication are not capable of running Dilithium due to a lack of RAM.

In order to estimate how much we can reduce memory footprint, we created a theoretical model of memory consumption for the standard versions of the key generation, signature and verification procedures, expressed in the Equations 3.1, 3.2 and 3.3

$$M^{fast}_{KeyGen}(k,\ell) = 4N \cdot (k \cdot \ell + 2\ell + 3k) + M_{app} \tag{3.1}$$

$$M^{fast}_{Sign}(k,\ell) = 4N \cdot (k \cdot \ell + 2\ell + 5k + 1) + M_{app} \tag{3.2}$$

$$M^{fast}_{Verify}(k,\ell) = 4N \cdot (k \cdot \ell + \ell + 3k + 1) + M_{app} \tag{3.3}$$

Where:

- k, ℓ - dimensions of the vectors and matrix A
- N - size of the polynomial (constant)
- $M^{fast}_{KeyGen}, M^{fast}_{Sign}, M^{fast}_{Verify}$ - functions which returns expected memory consumption in bytes
- M_{app} - A variable representing memory reserved for the main application. We approximated $M_{app} \approx 5KB$ for our test application.

Also, we created similar model for the modified version of the Dilithium, which is expressed in the Equations 3.4, 3.5 and 3.6

$$M_{KeyGen}^{slow}(k,\ell) = 4N(2k+4) + M_{app} \tag{3.4}$$
$$M_{Sign}^{slow}(k,\ell) = 4N(\ell+4k+3) + M_{app} \tag{3.5}$$
$$M_{Verify}^{slow}(k,\ell) = 4N(2k+4) + M_{app} \tag{3.6}$$

As can be seen from Equations 3.1 - 3.6, we were able to transform the memory consumption functions from quadratic to linear. To verify these results, we created a modified version of Dilithium according to the description above and measured actual memory footprint, using "Build Analyzer" tool from CubeIDE. We compared real memory consumption for both standarad and new versions of Dilithium with our theoretical estimations based on Equations 3.1 - 3.6 in the Table 3.2.

It is necessary to clarify that our theoretical model is extremely simplified and is intended to obtain a primary rough estimate of memory consumption. Therefore, it does not take into account auxiliary variables and memory buffers that can be used in the program. However, in our case, deviations from real measurements do not exceed 10% and will decrease with increasing parameters k, ℓ. Nevertheless, it can be seen that on average, we could decrease the memory consumption of each procedure by 55%, bringing real memory consumption in the case of key generation to 18KB and in the case of verification to only 16KB.

Procedure	Theor. (std)	Theor. (new)	Real (std)	Real (new)	$\Delta_{std/new}$
M_{KeyGen}	41KB	17KB	42KB	18KB	57.1%
M_{Sign}	50KB	28KB	55KB	29KB	47.3%
M_{Verify}	38KB	17KB	40KB	16KB	60%

Table 3.2: Comparison of memory consumption between the standard and new versions of Dilithium in KB.

3.3 Dilithium time complexity analysis

In order to assess the impact of our changes on performance, we again resorted to constructing a theoretical model of time complexity, which can be expressed in the Equations 3.7, 3.8 and 3.9 for the standard Dilithium variant and Equations 3.10, 3.11 and 3.12 for the new one.

$$C_{KeyGen}^{fast}(k,\ell) = O(t_H + k\ell(t_A + t_{mul}) + t_s(k+\ell) + 2t_{NTT} + 3t_{add} \tag{3.7}$$

$$C^{fast}_{Sign}(k,\ell,n) = O(t_H + t_{NTT}(2k+\ell) + t_s k + t_{add}\ell + t_A k\ell + \\ + n(t_y\ell + t_{mul}(k\ell + 2k + \ell) + 10t_{add}(k+\ell) + t_{NTT}(3k+\ell+1))) \quad (3.8)$$

$$C^{fast}_{Verify}(k,\ell) = O(3t_H + k\ell t_A + t_{NTT}(2k+1) + kt_{mul} + 4t_{add} \quad (3.9)$$

$$C^{slow}_{KeyGen}(k,\ell) = O(t_H + k(\ell(t_A + t_{mul} + t_s + t_{NTT} + t_{mul} + t_a add) + 2t_{add} + t_{NTT} + t_s) + 2t_{add}) \quad (3.10)$$

$$C^{slow}_{Sign}(k,\ell,n) = O(t_H + t_{NTT}k + n(k(\ell(t_A + t_y + t_{NTT} + t_{mul} + t_{add}) + t_{add} + t_{NTT}) + 8t_{add} + \\ + t_{NTT}(k+1) + t_H + \ell(t_s + 2t_{NTT} + t_y + t_{mul} + 2t_{add}) + k(t_s + 2t_{NTT} + t_{mul})) + kt_{mul}) \quad (3.11)$$

$$C^{slow}_{Verify}(k,\ell) = O(3t_H + t_{NTT} + k(\ell(t_A + t_{NTT} + t_{mul} + t_{add}) + 3t_{add} + 2t_{NTT} + t_{mul}) + 3t_{add}) \quad (3.12)$$

Where:

- k, ℓ - dimensions of the vectors and matrix A.
- n - Number of iterations needed to generate a signature.
- t_{mul} - time needed to point-wise multiple two polynomials.
- t_{add} - time needed to point-wise add or substract two polynomials.
- t_s - time needed to generate a single polynomial from s_1, s_2 secret vectors.
- t_A - time needed to generate a single polynomial from A matrix.
- t_y - time needed to generate a single polynomial from y vector.
- t_{NTT} - time needed to perform NTT transformation on single polynomial.
- t_H - average time needed to perform a hashing of 1024 bit of data.
- $C_{KeyGen}, C_{Sign}, C_{Verify}$ - functions which returns expected execution time of the procedure in CPU cycles.

We performed multiple tests according to the procedure described in Section 3.1 and found out the average values of the constants, namely t_{mult} =4E+4, t_{add} =5E+3, t_s =3.84E+5, t_A =1.14E+6, t_y =9.02E+5, t_{NTT} =1.8E+5, t_H =3E+5 CPU cycles. If we plug these values into the equations, then the average execution time for the key generation procedure of the standard Dilithium is **2.26E+7** CPU cycles and for our

implementation, it is **3.06E+7**. For the verification, it is **2.26E+7** and **2.46E+7** CPU cycles for the standard and our implementation respectively. The final results of our empirical measurements for key generation and verification at 64MHz are shown in Table 3.3. It can be seen that our real implementation performance is 11% worse than predicted by our model for key generation and 9% worse for the verification.

Procedure	C_{Real}(fast)	C_{Real}(slow)	$\frac{C_{slow}}{C_{fast}}$
KeyGen	2.55E+7 (399ms)	3.2E+7 (500ms)	1.25
Verification	2.41E+7 (377ms)	2.69E+7 (423ms)	1.12

Table 3.3: Performance test of Key generation and Verification procedures for standard and new implementations of Dilithium

Due to possible rejections during the signature process, the execution time of the signature procedure depends not only on the chosen parameters k, ℓ but also on the random parameter n , which represents the number of signature rounds. Therefore, we tested the signature procedure separately with a slightly different testing sequence. We generated 100 different signatures with different key-pairs to obtain statistics on the distribution of the n values. We found out that in 70% of cases, the signature was generated in one or two rounds. For another 15%, n was in the range between 3 and 7 rounds. The results of our test measurements (on the right) along with our theoretical estimations (on the left) are shown in Table 3.4. It can be seen from the table that our model is not exact, but despite this, we believe that it is accurate enough to be used to evaluate the impact of changes we introduced on the execution time of the signature algorithm with different values of k, ℓ and n. For greater clarity, we plotted the dependence of the signature procedure execution time on the number of rounds n on the Figure 3.1 using only real data measurements from the Table 3.4. It is clear, that time complexity of our algorithm is rising much faster compared to the original. For instance, in case $n = 3$, our implementation became 3 times slower than original and 4 times slower for $n = 7$.

We would like to point out that, as mentioned in Section 3.1, all practical measurements given in the Tables 3.2, 3.3 and 3.4 were obtained for the Dilithium mode-2-AES ($k = 4, \ell = 4$) with the software implementation of the AES-256-CTR.

n	C_{Theor}(fast)	C_{Theor}(slow)	$\frac{C_{slow}}{C_{fast}}$	C_{Real}(fast)	C_{Real}(slow)	$\frac{C_{slow}}{C_{fast}}$
1	3.23E+7	4.94E+7	1.54	3.18E+7	5.17E+7	1.63
2	4.08E+7	9.77E+7	2.43	4.14E+7	9.79E+7	2.37
3	4.93E+7	1.46E+7	3.01	4.82E+7	1.47E+8	3.05
4	5.77E+7	1.94E+8	3.43	5.78E+7	1.96E+8	3.38
5	6.62E+7	2.43E+8	3.73	6.59E+7	2.42E+8	3.67
6	7.47E+7	2.91E+8	3.97	7.79E+7	2.83E+8	3.64
7	8.32E+7	3.39E+8	4.17	8.59E+7	3.41E+8	3.97

Table 3.4: Performance test of Signature procedure for std-Dilithium and new-Dilithium

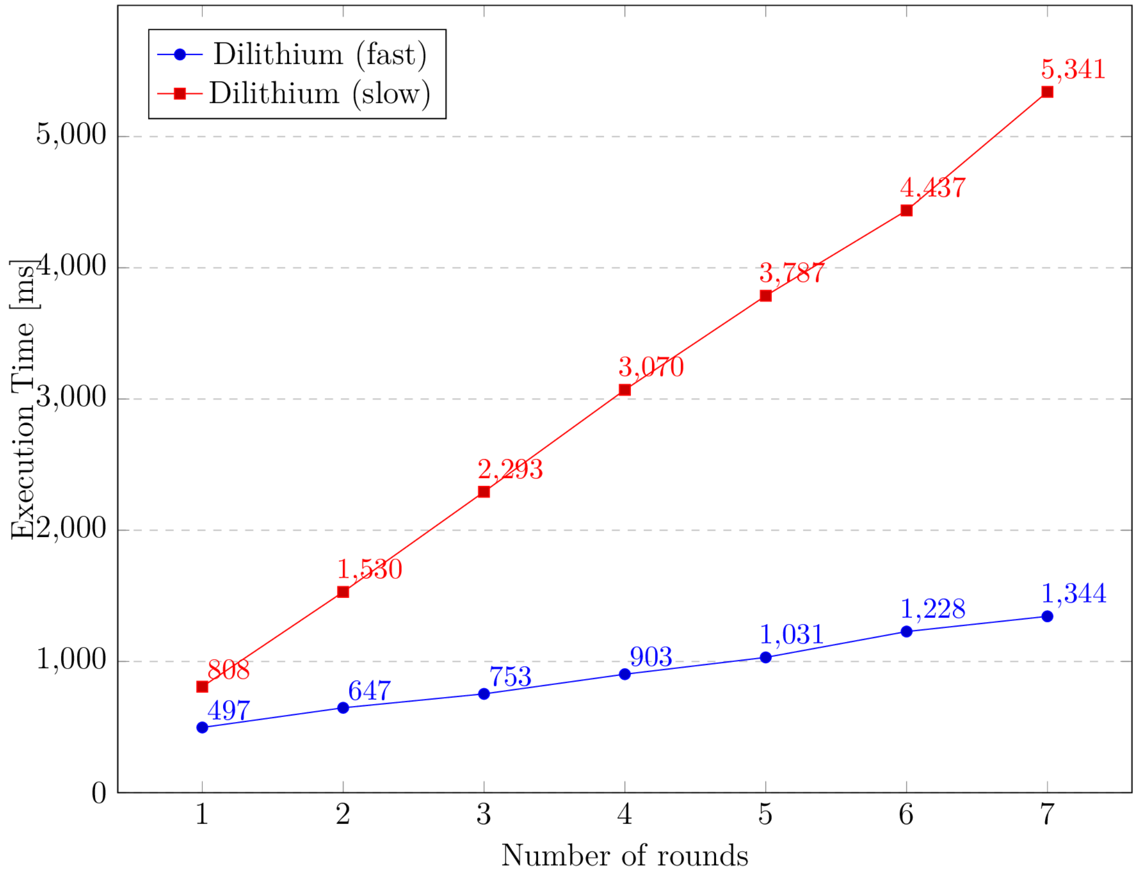


Fig. 3.1: Difference in signature execution time between standard and new Dilithium.

3.4 DS2 Memory consumption analysis

The memory footprint of the original DS2 implementation [35] consists of several parts: the static memory region (M_{static}), which stores data for the public key (M_{pk}) of the participant, secret key (M_{sk}), and the resulting signature (M_{sign}), as well as the stack-allocated memory (M_{stack}) and heap-allocated memory (M_{heap}), which are used for the storage of temporary values during computation. Let k, ℓ, T_c be the dimensions of the matrices, N the number of coefficients in each polynomial, and n the number of parties involved. Therefore, the memory complexity for static parameters without signature can be described by Equation 3.13, for the signature by Equation 3.14, for the stack by Equation 3.15, and for the heap by Equation 3.16.

$$M_{static} = 4N(k(\ell + (n + 2)) + \ell + (n + 2)k^2) \tag{3.13}$$

$$M_{sign} = 4N(k + \ell + kT_c) + s \tag{3.14}$$

$$M_{stack} = 4N((k + 1)T_c + 5\ell + 7k + 1) + 3N(2k + \ell + k^2) \tag{3.15}$$

$$M_{heap} = (k + 1)(3N \cdot T_c) \tag{3.16}$$

If we take into account that the parameters $N = 256$ and $T_c = 69$ are constants, the total memory footprint of the DS2 implementation developed by Dobias *et al.* [35] depends on the chosen security level and the number of parties. Additionally, the largest portion of the memory will be occupied by the commitment key ck, the pseudo random matrix r (which is part of the resulting signature), and additional memory buffers to store the shares of the matrix r, generated by each participant.

Application of our first optimization technique allows us to represent A, s_1, s_2, ck, and r by a single polynomial each, along with random seeds ρ ρ_{s_i}, ρ_{ck} and ρ_{r_i}, where $i \in [1 \cdots n]$. Each seed has a length of s bytes, thus eliminating the dependency of Equations 3.14, 3.15, and 3.16 on the parameter T_c. This, in turn, permits us to completely remove M_{heap} part and reduce the size of resulting signature M_{sign}.

Anyway, each node must store $n - 1$ received commitments to be able to check them at the final stage of signature generation and calculate their sum. As stated before, we address this issue by changing the topology of the local network through the introduction of the central node, namely CE, which is represented by the PC. We split our system into a security-critical part and a non-security-critical part. The security-critical part must always be calculated by the SE nodes, but SE no longer have to store commitments from other nodes. The final note is that all coefficients in each polynomial are taken modulus 8380417, which is a 23-bit number. To further reduce memory consumption and network traffic, vectors and matrices, which are

Table 3.5: Memory consumption of different DS2 variants.

Parameter set	Original DS2	This Work	Mem. Reduction
$k = 4, \ell = 4$	1039 KB	80 KB	92.4%
$k = 6, \ell = 5$	1542 KB	134 KB	91.4%
$k = 8, \ell = 7$	2093 KB	206 KB	90.2%

meant to be sent wirelessly, namely t_1, z_1, z_2, and the *com*, can be "packed" to take only 3 bytes instead of 4 for each coefficient. After applying both optimizations, we obtain Equations 3.17 and 3.18.

$$M_{static} = 4N(\ell + 2k) + 3N(2k + \ell + k^2) + 4s \tag{3.17}$$

$$M_{stack} = 4N(k^2 + 4k + 2\ell + 6) \tag{3.18}$$

Based on the equations above, we estimated the amount of memory required to run DS2: a) without our improvements, b) only with vector multiplication optimization (i.e., Compression Optimization, see Section 2.2.1), c) with multiplication optimization and outsourced computations on the CE node (i.e., Outsource Optimization, see Section 2.2.2). These estimations were calculated for the values $N = 256$, $T_c = 69$, $s = 64$ and for (k, ℓ) dimensions, which correspond to those stated in the ML-DSA standard, namely (4,4), (6,5), (8,7). The introduction of compression optimization reduces memory consumption by an average of 80%. Outsourcing optimization allows for an additional 10% reduction from the original. The total memory reduction after the application of all optimizations is, on average, 90%. The results of our calculations (considering applying both optimisation methods) are shown in Table 3.5.

Additionally, we have to reserve on average 20 KB of static memory space for the main application. Taking into account all of the above, we can confidently say that with all our changes, we are able to run DS2(4,4) and DS2(6,5) variants on the STM32WB55RGV6.

3.5 DS2 Time complexity analysis

First of all, we have to address two main limitations we encountered during our experiments. First, due to the memory limitations of the microcontroller, we could not conduct empirical experiments with DS2(8,7) variant. Second, we chose the number of CPU cycles as the metric for performance measurements in order to obtain frequency independent values. However, the register-counter for the number

of CPU cycles in Cortex-M4 is limited to 32 bits. If k is chosen to be larger than 4, this counter overflows multiple times during large matrix multiplication. Thus, we should rely only on estimations.

The key difference between the original signature algorithm and our modified version lies in the commitment function, see Algorithm 5. The commitment is represented by the multiplication of two large matrices, shown in Algorithm 9. We decided to analyze its impact on the overall performance. For each measurement, we do 10 experiments and find an average of the results. The average total execution time of DS2(4,4) for all operations conducted by the SE node is **4.4E+9** CPU cycles in case the signature was generated in single pass of the algorithm without rejections. At the same time, the average execution time of the matrix multiplication in DS2(4,4) is **4.2E+9** CPU cycles, which is 95% of the total execution time. Considering that the execution time of Algorithm 15 linearly depends on the number of rejections, from all of the above it follows that the analysis of Algorithm 9 will provide us with good approximation about the dependence of the execution time of the entire DS2(k, ℓ) implementation on the selected parameters (k, ℓ), allowing us to significantly simplify the theoretical time ucomplexity model, compared to the model we provided in the Section 3.3. It should be pointed out that, unlike our Dilithium study in section 3.3, we will not examine the dependence of execution time on the number of rejections. This is because statistics from our empirical experiments indicate that in 90% of experiments, the digital signature was generated with 2 or fewer rejections. Instead, we will focus on studying the dependence of execution time only on parameters k, ℓ.

To compare original version (the "Standard") and our optimized version (the "memory-optimized") of the matrix multiplication, we approximated their execution time as shown by Equations 3.19 and 3.20.

$$C_{fast}(k) = O(T_c k(k t_{mul} + t_{ck} + t_r)) \tag{3.19}$$

$$C_{slow}(k) = O(T_c(k(k(t_{ck} + t_{mul}) + t_r))) \tag{3.20}$$

Where:

- C_{fast}, C_{slow} are functions returning the time needed to multiply two matrices with dimensions $k \times T_c$ in CPU cycles.
- $T_c = 69$ is a matrix dimension constant.
- k is a matrix dimension variable.
- t_{ck} is time needed to generate a single polynomial of the commitment key ck from the seed in CPU cycles.
- t_r is time needed to generate a single polynomial of the random matrix r from the seed in CPU cycles.

Table 3.6: Comparison between theoretical execution time of the slow and fast algorithms of matrix multiplication.

k	C_{fast}	C_{fast} [s]	C_{slow}	C_{slow} [s]	C_{slow}/C_{fast}
2	7.54E+8	11.79	1.12E+9	17.52	1.49
3	1.14E+9	17.81	2.24E+9	35.00	1.96
4	1.53E+9	23.93	3.73E+9	58.29	2.44
5	1.93E+9	30.13	5.59E+9	87.40	2.90
6	2.33E+9	36.41	7.83E+9	122.32	3.36

Table 3.7: Comparison between theoretical and real execution time of the slow algorithm of matrix multiplication.

k	C_{real}	C_{real} [s]	C_{teor}	C_{teor} [s]	C_{real}/C_{teor}
2	1.22E+9	19.13	1.12E+9	17.52	1.09
3	2.48E+9	38.74	2.24E+9	35.00	1.11
4	4.18E+9	65.28	3.73E+9	58.29	1.12
5	6.27E+9	98.01	5.59E+9	87.40	1.12
6	8.54E+9	133.41	7.83E+9	122.32	1.09

- t_{mul} is time in CPU cycles needed to multiply two polynomials with N coefficients.

Parameters t_{ck}, t_r and t_{mul} are linearly dependent on the number of coefficients in the polynomial N. Through all test $N = 256$ remained constant. Therefore, t_{ck}, t_r and t_{mul} should also be constants and after our measurements, we obtained on average $t_{ck} = 2.6\text{E+}6$, $t_r = 2.7\text{E+}6$, $t_{mul} = 4\text{E+}4$ CPU cycles. If we substitute those values into Equations 3.19 and 3.20, we can plot those functions on the "k" axis. As can be seen from the values in Table 3.6, the modified matrix multiplication algorithm for the DS2(4,4) variant is 2.44 times slower than the original and is 3.36 times slower for the DS2(6,5) variant.

Finally, we compared our estimations from Table 3.6 with real measurements, obtained from the actual SE device for the memory-optimized version of multiplication. However, as was stated before, we were able to run only the DS2(6,5) version at maximum. Thus, we have no measurements for k greater than 6. The differences between our estimations and real-time values are shown in Table 3.7 and Figure 3.2. It can be seen that the actual implementation is 1.1 times slower than its theoretical model, but this was expected beforehand.

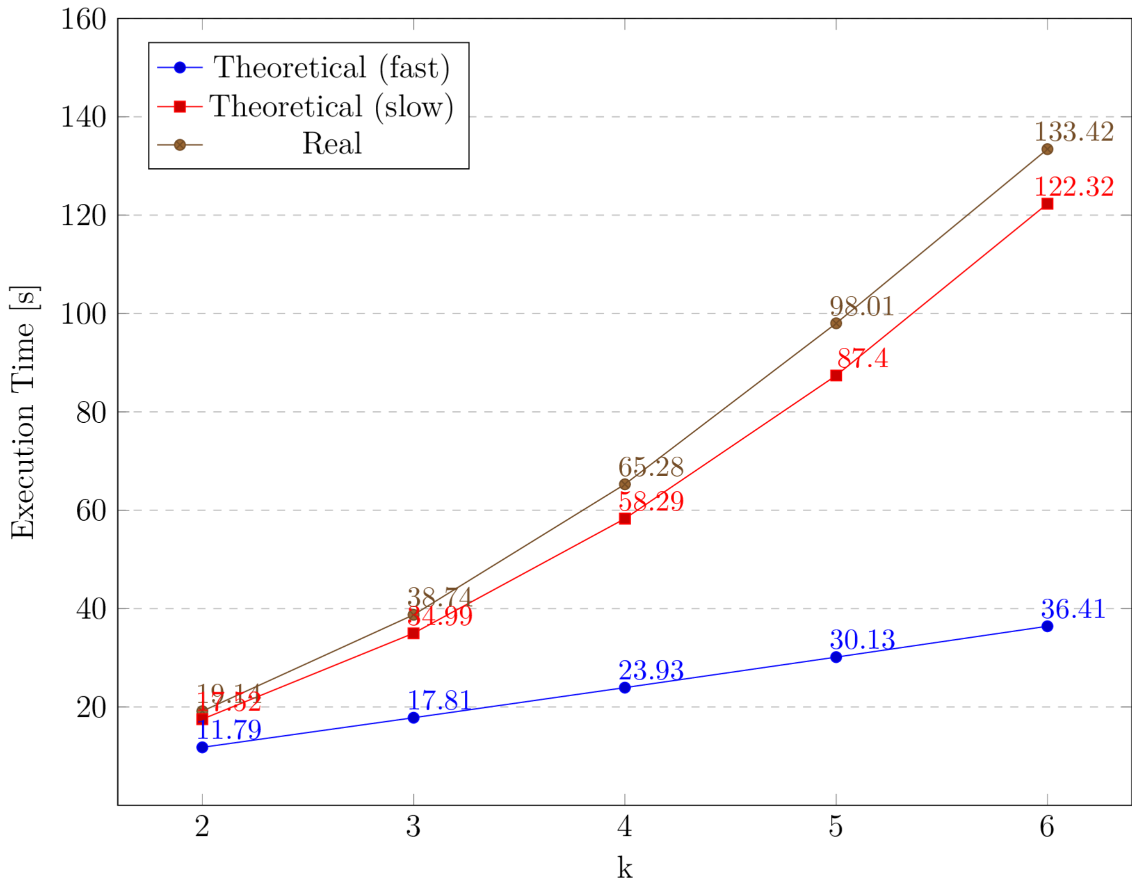


Fig. 3.2: Comparison between theoretical and real execution time of the algorithm of matrix multiplication.

3.6 Addition of hardware support for AES to Dilithium

The most expensive operation in Dilithium on par with inner product of vectors of polynomials is rejection sampling of polynomial coefficients. Rejection sampling is done on pseudo-random output of SHAKE-128/256 hash function. Later developers added support of AES-256-CTR anticipating the possibility of using existing hardware support for AES, including embedded devices. We specifically chose microcontroller with this kind of hardware accelerations. STM32WB55 provides two peripherals AES1 and AES2 that support encryption and decryption in multiple chaining modes: Electronic codebook (ECB), Cipher block chaining (CBC), Counter (CTR), Galois message authentication code (GMAC), Counter with CBC-MAC (CCM). Our modification of the Dilithium implies that in order to reuse some key structures, such as vectors s_1 and s_2, we will have to generate these structures every time unlike the original version where memory was allocated once and any part of the said structures could be accessed at any time. This is the main reason

behind performance degradation shown in the previous section. Using hardware acceleration in this context can significantly improve performance. We added AES acceleration to both std-Dilithium and new-Dilithium and conducted the same test, described in previous Sections 3.1 and 2.2.

We registered a significant reduction in the computation time for both the “slow” and “fast” implementations across all three procedures. For instance, the execution time for the key generation procedure (Algorithm 3) was reduced by **82%** to **4.59E+6** CPU cycles for the original implementation and by **78%** to **7.00E+6** CPU cycles for our implementation. In the case of the verification procedure (Algorithm 5), we achieved a reduction by **74%** (**6.25E+6** CPU cycles) for the standard implementation and by **67%** (**8.90E+6** CPU cycles) for our implementation, respectively. We also observed a significant decrease in time, in absolute values, for the signature procedure (Algorithm 4). Namely, the execution time for the standard implementation was reduced on average by **56%** and for our implementation by **76%** across the full range of feasible values of n. More detailed results for different values of n are given in Table 3.8. In addition, we observed a further reduction in memory consumption since we no longer have to allocate memory to run the software implementation of AES-256. The memory consumption for all procedures, which we registered with the built-in “Build Analyzer” tool from CubeIDE, is provided in Table 3.9.

Sign rounds	std-aes-Sign	$\Delta_{n-aes/aes}$[1]	new-aes-Sign	$\Delta_{n-aes/aes}$[1]	$\Delta_{std/new}$[2]
1	1.09E+7	-66%	1.42E+7	-73%	+23%
2	1.58E+7	-62%	2.59E+7	-74%	+39%
3	2.19E+7	-55%	3.54E+7	-76%	+38%
4	2.68E+7	-54%	4.29E+7	-78%	+38%
5	3.14E+7	-52%	5.67E+7	-77%	+45%
6	3.78E+7	-51%	6.63E+7	-77%	+43%
7	4.13E+7	-52%	7.37E+7	-78%	+44%

Table 3.8: Performance test of Signature procedure with AES acceleration for std-Dilithium and new-Dilithium

[1] Difference between implementations with software and hardware AES.

[2] Difference between std and new implementations with hardware AES.

Procedure	Mem. cost (std-aes)	Mem. cost (new-aes)	$\Delta_{std/new}$
Key Generation	41KB	14KB	-66%
Signing	55KB	26KB	-53%
Verification	39KB	14KB	-64%

Table 3.9: Comparison of memory consumption for std-Dilithium and new-Dilithium

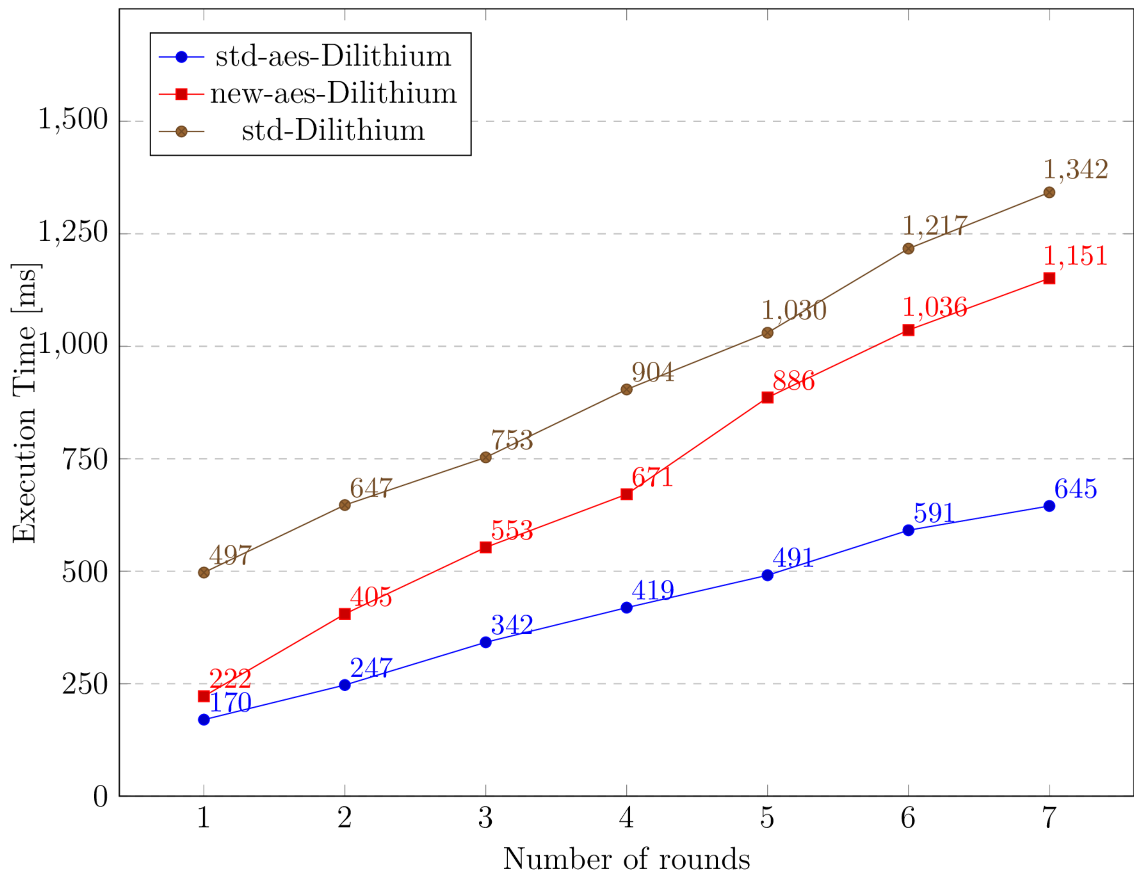


Fig. 3.3: Difference in signature execution time between std-Dilithium and modified Dilithium with AES acceleration.

Conclusion

In this thesis, we analyzed the potential application of the Dilithium and DS2 digital signatures in constrained devices. The main challenge was the large consumption of memory by both the algorithms, especially DS2. Thus, we introduced several optimizations to address this problem. Specifically, we successfully managed to reduce the memory footprint of the original Dilithium algorithm by more than 50% and DS2 by 90%. We achieved this by introducing two optimization methods tailored for microcontrollers with minimal RAM requirements. Firstly, we applied compression techniques for both Dilithium and DS2, enabling the implementation of a DS2 scheme with a (2,2)-threshold on the STM32WB55RGV6 chip. Additionally, for DS2 signature only, we proposed a method based on secure outsourcing of computations outside the microcontroller, effectively reducing memory complexity by 90% and facilitating the extension of the signature to an unlimited number of signers. Our optimization methods highlight the importance of efficient memory utilization for the practical deployment of threshold signature schemes on resource-constrained microcontrollers.

Next, we enumerate the publications that back the contents of this thesis:

- SHAPOVAL, Vladyslav and RICCI, Sara, 2024. Lattice-based Threshold Signature Optimization for RAM Constrained Devices. Online. STUDENT EEICT 2024: Proceedings of the 30th year of the student conference (submitted).
- RICCI, Sara; SHAPOVAL, Vladyslav; DZURENDA, Petr; ROENNE, Peter; OUPICKY, Jan et al. Lattice-based Multisignature Optimization for RAM Constrained Devices. Online. In ARES '24: Proceedings of the 189th International Conference on Availability, Reliability and Security (submitted). Roč. 2024.

Several key points emerge regarding the performance of our optimized matrix multiplication on the Cortex-M4 microcontroller compared to the original implementation. Firstly, unoptimized hash and floating-point calculations are identified as the primary factors contributing to the observed slowdown. If we also consider the high chance that signatures will not be generated from the first attempt, both algorithms can become up to four times slower than the original ones. Obtained test data are indicating that the main contributor to the speed deterioration is the software implementation of the hash function. Although the software implementation of SHAKE and AES are quite fast, given the increased time complexity of both Dilithium and DS2 signature procedures, their speed is insufficient. The introduction of a hardware AES-based hash function reduced signature computation time for our implementation of Dilithium by an average of 76%. We expect similar or better results in the case of DS2.

In conclusion, the reduction of memory footprint must always be supported with hardware acceleration for sampling of polynomials to ensure that the Signature algorithm is usable on constrained devices.

Moreover, optimizing floating-point calculations on microcontrollers emerges as another critical consideration. Notably, Dobias' DS2 implementation heavily relies on floating-point calculations to obtain normally and uniformly sampled values [35]. Unfortunately, floating-point operations exhibit significant slowness on the Cortex-M4, even when supported by a hardware Floating Point Unit (FPU). As a result, the generation of a single polynomial incurs a substantial performance overhead, being ten times slower than the Number Theoretic Transform (NTT) transformation and point-wise multiplication of two polynomials combined.

However, we achieved excellent results in the case of the verification procedure: we were able to reduce the memory consumption of the procedure by 61%, sacrificing only 11% of execution time. For instance, obtained test results suggest that with some adjustments, we can run Dilithium key generation and verification procedures on microcontrollers with just 20KB of SRAM. In the case of DS2, we were able to use the signature procedure on a microcontroller that, in principle, was not capable of doing this with the original version of DS2.

Based on all of the above, we have concluded that, given the potential for improvement in the form of hardware-hash support and optimized floating-point calculations, our implementations of Dilithium and DS2 have real-world application potential.

Bibliography

[1] BERNSTEIN, Daniel J., 2009. Introduction to post-quantum cryptography. Online. Berlin, Heidelberg: Springer. S. 1-14. Available at: https://doi.org/10.1007/978-3-540-88702-7_1.

[2] NIST - COMPUTER SECURITY RESOURCE CENTER (CSRC): Post-Quantum Cryptography - Round 3 Submissions, 2020. Online. 09-November-2020. Available at: https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[3] FIPS 204 (Initial Public Draft), Module-Lattice-Based Digital Signature Standard. Online. 2023. Available at: https://doi.org/10.6028/NIST.FIPS.204.ipd.

[4] FIPS 203 (Initial Public Draft), Module-Lattice-Based Key-Encapsulation Mechanism Standard. Online. 2023. Available at: https://doi.org/10.6028/NIST.FIPS.203.ipd.

[5] BOS, Joppe; DUCAS, Leo; KILTZ, Eike; LEPOINT, T; LYUBASHEVSKY, Vadim et al., 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. Online. IEEE European Symposium on Security and Privacy (EuroS&P). Roč. 2018, s. 353-367. Available at: https://doi.org/10.1109/EuroSP.2018.00032.

[6] DUCAS, Leo; KILTZ, Eike; LEPOINT, Tancrède; LYUBASHEVSKY, Vadim; SCHWABE, Peter et al., 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. Online. IACR Trans. Cryptogr. Hardw. Embed. Syst. Roč. 2018, č. 2018, s. 238-268. Available at: https://doi.org/10.13154/tches.v2018.i1.238-268.

[7] Landscape Security IoT 2021, 2022. Online. Available at: https://securingsam.com/2021-iot-security-landscape/.

[8] BOTROS, Leon; KANNWISCHER, Matthias J. and SCHWABE, Peter, 2019. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. Online. Progress in Cryptology – AFRICACRYPT 2019. Lecture Notes in Computer Science. S. 209-228. Available at: https://doi.org/10.1007/978-3-030-23696-0_11.

[9] DAMGÅRD, Ivan; ORLANDI, Claudio; TAKAHASHI, Akira and TIBOUCHI, Mehdi, 2021. Two-Round n-out-of-n and Multi-signatures and Trapdoor Commitment from Lattices. Online. Public-Key Cryptography – PKC 2021. Lecture Notes in Computer Science. S. 99-130. Available at: https://doi.org/10.1007/978-3-030-75245-3_5.

[10] BERNSTEIN, Daniel J. and LANGE, Tanja, 2017. Post-quantum cryptography. Online. Nature. Roč. 549, č. 7671, s. 188-194. Available at: https://doi.org/10.1038/nature23461.

[11] Lattices in Computer Science. Online. REGEV, Oded. Lecture notes of a course given in Tel Aviv University. 2009. Available at: https://cims.nyu.edu/~regev/teaching/lattices_fall_2009/. [cit. 2023-11-07].

[12] REGEV, Oded, 2010. The Learning with Errors Problem (Invited Survey). Online. 2010 IEEE 25th Annual Conference on Computational Complexity. S. 191-204. Available at: https://doi.org/10.1109/CCC.2010.26.

[13] MOHSEN, Ayman Wagih; BAHAA-ELDIN, Ayman M. and SOBH, Mohamed Ali, 2017. Lattice-based cryptography. Online. 2017 12th International Conference on Computer Engineering and Systems (ICCES). S. 462-467. Available at: https://doi.org/10.1109/ICCES.2017.8275352.

[14] MICCIANCIO, Daniele, 2011. The Geometry of Lattice Cryptography. Online. Foundations of Security Analysis and Design VI. FOSAD 2011. Lecture Notes in Computer Science. Roč. 2011, č. vol 6858, s. 185-210. Available at: https://doi.org/https://doi.org/10.1007/978-3-642-23082-0_7.

[15] CSE206A: Lattices Algorithms and Applications (Spring 2014), 2014. Online. MICCIANCIO, Daniele. UCSD CSE - University of California San Diego. Available at: https://cseweb.ucsd.edu//classes/sp14/cse206A-a/.

[16] AJTAI, Miklós, 1998. The shortest vector problem in L 2 is NP -hard for randomized reductions (extended abstract). Online. Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98. S. 10-19. Available at: https://doi.org/10.1145/276698.276705.

[17] VAN EMDE BOAS, Peter, 1981. Another NP-complete problem and the complexity of computing short vectors in a lattice. Tecnical Report, Department of Mathematics, University of Amsterdam Č. 04.

[18] MICCIANCIO, Daniele, 2007. Efficient reductions among lattice problems. Online. Proceedings of SODA. Roč. 2008, s. 84–93. Available at: https://cseweb.ucsd.edu/~daniele/papers/SIVP-CVP.pdf.

[19] MICCIANCIO, Daniele and VOULGARIS, Panagiotis, 2013. A Deterministic Single Exponential Time Algorithm for Most Lattice Problems Based on Voronoi Cell Computations. Online. SIAM Journal on Computing. Roč. 42, č. 3, s. 1364-1391. Available at: https://doi.org/10.1137/100811970.

[20] AGGARWAL, Divesh; DADUSH, Daniel; REGEV, Oded and STEPHENS-DAVIDOWITZ, Noah, 2015. Solving the Shortest Vector Problem in 2 n Time Using Discrete Gaussian Sampling. Online. Proceedings of the forty-seventh annual ACM symposium on Theory of Computing. 2015-06-14, s. 733-742. Available at: https://doi.org/10.1145/2746539.2746606.

[21] REGEV, Oded, 2009 (Preliminary version in 2005). On lattices, learning with errors, random linear codes, and cryptography. Online. Journal of the ACM. Roč. 56, č. 6, s. 1-40. Available at: https://doi.org/10.1145/1568318.1568324.

[22] BALBÁS, David, 2021. The Hardness of LWE and Ring-LWE: A Survey. IMDEA Software Institute Universidad Politécnica de Madrid. S. 1-48. Available at: https://eprint.iacr.org/2021/1358.pdf.

[23] LYUBASHEVSKY, Vadim; PEIKERT, Chris and REGEV, Oded, 2010. On Ideal Lattices and Learning with Errors over Rings. Online. Advances in Cryptology – EUROCRYPT 2010. Lecture Notes in Computer Science. S. 1-23. Available at: https://doi.org/10.1007/978-3-642-13190-5_1.

[24] CRAMER, Ronald; DUCAS, Léo; PEIKERT, Chris and REGEV, Oded, 2016. Recovering Short Generators of Principal Ideals in Cyclotomic Rings. Online. Advances in Cryptology – EUROCRYPT 2016. Lecture Notes in Computer Science. S. 559-585. Available at: https://doi.org/10.1007/978-3-662-49896-5_20.

[25] BRAKERSKI, Zvika; GENTRY, Craig and VAIKUNTANATHAN, Vinod, 2012. (Leveled) fully homomorphic encryption without bootstrapping. Online. Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. 2012-01-08, s. 309-325. Available at: https://doi.org/10.1145/2090236.2090262.

[26] PÖPPELMANN, Thomas, 2015. Efficient Implemmentation Of Ideal Lattice-Based Cryptography. Dissertation. Bochum, Germany: Ruhr-University.

[27] LONGA, Patrick and NAEHRIG, Michael, 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. Online. Cryptology and Network Security. Lecture Notes in Computer Science. S. 124-139. Available at: https://doi.org/10.1007/978-3-319-48965-0_8.

[28] SCOTT, Michael, 2017. A Note on the Implementation of the Number Theoretic Transform. Online. Cryptography and Coding. Lecture Notes in Computer Science. S. 247-258. Available at: https://doi.org/10.1007/978-3-319-71045-7_13.

[29] MERT, Ahmet Can; OZTURK, Erdinc and SAVAS, Erkay, 2019. Design and Implementation of a Fast and Scalable NTT-Based Polynomial Multiplier Architecture. Online. 2019 22nd Euromicro Conference on Digital System Design (DSD). S. 253-260. Available at: https://doi.org/10.1109/DSD.2019.00045.

[30] MCLOONE, M.; MCIVOR, C. and MCCANNY, J.V., 2004. Montgomery modular multiplication architecture for public key cryptosystems. Online. IEEE Workshop onSignal Processing Systems, SIPS 2004. Roč. 2004, s. 349-354. Available at: https://doi.org/10.1109/SIPS.2004.1363075.

[31] KOÇ, Çetin Kaya, 2019. Course materials for CS 154 Computer Architecture. Online. Available at: http://koclab.cs.ucsb.edu/teaching/cs154/docx/. [cit. 2023-11-11].

[32] SCHWABE, Peter, 2021. Dilithium. Online. CRYSTALS: Cryptographic Suite for Algebraic Lattices. Feb 16, 2021. Available at: https://pq-crystals.org/dilithium/index.shtml.

[33] FIPS 197, Advanced Encryption Standard (AES). Online. 2023. Available at: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[34] FIPS 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Online. 2023. Available at: https://doi.org/10.6028/NIST.FIPS.202.

[35] DOBIAS, Patrik; RICCI, Sara; DZURENDA, Petr; MALINA, Lukas and SNETKOV, Nikita, 2023. Lattice-Based Threshold Signature Implementation for Constrained Devices. Online. Proceedings of the 20th International Conference on Security and Cryptography. 2023-7-10, s. 724-730. Available at: https://doi.org/10.5220/0012112700003555.

[36] LYUBASHEVSKY, Vadim, 2009. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. Online. Advances in Cryptology – ASIACRYPT 2009. Lecture Notes in Computer Science. Pp. 598-616. Available at: https://doi.org/10.1007/978-3-642-10366-7_35.

[37] STMICROELECTRONICS, 2022. Datasheet - STM32WB55xx STM32WB35xx. Online. .: STMicroelectronics. Available at: https://www.st.com/en/microcontrollers-microprocessors/stm32wb55rg.html.

Symbols and abbreviations

AES Advanced Encryption Standard

LWE Learning With Errors

SVP Shortest Vector Problem

CVP Closest Vector Problem

SIVP Shortest In-dependent Vector Problem

NTT Number-Theoretic Transform

FFT Fast Fourier Transform

FSwA Fiat–Shamir with Aborts

std-Dilithium Standard implememntation of Dilithium mode 2, with software support of SHAKE-256 and AES-256-CTR and deterministic signature.

std-aes-Dilithium Standard implememntation of Dilithium mode 2, in which software SHAKE-256 and AES-256-CTR were replaced be hardware AES-256-CTR accelerator.

new-Dilithium Modified version of Dilithium with reduced memory consumption, with software support of SHAKE-256 and AES-256-CTR and deterministic signature.

new-aes-Dilithium Modified version of Dilithium with reduced memory consumption, in which software SHAKE-256 and AES-256-CTR were replaced be hardware AES-256-CTR accelerator.

$\Delta_{std/new}$ Difference between number of CPU cycles needed to generate a signature in std-Dilithium and new-Dilithium in percent for a given number of sign rounds.

$\Delta_{n-aes/aes}$ Difference between number of CPU cycles needed to generate a signature in Dilithium implementation without support of AES hardware acceleration and with support of AES acceleration new-Dilithium in percent, for a given number of sign rounds.

List of appendices

A Content of the Dilithium reference implementation package **65**

B Content of the DS2 reference implementation package **67**

C Content of the electronic attachment **68**

A Content of the Dilithium reference implementation package

```
/dilithium/ ............................................. root of the package
├── Additional_Implementations ...................... Implementation in AVX2
├── KAT ................................................ Signature examples
├── Supporting_Documantation
└── Reference_Implementation ........................ Source code written on C
    └── crypto_sign
        ├── dilithium2
        ├── dilithium2-AES
        └── dilithium2-AES-R .................. Chosen Source code for Dilithium
            ├── inc
            │   ├── aes256ctr.h ....... Definition of the software AES implementation
            │   ├── api.h ................................ Defines high level API
            │   ├── config.h ...................... Definition of Dilithium mode
            │   ├── dilithium_keys.h .......................... Precomputed keys
            │   ├── elapsed_time.h ........ Declaration of time measurement function
            │   ├── fips202.h ..................... Defines interface for SHAKE256
            │   ├── ntt.h ..................................... NTT implementation
            │   ├── packing.h ................ Bit packing of public and secret keys
            │   ├── params.h ......................... Defines Dilithium parameters
            │   ├── poly.h ...................... Definition of polynomial operation
            │   ├── polyvec.h ................... Definition of operations on vectors
            │   ├── randombytes.h ..................... Random Number Generator
            │   ├── reduce.h .............................. Montgomery reduction
            │   ├── rounding.h ........................ Special function definition
            │   ├── sign.h .......... Definition of KeyGen, Sign and Verify algorithms
            │   ├── symmetric.h
            │   └── symmetric-aes_stm.h
            └── src ............ Contains implementation for the header files from inc
                ├── aes256ctr.c
                ├── const_y.c
                ├── dilithium_keys.c
                ├── elapsed_time.c
                ├── fips202.c
                ├── ntt.c
                ├── packing.c
                ├── poly.c
                ├── polyvec.c
                ├── randombytes.c
                ├── reduce.c
                ├── rounding.c
                ├── sign.c
                └── symmetric-aes.c
```

```
│
│         ├── symmetric-aes_stm.c
│         └── symmetric-shake.c
├── dilithium2-R
├── dilithium3
├── dilithium3-AES
├── dilithium3-AES-R
├── dilithium3-R
├── dilithium5
├── dilithium5-AES
├── dilithium5-AES-R
└── dilithium5-R
```

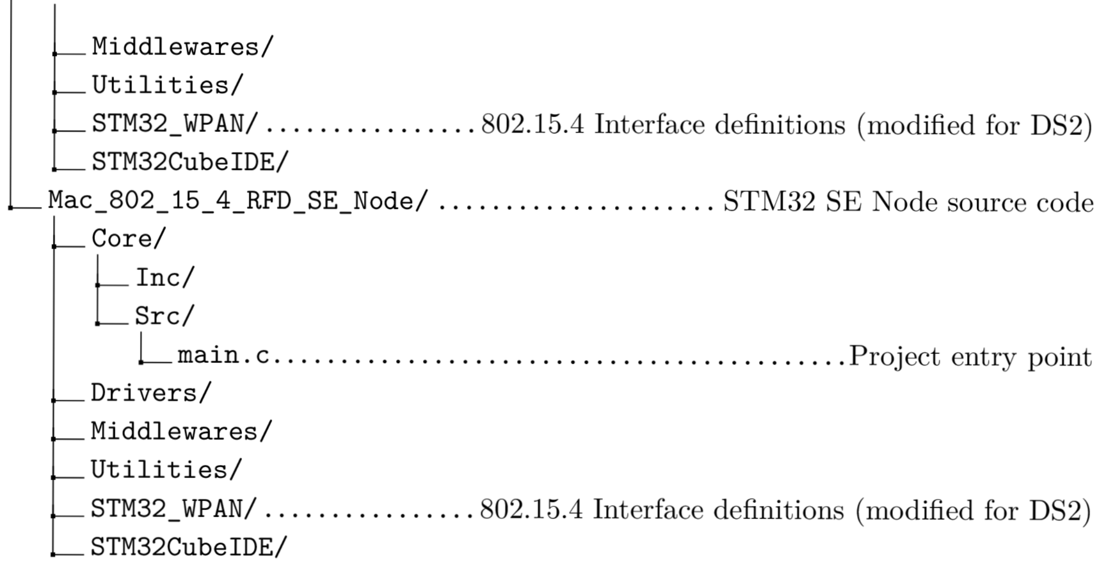
B Content of the DS2 reference implementation package

```
/ds2-opt-c-impl-main/ ...................................... root of the package
├── include/ ............................................... C header files
│   └── ds2
│       ├── benchmark.h
│       ├── commit.h .................... Contains commitment function declaration
│       ├── fips202.h
│       ├── gen.h ................ Contains declaration of Key Generation procedure
│       ├── ntt.h
│       ├── params.h
│       ├── party.h
│       ├── poly.h
│       ├── rand.h
│       ├── reduce.h
│       ├── sign.h ..................... Contains declaration of Signature procedure
│       ├── socket.h
│       ├── util.h
│       └── verify.h ................ Contains declaration of Verification procedure
├── scripts/
│   ├── build-ios.sh
│   ├── run.sh
│   ├── uart_proxy.py
│   └── wifi_proxy.py
├── src/
│   ├── benchmark.c
│   ├── commit.c
│   ├── fips202.c
│   ├── gen.c
│   ├── ntt.c
│   ├── party.c
│   ├── poly.c
│   ├── rand.c
│   ├── reduce.c
│   ├── sign.c
│   ├── socket.c
│   ├── util.c
│   ├── verify.c
│   ├── test_network.c
│   ├── test_file.c
│   └── CMakeLists.txt
├── .gitignore
├── CMakeLists.txt
└── README.md
```

C Content of the electronic attachment

Stabdard Files generated automatically by the CubeIDE in folders "Drivers", "Middlewares" and "USB_Device" will not not described in detail due to their large number.

```
/electronic_attachment/ ........................ root of the package
├── Diagrams/ ... Contains diagrams form describing call maps of Dilithium and DS2
│   ├── Dilithium_Diagram.svg
│   └── DS2_Diagram.svg
├── Dilithium_STM32/
│   ├── Core/
│   │   ├── Inc/
│   │   └── Src/
│   │       └── main.c ........................ Project entry point
│   ├── dilithium/
│   │   └── dilithium2-AES-R/ .... Copied and modified implementation of Dilithium
│   │       from Appendix A
│   ├── Drivers/
│   ├── Middlewares/
│   ├── USB_Device/
│   ├── .cproject
│   ├── .mxproject
│   ├── .project
│   ├── Dilithium_STM32.ioc
│   ├── Dilithium_STM32.launch
│   ├── STM32WB55RGVX_FLASH.ld
│   └── STM32WB55RGVX_RAM.ld
├── DS2_Host/ ........................ PC Host source code
│   ├── externals/
│   │   └── pybind11/ ....... External pybind11 library (not included due to the size)
│   ├── source/
│   │   ├── ds2/ ........................ Modified C++ implementation of DS2
│   │   ├── exec/
│   │   │   └── main.cpp ........................ Test program to simulate multiple nodes
│   │   └── module/
│   │       └── python_wrapper.cpp ........... pybind11 wrapper for DS2 C++ code
│   ├── CMakeLists.txt
│   └── main_app.py ........................ Test application to simulate Hots behaviour
└── Mac_802_15_4_FFD_CE_Node/ ..... STM32 CE Node responsible for local network
    establishment
    ├── Core/
    │   ├── Inc/
    │   └── Src/
    │       └── main.c ........................ Project entry point
    └── Drivers/
```



Middlewares/
Utilities/
STM32_WPAN/ .... 802.15.4 Interface definitions (modified for DS2)
STM32CubeIDE/
Mac_802_15_4_RFD_SE_Node/ .... STM32 SE Node source code
Core/
Inc/
Src/
main.c .... Project entry point
Drivers/
Middlewares/
Utilities/
STM32_WPAN/ .... 802.15.4 Interface definitions (modified for DS2)
STM32CubeIDE/