KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Optimizing Dilithium Using the Helium Vector Extension

Kevin Orbie

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, ei

Thesis supervisor:
Prof. dr. ir. Ingrid Verbauwhede

Assessors:
Prof. dr. ir. Marian Verhelst
Dr. Josep Balasch Masoliver

Mentors:
Ir.Jose Maria Bermudo Mera
Dr. ir. Angshuman Karmakar
Dr. Hanno Becker

Academic year 2021 – 2022

Preface

First and foremost, I would like to thank dr. ir. Angshuman Karmakar, who helped guide me throughout the year, and who was always quick to respond when I needed help. I would also like to extend a special thanks to dr. Hanno Becker, a co-supervisor to this thesis, working as a principal cryptography researcher at Arm, who helped me better understand the implementation platforms used throughout this thesis, and who was of vital importance to get all the Arm hardware correctly up and running. Furthermore, I am also very grateful for their proofreading of this thesis.

I would also like to thank Arm for donating the Arm MPS3 FPGA prototyping board in support of this thesis. Next, I also wish to thank ir. Jose Maria Bermudo Mera for explaining how to get the MPS3 FPGA prototyping board up and running. I also would like to thank Prof. dr. ir. Ingrid Verbauwhede, for helping me find a good research question, and providing me with an interesting thesis topic. I also want to thank Prof. dr. ir. Marian Verhelst for helping me search for an alternative implementation platform, when we thought we would not get access to the cycle-accurate model of the Cortex-M55. Finally, my sincere gratitude also goes to the jury for reading this thesis text.

Kevin Orbie

Contents

Abstract

In this thesis we study the workload of the Dilithium signature scheme, one of the winners of the NIST Post-Quantum Cryptography project, and research whether the Cortex-M55 processor, and the Armv8.1-M ISA that it implements is suited to its execution.

In the process, we find a higher level optimization technique, not constrained to the Cortex-M55 processor, that improves the average Dilithium execution time by 1.18%, 4.41%, and 2.1% for the NIST security level 2, 3, and 5 respectively. We also provide a visual representation of the Dilithium workload, as an alternative to the frequently used algorithmic representation from the Dilithium specification document [19]. Additionally, we also give a detailed explanation of the c-code NTT implementation from the reference CRYSTALS-Dilithium code [49], and the assembly NTT implementation generated using the Python-based code-generation framework also used in [25].

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

PQC	Post-Quantum Cryptography
NIST	National Institute of Standards and Technology
ISA	Instruction Set Architecture
RSA	Rivest-Shamir-Adleman
ECDH	Elliptic-Curve Diffie-Hellman
DES	Data Encryption Standard
AES	Advanced Encryption Standard
SHA	Secure Hash Algorithm
NTT	Number Theoretic Transform
iNTT	Inverse Number Theoretic Transform
CRT	Chinese Remainder Theorem
FFT	Fast Fourier Transform
FNT	Fermat Number Transform
CRC	Cyclic Redundancy Check
FIPS	Federal Information Processing Standards
XOF	eXtendable Output Functions
ML	Machine Learning
DSP	Digital Signal Processing
ILP	Instruction Level Parallelism
DLP	Data Level Parallelism
VLIW	Very Long Instruction Words
MVE	M-profile Vector Extension
SISD	Single Instruction/Single Data
MIMD	Multiple Instruction/Multiple Data
MISD	Multiple Instruction/Single Data
SIMD	Single Instruction/Multiple Data
ALU	Arithmetic Logic Unit

SVE	Scalable Vector Extension
GPRF	General Purpose Register File
VRF	Vector Register File
GPR	General Purpose Register
LOB	Low Overhead Branch
LR	Link Register
AI	Artificial Intelligence
MAC	Multiply- ACcumulate operation
NPU	Neural Processing Unit
GS	Gentileman-Sande
UML	Unified Modeling Language
CNN	Convolutional Neural Network
CMSIS	Common Microcontroller Software Interface Standard
MMU	Memory Management Unit
DWT	Data Watchpoint and Trigger

Symbols

Scalars are denoted with a lower case symbol (a)
Vectors are denoted with an arrow above ($\vec{a}$)
Matrices are denoted with an upper case symbol (A)
Polynomials are denoted with "(X)" or "(x)" after the symbol (a(x))

ζ_k	k-th primitive root
F_t	Fermat's number

Operators

$x \ mod^{\pm} \ \alpha$	Reduce x, to a representative within $[-\frac{\alpha}{2}, \frac{\alpha}{2}[$
$x \ mod^{+} \ \alpha$	Reduce x, to a representative within $[0, \alpha[$
$p(x) \odot q(x)$	Element-wise multiplication between the coefficients of $p(x)$ and $q(x)$
$p(x)q(x)$	Polynomial multiplication between $p(x)$ and $q(x)$
$\lfloor x \rfloor$	Rounding x down to the nearest integer
$\lceil x \rceil$	Rounding x up to the nearest integer
$\lfloor x \rceil$	Rounding x to the nearest integer, can be up or down

Chapter 1

Introduction

The introduction starts with an introduction to Post-Quantum Cryptography (PQC), specifically zooming into a post-quantum signature scheme the National Institute of Standards and Technology (NIST) plans to standardize [36], called CRYSTALS-Dilithium, colloquially known and further simply referred to as Dilithium. It goes on to introduce and explain some of the most important fundamentals supporting Dilithium, the Armv8.1-M Instruction Set Architecture (ISA), and the first processor that implements the Armv8.1-M ISA. Finally giving an overview of related work in literature, describing the goals of this thesis, and presenting the general structure of the remainder of this thesis.

1.1 Preparing cryptography for the dawn on quantum-computing

Quantum computing has received a lot of attention in the last two decades. With companies like Google, declaring quantum supremacy in 2019 [18], and IBM exceeding the 100 qubit mark in 2021 [20].

Using algorithms developed by Peter Shor in 1994 [50], future iterations of quantum computers would be able to more efficiently do integer factorization and solve the discrete logarithm problem [22]. These problems form the basis for the main public-key cryptographic systems currently in use, like the Rivest-Shamir-Adleman (RSA) cryptosystem and the Elliptic-Curve Diffie-Hellman (ECDH) key exchange mechanism. Symmetric cryptography on the other hand, while also threatened, but by Grover's search algorithm [32], is at less of a risk, as doubling the key sizes would remedy this [22]. A similar key sizing is not feasible in the case for public-key cryptography, as Shor's algorithm is polynomial time. Meaning that the time needed for Shor's algorithm to factor an N-digit integer is upper bounded by a polynomial expression in N. Thus, doubling the key size would only result in a power of two improvement, resulting in the need for very large key sizes, as noted in [22] by referring to the multi-GB keys used by the post-quantum RSA proposal, rendering the algorithm impractical.

Current communication security relies on a combination of public-key and symmetric cryptography, where public-key cryptography is used both to establish a common key for symmetric cryptography (Key Establishment Protocol) and to authenticate data in an untrusted environment (digital signatures). With the rise of quantum computing, and the current public-key cryptography under threat, there is a need for new standardised cryptography systems. That is where PQC comes in the picture.

1.1.1 Post Quantum Cryptography

The cryptographic algorithms required to combat the rise in quantum computing, or so called Post Quantum Cryptography (PQC) algorithms, should be able to run on current computer architectures (not quantum computers), and should be resistant to attacks leveraging the power of quantum computing.

That is why in 2016 [40] the National Institute of Standards and Technology (NIST) "*initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms*" [41]. NIST plans to standardize not only Public-Key Encryption but also Digital Signature and Key Establishment Protocol cryptographic algorithms, due to their vulnerability.

In July 2021 NIST announced which PQC candidates moved on to round three [44]. With round three, the competition entered its final phase, expecting the standards to be published by the end of 2024 [40]. With four finalist candidates for the public-key encryption/key-establishment algorithms category: Classic McEliece, CRYSTALS-KYBER, NTRU, and SABER [43] and three for the digital signature algorithm category: CRYSTALS-DILITHIUM, FALCON, and Rainbow [43].

In July 2022 NIST announced which PQC algorithms it already intends to standardize, with other algorithms moving on to the fourth round [36]. Stating that "*NIST will recommend two primary algorithms to be implemented for most use cases: CRYSTALS-KYBER (key-establishment) and CRYSTALS-Dilithium (digital signatures)" and went on to add that "the signature schemes FALCON and SPHINCS+ will also be standardized*" [36].

In the past NIST already successfully standardised some of the most widely used cryptography standards. Examples of which include DES[38], Triple-DES[45], AES[35] and SHA[39]. This means that there is a high likelihood that the PQC algorithms standardised this time, also will see wide adoption in the future.

1.1.2 Dilithium

This paper mainly focuses on CRYSTALS-DILITHIUM, which is referred to as Dilithium throughout the rest of this document. As mentioned in the previous section, Dilithium is one of the three digital signature algorithm finalists.

Dilithium, just like five of the seven finalists, is categorized as structured lattice-based cryptography [22]. This similarity has the benefit that some of the conclusions derived for the case of Dilithium will also be applicable in the broader scope of structured lattice-based cryptography. On top of that, as structured lattice-based

algorithms are so prevalent among the finalists, there is a large chance one algorithm of this type will eventually be standardized.

The Dilithium algorithm consists of three components: key-pair generation, signature generation, and signature verification. A more detailed explanation of Dilithium is given in chapter 2.

1.2 The Fundamentals Supporting Dilithium

To understand the Dilithium workload, it is necessary to have a good grasp of the fundamental functions and constructs that Dilithium utilizes. Therefore, this section expounds the basic working principle, and when relevant, known efficient implementations of multiple mathematical constructs. The following fundamentals are expanded upon in the remainder of this section: modular arithmetic, polynomial multiplication, and hashing.

1.2.1 Modular Arithmetic

The Merriam-Webster defines modular arithmetic as "*arithmetic that deals with whole numbers where the numbers are replaced by their remainders after division by a fixed number*" [29]. The fixed number referenced in the previous definition is often reffered to as the modulus, and the operation of replacing a number by their remainder, or a similar representative, is called a modular reduction.

Modular reductions are thus an essential part of modular arithmetic. Modular reductions of the form $a \ mod \ m$, can be defined as:

$$a - \left\lfloor \frac{a}{m} \right\rfloor * m$$

This way of implementing the modular reduction operation however, requires a division by the modulus. As the modulus is often not a power of two, but rather a prime, as is the case in Dilithium, this division operation is quite expensive to compute. That is why, in the remainder of this section, some alternatives to the classical modular reduction are introduced. These alternatives are often used in cryptography when doing modular arithmetic.

The basic premise of these alternatives is to replace the expensive integer division operation by a cheaper alternative, possibly introducing some approximation error in the process.

Barrett Reduction

In 1987 Barrett showed that the $\lfloor \frac{a}{m} \rfloor$ term can be approximated to avoid the expensive division operation. He achieved this by storing the inverse of m, multiplied by a power of two, to keep working in the integer domain [21]. Following the aforementioned reasoning pattern results in the following approximation of the traditional modular

reduction, as given in [25], called the Barrett Reduction:

$$a - \left\lfloor \frac{a * \left\lfloor \frac{2^l}{m} \right\rfloor}{2^l} \right\rfloor * m$$

In this equation, the $\left\lfloor \frac{2^l}{m} \right\rfloor$ factor can be calculated in advance, and can be reused for all values that need to be reduced modulus m.

Algorithm 1 Barrett Reduction [23],[25]

Input: $a \in [-R, R[$
Input: m *modulus*
Input: $R = 2^l > m$
Input: $m' = \left\lfloor \frac{R}{m} \right\rceil \in \mathbb{Z}$
Output: $t \equiv (a \ mod \ m) \in [-3m/2, 3m/2[$
$k \leftarrow \lfloor (am')/R \rceil$
$t \leftarrow (a - km)$

Montgomery Reduction

In 1984, Montgomery published a technique to efficiently calculate modular reductions [34]. The Montgomery reduction algorithm does not calculate $a \ mod \ m$, but rather $aR^{-1} \ mod \ m$.

As explained in [23], Montgomery reduction can be viewed as a reduction technique in which the target residue is "twisted" by R^{-1}. If the value to be reduced, a, is divisible by R, then this corresponds to a right shift of a by l bits, with $R = 2^l$. If a is not divisible by R, then before dividing by R, a multiple of m is added to a to make sure the sum $(= a + km)$ is divisible by R. The result of the reduction is then again the remaining bits after a right shift by l bits.

Algorithm 2 Montgomery Reduction [34],[23],[25]

Input: $a \in [0, mR[$
Input: m *modulus*
Input: $R = 2^l > m$
Input: $m' = -m^{-1} \ mod \ R$
Output: $t \equiv (aR^{-1} \ mod \ m) \in [0, 2m[$
$k \leftarrow ((a \ mod \ R)m') \ mod \ R$
$t \leftarrow (a + km)/R$

Algorithm 2 can be validated, as done in [34], by observing the following:

- As $km \equiv (am^{-1})m \equiv a(m^{-1}m) \equiv a(R^{-1}R - 1) \equiv -a \ mod \ R$. From $(a + km) \equiv (a - a) \equiv 0 \ mod \ R$, it follows that t is an integer.
- From $tR \equiv (a + km) \equiv a \ mod \ N$, it follows that the result is $t = aR^{-1} \ mod \ m$;

- As $(a + km) \in [0, mR + mR[$, it can be derived that output, t, is contained in the interval $[0, 2m[$. While most of the time, t is smaller then m, sometimes, to get the size of the result in the interval $[0, m[$, it is still required to subtract m from the result.

Montgomery Multiplication

Montgomery reductions can also be used to more efficiently do modular multiplication. By transforming elements to a Montgomery form, i.e. $aR \ mod \ m$, the Montgomery reduction algorithm can be used to calculate the product $abR \ mod \ m$. In what is traditionally called Montgomery multiplication, $abR \ mod \ m$ is than calculated as $((aR \ mod \ m)(bR \ mod \ m))\ R^{-1} \ mod \ m$.

If one factor is known to be constant, then the Montgomery reduction algorithm can be altered to calculate $abR^{-1} \ mod \ m$, as in algorithm 3.

Algorithm 3 Montgomery Multiplication with one known factor [23],[25]

Input: $a \in [0, mR[, b \in [0, mR[$
Input: $m \ modulus$
Input: $R = 2^l > m$
Input: $m'_b = -bm^{-1} \ mod \ R$
Output: $t \equiv (abR^{-1} \ mod \ m) \in [0, 2m[$
$k \leftarrow ((a \ mod \ R)m'_b) \ mod \ R$
$t \leftarrow (ab + km)/R$

For algorithm 3 to work, it is assumed that b remains constant so that m'_b can be calculated beforehand, hence the name Montgomery multiplication with one known factor.

1.2.2 Polynomial Multiplication

Most PQC algorithms work in a finite polynomial ring, which entails that most operations will be done with polynomials.

Addition and subtraction between polynomials can be calculated element wise, thus enabling a straightforward translation to software/hardware.

Polynomial multiplication however, introduces multiple complexities. The result of a polynomial multiplication between two degree n polynomials has a degree of 2n. But, as all PQC algorithms work in a finite ring, this resultant polynomial still needs to be reduced modulo some other polynomial. On top of that, every polynomial coefficient of the result is a sum of multiplications of different coefficients. The inherent complexity in polynomial multiplication leads to multiple different translations to software/hardware, further expanded upon in the remainder of this section.

Schoolbook Multiplication

In general, just as shown in equation 1.1, the multiplication of two polynomials is calculated by multiplying specific coefficients, and then summing the obtained terms to form the coefficients of the resulting polynomial. From equation 1.1, it is also clear that the schoolbook method of multiplying two $(n-1)$ degree polynomials requires n^2 coefficient multiplications.

$$a(x)b(x) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j x^{i+j} \tag{1.1}$$

Now, if the polynomials are represented as vectors: $a(x) = a_0 + a_1 x + ... + a_{n-1}x^{n-1}$ as $[a_0, a_1, ..., a_{n-1}]$ and $b(x) = b_0 + b_1 x + ... + b_{n-1}x^{n-1}$ as $[b_0, b_1, ..., b_{n-1}]$, with the index of each element indicating the degree of the term that it corresponds to, then the multiplication of both polynomials can be viewed as a one dimensional convolution, resulting in the vector representing polynomial $c(x)$, $[c_0, c_1, ..., c_{2n-2}]$, as derived in equation 1.2.

$$\begin{aligned} a(x)b(x) &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a[i]b[j]x^{i+j} \\ &= \sum_{k=0}^{2n-2}\sum_{j=0}^{n-1} a[k-j]b[j]x^{k} \\ &= \sum_{k=0}^{2n-2}\left(\sum_{j=0}^{k} a[k-j]b[j]\right)x^{k} \\ &\Downarrow \\ c[k] &= \sum_{j=0}^{k} a[k-j]b[j] \end{aligned} \tag{1.2}$$

Figure 1.1 shows an example of how a multiplication of two quadratics can be written as a one dimensional convolution, with the quadratics and the result represented as vectors.

When this operation is done in a finite ring, modulo $(x^n + 1)$, the result is a polynomial with a maximum degree of $(n-1)$. A naive way to calculate this would be to first do the full polynomial multiplication, and then take the modulus of the resulting polynomial.

NTT Multiplication

This section explains the Number-Theoretic Transform (NTT), all the way from the NTT domain to NTT multiplication. The explanation given here is loosely based on the explanation given in Daan Sprenkels' blog [51].

The basic idea behind the NTT is the following: reduce the degree of the polynomials to operate on, using the Chinese Remainder Theorem (CRT).

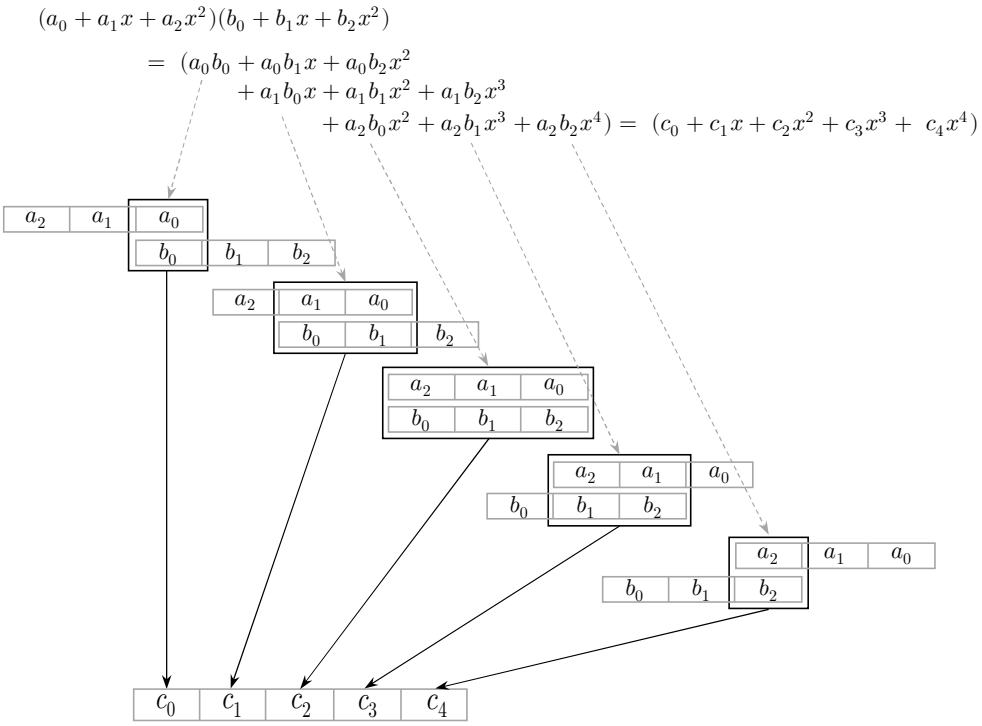


FIGURE 1.1: Polynomial multiplication written as a one dimensional convolution.

The CRT works by factoring the modulus, m, into multiple co-prime factors, $m_1, m_2, ..., m_k$, for which the following equation holds: $m = m_1 * m_2 * ... * m_k$. Then, it is known that there exists only one value of X $(mod\ m)$ for which the following system of equivalences holds:

$$\begin{cases} X \equiv c_1\ mod\ m_1 \\ X \equiv c_2\ mod\ m_2 \\ ... \\ X \equiv c_k\ mod\ m_k \end{cases} \tag{1.3}$$

This means that the transformation to the NTT domain, i.e. the representation as the system of equivalences modulo factors of m, transforms a polynomial of degree $(n-1)$ into k polynomials of degree $(n/k-1)$. In other words, it transforms a polynomial with n coefficients, into k polynomials with n/k coefficients.

The main advantage of working in the NTT domain is that the multiplication of two degree $(n-1)$ polynomials can be reduced to k times $(n/k-1)$ degree polynomial multiplications. Assuming that each multiplication between two polynomials of degree $(n-1)$ (with n coefficients) requires n^2 coefficient multiplications. After applying the CRT, the amount of coefficient multiplications is reduced to $k(\frac{n}{k})^2 = \frac{n^2}{k}$. If the modulus allows a factorisation into n terms, or in other words $k = n$, then the

amount of coefficient multiplications is reduced to n, allowing to do polynomial multiplication in linear time.

The first step to creating the NTT is to find a factorization of the modulus that will be used as the separate modulo in the CRT. The modulus for some of the finalist PQC schemes (including Dilithium) takes the form of a polynomial with the following structure: $(X^n + 1)$.

To factorize polynomials of the form $(X^n + 1)$, it helps to introduce the general factorization of polynomials of the form $(X^n \pm \alpha)$, as shown in equation 1.4 for the variant with a plus, and in equation 1.5 for the variant with a minus.

$$\begin{aligned}(X^n + \alpha) &= (X^{n/2} - \gamma)(X^{n/2} + \gamma) \\ &= X^n - \gamma^2 \\ &\Downarrow \\ \alpha &= -\gamma^2 \\ \gamma &= \sqrt{-\alpha} \\ \gamma &= \sqrt{(-1)\alpha}\end{aligned} \tag{1.4}$$

$$\begin{aligned}(X^n - \alpha) &= (X^{n/2} - \gamma)(X^{n/2} + \gamma) \\ &= X^n - \gamma^2 \\ &\Downarrow \\ \alpha &= \gamma^2 \\ \gamma &= \sqrt{\alpha}\end{aligned} \tag{1.5}$$

To keep the mathematics to a minimum, the remainder of this section specifically looks at the case where the modulus is $(X^{256} + 1)$, as is the case in Dilithium.

The first factorization step is shown in equation 1.6.

$$\begin{aligned}(X^{256} + 1) &= (X^{128} - \gamma)(X^{128} + \gamma) \\ &= X^{256} - \gamma^2 \\ &\Downarrow \\ \gamma^2 &= -1 \\ &\Downarrow \\ \gamma^4 &= 1 \\ &\Downarrow \\ \gamma &= \zeta_4\end{aligned} \tag{1.6}$$

The value found for γ in equation 1.6 is the fourth primitive root ζ_4. To explain: the more general k-th primitive root, ζ_k, is defined as the value for which $(\zeta_k)^k = 1$, with $(\zeta_k)^l \neq 1$ for all integers $l \in [1, k[$. This is indeed the case in derivation 1.6, as γ, γ^2, and γ^3 are all not equal to one, and thus four being the first integer power for which γ is equal to one.

From derivation 1.6, it is trivial to derive property 1, valid for the case of Dilithium.https://www.overleaf.com/project/62d723cff4d4a4320a435927

Property 1 *In the situation described above, i.e. when working in the ring* $\mathbb{Z}_q/(X^{256}+1)$*, as is the case for Dilithium, than the following equivalence holds:*

$$(\zeta_4)^2 = -1$$

A second property of the k-th primitve root is given below:

Property 2 *For the case of Dilithium, as described above, taking the square root of the k-th primitve root can be rewritten as a different primitive root, to the power of one, as in the equation below:*

$$\zeta_k^{1/2} = \zeta_{2k}$$

According to the definition of the k-th primitive root, property two is only valid if $(\zeta_k^{1/2})^{2k} = 1$ with $(\zeta_k^{1/2})^l \neq 1$ for $l \in [1, 2k[$.

For even $l = 2k'$, it follows from the definition of the k-th product that $(\zeta_k^{1/2*2})^{k'} = (\zeta_k)^{k'} \neq 1$. For uneven $l = 2k' + 1$, it follows from $k' + 1/2 \neq k$ that $(\zeta_k^{1/2*2})^{2k'+1} = (\zeta_k)^{k'+1/2} \neq 1$.

Property two leads to the following, more general, property:

Property 3 *In the case of Dilithium, as detailed above, raising the k-th primitive root to the power of i, with i a power of two, while also multiplying the k factor by i, results in the same value as the k-th primitve root itself, as detailed in the equation below:*

$$\zeta_k = \zeta_{i*k}^i \text{ with } i \text{ a power of two}$$

The result of the first factorization step, as shown in equation 1.6, thus gives the following result: $(X^{256} + 1) = (X^{128} - \zeta_4)(X^{128} + \zeta_4)$.

Each of the two factors found in the first factorization step can also be factorized. This factorization is then repeated on the resulting factors of the second factorization, and so on. In general, each factorization can be described as shown in figure 1.2. In this figure, the general behaviour of a single factorization step is given, applied to $(X \pm \zeta_{4k}^i)$. Where, in the first step, (a) and (b) refer to 1.5 and 1.4 respectively. And in the following operations, (c) refers to the 1st property of the k-th prime root , and (d) refers to the 2nd property of the k-th prime root.

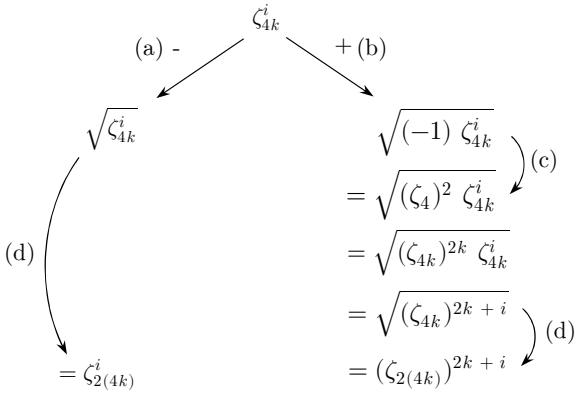


FIGURE 1.2: Factorization of $(X \pm \zeta_{4k}^i)$

Representing each constant γ, part of a pair of factorized binomials $(X^i \pm \gamma)$, as a node in a tree, results in the factorization tree depicted in figure 1.3.

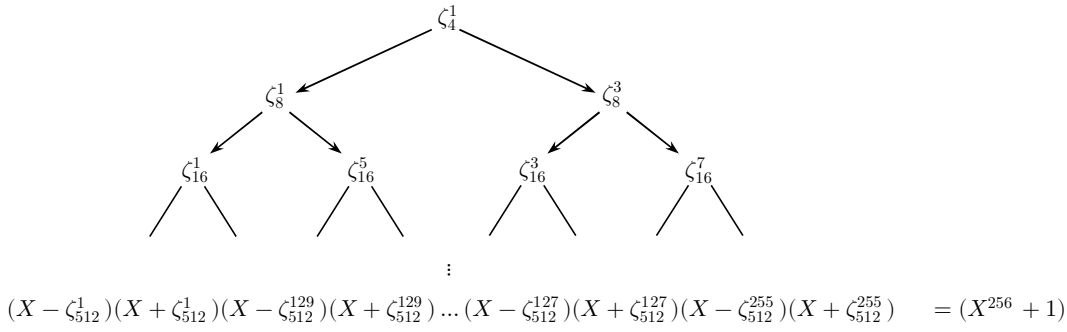


Figure 1.3: Dilithium factorization tree.

At the bottom of the tree in figure 1.3, after eight factorization operation layers, only terms of degree one remain. This shows that for Dilithium, it is possible to factor the 256 (=n) degree modulus into 256 (=k) polynomials of degree one. The benefit of this being that a multiplication of two polynomials only requires 256 scalar multiplications.

Thus, when applying the CRT in the case of Dilithium, the system of equivalences shown in equation 1.7 can be found.

$$\begin{cases} X \equiv c_1 \ mod \ (X - \zeta_{512}^1) \\ X \equiv c_2 \ mod \ (X + \zeta_{512}^1) \\ \ldots \\ X \equiv c_{255} \ mod \ (X - \zeta_{512}^{255}) \\ X \equiv c_{256} \ mod \ (X + \zeta_{512}^{255}) \end{cases} \tag{1.7}$$

The explanation given thus far shows how the NTT domain is equivalent to a system of equivalences derived from the CRT, and how to factorize a polynomial modulus with a specific structure. This does not yet however explain how to transform a degree n polynomial to the NTT domain.

The NTT transformation, just like the modulus factorization, can be described in different layers, where in each layer, the polynomial(s) in the previous layer are split in two different polynomials. This splitting works according to the previously explained factorization tree 1.3; Every polynomial in the previous layer, which will have a certain modulus $(X^l - \zeta_v^i)$, is split in two polynomials by taking the modulus of set previous polynomial with $(X^{l/2} - \zeta_{v'}^i)$ and $(X^{l/2} + \zeta_{v'}^{i'})$.

Every polynomial split can thus be viewed as finding the remainder of a degree n polynomial divided by a polynomial with half the degree, thus of degree n/2. In general, this remainder can be calculated as shown in figure 1.4.

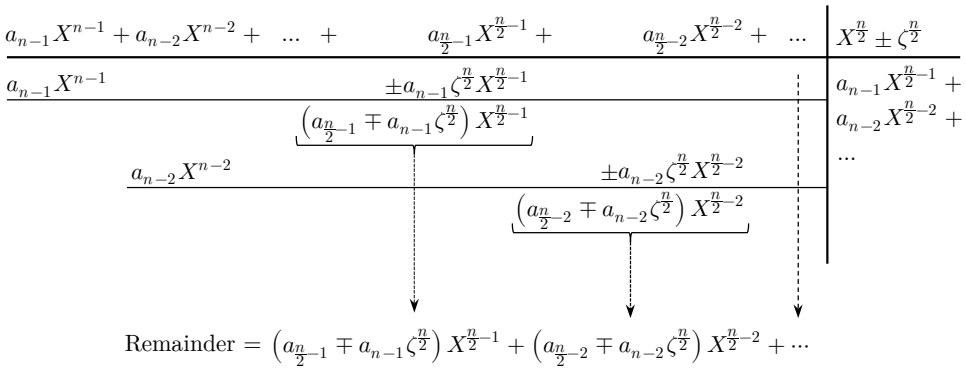


FIGURE 1.4: Obtaining the remainder of a half degree polynomial division.

From figure 1.4, it is clear that the two remainder polynomials can be directly written using the coefficients of the polynomial from the previous layer. To be more specific, every coefficient of the new polynomial, resulting from taking the modulus, $(X^{n/2} \pm \zeta^{n/2})$, of the previous layer polynomial, can be written as: $c_i = (a_i \mp a_{n/2-i} * \zeta^{n/2})$.

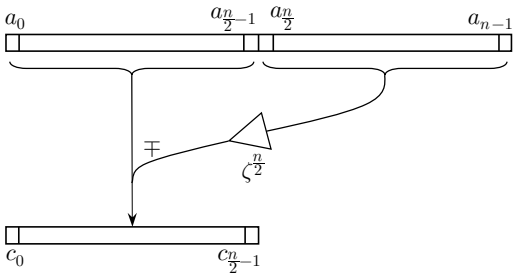


FIGURE 1.5: Abstract vector graphic of the reduction operation.

Figure 1.5 describes how the remainder of an $(n-1)$ degree polynomial divided by a $n/2$ degree polynomial can be calculated directly from the coefficients of the $(n-1)$ degree polynomial. This abstract view shows a large similarity to the butterfly operations present in the Fast Fourier Transform (FFT) [28]. The NTT can also be thought of as a finite field variant of the FFT. Because of this reason, the NTT is often written about in literature using FFT terminology, even though the NTT domain cannot be confused with the frequency domain, but is merely a mathematical construct with no direct physical interpretation [26]. For example, the $\zeta^{n/2}$ factors, also present in 1.5, are often referred to as twiddle factors, which was originally used to refer to the multiplication factors in the butterflies from the Cooley-Tukey FFT algorithm [28].

Repeating the polynomial splitting operation in the case of Dilithium (with polynomials of degree 255) results, after eight splitting operation layers, in 256

scalars. This operation is depicted in figure 1.6.

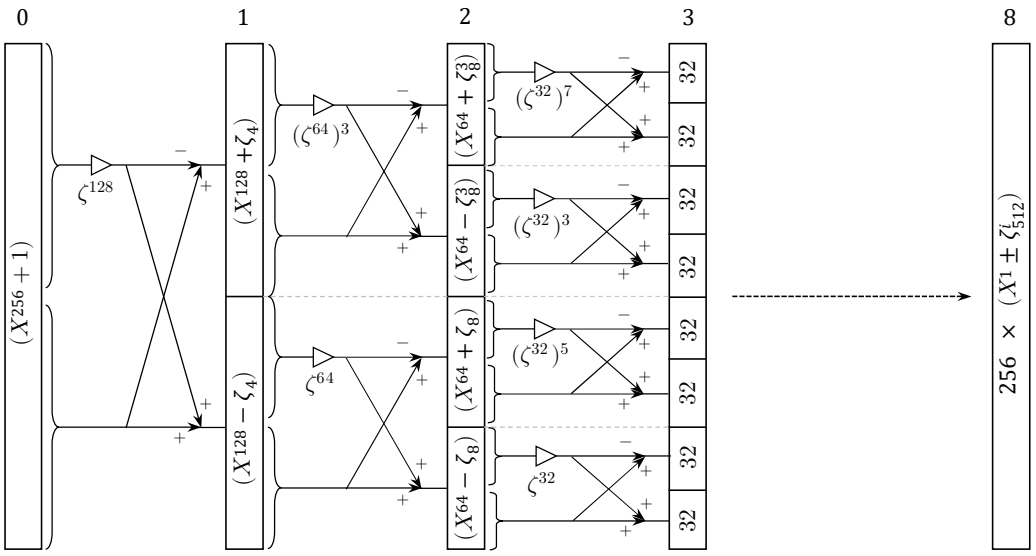


Figure 1.6: Full Dilithium NTT transformation diagram.

With a time complexity of $O(n \log n)$, just like the FFT, the transformation to the NTT domain of the two polynomials to be multiplied is more detrimental to the overall performance than the polynomial multiplication in the NTT domain itself (only requiring $O(n)$ operations). This transformation cost however can sometimes by amortized, by only transforming a polynomial once to the NTT domain, and using it to do multiple polynomial multiplications.

The transformation to the NTT domain can also be interpreted as an evaluation of an n degree polynomial in n different values for X. This results from the fact that taking the remainder of a polynomial, $p(X)$, divided by $(X - \beta)$, is equivalent to calculating $p(\beta)$. This can be better understood, without providing a formal proof, by working out the division for a general polynomial, as done in figure 1.7.

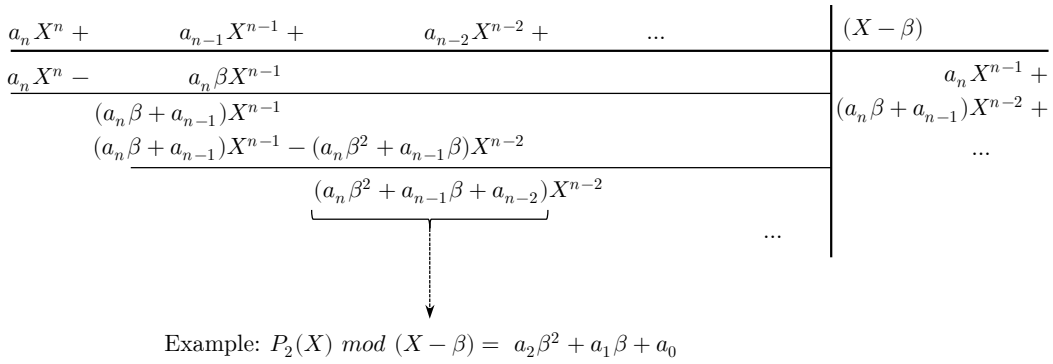


FIGURE 1.7: "proof" of evaluation by reduction modulo $(X - \beta)$.

FNT Multiplication

The Fermat Number Transform (FNT), as explained in [1], is a variant of the "*NTT, in that the modulus is a Fermat number:* $F_t = 2^{2^t} + 1$". Choosing a Fermat number as the modulus allows for a more efficient computation, as the twiddle factors for the initial t layers in the NTT transformation are powers of two. By virtue of choosing these twiddle factors, the multiplications with these factors can be calculated as shift operations, also reducing the need for large twiddle factors to be loaded from memory.

Karatsuba and Toom-Cook

This section explains the basics of Karatsuba and Toom-Cook multiplication by paraphrasing part of the explanation given in [33].

Karatsuba multiplication works by splitting a degree n polynomial, $a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_0$, into two degree n/2 polynomials (see equation 1.8), related to $a(x)$ according to $a(x) = a_h(x)x^{n/2} + a_l(x)$.

$$\begin{aligned} a_h(x) &= a_{n-1}x^{n/2-1} + a_{n-2}x^{n/2-2} + ... + a_{n/2} \\ a_l(x) &= a_{n/2-1}x^{n/2-1} + a_{n/2-2}x^{n/2-2} + ... + a_0 \end{aligned} \tag{1.8}$$

This way of rewriting a degree n polynomial also allows to rewrite the multiplication of two degree n polynomials, $c(x) = a(x)b(x)$, as in equation 1.9.

$$\begin{aligned} c(x) = {} & a_h(x)b_h(h)x^n \\ & + \left((a_h(x) + a_l(x))\left(b_h(x) + b_l(x)\right) - \left(a_h(x) \odot b_h(x) + a_l(x) \odot b_l(x)\right)\right)x^{n/2} \\ & + a_l(x)b_l(x) \end{aligned} \tag{1.9}$$

This method of multiplication alters the amount of multiplications from one, degree n polynomial multiplication, to three n/2 polynomial multiplications and some

extra summation and multiplication overhead. Applying the same splitting method to $a_h(x)$ and $a_l(x)$, further reduces the degree of the polynomials to be multiplied to n/4, n/8 and so on. This polynomial splitting is often done until the polynomials are small enough to be efficiently calculated using schoolbook multiplication.

Karatsuba multiplication can be viewed as a specific case of Toom-Cook k-way multiplication. Where Toom-Cook k-way multiplication, instead of splitting the polynomials of degree $(n-1)$ into two polynomials of degree $(n/2-1)$, splits the polynomials into k different polynomials of degree $(n/k-1)$. The details of the Karatsuba and Toom-Cook multiplication techniques are out scope for this thesis, more information on these techniques can be found in [33].

1.2.3 Hashing

A Hash function is a function that is able to digest a message and deterministically output a condensed representation of set message. Of particular interest in the domain of cryptography are cryptographic hashes. These hashes offer certain security properties, not generally offered by other hash variants.

Cryptographic hashes often offer one or multiple of the following properties:

1. **Pre-Image resistance**: For a given hash value, it is hard to find the input that corresponds to that hash value. Or in other words, it should be hard to calculate the hash operation in reverse. This property is for example desired when the hash is used to store password hashes in a password file. Even if an attacker is able to access the password file, as long as the hash is pre-image resistant, the attacker won't be able to reverse the hash function to obtain the original password. [47]

2. **Second Pre-Image resistance**: For a given input, and corresponding hash value, it is hard to find another input with the same hash value. This property is for example desired in the case of a courier, working for a company that needs to transfer documents with the employee salary data between buildings, without the courier being able to increase their own salary. To achieve this, the company communicates a hash of the original document over the phone to the building that receives the document. The receivers of the document in the other building then verify at arrival that the hash of the received document has still the same value as the hash value received over the phone. Now, only if the hash function used is second pre-image resistant, will the courier not be able to alter his/her salary (and some other parts of the document), so that the new document produces the same hash as the original document. [47]

3. **Collision resistance**: It is hard to find two input values with the same hash output. This property is for example desired in the case that two people, say Alice and Bob, decide to guess the amount of beans in a jar. To make sure that no one alters their initial guess, after revealing the real amount of beans in the jar, it is decided that both Alice and Bob hash their guessed value, and pass this hashed value to the other person. After the real number is revealed, both

> Alice and Bob are able to verify that the other person did not change their initial guess, by hashing the other persons guess, and comparing to the original hash value received from the other person. Now, only if the hash function is collision resistant, will Bob or Alice not be able to find a second guess with the same hash as their original guess value. [47]

In the definitions for the security properties given above, a problem is considered hard if in the order of 2^n attempts, with n the amount of bits in the hash output, need to be made to solve the problem.

An example of a hash function that is not considered to be a cryptographic hash is the function that generates the Cyclic Redundancy Check (CRC), generating a value check for blocks of data entering the function, thus fitting the defintion of a hash. However, as CRC functions are easily reversibly, it not considered a cryptographic hash function.

Currently, NIST specifies two Federal Information Processing Standards (FIPS) for cryptographic Hash algorithms [39].

The secure hash standard (FIPS 180-4), specifies both the SHA-1 and SHA-2 hash families. In 2011, NIST decided to deprecate the use of SHA-1, due to the potential for brute force attacks [42]. Eventually, in 2017, "*Google announced that a team of researchers from the CWI institute in Amsterdam and Google*" [42] were successfully able to find a collision attack on the SHA-1 hash, thereby producing two different files with the same hash value [52][42]. The SHA-2 family, first published in 2001, includes six fixed output length hash functions. These SHA-2 family member hash functions all adhere to the same naming convention; all aptly named "SHA-x", where x denotes the fixed output length of the hash function.

To further improve the robustness of the NIST hash toolkit, and to give an alternative to the SHA-2 family [37], NIST released the SHA-3 standard (FIPS 202) in 2015, specifying the SHA-3 hash family. The SHA-3 hash family contains four fixed output length hash algorithms; SHA3-224, SHA3-256, SHA3-384, and SHA3-512. Also, in contrary to the SHA-2 hash family, the SHA-3 hash family contains two eXtendile Output Functions (XOFs).

These XOFs, called SHAKE-128 and SHAKE-256, allow for a variable-length output, and are named according to their expected security level [39]. While the fixed length hash functions of the SHA-3 family are already approved, the XOFs are still subject to additional security considerations.

1.3 Hardware Overview

Besides understanding the Dilithium workload, it is equally important to understand the hardware on which this workload needs to run. That is why this section introduces the Armv8.1-M ISA and the Cortex-M55, i.e. the processor that implements the Armv8.1-M ISA, for which this thesis studies the Dilithium workload performance. It also gives an overview of the specific groups of instructions that are unique to the Armv8.1-M ISA, or that could be useful when implementing Dilithium or other lattice-based PQC algorithms.

As the exact architectural hardware information is not disclosed to the author of this thesis, this section will mainly focus on the higher level, more functional information that is publicly available.

1.3.1 Armv8.1-M ISA

The Armv8.1-M Instruction Set Architecture (ISA) is part of the M-Profile of the Arm architecture. The M-Profile architecture is one of the three, by Arm defined, profile architectures:

1. **A-Profile:** The Application-Profile architecture "*targets high performance markets, such as PC, mobile, gaming, and enterprise*" [3].

2. **R-Profile:** The Real-time Profile architecture "*provides high-performing processors for timing sensitive and safety-critical environments*" [17]

3. **M-Profile:** The Microcontroller-Profile architecture "*provides a standard instruction set and programmer models for secure microprocessors that are optimized for the lowest power consumption, low-latency and highly deterministic operation for deeply embedded systems*" [14].

The Armv8.1-M ISA introduces the Helium vector extension, which is designed to deliver "*a significant performance uplift for Machine Learning (ML) and Digital Signal Processing (DSP) applications for small, embedded devices.*" [11]

1.3.2 Parallelism in the Armv8.1-M ISA

Parallelism, originally from the Greek "para allēlois", literally meaning "beside" (para) "each other" (allēlois) [30], is here used to refer to the ability for multiple calculations to be performed simultaneously, i.e. at the same moment in time. This enables faster or more energy efficient execution, at the cost of a larger die area.

Hardware parallelism is often characterized by the amount of data and instruction streams the hardware has. This characterization leads to four main categories in the realm of hardware parallelism [46].

The first of which contains processors executing one instruction, operating on one data-stream at a time, hence referred to as Single Instruction/Single Data (SISD) processors [46]. Because of the low area and energy requirements of Cortex-M family processors, some of these processors, like the Cortex-M0, only support SISD instructions. A SISD processor however can support concurrent processing of instructions, for example, the Cortex-M0 is able to concurrently process multiple instructions due to its 3-stage pipeline.

On the other hand, some, typically more high end processors, offer the possibility of executing multiple instructions on different data streams at the same time, hence referred to as Multiple Instruction/Multiple Data (MIMD) processors [46]. Examples of these processors can be found under the A-profile architecture processors, typically in the form of multi-core superscalar processors [4]. These are able to execute multiple

instructions on different cores, all operating on different data, at the cost of a larger area and energy footprint.

Processors that implement parallelism by executing multiple different operations on a single data-stream, often referred to as Multiple Instruction/Single Data (MISD), are quite rare. These processors typically occur in fault-tolerant systems, executing the same instruction on the same data multiple times, giving the processor a better chance to detect and mask errors that might have occurred. The flight control computers on the Space Shuttle for example could thus be considered as MISD [4].

The last, and the most often encountered category of parallelism is referred to as Single Instruction/Multiple Data (SIMD). This category is explained in more detail in subsection 1.3.2.

Some forms of parallelism, like superscalar execution for example, only require support from the micro-architecture implementing a specific ISA, and do not influence the ISA itself. For SIMD however, this is most often not the case, as special instructions need to be added to the ISA in order for the user to be able to optimally make use of the SIMD available in hardware.

Another way parallelism is often categorized, depends on what it is applied to. Instruction Level Parallelism (ILP) for example, refers to the parallel execution of multiple instructions, while Data Level Parallelism (DLP) refers to the parallel execution of the same operation on independent data.

The Cortex-M55 offers a limited amount of ILP, as it implements a 4-stage pipeline (5-stage if the Helium vector extension is included), allowing multiple instructions to be processed at the same time. However, as it is part of the Cortex-M family, it does not implement ILP techniques that require more area, like superscalar execution or Very Long Instruction Words (VLIW). In that same spirit, the Cortex-M55 only supports in-order execution.

The Cortex-M55, with the addition of the M-profile Vector Extension (MVE), does however allow for an increased amount of DLP compared to previous M-profile processors, as will be elaborated upon in the remainder of this section.

Single Instruction/Multiple Data (SIMD)

SIMD processors allow a single instruction to be executed on multiple data-streams at the same time. In other words, SIMD operations operate on vectors of data, hence why architectures implementing SIMD instructions are sometimes referred to as vector architectures [25].

It should however be noted that the term "vector architecture" is sometimes also used to refer to processors with a design similar to the computers designed by Seymour Cray starting in the 1970s [46]. These vector processors, as they are called, use a pipelined Arithmetic Logic Unit (ALU) instead of multiple ALUs, to operate on the data vectors. This allows vector processors to operate on variable length vectors, with the vector length under the control of the programmer. This is in contrast to most of the fixed length SIMD extensions added by Arm and Intel over the years. This aside, the rest of this section will use the former definition of a "vector architecture".

Citing the explanation given in [25]: vector architectures "*add a set of wide registers that are viewed as vectors of equal-sized blocks called lanes, and provide instructions that operate on those lanes in parallel*".

The A-profile of the Arm architecture offers a range of vector extensions, including the Neon vector extension, the scalable vector extension (SVE), its successor SVE2, and the scalable matrix extension (SME). We briefly comment on those.

The Neon vector architecture, originally introduced with the Armv7-A architecture, offers the SIMD capability of operating on multiple lanes in 64-bit doubleword or 128-bit quadword vector registers. To accomplish this, the Neon architecture for AArch32 (equivalent to the ARMv7 Neon architecture) works with a vector register bank consisting of 32 x 64-bit registers, which can be viewed as 16 x 128-bit quadword registers (Q0-Q15), or 32 x 64-bit doubleword registers (D0-D31). A similar story is true for the Neon architecture for AArch64, which uses 32 x 128-bit registers [5]. Each quadword register can be used to operate on 16 x 8-bit, 8 x 16-bit, 4 x 32-bit or 2 x 64-bit values using a single SIMD instruction [10]. The scalable vector extension (SVE) is another vector extension, specifically designed for "*high performance computing (HPC) and machine learning (ML) applications*" [12]. It allows hardware implementations to be made with vector registers from 128-bit to 2048-bit, at 128-bit intervals. This increases the amount of operations per instruction over Neon by a factor of somewhere between 1 and 16, depending on the Neon architecture and the vector length of the SVE implementation. Arm also introduced its successor, the SVE2, a superset of the SVE and Neon extension, with the purpose of extending "*the SVE instruction set to enable data-processing domains beyond HPC and ML*" [13]. The SVE is mainly used in high performance computing and machine learning applications [12]. Which is exemplified by its implementation in the Neoverse V1 processor, used in Amazon's Graviton3 processors designed for cloud workloads, or its use in Japan's next-generation supercomputer [48]. The SVE2 on the other hand, as already mentioned, extends its use to other common algorithms [13]. For this reason, it is part of the Armv9 architecture and implemented in the Cortex-A510, Cortex-A710, Cortex-A715, Cortex-X2, and Cortex-X3 processors.

While increasing the amount parallelization through SIMD can theoretically process more data, this heavily depends on the application. Most desktop applications for example are largely sequential, and don't benefit very much from an increased amount of parallelization. In the realm of cryptography for example, when working with big number arithmetic, as is the case in RSA, leveraging the power of SIMD is made difficult by the presence of plenty of sequential carry chains. This is however not the case for Dilithium, as well as many other lattice-based cryptography schemes, as they make use of polynomials instead, where the use of SIMD is not hampered by the occurrence of carries.

When it comes to M-profile processor architectures, SIMD instructions are not easily supported within the usually very tight power and area constraints of the embedded market However, the Cortex-M4, Cortex-M7, Cortex-M33, and the Cortex-M35P did already support SIMD instructions that operate on 8- or 16-bit values [53], by adding a form of subword parallelism. This allows 32-bit registers to be split up into two 16-bit values, or four 8-bit values that a single SIMD instruction operates

on in parallel [8]. This is especially useful for video or audio applications, as they do not require the full 32-bit accuracy. With the introduction of the Armv8.1-M architecture, Arm expands the SIMD capabilities of edge devices even more, by introducing the M-profile Vector Extension (MVE).

M-profile Vector Extension (MVE)

The Armv8.1-M ISA introduces the M-profile Vector Extension (MVE), also known as the Helium vector extension. Helium can be seen as the M-profile equivalent to the Neon vector extension for the A-profile of the Arm architecture, bringing the benefits of SIMD to the embedded market and enabling speedups in a wide range of applications including signal processing and machine learning. This gives designers the option to speed up Cortex-M applications even more than previous Cortex-M processors, with the help of SIMD.

Similarly to the Neon vector extension, Helium uses 128-bit vector registers. However, unlike the 32 x 128-bit Neon vector registers, Helium only has 8 x 128-bit vector registers. This difference in the amount of registers is partially compensated for by the prevalent presence of SIMD instructions using both vector registers and scalar registers from the General Purpose Register File (GPRF), helping to alleviate some pressure from the smaller Vector Register File (VRF). The frequent use of scalar-vector operations is not very feasible in A-profile architectures as the physical distance between the GPRF and the VRF is to large for low latency operations. Yet the small area nature of M-profile architectures makes it possible to implement low latency scalar-vector instructions [25].

With Helium having a smaller VRF, a certain element of complexity is passed onto the programmer. Typical vectorization techniques, like batching, i.e. doing the same operation on multiple higher level constructs, as explained in [25], are not very suited to the Helium architecture, as they require a large number of vector registers and don't often use General Purpose Registers (GPRs).

Implementations of Helium use the concept of a beat; 32-bits of arithmetic compute. The processing of 128-bits of arithmetic compute can be split up into four beats. Every beat in itself can contain multiple lanes, for example, when operating on 8-bit elements, every beat, 4 x 8-bit lanes are processed. When working in a dual-beat implementation of Helium, as is the case for the Cortex-M55, two beats can be processed per cycle. A full 128-bit SIMD instruction than requires at least two clock cycles to complete [53].

The Armv8.1-M architecture allows neighbouring vector instructions to overlap by two beats: that is, the first two beats of the next instruction may execute in parallel with the last two beats of the previous instruction. Practically, however, this is only possible if there are no conflicting hardware requirements for the two instructions.

The capability to overlap operations, that the ArmV8.1-M ISA allows for, is strictly separate from a dual-issue capability. The dual-issue capability refers to the simultaneous fetching of two instructions. The instruction overlapping capability on the other hand still only allows one instruction to be fetched each clock cycle, but

allows for the concurrent processing of the last two beats of the previously fetched instruction and the first two beats of the current instruction.

Currently, there exist two implementations of the Armv8.1-M ISA with the addition of the MVE: the Cortex-M55 and the Cortex-M85. For the remainder of this thesis we will focus on the Cortex-M55, explained in more detail in section 1.3.4.

1.3.3 Noteworthy Instructions for Implementing Dilithium

This section goes over multiple instructions, part of the Armv8.1-M ISA, that will be relevant when implementing the Dilithium or the more general lattice-based PQC workload. For every group of similar instructions, this section explains why they are included in the ISA, and how they work.

Fixed Point Arithmetic

With Helium, Armv8.1-M offers support for fixed point arithmetic, as explained in [23]. In this way, the Armv8.1-M is similar to other Arm ISA that add support for DSP extentions, like the Neon or the SVE 2 extention.

In fixed point arithmetic, fractional values are represented with integers, implicitly scaled by a certain scaling factor. Often, this scaling factor is chosen to be a power of two for efficiency reasons.

When working with a n-bit single-width signed integer, $a \in \{-2^{n-1}, -2^{n-1}+1, ..., 2^{n-1}-1\}$, and choosing the scaling factor to be, 2^{n-1}, i.e. using $(n-1)$ bits as fractional bits, then the integer a represents the fractional value $\frac{a}{2^{n-1}}$. This particular choice for the scaling factor allows the integer a to represent values in the interval $[-1, 1[$.

Similarly, when keeping the range the same, but working with 2n-bit, double-width signed integers, than the integer a represents the value $\frac{a}{2^{2n-1}}$.

Multiplying two single-width signed integers, and storing the result as a double-width integer introduces a factor of two difference. This difference is an artifact of the dissimilarity between the fixed point scaling factors of the single and double-width representations, as detailed in equation 1.10.

$$\frac{a}{2^{n-1}}\frac{b}{2^{n-1}} = \frac{ab}{2^{2n-2}} = \frac{2ab}{2^{2n-1}} \tag{1.10}$$

For this reason, the Armv8.1-M ISA offers a doubling long multiplication instruction:

Vector Saturating Doubling Multiply Long

```
VQDMULL<T><v>.<dt> Qd, Qn, Qm
VQDMULL<T><v>.<dt> Qd, Qn, Rm
```

Likewise, for working with single-width approximations of the product of two single-width fixed point values, the Armv8.1-M ISA offers a returning high half variant:

Vector Saturating (Rounding)
Doubling Multiply Returning High Half

```
VQRDMULH<v>.<dt> Qd, Qn, Qm
VQRDMULH<v>.<dt> Qd, Qn, Rm
```

This instruction allows for two types of rounding, calculating $\frac{\lfloor \frac{ab}{2^n} \rceil}{2^{n-1}}$ or $\frac{\lfloor \frac{ab}{2^n} \rfloor}{2^{n-1}}$, depending on if R is or is not present respectively.

Finally, Armv8.1-M also supports multiply-accumulate-add variants for the fixed point single-width approximation:

Vector Saturating (Rounding)
Doubling Multiply Accumulate High Half

```
VQRDMLAH<v>.<dt> Qda, Qn, Rm
```

In all of the mnemonics above, the Q stands for saturating, this means that if the result exceeds the range after rounding, then the result is clipped to the minimum or maximum value in the $[-2^{dt-1}, 2^{dt-1} - 1]$ interval, with dt: the number of bits each element in the 128-bit register represents.

Low overhead loops

Because signal processing applications often operate on an array of data, it is important that loop operations can be computed efficiently. To help accomplish this, the Armv8.1-M ISA adds the low-overhead-branch (LOB) extension, which helps to reduce the overhead often associated with loops [54].

Creating a low-overhead loop using the Armv8.1-M ISA requires at least two instructions; an instruction at the start of the loop, and an instruction at the end of the loop. The instruction at the start of the loop is passed the number of loop iterations that need to be performed, i.e. the loop counter. The loop counter value is then stored in the Link Register (LR). This means that the loop counter is automatically saved in the event an interrupt or an OS context switch occurs in the middle of the low-overhead loop, and thus that it will be restored when execution is resumed. Also improving software portability, as existing C/C++ code does not need to be changed to take advantage of the LOB extension [54].

Loop Start Instructions

```
WLS LR, Rn, <end loop label>              // Similar to for-loop
DLS LR, Rn       // For-loop, but always executes the first loop
```

In the first iteration, both the start and the end loop instructions need to be executed. But, after the execution of the end loop instruction, the loop information is cached internally in the processor. This allows subsequent loop iterations to skip the loop start and end instruction execution, and in turn reduces the loop overhead.

Loop End Instruction

```
LE LR, <start loop label>
```

The last iteration of a loop executing SIMD operations, operating on a fixed amount of elements each iteration, often needs to operate on an amount of elements that does not take up all the parallelisation offered by the SIMD instructions. Typically, this is handled by executing element per element. In order to avoid this, the Low Overhead Branch (LOB) extension adds tail predicated instructions.

These instructions will predicate the last SIMD execution to make sure only the correct elements are operated on.

Taking the example of operating on 15 32-bit elements using Helium SIMD instructions, that operate on 4 elements every loop iteration, then the last iteration is predicated to only operate on three elements, and for example not writing the last element (not operated on) to memory.

Tail Predicated Instructions

```
WLSTP.<size> LR, Rn, <label>
DLSTP.<size> LR, Rn
LETP LR, <label>
```

Non-Consecutive Memory Accesses

The Armv8.1-M ISA adds instructions to more efficiently load and store data with a specific structure in memory [54]. This can for example be used when working with images represented using the RGBA (or CMYK) color model in memory, where data is loaded to 4 single color-value 128-bit registers using a 4 way interleaving operation. Another example would be when working with stereo audio, to load (store) the left and right audio channels to (in) two separate 128-bit registers.

For these purposes, Armv8.1-M provides (de-)interleaving instructions, for storing (loading) data to (from) memory. For both loading from or storing to memory, there exist both 2-way and 4-way interleaving instructions. These 2-way and 4-way interleaving instructions are not able to load/store all the values from two and four 128-bit registers at once, and hence why they need to be called two or four times with different pat variables respectively. With the pat variable (see assembly instruction below) indicating the pattern of values that is stored using that instruction, as exemplified in figure 1.8.

(De-)Interleaving Instructions

```
VLD2<pat>.<size> {Qd, Qd+1}, [Rn]{!}
VLD4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]{!}
VST2<pat>.<size> {Qd, Qd+1}, [Rn]{!}
VST4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]{!}
```

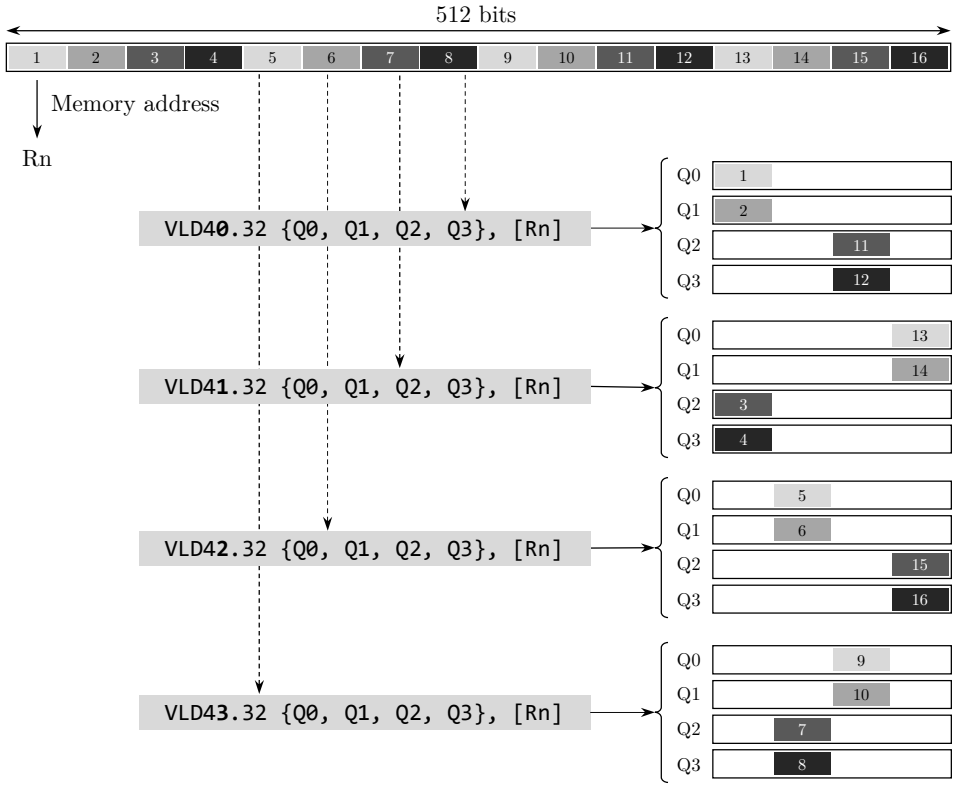


Figure 1.8: VLD4 memory access patterns

Other common excess patterns include circular and bit-reverse addressing. For these purposes, Armv8.1-M provides scatter and gather memory access instructions. These instructions allow to transfer data between the 128-bit registers and memory using a vector of addresses or a vector of offsets from a base address. On top of that, Armv8.1-M also provides specific instructions to create the correct addressing vector, which can for example be used to create circular addresses (VIWDUP). With the benefit of a large variety in applicability, due to the general nature of these memory access instructions.

Scatter-Gather Instructions

```
VSTR{B,H,W,D}<v>.<dt> Qd, [Rn, Qm]
VLDR{B,H,W,D}<v>.<dt> Qd, [Rn, Qm]
```

1.3.4 Cortex-M55

The Cortex-M55 processor is the first processor to implement the Armv8.1-M Instruction Set Architecture (ISA). It aims to improve performance for Artificial Intelligence (AI) and Signal Processing in endpoint devices.

The Cortex-M55 CPU optionally includes a dual-beat implementation of the Helium vector extension. This means that, with the extension, the Cortex-M55 is able to do a 64-bit multiply and accumulate operation (MAC) per cycle. As the Helium registers are 128-bit, every vector operation requires at least two cycles.

The pipeline design in the Cortex-M55 allows the processing of the last two beats of the previous instruction to overlap with the processing of the first two beats of the current instruction, as long as there are no conflicting hardware requirements. Citing from [53]: "*In simple terms, the pipeline partition allows overlapping of instructions belonging to vector load/store, vector integer, and vector floating-point categories*". In other words, while the second 64-bit operation is done for an earlier instruction, an instruction from another group can already operate on the result from the first 64-bit operation.

The Cortex-M55 implementation also allows for a so called, *limited dual issue* capability. This entails that some pairs of 16-bit Thumb instructions can be fetched at the same time.

While high end processors often implement out-of-order execution, this is not the case for the single-issue in-order execution of the Cortex-M55. Here the burden is placed on the programmer or the compiler to make sure that instructions are ordered in such a way as to not introduce stalls in the pipeline execution, e.g. by obeying latencies and instruction overlapping capabilities.

It should be noted here that the Software Optimization Guide (SWOG), which contains the details about which combination of instructions stalls the pipeline, is not yet publicly available.

1.3.5 Ethos-U55

The Ethos-U55 is a Neural Processing Unit (NPU), designed to speed up inference of machine learning (ML) in the cortex-M family. Arm claims to achieve "*a 480x uplift in ML performance over existing Cortex-M based systems*" [16]. However, it should be noted that it only supports 8-bit weights, and 8/16-bit activations. This microNPU boasts a maximum of 256 8-bit MAC units, and somewhere between 18 to 50 KB of internal SRAM [16].

1.4 Previous Work

Computing the NTT requires some form of modular reduction after the twiddle factor multiplications. The original Dilithium implementation, i.e., the c implementation part of the third round NIST submission, utilises Montgomery reductions for this purpose. However, as explained in section 1.2.1, this introduces a shift by R^{-1}, and this requires the inverse shift to be performed when doing the inverse NTT.

The Neon NTT paper [23] introduces the concept of Barrett multiplication. The Barrett multiplication with one-known-factor, similar to how the Montgomery one-known-factor multiplication alters the Montgomery reduction algorithm, alters Barrett reduction for the case of multiplying two values, as shown in equation 1.11.

$$\begin{aligned} ab \ mod^{\pm} \ m &\approx ab - \left\lfloor \frac{ab * \left\lfloor \frac{2^l}{m} \right\rfloor}{2^l} \right\rfloor * m \\ &\approx ab - \left\lfloor \frac{a * \left\lfloor \frac{b2^l}{m} \right\rfloor}{2^l} \right\rfloor * m \end{aligned} \tag{1.11}$$

Here, one of the two factors to be multiplied, b, is pulled into the approximation of m^{-1}, which is calculated in advance and stored in memory.

Implementations of Montgomery multiplication, Montgomery reduction, and Barrett reduction are found for the Cortex-M55 in [25], and for the neon SIMD extention of the Armv8-A ISA in [23].

The authors of [25] study polynomial multiplications used in PQC on the Cortex-M55. Implementing both the striding Toom-Cook/Karatsuba, the ancestors of which are explained in 1.2.2, and the NTT on the Cortex-M55, they find that, while Toom-Cook allows for a reduction in memory-usage and computation time, the NTT implementations still remain faster. The polynomial multiplication paper [25] also gives a summary of the M-profile Vector Extension (MVE) and discusses some implications its structure brings with it.

The authors of [2] propose a different implementation to calculate some of the vector multiplications, with an upper bound on the size, present in Dilithium. To be more specific, they propose reducing the value of the original modulus to some Fermat prime, like 257 or 769, depending on the NIST security level. This allows them to implement these polynomial multiplications using the FNT, as explained in section 1.2.2. For this, they implement the "*FNT on the Cortex-M4 and make use of its barrel shifter*" [2].

In the embedded processing environment, where costs play an important role, the processing memory preferably is kept to a minimum. The transition to PQC however, brings with it larger key and signature sizes, and thus a necessary increase in the amount of memory that will be needed. As mentioned in [27], the Dilithium implementation in the bench-marking framework pqm4 (Cortex-M4) requires 50 to a 100 KiB of memory. At the cost of sacrificing performance, the authors of [27] were able to get the memory requirement below 8 KiB for both key and signature generation, and below 3 KiB for signature verification.

This was achieved using a combination of techniques, all exploiting a trade-off between cycle count and memory requirement. For example, storing intermediate results that don't require a full 32-bits in a compressed format, at the cost of needing to decompress them when they need to be operated on.

Another way to reduce the memory overhead explained in [27] is to not store full vectors when this can be avoided. Some vectors, where each element is generated from a seed using SHAKE-256, do not need to exist fully in memory at any moment in time, but can be streamed polynomial by polynomial when they are needed. Another example would be not to precompute NTTs, but only generate them when they are needed. This of course means that an extra cost of needing to recompute the NTT is incurred whenever you want to do a multiplication.

The authors conclude by stating "*that storing the pubic keys and signatures has arguably become a bigger challenge than storing its run-time state*" [27].

1.5 Research Question

Originally this thesis aimed at finding whether the Cortex-M55, and the ISA that it implements, is suited to execution of PQC workloads. However, as there is a lot of variety between PQC workloads, we decided to focus on the Dilithium workload, which results in the following research question: "Is the Cortex-M55, and the ISA that it implements, suited to the execution of the Dilithium workload ?" To test the found results in practice, this thesis derives an implementation of the Dilithium signature scheme for the Cortex-M55 with the Helium vector extension.

1.6 Thesis Structure

After this chapter explained the fundamental concepts needed to understand the rest of this thesis, the following chapter gives a more in-depth explanation of Dilithium, the lattice-based PQC digital signature algorithm that this thesis focuses on. Followed by chapter 3, where we present some high level optimization techniques, used to improve the Dilithium performance. In chapter 4, we then go through some lower, assembly level optimizations. Finishing with chapter 5, where we describe the difficulties we faced, give some suggestions for further research, and conclude this thesis.

Chapter 2

CRYSTALS-Dilithium

This chapter gives a more in depth explanation of the Dilithium workload, briefly mentioned in chapter 1. It starts by giving a practical overview of the three components that together compose Dilithium, followed by a schematic representation of each of these three components, and ending with a characterization of the Dilithium workload.

2.1 Dilithium Operation Overview

Dilithium is a lattice based digital signature scheme, that operates over the polynomial ring $R_q = \mathbb{Z}_q/(X^{256}+1)$ with $q = 2^{23} - 2^{13} + 1 = 8380417$ [19]. It consists of three sub algorithms; key-pair generation, signature generation, and signature verification.

The key-pair generation algorithm produces a private and public key-pair, that can be used to sign and verify respectively. The signature generation algorithm takes both a message and a secret (private) key, and outputs a signature. The signature verification algorithm then also takes a message, and tests with the public key, if the signature is valid for the given message.

The working principle of Dilithium is often explained in literature with the help of algorithmic descriptions based on the algorithmic descriptions given in the Dilithium specification document [19], similar to the descriptions provided in Appendix A. These descriptions however fail to give readers, not coming from a mathematical background, a clear overview of the operations that constitute Dilithium, and the value structures that are operated on (like scalars or vectors of bytes/polynomials).

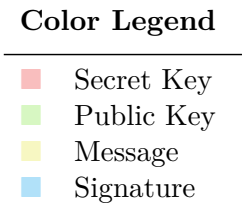
2.2 Schematic Representation

To get a better picture of the Dilithium workload, this section translates the algorithmic descriptions given in Appendix A to a more readable schematic format. These summarizing schematics depict the operations in rounded boxes, connected to the input and output data structures with solid arrows, indicting the data flow of each algorithm. Dotted lines on the other hand indicate control flow, often connected to

diamond shaped condition boxes. At the end of some of the schematics, a dotted line is present, below which the data structures, produced by the function, are depicted.

To keep an overview of the large scale data and control flow of each of the three components of Dilithium, the schematics below make sure to fit the schematic of every one of the components on a single page. To achieve this, it is sometimes necessary to group some operations, and depict them in another figure. This is done by representing this group of operations as a rounded box, with, if relevant, an all-encompassing name of the operations contained within, and a letter (case sensitive) referring to the same box depicted in more detail in another figure.

Further, the schematics use a specific color coding to represent each of the logical groups that gets passed between these different algorithms. Each vector that is part of a certain logical group is assigned that corresponding color. This allows to quickly identify where vectors come from, or go to. Which color corresponds to which logical group can be found in the color legend below.



Vectors are represented as wide rectangles with B or P boxes in it, indicating that those vectors contain byte or polynomial values at each index respectively. These B or P boxes are further referred to as element boxes. In addition, the length of each vector, i.e. in the amount of bytes or polynomials in set vector, is depicted below each vector. Matrices are represented similar to vectors, but containing multiple rows of element boxes. The amount of element boxes does not correspond to the real amount of elements in each vector, as this depends on the NIST security level that is used.

SHAKE operations often produce multiple concatenated outputs, or absorb multiple concatenated input values. This concatenation in the schematics below is represented as a curly bracket, extending over the vectors to be concatenated. With the values absorbed/produced by SHAKE from left to right and top to bottom.

Some operations, like MakeHint, UseHint and SampleInBall are not depicted in detail in the schematics below, as these functions do not contribute much to the total amount of processing required for the Dilithium workload. For extra information on these functions, see [19].

Lastly, to more clearly show the dominance of multiplication and hashing operations, round boxes with **X** and # icons are added to represent the presence of polynomial multiplication and hashing (128 or 256) operations respectively.

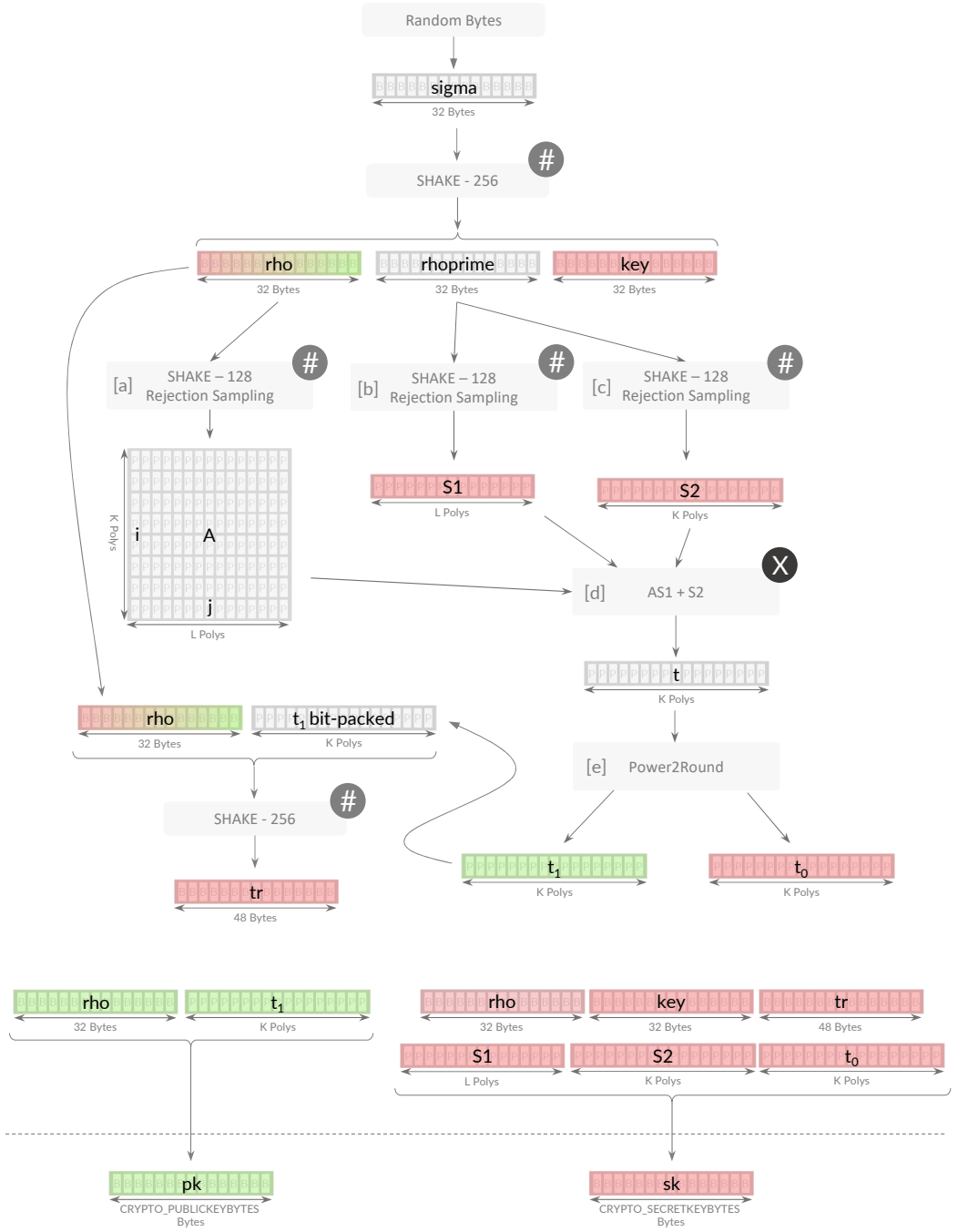


FIGURE 2.1: A schematic of the key generation part of the Dilithium algorithm shown in algorithm 5

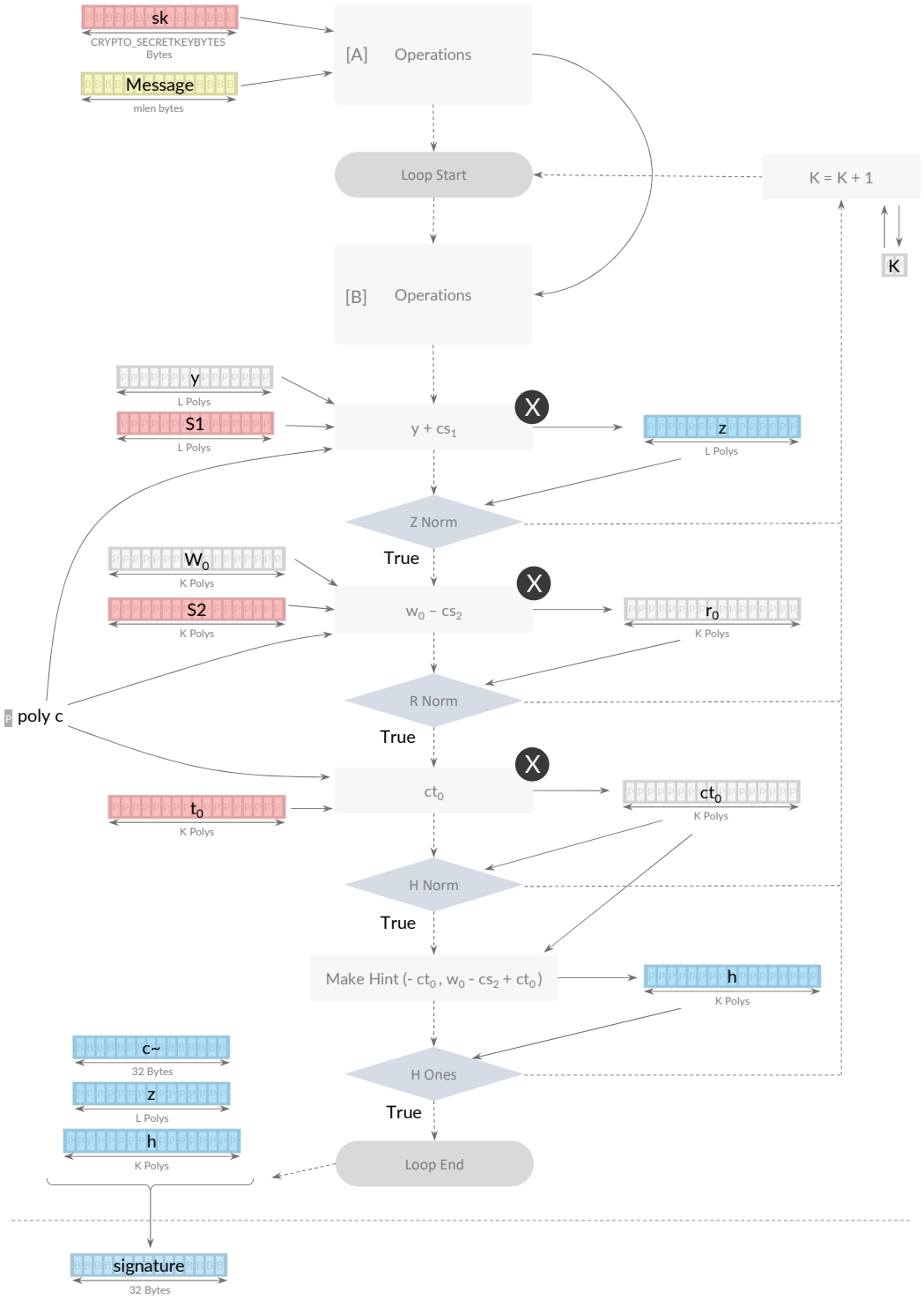


Figure 2.2: A schematic of the signature generation part of the Dilithium algorithm shown in algorithm 6

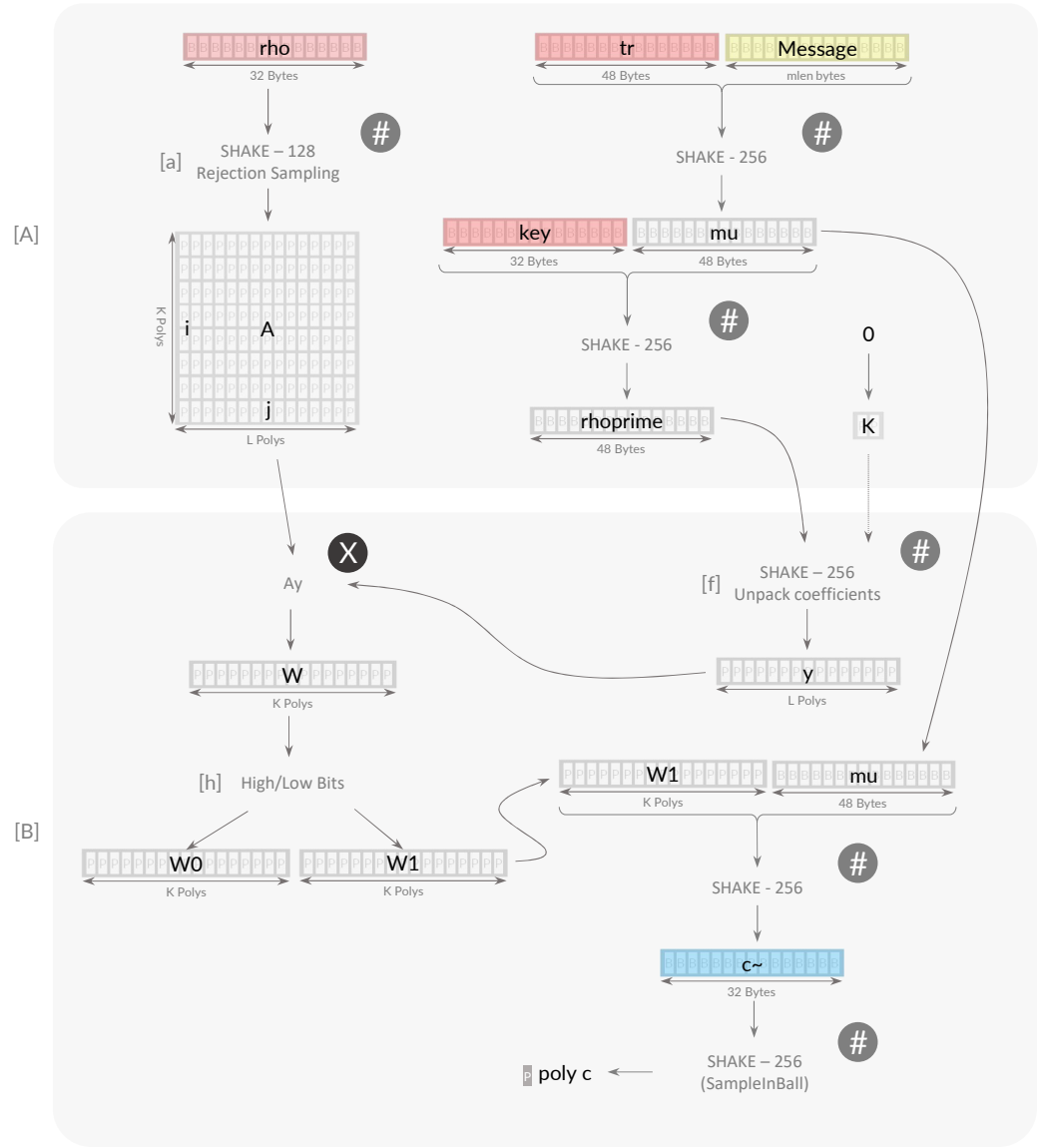


FIGURE 2.3: A schematic, detailing the operation boxes of Figure 2.2

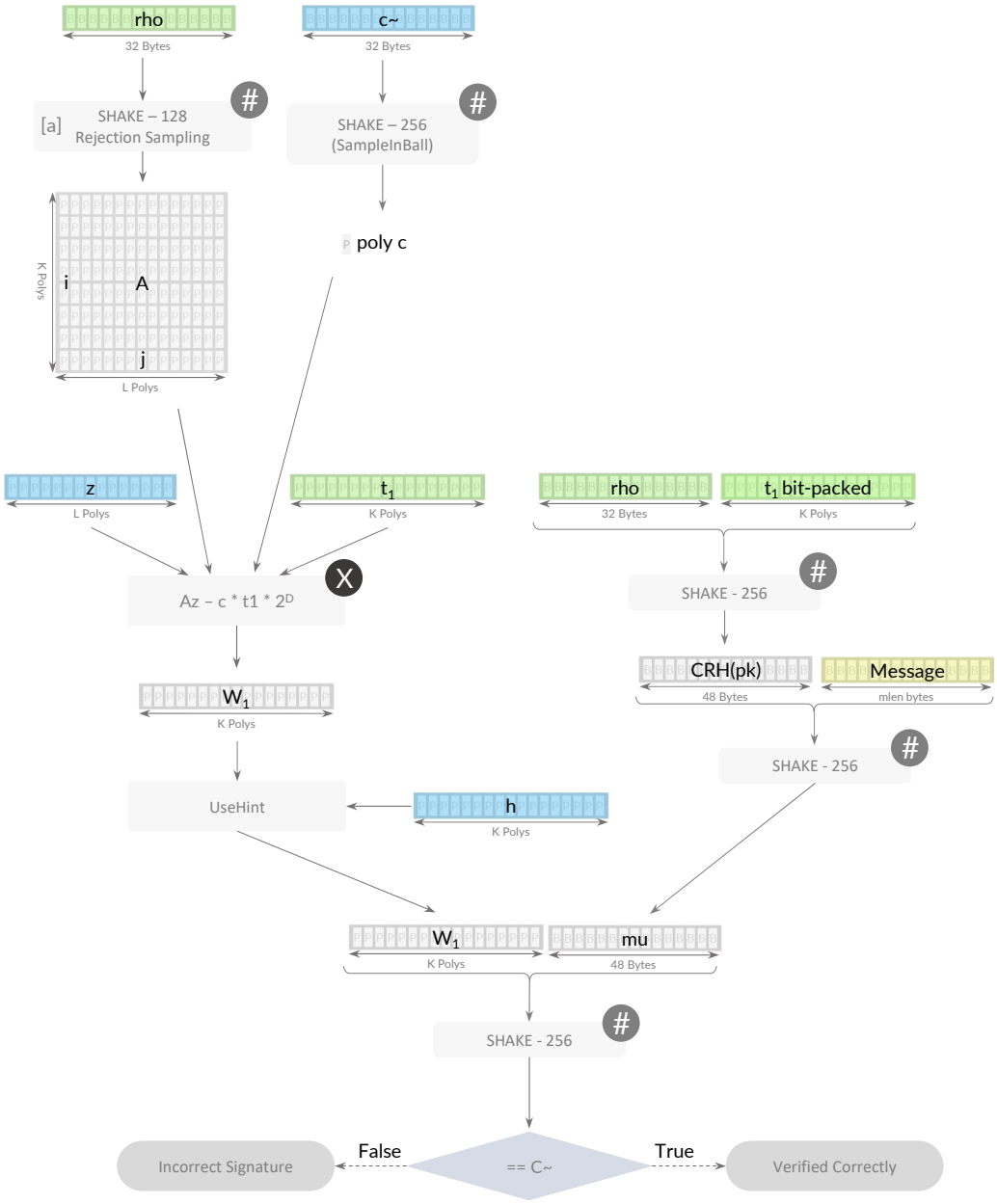


Figure 2.4: A schematic of the signature verification part of the Dilithium algorithm shown in algorithm 7

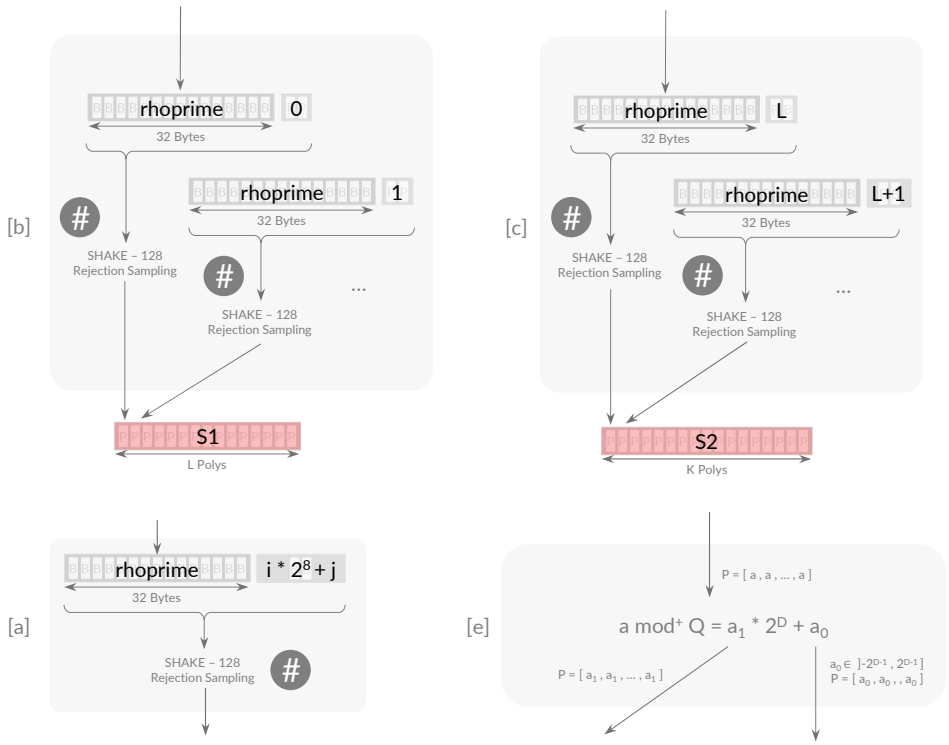


FIGURE 2.5: A schematic, detailing operation boxes of Figures 2.1, 2.3 and, 2.4

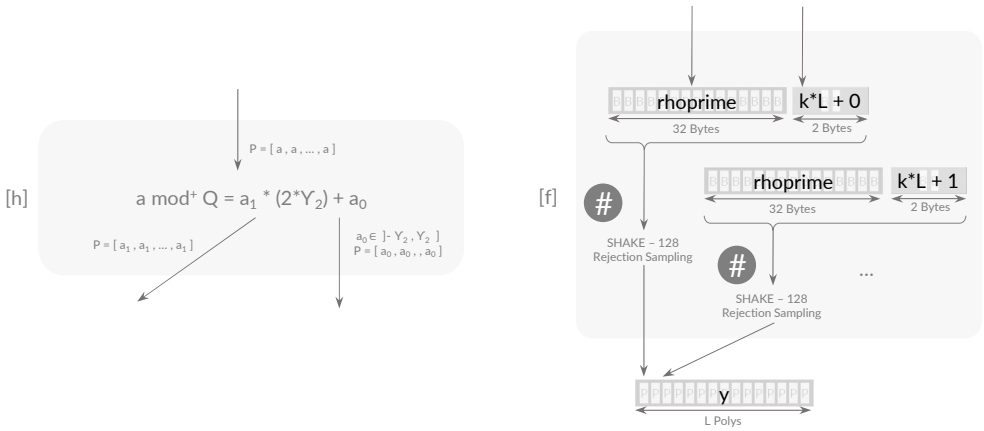


FIGURE 2.6: A schematic, detailing operation boxes specific to Figure 2.3

The schematic descriptions given above show that the three components that comprise Dilithium are all dominated by polynomial multiplication and SHAKE operations [19].

2.2.1 Polynomial Multiplication

There are two types of polynomial multiplications present in the Dilithium scheme, see appendix A or the previous chapter.

The polynomial multiplication that takes the form of a matrix-to-vector polynomial multiplication requires the most multiplications. Present as $A\vec{s_1}$, $A\vec{y}$, and $A\vec{z}$ in the Dilithium scheme for key-pair generation, signature generation, and verification respectively. This operation, when performed at the NIST Level 3 parameter setting, requires 30 polynomial multiplications. However, as this multiplication is also part of the rejection loop in the signing algorithm, it might be performed multiple times.

The other type of polynomial multiplication, i.e. polynomial-to-vector polynomial multiplication is only part of the signing algorithm, present as $c\vec{s_1}$ and $c\vec{s_2}$ in the signing algorithm. However, it is again also part of the rejection loop, effecting the worst case execution time. It should be noted that the c polynomial consists only of τ plus or minus ones. This in turn allows for more efficient implementations of this multiplication, as used in [1].

To help improve polynomial multiplications, Dilithium was designed with a polynomial ring that allows to implement polynomial multiplications using a complete NTT [19], as explained in the introduction (subsection 1.2.2).

2.2.2 SHAKE

From the schematics in appendix A, it is clear that the SHA-3 eXtendable Output Functions (XOFs) are heavily used.

The SHAKE-128 functions are used to generate the A matrix (via ExpandA), and the vectors $\vec{s_1}$ and $\vec{s_2}$ from a seed. Generating the A matrix from seed, and only storing the seed in the public key (not the A matrix), allowed to reduce the size of the public key [19]. Besides SHAKE-128, SHAKE-256 is also used a lot in all three algorithms, absorbing multiple vectors, and producing a variable length condensed representation.

2.3 Conclusion

The schematics in section 2.2 highlight that of the three Dilithium components, the signing operation requires the most processing to execute. This is not only because the signing operation contains more SHAKE and polynomial multiplication operations than the other two components, but also because a large chunk of these operations sometimes need to be calculated multiple times, due to the presence of a rejection loop. This rejection loop however, also offers some high level optimization possibilities, the subject of chapter 3.

The current chapter also showed that the Dilithium workload is mainly dominated by the SHAKE and polynomial multiplication operations. Hence, these operations are often the focus of low level optimizations, the subject of chapter 4.

Chapter 3

High Level Optimizations

This chapter goes over possible optimization tactics that could improve the overall performance of the Dilithium code. Starting to optimize from a higher level gives a good overview of the workload, and by taking on different perspectives, often allows for greater optimization possibilities. As the execution of the loop in the signature generation algorithm requires the most processing time, that is where the main focus of this section will be.

3.1 Loop Conditions

For security reasons, the signing operation makes use of rejection sampling [19]. In implementations, this rejection sampling often takes the form of a loop with four rejection conditions, which if true, jump to a new loop iteration, as shown in algorithm 4. In other words, only if all four rejection conditions are false, then will the loop terminate.

Algorithm 4 also shows that implementations defer the calculation of certain variables, to be calculated only before the conditions where they are needed. Meaning that the first invalid condition branches to the beginning of the loop again, skipping deferred operations beyond that invalid condition, and thus reducing the average processing required in each loop iteration.

Observing the behavior of the four rejection conditions is done by recording the values of all four conditions for 39000 loop iterations. For each of the four conditions, the number of times that condition evaluated to true, i.e. causing the loop to go to the following loop iteration, is counted. The results are shown in table 3.1 and figure 3.1, where the percentage refers to the percentage of iterations the corresponding condition evaluates to true. It should be noted however, that the *H Norm* and *H Ones* conditions are only valid when both the other conditions evaluate to false, and thus are not evaluated/counted when one of the *Z Norm* or *R Norm* conditions is true.

Algorithm 4 Reference Implementation [49]: Loop Conditions

while (true) **do**
... Loop Code ...

$\vec{z} = \vec{y} + iNTT(c\vec{s_1})$
if $\|\vec{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ *Z Norm* Condition
continue
end if

$\vec{r_0} = LowBits_q(\vec{w} - iNTT(c\vec{s_2}), 2\gamma_2)$
if $\|\vec{r_0}\|_\infty \geq \gamma_2 - \beta$ **then** ▷ *R Norm* Condition
continue
end if

if $\left\|iNTT(\vec{ct_0})\right\|_\infty \geq \gamma_2$ **then** ▷ *H Norm* Condition
continue
end if

$h = MakeHint_q(-\vec{ct_0}, \vec{w} - c\vec{s_2} + \vec{ct_0}, 2\gamma_2)$
if # of 1's in h $> \omega$ **then** ▷ *H Ones* Condition
continue
end if

break
end while

Condition	Security Level					
	2		3		5	
	Count	%	Count	%	Count	%
Z Norm	17892	45.88	14829	38.02	13212	33.88
R Norm	22314	57.22	26644	68.32	23835	61.12
H Norm	0	0.00	0	0.00	0	0.00
H Ones	143	0.37	32	0.08	89	0.23

Table 3.1: A table with all conditions

Security Level	K	L
2	4	4
3	6	5
5	8	7

TABLE 3.2: A table with the values for L and K

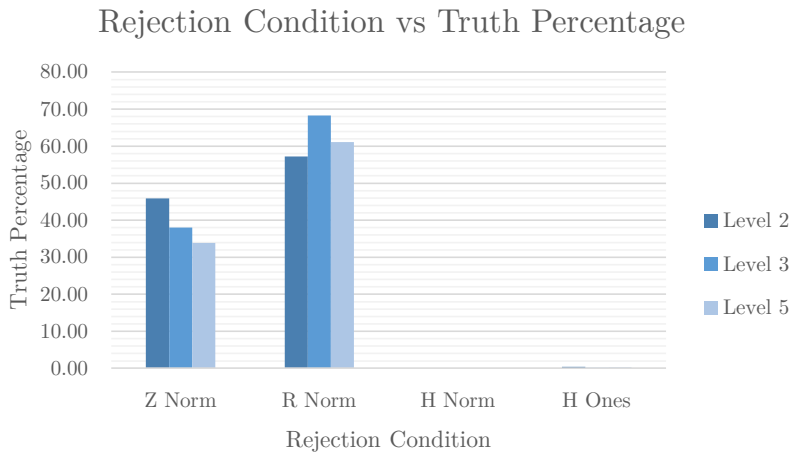


FIGURE 3.1: A bar diagram showing the percentage of iterations that each of the rejection conditions is true, as in table 3.1

From figure 3.1, it is clear that for all the security levels, for more than 99% of the iterations, the last two conditions are not the reason a certain signature would get rejected. But instead, it's the *Z Norm* and *R Norm* rejection conditions that are the cause for the bulk of the rejections. With the *R Norm* condition more often evaluating to true than the *Z Norm* condition, and the size of the difference depending on the security level.

Both the *Z Norm* and *R Norm* conditions are preceded by a similar calculation. This calculation, in both cases involves a multiplication in the NTT domain, followed by an inverse NTT transform and an addition or subtraction operation, only differing in the amount of elements the vectors to operate on contain. It should be noted that the $\vec{w_0} = LowBits_q(\vec{w})$ is already calculated in the "Loop Code", and thus $LowBits_q(x)$ does not need to be used again just before the *R Norm* condition. It is sufficient to use $\vec{w_0}$ instead of $\vec{w}$ in the calculation preceding the condition, as explained in [19]. The calculation preceding the *Z Norm* condition operates on vectors with L elements, while the calculation before the *R Norm* condition operates on vectors with K elements. The values of L and K change with the security level, as shown in table 3.2.

In algorithm 4, based on the round three submission source code of Dilithium [49], the *Z Norm* condition is evaluated first, but, as already explained, it is the *R Norm* that more often is the cause for a rejection. That is why the rest of this

Z Norm First						
Condition	Security Level					
	2		3		5	
	Evaluations	%	Evaluations	%	Evaluations	%
Z Norm	39000	100.00	39000	100.00	39000	100.00
R Norm	21108	54.12	24171	61.98	25788	66.12
H Norm	9101	23.34	7625	19.55	10037	25.74
H Ones	9101	23.34	7625	19.55	10037	25.74

Table 3.3: A table with the number of evaluations for each condition, with the Z Norm condition evaluated first (as in the original implementation)

R Norm First						
Condition	Security Level					
	2		3		5	
	Evaluations	%	Evaluations	%	Evaluations	%
Z Norm	16686	42.78	12356	31.68	15165	38.88
R Norm	39000	100.00	39000	100.00	39000	100.00
H Norm	9101	23.34	7625	19.55	10037	25.74
H Ones	9101	23.34	7625	19.55	10037	25.74

Table 3.4: A table with the number of evaluations for each condition, with the R Norm condition evaluated first

section studies the effect of swapping the *Z Norm* and *R Norm* conditions.

Using the data with the values of the four conditions collected for 39000 loop iterations, it is possible to derive tables 3.3 and 3.4. These tables show the number of times each of the conditions needed to be evaluated over 39000 iterations, if the *Z Norm* condition is evaluated before or after the *R Norm* condition respectively. Table 3.3 for example, shows that the *Z Norm* needs to always be evaluated, as it is the first condition to check, and the *R Norm* only needs to be evaluated if the *Z Norm* condition is false.

Comparing tables 3.3 and 3.4 shows that first calculating the *R Norm*, results in less overall evaluations that need to be performed. Taking into account the different sizes of the vectors used in the calculations before the *Z Norm* and the *R Norm* condition can be done by estimating the work done for the calculations by L and K respectively. Combining the work done by the calculation before the two conditions, with the percentage, indicating the chance that each of the conditions is evaluated in an iteration, allows to find an estimate of the work improvement, related to the *Z Norm* and *R Norm* conditions, that might result from swapping these conditions.

Security Level	Z Norm First Work	R Norm First Work	Improvement (%)
2	616	568	7.79
3	872	760	12.84
5	1228	1073	12.62

TABLE 3.5: A table with the improvement obtained for each security level

Security Level	Cycles (Kaby Lake)		Improvement (%)
	Z Norm First	R Norm First	
Level 2			
Average	1182687	1168759	1.18
Median	910608	910490	0.01
Level 3			
Average	1917169	1832583	4.41
Median	1552239	1517203	2.26
Level 5			
Average	2367602	2317952	2.10
Median	1956428	1887166	3.54

TABLE 3.6: A table with the test results of swapping the *Z Norm* and the *R Norm* conditions

The work related to the original implementation, i.e. the case where the *Z Norm* condition is evaluated first, with security level 5, is estimated as $(100L + 66K)$, while the work related with the case where the *R Norm* condition is evaluated first, can be estimated as $(100K + 39L)$. Working from these estimations, it is possible to find the improvement in the work required to calculate these two conditions as:

$$\frac{(100L + 66K) - (100K + 39L)}{(100L + 66K)} = \frac{(61L - 34K)}{(100L + 66K)} = \frac{(61 * 7 - 34 * 8)}{(100 * 7 + 66 * 8)} = 0.1262$$

Thus, switching the *Z Norm* and the *R Norm* condition gives an almost 13% improvement (when only looking at the pre-calculations of these two conditions) at security level five. A similar calculation is done for the other results, which are shown in table 3.5.

Unfortunately however, due to Amdahl's law, the improvements shown in table 3.5, do not translate directly to the overall performance improvement of the entire signing algorithm. Swapping the *Z Norm* and the *R Norm* conditions gives a smaller overall improvement in reality, as shown in table 3.6.

Table 3.6 is obtained by running 10000 tests using the speed test provided with the round three NIST submission for Dilithium [49], on a Kaby Lake Intel CPU. The results show that while the overall improvement in cycle-count is not as large as in table 3.5, it definitely still results in an overall improvement. Another major benefit

of this improvement is that it works for a whole range of hardware implementations, and not only for implementations on the Cortex-M55.

3.2 Conclusion

As far as the authors are aware of, the optimization introduced in this chapter is not yet present in any Dilithium implementation out there already. This might have multiple reasons, first off all, and maybe most importantly, it could be that the optimization introduces security vulnerabilities that the authors are not aware of. Another reason, and probably the most likely, could be that, while the change is fairly straightforward, in the field of hardware optimization, it is not a standard practice to check the statistical nature of return conditions.

While high level optimizations have the possibility of offering a large performance improvement, the goal of this thesis is to research how well the Armv8.1-M ISA suits the Dilithium workload. To accomplish this, it is necessary to descend to the assembly level, and research how the Armv8.1-M ISA can be used to increase performance, which is discussed in the next chapter.

Chapter 4

Assembly Optimization

Chapter 2 showed that the Dilithium workload is dominated by the SHAKE and polynomial multiplication operations. This means that whether the Armv8.1-M ISA is suited to implement the Dilithium workload, is mainly determined by whether the ISA allows for the efficient implementation of the SHAKE and polynomial multiplication operations.

With regards to the SHAKE operation, as mentioned in [25], this will most likely be implemented directly in hardware, by way of an accelerator. The reasoning given in [25] being that, the use of hardware accelerators for Cortex-M processors only introduces a small amount of overhead, as the software executing on Cortex-M processors directly runs in the physical address space. Because of this, as the authors of [25] go on to explain, accelerators for cryptography and hashing operations are already included with many Cortex-M microcontrollers.

With the SHAKE operation being delegated to hardware, the only main operation that remains to be optimized is the polynomial multiplication. As explained in chapter 1, the polynomial multiplication can be implemented in multiple different ways. From these different implementation methods, both the standard schoolbook and Toom-Cook-based polynomial multiplications are outperformed by the NTT-based polynomial multiplication [25]. On top of that, Dilithium was designed with the NTT-based polynomial multiplication in mind. Because of these reasons, we decided to focus on the NTT-based implementation of the polynomial multiplication for this thesis.

This chapter discusses the Dilithium NTT, the details of its implementation, and the improvements made using the Armv8.1-M ISA.

4.1 Reference NTT Implementation

As Dilithium is designed with the NTT-based polynomial multiplication in mind, the reference implementation already contains a basic NTT transformation implementation in c.

A schematic of how this implementation calculates the forward NTT transformation is shown in Figure 4.2, based on Figure 1.6 from chapter 1. The grey and

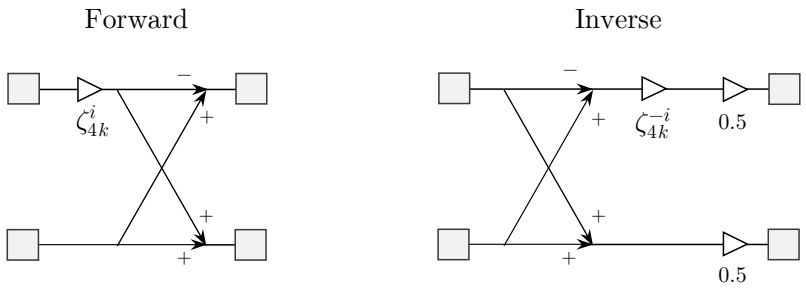


Figure 4.1: The butterfly used in the reference implementation for the forward and inverse NTT transformations

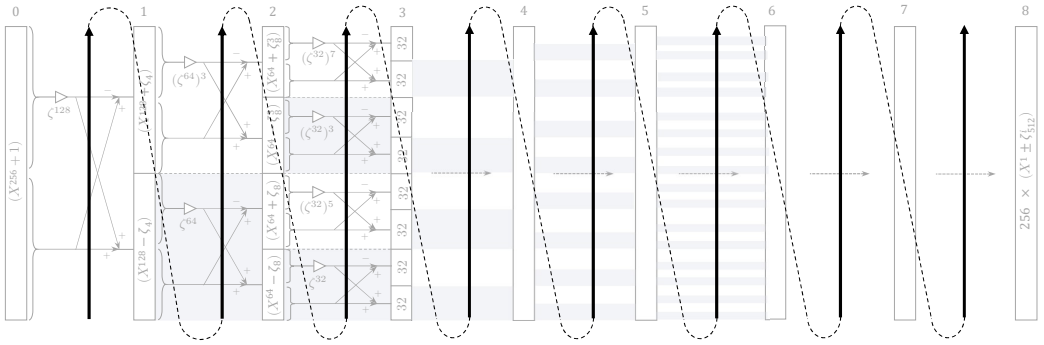


Figure 4.2: A schematic, detailing the calculation order of the reference forward NTT implementation

white stripe pattern in each of the layers in this schematic highlights the butterfly operations with a different twiddle factor. The group of operations inside one of these grey or white stripes is further referred to as a block. The first layer only contains one block, the second layer contains two blocks, the third layer four, and so on. Each layer doubling the amount of blocks from the previous layer, all the way to the last layer, containing 128 blocks, not depicted as to not clutter the schematic. In this schematic, the black arrows highlight the order in which the reference NTT implementation calculates the NTT. From this, it is clear that the reference implementation processes layer by layer, going through all the blocks in each layer, block by block, from the bottom to the top. In each block, two elements get processed each iteration, using the radix-2 butterfly operation shown in figure 4.1, which is also visible in the blocks in figure 4.2.

Before every block, a new twiddle factor is loaded, used in all the butterfly operations in that block. In the reference implementation, these twiddle factors are stored in an array, in the order they are needed, so that before every new block, the next twiddle factor can be loaded into a local variable. This twiddle factor array is indicated as array [a] in figure 4.3.

Similarly to the NTT, the reference Dilithium implementation also provides an implementation of the inverse NTT (iNTT). This implementation operates in a similar fashion to the forward NTT transformation, allowing it also to be divided into blocks. The iNTT differs from the NTT by operating in reverse order, starting with 128 blocks. Even though this 128-block calculation layer is technically the first layer to be calculated in the iNTT transformation, we will refer to it as layer eight for the remainder of this thesis, in an effort to highlight its relation with its forward counterpart. On top of that, the iNTT also uses a different butterfly operation $(c_{top}, c_{bot}) \mapsto (\zeta^{-1}(c_{bot} - c_{top}), c_{bot} + c_{top})$, frequently referred to as the Gentileman-Sande (GS) butterfly operation [31].

In the iNTT, just like in the forward NTT transformation, a new twiddle factor is loaded before every block, only this time, the inverse of the twiddle factor loaded in the forward NTT transformation is used. The reference Dilithium implementation does not store the inverse twiddle factors separately, but makes use of the twiddle factors already used for the forward NTT transformation to obtain the inverse values. By taking the negative of all the forward NTT twiddle factors, all the inverse twiddle factors can be obtained. However, the obtained inverse twiddle factors are not in the same order as the twiddle factors they were derived from, as shown in figure 4.3. To understand the permutation behavior visible in figure 4.3, take ζ^{-64} for example, which is equivalent to $-(\zeta^{64})^3$, as explained in equation 4.1, where the $\zeta_4^2 = -1 \ mod \ Q$ equivalence is derived from property 1 given in chapter 1.

$$\begin{aligned} \zeta^{64}(\zeta^{64})^3 = \zeta^{256} = \zeta_4^2 = -1 \ mod \ Q \\ \Downarrow \\ (\zeta^{64})^3 = -(\zeta^{-64}) \ mod \ Q \end{aligned} \tag{4.1}$$

This same explanation can be applied to all the other twiddle factors, effectively reversing the order of the twiddle factors in each layer. The benefit of this permutation is that by starting to read at the end of the permuted twiddle factor array, the order of the inverse twiddle factors in each layer is again flipped, resulting in the same block order as in the forward NTT. With the extra benefit that, as the iNTT starts at layer eight, the layers as a whole are already in the correct order.

While the explanation given above would be complete if the standard modular reduction technique would be used, the reference NTT implementation makes use of the Montgomery reduction technique explained in chapter 1, to reduce the result of each twiddle factor multiplication. To correct for the fact that the Montgomery reduction technique calculates $(a(2^{-32}) \ mod \ Q)$ instead of $(a \ mod \ Q)$, each of the twiddle factors stored in the array is already multiplied by 2^{32}.

As the same twiddle factor array is used in the iNTT, the 2^{32} factor is also already included in the inverse twiddle factors. The iNTT reference implementation however, after the standard iNTT transformation is complete, additionally multiplies the result with $(2^{32} \ mod \ Q)$. This is again done to compensate for a Montgomery reduction operation, this time however from the polynomial multiplication in the NTT domain between the NTT and the iNTT operations.

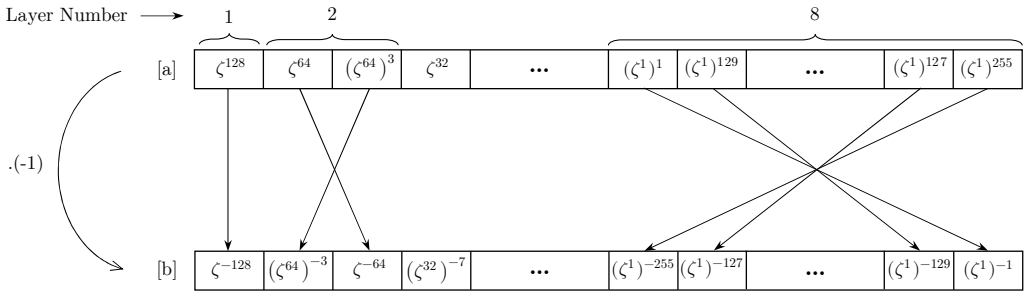


Figure 4.3: Obtaining inverted roots

4.2 Python Compiler Framework

The authors of [25] make use of a Python-based code-generation framework. They state that: "*the main tasks the framework performs are register management and address offset calculations when loading contiguous or scattered buffers*" [25]. In [25], this framework is then used to generate Armv8.1-M assembly code for specific operations from a single "compiler" python module. This "compiler" Python file than generates a file in assembly language, filling in certain offsets and register names automatically. This can also be viewed as a kind of compilation from the python framework to Armv8.1-M assembly code. One of the operations the framework was used for in [25], was the NTT transformation, which we decided would be a good place to start from, for optimizing the NTT in this thesis.

Before the Python-based framework, and associated python programs from [25] could be used, it was first necessary to understand it. To accomplish this, a UML diagram of the Python-based framework was created, as shown in figure 4.4.

This UML diagram offers multiple insights about the Python-based framework. Python classes belonging to the core of the Python framework are indicated through the use of a light header color. On the other hand, Python classes specifically related to implementations with the Helium architecture make use of darker colored headers. The UML diagram makes use of three main colors to indicate which classes are logically related to each other, as explained below:

1. **Blue classes**: Used to indicate classes that are related to registers and register allocation. In a "compiler" Python module, first an instance of the [MVE] Allocator class is included, which defines two Allocator classes, one for the GPRF, and one for the VRF. These Allocator instances are initiated with a list containing all the available register names in the Cortex-M55. Next, every time a register is needed, a Reg class instance is created, and linked to either the vector or scalar Allocator. When that register is needed in the asm code, this Reg instance then allows a register that is not yet in use to be allocated to it, using the methods from the Allocator instance that it was linked to. In the case there are no free registers left, the value of the register allocated the

longest time ago can be stored in memory, before that register is reallocated to the new Reg class instance. When that register is no longer needed, the Reg instance can deallocate the register, again using the methods from the Allocator instance linked to it.

2. **Red classes**: These classes are best explained using the following explanation given in the framework: "*Immediate offsets used in Load and Store instructions are bounded by a range specific to the instruction. For example, immediate offsets in Cortex-M Vector Extensions (MVE) are limited to the index range -127,..,127. In case random access is required for a buffer which is so large that it cannot be addressed using immediate offsets from a single base address stored in a GPR, this class helps allocating and configuring a set of 'marker' GPRs to minimally cover the given address range.*" [24].

3. **Green classes**: These classes make use of the red marker classes to load and store registers at a certain index from a large buffer in memory. The abstract Base class is extended by the [MVE] ReadWriteVector class, which allows 128-bit vectors to be read from or written to memory. In the NTT "compiler" Python module, this class is used to interact with the 256 coefficients from the input polynomial.

The partially optimized NTT assembly implementation generated by the NTT "compiler" Python module from [24], but altered for the case of Dilithium, works as shown in figure 4.5.

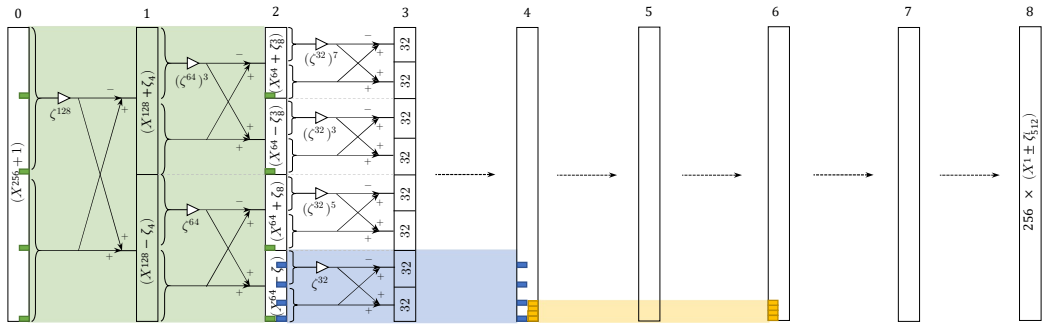


FIGURE 4.5: A schematic, detailing the working of the forward NTT implementation, generated by the NTT "compiler" Python module

Comparing figure 4.5 with the schematic of the reference NTT implementation in figure 4.2, shows that the NTT "compiler" Python module computes two layers of the Dilithium NTT at the same time. This requires three times more twiddle factors to be loaded for every butterfly operation, compared to the reference implementation that requires one twiddle factor to be loaded, more optimally making use of the available registers in the GPRF. This is especially helpful due to the large amount of scalar-vector instructions present in the Armv8.1-M ISA.

For the initial six layers of the forward Dilithium NTT transformation, two layers are processed at the same time using the combination of three radix-2 butterfly operations, depicted in figure 4.6. This butterfly operation makes use of three twiddle factors, one of the first layer, and two from the second layer that is being processed.

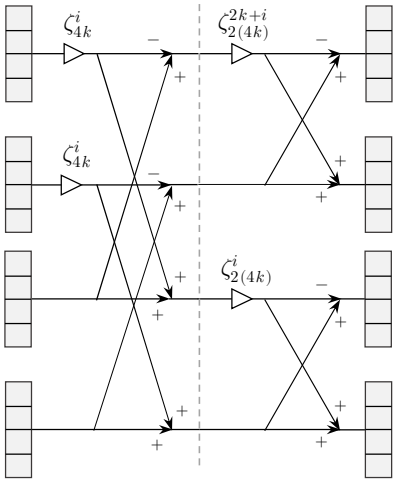


Figure 4.6: The butterfly used in the initial layers of the forward NTT implementation, generated by the NTT "compiler" Python module

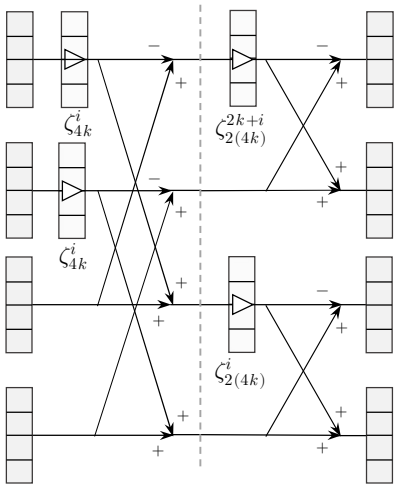


Figure 4.7: The butterfly used in the last few layers of the forward NTT implementation, generated by the NTT "compiler" Python module

Figure 4.6 also shows that every one of these butterfly operations operates on four 128-bit vector registers, filled with 4 x 32-bit coefficient values (each 32-bit value is indicated by a grey box in figure 4.6). These vector registers are also visible in

figure 4.5, where the first block that is processed in every one of the initial three layer-pairs is highlighted with a different color. For every one of these blocks, the input and output vector locations for the first butterfly operation are visualized in the layers beside the block. Every one of these vector boxes contains 4 x 32-bit coefficient values. In layer-pair three, which starts from layer four, the amount of values to be processed by every block is equal to 16. This means that, with every block using a concatenation of radix-2 butterflies from two layers, and with four input vectors containing four coefficient values each, every block can be calculated with one butterfly calculation.

While the first three layer-pairs all have block sizes that are a multiple of 16 values, allowing every block to be calculated using the butterfly in figure 4.6, without operating on half filled vector registers, this is not the case for the last layer-pair. The last layer-pair, starting at layer six, only has four input values for every block. For this reason, the last layer-pair makes use of the butterfly operation shown in figure 4.7.

The butterfly operation shown in figure 4.7 works by processing four blocks at the same time, which is accomplished by also storing the twiddle factors in a vector. Every vector register used then contains the values of four blocks.

4.3 The search for stalls

Instead of checking for sequences of instructions that could cause stalls to occur by hand, this thesis introduces a Python program that scans assembly files, and highlights some of the locations where stalls occur. To better understand how this Python program works, we again created a UML diagram of the classes involved, shown in figure 4.8.

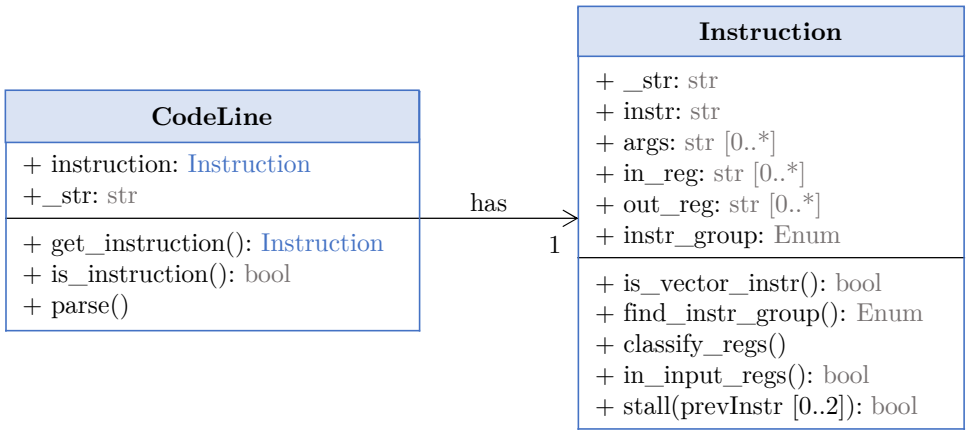


FIGURE 4.8: A UML diagram of the Python-based stall checker module

The stall checker Python program works by making a CodeLine instance for

every line in an assembly file. This CodeLine instance than parses the code line, and if relevant (i.e. an assembly instruction is present), extracts the assembly instruction name, and the arguments passed to it. This generated Instruction instance is then used to verify whether any stalls would occur, given the previous two instructions.

Currently, this Python module supports all instructions used in the assembly implementation generated by the original NTT Python "compiler", explained in the previous section. The following instructions are currently included: add, bx, ldr, ldrd, pop, push, vadd, vld40-4, vldrw, vmul, vpop, vpush, vqrdmlah, vqrdmulh, vstrw, and vsub. This list is easily extendable. For these instructions, the stall checker is already able to estimate where stalls might occur due to the presence of a structural hazard, i.e. the use of the same hardware by two consecutive instructions. This is done by grouping all the instructions in three separate instruction groups (manually); load/store instructions, addition/subtraction/logical instructions, or multiply/floating-point instructions. This grouping is then used to detect stalls, by verifying whether or not two successively executed instructions belong to the same instruction group, indicating that a structural stall is present or not.

Multiplication operations have a latency of 2 clock cycles, referring to the fact that the output of a Helium SIMD vector multiplication operation is only available after two clock cycles. The stall checker is also able to estimate stalls that occur as a result of a data hazard, generated by an instruction using the output of a multiplication operation in the next clock cycle. While true for all other instructions, if the instruction that is executed immediately after the multiplication instruction is a vector store instruction, then no stall will be present, and hence is excluded from the previous stall condition in the stall checker. Lastly, the stall checker also checks for store-load related stalls.

For every stall, a comment is added to an annotated version of the original asm file, and a warning is written to the stdout stream. While the current version of the Python stall checker program is able to find stalls in the assembly NTT implementation generated by the original NTT Python "compiler", explained in the previous section, this will only approximate the real behaviour, and can still be improved upon. For example, detecting stalls that occur at the end or the beginning of a loop is not yet supported.

4.4 The MPS3 FPGA Prototyping Board

In order to benchmark the optimizations on the Cortex-M55 processors, we make use of the Arm®MPS3 FPGA Prototyping Board, revision HPI-0309C. The Arm®MPS3 FPGA Prototyping Board will further be referred to as the MPS3.

4.4.1 Setup

The code that needs to run on the MPS3, with the Application Note 552 (AN552) FPGA image installed [7], is compiled using the Arm GNU Toolchain (version 10.3) [9], and makes use of the Common Microcontroller Software Interface Standard (CMSIS) [6], and a CMSIS support package for the MPS3 [15]. Arm defines CMSIS

as "*a vendor-independent abstraction layer for microcontrollers that are based on Arm Cortex processors. CMSIS defines generic tool interfaces and enables consistent device support. The CMSIS software interfaces simplify software reuse, reduce the learning curve for microcontroller developers, and improve time to market for new devices.*" [6].

The compilation process is implemented using a makefile. This makefile offers the option to compile with one of three files, each implementing a different main function, depending on the functionality that is required:

1. **speed.c**: This file measures the execution time (in clock cycles) for each of the different Dilithium components. It also measures the execution time of some relevant sub-components, namely: the NTT transformation, the iNTT transformation, a single polynomial multiplication, and the matrix-vector polynomial multiplication. In addition, it also offers the possibility to measure the execution time using one of three methods: using the Systick Timer, using a timer peripheral, or using the Data Watchpoint and Trigger (DWT) debug unit, by reading from the Cycle Count Register. These timing methods are based on an example extended "hello world" program provided by my supervisors.

2. **verify.c**: This file verifies whether the Dilithium components, and some of the relevant sub-components, still produce the correct result. Most of the code here is copied from the original verification code, i.e., the round three Dilithium submission reference code [49]. If needed, this file can also be altered to read the data from known-answer-test header files, created by another custom program, which stores input and the corresponding output values generated by the reference Dilithium implementation [49].

3. **debug.c**: This file provides the functionality to help debug the Dilithium NTT. It runs both the original reference NTT from [49], and the NTT version that needs to be debugged. With the help of print statements, this allows intermediate values to be compared, in order to find where the problems with the custom NTT implementation are located.

The software on Cortex-M processors typically does not run on a full-fledged operating system, as these processors do not have a Memory Management Unit (MMU). For this thesis, we focus on a bare-metal implementation, and thus, the tasks typically handled by the operating system, need to be handled by the programmer. One example of this can for example be found in the extended "hello world" program provided by my supervisors, which retargets the internal c "_write()" function, used by the "stdio.h" library's "printf()" function, to not write to a file, but to write over the UART interface that is connected via the MPS3's serial debug interface to a laptop, which can then be used to read what is printed.

To help with debugging, we also needed to implement some fault handlers, which get called whenever a fault is encountered. The fault handler we implemented works by printing the stack frame, and the contents of all the fault registers over the UART interface. It also partially parses the 32-bit fault register values, to reduce the time spent sifting through the documentation, and finding what each bit value represents.

4.5 Conclusion

This chapter goes over everything related to low level optimization that we were able to work on, before we had to fully focus on writing this thesis. In its current state, we are able to run the Dilithium workload on the MPS3, with partial debugging support added by the fault handlers that print values over the serial debug interface. A plan through which the content of this chapter can be improved upon is the explained in chapter 5.

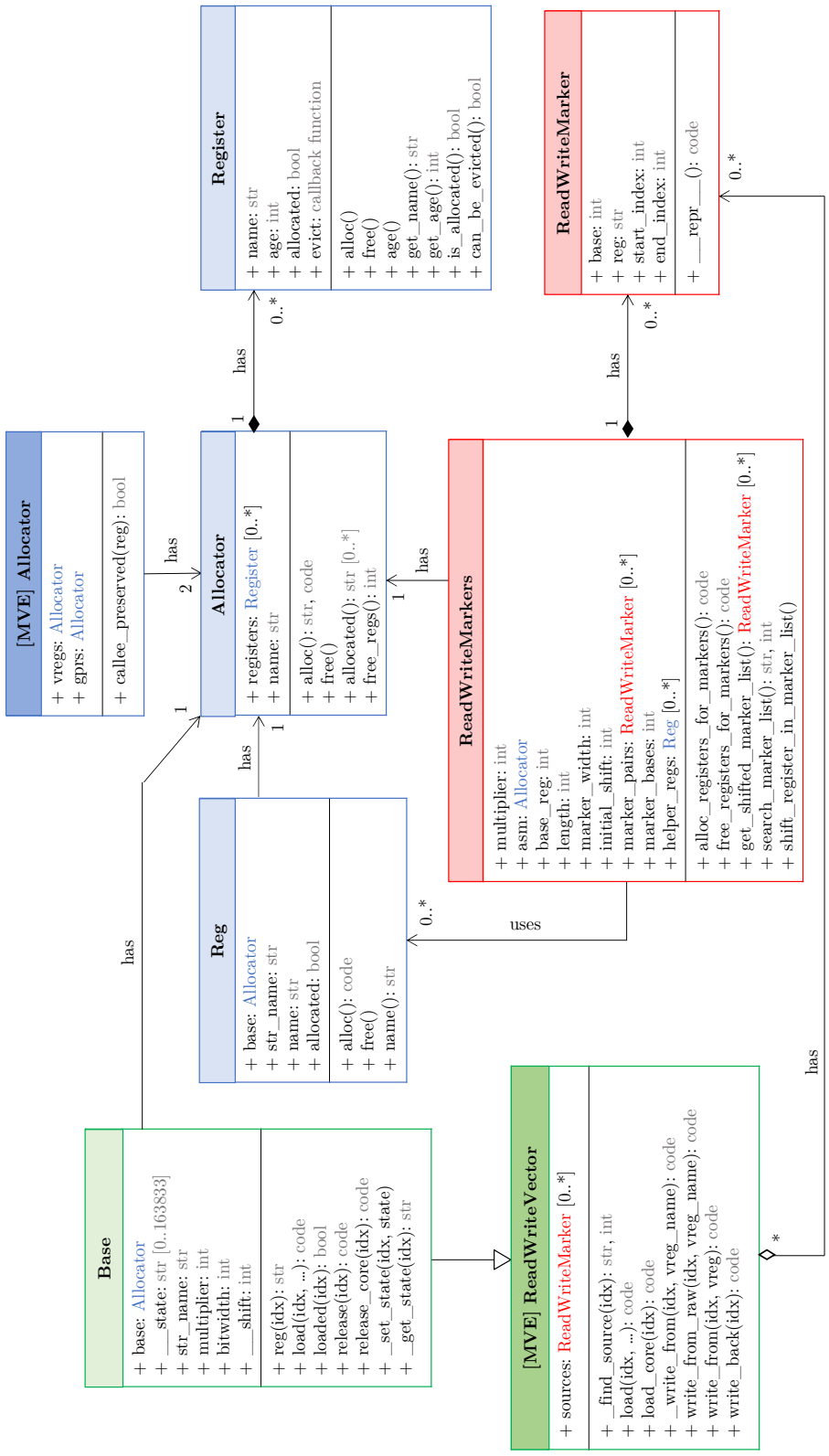


FIGURE 4.4: A UML diagram of the Python-based framework

Chapter 5

Conclusion

In this thesis, we studied the Dilithium workload, creating some visual alternatives for the algorithmic descriptions generally found in literature. Initially, our focus was on finding some high level optimizations in the Dilithium workload. Where we found that, by changing the order of two rejection conditions in the rejection loop, we were able to improve the average Dilithium execution time by 1.18%, 4.41%, and 2.1% for the NIST security level 2, 3, and 5 respectively. With the extra benefit that this optimization is not specific to any hardware architecture.

Next, we worked on low level optimizations on the Cortex-M55, using the Armv8.1-M ISA. For this, we focused on optimizing the Dilithium NTT on the Cortex-M55, using the SIMD capabilities of the Helium vector extension. While we were eventually able to run the Dilithium workload with an assembly implementation of the Dilithium NTT, generated by the Python-based framework used in [25], on the Arm MPS3 FPGA prototyping board, we did not have the time to come up with further improvements.

Even though I was not able to complete all the goals I started with, especially related to the hardware side of things, I was able to find some improvements on the algorithmic level, and learned a lot from this thesis.

To fully answer the research question, chapter 4 would still need to be further worked out. However, to not leave this thesis's research question fully unanswered, below we try answering it with what we have already learned thus far.

While the initial research question referred to PQC workloads in general, as stated in chapter 1, NIST released a statement that they already intend to standardize CRYSTALS-Dilithium and CRYSTALS-KYBER for most use cases [36]. This means that the Dilithium workload will represent a large part of the digital signature PQC workloads in the future. Thus, even though this thesis mainly focuses on Dilithium, the results will most likely give a good indication of whether the Armv8.1-M ISA is suited to implement the most frequently used standardized PQC workloads in the future.

The Armv8.1-M ISA distinguishes itself from other Arm processors in the following ways. Compared to its Cortex-M predecessors, it provides a larger amount of SIMD capabilities. When comparing its SIMD capabilites to those for the A-

profile processors, we find that it offers a much larger variety of scalar-vector SIMD instructions, but, as it is part of the M-profile processors, it only offers eight 128-bit vector registers. As shown in chapter 4 however, based on the NTT implementation of the NTT Python "compiler" module from [25], we can see that even though there are less vector registers available than on Neon architectures, the NTT can still be implemented without the need to write intermediate vector register values, for a single butterfly operation, back to memory. The generated assembly NTT implementation also shows that it is possible to make frequent use of the scalar-vector operations.

Compared to other Arm processors, as stated in chapter 1, the Armv8.1-M ISA makes the user responsible for ordering instructions to make optimal use of the SIMD instruction overlapping. While this gives the software programmer an extra burden to think about, it can be made easier on the programmer with the design of custom tools that detects these stalling instructions, a prototype of which we created in chapter 4. Also in chapter 4, the assembly NTT implementation generated by the Python-based framework indicates that it is possible to order the assembly instructions to generate very little stalls in the execution pipeline.

Thus, I would say that the Cortex-M55, implementing the Armv8.1-M ISA, is suited to the Dilithium workload, in that it allows for the speedup of the Dilithium workload over the Cortex-M4, by using SIMD to speed up the NTT transformation. However, this does not account for the SHAKE operation, which we here assume to be implemented using a hardware accelerator in most cases.

We are also not yet able to specify to exactly what degree the Dilithium workload is able to make use of the SIMD extension for the Cortex-M55.

In hindsight, choosing a PQC thesis, while having very little knowledge of cryptography in general, might not have been the best choice to start with. However, I do not regret choosing this thesis subject, as I did learn a lot about embedded systems development, (post-quantum) cryptography, cryptography related coding practices and even Arm licensing practices. On top of that, I also got some first hand experience on working with an FPGA prototyping board. Finally, I'm also glad to have overcome numerous hurdles (lack of publicly available documentation, some licensing issues, longer than expected thesis writing time, etc.), better preparing me for industry.

The rest of this chapter starts by categorizing the time spent on this thesis, followed by a section offering some suggestions for future research.

5.1 Time Management

Throughout this year, every single day I filled in an excel time-sheet, which I created in the beginning of the first semester, noting what I did for the thesis that day, and the total amount of time I spent on it. These time measurements were initially done by hand, writing down the starting time of each thesis working session, and the stop time of that same session. Eventually, to make this timing process more streamlined, less tedious, and to get more experience with the c-language, I created a c program

which allows, just by pressing the F4 key, to record the start and stop time of a certain session, exactly recording the amount of time elapsed during that session.

Using this excel time-sheet, I estimated the total amount of hours spent on this thesis to be around 688 hours. This total amount of time can be divided into multiple categories, also using the excel time-sheet, as done in figure 5.1.

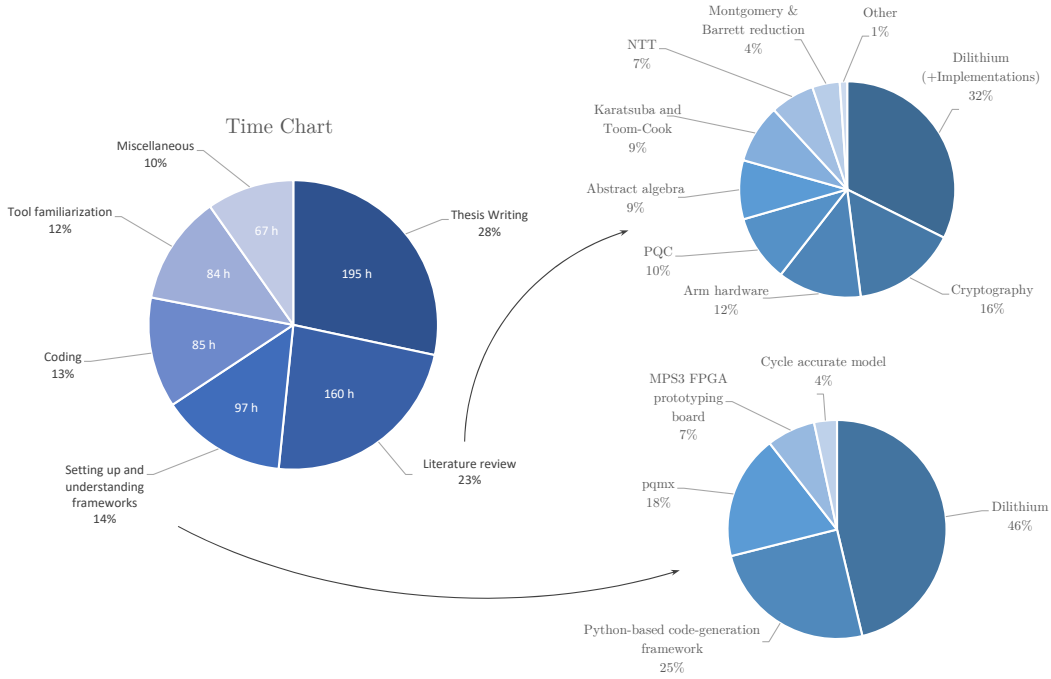


FIGURE 5.1: A pie-chart, categorizing where my time went, with two extra pie charts detailing two of the main categories

The following categories are present in the time chart pie-diagram of figure 5.1:

1. **Literature review**: Before being able to work on this thesis, it was necessary to read some of the literature out there already. This includes reading mainly academic papers, but also white papers and developer blogs for example. Figure 5.1 also gives a time distribution of the subjects I spent time reading literature about.

2. **Thesis writing**: This indicates the time spent writing the thesis, and making figures used in the thesis, of which the time spent is not classify-able elsewhere (understanding frameworks for example).

3. **Setting up and understanding frameworks**: Throughout the entirety of this thesis, I needed to work in certain already existing frameworks. Before being able to work with these frameworks, they first needed to be set up on a local computer, and I had to have a sufficient amount of understanding of the total structure. Similar to the literature review category, figure 5.1 shows the

distribution of the time spent working on setting up and understanding each framework.

4. **Coding**: The coding category reflects the actual amount of time spent creating new tools, programs or tests. This includes coding in multiple languages, like c, Python or Arm assembly.

5. **Tool familiarization**: This thesis required me to refresh how to use certain tools or languages, examples of which include Git version control, the c programming language, the commonly used arm assembly instructions, and Linux. It also required me to learn to work with multiple tools or languages I was not yet familiar with. Examples of which include the Arm assembly instructions specific to the Armv8.1-M ISA (mainly MVE instructions), the Arm MPS3 prototyping board, and multiple Linux specific commands or tools, like the Makefile and associated make tool for example, or the screen command to establish a serial debug connection with the MPS3 FPGA prototyping board.

6. **Miscellaneous**: This category includes all the time spent on tasks that do not fit in the categories above. This includes the time spent planning, setting up mails, looking for alternative implementation platforms, and attending a cryptography conference.

5.2 Future Research

5.2.1 Work in progress

As mentioned before, this thesis is still a work in progress. This is because we were only able to get the Dilithium workload working with the generated assembly Dilithium NTT implementation, shortly before we had to focus on writing the thesis.

Thus, this thesis can still be improved upon by completing the assembly level optimizations part. Carrying this task out can be done by starting to document precisely which sequence of instructions results in a pipeline stall. Which can be accomplished by creating a program that finds the execution time of two instructions separately, and then finds the execution time of these same instructions executed in sequence. This information could than be used to update the stall checker Python module. Which could in turn then be used to find a way to optimally make use of all the scalar or vector registers and the arithmetic logic units on the Cortex-M55. Either through reordering instructions, using different instructions, or calculating the NTT in a different fashion, more suited to the Cortex-M55. A more concrete example might be to change the Montgomery reduction to a form of Barrett reduction, and finding out if this change would allow for a reduction in execution time, as suggested by Hanno Becker.

5.2.2 Suggestions for Future Research

More on the practical side of things, I would advice future research done on new Arm devices to consider using the IDEs provided by Arm, with built-in support for the embedded device development, and other Arm tools. Mainly because I had a lot of trouble understanding and finding documentation for all the tools I was using, and getting everything up and running.

The original plan for this thesis, of trying to increase the energy efficiency of the Dilithium workload might be a good topic for future research. This is especially useful in embedded devices for the Internet Of Things market, as these type of applications often have less strict time requirements, but do need to be energy efficient, as they often run on batteries.

Additionally, the next two subsections describe some related research topics, that could be the subject of future research.

Ethos-U55

The Ethos-U55, already touched upon in chapter 1, is referred to as a microNPU, designed to speed up Machine Learning (ML) workloads. More specifically, it allows to speed up Convolutional Neural Network (CNN) workloads, which mainly consist of 2D convolution operations between tensors. Related to this, chapter 1 shows how the polynomial multiplication operation can also be viewed as a convolution operation, only in one dimension.

This made us curious, whether it would be possible to take advantage of the Ethos-U55 microNPU to more efficiently compute the polynomial multiplication. Meaning that the polynomial multiplication is calculated using a variant of the school-book technique, and not using the NTT-based multiplication, thus requiring more operations to be processed in total. This however does not mean that it would be slower, as the Ethos-U55 allows for a much larger degree of parallelization, and would be able to make use of spacial and temporal reusability.

One major road block in being able to use the Ethos-U55 for polynomial multiplication, in the case of Dilithium, is the fact that the Ethos-U55 only supports 8-bit weights, and 8 or 16-bit activations. While CNN applications allow the bit-size to be reduced, as they can be retrained to work at lower bit-sizes, the Dilithium workload requires all operations to be done modulo 8380417 ($2^{23} - 2^{13} + 1$), which requires at least 23-bit arithmetic operations. Whether or not a solution can be found, we leave as a question for future research.

Light Loops

Chapter 2 explains the Dilithium workload, describing the three component algorithms that make up Dilithium, of which, it is the signing operation that takes up most of the processing time. This can be partially attributed to the rejection loop, which requires the same calculation to be done over and over again, until none of the rejection conditions are satisfied.

Z Norm First				
Condition	Threshold	Security Level		
		2	3	5
Z Norm	$\gamma_1 - \beta$	0x0001FFB2	0x0007FF3C	0x0007FF88
R Norm	$\gamma_2 - \beta$	0x000173B2	0x0003FE3C	0x0003FE88

Table 5.1: A table with the threshold values used for the two main norm rejection conditions, and for all security levels

This leads to the following idea: would it be possible to create a so called, "light" version of the rejection loop, that only calculates what is necessary to execute the rejection conditions, and verify whether to reject the iteration or not. Eventually, when a "light" loop iteration passes all the rejection conditions, a single iteration of the full loop can be run, to obtain the full Dilithium signature.

As explained in chapter 3, more than 99 percent of the loop rejections are caused by the *Z Norm* and *R Norm* conditions. These two rejection conditions check whether the sup-norm of a polynomial vector, i.e. the largest coefficient in any of the polynomials in the vector, is larger than a certain fixed (but dependent on the security level) value. The value of these two threshold values is given in table 5.1.

As these threshold values only require 19-bits to represent, the idea would be to only calculate using 19-bit values, but with an overflow carry bit. The problem with this is that the Arm architecture only supports operations on 8, 16, 32 or 64 bit values.

Another problem is that, on average, only four rejection loop iterations are done, while an extra full iteration loop needs to be added when using light loops. This means that, to get a better average performance, the light rejection loop operation needs to be at least four times as fast as the original full loop implementation. On the other hand, Cortex-M applications often need to be highly-deterministic, and are sometimes time sensitive. In this type of environment, the worst case execution time of an application is an important metric. The addition of "light" loops, of which the effectiveness increases with the amount of rejected iterations, will likely be able to reduce the worst case time needed to sign a message.

Again, just like for the previous research topic, whether or not a solution exists that allows these calculations to be done using 16-bit values for example, we leave as the topic of future research.

Custom instructions

The Armv8.1-M ISA also allows for the addition of custom instructions. Finding which instructions would offer the greatest advantage to speed up the Dilithium workload, could be a topic of future research.

Appendices

Appendix A

The Dilithium Algorithm

This appendix elaborates on the key-pair generation, signature generation, and signature verification algorithms that together make up Dilithium. More specifically, it gives some variants of the algorithmic descriptions given in [19].

A.1 Algorithm Descriptions

Algorithm 5, 6, and 7 show the descriptions of the Key-pair generation, Signature generation, and Signature verification algorithms respectively, that together make up the Dilithium scheme. Where S_η denotes all elements $w \in R$, but with $\|w\|_\infty < \eta$; H denotes a hashing operation; $\|$ denotes a concatenation operation; and $\|.\|_\infty$ the sup-norm [23][19].

The expand functions (ExpandS, ExpandA, and ExpandMask), generating values from a seed, are depicted more in depth in section 2.2, with an explanation given in [19]. For an explanation of the MakeHint, Power2Round, HighBits, LowBits, and hint functions (MakeHint, UseHint) see [19].

Algorithm 5 Key-Pair Generation [23][19]

Output: $sk = (\rho, K, tr, \vec{s_1}, \vec{s_2}, \vec{t_0})$
Output: $pk = (\rho, \vec{t_1})$

1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} \leftarrow H(\zeta)$
3: $(\vec{s_1}, \vec{s_2}) \in S_\eta^l \times S_\eta^k \leftarrow ExpandS(\rho')$
4: $NTT(A) \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
5: $\vec{t} \leftarrow A\vec{s_1} + \vec{s_2}$
6: $(\vec{t_1}, \vec{t_0}) \leftarrow Power2Round(\vec{t})$
7: $tr \in \{0,1\}^{256} \leftarrow H(\rho || \vec{t_1})$

Algorithm 6 Signature Generation [23][19]

Input: $sk = (\rho, K, tr, \vec{s_1}, \vec{s_2}, \vec{t_0})$
Input: $Message\ M \in \{0,1\}^*$
Output: $Signature\ \sigma = (\vec{z}, \vec{h}, \tilde{c})$

$NTT(A) \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
$\mu \in \{0,1\}^{512} \leftarrow H(tr||M)$
$\rho' \in \{0,1\}^{512} \leftarrow H(K||\mu)$
$\kappa \leftarrow 0$
while True **do**
    $\vec{y} \in S_{\gamma_1}^l \leftarrow ExpandMask(\rho', \kappa)$
    $\kappa \leftarrow \kappa + l$
    $\vec{w} \leftarrow A\vec{y}$
    $\vec{w_1} \leftarrow HighBits_q(\vec{w}, 2\gamma_2)$
    $\tilde{c} \in \{0,1\}^{256} = H(\mu||\vec{w_1})$
    $c \leftarrow SampleInBall(\tilde{c})$

    $\vec{z} = \vec{y} + iNTT(c\vec{s_1})$
    if $\|\vec{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ *Z Norm* Condition
        continue
    end if

    $\vec{r_0} = LowBits_q(\vec{w} - iNTT(c\vec{s_2}), 2\gamma_2)$
    if $\|\vec{r_0}\|_\infty \geq \gamma_2 - \beta$ **then** ▷ *R Norm* Condition
        continue
    end if

    if $\left\|iNTT(c\vec{t_0})\right\|_\infty \geq \gamma_2$ **then** ▷ *H Norm* Condition
        continue
    end if

    $h = MakeHint_q(-c\vec{t_0}, \vec{w} - c\vec{s_2} + c\vec{t_0}, 2\gamma_2)$
    if # of 1's in h $> \omega$ **then** ▷ *H Ones* Condition
        continue
    end if

    Break
end while

Algorithm 7 Signature Verification [23][19]

Input: $pk = (\rho, \vec{t_1})$
Output: $Message\ M \in \{0,1\}^*$
Output: $Signature\ \sigma = (\vec{z}, \vec{h}, \tilde{c})$

1: $NTT(A) \in R_q^{k \times l} \leftarrow ExpandA(\rho)$
2: $\mu \in \{0,1\}^{512} \leftarrow H(H(\rho||\vec{t_1})||M)$
3: $c \leftarrow SampleInBall(\tilde{c})$
4: $\vec{w_1'} \leftarrow UseHint_q(\vec{h}, A\vec{z} - c\vec{t_1}.2^d, 2\gamma_2)$
5: **return** $\left[\tilde{c} = H(\mu||\vec{w_1'})\right]$ **and** $[\|\vec{z}\|_\infty < \gamma_1 - \beta]$ **and** $\left[\#\ of\ 1's\ in\ \vec{h}\right]$

Appendix B

The Article

Optimizing Dilithium Using the Helium Vector Extension

Kevin Orbie, **Supervisor:** Prof. dr. ir. Ingrid Verbauwhede,
Mentors: Dr. ir. Angshuman Karmakar, Ir. Jose Maria Bermudo Mera, Dr. Hanno Becker

***Abstract*—We investigate the workload of the Dilithium signature scheme, one of the winners of the NIST Post-Quantum Cryptography project, and research whether the Cortex-M55 processor, and the Armv8.1-M ISA that it implements is suited to its execution. In the process, we find a higher level optimization technique, not constrained to the Cortex-M55 processor, that improves the average Dilithium execution time by 1.18%, 4.41%, and 2.1% for the NIST security level 2, 3, and 5 respectively. We also provide a visual representation of the Dilithium workload, as an alternative to the frequently used algorithmic representation from the Dilithium specification document [1]. Additionally, we also give a detailed explanation of the c-code NTT implementation from the reference CRYSTALS-Dilithium code [2], and the assembly NTT implementation generated using the Python-based code-generation framework also used in [3].**

***Index Terms*—Post-Quantum Cryptography (PQC), CRYSTALS-Dilithium, Cortex-M55, Helium Vector Extension, M-profile Vector Extension (MVE), Armv8.1-M Instruction Set Architecture (ISA), Number Theoretic Transform (NTT).**

I. Introduction

THE current communication security is dependent on the assumption that the factorization of large integers and the discrete logarithm problem can not be solved in polynomial time. However, this assumption stands to be broken by future iterations of quantum computers, which, using Shor's algorithm [4], would be able to break some of the most commonly used cryptosystems, like the public-key RSA cryptosystem, and the ECDH key exchange mechanism.

That is why in 2016 [5] the National Institute of Standards and Technology (NIST) "*initiated a process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms*" [6]. These new cryptography algorithms are aptly named Post Quantum Cryptography (PQC) algorithms. NIST plans to standardize not only Public-Key Encryption but also Digital Signature and Key Establishment Protocol cryptographic algorithms, due to their vulnerability. In July of 2022 NIST announced that, among the finalists, it already intends to standardize the CRYSTALS-Dilithium scheme as the primary digital signature scheme to be "*implemented for most use cases*" [7].

With the rise of Internet-of-Things (IoT) devices, and the increasing importance of user privacy, the next generation of cryptographic algorithms will need to be able to efficiently run on embedded processors. As Arm, with its M-profile of the Arm architecture, is a market leader in the embedded computing market worldwide, it is important that research is done on whether the devices it offers are suited to the execution of PQC algorithms. While the PQC performance on the Cortex-M4 processor has already been extensively studied in literature [8] [9], the more recent Cortex-M55 processor released by Arm, with the Helium vector extension, has not yet received a lot of attention.

II. CRYSTALS-Dilithium

Dilithium is a lattice based digital signature scheme, that operates over the polynomial ring $R_q = \mathbb{Z}_q/(X^{256}+1)$ with $q = 2^{23} - 2^{13} + 1 = 8380417$ [1]. It consists of three sub algorithms; key-pair generation, signature generation, and signature verification. All three components of the Dilithium workload are dominated by both hashing and polynomial multiplication operations.

A. Polynomial Multiplication

There are two types of polynomial multiplications present in the Dilithium scheme. The polynomial multiplication that takes the form of a matrix-to-vector polynomial multiplication requires the most multiplications. Present as $A\vec{s_1}$, $A\vec{y}$, and $A\vec{z}$ in the Dilithium scheme for key-pair generation, signature generation, and verification respectively. This operation, when performed at the NIST Level 3 parameter setting, requires 30 polynomial multiplications. However, as this multiplication is also part of the rejection loop in the signing algorithm, it might be performed multiple times.

The other type of polynomial multiplication, i.e. polynomial-to-vector polynomial multiplication, is only part of the signing algorithm, present as $c\vec{s_1}$ and $c\vec{s_2}$ in the signing algorithm. However, it is again also part of the rejection loop, effecting the worst case execution time. It should be noted that the c polynomial consists only of τ plus or minus ones. This in turn allows for more efficient implementations of this multiplication, as used in [10].

To help improve polynomial multiplications, Dilithium was designed with a polynomial ring that allows to implement polynomial multiplications using a complete NTT [1]. Figure 1 shows a schematic of the full Dilithium NTT, transforming a 256 coefficient polynomial over eight layers.

B. Hashing operation

Dilithium also heavily makes use of the SHA-3 eXtendable Output Functions (XOFs).

The SHAKE-128 functions are used to generate the A matrix (via ExpandA), and the vectors $\vec{s_1}$ and $\vec{s_2}$ from a seed. Generating the A matrix from seed, and only storing the seed

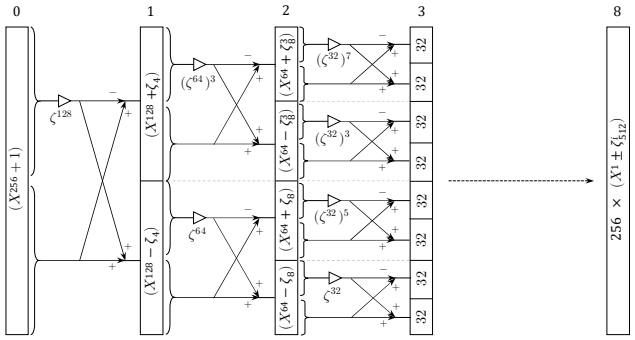


Fig. 1: Full Dilithium NTT transformation diagram.

in the public key (not the A matrix), allowed to reduce the size of the public key [1]. Besides SHAKE-128, SHAKE-256 is also used a lot in all three algorithms, absorbing multiple vectors, and producing a variable length condensed representation.

III. Hardware Preliminaries

A. Armv8.1-M ISA

The Armv8.1-M Instruction Set Architecture (ISA) is part of the M-Profile of the Arm architecture, which "*provides a standard instruction set and programmer models for secure microprocessors that are optimized for the lowest power consumption, low-latency and highly deterministic operation for deeply embedded systems*" [11].

B. M-profile Vector Extension (MVE)

The Armv8.1-M ISA introduces the M-profile Vector Extension (MVE), also known as the Helium vector extension. Helium can be seen as the M-profile equivalent to the Neon vector extension for the A-profile of the Arm architecture, bringing the benefits of SIMD to the embedded market and enabling speedups in a wide range of applications including signal processing and machine learning. This gives designers the option to speed up Cortex-M applications even more than previous Cortex-M processors, with the help of SIMD.

Similarly to the Neon vector extension, Helium uses 128-bit vector registers. However, unlike the 32 x 128-bit Neon vector registers, Helium only has 8 x 128-bit vector registers. This difference in the amount of registers is partially compensated for by the prevalent presence of SIMD instructions using both vector registers and scalar registers from the General Purpose Register File (GPRF), helping to alleviate some pressure from the smaller Vector Register File (VRF). The frequent use of scalar-vector operations is not very feasible in A-profile architectures as the physical distance between the GPRF and the VRF is too large for low latency operations. Yet the small area nature of M-profile architectures makes it possible to implement low latency scalar-vector instructions [3].

With Helium having a smaller VRF, a certain element of complexity is passed onto the programmer. Typical vectorization techniques, like batching, i.e. doing the same operation on multiple higher level constructs, as explained in [3], are not very suited to the Helium architecture, as they require a large number of vector registers and don't often use General Purpose Registers (GPRs).

Implementations of Helium use the concept of a beat; 32-bits of arithmetic compute. The processing of 128-bits of arithmetic compute can be split up into four beats. Every beat in itself can contain multiple lanes, for example, when operating on 8-bit elements, every beat, 4 x 8-bit lanes are processed. When working in a dual-beat implementation of Helium, as is the case for the Cortex-M55, two beats can be processed per cycle. A full 128-bit SIMD instruction than requires at least two clock cycles to complete [12].

The Armv8.1-M architecture allows neighbouring vector instructions to overlap by two beats: That is, the first two beats of the next instruction may execute in parallel with the last two beats of the previous instruction. Practically, however, this is only possible if there are no conflicting hardware requirements for the two instructions.

The capability to overlap operations, that the ArmV8.1-M ISA allows for, is strictly separate from a dual-issue capability. The dual-issue capability refers to the simultaneous fetching of two instructions. The instruction overlapping capability on the other hand still only allows one instruction to be fetched each clock cycle, but allows for the concurrent processing of the last two beats of the previously fetched instruction and the first two beats of the current instruction.

Currently, there exist two implementations of the Armv8.1-M ISA with the addition of the MVE: the Cortex-M55 and the Cortex-M85.

C. Cortex-M55

The Cortex-M55 processor is the first processor to implement the Armv8.1-M Instruction Set Architecture (ISA). It aims to improve performance for Artificial Intelligence (AI) and Signal Processing in endpoint devices.

The Cortex-M55 CPU optionally includes a dual-beat implementation of the Helium vector extension. This means that, with the extension, the Cortex-M55 is able to do a 64-bit multiply and accumulate operation (MAC) per cycle. As the Helium registers are 128-bit, every vector operation requires at least two cycles.

The pipeline design in the Cortex-M55 allows the processing of the last two beats of the previous instruction to overlap with the processing of the first two beats of the current instruction, as long as there are no conflicting hardware requirements. Citing from [12]: "*In simple terms, the pipeline partition allows overlapping of instructions belonging to vector load/store, vector integer, and vector floating-point categories*". In other words, while the second 64-bit operation is done for an earlier instruction, an instruction from another group can already operate on the result from the first 64-bit operation.

The Cortex-M55 implementation also allows for a so called, *limited dual issue* capability. This entails that some pairs of 16-bit Thumb instructions can be fetched at the same time.

While high end processors often implement out-of-order execution, this is not the case for the single-issue in-order

execution of the Cortex-M55. Here the burden is placed on the programmer or the compiler to make sure that instructions are ordered in such a way as to not introduce stalls in the pipeline execution, e.g. by obeying latencies and instruction overlapping capabilities.

IV. Rejection Sampling Optimization

For security reasons, the signing operation makes use of rejection sampling [1]. In implementations, this rejection sampling often takes the form of a loop with four rejection conditions, which if true, jump to a new loop iteration, as shown in algorithm 1. In other words, only if all four rejection conditions are false, then will the loop terminate.

Algorithm 1 also shows that implementations defer the calculation of certain variables, to be calculated only before the conditions where they are needed. Meaning that the first invalid condition branches to the beginning of the loop again, skipping deferred operations beyond that invalid condition, and thus reducing the average processing required in each loop iteration.

Algorithm 1 Reference Implementation [2]: Loop Conditions

while (true) **do**
... Loop Code ...

$\vec{z} = \vec{y} + iNTT(c\vec{s_1})$
if $\|\vec{z}\|_\infty \geq \gamma_1 - \beta$ **then** ▷ *Z Norm* Condition
continue
end if

$\vec{r_0} = LowBits_q(\vec{w} - iNTT(c\vec{s_2}), 2\gamma_2)$
if $\|\vec{r_0}\|_\infty \geq \gamma_2 - \beta$ **then** ▷ *R Norm* Condition
continue
end if

if $\left\|iNTT(c\vec{t_0})\right\|_\infty \geq \gamma_2$ **then** ▷ *H Norm* Condition
continue
end if

$h = MakeHint_q(-c\vec{t_0}, \vec{w} - c\vec{s_2} + c\vec{t_0}, 2\gamma_2)$
if # of 1's in h $> \omega$ **then** ▷ *H Ones* Condition
continue
end if

break
end while

In algorithm 1, based on the round three submission source code of Dilithium [2], the *Z Norm* condition is evaluated first, but it is the *R Norm* that more often is the cause for a rejection, as shown in figure 2, obtained by recording the evaluation conditions for 39000 loop iterations.

The results visible in figure 2 show that, even though the calculation before the Z Norm condition is always somewhat larger than the calculation before the R Norm condition, it might still be beneficial to swap the Z Norm and N Norm rejection condition in the rejection loop.

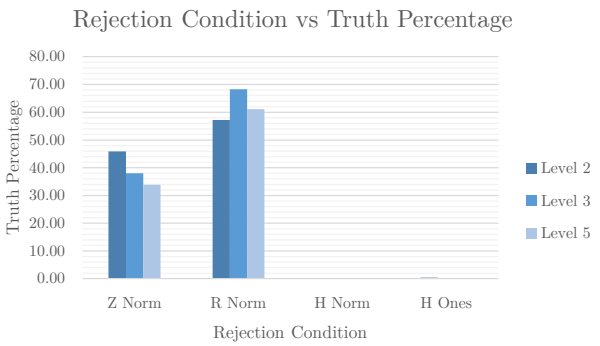


Fig. 2: A bar diagram showing the percentage of iterations that each of the rejection conditions is true.

TABLE I: A table with the test results of swapping the *Z Norm* and the *R Norm* conditions

Security Level	Cycles (Kaby Lake)		Improvement (%)
	Z Norm First	R Norm First	
Level 2			
Average	1182687	1168759	1.18
Median	910608	910490	0.01
Level 3			
Average	1917169	1832583	4.41
Median	1552239	1517203	2.26
Level 5			
Average	2367602	2317952	2.10
Median	1956428	1887166	3.54

To test this, we measured the time needed to run 10000 signature generation operations, using the speed test provided with the round three NIST submission for Dilithium [2], on a Kaby Lake Intel CPU. The obtained results are given in table I, and show a definite overall execution time reduction over the Dilithium reference implementation. Another major benefit of this improvement is that it works for a whole range of hardware implementations, and not only for implementations on the Cortex-M55.

V. Assembly Level Optimization

Section II showed that the Dilithium workload is dominated by the SHAKE and polynomial multiplication operations. This means that whether the Armv8.1-M ISA is suited to implement the Dilithium workload, is mainly determined by whether the ISA allows for the efficient implementation of the SHAKE and polynomial multiplication operations.

A. Existing NTT implementations

1) Reference Dilithium NTT: As Dilithium is designed with the NTT-based polynomial multiplication in mind, the reference implementation already contains a basic NTT transformation implementation in c.

A schematic of how this implementation calculates the forward NTT transformation is shown in Figure 3, based on Figure 1 from chapter II. The grey and white stripe pattern in each of the layers in this schematic highlights the

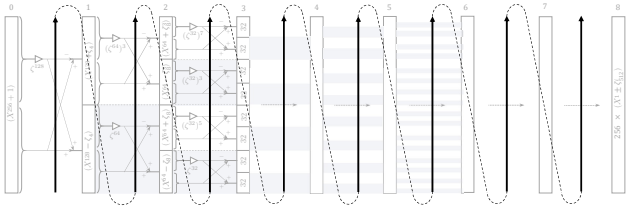


Fig. 3: A schematic, detailing the calculation order of the reference forward NTT implementation

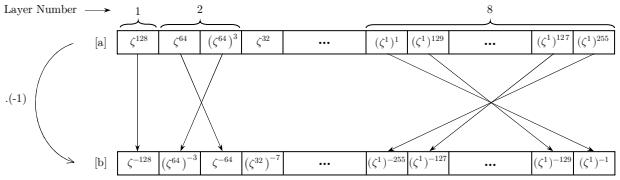


Fig. 4: Obtaining inverted roots

butterfly operations with a different twiddle factor. The group of operations inside one of these grey or white stripes is further referred to as a block.

The reference implementation processes the forward NTT transformation layer by layer, going through all the blocks in each layer, block by block, from the bottom to the top. In each block, two elements get processed each iteration, using a radix-2 butterfly operation.

Before every block, a new twiddle factor is loaded, used in all the butterfly operations in that block. In the reference implementation, these twiddle factors are stored in an array, in the order they are needed. This twiddle factor array is indicated as array [a] in figure 4.

The reference Dilithium implementation also provides an implementation of the inverse NTT (iNTT). The iNTT uses the Gentileman-Sande (GS) [13] butterfly operation: $(c_{top}, c_{bot}) \mapsto (\zeta^{-1}(c_{bot} - c_{top}), c_{bot} + c_{top})$.

In the iNTT, just like in the forward NTT transformation, a new twiddle factor is loaded before every block, only this time, the inverse of the twiddle factor loaded in the forward NTT transformation is used. The reference Dilithium implementation does not store the inverse twiddle factors separately, but makes use of the twiddle factors already used for the forward NTT transformation to obtain the inverse values. By taking the negative of all the forward NTT twiddle factors, all the inverse twiddle factors can be obtained. However, the obtained inverse twiddle factors are not in the same order as the twiddle factors they were derived from, as shown in figure 4. To understand the permutation behavior visible in figure 4, take ζ^{-64} for example, which is equivalent to $-(\zeta^{64})^3$, as explained in equation 1.

$$\begin{aligned} \zeta^{64}(\zeta^{64})^3 = \zeta^{256} = \zeta_4^2 = -1 \ mod\ Q \\ \Downarrow \\ (\zeta^{64})^3 = -(\zeta^{-64}) \ mod\ Q \end{aligned} \tag{1}$$

This same explanation can be applied to all the other twiddle factors, effectively reversing the order of the twiddle factors in each layer. The benefit of this permutation is that by starting to read at the end of the permuted twiddle factor array, the order of the inverse twiddle factors in each layer is again flipped, resulting in the same block order as in the forward NTT. With the extra benefit that, as the iNTT starts at layer eight, the layers as a whole are already in the correct order.

The twiddle factors here refer to powers of the primitive root multiplied by a factor of 2^{32}, in order to compensate for the Montgomery reduction technique used after the twiddle factor multiplication. The reference iNTT transformation also multiplies, after transforming from the NTT domain, by a factor of 2^{32}. This extra multiplication compensates for the Montgomery reduction used in the polynomial multiplication operation between the NTT and the iNTT operations.

2) NTT Python-based "compiler" module: The partially optimized NTT assembly implementation generated by the NTT "compiler" Python module from [14] [3], but altered for the case of Dilithium, works as shown in figure 5.

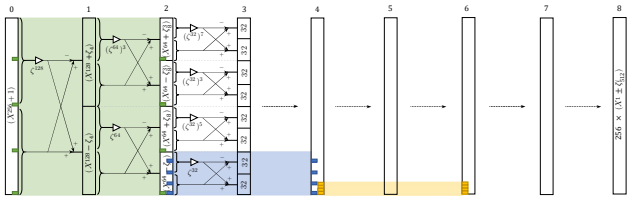
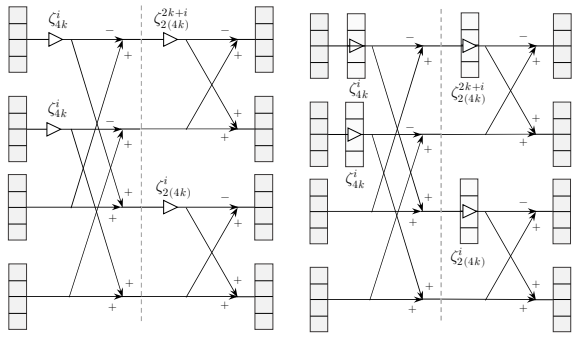


Fig. 5: A schematic, detailing the working of the forward NTT implementation, generated by the NTT "compiler" Python module

The NTT "compiler" Python module computes two layers of the Dilithium NTT at the same time. This requires three times more twiddle factors to be loaded for every butterfly operation, compared to the reference implementation that requires one twiddle factor to be loaded, more optimally making use of the available registers in the GPRF. This is especially helpful due to the large amount of scalar-vector instructions present in the Armv8.1-M ISA.

For the initial six layers of the forward Dilithium NTT transformation, two layers are processed at the same time using the combination of three radix-2 butterfly operations, depicted in figure 6a.

Figure 6a also shows that every one of these butterfly operations operates on four 128-bit vector registers, filled with 4 x 32-bit coefficient values (each 32-bit value is indicated by a grey box in figure 6a). These vector registers are also visible in figure 5, where the first block that is processed in every one of the initial three layer-pairs is highlighted with a different color. For every one of these blocks, the input and output vector locations for the first butterfly operation are visualized in the layers beside the block. Every one of these vector boxes contains 4 x 32-bit coefficient values. In layer-pair three, which starts from layer four, the amount of values to be processed by every block is equal to 16. This means that, with every block using a concatenation of radix-2 butterflies from two layers, and with four input vectors containing four coefficient



(a) Initial layers (b) Last few layers

Fig. 6: The butterflies used in the forward NTT implementation generated by the NTT ”compiler” Python module

values each, every block can be calculated with one butterfly calculation.

While the first three layer-pairs all have block sizes that are a multiple of 16 values, allowing every block to be calculated using the butterfly in figure 6a, without operating on half filled vector registers, this is not the case for the last layer-pair. The last layer-pair, starting at layer six, only has four input values for every block. For this reason, the last layer-pair makes use of the butterfly operation shown in figure 6b.

The butterfly operation shown in figure 6b works by processing four blocks at the same time, which is accomplished by also storing the twiddle factors in a vector. Every vector register used then contains the values of four blocks.

B. The search for stalls

Instead of checking for sequences of instructions that could cause stalls to occur by hand, this thesis introduces a Python program that scans assembly files, and highlights some of the locations where stalls occur.

Currently, this Python module supports all instructions used in the assembly implementation generated by the original NTT Python ”compiler”, explained in the previous section. The following instructions are currently included: add, bx, ldr, ldrd, pop, push, vadd, vld40-4, vldrw, vmul, vpop, vpush, vqrdmlah, vqrdmulh, vstrw, and vsub. This list is easily extendable. For these instructions, the stall checker is already able to estimate where stalls might occur due to the presence of a structural hazard. This is done by grouping all the instructions in three separate instruction groups (manually); load/store instructions, addition/subtraction/logical instructions, or multiply/floating-point instructions. This grouping is then used to detect stalls, by verifying whether or not two successively executed instructions belong to the same instruction group, indicating that a structural stall is present or not.

Multiplication operations have a latency of 2 clock cycles, referring to the fact that the output of a Helium SIMD vector multiplication operation is only available after two clock cycles. The stall checker is also able to estimate stalls that occur as a result of a data hazard, generated by an instruction using the output of a multiplication operation in the next clock cycle. While true for all other instructions, if the instruction that is executed immediately after the multiplication instruction is a vector store instruction, then no stall will be present, and hence is excluded from the previous stall condition in the stall checker. Lastly, the stall checker also checks for store-load related stalls.

For every stall, a comment is added to an annotated version of the original asm file, and a warning is written to the stdout stream. While the current version of the Python stall checker program is able to find stalls in the assembly NTT implementation generated by the original NTT Python ”compiler”, explained in the previous section, this will only approximate the real behaviour, and can still be improved upon. For example, detecting stalls that occur at the end or the beginning of a loop is not yet supported.

C. The MPS3 FPGA Prototyping Board

In order to benchmark the optimizations on the Cortex-M55 processors, we make use of the Arm®MPS3 FPGA Prototyping Board, revision HPI-0309C. The Arm®MPS3 FPGA Prototyping Board will further be referred to as the MPS3.

The code that needs to run on the MPS3, with the Application Note 552 (AN552) FPGA image installed [15], is compiled using the Arm GNU Toolchain (version 10.3) [16], and makes use of the Common Microcontroller Software Interface Standard (CMSIS) [17], and a CMSIS support package for the MPS3 [18].

The compilation process is implemented using a makefile. This makefile offers the option to compile with one of three files, each implementing a different main function, depending on the functionality that is required:

1) **speed.c**: This file measures the execution time (in clock cycles) for each of the different Dilithium components. It also measures the execution time of some relevant sub-components, namely: the NTT transformation, the iNTT transformation, a single polynomial multiplication, and the matrix-vector polynomial multiplication. In addition, it also offers the possibility to measure the execution time using one of three methods: using the Systick Timer, using a timer peripheral, or using the Data Watchpoint and Trigger (DWT) debug unit, by reading from the Cycle Count Register. These timing methods are based on an example extended ”hello world” program provided by my supervisors.
2) **verify.c**: This file verifies whether the Dilithium components, and some of the relevant sub-components, still produce the correct result. Most of the code here is copied from the original verification code, i.e., the round three Dilithium submission reference code [2]. If needed, this file can also be altered to read the data from known-answer-test header files, created by another custom program, which stores input and the corresponding output values generated by the reference Dilithium implementation [2].
3) **debug.c**: This file provides the functionality to help debug the Dilithium NTT. It runs both the original

reference NTT from [2], and the NTT version that needs to be debugged. With the help of print statements, this allows intermediate values to be compared, in order to find where the problems with the custom NTT implementation are located.

VI. Conclusion

Initially, our focus was on finding some high level optimizations in the Dilithium workload. Where we found that, by changing the order of two rejection conditions in the rejection loop, we were able to improve the average Dilithium execution time by 1.18%, 4.41%, and 2.1% for the NIST security level 2, 3, and 5 respectively. With the extra benefit that this optimization is not specific to any hardware architecture.

Next, we worked on low level optimizations on the Cortex-M55, using the Armv8.1-M ISA. For this, we focused on optimizing the Dilithium NTT on the Cortex-M55, using the SIMD capabilities of the Helium vector extension. We were eventually able to run the Dilithium workload with an assembly implementation of the Dilithium NTT, generated by the Python-based framework used in [3], on the Arm MPS3 FPGA prototyping board.

References

[1] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-dilithium algorithm specifications and supporting documentation (version 3.1)," 2021, https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[2] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, and D. Stehle, "Dilithium reference implementation," 2021, https://pq-crystals.org/dilithium/resources.shtml.

[3] H. Becker, J. M. B. Mera, A. Karmakar, J. Yiu, and I. Verbauwhede, "Polynomial multiplication on embedded vector architectures," Cryptology ePrint Archive, Report 2021/998, 2021, https://ia.cr/2021/998.

[4] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[5] NIST, "Nist pqc timeline," URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline, last checked on 2022-08-01.

[6] ——, "Post-quantum cryptography," URL: https://csrc.nist.gov/Projects/post-quantum-cryptography, last checked on 2022-04-21.

[7] ——, "Third round candidate announcement," URL: https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement, last checked on 2022-08-01.

[8] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and D. Sprenkels, "Faster kyber and dilithium on the cortex-m4," Cryptology ePrint Archive, Report 2022/112, 2022, https://ia.cr/2022/112.

[9] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "PQM4: Post-quantum crypto library for the ARM Cortex-M4," https://github.com/mupq/pqm4.

[10] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and D. Sprenkels, "Faster kyber and dilithium on the cortex-m4," Cryptology ePrint Archive, Report 2022/112, 2022, https://ia.cr/2022/112.

[11] Arm, "M-profile architecture," URL: https://developer.arm.com/Architectures/M-Profile Architecture, last checked on 2022-04-26.

[12] J. Yiu, "Whitepaper: Blending dsp and ml features into a low-power general-purpose processor - how far can we go?" ARM, Tech. Rep., 2020. [Online]. Available: URL: https://developer.arm.com/documentation/102892/0100/?lang=en

[13] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit," in *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, pp. 563–578. [Online]. Available: https://doi.org/10.1145/1464291.1464352

[14] H. Becker and A. Karmakar, "pqmx," https://gitlab.com/arm-research/security/pqmx, 2021.

[15] Arm, "Download fpga images," URL: https://developer.arm.com/downloads/-/download-fpga-images, last checked on 2022-08-13.

[16] ——, "Gnu arm embedded toolchain downloads," URL: https://developer.arm.com/downloads/-/gnu-rm, last checked on 2022-08-13.

[17] ——, "Cmsis," URL: https://developer.arm.com/tools-and-software/embedded/cmsis, last checked on 2022-08-13.

[18] ——, "Mdk5 software packs," URL: https://www.keil.com/dd2/pack/, last checked on 2022-08-13.

Bibliography

[1] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and D. Sprenkels. Faster kyber and dilithium on the cortex-m4. Cryptology ePrint Archive, Report 2022/112, 2022. https://ia.cr/2022/112.

[2] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and D. Sprenkels. Faster kyber and dilithium on the cortex-m4. Cryptology ePrint Archive, Report 2022/112, 2022. https://ia.cr/2022/112.

[3] Arm. A-profile architecture. URL: https://developer.arm.com/Architectures/A-ProfileArchitecture, last checked on 2022-04-26.

[4] Arm. Arm cortex-a series programmer's guide for armv7-a. URL: https://developer.arm.com/documentation/den0013/d/Introducing-NEON/SIMD, last checked on 2022-08-03.

[5] Arm. Arm cortex-a series programmer's guide for armv8-a. URL: https://developer.arm.com/documentation/den0024/a/AArch64-Floating-point-and-NEON?lang=en, last checked on 2022-08-4.

[6] Arm. Cmsis. URL: https://developer.arm.com/tools-and-software/embedded/cmsis, last checked on 2022-08-13.

[7] Arm. Download fpga images. URL: https://developer.arm.com/downloads/-/download-fpga-images, last checked on 2022-08-13.

[8] Arm. The dsp capabilities of the arm cortex-m4 and cortex-m7 processors. URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwj5m4iWt6v5AhWNOewKHThKC94QFnoECAgQAQ&url=https%3A%2F%2Fcommunity.arm.com%2Fcfs-file%2F__key%2Fcommunityserver-blogs-components-weblogfiles%2F00-00-00-21-42%2F7563.ARM-white-paper-_2D00_-DSP-capabilities-of-Cortex_2D00_M4-and-Cortex_2D00_M7.pdf&usg=AOvVaw1b5YLuHoGCWt6fADtBPQci, last checked on 2022-08-03.

[9] Arm. Gnu arm embedded toolchain downloads. URL: https://developer.arm.com/downloads/-/gnu-rm, last checked on 2022-08-13.

[10] Arm. Introducing neon development article. URL: https://developer.arm.com/documentation/dht0002/a/Introducing-NEON, last checked on 2022-08-4.

[11] Arm. Introduction to helium. URL: https://developer.arm.com/documentation/102102/0102, last checked on 2022-08-03.

[12] Arm. Introduction to sve. URL: https://developer.arm.com/documentation/102476/0100/Introducing-SVE, last checked on 2022-08-4.

[13] Arm. Introduction to sve2. URL: https://developer.arm.com/documentation/102340/0001/Introducing-SVE2, last checked on 2022-08-14.

[14] Arm. M-profile architecture. URL: https://developer.arm.com/Architectures/M-ProfileArchitecture, last checked on 2022-04-26.

[15] Arm. Mdk5 software packs. URL: https://www.keil.com/dd2/pack/, last checked on 2022-08-13.

[16] Arm. Product brief: Ethos-u55 processor. URL: https://armkeil.blob.core.windows.net/developer/Files/pdf/ML%20on%20Arm/Arm_Ethos_U55_Product_Brief_v4.pdf, last checked on 2022-05-19.

[17] Arm. R-profile architecture. URL: https://developer.arm.com/Architectures/R-ProfileArchitecture, last checked on 2022-04-26.

[18] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. MandrA, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven, and J. M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature (London)*, 574(7779):505–510, 2019.

[19] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé. Crystals-dilithium algorithm specifications and supporting documentation (version 3.1), 2021. https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[20] P. Ball. First 100-qubit quantum computer enters crowded race. *Nature (London)*, 599(7886):542–542, 2021.

[21] P. Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[22] H. Becker. Whitepaper: Post-quantum cryptography. Technical report, ARM, 2020.

[23] H. Becker, V. Hwang, M. J. Kannwischer, B.-Y. Yang, and S.-Y. Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. Cryptology ePrint Archive, Report 2021/986, 2021. https://ia.cr/2021/986.

[24] H. Becker and A. Karmakar. pqmx. https://gitlab.com/arm-research/security/pqmx, 2021.

[25] H. Becker, J. M. B. Mera, A. Karmakar, J. Yiu, and I. Verbauwhede. Polynomial multiplication on embedded vector architectures. Cryptology ePrint Archive, Report 2021/998, 2021. https://ia.cr/2021/998.

[26] M. Bhattacharya and J. Astola. Some historical notes on number theoretic transform. 2004.

[27] J. W. Bos, J. Renes, and D. Sprenkels. Dilithium for memory constrained devices. Cryptology ePrint Archive, Report 2022/323, 2022. https://ia.cr/2022/323.

[28] J. COOLEY and J. TUKEY. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[29] M.-W. Dictionary. "modular arithmetic". URL: https://www.merriam-webster.com/dictionary/modular%20arithmetic, last checked on 2022-08-02.

[30] O. E. Dictionary. The etymology of "parallelism". URL: https://www.etymonline.com/word/parallelism, last checked on 2022-08-03.

[31] W. M. Gentleman and G. Sande. Fast fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. Association for Computing Machinery.

[32] L. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on theory of computing*, STOC '96, pages 212–219. ACM, 1996.

[33] J. M. B. Mera, A. Karmakar, and I. Verbauwhede. Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. Cryptology ePrint Archive, Report 2020/268, 2020. https://ia.cr/2020/268.

[34] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[35] NIST. Advanced encryption standard (aes). URL: https://csrc.nist.gov/publications/detail/fips/197/final, last checked on 2022-04-21.

[36] NIST. Announcing four candidates to be standardized. URL: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4, last checked on 2022-08-01.

[37] NIST. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (sha-3) family. URL: https://csrc.nist.gov/News/2017/Research-Results-on-SHA-1-Collisions, last checked on 2022-05-17.

[38] NIST. Data encryption standard (des). URL: https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25, last checked on 2022-04-21.

[39] NIST. Hash functions. URL: https://csrc.nist.gov/projects/hash-functions, last checked on 2022-05-17.

[40] NIST. Nist pqc timeline. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline, last checked on 2022-08-01.

[41] NIST. Post-quantum cryptography. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography, last checked on 2022-04-21.

[42] NIST. Research results on sha-1 collisions. URL: https://csrc.nist.gov/News/2017/Research-Results-on-SHA-1-Collisions, last checked on 2022-05-17.

[43] NIST. Round 3 submissions. URL: https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions, last checked on 2022-08-01.

[44] NIST. Third round candidate announcement. URL: https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement, last checked on 2022-08-01.

[45] NIST. Triple data encryption algorithm (tdea, triple des). URL: https://csrc.nist.gov/publications/detail/sp/800-67/rev-2/final, last checked on 2022-04-21.

[46] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[47] B. Preneel. Lecture notes of cryptography and network security (h05e1a), 2021.

[48] M. Sato. The supercomputer "fugaku" and arm-sve enabled a64fx processor for energy-efficiency and sustained application performance. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 1–5. IEEE, 2020.

[49] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, and D. Stehle. Dilithium reference implementation, 2021. https://pq-crystals.org/dilithium/resources.shtml.

[50] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[51] D. Sprenkels. The kyber/dilithium ntt, Sep 2020.

[52] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, and C. Baisse. Announcing the first sha1 collision, Feb 2017.

[53] J. Yiu. Whitepaper: Blending dsp and ml features into a low-power general-purpose processor - how far can we go? Technical report, ARM, 2020.

[54] J. Yiu. Whitepaper: Blending dsp and ml features into a low-power general-purpose processor - how far can we go? Technical report, ARM, 2020.