RUHR-UNIVERSITÄT BOCHUM

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Dilithium on OpenTitan

Amin Abdulrahman

Master's Thesis. December 20, 2023.
Ruhr University Bochum
Max Planck Institute for Security and Privacy (MPI-SP)
First Examiner: Prof. Dr. Peter Schwabe
Second Examiner: Prof. Dr.-Ing. Tim Güneysu

With our nowadays deployed public-key cryptography schemes at risk for compromise by large-scale quantum computers, research on schemes secure under such circumstances has been ongoing for many years. As the standardization effort for such schemes by the National Institute of Standards and Technology (NIST) progresses toward an end, integrating the newly developed methods into existing applications and devices also needs to advance.

In this work, we evaluate the feasibility of implementing the digital signature scheme CRYSTALS-Dilithium on the OpenTitan Big Number (OTBN) coprocessor of the OpenTitan hardware Root of Trust (RoT). While OTBN offers several hardware components aiding the computation of classical cryptography, no such accelerators are available for Post-Quantum Cryptography (PQC). Thus, we additionally consider extensions to OTBN's architecture to speed up lattice-based cryptography and evaluate our extensions' performance benefits with Dilithium as a case study. We propose the addition of Single Instruction Multiple Data (SIMD) instructions for (modular) additions, subtractions, and multiplications, as well as a permutation instruction and SIMD bit-shifts.

In order to evaluate our implementation, we use a modified variant of the OTBN's cycle-accurate Python simulator. Additionally, we use a feature under development by the OpenTitan team that allows access to a Keccak Message Authentication Code (KMAC) accelerator for offloading SHAKE computations.

For our baseline implementation, we achieve cycle counts of 257888, 1100386, and 315586 for key generation, signing, and verification, of which 45%, 68%, and 53% are spent on polynomial arithmetic, respectively.

Thus, the main focus of our extensions is the acceleration of polynomial arithmetic, especially polynomial multiplication. The extensions enable an approximately tenfold speed-up for the Number-Theoretic Transform (NTT), Inverse Number-Theoretic Transform (INTT), and pointwise multiplication for Dilithium when compared to our baseline implementation. Our full-scheme implementation, in comparison to the baseline, achieves median cycle counts of 127353, 273755, and 128195 and speed-ups of factor 2.02, 4.02, and 2.46 for key generation, signing, and verification, respectively.

Titel meiner Abschlussarbeit / title of the final thesis

Dilithium on OpenTitan

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich erkläre mich des Weiteren damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Statutory Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university to obtain an academic degree.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I furthermore agree that the digital version of this thesis will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

20.12.2023	Abdulrahman, Amin	Amin Abdulrahman
Datum / Date	Name, Vorname	Unterschrift / Signature

Contents

Acronyms

AES . . . . . Advanced Encryption Standard

ALU . . . . . Arithmetic Logic Unit

ASIC . . . . . Application-Specific Integrated Circuit

BSI . . . . . Bundesamt für Sicherheit in der Informationstechnik

CISC . . . . . Complex Instruction Set Computer

CRT . . . . . Chinese Remainder Theorem

CSR . . . . . Control and Status Register

CSRNG . . . . . Cryptographically Secure Random Number Generator

CT . . . . . Cooley–Tukey

DFT . . . . . Discrete Fourier Transform

DOM . . . . . Domain-oriented Masking

ECC . . . . . Elliptic Curve Cryptography

EM . . . . . Electro-Magnetic

FFT . . . . . Fast Fourier Transform

FIPS . . . . . Federal Information Protection Standard

FNT . . . . . Fermat Number Transform

FPGA . . . . . Field Programmable Gate Array

GLWE . . . . . General Learning with Errors

GPR . . . . . General Purpose Register

GS . . . . . Gentleman–Sande

HMAC . . . . . Hash-based Message Authentication Code

INTT . . . . . Inverse Number-Theoretic Transform

IP . . . . . Intellectual Property

ISA . . . . . Instruction Set Architecture

KEM . . . . . Key Encapsulation Mechanism

KMAC . . . . . Keccak Message Authentication Code

LSB . . . . . Least Significant Bit

LWE . . . . . Learning with Errors

MAC . . . . . Message Authentication Code

ML-DSA . . . . . Module-Lattice-Based Digital Signature Standard

ML-KEM . . . . . Module-Lattice-Based Key-Encapsulation Mechanism Standard

MLWE . . . . . Module Learning with Errors

MSB . . . . . Most Significant Bit

MSIS . . . . . Module Short Integer Solution

NIST . . . . . National Institute of Standards and Technology

NTT . . . . . Number-Theoretic Transform

OoO . . . . . Out-of-Order

OTBN . . . . . OpenTitan Big Number

PKC . . . . . Public-Key Cryptography

PQC . . . . . Post-Quantum Cryptography

RAM . . . . . Random Access Memory

RISC . . . . . Reduced Instruction Set Computer

RLWE . . . . . Ring Learning with Errors

RNG . . . . . Random Number Generator

RoT . . . . . Root of Trust

RSIS . . . . . Ring Short Integer Solution

SHA-3 . . . . . Secure Hash Algorithm 3

SIMD . . . . . Single Instruction Multiple Data

SIS . . . . . Short Integer Solution

SIVP . . . . . Shortest Independent Vector Problem

SLH-DSA . . . . . Stateless Hash-Based Digital Signature Standard

SoC . . . . . System on a Chip

SUF-CMA . . . . . Strong Existential Unforgeability under Chosen-Message Attack

SVP . . . . . Shortest Vector Problem

WDR . . . . . Wide Data Register

WSR . . . . . Wide Special Purpose Register

XOF . . . . . Extended Output Function

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

This chapter explains our motivation for implementing Dilithium on OpenTitan. Moreover, we introduce related work and give an overview of the organization of this thesis.

1.1 Motivation

The security of nowadays widely deployed Public-Key Cryptography (PKC) schemes such as RSA or ECC-based schemes is threatened by Shor's algorithm, presented in 1994 by Peter Shor [Sho94]. The proposed algorithm allows for efficient prime factorization of integers and solving the discrete logarithm problem if powerful enough quantum computers would become available. In fact, a recent survey has shown that over 50% of the participating experts think it is more likely than not that within the next 15 years, a quantum computer will be able to break RSA-2048 in less than 24 hours [MP22]. Meanwhile, for more than two decades, researchers have been working on developing new cryptographic schemes that are secure even in the presence of powerful quantum computers. This field of research is usually referred to as "Post-Quantum Cryptography", a term coined by Bernstein [BGD+04].

As a reaction to this threat, in 2016, National Institute of Standards and Technology (NIST) announced the Post-Quantum Cryptography (PQC) standardization process for selecting one or multiple Key Encapsulation Mechanisms (KEMs) and digital signature schemes secure against attacks using powerful classical, as well as quantum computers [Nat]. The standardization process follows the mode of operation in which Rijndael has been selected as the Advanced Encryption Standard (AES), as well as Keccak as Secure Hash Algorithm 3 (SHA-3).

Out of the initial 82 submissions, four have been selected for standardization in July 2022 [NIS22]. NIST selected CRYSTALS-Kyber as a KEM, as well as CRYSTALS-Dilithium, Falcon, and SPHINCS$^+$ as digital signature schemes. Draft standards for three of the schemes have already been published as Federal Information Protection Standard (FIPS): For CRYSTALS-Kyber in FIPS 203 under the name Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) [NIS23a], for CRYSTALS-Dilithium in FIPS 204 as Module-Lattice-Based Digital Signature Standard (ML-DSA) [NIS23b], and for SPHINCS$^+$ in FIPS 205 as Stateless Hash-Based Digital Signature Standard (SLH-DSA) [NIS23c]. There is an ongoing fourth round for potentially standardizing an additional KEM, as well as an additional round for digital signature schemes in which new submissions have been accepted.

While schemes from a variety of types, for example, based on lattices, codes, hashes, or supersingular-elliptic-curve isogenies, had been submitted to the NIST process, all the

schemes selected for standardization, except SPHINCS$^+$, base their hardness on problems over lattices.

From an implementation perspective, as the Arm Cortex-M4 has been chosen as the main optimization target for microcontrollers by the NIST, it has received the most attention, for example, in [BKS19, ABCG20, GKS20, BGR$^+$21, AHKS22, HZZ$^+$22, BRS22], resulting in thoroughly optimized implementations curated in [KRSS19]. In addition, there are several implementations on architectures with Single Instruction Multiple Data (SIMD) capabilities, also known as vector extensions, e.g., presented in [Dil23, Kyb23, NG21, BHK$^+$22]. Hardware/Software co-designs have also gained attention from the research community, with several publications proposing Instruction Set Architecture (ISA) extensions to RISC-V-based systems using loosely or tightly coupled accelerators, for example, in [AEL$^+$20, FSS20, KA20, NDMZ$^+$21, ZXXH22, KSFS23].

After many publications on software implementations for microcontrollers, as well as proposals of hardware/software co-designs, we think it is worthwhile to extend this series of work to an existing System on a Chip (SoC) that is built with the application in the real world in mind. For this endeavor, we chose an SoC designed by the OpenTitan project, which is building a hardware Root of Trust (RoT) device [Tea23a]. One advantage of the OpenTitan project is that it is fully open source, meaning its hardware designs and all the related software are publicly accessible, thus making it attractive as a research target. Additionally, the RoT already offers specialized hardware components to aid with classical cryptographic computations as well as to ensure resistance against side-channel attacks. One of those components is called the OTBN core, which is a separate, programmable processor on the RoT that is designed for cryptographic implementations. Thus, we think it is a logical step to consider the implementation of PQC on the OTBN core and to explore extensions to it that may improve its suitability for PQC.

1.2 Related Work

There exists a rich literature with respect to implementations of Dilithium on a variety architectures. For example, on the Arm Cortex-M4, [AHKS22] provides a highly optimized implementation from a performance perspective, while [GKS20] as well as [BRS22] explore approaches to reducing the memory requirements, which is of special interest on resource-constrained devices. The latter presents methods to make Dilithium's security levels 2 and 3 fit into 8 KiB of Random Access Memory (RAM).

Implementations on architectures with SIMD instructions have been proposed for Intel's AVX2/AVX512, as well as Arm's Neon vector extension in [Dil23, ZZS$^+$23, LHP$^+$23] and [BHK$^+$22, ZHS$^+$22], respectively.

There also exists a line of work proposing instruction-set extensions in hardware/software co-designs, which aim to speed up PQC schemes [FSS20, KA20, NDMZ$^+$21, ZHL$^+$21, ZXXH22, GJCJ23, MBB$^+$23]. One of the most early works on tightly coupled accelerators has been presented in [AEL$^+$20], where the authors target a RISC-V platform. More recently, a set of masked accelerators has been suggested in [FBR$^+$22]. Building up on this approach, [KSFS23] narrows the accelerators down for Dilithium specifically.

On OpenTitan's OTBN core specifically, the work on two different approaches has been published concurrently to our research. In a Master's thesis, the applicability of the Kronecker+ method, presented in [BRv22], on OTBN has been studied [Tur23]. More closely related to our work, [SOSK23] proposes an instruction-set extension to the OTBN core, adding a so-called PQ-Arithmetic Logic Unit (ALU), capable of, for example, performing the Number-Theoretic Transform (NTT)-butterfly operations in one single cycle, as well as additional instructions aiding with the computation of Keccak. As a case study, the authors implement Kyber's and Dilithium's NTT, Inverse Number-Theoretic Transform (INTT), and pointwise multiplication, as well as the signature verification, for which they also make use of the Ibex RISC-V core located on the OpenTitan SoC, in addition to the OTBN core. A Master's thesis by Stelzer on this topic also exists, although it is not publicly available [Ste22].

In the context of a standardization effort for RISC-V cryptography extensions, [Saa23a, Saa23b] present proposals for an instruction-set extension that aids with the vectorized computation of PQC, mostly in terms of accelerating the NTT, as well as Keccak computations.

1.3 Contribution

The contribution of this thesis is threefold:

- We present a baseline implementation of Dilithium on a slightly modified variant of OpenTitan's OTBN core, without changing the instruction set.
- We propose five SIMD-instructions and a number of sub-variants as additions to OTBN's instruction set in order to speed up lattice-based cryptography. Note that we omit a concrete hardware implementation.
- We provide an optimized version of our baseline implementation that deploys our instruction-set extensions. Based on the impact on performance, we judge the efficacy of our extensions and make recommendations to the OpenTitan Team.

The source code for our implementations, as well as the testing infrastructure, is available on GitHub[1], and Appendix A gives instructions on how to replicate our setup and results.

1.4 Organization of this Thesis

In Chapter 2, we introduce the foundations and basic concepts underlying our work. This includes touching on the cryptographic background, an introduction to the Dilithium digital signature schemes, as well as details on fast polynomial multiplications and modular arithmetic. We also provide an overview of the OpenTitan SoC, as well as more detailed insight regarding the OTBN core.

[1] https://github.com/dop-amin/dilithium-on-opentitan-thesis.git

Chapter 3 presents our baseline implementation. We start off by defining our goals and introducing the limitations of our implementation before continuing with the description of our approach to the development. After that, we give insight into our implementation and describe the results from its evaluation.

Chapter 4 introduces our modifications to OTBN's instruction set and describes our optimized implementation under the application of said extensions.

We present the results of our work in Chapter 5, comparing our implementations to other implementations on OTBN, as well as on other popular platforms.

In Chapter 6, we conclude our results and give an overview over different avenues for future work.

2 Preliminaries

This section introduces the cryptographic background on the digital signature scheme Dilithium, as well as the scheme itself, aspects relevant to its implementation – especially regarding polynomial arithmetic – and the target platform of this work, OpenTitan's OTBN core and its capabilities.

2.1 Notation

We denote the polynomial ring $\mathbb{Z}_q[X]/(X^n+1)$ by $\mathcal{R}_q$ for a prime q and a power of two n. A polynomial $a \in \mathcal{R}_q$ is made up of a vector containing the individual coefficients $a_i \in \mathbb{Z}_q$ such that $a = \sum_{i=0}^{n-1} a_i X^i$. By convention, we denote polynomials using lowercase letters (e.g., a), vectors of polynomials using lowercase boldfaced letters (e.g., $\mathbf{s}$), and matrices of polynomials using uppercase boldfaced letters (e.g., $\mathbf{A}$). We follow the same pattern for elements from $\mathbb{Z}_q$, where we denote elements by lowercase letters, vectors of elements in $\mathbb{Z}_q$ by boldfaced lowercase letters, and matrices by uppercase boldfaced letters.

For intervals, we use the convention that parentheses indicate the exclusion of the respective element, while square brackets stand for inclusion. For example, $[0, q)$ contains the elements $0, 1, \ldots, q-1$, while $[0, q]$ contains the elements $0, 1, \ldots, q-1, q$.

For congruences, we follow the notation from [LDK+22]: For odd (respectively even) q, the central reduction $r' = r \bmod^{\pm} q$ is defined as the unique element in $[-\frac{q-1}{2}, \frac{q-1}{2}]$ (respectively $[-\frac{q}{2}, \frac{q}{2}]$) that satisfies $r' \equiv r \bmod q$. Similarly, $r' = r \bmod^{+} q$ denotes the unique element in $[0, q)$ that satisfies $r' \equiv r \bmod q$.

The size of an element $w \in \mathbb{Z}_q$ is denoted by $\|w\|_\infty$, which means $|\ w \bmod^{\pm} q\ |$ [LDK+22]. For a polynomial $w = \sum_{i=0}^{n-1} w_i X^i \in \mathcal{R}$, we define the l_∞ and l_2 norms like in [LDK+22]:

$$\|w\|_\infty = \max_i \|w_i\|_\infty \,, \|w\| = \sqrt{\|w_0\|_\infty^2 + \cdots + \|w_{n-1}\|_\infty} \tag{2.1}$$

We also use the same notation for embeddings of polynomials into $\mathbb{Z}_q^n$. The definition for elements $\boldsymbol{w} = (w_1, \ldots, w_k) \in \mathcal{R}^k$ is similarly:

$$\|\boldsymbol{w}\|_\infty = \max_i \|w_i\|_\infty \,, \|w\| = \sqrt{\|w_1\|^2 + \cdots + \|w_k\|} \tag{2.2}$$

When sampling w from $\mathcal{R}$ with a size restriction η, we write S_η meaning $\|w\|_\infty \leq \eta$ [LDK+22].

In algorithms, we denote equality using the equal sign "=", assignment using the "$\leftarrow$"-operator, and random sampling using "$\leftarrow\!\!\$$".

2.2 Cryptographic Background

This section introduces cryptographic assumptions and problems that lattice-based cryptographic primitives, such as DILITHIUM, are based on.

2.2.1 Learning with Errors

The Learning with Errors (LWE) problem was introduced by Regev in [Reg09] in 2005. It is proven to be as hard as the worst-case instances of the Gap Shortest Vector Problem (SVP) and Shortest Independent Vector Problem (SIVP) [Reg09].

Informally, the LWE problem can be seen as the task of solving a system of linear equations with elements from $\mathbb{Z}_q$, where a small error $e_i \in \mathbb{Z}_q$ is added to each of the equations [Pei16, Section 4.2]. This small error is crucial for the security; otherwise, this system could be trivially solved using, e.g., Gaussian elimination [Pei16, Section 4.2]. We provide more formal definitions of two variants of the problem in Definitions 2.2.1 and 2.2.2.

Definition 2.2.1 (Search Learning with Errors Problem, following [Pei16, Section 4.2])
Let $\boldsymbol{a}_i, \boldsymbol{s} \in \mathbb{Z}_q^n, i \in [1, m]$ be uniformly random with $\boldsymbol{s}$ the secret, error $e_i \in \mathbb{Z}_q$ chosen over an error distribution, and $\langle \cdot, \cdot \rangle$ denote the inner product. Given m independent samples $(\boldsymbol{a}_i, b_i) = (\boldsymbol{a}_i, \langle \boldsymbol{a}_i, \boldsymbol{s} \rangle + e_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, the **Search LWE problem** is defined as solving for the secret key, $\boldsymbol{s}$.

Definition 2.2.2 (Decisional Learning with Errors Problem, following [Pei16, Section 4.2])
Let $\boldsymbol{a}_i, \boldsymbol{s} \in \mathbb{Z}_q^n, i \in [1, m]$ be uniformly random with $\boldsymbol{s}$ the secret, error $e_i \in \mathbb{Z}_q$ chosen over an error distribution, and $\langle \cdot, \cdot \rangle$ denote the inner product. Given m independent samples $(\boldsymbol{a}_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, where b_i is either defined by $\langle \boldsymbol{a}_i, \boldsymbol{s} \rangle + e_i$, or sampled randomly from a uniform distribution for every sample, the **Decisional LWE problem** is defined as distinguishing between the two cases.

A public-key cryptosystem that Regev created based on the LWE approach [Reg09] gives public key lengths of $\mathcal{O}(n^2)$ elements in $\mathbb{Z}_q$, while secret keys and ciphertexts have a length of $\mathcal{O}(n)$ elements.

It is common to state the problem in terms of linear algebra, where a matrix $\boldsymbol{A} \in \mathbb{Z}_q^{n \times m}$ is constructed from the $\boldsymbol{a}_i$ as its rows and $\boldsymbol{e} \in \mathbb{Z}_q^m$ is the vector made up of the e_i for each individual sample. The result will then be a vector $\boldsymbol{b} \in \mathbb{Z}_q^m$ with its components in $b_i \in \mathbb{Z}_q$: $\boldsymbol{b} = \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e}$ [Pei16].

2.2.2 Ring Learning with Errors

Ring Learning with Errors (RLWE) [LPR10] was proposed in 2010 by Lyubashevsky, Peikert, and Regev. As described in [Pei16, Section 4.4], it forms the ring-based analog to the LWE problem, replacing the vectors from $\mathbb{Z}_q^n$ by polynomials from the ring $\mathcal{R}_q$. Commonly, $\mathcal{R}_q$ is chosen as $\mathbb{Z}_q[X]/(X^n + 1)$. Thus, a polynomial $r = r_0 + r_1 X + \cdots +$

$r_{n-1}X^{n-1} \in \mathcal{R}_q$ can also be represented using its coefficient embedding as a vector: $\boldsymbol{r} = (r_0, \ldots, r_{n-1}) \in \mathbb{Z}_q^n$ following the notation from [Deo18].

According to [Pei16, Section 4.4], this construction is more rich in structure compared to standard LWE, and thus, a single vector $\boldsymbol{a}_1 = (a_1, \ldots, a_n) \in \mathbb{Z}_q^n$ is enough to construct all the remaining ones [Reg10]. The remaining vectors $\boldsymbol{a}_i$ are then given as $(a_i, \ldots, a_n, -a_1, \ldots, -a_{i-1})$. This fact already hints at one of the advantages that RLWE offers over LWE: The public keys only require $\mathcal{O}(n)$ elements in $\mathbb{Z}_q$, instead of $\mathcal{O}(n^2)$. An additional advantage of the ring-structure is that efficient Fast Fourier Transform (FFT)-based multiplication methods can be deployed, allowing for multiplication in $\mathcal{O}(n \log n)$ integer operations [LPR10].

A formal definition of the problems analog to the LWE variants can be found in Definitions 2.2.3 and 2.2.4.

Definition 2.2.3 (Search Ring Learning with Errors Problem, following [Pei16, Section 4.4])
Let $a_i \in \mathcal{R}_q$ and the fixed secret $s \in \mathcal{R}_q$ with coefficients chosen uniformly at random and error $e_i \in \mathcal{R}_q$ chosen over an error distribution. Given m independent samples $(a_i, b_i) = (a_i, a_i \cdot s + e_i) \in \mathcal{R}_q \times \mathcal{R}_q$, the **Search RLWE problem** is defined as solving for the secret s.

Definition 2.2.4 (Decisional Ring Learning with Errors Problem, following [Pei16, Section 4.4])
Let $a_i \in \mathcal{R}_q$ and the fixed secret $s \in \mathcal{R}_q$ with coefficients chosen uniformly at random, and error $e_i \in \mathcal{R}_q$ chosen over an error distribution. Given m independent samples (a_i, b_i), where b_i is either $(a_i \cdot s + e_i) \in \mathcal{R}_q$ or sampled from a random distribution, the **Decisional RLWE problem** is defined as distinguishing between the two cases.

From a hardness perspective, the results on RLWE are limited to the class of ideal lattices. For those, it has been shown in [LPR10] that the Search RLWE problem is as hard as an approximate version of the SVP (in the worst case).

2.2.3 Module Learning with Errors

The Module Learning with Errors (MLWE) problem, initially also called General Learning with Errors (GLWE) problem [BGV12], is a generalization of the former two. It was proven to be at least as hard as standard lattice problems for so-called module lattices, which lay in between arbitrary and ideal lattices [LS15].

Concretely, this means that $\boldsymbol{s}$ will be sampled from $\mathcal{R}_q^k$ for $k \in \mathbb{N}$. Consequently, samples will have the form $(\boldsymbol{a}, b) \in \mathcal{R}_q^k \times \mathcal{R}_q$. The elements from $\mathcal{R}_q^k$ are called modules, where k is the rank. For $k = m$, we get an instance of LWE, while $k = 1$ yields an instance of RLWE. We give formal descriptions in Definitions 2.2.5 and 2.2.6.

Definition 2.2.5 (Search Module Learning with Errors Problem, based on [Pei16, Section 4.4.3])
Let the secret be $\boldsymbol{s}$ and $\boldsymbol{a}_i$ drawn uniformly random from $\mathcal{R}_q^k$ and $\langle \cdot, \cdot \rangle$ denote the inner product. Let the error $e_i \in \mathcal{R}_q$ be chosen over an error distribution. Given m independent

samples $(\boldsymbol{a}_i, b_i) = (\boldsymbol{a}_i, \langle \boldsymbol{a}_i, \boldsymbol{s} \rangle + e_i) \in \mathcal{R}_q^k \times \mathcal{R}_q$, the **Search MLWE problem** is defined as solving for the secret key, $\boldsymbol{s}$.

Definition 2.2.6 (Decisional Module Learning with Errors Problem, following [Pei16, Section 4.4.3])
Let the secret be $\boldsymbol{s}$ and $\boldsymbol{a}_i$ drawn uniformly random from $\mathcal{R}_q^k$ and $\langle \cdot, \cdot \rangle$ denote the inner product. Let the error $e_i \in \mathcal{R}_q$ be chosen over an error distribution. Given m independent samples $(\boldsymbol{a}_i, b_i)$ where b_i is either $(\langle \boldsymbol{a}_i, \boldsymbol{s} \rangle + e_i) \in \mathcal{R}_q$ or sampled form a random distribution, the **Decisional MLWE problem** is defined as distinguishing between the two cases.

Using this approach allows for flexibly adjusting the hardness of the problem by varying k without the need to modify the underlying ring structure. From an implementation perspective, this is beneficial since the (FFT-based) multiplication only needs to be implemented once, no matter the security level.

2.2.4 Short Integer Solution

The Short Integer Solution (SIS) problem was initially introduced by Ajtai in [Ajt96]. The problem has been proven to be at least as hard as the GapSVP and SIVP – two variants of the SVP – with a very large probability [Pei16].

The problem's parameters are $n, q \in \mathbb{N}$ defining a group $\mathbb{Z}_q^n$ as well as a real $\beta \geq 0 \in \mathbb{R}$ called norm bound, and a number m [Pei16]. The formal definition of the problem can be obtained from Definition 2.2.7.

Definition 2.2.7 (Short Integer Solution Problem, following [Pei16])
Given m uniformly random vectors $\boldsymbol{a}_i \in \mathbb{Z}_q^n$ making up the columns of a matrix $\boldsymbol{A} \in \mathbb{Z}_q^{n \times m}$, find a nonzero integer vector $\boldsymbol{z} \in \mathbb{Z}^m$ with $\|\boldsymbol{z}\| \leq \beta$, such that

$$\sum_{i=0}^{m} \boldsymbol{a}_i z_i = \boldsymbol{0} \in \mathbb{Z}_q^n.$$

Note that omitting the constraint on $\|z\|$ would make the problem trivially solvable using Gaussian elimination.

2.2.5 Ring Short Integer Solution

Analogous to RLWE, the Ring Short Integer Solution (RSIS) problem is a more structured and, thus, compact version of SIS that operates over a polynomial ring $\mathcal{R}_q$. The formal definition is given in Definition 2.2.8.

Definition 2.2.8 (Ring Short Integer Solution Problem, following [Pei16])
Given m uniformly random elements $a_i \in \mathcal{R}_q$ making up a vector $\boldsymbol{a} \in \mathcal{R}_q^m$, find a nonzero integer vector $\boldsymbol{z} \in \mathcal{R}^m$ with $\|\boldsymbol{z}\| \leq \beta$, such that

$$\sum_{i=0}^{m} a_i z_i = 0 \in \mathcal{R}_q.$$

2.2.6 Module Short Integer Solution

The Module Short Integer Solution (MSIS) problem is a generalization of the SIS and RSIS problems. We give the formal definition in Definition 2.2.9.

Definition 2.2.9 (Module Short Integer Solution Problem, following [LS15])
Given m uniformly random elements $\boldsymbol{a}_i \in \mathcal{R}_q^k$, find a nonzero integer vector $\boldsymbol{z} = (z_1, \dots z_m) \in \mathcal{R}^m$ with $\|\boldsymbol{z}\| \leq \beta$, such that

$$\sum_{i=0}^{m} \boldsymbol{a}_i z_i = 0 \in \mathcal{R}_q.$$

2.3 Digital Signatures

Digital signatures are a mechanism to achieve integrity and authenticity in a public-key setting [KL21, Section 13.1]. These signatures allow a signer S, who has a public/secret key pair (pk, sk), to process a message in such a way that they can prove to every other party having a copy of pk, that they are the origin of said message, as well as that the message was not altered during the transmission [KL21, Section 13.1]. An everyday example of this is the sending of signed e-mails. In order to send a signed e-mail to a communication partner, S transmits her public key pk to the receiver via a trusted channel. Then, S computes the "signature" σ over the message M she wants to send under the usage of her secret key sk and attaches the signature to the message. Upon receiving (M, σ), the receiver can verify that σ could have only been created by S and matches M under the use of pk. We give a more formal definition of a digital signature scheme in Definition 2.3.1.

Definition 2.3.1 (Digital Signature Scheme, following [KL21, Section 13.1])
A digital signature scheme consists of three probabilistic polynomial-time algorithms (KGen,Sign,Vf) such that:

- The key generation algorithm KGen takes the security parameter 1^n as its input and returns a key pair (pk, sk), which represent the public and secret key, respectively.
- The signing algorithm Sign takes a secret key sk, as well as a message M as its input and returns a signature σ. We write $\sigma \leftarrow \mathsf{Sign}_{sk}(M)$.
- The verification algorithm Vf takes a public key pk, as well as a message-signature pair (M, σ) as its input and returns $b = 1$ as "Accept" if the signature is valid and $b = 0$ as "Reject" otherwise. We write $b \leftarrow \mathsf{Vf}_{pk}(M, \sigma)$.

It is required that – except with negligible probability – $\mathsf{Vf}_{pk}(M, \mathsf{Sign}_{sk}(M)) = 1, \forall M$, when (pk, sk) was the output of $\mathsf{KGen}(1^n)$ [KL21, Section 13.1].

2.4 Dilithium

Dilithium [LDK+22] is a digital signature scheme that is designed to be secure even in the presence of powerful quantum computers. It is believed to fulfill the Strong Existential Unforgeability under Chosen-Message Attack (SUF-CMA) security notion [LDK+22]. Its security is based on the hard problem of finding short vectors in a lattice [LDK+22]. More precisely, the MLWE, as well as a variant of the MSIS problem called SelfTargetMSIS, are the problems Dilithium's hardness resides on [LDK+22].

2.4.1 Basic Approach

In order to get a better understanding of the construction of Dilithium, we will introduce the foundations on which the scheme is built. As a starting point, let us consider the well-known identification protocol by Schnorr [Sch90], given in Figure 2.1, for which we follow the explanations from [Jus11, Kog19]. In the protocol, the prover can prove to the verifier that she knows the discrete logarithm $a \in \mathbb{Z}_q^*$ of $g^a \in \mathbb{Z}_q^*$. The prover starts by picking a commitment m and computes a witness w from it. This witness is transferred to the verifier, who replies with a uniformly chosen challenge c. From c, the prover can compute s as $ac + m \bmod q$, which the verifier can use to perform the final step of the protocol. An intuition for the correctness can be obtained from expanding z: $g^s y^{-c} = g^{ac+m}(g^a)^{-c} = g^m$, which is exactly equal to w that the verifier checks against.

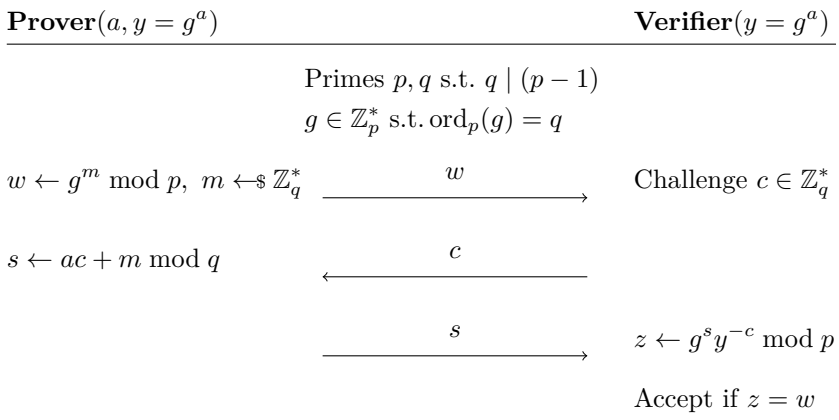


Figure 2.1: Schnorr Identification Protocol, following [Jus11, Kog19].

Fiat-Shamir Heuristic

A first step towards a lattice-based signature scheme can be taken by applying the Fiat-Shamir Heuristic [FS87] to an identification protocol. The technique introduced by Fiat and Shamir in 1987 can be applied to turn an interactive proof of knowledge into a non-interactive one. In the concrete case of the protocol from Figure 2.1, the challenge c

is replaced by the output of a hash function, removing the necessity for the verifier to send a message to the prover. This is the commonly desired setup for digital signatures.

A representation of the Schnorr signature scheme [Sch91], following the notation from [Jus11], can be found in Algorithms 2.4.1 to 2.4.3.

Algorithm 2.4.1: Schnorr signature: Key generation, following [Jus11].

Input : $\tau, \lambda \in \mathbb{Z}, \tau > \lambda$
Output : Public key $pk = (p, q, g, y, H)$, secret key $sk = (p, q, g, \alpha, H)$

1 $q \leftarrow\$ \{0,1\}^{\lambda}$
2 $p \leftarrow\$ \{0,1\}^{\tau}$ s.t. $q \mid (p-1)$
3 $g \in \mathbb{Z}_p^*$ of order q
4 $\alpha \leftarrow\$ [1, q]$
5 $y \leftarrow g^{\alpha} \in \mathbb{Z}_p^*$
6 Let H be a hash function $H : \{0,1\}^* \rightarrow \mathbb{Z}_q$

Algorithm 2.4.2: Schnorr signature: Verification, following [Jus11].

Input : Public key $pk = (p, q, g, y, H)$, message/signature pair $(m, (s, c))$
Output : Accept/Reject

1 $v \leftarrow g^s y^{-c} \in \mathbb{Z}_p$
2 **if** $c = H(m \| v)$ **then**
3 | Accept.
4 **else**
5 | Reject.

Algorithm 2.4.3: Schnorr signature: Signing, following [Jus11].

Input : Secret key $sk = (p, q, g, \alpha, H)$, message $m \in \{0,1\}^*$
Output : Signature $(s, c) \in \mathbb{Z}_q^2$ of m

1 $k \leftarrow\$ \mathbb{Z}_p^*$
2 $r \leftarrow g^k \in Z_p^*$
3 $c \leftarrow H(m \| r) \in \mathbb{Z}_q$
4 $s \leftarrow \alpha c + k \in \mathbb{Z}_q$

While the key generation is highly similar to the setup for Schnorr's identification protocol, the crucial difference resides inside the signature generation in Algorithm 2.4.3. At first, similarly to the identification protocol, the prover picks a commitment k and computes a witness r from it. However, instead of sending the witness to the verifier, she hashes it alongside the message to be signed and uses the output as a challenge.

We can get an intuition of why the verification in Algorithm 2.4.2 accepts every proper message/signature pair by expanding: $v = \beta^s y^{-c} = \beta^{\alpha c + k} y^{-c} = (y^c \beta^k) y^{-c} = \beta^k \in \mathbb{Z}_p$. Thus, $H(m \| v) = H(m \| \beta^k)$.

Fiat-Shamir with Aborts

In Algorithm 2.4.3, we can intuitively understand k as a form of "mask" to conceal the value of αc [Lyu09]. This is required since, in the case of αc being leaked, α could straightforwardly be computed as c is public. In the concrete case of Schnorr's scheme, k can conveniently be picked uniformly random, as the group in which this operation takes place is finite.

As Lyubashevsky notes in [Lyu09], this is not always the case: For protocols like, for example, Girault's identification protocol [Gir91], the multiplication of αc is performed over the integers instead of over $\mathbb{Z}_q$. An example in [Lyu09] outlines that for $\alpha c \in [0, R]$, k could be picked to be in $[0, 2^{64}R]$ such that with a high probability, $\alpha c + k \in [R, 2^{64}R]$. This way, αc is properly masked and an attacker could not infer anything about it without knowledge of k.

For lattice-based constructions, the situation is similar, with the difference that the "trick" of picking k from a much larger range is not feasible due to efficiency concerns [Lyu09]. The novel idea introduced in [Lyu09] solves this problem: Picking the mask from a rather small range, i.e., $k \in [0, \ldots, 2R]$, but aborting the process in case the result $\alpha c + k$ is not inside the desired range of $[R, \ldots, 2R]$. This way, no "weakly" masked values for s are leaked.

2.4.2 Dilithium Template

This section brings together the basic principles previously introduced with a simplified version of the DILITHIUM signature scheme, following the explanations from [LDK+22].

Algorithm 2.4.4: DILITHIUM signature template: Key generation [LDK+22].

Input :
Output : Secret key $sk = (\boldsymbol{A}, \boldsymbol{t}, \boldsymbol{s}_1, \boldsymbol{s}_2)$, public key $pk = (\boldsymbol{A}, \boldsymbol{t})$

1 $\boldsymbol{A} \leftarrow\$ \mathcal{R}_q^{k \times l}$
2 $(\boldsymbol{s}_1, \boldsymbol{s}_2) \leftarrow\$ S_\eta^l \times S_\eta^k$
3 $\boldsymbol{t} \leftarrow \boldsymbol{A}\boldsymbol{s}_1 + \boldsymbol{s}_2$

Algorithm 2.4.5: DILITHIUM signature template: Verification [LDK+22].

Input : $pk = (\boldsymbol{A}, \boldsymbol{t})$, message M, signature $\sigma = (\boldsymbol{z}, c)$
Output : Accept/Reject

1 $\boldsymbol{w}_1' \leftarrow \mathsf{HighBits}(\boldsymbol{A}\boldsymbol{z} - c\boldsymbol{t}, 2\gamma_2)$
2 **if** $\|\boldsymbol{z}\|_\infty < \gamma_1 - \beta$ *and* $c = \mathsf{H}(M \| \boldsymbol{w}_1')$ **then**
3 | Accept.
4 **else**
5 | Reject.

Algorithm 2.4.6: DILITHIUM signature template: Signing [LDK+22].

Input : $sk = (\boldsymbol{A}, \boldsymbol{t}, \boldsymbol{s}_1, \boldsymbol{s}_2)$, message M
Output : Signature $\sigma = (\boldsymbol{z}, c)$

1 $\boldsymbol{z} \leftarrow \perp$
2 **while** $\boldsymbol{z} = \perp$ **do**
3 | $\boldsymbol{y} \leftarrow\$ S_{\gamma_1 - 1}^l$
4 | $\boldsymbol{w}_1 \leftarrow \mathsf{HighBits}(\boldsymbol{A}\boldsymbol{y}, 2\gamma_2)$
5 | $c \in B_\tau \leftarrow \mathsf{H}(M \| \boldsymbol{w}_1)$
6 | $\boldsymbol{z} \leftarrow \boldsymbol{y} + c\boldsymbol{s}_1$
7 | **if** $\|\boldsymbol{z}\|_\infty \geq \gamma_1 - \beta$ *or* $\|\mathsf{LowBits}(\boldsymbol{A}\boldsymbol{y} - c\boldsymbol{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ **then**
8 | | $\boldsymbol{z} := \perp$

Key Generation

First, consider Algorithm 2.4.4, where we pick $\boldsymbol{A}, \boldsymbol{s}_1, \boldsymbol{s}_2$ such that they contain elements from the ring $\mathcal{R}_q$, where $\boldsymbol{s}_1, \boldsymbol{s}_2$ are picked such that they have small coefficients. $\boldsymbol{t}$ is then computed so that $(\boldsymbol{A}, \boldsymbol{t})$ make up an MLWE sample with $\boldsymbol{s}_1$ resembling the secret and $\boldsymbol{s}_2$ the error. In the context of Schnorr signatures, $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ correspond to the secret α, while $(\boldsymbol{A}, \boldsymbol{t})$ can be thought of as the public $y = g^\alpha$.

Signature Generation

The signature generation in Algorithm 2.4.6 clearly exhibits the deployment of the Fiat-Shamir heuristic, computing the challenge c from a hash function in Line 5. Here, $\boldsymbol{y}$ takes the role of the mask to the signature, which corresponds to k in Schnorr's scheme. In contrast to Schnorr signatures, the sampling of the mask $\boldsymbol{y}$ in Line 3 does not guarantee $\boldsymbol{z}$ to not leak the secret key, and thus, the aborting technique from [Lyu09] is used. The first check that is performed is to ensure security: β represents the maximum value for a coefficient from $c\boldsymbol{s}_i$, while every coefficient from $\boldsymbol{y}$ is less than γ_1. If a coefficient in $\boldsymbol{z}$ exceeds $\gamma_1 - \beta$, we need to reject. The second check ensures security and correctness [LDK+22] as it is required that there will be no carry caused by the low bits for the verification to succeed. The parameters are chosen so that, on average, four iterations of the rejection sampling loop are performed [LDK+22]. Note that the number of iterations can vary significantly in practice. The $\mathsf{HighBits}$ function can be thought of as a rounding function that rounds to a nearby multiple of its second argument.

Verification

In the verification, the first check verifies that the signature is of a proper form, while the second check verifies that the signature matches the message M.

The verification from Algorithm 2.4.5 is correct when it holds that $\mathsf{HighBits}(\boldsymbol{A}\boldsymbol{z} - c\boldsymbol{t}, 2\gamma_2) = \mathsf{HighBits}(\boldsymbol{A}\boldsymbol{y}, 2\gamma_2)$. First, we can note that $\boldsymbol{A}\boldsymbol{y} - c\boldsymbol{s}_2$ is the same as $\boldsymbol{A}\boldsymbol{z} - c\boldsymbol{t}$ [LDK+22]. Thus, we only need to show that $\mathsf{HighBits}(\boldsymbol{A}\boldsymbol{y}, 2\gamma_2) = \mathsf{HighBits}(\boldsymbol{A}\boldsymbol{y} - c\boldsymbol{s}_2, 2\gamma_2)$. Since every valid signature fulfills $\|\mathsf{LowBits}(\boldsymbol{A}\boldsymbol{y} - c\boldsymbol{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta$ and every coefficient in $c\boldsymbol{s}_2$ is smaller than β, adding $c\boldsymbol{s}_2$ will not cause a carry from the low bits, to the high bits and thus, the verification algorithm will accept.

2.4.3 Dilithium in Detail

In this section, we turn to the full version of DILITHIUM, going over its parameter sets, the algorithms for key generation, signing, and verification, as well as several of the supporting functions.

Parameters

The polynomial ring DILITHIUM operates on is chosen as $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$, where q is $8380417 = 2^{23} - 2^{13} + 1$ and $n = 256$ [LDK+22]. From this point onwards, mentions

of q or n in the context of Dilithium will always refer to those choices, except when explicitly stated differently.

Dilithium offers three different parameter sets, all of which operate over the same polynomial ring $\mathcal{R}_q$. This is possible because the scheme is based on MLWE, meaning the scaling of hardness can be achieved by varying the module rank, i.e., picking vectors and matrices of different sizes. An overview of the different parameter sets can be obtained from Table 2.1. The dimensions for the matrix are given by (k, l), η reflects the range the secret polynomials' coefficients are sampled from, τ is defined as the number of coefficients set to ± 1 in the challenge polynomial c, and #reps refers to the average number of iterations of the "Fiat-Shamir with Aborts"-induced rejection sampling loop [LDK+22, Table 2]. While γ_1 defines the range the coefficients $\boldsymbol{y}$ is sampled from, γ_2 is relevant concerning the low-order rounding [LDK+22, Table 2].

Table 2.1: Overview of Dilithium's parameter sets [LDK+22].

Scheme	NIST level	$\mid pk \mid$	$\mid sig \mid$	(k, l)	η	τ	γ_1	γ_2	#reps
Dilithium2	2	1312 B	2420 B	$(4, 4)$	2	39	2^{17}	$(q-1)/88$	4.25
Dilithium3	3	1952 B	3293 B	$(6, 5)$	4	49	2^{19}	$(q-1)/32$	5.1
Dilithium5	5	2592 B	4595 B	$(8, 7)$	2	60	2^{19}	$(q-1)/32$	3.85

Algorithms

Algorithms 2.4.7 to 2.4.9 show the full version of Dilithium as defined by [LDK+22]. Compared to the scheme outlined in Algorithms 2.4.4 to 2.4.6, the most notable differences are given below [LDK+22, Section 1.2]:

- Matrix $\boldsymbol{A}$ is not part of the keys, as it would have a rather large representation. It is generated from a seed ρ using SHAKE-128 on invocation.
- A trade-off between signature and public key size is made: Some low-order bits of $\boldsymbol{t}$ are not included in the key. Instead, information to recover this loss of information is included in the signature itself, called "hint".
- Inclusion of NTT-based polynomial multiplication into the specification: E.g., matrix $\boldsymbol{A}$ is already sampled into NTT-domain to save the cost of transformation. Here, "$\circ$" explicitly refers to the pointwise multiplication inside the NTT-domain.
- Various optimizations: E.g., M is pre-hashed outside of the rejection loop.

The hash functions H and CRH are instantiated with SHAKE-256 throughout the whole execution. The function $\mathsf{ExpandA}$ generates the matrix $\boldsymbol{A} \in \mathcal{R}_q^{k \times l}$ from a seed $\rho \in \{0,1\}^{256}$ using SHAKE-128 such that it is already inside NTT-domain, denoted as $\hat{\boldsymbol{A}}$. $\mathsf{ExpandMask}$ deterministically samples the "mask" $\boldsymbol{y}$ using SHAKE-256 with seed ρ' and a nonce κ,

Algorithm 2.4.7: DILITHIUM: Key generation, following [LDK+22].

Output : Secret key $sk = (\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0)$, public key $pk = (\rho, \boldsymbol{t}_1)$

$\zeta \leftarrow\$ \{0,1\}^{256}$
$(\rho, \varsigma, K) \in \{0,1\}^{256\times 3} \leftarrow \mathsf{H}(\zeta)$
$(\boldsymbol{s}_1, \boldsymbol{s}_2) \in S_\eta^l \times S_\eta^k \leftarrow \mathsf{H}(\varsigma)$
$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k\times l} \leftarrow \mathsf{ExpandA}(\rho)$
$\boldsymbol{t} \leftarrow \mathsf{INTT}(\hat{\boldsymbol{A}} \circ \mathsf{NTT}(\boldsymbol{s}_1)) + \boldsymbol{s}_2$
$(\boldsymbol{t}_1, \boldsymbol{t}_0) \leftarrow \mathsf{Power2Round}(\boldsymbol{t}, 13)$
$tr \in \{0,1\}^{384} \leftarrow \mathsf{CRH}(\rho \| \boldsymbol{t}_1)$
return (pk, sk)

Algorithm 2.4.8: DILITHIUM: Verification, following [LDK+22].

Input : Public key $pk = (\rho, \boldsymbol{t}_1)$, message: $M \in \{0,1\}^*$, signature $\sigma = (\boldsymbol{z}, \boldsymbol{h}, \tilde{c})$
Output : Accept/Reject

$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k\times l} \leftarrow \mathsf{ExpandA}(\rho)$
$c \leftarrow \mathsf{SampleInBall}(\tilde{c})$
$\mu \in \{0,1\}^{384} \leftarrow \mathsf{CRH}(\mathsf{CRH}(\rho \| \boldsymbol{t}_1) \| M)$
$\boldsymbol{w}_1' \leftarrow \mathsf{UseHint}(\boldsymbol{h}, \mathsf{INTT}(\hat{\boldsymbol{A}} \circ \mathsf{NTT}(\boldsymbol{z}) - \mathsf{NTT}(c) \circ \mathsf{NTT}(2^d \cdot \boldsymbol{t}_1)))$
if $\|\boldsymbol{z}\|_\infty < \gamma_1 - \beta$ and $\tilde{c} = \mathsf{H}(\mu \| \boldsymbol{w}_1')$ and # of 1's in $\boldsymbol{h} \leq \omega$ **then**
  return Accept.
else
  return Reject.

which gets updated with every rejected signature candidate. The remaining supporting algorithms used in Algorithms 2.4.7 to 2.4.9 are defined in Algorithms 2.4.10 to 2.4.16. The randomness used in the variant of the Fisher-Yates shuffle in Algorithm 2.4.14 is provided by an Extended Output Function (XOF), SHAKE-256 in the case of Dilithium, based on the seed ρ [LDK+22, Figure 2].

Algorithm 2.4.10: Decompose, following [LDK+22].

Input : r, α

$r \leftarrow r \bmod^+ q$
$r_0 \leftarrow r \bmod^\pm \alpha$
if $r - r_0 = q - 1$ **then**
  $r_1 := 0, r_0 := r_0 - 1$
else
  $r_1 := (r - r_0)/\alpha$
return (r_1, r_0)

Algorithm 2.4.11: LowBits, following [LDK+22].

Input : r, α

$(r_1, r_0) \leftarrow \mathsf{Decompose}(r, \alpha)$
return r_0

Algorithm 2.4.12: HighBits, following [LDK+22].

Input : r, α

$(r_1, r_0) \leftarrow \mathsf{Decompose}(r, \alpha)$
return r_1

Algorithm 2.4.9: DILITHIUM: Signing, following [LDK+22].

Input : secret key $sk = (\rho, K, tr, \boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{t}_0)$, message: $M \in \{0,1\}^*$
Output : signature $\sigma = (\boldsymbol{z}, \boldsymbol{h}, \tilde{c})$

$\hat{\boldsymbol{A}} \in \mathcal{R}_q^{k \times l} \leftarrow \mathsf{ExpandA}(\rho)$
$\mu \in \{0,1\}^{512} \leftarrow \mathsf{CRH}(tr \| M)$
$\kappa \leftarrow 0, (\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
$\rho' \in \{0,1\}^{512} \leftarrow \mathsf{CRH}(K \| \mu)$
$\hat{\boldsymbol{s}}_1 \in \mathcal{R}_q^l \leftarrow \mathsf{NTT}(\boldsymbol{s}_1)$, $\hat{\boldsymbol{s}}_2 \in \mathcal{R}_q^k \leftarrow \mathsf{NTT}(\boldsymbol{s}_2)$, $\hat{\boldsymbol{t}}_0 \in \mathcal{R}_q^k \leftarrow \mathsf{NTT}(\boldsymbol{t}_0)$
while $(\boldsymbol{z}, \boldsymbol{h}) = \perp$ **do**
  $\boldsymbol{y} \in \tilde{S}_{\gamma_1}^l \leftarrow \mathsf{ExpandMask}(\rho', \kappa)$
  $\boldsymbol{w} \in \mathcal{R}_q^k \leftarrow \mathsf{INTT}(\hat{\boldsymbol{A}} \circ \mathsf{NTT}(\boldsymbol{y}))$
  $\boldsymbol{w}_1 \in \mathcal{R}_q^k \leftarrow \mathsf{HighBits}(\boldsymbol{w}, 2\gamma_2)$
  $\tilde{c} \in \{0,1\}^{256} \leftarrow \mathsf{H}(\mu \| \boldsymbol{w}_1)$
  $c \leftarrow \mathsf{SampleInBall}(\tilde{c})$
  $\hat{c} \leftarrow \mathsf{NTT}(c)$
  $\boldsymbol{z} \in \mathcal{R}_q^l \leftarrow \boldsymbol{y} + \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{s}}_1)$
  $\boldsymbol{r}_0 \in \mathcal{R}_q^k \leftarrow \mathsf{LowBits}(\boldsymbol{w} - \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{s}}_2), 2\gamma_2)$
  if $\|\boldsymbol{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\boldsymbol{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
  else
    $\boldsymbol{h} \leftarrow \mathsf{MakeHint}(-\mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{t}}_0), \boldsymbol{w} - \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{s}}_2 + \mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{t}}_0)), 2\gamma_2)$
    if $\|\mathsf{INTT}(\hat{c} \circ \hat{\boldsymbol{t}}_0)\|_\infty \geq \gamma_2$ or # of 1's in $\boldsymbol{h} > \omega$ **then**
      $(\boldsymbol{z}, \boldsymbol{h}) \leftarrow \perp$
  $\kappa \leftarrow \kappa + l$
return $\sigma = (\boldsymbol{z}, \boldsymbol{h}, \tilde{c})$

Algorithm 2.4.13: UseHint, following [LDK+22].

Input : h, r, α
$m \leftarrow (q-1)/\alpha$
$(r_1, r_0) \leftarrow \mathsf{Decompose}(r, \alpha)$
if $h = 1$ *and* $r_0 > 0$ **then return** $(r_1 + 1) \bmod^+ m$;
if $h = 1$ *and* $r_0 \leq 0$ **then return** $(r_1 - 1) \bmod^+ m$;
return r_1

Algorithm 2.4.14: SampleInBall, following [LDK+22].

Input : ρ
Initialize $\boldsymbol{c} = c_0 c_1 \ldots c_{255} = 00 \ldots 0$
for $i := 256 - \tau$ *to* 255 **do**
  $j \leftarrow\$ \{0, 1, \ldots, i\}$
  $s \leftarrow\$ \{0,1\}$
  $c_i \leftarrow c_j$
  $c_j \leftarrow (-1)^s$
return c

Algorithm 2.4.15: Power2Round, following [LDK+22].

Input : r, d

1 $r \leftarrow r \bmod^{+} q$

2 $r_0 \leftarrow r \bmod^{\pm} 2^d$

3 **return** $((r - r_0)/2^d, r_0)$

Algorithm 2.4.16: MakeHint, following [LDK+22].

Input : z, r, a

1 $r_1 \leftarrow \mathsf{HighBits}(r, \alpha)$

2 $v_1 \leftarrow \mathsf{HighBits}(r + z, \alpha)$

3 **return** $[\![r_1 \neq v_1]\!]$

2.4.4 ML-DSA

A draft for the standardization of Dilithium by NIST was published in August 2023 [NIS23b]. The standardized variant of Dilithium will be called ML-DSA. Because the algorithm only minorly differs from its original from [LDK+22] and the draft of the standard was only published after the majority of this work had been finished, we will not consider details or the implementation of ML-DSA.

2.5 Polynomial Arithmetic

Since polynomial arithmetic, specifically, multiplication, belongs to the most time-consuming operations for Dilithium [LDK+22], we will introduce means for fast polynomial multiplication. As Dilithium operates over vectors and matrices from the polynomial ring $\mathcal{R}_q$, it was constructed such that it can profit from FFT-based polynomial multiplication using the NTT. This section follows the explanations from [Abd21], which is based on [Pöp16].

2.5.1 Schoolbook Multiplication

In order to recap the base-case for polynomial multiplication, we consider the schoolbook approach first. This algorithm takes $\mathcal{O}(n^2)$ coefficient multiplications to multiply two polynomials $a, b \in \mathbb{Z}[X]$ of degree n, where e.g. $a = \sum_{i=0}^{n} a_i X^i$. To retrieve the result polynomial c of degree $2n$, we compute:

$$c = a \cdot b = \sum_{i=0}^{n} \sum_{j=0}^{n} a_i b_j X^{i+j}.$$

2.5.2 The Number-Theoretic Transform

The number of coefficient operations can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ by making use of the NTT, a variation of the Discrete Fourier Transform (DFT) defined over finite fields [Pol71]. As described in [Win96, Section 3.3], the basic concept is to regard the polynomial multiplication as a convolution of the coefficient vectors of two polynomials, which becomes a pointwise multiplication once the polynomials are transformed into the NTT-domain. Definition 2.5.1 provides a formal definition of the NTT and its inverse, the INTT. Note that ω^{-1} as well as n^{-1} exist due to q being a prime.

Definition 2.5.1 (Number-Theoretic Transform, following [Nus82, Section 8])
Let q be a prime, ω a primitive n-th root of unity, and $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ a coefficient vector. The NTT of $\boldsymbol{a}$, denoted by $\hat{\boldsymbol{a}} = \mathsf{NTT}(\boldsymbol{a})$, where $\hat{\boldsymbol{a}} = (\hat{a}_0, \ldots, \hat{a}_{n-1})$ is defined as

$$\hat{a}_i = \sum_{j=0}^{n-1} a_j \cdot \omega^{-ij} \bmod q, \text{ for } i \in [0, n-1].$$

The inverse NTT, written as $\boldsymbol{a} = \mathsf{INTT}(\hat{\boldsymbol{a}})$, is defined as

$$a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \cdot \omega^{-ij} \bmod q, \text{ for } i \in [0, n-1].$$

Informally speaking, a convolution can be thought of as multiplying each element from one vector with each element of another vector and then summing up the results from indices. A concrete definition is given in Definition 2.5.2.

Definition 2.5.2 (Convolution, following [Win96, Section 3.3])
The convolution of two vectors, denoted by $\boldsymbol{c} = \boldsymbol{a} \odot \boldsymbol{b}$, with $(\boldsymbol{a}, \boldsymbol{b}) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^n$ and $\boldsymbol{c} \in \mathbb{Z}_q^{2n}$ is defined via the elements of $\boldsymbol{c}$

$$c_i = \sum_{j=0}^{n-1} a_j b_{i-j}, \text{ where } b_k = 0 \text{ for } k \notin [0, n-1].$$

As also noted in [Pöp16], NTTs that support circular convolutions are the ones relevant to polynomial multiplication. The two types of circular convolutions are defined in Definition 2.5.3. Most notably, multiplying two polynomials in $\mathbb{Z}[X]/(X^n+1)$ is equivalent to computing the negative wrapped convolution of their coefficient vectors [Pöp16], meaning the results will be implicitly reduced modulo $(X^n + 1)$. This is also desired in the case of Dilithium. An NTT of length n over $\mathbb{Z}_q$ that supports such convolutions exists if and only if $n \mid (q-1)$ [Nus82, Theorem 8.2].

Definition 2.5.3 (Circular Convolution, following [Win96, Section 3.3])
Let $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$, $\boldsymbol{b} = (b_0, \ldots, b_{n-1}) \in \mathbb{Z}_q^n$.

The positive wrapped convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$ is the vector $\boldsymbol{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{Z}_q^n$ where

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

The negative wrapped convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$ is the vector $\boldsymbol{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{Z}_q^n$ where

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

Theorem 2.5.4 brings together the previously defined constructs and describes the convolution, i.e., polynomial multiplication, in terms of pointwise multiplication of the coefficients, which have been transformed into NTT-domain.

Theorem 2.5.4 (Convolution Theorem, following [Win96, Section 3.3])
Let ω be a $2n$-th primitive root of unity in $\mathbb{Z}_q$ used in NTT and INTT. Let $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$, $\boldsymbol{b} = (b_0, \ldots, b_{n-1}) \in \mathbb{Z}_q^n$ and $\boldsymbol{a}', \boldsymbol{b}'$ be their to a length of $2n$ zero-extended counterparts. Then, $\boldsymbol{a} \odot \boldsymbol{b} = \mathsf{INTT}(\mathsf{NTT}(\boldsymbol{a}') \circ \mathsf{NTT}(\boldsymbol{b}'))$, where "$\circ$" means pointwise multiplication.

One disadvantage of straightforwardly applying the method from Theorem 2.5.4 is the doubled size of the coefficient vectors due to the zero extension. To work around this limitation, it is common to proceed as in Theorem 2.5.5, which removes the need for zero extension and explicitly states how to compute the negative wrapped convolution, as also explained in [Pöp16]. Note that this approach requires the existence of a $2n$-th primitive root of unity.

Theorem 2.5.5 (Negative Wrapped Convolution, following [Win96, Theorem 3.3.7])
For a primitive n-th root of unity ω in $\mathbb{Z}_q$ and $\psi^2 = \omega$. Let $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$, $\boldsymbol{b} = (b_0, \ldots, b_{n-1}) \in \mathbb{Z}_q^n$, and $\boldsymbol{d} = (d_0, \ldots, d_{n-1}) \in \mathbb{Z}_q^n$ be the negative wrapped convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$. Let $\bar{\boldsymbol{a}}, \bar{\boldsymbol{b}}, \bar{\boldsymbol{d}}$ be defined as $(a_0, \psi a_1, \ldots, \psi^{n-1} a_{n-1})$, $(b_0, \psi b_1, \ldots, \psi^{n-1} b_{n-1})$, and $(d_0, \psi d_1, \ldots, \psi^{n-1} d_{n-1})$, respectively. Then, $\bar{\boldsymbol{d}} = \mathsf{INTT}(\mathsf{NTT}(\bar{\boldsymbol{a}}) \circ \mathsf{NTT}(\bar{\boldsymbol{b}}))$, where "$\circ$" means pointwise multiplication.

Though, computing the polynomial multiplication as described in Theorem 2.5.5 and transforming in NTT-domain according to Definition 2.5.1 would not yield a performance benefit over the schoolbook approach as we still require $\mathcal{O}(n^2)$ integer multiplication operations. Thus, Section 2.5.3 introduces algorithms to compute the NTT more efficiently.

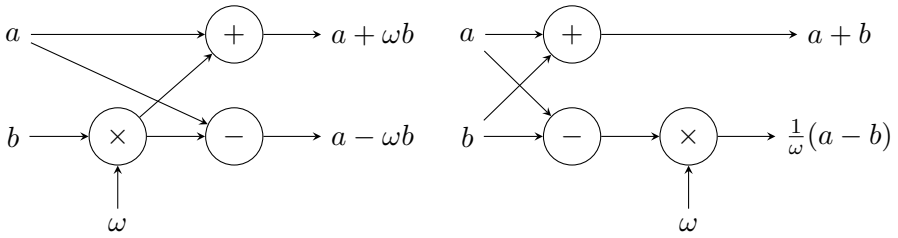
2.5.3 Fast NTT Algorithms

A fast method to compute the DFT was already introduced by Gauss in 1805 [Gau66]. The FFT variants that are most frequently used in implementations of the NTT today are the Cooley–Tukey (CT) [CT65] and Gentleman–Sande (GS) [GS66] algorithms, where the former is commonly used for the computation of the NTT, while the latter is chosen for the INTT. The general approach both algorithms follow is a divide-and-conquer pattern, where they split the computation of an NTT of size $2n$ into two NTTs of size n, which is enabled by the Chinese Remainder Theorem (CRT). By doing so, the NTT can be computed in only $\mathcal{O}(n \log n)$ integer operations. In the following, we will illustrate this splitting idea based on the description in [Kan22, Section 2.2.5].

For the CT algorithm, a ring $\mathbb{Z}_q[X]/(X^{2^k} - c^2)$ is split into $\mathbb{Z}_q[X]/(X^{2^{k-1}} - c)$ and $\mathbb{Z}_q[X]/(X^{2^{k-1}} + c)$. The inverse applies for the GS algorithm, where $\mathbb{Z}_q[X]/(X^{2^k} - c^2)$ is constructed from $\mathbb{Z}_q[X]/(X^{2^{k-1}} - c)$ and $\mathbb{Z}_q[X]/(X^{2^{k-1}} + c)$ using the CRT. More generally, this can be expressed as a ring isomorphism $\phi : \mathbb{Z}_q[X]/(f(X)g(X)) \cong \mathbb{Z}_q[X]/(f(X)) \times \mathbb{Z}_q[X]/(g(X))$, where $\phi(h) = (h \bmod f, h \bmod g)$.

In the case where $f(X) = X^n - c$ and $f(X) = X^n + c$, the transformation becomes

$$\phi\left(\sum_{i=0}^{2n-1} h_i X^i\right) = \left(\sum_{i=0}^{n-1} \underbrace{(h_i + c h_{n+i})}_{\text{CT}} X^i, \sum_{i=0}^{n-1} \underbrace{(h_i - c h_{n+i})}_{\text{CT}} X^i\right) \tag{2.3}$$



(a) Cooley–Tukey butterfly. (b) Gentleman–Sande butterfly.

Figure 2.2: NTT butterfly operations, visualization from [Abd21].

while its inverse can be described by

$$\phi^{-1}\left(\sum_{i=0}^{n-1} h'_i X^i, \sum_{i=0}^{n-1} h''_i X^i\right) = \sum_{i=0}^{n-1} \frac{1}{2} \underbrace{(h'_i + h''_i)}_{\text{GS}} X^i + \sum_{i=0}^{n-1} \frac{1}{2} \underbrace{\frac{1}{c}(h'_i - h''_i)}_{\text{GS}} X^{n+i}. \tag{2.4}$$

As we can see from Equations (2.3) and (2.4), one polynomial of degree $2n$ can be split into two polynomials of degree n. For a reduction polynomial $(X^n + 1)$, where n is a power of two, this technique can be applied recursively $\log n$ times until we are left with polynomials of degree 0 and can perform pointwise multiplication, as also indicated in Theorem 2.5.5. Note that this requires the existence of a $2n$-th root of unity. The factor of $\frac{1}{2}$ in the inverse transformation resembles the factor of n^{-1} over all splits that can be found in Definition 2.5.1. An illustration of the characteristic CT- and GS-"butterfly" operations, which are also highlighted in Equations (2.3) and (2.4), can be found in Figure 2.2. We can note that the most important operations for computing the butterfly computations are (modular) additions, subtractions, and multiplications with powers of the root of unity, also named "twiddle factors".

Most implementations do not follow the recursive approach mentioned before but an iterative and in-place one. Algorithms 2.5.1 and 2.5.2 reflect such an implementation for the NTT and INTT, following the notation form [LN16]. For a sequence of values to be in bit-reversed order means that, for example, for $a \in \mathcal{R}_q$ with its coefficient embedding $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$, the coefficient a_i will be placed at index $\mathsf{BitRev}(i)$, where BitRev performs a bit-reversal on $\lceil \log n \rceil$ bits. For example, for $n = 256$, the coefficient at index $5 = (00000101)_2$ will be placed to index $(10100000)_2 = 160$. However, the bit-reversal is not of relevance for the application in Dilithium because the NTT from Algorithm 2.5.1 outputs in bit-reversed order while INTT from Algorithm 2.5.2 takes this representation as the input and outputs in regular order again. The ordering of the coefficients during the pointwise multiplication does not matter as long it is consistent between its inputs.

2.5.4 Polynomial Multiplication for Dilithium

As previously mentioned, Dilithium was constructed in such a way that it can profit from the fast polynomial multiplication strategies introduced in Sections 2.5.2 and 2.5.3,

Algorithm 2.5.1: NTT based on the CT-butterfly, following [LN16, Algorithm 1].

Input : $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ with q prime and $q \equiv 1 \bmod 2n$, $n = 2^k$ for $k \in \mathbb{N}$, precomputed table Ψ^{rev} built from powers of ψ in bit-reversed order, for ψ a $2n$-th root of unity

Output : $\boldsymbol{a} \leftarrow \mathsf{NTT}(\boldsymbol{a})$ inplace and in bit-reversed order

1 $t \leftarrow n$
2 **for** $m = 1;\ m < n;\ m = 2m$ **do**
3 $t \leftarrow t/2$
4 **for** $i = 0;\ i < m;\ i++$ **do**
5 $j_1 \leftarrow 2 \cdot i \cdot t$
6 $j_2 \leftarrow j_1 + t - 1$
7 $S \leftarrow \Psi_{m+i}^{rev}$
8 **for** $j = j_1;\ j \leq j_2;\ j++$ **do**
9 $U \leftarrow a_j$
10 $V \leftarrow a_{j+t} \cdot S \bmod q$
11 $a_j \leftarrow U + V \bmod q$
12 $a_{j+t} \leftarrow U - V \bmod q$
13 **return** $\boldsymbol{a}$

with the specification already including the NTT-based approach as it can bee seen from Algorithms 2.4.7 to 2.4.9. Its modulus fulfills the criterion $2n \mid (q-1)$ as $(8380417 - 1) \equiv 0 \bmod 512$, and therefore, a negative wrapped convolution supporting NTTs, as given in Theorem 2.5.5, exists. The existence of the $2n$-th root of unity $\psi = 1753$ [LDK+22] further allows for a "full" splitting down to polynomials of degree 0. With $\log n = 8$, we say that we compute eight "layers" of NTT, of which each consists of 128 butterfly operations on coefficient pairs. The iteration over the layers is also reflected in Line 2 of Algorithm 2.5.1 and Line 2 from Algorithm 2.5.2.

While in [AHKS22] the Fermat Number Transform (FNT), a variant of the NTT, enabled faster multiplication for polynomials with small coefficients, we will not consider this approach in the following.

2.5.5 Kronecker Substitution

Another approach for polynomial multiplication, especially on platforms with hardware acceleration for classical, asymmetric cryptography, is a technique based on Kronecker substitution. Informally speaking, the technique can be used to reduce polynomial multiplications to (big-)integer multiplications. The substitution method itself dates back to 1882 [Kro82], while Schönhage also proposed a similar approach to polynomial multiplication in [Sch82]. Building on the work by Harvey [Har09], Albrecht, Hanser, Hoeller, Pöppelmann, Virdia, and Wallner applied the Kronecker method to an RLWE scheme under the use of an RSA coprocessor [AHH+19]. A recent work in this direction was published by Bos, Renes, and van Vredendaal in 2022 [BRv22], confirming a finding by

Algorithm 2.5.2: INTT based on the GS-butterfly, following [LN16, Algorithm 2].

Input : $\boldsymbol{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ in bit-reversed order with q prime and $q \equiv 1 \bmod 2n$, $n = 2^k$ for $k \in \mathbb{N}$, pre-computed table $\Psi^{-1,rev}$ built from powers of ψ^{-1} in bit-reversed order, for ψ a $2n$-th root of unity

Output : $\boldsymbol{a} \leftarrow \mathsf{INTT}(\boldsymbol{a})$ inplace

```
1  t ← 1
2  for m = n; m > 1; m = m/2 do
3  |  j1 ← 0
4  |  h ← m/2
5  |  for i = 0; i < h; i++ do
6  |  |  j2 ← j1 + t − 1
7  |  |  S ← Ψ^{−1,rev}_{h+i}
8  |  |  for j = j1; j ≤ j2; j++ do
9  |  |  |  U ← a_j
10 |  |  |  V ← a_{j+t}
11 |  |  |  a_j ← U + V mod q
12 |  |  |  a_{j+t} ← (U − V) · S mod q
13 |  |  j1 ← j1 + 2t
14 |  t ← 2t
15 for j = 0; j < n; j++ do
16 |  a_j = a_j · n^{−1} mod q
17 return a
```

Albrecht et al.: For the current state of Kyber and Dilithium, where the public matrix $\boldsymbol{A}$ is sampled directly in NTT-domain as $\hat{\boldsymbol{A}}$, it is of no use to apply Kronecker substitution based polynomial multiplication as the matrix would first need to be transformed from NTT to regular domain in order for the method to be applicable. Moreover, [AHH+19], as well as [BRv22], consider very large hardware-accelerated multipliers supporting operands of up to > 2048 and 4096 bits, respectively. In comparison, the 64×64-bit multiplier of OTBN is less powerful. For these reasons, we do not consider the Kronecker approach in more detail. However, it has to be noted that in concurrent research, implementation results for the polynomial multiplication for Dilithium on OTBN using the approach from [BRv22] have been presented in [Tur23].

2.6 Modular Reduction & Multiplication Algorithms

With the NTT offering an efficient approach to polynomial multiplication on a high level, it is similarly important to perform the underlying integer operations at a high speed. As it becomes clear from Algorithms 2.5.1 and 2.5.2, the integer operations relevant are modular multiplication, addition, and subtraction in $\mathbb{Z}_q$. More precisely, the type of multiplication required during the transformations is multiplication of a variable with a

constant, while the pointwise multiplication inside the NTT domain is on two variables.

This section introduces several different modular-multiplication algorithms that have been used for high-speed implementations of lattice-based cryptography in the past, for example, in [ADPS16, Sei18, LDK+22, BHK+22, HZZ+22]. Besides the performance, it is crucial for such an algorithm to be implementable in constant-time – meaning its runtime must not depend on any of the secret inputs – in order to avoid leakage of timing side-channel information. Naively performing modular reductions via, e.g., trial division violates this constraint in almost all cases due to timing variability of division instructions on many platforms.

2.6.1 Montgomery Multiplication

The algorithm due to Montgomery [Mon85] is one of the most common approaches for multiplication with modular reduction; for example, it is used in the reference implementations of Dilithium [Dil23], as well as Kyber [Kyb23].

Montgomery's algorithm operates on a special representation of the elements, called m-residues, which we define in Definition 2.6.1. We say that computing the m-residue of an element transforms it into Montgomery domain.

Let in the following modulus $m > 0$, product T, and radix R be integers, where $R > m$, $\gcd(m, R) = 1$, and $T \in [0, mR)$ [MKvOV96].

Definition 2.6.1 (m-Residue [Mon85])
The m-residue of $x \in \mathbb{Z}_m$ is defined as

$$x' = x \cdot R \bmod m.$$

Before turning to the multiplication, let us consider the case of reduction first: The Montgomery reduction computes $\mathsf{MRed}(T) = TR^{-1} \bmod m$ efficiently for appropriate choices of R. A representation of the technique can be found in Algorithm 2.6.1. The radix R is commonly chosen as a power of two matching a multiple of the machine word-size of the target architecture such that the modulo and division operations from Lines 1 and 2 in Algorithm 2.6.1 can be computed as bit shifts or dropping of bits.

Considering modular multiplication, suppose we were to multiply two integers a and b. In a first step, they are transformed into Montgomery domain with their respective m-residues being a', b'. Their product then amounts to $a'b' = abR^2$. To efficiently reduce this result modulo m, Algorithm 2.6.1 can be applied, giving $\mathsf{MRed}(abR^2) = abR \bmod m$, which is the result of the multiplication of a and b reduced modulo m in Montgomery domain. A result in "regular domain" can be computed by applying Algorithm 2.6.1 a second time to the output in Montgomery domain. Note that this is usually only done after a series of modular operations inside Montgomery domain is completed, as the transformations to and from the domain are rather costly. Due to the stability of the Montgomery domain representation over addition and subtraction, staying in the domain does not pose an issue in most applications.

Algorithm 2.6.2 shows a version of the Montgomery reduction which takes a signed integer as its input and also returns a signed value. Note that the output range for the

Algorithm 2.6.1: Montgomery reduction [Mon85] MRed.

Input : Product $T \in [0, mR)$, modulus m, radix R, $m' = -m^{-1} \bmod R$
Output : $\mathsf{MRed}(T) = TR^{-1} \bmod m$, $T < m$

1 $U \leftarrow Tm' \bmod R$
2 $t \leftarrow \left\lfloor \frac{T+Um}{R} \right\rfloor$
3 **if** $t \geq m$ **then**
4 | **return** $t - m$
5 **else**
6 | **return** t

Algorithm 2.6.2: Signed Montgomery reduction [Sei18] $\mathsf{MRed}^{\pm}$.

Input : Product $T \in [-\frac{R}{2}m, \frac{R}{2}m)$, with $T = T_1R + T_0$, $T_0 \in [0, R)$, modulus $m \in (0, \frac{R}{2})$, radix R, $m' = m^{-1} \bmod {}^{\pm}R$
Output : $\mathsf{MRed}^{\pm}(T) = TR^{-1} \bmod {}^{\pm}m, T \in (-m, m)$

1 $U \leftarrow T_0m' \bmod {}^{\pm}R$
2 $t \leftarrow \left\lfloor \frac{Um}{R} \right\rfloor$
3 **return** $T_1 - t$

algorithm as given in [Sei18] is larger because the algorithm omits the final conditional correction, which can be beneficial depending on the use case.

2.6.2 Barrett Multiplication

Just as before, we start by introducing the Barrett reduction algorithm first before coming to the multiplication variant. In fact, the notion of Barrett multiplication in an analogous sense to Montgomery multiplication is rather recent [BHK+22], while the reduction algorithm is widely known [Bar87]. We give a description of the reduction algorithm in Algorithm 2.6.3. Note that without conditional subtractions, the algorithm generally yields results in $[0, 3m)$ [Bar87], while in some instances, where R is picked larger, the approximation can be improved and thus the range for the results can be narrowed down. Compared to Montgomery's algorithm, one advantage of the Barrett reduction is that it does not require the inputs or outputs to be transformed to or from a special representation.

A signed variant of the Barrett reduction was introduced by Seiler in [Sei18], where the accuracy of the approximation is improved over the initial approach as in Algorithm 2.6.3. We give the corresponding algorithm in Algorithm 2.6.4.

For performing modular multiplication, T in Algorithms 2.6.3 and 2.6.4 could be computed using a multiplication followed by the application of the Barrett reduction.

[BHK+22] has shown that instead of flooring, a variety of integer approximations can be used when computing v and also, e.g., rounding can be considered in Line 1 of Algorithm 2.6.3. Finally, Algorithm 2.6.5 shows the method of Barrett multiplication [BHK+22].

Algorithm 2.6.3: Barrett reduction [Bar87] BRed.

Input : Product $T \in [0, R]$, modulus m, radix $R = 2^k > m, k \in \mathbb{N}$, constant $v = \lfloor \frac{R}{m} \rfloor$
Output : $\mathsf{BRed}(T) = T \bmod m$, $\mathsf{BRed}(T) \in [0, m)$

$t \leftarrow \left\lfloor \frac{Tv}{R} \right\rfloor m$
if $t \geq m$ **then** $t \leftarrow t - m$
if $t \geq m$ **then** $t \leftarrow t - m$
return t

Algorithm 2.6.4: Signed barrett reduction [Sei18].

Input : Product $T \in [-\frac{R}{2}, \frac{R}{2}]$, modulus $m \in [0, \frac{R}{2})$, radix R, constant $v = \left\lfloor \frac{2^{\lfloor \log m \rfloor - 1} R}{m} \right\rceil$
Output : $r \equiv T \bmod m$, $r \in [0, m]$

$t \leftarrow \left\lfloor \frac{Tv}{2^{\lfloor \log m \rfloor - 1} R} \right\rfloor m \bmod R$
return $T - t$

Algorithm 2.6.5: Signed Barrett multiplication [BHK+22].

Input : $a \in \mathbb{Z}$ with $\mid a \mid < R$, $b \in \mathbb{Z}$, constant $v = [\![\frac{bR}{m}]\!]$, modulus $m \in [0, R)$, radix $R = 2^k, k \in \mathbb{N}$
Output : $r \equiv ab \bmod m$, $r \in (-m, m)$

$T \leftarrow ab$
$t \leftarrow \left\lfloor \frac{Tv}{R} \right\rceil m$
return $T - t$

2.6.3 Plantard Multiplication

In 2021, Plantard published a new approach to modular multiplication, which saves one multiplication operation when considering multiplication with a constant, compared to its Montgomery and Barrett counterparts [Pla21]. A representation is given in Algorithm 2.6.6. Similar to Montgomery's technique, it is required to transform the inputs to the multiplication into another domain, the Plantard domain. The outputs of the reduction will also be in Plantard domain. Again, just as with Montgomery, applying the reduction once more transforms the output to regular domain. Further, ℓ is also commonly picked as a multiple of the machine word-size for efficient operation.

Algorithm 2.6.6 still contains a total of three multiplications, but it can be noted that, w.l.o.g., considering b as the constant, the product of $bm' \bmod 2^{2\ell}$ can be computed and stored ahead of time. In theory, this would also be possible for Montgomery and Barrett, but in practice, it is of no use, as the product of a and b is later on input to a subtraction or addition in which it is not supposed to include the factor m' or v, respectively. One downside of this precomputation is that usually more memory will be required to store the constant in $\mathbb{Z}_{2\ell}$.

A signed version with a larger input range was presented in [HZZ+22]. Note that the enlarged input range also straightforwardly transfers to the unsigned version as it is induced via a restriction on m. Algorithm 2.6.7 shows the modified version.

Algorithm 2.6.6: Plantard multiplication [Pla21].

Input : $a, b \in [0, m]$,
$m' = m^{-1} \bmod 2^{2\ell}$,
$\ell = 2^k, k \in \mathbb{N}$
Output : $T = ab(-2^{-2\ell}) \bmod m$,
$T \in [0, m)$

1 $t \leftarrow abm' \bmod 2^{2\ell}$
2 $t \leftarrow \left\lfloor \frac{t}{2^\ell} \right\rfloor + 1$
3 $t \leftarrow \left\lfloor \frac{tm}{2^\ell} \right\rfloor$
4 **if** $t = m$ **then return** 0
5 **return** t

Algorithm 2.6.7: Improved Plantard multiplication [HZZ+22].

Input : $a, b \in [-q2^\alpha, q2^\alpha]$, modulus
$m < 2^{\ell-\alpha-1}$,
$m' = m^{-1} \bmod {}^{\pm}2^{2\ell}$, ,
$\ell = 2^k, k \in \mathbb{N}$
Output : $T = ab(-2^{-2\ell}) \bmod {}^{\pm}m$,
$T \in [-\frac{m-1}{2}, \frac{m-1}{2}]$ for odd m

1 $t \leftarrow abm' \bmod 2^{2\ell}$
2 $t \leftarrow \left\lfloor \frac{t}{2^\ell} \right\rfloor + 2^\alpha$
3 $t \leftarrow \left\lfloor \frac{tm}{2^\ell} \right\rfloor$
4 **return** t

2.6.4 Specialized Reductions

The Dilithium reference implementation [Dil23] contains another approach for modular reduction, specifically designed for the use with Dilithium's parameters. We give a formal description of the algorithm in Algorithm 2.6.8.

Algorithm 2.6.8: Specialized Reduction for Dilithium [Dil23] reduce32.

Input : $T \leq 2^{31} - 2^{22} - 1$, modulus $m = 8380417$, constant $c = 2^{22}$
Output : $T \bmod m$, $T \in [-6283009, 6283007]$

1 $t \leftarrow \left\lfloor \frac{T+c}{2^{23}} \right\rfloor$
2 **return** $T - tm$

2.7 Root of Trust

Casper and Papa define a *Root of Trust (RoT)* as an element in a system that, among other features, allows for verification of a system, monitoring software, as well as data integrity and confidentiality [CP11]. Such an element can be implemented in software or hardware, depending on the required level of protection and the threat model [CP11]. Hardware RoTs can be used to achieve a higher degree of trust.

A hardware RoT aids in ensuring the security of a system by first verifying its own integrity (e.g., during boot) and then extending the trust to other system components, usually by deploying cryptographic authentication measures [CP11]. Among other features, an RoT commonly offers symmetric and asymmetric cryptographic primitives, a secure Random Number Generator (RNG), secure key storage, and encrypted memory [CP11].

RoTs can be found in a wide variety of devices, ranging from smartphones [Xin18] over security tokens [Ehr19] to data center hardware [Tit17].

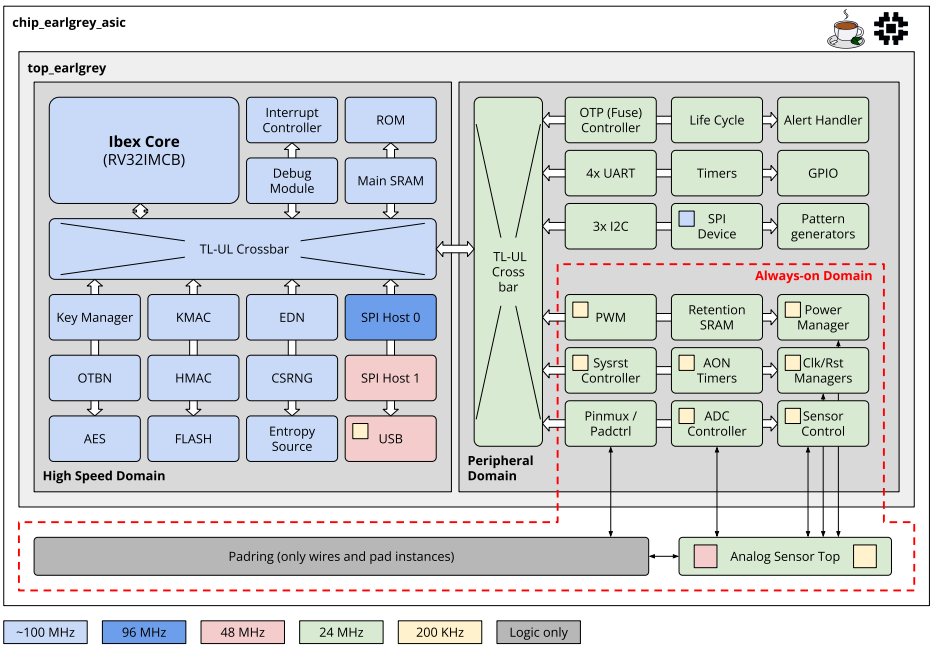


Figure 2.3: Block diagram of the OpenTitan Earl Grey microcontroller [Tea23a, Section 6.1].

2.8 OpenTitan

OpenTitan is a project with the goal to build an open-source-silicon RoT [Tea23a] following a transparency-first approach, publishing the entire hardware design, firmware, and cryptographic software related to the project.

The not-for-profit company lowRISC CIC stewards the project, using collaborative engineering to develop and maintain open-source silicon designs [Tea23a]. Engineers by ETH Zürich, G+D Mobile Security, Google, lowRISC, Nuvoton Technology, and Western Digital have contributed to the project [Tea23a].

From a technical perspective, the OpenTitan team is developing several comportable Intellectual Property (IP) blocks, as well as a processor named OTBN. Together with an instantiation of a RISC-V Ibex core designed by the lowRISC team and a number of peripheral components, these building blocks make up the Earl Grey microcontroller [Tea23a, Section 6.1], depicted in Figure 2.3. The majority of the aforementioned IP blocks are related to security: For example, there are blocks for AES, Keccak Message Authentication Code (KMAC), and Hash-based Message Authentication Code (HMAC), as well as a Cryptographically Secure Random Number Generator (CSRNG) compliant to Bundesamt für Sicherheit in der Informationstechnik (BSI) and FIPS standards.

There are multiple ways to interact with the hardware designed by the OpenTitan Team. For example, OpenTitan can be run on a Field Programmable Gate Array (FPGA) like the ChipWhisperer CW310 or CW340 board. This approach will remain the closest

to real hardware until the Application-Specific Integrated Circuit (ASIC) tapeout samples for Earl Grey are available [Tea23b]. Without any specialized hardware, the project can also be simulated using Verilator. Verilator is a cycle-accurate simulation tool that transforms Verilog code into a C++ program that can be compiled and run [Tea23a, Section 3.2]. Specifically for one of the OpenTitan SoC's components, the OTBN core, there also exists a cycle-accurate Python simulator on which we give more details in Section 2.8.1.

2.8.1 OpenTitan Big Number Core

The OTBN core [Tea23a, Section 8.2] is a programmable coprocessor designed to accelerate computations for classical, asymmetric cryptography schemes such as RSA [RSA78] and Elliptic Curve Cryptography (ECC) [Mil86, Kob87]. In addition, it offers several features to enhance the security throughout and after the execution of cryptographic code.

In the following, we will give a brief overview of the coprocessor and details on aspects relevant to the implementation of PQC on OTBN.

OTBN Security

The design of OTBN prioritizes security over performance. This section will introduce some of the unique security features OTBN deploys.

The processor does not feature any form of speculative execution. Conditional jump or branch instructions always cause a stall rather than using a branch predictor to execute a possible next instruction ahead of time. This approach is deployed to mitigate Spectre-style attacks [Tea23a, Section 8.2.2][KHF+19]. In the same line, loads and stores to the data memory are not cached, and thus, cache timing attacks are mitigated [Tea23a, Section 8.2.2]. Additionally, memory scrambling and register blanking are deployed to further counteract side-channel leakage [Tea23a, Section 7.2.2]. Concerning fault injection attacks, a checksum register for instruction and data memory accesses, as well as an instruction counter to detect skipped instructions from the Ibex core, are part of the system [Tea23a, Section 8.2.2].

OTBN Registers

On the OTBN core, a total number of 32 32-bit wide General Purpose Registers (GPRs), named `x0` through `x31`, as well as 32 256-bit wide Wide Data Registers (WDRs), named `w0` through `w31`, are available. For the GPRs, `x0` is hard-wired to zero and will ignore any writes, while `x1` is used to access the call stack [Tea23a, Section 8.2]. By convention, we call `w31` `bn0` if it contains all zeros.

Next to the two aforementioned register types, there are two additional varieties: The Control and Status Registers (CSRs) and Wide Special Purpose Registers (WSRs). These registers give access to randomness sources, arithmetic flags, to the key material provided by a dedicated key manager, and also include a "modulus register" [Tea23a, Section 8.2]. The modulus register `MOD` can be used to define an integer of up to 256 bits in length that is used by certain instructions we will introduce later.

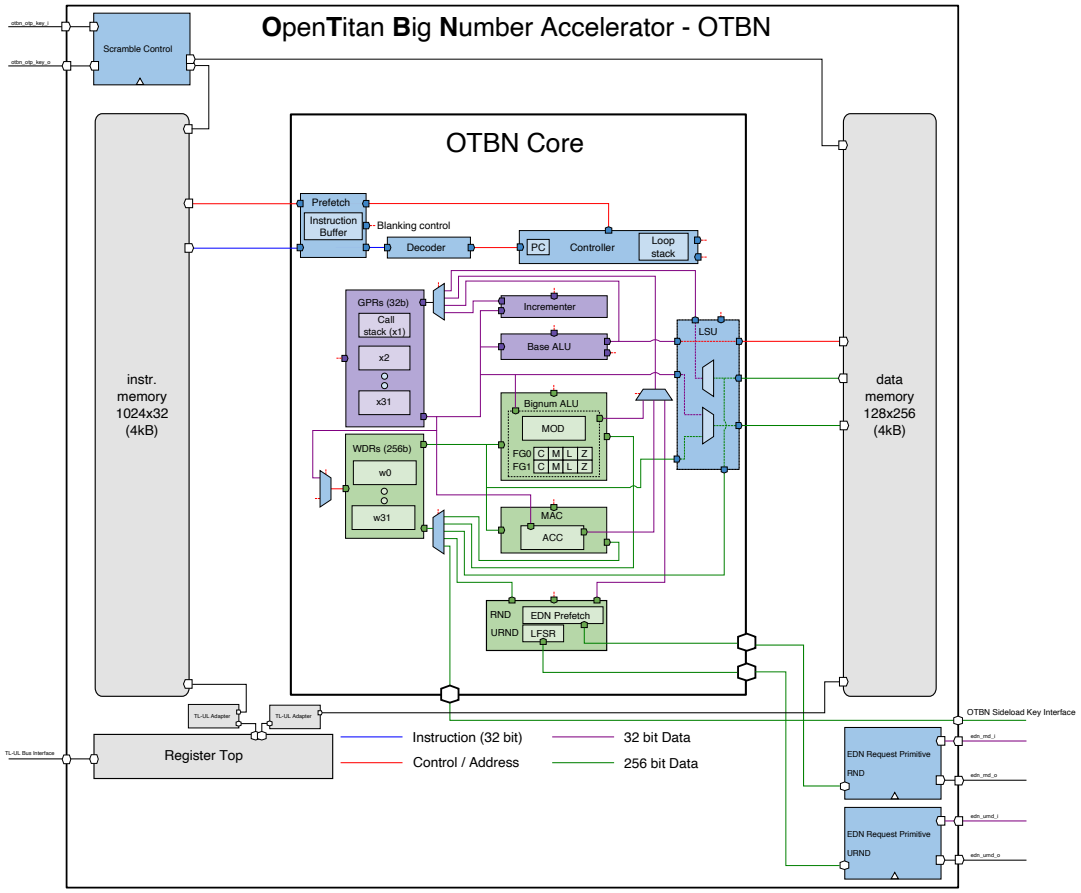


Figure 2.4: OTBN block diagram [Tea23a, Section 8.2.1].

There are two separate flag groups available on OTBN, consisting of a carry, Most Significant Bit (MSB), Least Significant Bit (LSB), and zero flag [Tea23a, Section 8.2].

OTBN Memory

As shown in Figure 2.4, OTBN offers a total of 4 KiB of instruction memory and 4 KiB of data memory. The 4 KiB of data memory are made up of 3 KiB, which are accessible from the Ibex Core, while 1 KiB, the so-called scratchpad memory, is not accessible – even when OTBN enters its idle state [Tea23a, Section 8.2.1]. The data memory allows either 4-byte aligned reads and writes from the GPRs or 32-byte aligned access from the WDRs. The instruction memory is neither writable nor readable from user code.

OTBN Instruction Set

OTBN deploys a custom instruction set, with its instructions separated into two groups [Tea23a, Section 8.2.4]:

- Base instruction subset: The base instructions operate on 32-bit GPRs. The primary use of these instructions is to manage the control flow. While the RV32I RISC-V instruction set inspired the instructions, they are not compatible.
- Big number instruction subset: Instructions dedicated to data processing on 256-bit WDRs.

Most instructions only take one single cycle to complete, with exceptions being jump and branch instructions like `jal`, `jalr`, `bne`, and `beq`, as well as most memory operations like `lw`, `bn.sid`, and `bn.lid` taking two cycles.

In the following, we will go over some notable instructions that are uncommon on other platforms or of potential interest for implementations of PQC. For a complete description of the instruction set, we refer the reader to [Tea23a, Section 8.2.4].

- `bn.addm <wrd>, <wrs1>, <wrs2>`: The "Pseudo-Add Modulo" instruction adds `<wrs1>` and `<wrs2>`. If the result is greater or equal to `MOD`, `MOD` is subtracted from the result. This amounts to computing addition modulo `MOD` for inputs in [0, `MOD`). Note that this instruction will not give correct results when used on, e.g., 32-bit, signed integers in two's complement representation. There are two reasons for this behavior: First, the addition is performed on 256 bits and the potential overflow will not be cut off as with 32-bit arithmetic. Second, when checking whether the result is greater or equal to `MOD`, it is interpreted as an unsigned number.
- `bn.subm <wrd>, <wrs1>, <wrs2>` The "Pseudo-Subtract Modulo" instruction subtracts `<wrs2>` from `<wrs1>`. If the result is less than 0, `MOD` is added to the result. This amounts to computing subtraction modulo `MOD` for inputs in [0, `MOD`). Similar to `bn.addm`, this instruction is unsuitable for computing arithmetic on, e.g., 32-bit, signed integers in two's complement representation.

- `bn.mulqacc.wo[<zero_acc>] <wrd>, <wrs1>.<wrs1_qwsel>, <wrs2>.<wrs2_qwsel>, <acc_shift_imm>`: The "Quarter-word Multiply and Accumulate with full-word writeback" instruction multiplies two 64-bit quarter-words from `<wrs1>` at index `<wrs1_qwsel>` and from `<wrs2>` at index `<wrs2_qwsel>`. If `<zero_acc>` is set to `.z`, the accumulator is set to zero before adding the result. The parameter `<acc_shift_imm>` allows to shift the result to the left by a multiple of 64 between 0 and 192 before writing the result to `<wrd>`.
- `{bn.and, bn.or, bn.xor} <wrd>, <wrs1>, <wrs2>[ <shift_type> <shift_bits>]`: The basic logical instructions offer the ability to shift one of the two inputs to the right or left by a multiple of 8 between 0 and 248.
- `bn.rshi <wrd>, <wrs1>, <wrs2> >> <imm>`: The "Concatenate and right shift immediate" instruction concatenates the registers `<wrs1>` and `<wrs2>`, where `<wrs1>` makes up the upper part, shifts the result by `<imm>` to the right, and truncates the result to 256 bits. Note that this instruction can also serve as a left shift by selecting `<wrs2>` to be all zeros and choosing `<imm>` as $256 - m$ for m being the desired amount of bits to shift to the left.
- `bn.cmp <wrs1>, <wrs2>[ <shift_type> <shift_bits>][, FG <flag_group>]`: The compare instruction subtracts `<wrs2>` from `<wrs1>`, sets the flags accordingly, and discards the result. Optionally, one of the two inputs can be shifted to the right or left by a multiple of 8 in the range between 0 and 248.
- `bn.sel <wrd>, <wrs1>, <wrs2>, [FG<flag_group>.]<flag>`: The instruction writes `<wrs1>` to `<wrd>` if the flag `<flag>` is set, `<wrs2>` otherwise.
- `loopi <iterations>, <bodysize>`: The "loop immediate" instruction exhibits access to the hardware loop feature of OTBN. Hardware loops come at no extra cost compared to unrolling except for a single cycle that is spent on the `loopi` instruction. `<iterations>` defines how many iterations of the loop to perform, `<bodysize>` has to be set to the number of instructions that are part of the loop. There also exists a variant of this instruction where `<iterations>` can be defined by a GPR called `loop`. The last instruction of the loop always needs to be executed `<iteration>` times, else the loop operation will not be removed from the "loop stack", and thus, it will be polluted. This especially means that it is impossible to early-exit a hardware loop. Jumps within the loop body are allowed as long as the last instruction is executed. Two nested loops must not end on the same instruction.

It is important to note that although OTBN offers registers the same size as AVX2, there are no instructions that consider the wide registers as vectors of individual elements of a certain size. These instructions are often referred to as SIMD or just vector instructions. We will later see that this significantly impacts the performance with respect to the implementation of lattice-based cryptography, as the vectorization typically enables much faster implementations [LDK+22, BHK+22].

Also, no compiler is available for the OTBN platform, meaning that the development requires implementation in the assembly language.

Python Simulator

The previously mentioned Python simulator is mainly a tool to aid with the development of software for the OTBN core. It offers cycle-accurate performance results and extensively models the hardware architecture of the OTBN core while remaining modifiable and extensible. The simulator can take elf files that have been assembled for OTBN as its input, execute the code, and dump the register contents. Additionally, it is possible to retrieve detailed statistical data on the execution, such as counts of cycles and stalls, as well as an instruction histogram and numbers of function calls.

We make extensive use of the simulator in our work, using it to obtain benchmarks and test our implementations.

2.8.2 Keccak Message Authentication Code Core

According to the documentation [Tea23a, Section 9.13], the KMAC core can be used to compute Keccak-based Message Authentication Codes (MACs), as well as unauthenticated SHA-3, including its XOF operation mode called SHAKE. The latter is especially of interest for the implementation of PQC schemes such as Kyber or Dilithium. The hardware design offers a compile-time choice between a version with first-order masking enabled and a version without masking. The technique applied is called Domain-oriented Masking (DOM) and increases the area required by the logic design by a factor greater than two [Tea23a, Section 9.13.1].

Computing the permutation Keccak-f for a state of $b = 1600$ bit takes four cycles per round for a number of $(12 + 2 \cdot \log_2(\frac{b}{25})) = 24$ rounds, resulting in 96 cycles in total [Tea23a, Section 9.13].

2.8.3 OpenTitan Project Structure

All code related to the OpenTitan Project is published on GitHub [Ope23]. The repository contains the hardware descriptions, software, as well as auxiliary utilities related to the project. The software can be built and tested by interacting with the build system that is set up using Google's Bazel[1].

Most relevant to this thesis are the directories `hw` and `sw`. The former contains the OTBN Python simulator under `opentitan/hw/ip/otbn/dv/otbnsim`, which we will make use of and modify throughout this work. The `sim/insn.py` is of particular interest, as it contains the instruction semantics the simulator uses. The three files `enc-schemes.yml`, `insns.yml`, and `bignum-insns.yml` are located in a subdirectory of `hw`, `opentitan/hw/ip/otbn/data/`; these files are relevant for extending the instruction set and the definition of instructions. The latter directory, `sw`, holds the most

[1] `https://bazel.build/`

relevant part regarding cryptographic implementations under `opentitan/sw/otbn/crypto`. It contains a cryptographic library for OTBN, as well as exemplarily code snippets.

3 Implementing Dilithium on OTBN

This chapter presents our implementation of Dilithium on the OTBN core. We will first outline general constraints on the implementation before coming to concrete details and optimizations. At the end of this chapter, we give the evaluation results of our implementation, describing our testing setup, performance results, and discussing our implementation's security.

3.1 Goals of the Implementation

The goal of obtaining an implementation of Dilithium on OTBN is twofold: First, we aim to find out whether it is at all possible to implement Dilithium on OTBN in its current state or if there exist hurdles making an implementation impossible. Second, if possible, we want to establish a "baseline" implementation that can be compared against when considering modifications to the instruction set as in Chapter 4.

Therefore, we explore and implement several optimization strategies to make following comparisons fair and meaningful. As the main target for these optimizations, we consider the performance in terms of cycle counts. When optimizing the memory usage, various trade-offs with the cycle count exist [BRS22, GKS20]. However, none of them was majorly influenced by the underlying instruction set, so we will not consider this path for our work. In terms of code size, an implementation that takes fewer cycles will intuitively – to some degree – also use fewer instructions, as most of the instructions take a single cycle to execute, disregarding techniques such as unrolling and inlining.

In our effort, we focus on the deterministic variant of Dilithium2 to avoid additional implementation overhead and to keep the requirements for the "hardware" as low as possible. We expect this decision to have no impact on the significance of our results as all three versions of Dilithium are highly similar and only minorly differ from an implementation perspective.

As we only target the OTBN core with our implementation, we refrain from targeting the (simulated) hardware design and exclusively operate on the Python-based simulator.

3.2 Modifications to the Architecture

This section describes two modifications we made to the architecture for our base implementation of Dilithium. Note that these modifications do not yet include extensions to the instruction set.

3.2.1 Data & Instruction Memory

As described in Section 2.8.1, the OTBN core has an instruction and data memory, both of which have a size of 4 KiB. In recent work on Dilithium on memory-constrained devices, Bos, Renes, and Sprenkels [BRS22] present an implementation, which for the smallest parameter set, Dilithium2, and even excluding the arguments, still requires 4.9 KiB, 5.0 KiB, 2.7 KiB for key generation, signing, and verification, respectively. This result already indicates that fitting an implementation of Dilithium into the 4 KiB will certainly be a challenging task. As we do not consider a low memory footprint as a goal of our implementation, we will not try to undercut [BRS22]. Instead, we will make use of the fact that we use the OTBN Python simulator for our experiments and, thus, are not bound to the restrictions incurred by the hardware. The file `opentitan/hw/ip/otbn/data/otbn.hjson` defines the memory setup for OTBN that is also used in the definition of the memory in the Python simulator. Thus, modifying this file is enough to gain access to a larger data memory from the simulator. We increase the size from 4 KiB to 64 KiB. The requirement for a significant amount of memory in an unoptimized implementation becomes clear when reconsidering Algorithm 2.4.9: Each polynomial in NTT-domain takes up 32 bits per coefficient, meaning one polynomial for $n = 256$ takes 1 KiB. With one matrix of size $k \times l$, three vectors of polynomials of size l, five vectors of polynomials of size k and one single polynomial for the challenge, this sums up to $(16 + 12 + 20)\text{KiB} = 48\,\text{KiB}$ in the case for Dilithium2. We proceed similarly for the instruction memory, although we only increase its size by a smaller amount, from 4 KiB to 8 KiB.

3.2.2 Interface to KMAC

As Dilithium uses SHAKE on various occasions and Earl Grey contains an IP block for KMAC, which is also capable of computing SHAKE, the idea of interfacing between the two naturally arises. While we have been in the process of implementing such an interface based on instructions corresponding to the definitions of the device interface functions from the documentation [Tea23a, Section 9.13.6], Philpoom from Google's OpenTitan Team independently published such an interface in a development branch of the OpenTitan repository[1]. Her approach is based on interaction with the CSRs and WSRs: The CSRs are used to configure the desired KMAC operation. Data absorption into the SHAKE state is modeled through writes to one of the WSRs while squeezing data corresponds to a read. The timing constraints of the KMAC core are respected by stalling the execution of the respective instruction during the operation. This way, computing one round of Keccak takes four cycles.

We already use this interface in our base implementation for multiple reasons: On the one hand, we see this as an indication that the OpenTitan team is indeed interested in having this type of interface available, and thus, it is a realistic addition to OTBN. On the other hand, this work focuses on implementing Dilithium on OTBN, not SHAKE. While

[1] `https://github.com/jadephilipoom/opentitan/commit/e86be3446204f439c41c142b077a4ca8b449b1c9`

SHAKE is part of Dilithium, our research interest lies in extending the instruction set for Dilithium (and lattice-based cryptography in general) instead of optimizing SHAKE on a variant of a RISC-V platform.

3.3 Development for OTBN

In the following, we describe how we approached the development of Dilithium on OTBN from a workflow-perspective.

The first step in our development workflow was to start implementing a small functionality, e.g., a function `ntt_fwd` for the NTT, that will reside inside an assembly file, e.g., `ntt.s`. This file on its own cannot be meaningfully executed as it is missing an entry point, data inputs, and more. At the same time, it is usually of great interest to implement some test to verify the correct operation of the function implemented – especially concerning cryptographic software. Thus, in a next step, we write a second assembly file `ntt_test.s`. This file defines a `main` function that serves as the starting point to the OTBN program, loads the data that will be used during the test, executes our function `ntt_fwd`, and performs an `ecall` instruction to exit the OTBN and return to the Ibex core. To integrate both files with the bazel build system, we need to define how they should be handled inside so-called `BUILD` files. We define an instance of the `otbn_library` build rule for `ntt.s` as the source and give it the name `ntt_lib`. `otbn_library` is used to construct object files from assembly sources, internally invoking `otbn-as` for pre-processing the OTBN-specific instructions before passing the result on to the regular `rv32-as`. Note that an `otbn_library` is not executable on its own and needs to be processed into an elf file, e.g., via the `otbn_binary` rule. The `otbn_binary` rule links together library object files with the object file containing a `main` function using a combination of `otbn-ld` and `rv32-ld`. For the test, we instantiate a rule of the type `otbn_sim_test` and define `ntt_test.s` as the source file, `ntt_lib` as a dependency, and also provide a `ntt_test.exp` file, which is used for comparing the output of `ntt_fwd` to a predefined reference.

In the following, we will explain `otbn_sim_test` in more detail, before describing a slight variation we implemented. The rule takes the following inputs:

`name` This defines the name of the test case, which can be used to address it from the command line.

`srcs` Usually, `srcs` is used to define the assembly file containing a wrapper function for the function(s) to be tested, called `main`. This file is usually used to prepare the actual test, load input data, and to place the output data into a desired register or memory address.

`exp` This file defines the expected contents of the registers after the `main` test function has finished. This way, a test's output can be validated for correctness.

`deps` This input defines dependencies for the test; most of the time, at least one will be defined. These dependencies are instances of the `otbn_library` rule.

Inside the implementation of the `otbn_sim_test` rule, the arguments from `srcs` and `deps` are passed to an instance of the `otbn_binary` rule to create an elf file. In the following, the elf file created from the sources and dependencies alongside the path to the Python-based OTBN simulator is passed to a Python wrapper script. This Python script then executes a shell command, in which the simulator is called on the elf file and evaluates the simulator's output by reading from stdout. The output's register state is then compared to the values defined in the `exp` file, and the execution is checked for errors. A simplified visualization of the process can be obtained from Figure 3.1.

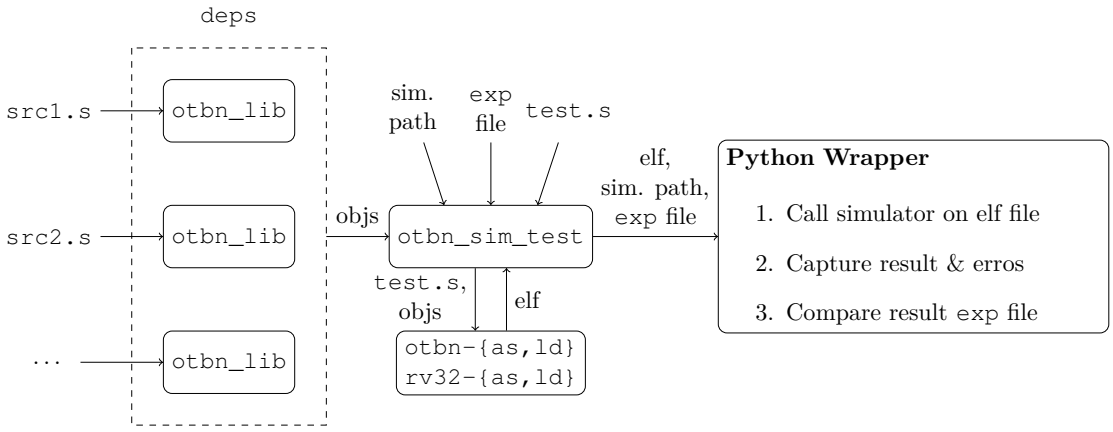


Figure 3.1: Simplified flow of operation for `otbn_sim_test`.

We chose a "bottom-up" approach for our implementation, meaning we started the implementation with small subroutines first. In order to test all these subroutines without the need to write tests for each individual one, we use a Python implementation of Dilithium [Pop23] in which we can replace parts of the code with calls to the OTBN simulator successively. For this, we implement a new build rule `otbn_sim_py_test`, based on `otbn_sim_test`. Our new rule takes multiple `otbn_binary` or `otbn_sim_test` instantiations as dependencies, allowing it to access multiple elf files. It then passes the elf file paths to a user-definable wrapper Python script that will handle the logic of the test. It can be understood as an abstraction of `otbn_sim_test` to some degree. Figure 3.2 gives an overview of the workflow, where it is applied for replacing the NTT, INTT, and pointwise multiplication in a Python implementation of Dilithium by calls to the simulator. In a first step, it is required to implement an interface between OTBN and the Python implementation, such that polynomials present in OTBN's data memory can be parsed into the polynomial data structure of the Python implementation and vice versa. After that, the calls to the NTT, INTT, and pointwise multiplication functions of the Python implementation can be replaced by calls to the simulator. Finally, the existing checking mechanism of the Python implementation can be used to validate the correctness. As the implementation progressed, we merged more and more of the individual subroutines until we arrived at a single `otbn_sim_test` for key generation, signing, and verification, respectively.

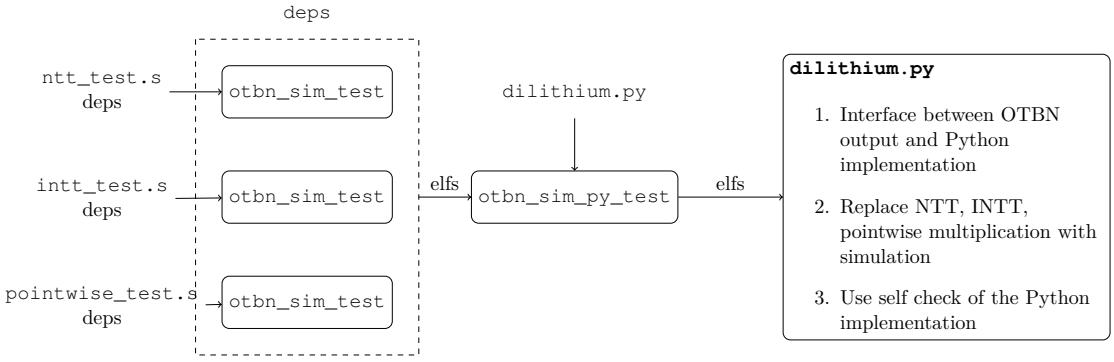


Figure 3.2: Exemplarily application of `otbn_sim_py_test` for replacing polynomial multiplication in a Python implementation of Dilithium.

In order to make the simulator even more useful and to enable some functionalities of `otbn_sim_py_test`, we implement a number of extensions and modifications:

- The file containing the expected result can now also contain data-memory addresses and the expected value to be stored at the respective address. This way, it is more convenient to compare the output for very large results.
- While the wrapper script for `otbn_sim_test` runs the simulator in a shell using Python's `subprocess` module, we decided to call the simulator from within Python such that we can more easily get access to its internals, for example, for manipulating the data memory after loading the elf file, as detailed in the following.
- We implement a simple interface to the simulator's data memory such that we can write to and read from it from the Python script. This way, we can dynamically inject data before starting the execution of the simulator and evaluate the output state from the same Python script. In order for this approach to work, the memory requirements need to be known ahead of time and allocated appropriately. Further, for injecting and reading, the offsets at which the desired data should be/is stored need to be manually entered.
- We significantly extend the simulator's statistics module so that it can also give per-function cycle counts as well as a per-function instruction histogram. Moreover, we separate between instructions executed and the number of stalls caused by the respective instructions. These extensions have been of high value in the profiling of our implementation.

3.4 Description of the Implementation

In this section, we will describe certain parts of our implementation that are either critical with respect to the performance or that exhibit interesting challenges and opportunities incurred by our target platform, OTBN.

3.4.1 To be signed, or not to be signed

One fundamental decision for the implementation of polynomial arithmetic is whether to store and process the coefficients as signed or unsigned integers. On the Arm Cortex-M4 platform, [GKS20] introduced the usage of signed integer coefficients. The reason behind this decision is to avoid the result wrapping around zero when $b > a$ for $a - b$. In the unsigned case, for example, implemented by [GKOS18], it is required to add a multiple of the modulus to the result until it wraps back around into the positive domain. This is not a concern when interpreting the result as a signed integer. According to [GKS20], the signed approach not only yields benefits in the NTT and INTT but also in the functions sampling $\boldsymbol{s}_1$, $\boldsymbol{s}_2$, $\boldsymbol{y}$, as well as the packing. The C reference implementation also makes use of a signed representation for the coefficients [Dil23].

On OTBN, the situation is different: The `bn.subm` instruction does exactly mitigate the aforementioned issue of wrapping around zero if the value being subtracted is less or equal to q and the `MOD` register is set to q. We explain why the former condition is always fulfilled in Section 3.4.3.

Another reason why signed arithmetic is unfavorable on OTBN is that the instruction set is not geared towards processing 32-bit signed integers but rather 256-bit ones. This brings some disadvantages; for example, widening multiplications require the sign-extension of the inputs up to a length that is as long as the number of correct bits of the low part of the result. Also, for additions and subtractions of signed numbers, the bits overflowing the 32-bit result will not be cut off as usual on a 32-bit architecture but remain in the WDR and thus need to be masked out for a correct result w.r.t. 32-bit signed arithmetic. Sign-extending all coefficients to 256 bits would be a remedy for this, but as there is no dedicated instruction for sign extension, this is a task taking multiple cycles, manually extracting the sign bit, and modifying the value. Finally, as detailed in Section 2.8.1, the especially useful `bn.addm` and `bn.subm` instructions treat their inputs and outputs as unsigned 256-bit integers, thus having no use for the application to 32-bit signed integer arithmetic.

In Sections 3.4.4 and 3.4.5, we will outline how we can obtain unsigned representations of the coefficients with no or very little additional cost when compared to signed operation, as, for example, in the reference implementation [Dil23].

3.4.2 Modular Arithmetic

To lay the groundwork for polynomial arithmetic, we will first focus on giving details on our implementation of modular arithmetic.

Modular Addition & Subtraction

Usually, additions and subtractions are not implemented including a modular reduction in each step, as they only cause very minor growth to the coefficients. For example, in DILITHIUM, coefficients of size up to $512q$ can fit in a 32-bit integer. Instead, the coefficients are allowed to grow and are only reduced if the following operation could

potentially cause an overflow. This approach is especially common in implementations of the NTT and INTT and is also called "lazy reduction".

On OTBN, we have the possibility to almost entirely eliminate the growth of the coefficients beyond a value of q by making use of the `bn.addm` and `bn.subm` instructions. If the inputs to an addition or subtraction are unsigned integers in a range of $[0, q]$, the result will also be in the exact same range. Thus, if we can have a modular multiplication algorithm that produces outputs in $[0, q]$ and handle all additions and subtractions with the aforementioned instructions, the size of a coefficient would never exceed q.

Modular Multiplication

The choice of the modular multiplication algorithm is one of the most crucial decisions for the implementation of Dilithium as it majorly contributes to the runtime. In the following, we will only consider (versions of) algorithms that output an unsigned result in the range between zero and at most q in order to profit from the addition and subtraction instructions with implicit modular reduction, as explained in the previous section. We will not consider the Barrett multiplication approach from Algorithm 2.6.5 as there are no rounding shift instructions on OTBN.

Let us first consider the case of multiplications with a constant, like the twiddle factors in the NTT and INTT. Using Montgomery's approach as in Algorithm 2.6.1, at least three multiplications are required: One for computing T, one multiplication with m', and one with m. The reduction modulo R can be performed at no additional cost for $R = 2^{64}$ by selecting the respective quarter word as the input to the subsequent multiplication. The final conditional subtraction by m can be implemented highly efficiently by adding a zero to t using `bn.addm`, which will implicitly subtract m if the result is larger or equal to m. The flooring division by R can be performed efficiently using a right shift but needs to be performed explicitly and cannot be merged with the subsequent `bn.addm`, as the instruction does not offer to apply a shift to one of the inputs. An exemplary implementation can be obtained from Listing 3.1, taking six cycles for one modular multiplication.

The variant of the Barrett reduction presented in [Sei18], as given in Algorithm 2.6.4, also fulfills the constraint on the output range. Just as with Montgomery's algorithm, three multiplications are required: Computing $T = ab$ and subsequently multiplying by v and m. Additionally, one subtraction needs to be computed. The reduction modulo R can be obtained for free by a technique that we are going to introduce for the Plantard multiplication in the following. So, the only operation left is the flooring division. For Dilithium, $\lfloor \log m \rfloor - 1 = \lfloor \log q \rfloor - 1 = 21$, such that the amount we need to shift by will always be uneven, e.g., 85 for $R = 2^{64}$. This prohibits merging it into any of the arithmetic instructions as they only allow for shifts by multiples of 8 or even 64, depending on the instruction. Thus, we need one `bn.rshi` for performing the shift and therefore spend a total of five cycles as reflected in Listing 3.2.

Finally, let us consider the improved Plantard multiplication as given in Algorithm 2.6.7 with $\ell = 32$. For our implementation, we prefer the improved variant from [HZZ+22] over the original one from [Pla21] because of the larger input range, allowing us to omit

```
1 bn.mulqacc.wo.z T, a.0, b.0, 0
2 bn.mulqacc.wo.z U, T.0, mprime.0, 0
3 bn.mulqacc.wo.z t, m.0, U.0, 0
4 bn.add          t, T, t
5 bn.rshi         t, bn0, t >> 64
6 bn.addm         t, bn0, t
```

Listing 3.1: Montgomery multiplication on OTBN.

```
1 bn.mulqacc.wo.z T, a.0, b.0, 0
2 bn.mulqacc.wo.z t, T.0, v.0, 0
3 bn.rshi         t, bn0, t >> 85
4 bn.mulqacc.wo.z t, t.0, m.0, 192
5 bn.sub          t, T, t >> 192
```

Listing 3.2: Multiplication followed by Barrett reduction on OTBN.

the final conditional subtraction. In contrast to the two previously discussed algorithms, Plantard's approach requires one multiplication less for multiplication with a constant as the multiplication with m' in Line 1 of Algorithm 2.6.7 can be merged into the constant ahead of time. So, we only need to multiply the constant with the non-constant input and then multiply by m later on. In addition to that, one addition of the constant 2^α is needed, as well as two flooring divisions by 2^ℓ as well as one reduction modulo $2^{2\ell}$. In the following, we will present a strategy to get one of the flooring divisions as well as the modulo operation at no cost. A visualization of the idea is given in Figure 3.3. In the diagram, the striped pattern refers to unwanted parts of the result, while the hatched pattern is the part of the result we want to extract. The multiplication in Line 1 of Algorithm 2.6.7 gives a product of 96 bits since the coefficient is at most 32 bits large and the constant is $2\ell = 64$ bits in size. In Algorithm 2.6.7, the reduction modulo $2^{2\ell}$ from Line 1 and the right shift from Line 2 amount to extracting the upper 32 bits of the lower 64-bit quarter word of the result. A straight-forward implementation of this could use `bn.and` to mask out 32 bits while the input is shifted to the right by 32. However, this comes at the cost of an additional cycle for the and-operation. The same result can also be achieved as follows: In the `bn.mulqacc.wo.z` instruction, set the shift amount to 192 such that the result will be shifted to the very top, discarding the undesired upper 32 bits of the multiplication result. As the next operation is the addition of 2^α using `bn.add`, we can again apply a shift – but this time, to one of the operands prior to execution of the actual operation. This way, we can shift the input to the right by 160 bits, such that the desired 32 bits align with 2^α in the constant register. The fact that the addition will introduce q in the third and some rest of the multiplication result in the bottom quarter word is not of concern, as the next instruction is a `bn.mulqacc.wo.z` for which we can select the second quarter word as the input, only precisely containing $\left\lfloor \frac{abq' \bmod 2^{2\ell}}{2^\ell} \right\rfloor + 2^\alpha$.

Listing 3.3 gives the instruction sequence for computing the Plantard multiplication in four cycles on OTBN, where `bprime` refers to the precomputed constant and `const_reg` contains 64 bits of zeros in the bottom quarter word, 2^α in the second one and q in the third one.

```
1 bn.mulqacc.wo.z a, a.0, bprime.0, 192
2 bn.add          a, const_reg, a >> 160
3 bn.mulqacc.wo.z a, a.1, const_reg.2, 0
4 bn.rshi         t, const_reg, a >> 32
```

Listing 3.3: Plantard Multiplication on OTBN.

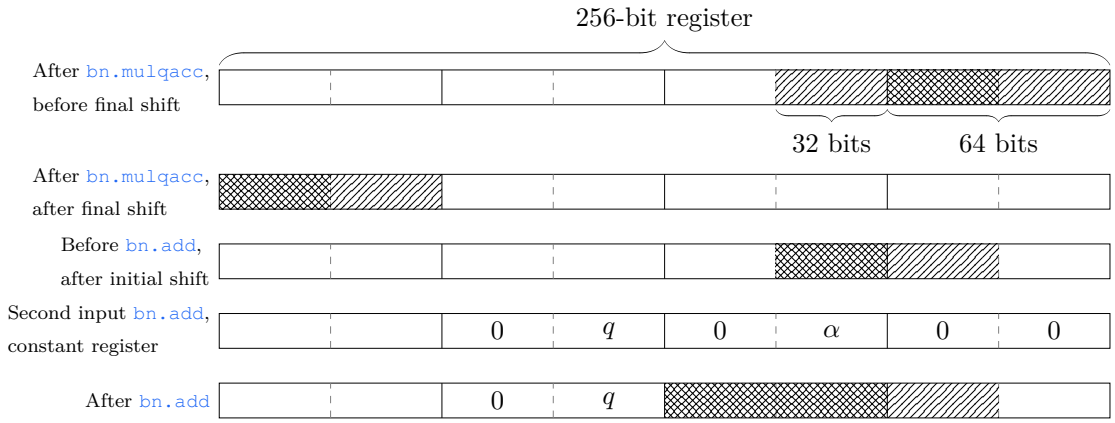


Figure 3.3: Register view of implicit reduction modulo $2^{2\ell}$ and right shift by ℓ in Plantard multiplication.

In the following, we want to comment on one disadvantage of the improved Plantard multiplication that has been highlighted in [HZZ+22]: As the multiplication with m' in Algorithm 2.6.7 takes place in $\mathbb{Z}_{2^{2\ell}}$, and the twiddle factors in Dilithium normally fit into a 32-bit integer, the amount of data is doubled for $\ell = 32$. While this takes up more data memory to store the values, we argue that on OTBN, there is no runtime overhead due to this. When loading a set of 32-bit twiddle factors from data memory to a WDR, eight twiddle factors will be packed inside the register. As `bn.mulqacc.wo.z` takes 64-bit quarter words as the input, the twiddle factor that is supposed to be input to the multiplication needs to be extracted from the register containing the other ones first. This extraction costs at least one additional cycle per twiddle factor, at least matching the increased cost incurred by the higher number of loads from the memory.

For the multiplication of two variables, the improved Plantard algorithm will take one cycle more as the multiplication with m' cannot be precomputed and needs to be explicitly performed. Still, the cycle count is on par with the approach of the Barrett reduction from [Sei18], and thus, for consistency reasons, we decided to also choose Plantard's approach for the variable-input pointwise multiplication in our implementation.

3.4.3 Polynomial Arithmetic

This section explains our implementation of polynomial arithmetic for Dilithium on OTBN, with special emphasis on polynomial multiplication.

Polynomial Addition & Subtraction

On OTBN, one approach to add two polynomials is to load eight of their coefficients to two WDRs, extract the individual coefficients, add them together using the `bn.addm` instruction, and store the result after eight additions, filling one WDR, have been computed. This way, the resulting polynomial's coefficients are also implicitly reduced

modulo q. For the subtraction, the exact same method works using `bn.subm` instead.

Instead of processing individual 32-bit coefficients inside the 256-bit wide WDRs, an alternative approach is to imitate SIMD-arithmetic using the existing big-number instructions. Adding two WDRs containing eight unsigned 32-bit coefficients each using `bn.add` gives a correct result w.r.t. the addition of their individual coefficients if the results do not overflow 32 bits. We call this "pseudo-vector addition". Unfortunately, this approach does not transfer to subtractions and multiplications: Subtractions that cause a negative result would influence the other coefficients inside the WDR. For multiplications, there only is a 64×64-bit multiplier available, which prohibits the use of the register's full width. If considering packing two coefficients inside 64 bits and performing the multiplication, the result of the "lower" part could affect the part in the top, depending on the size of the operands.

Despite the savings in terms of cycles, this technique prevents the implicit reduction that we get from the `bn.addm` instruction, meaning the results will grow beyond q.

Therefore, we decided to apply this approach for polynomial additions when this would not cause an additional modular reduction in the following, as this would cost more cycles than we save in the first place. We compare the implementations in Listings 3.4 and 3.5.

```
1  loopi 32, 9
2    bn.lid vals_1_idx, 0(src1++)
3    bn.lid vals_2_idx, 0(src2++)
4
5    loopi 8, 5
6      /* Mask one coefficient to working
       registers */
7      bn.and val_1, vals_1, mask
8      bn.and val_2, vals_2, mask
9      /* Shift out used coefficient */
10     bn.rshi vals_1, bn0, vals_1 >> 32
11
12     bn.addm val_1, val_1, val_2
13     /* Append to result */
14     bn.rshi vals_2, val_1, vals_2 >> 32
15
16   bn.sid vals_2_idx, 0(dst++)
```

Listing 3.4: Polynomial addition on OTBN.

```
1  loopi 32, 4
2    bn.lid vec_1_idx, 0(src1++)
3    bn.lid vec_2_idx, 0(src2++)
4
5    bn.add vec_1, vec_1, vec_2
6
7    bn.sid vec_1_idx, 0(dst++)
```

Listing 3.5: Pseudo-vectorized polynomial addition on OTBN.

In the case of the accumulation in the pointwise multiplication during the matrix-vector product, we add to the same polynomial multiple times in a row. Implementing the accumulation on individual coefficients, assuming that eight coefficients to accumulate onto have been loaded to a WDR, requires three steps that, repeated $n = 256$ times:

1. For each pointwise multiplication, the exact coefficient that is the target of the accumulation is extracted from the WDR using, e.g., `bn.and`.

2. The actual accumulation is implemented using `bn.addm`, giving an implicit reduction by q.

3. Finally, the WDR holding the eight coefficients needs to be advanced to make the next coefficient the bottom one.

This comes down to a cost of $256 \cdot 3 = 768$ cycles for the accumulation per polynomial. Using the pseudo-vector approach, we can trade these 768 cycles for only $\frac{256}{8} = 32$ `bn.add`, saving 736 cycles per pointwise multiplication with accumulation. As for Dilithium2, there are $l - 1 = 3$ of those accumulating polynomial multiplications in the matrix-vector product, we save a total of $3 \cdot 736 = 2208$ cycles. However, this also comes at the disadvantage of the coefficients not being implicitly reduced. As the output of the matrix-vector product is input to an INTT, not reducing the coefficients beforehand could cause 32-bit integer overflows. Thus, we reduce the result using a variant of the reduce32 function, returning coefficients in $[0, q)$, which takes 1696 cycles per polynomial. More details on the reduction can be obtained from Section 3.4.6. Therefore, from a performance standpoint, it is beneficial to make use of the pseudo-vector approach for the accumulation during the matrix-vector product. A comparison of the two implementations can be obtained from Listings 3.6 and 3.7.

The NTT

In this section, we will give details on our implementation of the NTT containing various common optimization techniques as well as some specific aspects with respect to the OTBN platform.

The Butterfly. Listing 3.8 shows our implementation of the CT-butterfly on OTBN. The first four instructions implement the Plantard multiplication with the twiddle factor. We implement the addition and subtraction of the butterfly using `bn.addm` and `bn.subm` to prevent the growth of the coefficients throughout the NTT.

```
1 bn.mulqacc.wo.z coeff8, coeff8.0, tf1.0, 192 /* c8*(wq') mod 2^21 */
2 bn.add          coeff8, wtmp3, coeff8 >> 160 /* + 2^alpha, >> 1 */
3 bn.mulqacc.wo.z coeff8, coeff8.1, wtmp3.2, 0 /* *q */
4 bn.rshi         wtmp, wtmp3, coeff8 >> 32    /* >> 1 */
5 bn.subm         coeff8, coeff0, wtmp
6 bn.addm         coeff0, coeff0, wtmp
```

Listing 3.8: CT-butterfly on OTBN.

Layer Merge. Section 2.5.3 explains how usually $\log n$ layers are consecutively computed, each consisting of $\frac{n}{2}$ butterfly operations over all of the n coefficients. This implies that every coefficient needs to be loaded on every layer. A common optimization technique for the NTT and INTT is called "layer merging". The idea behind it is to load sets of coefficients such that they are the inputs to multiple layers in a row. Thus, merging two of the eight layers, we save $\frac{1}{8}$ of the load operations. Let us consider an example for an architecture with four registers and $n = 256$. In an implementation that does not make use of layer merging, coefficients $a_0, a_{128}, a_1, a_{129}$ would be loaded, and the butterflies on (a_0, a_{128}) and (a_1, a_{129}) computed before storing the results back to memory and

```
loopi 32, 17
  bn.lid vals_1_idx, 0(src1++)
  bn.lid vals_2_idx, 0(src2++)
  bn.lid vals_acc_idx, 0(acc)
  loopi 8, 12
    /* Mask one coefficient to working
    registers */
    bn.and val_1, vals_1, mask
    bn.and val_2, vals_2, mask
    bn.and val_acc, vals_acc, mask

    /* Shift out used coefficient */
    bn.rshi vals_1, bn0, vals_1 >> 32
    bn.rshi vals_2, bn0, vals_2 >> 32

    /* Multiply */
    bn.mulqacc.wo.z val_2, qprime,
    val_2.0, 0
    bn.mulqacc.wo.z val_1, val_2.0,
    val_1.0, 192
    bn.add          val_1, const_reg,
    val_1 >> 160
    bn.mulqacc.wo.z val_1, val_1.1, q, 0
    bn.rshi         val_1, const_reg,
    val_1 >> 32

    /* Accumulate */
    bn.addm val_1, val_1, val_acc
    /* Append result to output */
    bn.rshi vals_acc, val_1, vals_acc >>
    32

  /* Store 8 coefficients */
  bn.sid vals_acc_idx, 0(acc++)
```

Listing 3.6: Pointwise multiplication with 32-bit accumulation on OTBN.

```
loopi 32, 15
  bn.lid vals_1_idx, 0(src1++)
  bn.lid vals_2_idx, 0(src2++)
  bn.lid vec_acc_idx, 0(acc)
  loopi 8, 9
    /* Mask one coefficient to working
    registers */
    bn.and val_1, vals_1, mask
    bn.and val_2, vals_2, mask

    /* Shift out used coefficient */
    bn.rshi vals_1, bn0, vals_1 >> 32

    /* Multiply */
    bn.mulqacc.wo.z val_2, qprime,
    val_2.0, 0
    bn.mulqacc.wo.z val_1, val_2.0,
    val_1.0, 192
    bn.add          val_1, const_reg,
    val_1 >> 160
    bn.mulqacc.wo.z val_1, val_1.1, q, 0
    bn.rshi         val_1, const_reg,
    val_1 >> 32

    /* Append result to output */
    bn.rshi vals_2, val_1, vals_2 >> 32

  /* Accumulate */
  bn.add vals_2, vals_2, vec_acc

  /* Store 8 coefficients */
  bn.sid vals_2_idx, 0(acc++)
```

Listing 3.7: Pointwise multiplication with pseudo-vectorized accumulation on OTBN.

proceeding with loading $a_2, a_{130}, a_3, a_{131}$. If we were to merge two layers, we would load the coefficients in a different order, starting with $a_0, a_{128}, a_{64}, a_{192}$. Then, the butterflies on (a_0, a_{128}) and (a_{64}, a_{192}) could be computed, immediately followed by the computation of the butterflies between (a_0, a_{64}) and (a_{128}, a_{192}) for the second layer. This way, we merge the first and second layer.

The amount of layers that can be merged is mostly limited by the number of registers; in order to merge m layers, 2^m coefficients are required. Usually, it is the best approach to merge as many layers as possible, such that the number of times every coefficient needs to be loaded is reduced. For example, in a 4–4 layer merge, meaning the first four and last four layers are merged, every coefficient only needs to be loaded twice, while a 1–3–4 merge would result in every coefficient being loaded three times. Although the 4–4 merge is also the best strategy on OTBN, we want to argue why this is not immediately clear. Consider the first four layers: For example, we would first compute the butterflies between coefficients $a_0, a_{16}, a_{32}, \ldots, a_{240}$, then $a_1, a_{17}, a_{33}, \ldots, a_{241}$ in the next iteration, and so on. We can note that the coefficients used in one iteration are

immediately adjacent in memory to the ones used in the next iteration (i.e., the coefficient at index 0 is next to the one at index 1). Thus, we could load eight adjacent coefficients using one `bn.lid`, e.g., $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$, "cache" them inside one WDR and extract one coefficient into a "working WDR" in each iteration. This strategy avoids "wasting" data that has already been loaded but is not immediately required. Once the computation of four layers of NTT on 16 coefficients is done, the coefficients could be individually shifted into the caching registers from the top, making the coefficient that will be used in the next iteration the bottom-most. Employing this strategy for all of the 16 coefficients in an iteration would require 16 registers to hold the "cached" data and 16 additional registers for the "working state", the coefficients the current iteration operates on. Since there are only 32 WDRs available, this would leave no registers for twiddle factors, temporary results, or other constants. The minimum of WDRs needed for other purposes is three: One register to hold twiddle factors, one for various constants like q, α, as well as a number of zero-bytes, and one for storing the temporary result of the multiplication with the twiddle factor. Therefore, we can apply the caching technique only for 13 registers. For the other registers, we need to load the correct coefficient just in time, costing more memory operations and cycles. This comes at a high cost mainly due to the fact that loads to a WDR need to aligned on 32 bytes, and thus, we cannot load arbitrary coefficients without additional effort. One approach to solving this issue is to load a full WDR, shift the desired coefficient to the bottom, and mask it out. The problem on OTBN is that there is no big-number shift instruction that takes the shift amount from a register, meaning it is not possible to automatically increment the shift amount with each iteration (i.e., advancing to the next coefficient) without fully unrolling the loop. Thus, we deem this strategy non-favorable. Another possible solution is to load the data from memory using a 32-bit `lw`, such that the alignment is of no relevance, then store this value to a 32 B-aligned address using `sw`, and read it again using a `bn.lid` from there into a WDR (the opposite way works for storing). This way, we can precisely load/store the coefficients we desire at the cost of five cycles per coefficient, causing a total cost of $16 \cdot (3 \cdot 5 \cdot 2) = 480$ cycles. The factor of 16 is due to the number of iterations; the factor of two stems from the fact that we need to load and store arbitrarily once per iteration, and we need to multiply the cost by three as we have three registers for which we cannot cache. Though, in order to have 13 registers available for the "caching", we reload the twiddle factors in each iteration, which costs additional cycles compared to loading them only once in the beginning. The cost for this comes down to four `bn.lid` per inner iteration, totaling to $16 \cdot 4 \cdot 2 = 128$ cycles.

When merging less than four layers, this situation cannot occur since we only require eight registers for our working state and eight more registers for "caching" coefficients, leaving us with 16 freely usable registers. However, one more merge comes at the cost of a full layer of loads and stores, which amounts to $\frac{256}{8}$ `bn.lid` and `bn.sid` each, and an additional 256 `bn.and` and 256 `bn.rshi` for extracting/packing the coefficients. This sums up to a total of $2 \cdot \frac{256}{8} \cdot 2 + 256 \cdot 2 = 640$ cycles.

Table 3.1 gives a structured overview of the comparison of the two approaches, showing that merging four initial layers saves 160 cycles. We only compare the first four layers since the handling of the data can be done optimally in the case of the last four layers.

The reason for this is that the required coefficients in one iteration are immediately adjacent in memory, making the use of caching superfluous.

Table 3.1: Comparison of layer merge approaches for the first four layers. Results refer to number of cycles.

	Instruction	4–X	1–3–X
Cached	`bn.lid`	$2 \cdot 13 \cdot 2 = 52$	$2 \cdot \frac{256}{8} \cdot 2 = 128$
	`bn.sid`	$2 \cdot 13 \cdot 2 = 52$	$2 \cdot \frac{256}{8} \cdot 2 = 128$
	`bn.and`	$16 \cdot 13 = 208$	$2 \cdot 256 = 512$
	`bn.rshi`	$16 \cdot 13 = 208$	$2 \cdot 256 = 512$
Non-Cached	`bn.lid`	$16 \cdot 3 \cdot 2 = 96$	—
	`bn.sid`	$16 \cdot 3 \cdot 2 = 96$	—
	`lw`	$16 \cdot 3 \cdot 2 \cdot 2 = 192$	—
	`sw`	$16 \cdot 3 \cdot 2 \cdot 1 = 96$	—
Twiddle Factors	`bn.lid`	$16 \cdot 4 \cdot 2 = 128$	$4 \cdot 2 = 8$
Total		1128	1288

Twiddle Factors. We apply the common optimization of precomputing the twiddle factors as powers of the root of unity and reading them from the memory when necessary during the NTT and INTT. This way, we save additional multiplications and exponentiations at the cost of space in the memory for the constants.

An optimization suggested in [ADPS16, Section 7.2] is to transform the twiddle factors into Montgomery domain ahead of time such that this costly procedure does not need to be performed at runtime. We apply the same approach to our implementation but for the Plantard domain instead of Montgomery. This approach is also applied in [HZZ+22].

Pseudo-vectorization. In the following, we consider the application of the pseudo-vector addition to the CT-based NTT: For 1024 butterfly operations over all eight layers, this approach saves $\frac{7}{8}$ of the addition instructions, which results in 896 cycles less. However, this comes at a major disadvantage: The coefficients that are the inputs to the addition need to be packed into a WDR. Consequently, for the output of the butterfly operation of one layer, the coefficients that will be multiplied with a twiddle factor or subtracted from on the next layer need to be extracted into individual WDRs. On each layer, we spend eight cycles packing the result of the multiplication into a vector and another eight cycles extracting the elements from the previous result vector to perform the subtraction. Finally, we spend at least eight more cycles while constructing the vector that will be added to on the next layer (e.g., going from $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$ to $(a_0, a_1, a_2, a_3, a_8, a_9, a_{10}, a_{11})$). This sums up to 24 cycles per layer for 16 coefficients, which gives a total of $24 \cdot 8 \cdot \frac{256}{16} = 3072$. The factor of eight is due to the circumstance that this packing and unpacking needs to be repeated for every layer individually. While for the

non-pseudo-vectorized implementation, packing and unpacking of the coefficients is also required, the coefficients need to be unpacked and packed into individual registers only after loading and before storing once per *merge*, not *layer*. This results in $2 \cdot 256 \cdot 2 = 1024$ cycles. Thus, from a performance perspective, the application of pseudo-vectorization inside the NTT is not beneficial. Additionally, due to the missing implicit reduction provided by `bn.addm`, some optimizations following the NTT could not be applied.

The INTT

Our INTT implementation is generally highly similar to our NTT, so we will refrain from reiterating the details given in the previous section and focus on the important differences.

The Butterfly. Listing 3.9 shows an implementation of the GS-butterfly on OTBN. Just as for the CT-variant, we implement the addition and subtraction using `bn.addm` and `bn.subm` for keeping the coefficients in $[0, q]$.

```
1 bn.subm         wtmp, coeff0, coeff1
2 bn.addm         coeff0, coeff0, coeff1
3 bn.mulqacc.wo.z wtmp, wtmp.0, tf1.0, 192  /* (c0-c1)*(wq') mod 2^2l */
4 bn.add          wtmp, wtmp3, wtmp >> 160  /* + 2^alpha, >> l */
5 bn.mulqacc.wo.z wtmp, wtmp.1, wtmp3.2, 0  /* *q */
6 bn.rshi         coeff1, wtmp3, wtmp >> 32 /* >> l */
```

Listing 3.9: GS-butterfly on OTBN.

Further Optimizations. We apply the common optimization of merging the multiplication with n^{-1} into the multiplication with the twiddle factor on the last layer, which saves $\frac{n}{2} = 128$ modular multiplication operations in total.

Pointwise Multiplication

Efficiently computing the pointwise multiplication inside the NTT-domain is essential for a high-speed implementation as it majorly contributes to the runtime. In Section 3.4.2, we outline how the improved Plantard multiplication can be used to compute the product of two coefficients in five cycles. Doing so, we implicitly apply a trick first introduced by Lyubashevsky and Seiler in [LS19, Section 5.3]. In their work, the authors postpone the transformation of one of the input coefficients to Montgomery domain to a later point in time, where it can be performed at no extra cost. More precisely, the multiplication with R for transformation into Montgomery domain is merged into the multiplication with n^{-1} at the end of the INTT. So instead of $ab \bmod m$, the result retrieved from the modular multiplication is $abR^{-1} \bmod m$ at first and only correct in the regular domain after the INTT. This trick is enabled by the linearity of the NTT and INTT. The only difference in our case is that the factor we merge into the multiplication at the end of the INTT is $-2^{2\ell}$, instead of the power-of-two R for Montgomery domain.

3.4.4 Sampling

This section explains our implementation of the sampling functions from DILITHIUM. We believe this is of special interest as we make heavy use of the interface to the KMAC core, present some mitigations for architecture-induced limitations with respect to the hardware loops, and also demonstrate interaction with the flag register.

Note that due to the lack of complex vector instructions, it is not possible to apply the optimization of parallelized rejection sampling as implemented presented in [GS16] and used in [LDK+22].

Uniform Sampling

Overview. First, we give a high-level overview of how we implement the uniform sampling of coefficients in $[0, q)$. Our goal is to output 256 coefficients, each made up of 3 B of SHAKE output. Using the KMAC interface as described in Section 3.2.2, we can squeeze 32 B of SHAKE output into a WDR. From these 32 B, we can construct $\left\lfloor \frac{32\,\mathrm{B}}{3\,\mathrm{B}} \right\rfloor = 10$ coefficient candidates. We then check if the candidate is less than q and shift it into a WDR, which "accumulates" the elements fulfilling the condition. If this accumulator register is full, containing eight coefficients, we store it to the destination and check whether we have written 256 coefficients yet. We need to perform this check not for every candidate we test but only for every full WDR we store since we know that 256 is a multiple of 8 – the number of coefficients we store each time. After performing this check on all of the 10 candidates, we are left with 2 B of remaining SHAKE output. We then read the next 32 B from the KMAC core and take the first byte of those to make up one more coefficient with the two remaining bytes. We then proceed as before, ending up with 1 B of SHAKE output left. Loading 32 B once more gives us a total of 33 B, which is exactly enough for 11 coefficient candidates. If we have not constructed 256 coefficients at this point, we proceed again in the same way as described before. This means we use a total of $\mathrm{lcm}(3, 32\,\mathrm{B}) = 96\,\mathrm{B}$ of SHAKE output per iteration of the "main" loop.

The Details. The most crucial part of the sampling is inside the inner loop: checking if the coefficient candidate is less than q. Our implementation can be found in Listing 3.10. From the code, we can see that the actual comparison with q takes up three instructions in total as the flags need to be separately loaded and masked, while on platforms like, e.g., Armv7-M, there exist branch instructions that operate on the flags immediately. So, it is worth considering performing the check of the candidate using the 32-bit instruction set, which allows for more direct manipulation of the control flow: For a candidate c, instead of checking whether $q < c$, we can also check whether $q - c < 0$. This can be implemented using a subtraction followed by a right shift of 31 to get the sign bit. If the sign bit is set to one, we accept the coefficient; if it is set to zero, we reject it. So, we can reduce the instruction sequence to two instead of three instructions for the comparison. Though, to get the candidate from the WDR we read the SHAKE output to, into a GPR, we need to store it to the data memory first and load it from there. Even totally neglecting

the cost of storing the SHAKE output to the memory in the first place, this causes an overhead of two cycles per candidate for loading the coefficient and also prevents the approach of "accumulating" multiple checked coefficients inside a WDR before storing them to memory with a single `bn.sid`, meaning eight `sw` are required. It is clear that this additional effort would not be worth saving one cycle during the candidate check.

A second decision we have to make is whether to use a hardware loop or a while loop for iterating over the candidates. The advantage of the while loop is that we can early-exit it easily, while a disadvantage is that we require some instructions to handle the loop logic. With the hardware loop, we get the loop basically for free (one cycle for the `loopi` instruction), but we cannot immediately early-exit. Since we know that the early-exit case will only happen exactly once while sampling a full polynomial, we optimize the routine for the more frequent case of not exiting early, meaning we make use of a hardware loop. This loop has one branch instruction at the very top comparing the current destination pointer to a predefined value marking the last address of the output. If this address is reached, we simply branch to the (mandatory) last instruction of the loop. This way, we get an overhead of two cycles per iteration for the branch instruction and perform at most 10 superfluous iterations in which we skip to the end of the loop immediately, costing two cycles for the branch and one cycle for the last instruction. In contrast, implementing a while loop would require two additional instructions in every single iteration: One increment of a counter for 10 or 11 iterations and one branch based on this counter. So in the worst-case for the hardware loop approach, we have a cost for the control logic of $2 \cdot 257 + 4 \cdot 1$ and a maximum overhead of $10 \cdot (2 + 1)$, summing up to 548 cycles. For the while loop approach, we require three cycles in every iteration $3 \cdot 257 = 771$, meaning the runtime of the hardware loop approach causes at least about one third fewer cycles for performing the looping.

```
_poly_uniform_base_inner_loop:
  loopi 10, 12
    /* If done, skip to end */
    beq    dst, dst_done, _skip_store1
    /* Mask candidate from SHAKE output */
    bn.and cand, coeff_mask, shake_reg

    bn.cmp cand, mod
    csrrs  flags, 0x7C0, zero /* Read flags */

    /* flags = (Zero|Lsb|Msb|Carry) */
    andi flags, flags, 3 /* Mask flags */
    /* Reject if M, C are NOT set to 1, meaning NOT (q > cand) = (q <= cand) */
    bne  flags, three, _skip_store1

    bn.rshi accumulator, cand, accumulator >> 32
    addi    accumulator_count, accumulator_count, 1
    bne     accumulator_count, eight, _skip_store1 /* Accumulator not full yet */

    bn.sid accumulator_idx, 0(dst++) /* Store to memory */
    li     accumulator_count, 0
_skip_store1:
    /* Shift out the 3 bytes we have read */
    bn.or shake_reg, bn0, shake_reg >> 24
```

Listing 3.10: Inner loop of uniform sampling on OTBN.

Uniform Sampling in $[-\eta, \eta]$

The implementation for the sampling routine returning coefficients in $[-\eta, \eta]$ is very similar to the one for general, uniform sampling. However, there are some interesting differences we want to highlight.

First, we can avoid the masking of the flags returned by the `csrrs` instruction. This is due to the fact that instead of checking whether the 4-bit coefficient candidate is < 15, we can also check whether it is $\neq 15$, as this is the maximum value for a 4-bit integer nonetheless. When subtracting a candidate c from 15 and the result is zero, we will reject it. In this case, the zero flag will be set. At the same time, if the zero flag is set, the MSB and LSB flags will never be set. Moreover, the carry flag also cannot be set since we are subtracting $c \leq 15$ from 15 and thus $0 \leq 15 - c$. Thus, there is a unique flag bit-pattern that we can check against in the branch instruction.

Second, we implement a slight modification to the final subtraction in the construction of the coefficient: We perform the subtraction using `bn.subm` to get the coefficient in its unsigned representation at no additional cost.

3.4.5 Bit-packing

For the bit-packing, we can make use of OTBN's large registers and especially its `bn.rshi` instruction. Since all the packing and unpacking functions are highly similar in structure, we will only give a detailed description of one of them. As an example, we consider the function for packing coefficients that are in $[-\eta, \eta]$. In the case of Dilithium2, $\eta = 2$, meaning the coefficients can fit into three bits. We proceed by loading one WDR

containing eight coefficients and masking one of them out by applying an "and" operation with the value of `0xFFFFFFFF`. In the C reference implementation [Dil23], the coefficient to be packed is a signed integer, and thus, it is subtracted from η in order to retrieve an unsigned result in $[0, 2\eta]$. As we made the choice to operate on unsigned integers, we cannot simply perform this subtraction, as, e.g., -1 maps to $q-1$ in our case, and $\eta - (q-1)$ is certainly not in the desired range. All we need to do is to apply `bn.subm` instead of the regular `bn.sub`, which will move the result of the subtraction back into the positive domain, yielding values in $[0, 2\eta]$. Following that, we "append" the 3-bit value to a WDR using `bn.rshi` before finally discarding the used-up coefficient from the input register by shifting it to the right by 32 bits. The unpacking works exactly the other way around. For packing and unpacking functions that do not require a subtraction before the packing, the masking can be saved, and one coefficient can be packed in two cycles, disregarding the memory operations. The instruction sequences of the innermost loops are shown in Listings 3.11 and 3.12.

```
/* Mask */
bn.and work, in, mask
/* Subtract coefficient from eta */
bn.subm work, eta, work
/* Move coefficient into the output WDR */

bn.rshi out, work, out >> 3
/* Shift out used coefficient */
bn.rshi in, bn0, in >> 32
```

Listing 3.11: Inner loop of packing function for coefficients in $[-\eta, \eta]$ on OTBN.

```
/* Move coefficient into the output WDR */

bn.rshi out, in, out >> 6
/* Shift out used coefficient */
bn.rshi in, bn0, in >> 32
```

Listing 3.12: Inner loop of packing function for $\boldsymbol{w}_1$ on OTBN.

The situation is rather different for the implementation of the encoding and decoding of the hint polynomial vector $\boldsymbol{h}$. Especially the decoding in the C reference implementation [Dil23] uses a lot of control logic based on the signature data, as well as unaligned memory accesses, both of which are weaknesses of OTBN. Thus, we decided to implement the decoding and encoding using the base instruction set operating on 32-bit GPRs, which is more helpful for managing the control flow and less restricted regarding memory access. The reason why this operation still costs many cycles is the manual 4 B-alignment of addresses and the subsequent extraction of the desired byte, based on the lower two bits of the unaligned address, to simulate byte-aligned memory access.

3.4.6 Reductions

There are multiple places throughout the implementation where we require some form of explicit modular reductions:

1. After the application of pointwise multiplication with pseudo-vector accumulation, where the values can grow beyond q before the INTT. It is important for the input

values to the INTT to be in $[0, q]$ in order to avoid getting negative results that cannot be reduced back into the positive domain implicitly using `bn.subm`.

2. Before checking the norm bound: While the coefficients of the polynomials in our implementation mainly stay small, i.e., in $[0, q]$, we sometimes require their centralized representative in $[-\frac{q-1}{2}, \frac{q-1}{2}]$. Most notably, this is the case before checking the norm bound, as $\|w\|_\infty$ is defined as $\mid w \bmod^{\pm} q \mid$. In the reference implementation [Dil23], reduce32, as introduced in Algorithm 2.6.8, is applied to achieve this canonization.

For the latter case, knowing that the coefficients will be small, i.e., in $[0, q]$, before this operation allows us to implement a "short" version of the reduce32, which comes down to a conditional subtraction. For a comparison of the regular and the short version, see Listings 3.13 and 3.14. We additionally implement a third version that applies the same strategy as reduce32 but outputs the result reduced $\bmod^{+} q$, which is useful if we are going to keep computing on the result instead of checking the norm bound on it, for example, after a matrix-vector product using pseudo-vector accumulation. This can be achieved at no extra cost by replacing the final `bn.sub` with `bn.subm`.

```
1  loopi 32, 9
2    bn.lid coeffs_idx, 0(src++)
3
4    loopi 8, 6
5      bn.and coeff, coeffs, mask
6
7      bn.add          tmp, coeff,
       const_4194304
8      bn.rshi         tmp, bn0, tmp >> 23
9      bn.mulqacc.wo.z tmp, tmp.0, q.0, 0
10     bn.sub          coeff, coeff, tmp
11
12     bn.rshi coeffs, coeff, coeffs >> 32
13
14   bn.sid coeffs_idx, 0(dst++)
```

Listing 3.13: Implementation of reduce32 on OTBN.

```
1  loopi 32, 8
2    bn.lid coeffs_idx, 0(src++)
3
4    loopi 8, 5
5      bn.and coeff, coeffs, mask /* Mask
       out one coefficient */
6
7      bn.cmp coeff, const_4194304 /* Check
       threshold */
8      bn.sel tmp, bn0, q, C /* Conditional
       subtract */
9      bn.sub coeff, coeff, tmp
10
11     bn.rshi coeffs, coeff, coeffs >> 32
12
13   bn.sid coeffs_idx, 0(dst++)
```

Listing 3.14: Short version of reduce32 on OTBN.

3.4.7 Rounding

In this section, we will not go into the details of the implementation of all rounding functions but rather discuss the impact of an artifact incurred by the deployment of the improved Plantard arithmetic, namely, the fact that the coefficients are in $[0, q]$ instead of $[0, q)$, as well as the effect of operating on unsigned, instead of signed representatives. Generally, we follow the approach from the reference implementation [Dil23], diverging from Algorithms 2.4.10 and 2.4.16 for MakeHint and Decompose, computing as outlined in [LDK+22, Section 5.1] instead.

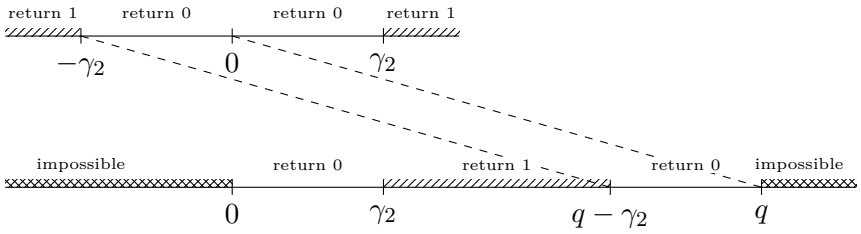


Figure 3.4: Modification to MakeHint for unsigned inputs.

For our version of MakeHint, we apply two modifications compared to the reference implementation [Dil23]: The first modification allows handling cases in which a coefficient is input as q instead of 0 by simply adding an additional check for one of the coefficients. This check only causes a minor overhead as it is only performed when all the previous ones did not cause a branching decision already. The second adapts the function to the fact that we operate on unsigned representatives, as opposed to signed ones, e.g., in the reference implementation. A visualization of this modification can be obtained from Figure 3.4. The implementations of UseHint and Decompose do not require any modification; they behave the same, regardless of whether an input is q or 0. The Power2Round function assumes a standard representative as its input and thus needs special treatment. We work around this by applying a single `bn.addm` to the input coefficient, in which we add "zero" to the coefficient in order to retrieve a standard representative.

3.5 Evaluation

In this section, we will give the results of the evaluation of our implementation in multiple ways. First, we are going to outline how we implemented the setup used to test our implementation for correctness. Second, we will address the performance of our implementation, giving details on the cycle counts and profiling information. Third, we will briefly touch on aspects relevant to our implementation's security.

Figure 3.5 gives an overview of how the setup we use for testing and benchmarking works. It is based on an instance of `otbn_sim_py_test`, which depends on `otbn_sim_test` instances for key generation, signing, and verification. The script `test_bench.py`, on the one hand, interfaces to the OTBN implementations, while it also has access to a pure Python implementation of Dilithium from [Pop23] called "dilithium-py".

3.5.1 Testing

For testing our implementation, we generate random inputs from inside `test_bench.py` and input them into our OTBN implementation as well as into dilithium-py. We compare the outputs of both operations in order to attest an error-free operation based on the random inputs. To verify our implementation for correct operation, we run more than 10 000 iterations for key generation, signing, and verification, respectively. On an Intel

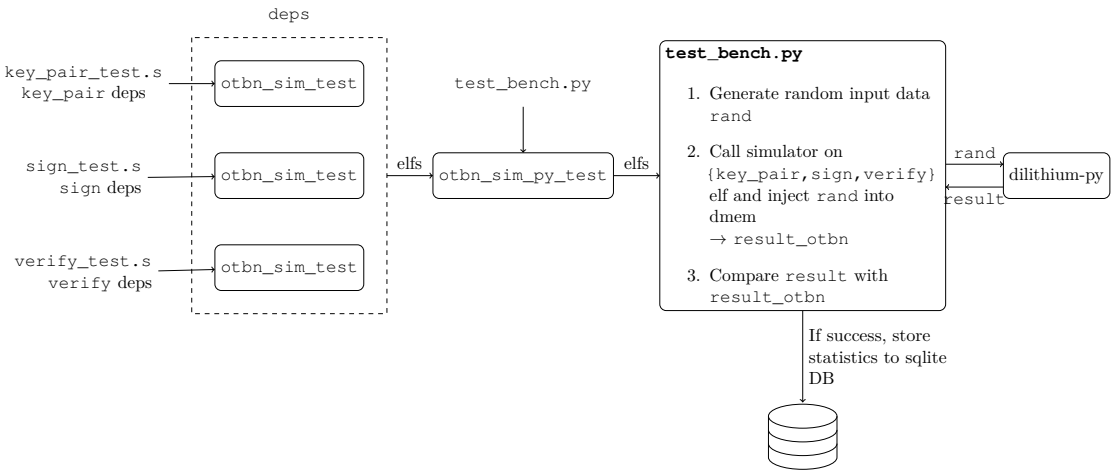


Figure 3.5: Flow for testing and benchmarking our Dilithium implementation on OTBN.

Core i9-9900, simulating one key generation or verification takes approximately 20 s, while generating one signature takes at least 45 s and can take up to multiple minutes, depending on how often a signature gets rejected. As a consequence, we parallelize the testing.

We deem this kind of testing setup robust for capturing major implementation flaws, although it cannot replace a formal verification of the implementation.

3.5.2 Performance

This section presents the performance results for our implementation. We obtain the statistical data using the `ExecutionStats` module from the OpenTitan repository, which provides this information for each simulator run.

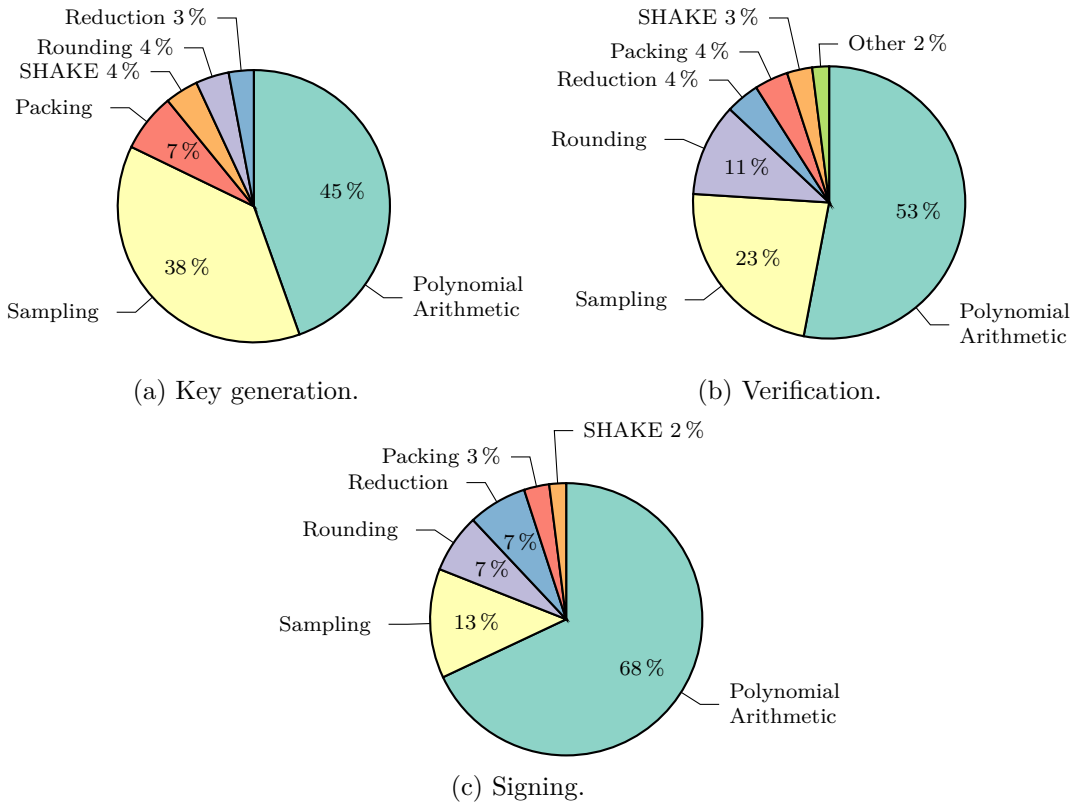
For the signature generation, and verification we consider messages of length 64 B, just as the benchmarking setup for the AVX2 implementation does [Dil23].

Profiling

Tables 3.2 to 3.4 give the accumulated cycle count spent in each of Dilithium's subroutines, averaged over 10 000 executions.

As the SHAKE computation on OTBN using the KMAC interface is invoked by an instruction, the cycle count for this specific instruction would normally count toward one of the other functions. However, in our analysis, we filtered out this specific instruction and separately counted the cycles towards a "virtual" function `SHAKE` because we think it is worthwhile to consider it independently. Therefore, we cannot give a proper number of calls for this "function".

In Figure 3.6, we group together functions based on their purpose and give the percentage for each group's runtime. A mapping between the function name and the group can be obtained from Table B.1. For all three operations, the polynomial arithmetic



(a) Key generation.

(b) Verification.

(c) Signing.

Figure 3.6: Dilithium2 cycle count profiling on OTBN, median values over 10 000 iterations. Groups with less than 1% not displayed. Percentages may not add up to 100% due to rounding.

is the largest group with respect to the cycle count, always followed by the sampling. Notably, SHAKE accounts for only a small fraction of the runtime, e.g., 1% for the signature Dilithium2 generation, while, for example, on Intel Skylake with AVX2, the SHAKE computation takes up around 50% of the total runtime in the signing.

More precisely, the largest part of the cycles spent for polynomial arithmetic in our implementations is incurred by NTT and INTT computations, as well as pointwise multiplications. Apart from that, the (sub)function handling the decomposition, as well as the uniform sampling make up a notable part of the runtime.

Polynomial Multiplication. In the following, we want to analyze the runtime of the polynomial multiplication-related functions in further detail.

As we know from Listings 3.8 and 3.9, each butterfly operation inside one of the transformations takes six cycles. As we need to compute a total of 1024 butterflies per eight layers on 256 coefficients, this amounts to $6 \cdot 1024 = 6144$ cycles. Since computing, e.g., one NTT takes a total of 8203 cycles, 2059 cycles are spent on other operations. Part of these other operations are also memory accesses, which account for $584+128+128 = 840$

Table 3.2: Dilithium2 key generation. Median number of instructions, stalls, total cycles, and number of calls for subroutines on OTBN, 10 000 iterations.

Function	#Instructions	#Stalls	Total Cycles	#Calls
poly_uniform_base_dilithium	44 735	13 982	58 717	16
poly_uniform_eta_base_dilithium	31 147	7145	38 292	8
intt_base_dilithium	32 980	1812	34 792	4
ntt_base_dilithium	31 000	1812	32 812	4
poly_pointwise_acc_base_dilithium	30 216	1572	31 788	12
SHAKE	432	10 330	10 762	0
poly_pointwise_base_dilithium	9812	396	10 208	4
poly_power2round_base_dilithium	8752	392	9144	4
polyeta_pack_dilithium	8744	360	9104	8
poly_reduce32_pos_dilithium	6596	268	6864	4
poly_add_base_dilithium	5664	388	6052	4
polyt0_pack_base_dilithium	4688	316	5004	4
polyt1_pack_dilithium	2584	300	2884	4
keccak_send_message	496	249	745	50
key_pair_base_dilithium	542	140	682	1
main	17	3	20	1

cycles, as analyzed in Section 3.4.3. This implies that we spend 1219 on some other operations. These other operations include the setup of the constants, but most notably, the extraction of individual coefficients to and from WDRs in order to operate on them one by one. This cost is incurred by the fact that, on the one hand, we need to load the data to the WDRs as they provide suitable arithmetic instructions (compared to the instructions operating on the GPRs). On the other hand, the architecture does not allow for computing the butterfly operation on multiple coefficients that have been loaded into a WDR at once.

Apart from that, the result of the INTT taking more cycles than the NTT follows our expectations as it requires additional multiplication with n^{-1} for half of the coefficients after the last layer.

Advantage of Pseudo-Vectorization. In Section 3.4.3, we introduced the notion of pseudo-vectorization to speed up polynomial additions as well as the accumulation in the pointwise multiplication for the matrix-vector product. Table 3.3 undergirds our theoretical considerations: One polynomial addition using `poly_add_base_dilithium` takes 1513 cycles compared to 233 using `poly_add_pseudovec_base_dilithium`. Both of these numbers include the same cost for memory accesses that sums up to $\frac{256}{8} \cdot 3 \cdot 2 = 192$ cycles, implying a $> 30\times$ speed-up for the parts of the code that differ between the two implementations. The reason we still have instances of `poly_add_base_dilithium` throughout the implementation is that it is not worth introducing an additional reduction

costing 1460 cycles. Only in the case where multiple polynomial additions in a row are performed the savings exceed the cost of an additional reduction.

Full Scheme

We give the cycle counts for the full-scheme benchmarks in Table 3.5.

For the key generation and verification, we can observe from Table 3.5 that the mean and median are almost identical and the standard deviation is also low, while the signing in contrast has a large difference between the mean and median and a high standard deviation. This result is to be expected, as Dilithium is based on the Fiat-Shamir with Aborts approach, and thus, depending on the randomness, multiple signature candidates need to be generated before a suitable one is found.

A comparison to other architectures will follow in Chapter 5.

3.5.3 Security

During the development of our implementation, we oriented ourselves at the C reference implementation from [Dil23] and took great care to implement all operations on secret data in constant-time. This means that we do not perform branching decisions based on secrets or secret-dependant table lookup. As OTBN does not perform any forms of speculative execution, we do not need to apply special protection against Spectre-type attacks.

We do not implement any countermeasures against other side-channel attacks, such as power or Electro-Magnetic (EM) analysis. An interesting avenue for future work would be to explore how masking or other side-channel protection measures for lattice-based cryptography schemes would be applicable to OTBN.

Table 3.3: Dilithium2 signature generation. Median number of instructions, stalls, total cycles, and number of calls for subroutines on OTBN, 10 000 iterations.

Function	#Instructions	#Stalls	Total Cycles	#Calls
intt_base_dilithium	296 820	16 308	313 128	36
ntt_base_dilithium	209 250	12 231	221 481	27
poly_pointwise_acc_base_dilithium	90 648	4716	95 364	36
poly_pointwise_base_dilithium	88 308	3564	91 872	36
poly_chknorm_base_dilithium	49 507	14 812	64 319	20
decompose_base_dilithium	59 136	384	59 520	384
poly_uniform_base_dilithium	44 740	13 983	58 723	16
poly_reduce32_pos_dilithium	19 788	804	20 592	12
poly_reduce32_dilithium	19 788	804	20 592	12
poly_reduce32_short_dilithium	16 716	804	17 520	12
poly_caddq_base_dilithium	16 692	792	17 484	12
poly_make_hint_dilithium	13 004	4168	17 172	4
SHAKE	617	15 565	16 182	0
poly_uniform_gamma1_base_dilithium	14 292	1248	15 540	12
poly_sub_base_dilithium	11 328	776	12 104	8
polyeta_unpack_base_dilithium	8680	360	9040	8
polyw1_pack_dilithium	7176	660	7836	12
polyvec_encode_h_dilithium	4716	2116	6832	1
poly_add_base_dilithium	5664	388	6052	4
polyz_pack_base_dilithium	4920	468	5388	4
polyt0_unpack_base_dilithium	4576	316	4892	4
poly_challenge	2997	968	3965	3
poly_decompose_dilithium	1860	1596	3456	12
sign_base_dilithium	2590	649	3239	1
poly_add_pseudovec_base_dilithium	1632	1164	2796	12
keccak_send_message	724	363	1087	69
main	20	4	24	1

Table 3.4: Dilithium2 verification. Median number of instructions, stalls, total cycles, and number of calls for subroutines on OTBN, 10 000 iterations.

Function	#Instructions	#Stalls	Total Cycles	#Calls
ntt_base_dilithium	69 750	4077	73 827	9
poly_uniform_base_dilithium	44 735	13 982	58 717	16
intt_base_dilithium	32 980	1812	34 792	4
poly_pointwise_acc_base_dilithium	30 216	1572	31 788	12
poly_pointwise_base_dilithium	19 624	792	20 416	8
decompose_base_dilithium	19 712	128	19 840	128
poly_use_hint_dilithium	10 732	4789	15 521	4
poly_chknorm_base_dilithium	10 320	3088	13 408	4
SHAKE	393	8877	9270	0
poly_reduce32_pos_dilithium	6596	268	6864	4
poly_sub_base_dilithium	5664	388	6052	4
poly_caddq_base_dilithium	5564	264	5828	4
polyz_unpack_base_dilithium	4740	468	5208	4
verify_base_dilithium	4263	452	4715	1
polyt1_unpack_dilithium	2660	304	2964	4
polyw1_pack_dilithium	2392	220	2612	4
polyvec_decode_h_dilithium	1404	345	1749	1
poly_challenge	993	320	1313	1
keccak_send_message	434	218	652	38
main	22	4	26	1

Table 3.5: Full-scheme benchmarks on OTBN, 10 000 iterations. All numbers given refer to cycles.

Operation	Key Generation	Signing	Verification
Mean	257 891	1 421 552	315 588
Median	257 888	1 100 386	315 586
Standard Deviation	100	951 060	151

4 Extending OTBN for Lattice-based Cryptography

In this chapter, we describe the process of extending the OTBN core's instruction set in order to allow efficient computation of lattice-based cryptography and give details on how we optimize our base implementation from Chapter 3 under the application of the aforementioned extensions. We follow a similar structure as in Chapter 3 by first describing the goal of our endeavor and then describing how we modify the architecture before coming to the description of our implementation.

We will refer to OTBN including our instruction-set extensions as OTBN* in the following, which is a superset of the OTBN core.

4.1 Goals of the Extension

As already mentioned in the introduction to this chapter, the goal is to improve the efficiency of implementations of lattice-based cryptography on OTBN, with Dilithium as a case study by extending the OTBN architecture. Just as for the base implementation, we consider efficiency as a measure of cycle count and disregard the memory usage and code size.

While instruction sets, for example, on the Intel platform, offer a large quantity of helpful yet specialized instructions, this also causes a blow-up of the instruction set. This may not be a concern for Complex Instruction Set Computer (CISC)-based architectures, but Reduced Instruction Set Computer (RISC) architectures like, e.g., Armv8-A usually offer fewer instructions. Still, with Armv8-A having instructions of 64 bits in size, it offers dozens of rather complex ones. In contrast, the RISC-based OTBN only offers 32-bit wide instructions, which makes opcodes scarce and leaves less room for inputs to the instructions. Thus, we plan to take a more limited approach by adding only a handful of instructions. In doing so, we try to avoid adding instructions that are too specialized in their functionality so that other applications apart from lattice-based cryptography can also profit from them.

4.2 Modifications to the Architecture

We apply the same fundamental modifications to the architecture that we also describe in Chapter 3, namely interfacing with the KMAC core, as well as expansion of the data and instruction memory. Since we disregard the concrete hardware implementations, all

of our extensions described in the following will be implemented as extensions to the OTBN Python simulator.

With respect to the instruction set, we mainly base our decision on the observations from Section 3.5.2, where we found that the major contributor to the cycle count is the polynomial arithmetic, especially the polynomial multiplication. Thus, it is key to speed up the NTT, INTT, and pointwise multiplication, which basically consist of (modular) additions, subtractions, and multiplications.

In order to achieve this, we deem the addition of vector instructions worthwhile. First, we have seen in Section 3.5.2 how much performance gain could be achieved by applying the technique of pseudo-vectorization for the polynomial addition, fully utilizing the 256-bit wide registers. Second, the NTT and INTT lend themselves to parallelization as the butterfly operations on one layer are all independent of each other. The same holds for the pointwise multiplication, in which the products do not depend on each other. Thirdly, prior work also indicates that vectorized implementations of lattice-based cryptography schemes, for example, using Intel AVX2 or Arm Neon, improve the performance significantly [LDK+22, BHK+22].

In the following, we will refer to a WDR as a "vector" if its individual 32-bit (or 16-bit) sub-words have a meaningful interpretation. We refer to these sub-words as "elements". The position such an element resides at inside the vector is referenced by an "index" between 0 and 7 (or 15), which is also called "lane". For example, the element at index 0, or in lane 0, is stored at bits 0 to 31 (or 15) of the 256-bit wide register.

Our first three proposals immediately follow from our above observations. We decided to add three instructions, namely `bn.addv`, `bn.subv`, and `bn.mulv`, for (modular) vector addition, subtraction, and multiplication, respectively. Additionally, to provide one more fundamentally basic instruction, we add support for vectorized bit-shifts through an instruction `bn.shv`, that allows to apply an immediate-defined shift to each element of its operand WDR. Lastly, we add an instruction for data permutation called `bn.trans`, which can be used for distributing data from one WDR to multiple others. We go into more detail regarding the added instructions and their operation in the following.

Most of our instructions follow a syntax-wise similar pattern. The mnemonic is followed by an operand called `<type>`, which determines if a modular reduction should be applied and whether to interpret the WDRs of the inputs and outputs as eight elements of 32 bits each (`.8S`) or 16 elements of 16 bits each (`.16H`), e.g., `bn.addvm.8S`. The latter notation style follows the Armv8-A instruction set. Note that for our implementation of Dilithium, we only make use of the type `.8S`, but the `.16H` one may be interesting for small-moduli NTTs for Dilithium as in [AHKS22] or for implementations of Kyber.

- `bn.addv<type> <wrd>, <wrs1>, <wrs2>`: This instruction implements vectorized addition, optionally using centralized pseudo-modulo reduction. The instruction adds the individual elements of the source WDRs `<wrs1>` and `<wrs2>`, truncates the results to 32 bits, and stores them to the respective elements in `<wrd>`. Each individual element is interpreted as a signed integer of the respective bit-length, and the result is also stored in signed representation. If `m` is set in the type, then after the addition is performed, `MOD` is added to the result if it is smaller

than $-\lfloor\frac{\texttt{MOD}}{2}\rfloor$ or subtracted from the result is greater than $\lfloor\frac{\texttt{MOD}}{2}\rfloor$. This way, the operation is fully compatible with signed arithmetic. Note that this approach for the reduction only yields a result in $[-\lfloor\frac{\texttt{MOD}}{2}\rfloor, \lfloor\frac{\texttt{MOD}}{2}\rfloor]$ if the input elements have been in the same range prior to the computation.

- `bn.subv<type> <wrd>, <wrs1>, <wrs2>`: This instruction implements vectorized subtraction, optionally using centralized pseudo-modulo reduction. It operates in the exact same fashion as `bn.addv`, with the exception of performing subtraction instead of addition.

- `bn.mulv<type> <wrd>, <wrs1>, <wrs2>[, <lane>]`: The "Centralized modular vector multiplication" instruction offers one more option for the type: If the specifier of the element width is preceded by `.l`, e.g., `bn.mulvm.l.8S`, a mode for lane-wise operation is used ("vector-by-scalar multiplication"). This means, instead of multiplying the i-th element of `<wrs1>` with the i-th element of `<wrs2>`, a fixed element of `<wrs2>` at index `<lane>` is selected which every element of `<wrs1>` will be multiplied by. In either case, the results of the individual multiplications are truncated to the respective element size and stored to `<wrd>`. If the option for modular reduction is selected in the type, the result of the multiplication will be reduced as the centralized representative modulo `MOD`.

- `bn.shv<type> <wrd>, <wrs>[<shift_arith> ][<shift_type> <shift_bits>]`: This instruction implements the bitwise shift operation for individual vector elements. `<type>` can only be set to `.8S` or `.16H`, `<wrd>` gives the destination register, while `<wrs>` represents the input WDR. Optionally, `<shift_type>` defines whether to perform a left (<<) or right (>>) shift, and `<shift_bits>` is a 5-bit unsigned immediate holding the shift amount. When the optional operand `<shift_arith>` is set to `a`, the shift is performed as an arithmetic shift.

- `bn.trans<type> <wrd>, <wrs>, <wrd_lane>`: The "Vector transpose" instruction writes elements from one source register in `<wrs>` to lane at index `<wrd_lane>` of multiple destination registers `<wrd+i>` for `i`$\in [0, \#\text{ elements} - 1]$, where the number of elements is either 8 for `.8S` as `<type>` or 16 for `.16H` as `<type>`. We apply this indirect form of addressing for the destination registers as the 32-bit instructions do not allow to encode eight or even 16 registers into one instruction.

Next, we are going to justify the addition of each instruction to the instruction set. First, for `bn.addv`, `bn.subv`, and `bn.mulv`, it is clear that they can be applied to compute NTT butterfly operations very efficiently. The same holds for the pointwise multiplication, which basically collapses to three memory operations and a single arithmetic instruction per eight coefficients. Additionally, the polynomial addition and subtraction also heavily profit from the `bn.addv` and `bn.subv` instructions. Note that while our `bn.mulvm` instruction operates in a manner highly similar to the multiplication instruction presented

in [Saa23a], our approach has been developed independently. Additionally, we generally deem this approach rather natural.

We add the `bn.shv` instruction to perform bit shifts on the elements of a vector with bitwise precision. This can be useful in multiple scenarios, e.g., to fully vectorize the decomposition – making up a significant part of the cycle count for the signing as stated in Table 3.3 – or to extract sign-bits for a conditional addition with q. Also, the arithmetic part of the sampling of coefficients in $[-\eta, \eta]$ can profit from this.

We implement `bn.trans` to perform data movements of multiple elements between vectors in one single instruction. This instruction is of relevance to the NTT and INTT, which require a transposition of the vectors holding the coefficients at some point. We give more details on this in Section 4.3.2. Note that the exact operation for this transposition instruction is not important as long as it allows some meaningful permutation. On Arm Neon, for example, the instructions `trn1` and `trn2` allow moving elements at even or odd indices from two source registers to one single destination register, which could also be a viable option for implementing a transposition instruction on OTBN.

Finally, we comment on the decision to implement centralized (pseudo) reductions for our instructions. While for the majority of the implementation, it has no impact whether we use centralized representatives or ones in $[0, q)$, it makes a notable difference in the signing procedure: As the check of the norm bound is performed on $| w \bmod^{\pm} q |$ for a coefficient w, it removes the need of computing the centralized representatives before the check because the values are already reduced $\bmod^{\pm} q$. However, this comes at the cost of three additional conditional additions with q for a polynomial vector in order to move the coefficients into a positive domain: Once before Power2Round in the key generation, once before decomposition in the signing, and once before using the hint in the verification. We consider this trade-off as beneficial based on the following observations:

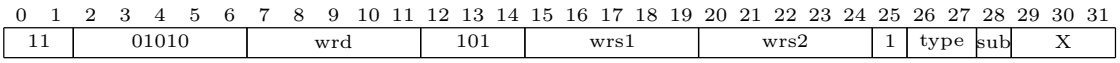
- The rejection sampling loop contains three checks for the norm bound, meaning we can save three reduce32 on polynomial vectors while adding only one conditional addition with q on polynomial vectors inside this loop.

- The rejection sampling loop is estimated to be executed more than four times for Dilithium2 [LDK+22], meaning a saving inside this loop is, on average, four times as beneficial as outside.

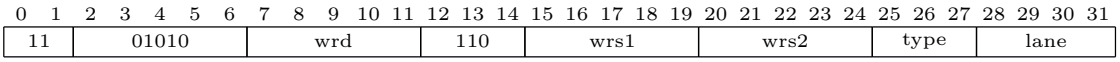
- The conditional additions with q inside the key generation and verification are not subject to rejection sampling.

- When implemented on OTBN*, conditional addition with q costs one cycle less than reduce32 per eight coefficients.

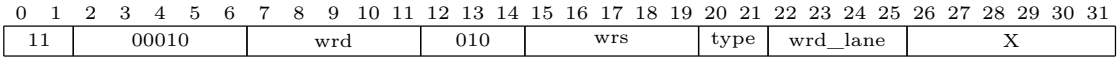
The encodings for our new instructions can be obtained from Figure 4.1. The naming of the fields in the encoding is equivalent to the naming of the operands of the previously introduced instructions. In Figure 4.1d, "art" is short for the parameter `<shift_arith>`, "ty" for `<type>`, and "st" for `<shift_type>`.



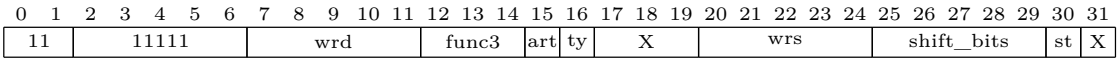
(a) bn.addv/bn.subv, integrated with bn.addm/bn.subm.



(b) bn.mulv.



(c) bn.trans.



(d) bn.shv.

Figure 4.1: Instruction encoding for proposed extensions.

4.2.1 Hardware Considerations

As the hardware implementation of the extensions we propose is out of the scope of this work, we will not perform a detailed analysis on this topic. Still, we want to briefly comment on the impact we expect our suggested extensions to have.

First, let us consider bn.addv and bn.subv: Via personal communication with the OpenTitan Team, we learned that the 256-bit wide addition for bn.add and bn.addm is implemented using eight separate 32-bit adders internally. Thus, the hardware to add this instruction is mostly already present; only more comparators are required to check the individual results before the conditional addition or subtraction.

In contrast to that, implementing the instruction bn.mulvm for arbitrary moduli could require an extensive amount of additional hardware in case a single-cycle execution is desired, according to the OpenTitan Team. At the same time, implementing this functionality in multiple cycles could drastically reduce the required amount of additional hardware. We will explore this avenue in a theoretical consideration in Sections 5.2.1 and 5.2.2. A different approach to this would be to refrain from implementing arbitrary-moduli modular reductions and give the choice between, e.g., Dilithium's and Kyber's moduli, as the two schemes have already been selected for standardization by NIST. The hardware cost for fixed-modulus reductions is usually less than for arbitrary moduli.

However, the fact that [Saa23a, Saa23b] propose very similar instructions as ours shows that, in general, our proposal is not preposterous.

We deem the implementation of bit shifts by the bn.shv instruction as appropriate compared to the additional hardware usage due to the barrel shifters, as this is a common feature on vector architectures.

Last, while the bn.trans instruction mainly requires wiring and multiplexers for its operation, the fact that the ALU has to address a total of nine registers for this

instruction may be a tougher constraint from a hardware perspective. Note that it would also be feasible to satisfy our needs with other types of permutations that require fewer registers as their input. However, for this work, we stick to our proposal due to its elegance.

4.3 Description of the Implementation

In this section, we will give details on our implementation of Dilithium on OTBN*. As we base our work on the implementation for the non-extended OTBN from Section 3.4, we will only focus on describing relevant differences. Also, it has to be noted that our extensions generally make the implementation more elegant and, in large parts, more straightforward than the baseline implementation.

4.3.1 Modular Arithmetic

As we include modular reduction in all of our added arithmetic instructions, we do not need to perform any explicit modular arithmetic.

4.3.2 Polynomial Arithmetic

Just as for our implementation on OTBN, we split the description on polynomial arithmetic into two parts. We start by briefly covering additions and subtractions before we give a detailed insight into our implementation of the polynomial multiplication.

Polynomial Addition & Subtraction

Already from the results of the pseudo-vector polynomial addition presented in Section 3.5.2, it has been clear that using proper vectorization on OTBN*, polynomial additions and subtractions can be implemented very elegantly – even including modular reductions of the coefficients. We show the main loop of the addition and subtraction functions in Listings 4.1 and 4.2. Note how our new implementation has a drastically reduced overhead for the data movement compared to Listing 3.4.

```
1 loopi 32, 4
2   bn.lid vec_1_idx, 0(src1++)
3   bn.lid vec_2_idx, 0(src2++)
4
5   bn.addvm.8S vec_1, vec_1, vec_2
6
7   bn.sid vec_1_idx, 0(dst++)
```

Listing 4.1: Polynomial addition on OTBN*.

```
1 loopi 32, 4
2   bn.lid vec_1_idx, 0(src1++)
3   bn.lid vec_2_idx, 0(src2++)
4
5   bn.subvm.8S vec_1, vec_1, vec_2
6
7   bn.sid vec_1_idx, 0(dst++)
```

Listing 4.2: Polynomial subtraction on OTBN*.

The NTT

This section presents implementation details of the NTT on OTBN*.

Butterfly. In contrast to the implementation from Section 3.4.3, the instruction sequence for the vectorized butterfly is simpler. Assume we were to compute the butterflies for the first layer on the elements in `vec0` and `vec8`, while the twiddle factors are loaded to `tf1`. The butterflies can be computed with just three instructions, as visible in Listing 4.3. Note that for the butterflies on later layers, where each of the eight parallel butterflies requires its own twiddle factor, we do not use the lane-indexed vector-by-scalar multiplication but rather a full vector-by-vector multiplication using `bn.mulvm.8S`.

```
1 bn.mulvm.l.8S tmp, vec8, tf1, 0
2 bn.subvm.8S   vec8, vec0, tmp
3 bn.addvm.8S   vec0, vec0, tmp
```

Listing 4.3: CT-butterfly on OTBN*.

Layer Merge. Our implementation of the NTT on OTBN* is structurally similar to the one presented in Section 3.4.3. We implement a 4–4 layer merge, meaning we use 16 WDRs to hold the coefficients throughout the execution. Also, we load from the exact same memory indices as in the first merge. This approach to merging in a vectorized implementation is closely related to the one taken in, for example, [BHK+22]. We can straightforwardly compute four layers as we load the vectors with an offset of 64 B, meaning 16 coefficients. For example, we have registers `w0` $= (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$, `w1` $= (a_{16}, a_{17}, a_{18}, a_{19}, a_{20}, a_{21}, a_{22}, a_{23})$, `w2` $= (a_{32}, a_{33}, a_{34}, a_{35}, a_{36}, a_{37}, a_{38}, a_{39})$, `w3` $= (a_{64}, a_{65}, a_{66}, a_{67}, a_{68}, a_{69}, a_{70}, a_{71})$, and `w4` $= (a_{128}, a_{129}, a_{130}, a_{131}, a_{132}, a_{133}, a_{134}, a_{135})$ with a_i being coefficient at index i. This allows for straightforward computation of four layers of vectorized NTT, as the stride for each element between (`w0`, `w4`) is 128, between (`w0`, `w3`) is 64, between (`w0`, `w2`) is 32, and between (`w0`, `w1`) is 16. In the next iteration, we increment the input pointer by 32 B and thus load from the indices with an offset of eight coefficients, meaning `w0` will start with a_8.

Computing the second merge requires more attention: We load the coefficients into the registers with no additional offset, meaning we end up with a_0 to a_{127} in our working state WDRs, where we, for example, have `w0` $= (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$, and `w1` $= (a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15})$. This means we can compute butterflies for the fifth layer, e.g., on (`w0`, `w1`), as the stride between the elements is eight. But for the remaining three layers, elements of stride four, two, and one are located inside the same vector. This means that we have to perform some form of data permutation in order to continue with our vectorized butterflies. This is where our `bn.trans` instruction comes into play: When we interpret eight WDRs of our working state as rows of a matrix, we get an 8×8 coefficient-matrix. Transposing this matrix will bring the working state into a shape that allows us to continue with the vectorized butterfly computations. Listing 4.4 shows an implementation of such transposition, where the state initially is stored in `w0`–`w7`, and the transposed state is output to `w8`–`w15`. Note that the working state includes 16

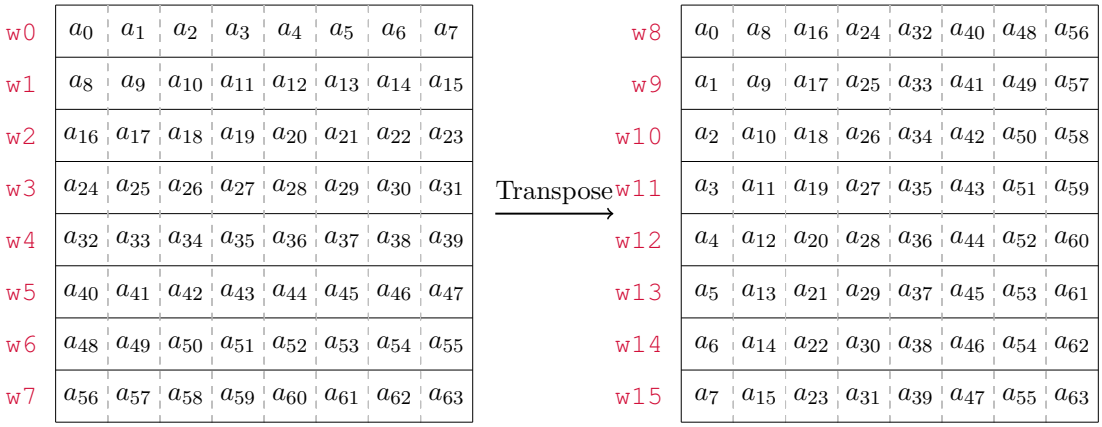


Figure 4.2: Visualization of the transposition.

WDRs, which means that the instruction sequence from Listing 4.4 will occur twice for transposing the complete working state. As it becomes clear from Figure 4.2, we would then proceed to compute the butterflies between (w8, w12), (w8, w10), and finally (w8, w9). After this step, the NTT computation is complete, though for a canonical coefficient ordering of the result, the transposition needs to be inverted. Incidentally, this is not strictly necessary: The pointwise multiplication would work regardless as long as the coefficient-ordering is consistent between the individual polynomials.

```
1 bn.trans.8S w8, w0, 0
2 bn.trans.8S w8, w1, 1
3 bn.trans.8S w8, w2, 2
4 bn.trans.8S w8, w3, 3
5 bn.trans.8S w8, w4, 4
6 bn.trans.8S w8, w5, 5
7 bn.trans.8S w8, w6, 6
8 bn.trans.8S w8, w7, 7
```

Listing 4.4: WDR state transposition during the NTT on OTBN*.

It is worth noting that such a transposition without the bn.trans instruction would cost two cycles per coefficient: One for extracting it from its source WDR and another to move it into the destination WDR. This would amount to $2 \cdot 256 \cdot 2 = 1024$ cycles per NTT or INTT, while the approach using bn.trans takes $2 \cdot 2 \cdot 16 = 64$ cycles, which is 16 times less.

The INTT

Our implementation of the INTT is highly similar to the one of the NTT, and therefore, we will skip over otherwise redundant explanations. The instruction sequence for the GS-butterfly can be obtained from Listing 4.5. We apply the same optimization as mentioned in Section 3.4.3 by merging the factor of n^{-1} into the twiddle factors of the last layer to save half of the multiplications with n^{-1}.

```
bn.subvm.8S   tmp, vec0, vec8
bn.addvm.8S   vec0, vec0, vec8
bn.mulvm.l.8S vec8, tmp, tf1, 6
```

Listing 4.5: GS butterfly on OTBN*.

Pointwise Multiplication

Using the `bn.mulvm.8S` instruction, the pointwise multiplication on two polynomials in NTT-domain can be computed highly efficiently costing only 224 cycles. The accumulation also only adds minimal overhead. We give a depiction of the implementations in Listings 4.6 and 4.7.

```
loopi 32, 4
  bn.lid      vec_1_idx, 0(src1++)
  bn.lid      vec_2_idx, 0(src2++)
  bn.mulvm.8S vec_1, vec_1, vec_2, 0
  bn.sid      vec_1_idx, 0(dst++)
```

Listing 4.6: Pointwise multiplication on OTBN*.

```
loopi 32, 6
  bn.lid      vec_1_idx, 0(src1++)
  bn.lid      vec_2_idx, 0(src2++)
  bn.mulvm.8S vec1, vec1, vec2
  /* Accumulate */
  bn.lid      vec_2_idx, 0(dst)
  bn.addvm.8S vec1, vec1, vec2
  bn.sid      vec_1_idx, 0(dst++)
```

Listing 4.7: Pointwise multiplication with accumulation on OTBN*.

4.3.3 Sampling

For the sampling of coefficients in $[-\eta, \eta]$, we can apply an optimization based on our vector instructions. When one of the sampled coefficient candidates satisfies the condition to be selected, we do not perform the arithmetic operations on it immediately but rather shift it into an "accumulator" register. We continue doing so until we collected eight coefficients inside the accumulator register and perform the arithmetic on all of them at the same time using our vector instructions. This way, the cost for arithmetic operations during the sampling is reduced by $\frac{7}{8}$. This optimization is enabled by the vectorized arithmetic, as well as shift instructions.

While we add a number of vector instructions to the architecture, implementing the vectorized rejection sampling presented in [GS16] and, for example, implemented in [Dil23] still requires some more advanced operations like vector comparisons and lookup table-based data transposition. Thus, is not feasible on OTBN* in its current state.

4.3.4 Bit-Packing

While the impact of our extensions to the bit-packing functions is rather limited, there is one interesting application: The subtraction of the coefficient from, e.g., η can be performed on the vector containing the coefficients prior to packing or after the construction

of the coefficient vector during unpacking. This way, eight extractions and subtractions of coefficients can be traded for a single instruction. The modified instruction sequence from Listing 3.11 can be found in Listing 4.8.

```
1 bn.subv.8S in, eta_vec, in
2 loopi 8, 2
3   bn.rshi out, in, out >> 3 /* Write one coefficient into the output WDR */
4   bn.rshi in, bn0, in >> 32 /* Shift out used coefficient */
```

Listing 4.8: Inner loop of packing function for coefficients in $[-\eta, \eta]$ on OTBN*.

4.3.5 Reductions

Throughout the whole implementation, we do not require explicit modular reductions as our arithmetic instructions already include them. Though, there are three occasions, already mentioned in Section 4.2, where we need to transform centralized representatives into representatives in the positive domain. For this, we re-implement the `poly_caddq` function from the C reference implementation [Dil23]. In this implementation, we make use of our vectorized shift, which we use to build a mask from the sign bit, as well as addition to bring the coefficients into positive domain, if they were not already.

4.3.6 Rounding

We implement fully vectorized variants of Decompose and Power2Round that allow us to process eight coefficients at once. In Listings 4.9 and 4.10, we show the implementation of both functions for a full polynomial. While it may be clear, note that although the decomposition requires "exclusive or" and logical "and" operations, we do not require specialized instructions for them as bitwise operations result in exactly the same output, no matter whether performed on individual elements or the whole vector. The implementations MakeHint and UseHint – apart from the decomposition – do not lend themselves to vectorization using our added instructions very well. For this reason, and their only minor contribution to the cycle count in the base implementation, we omit optimizing them further.

As Listings 4.9 and 4.10 exhibit, these optimizations are enabled by an interplay of our vector arithmetic and shift instructions.

```
loopi 32, 4
  bn.lid a_idx, 0(src++)
  /* Compute "a1" */
  bn.addv.8S tmp, a, vec_127
  bn.shv.8S  tmp, tmp >> 7
  bn.mulv.8S tmp, tmp, vec_11275
  bn.shv.8S  tmp2, vec_1 << 23
  bn.addv.8S tmp, tmp, tmp2
  bn.shv.8S  tmp, tmp >> 24
  bn.subv.8S tmp2, vec_43, tmp
  bn.shv.8S  tmp3, tmp2 a >> 3
  bn.and     tmp2, tmp, tmp3
  bn.xor     a_1, tmp, tmp2

  /* Compute "a0" */
  bn.mulv.8S a_0, vec_gamma2, a_1
  bn.shv.8S  tmp, tmp << 1
  bn.subv.8S a_0, a, tmp
  bn.subv.8S tmp, vec_qhalf, a_0
  bn.shv.8S  tmp, tmp a >> 3
  bn.and     tmp, vec_q, tmp
  bn.subv.8S a_0, a_0, tmp

  bn.sid a_0_idx, 0(a_0_dst++)
  bn.sid a_1_idx, 0(a_1_dst++)
```

Listing 4.9: Vectorized Decompose on OTBN*.

```
loopi 32, 7
  bn.lid a_idx, 0(src++)

  /* (a + (1 << (D-1)) - 1) */
  bn.addv.8S a_1, vec_const, a
  /* a1 = (a + (1 << (D-1)) - 1) >> D */
  bn.shv.8S a_1, a_1 >> D
  /* a0 = (a1 << D) */
  bn.shv.8S a_0, a_1 << D
  /* a0 = a - (a1 << D) */
  bn.subv.8S a_0, a, a_0

  bn.sid a_0_idx, 0(a_0_dst++)
  bn.sid a_1_idx, 0(a_1_dst++)
```

Listing 4.10: Vectorized Power2Round on OTBN*.

5 Results

In this chapter, we present the benchmarking results of our implementation of Dilithium2 on OTBN* and compare them to the results of our baseline and other recent implementations on OTBN, as well as the performance on other platforms.

5.1 Testing on OTBN*

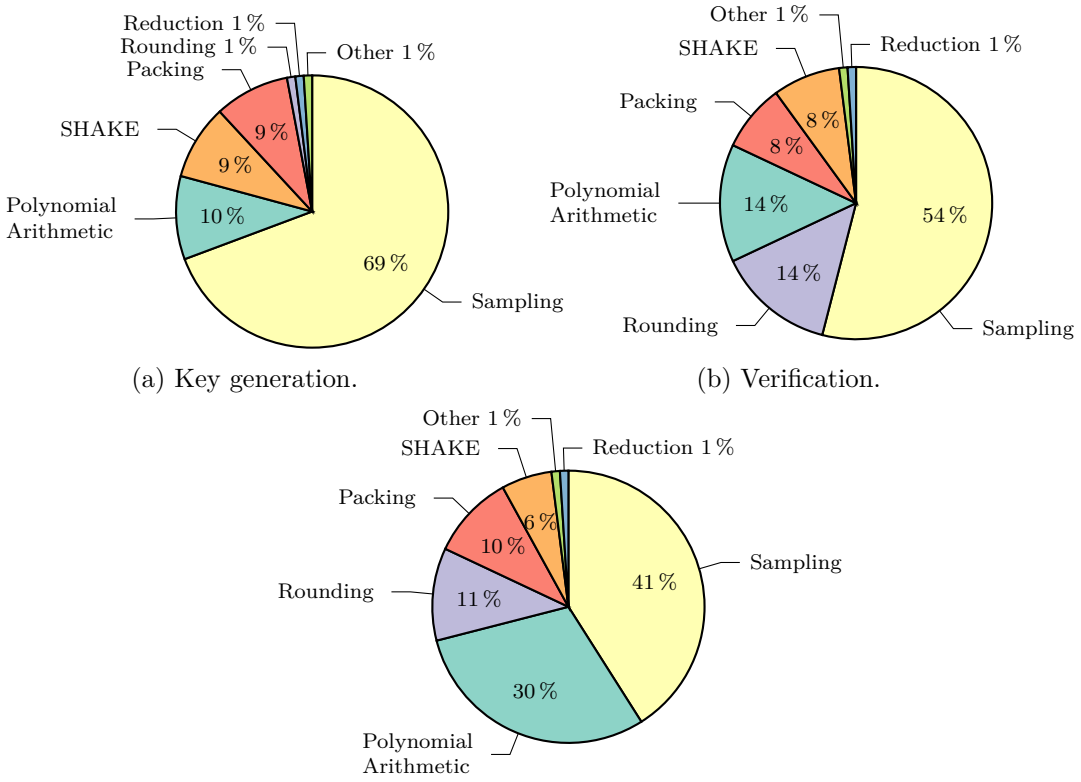
We use the exact same setup to test our implementation on OTBN*, as already used for the baseline implementation and described in Section 3.5.1. Just as for our base implementation, we run more than 10 000 iterations for key generation, signing, and verification, respectively, and check the results for correctness.

5.2 Performance on OTBN* & Comparison

In this section, we give the performance results for our implementation of Dilithium2 and its subroutines on OTBN*. Although comparing our results to other architectures' results is generally not straightforward, we still include performance results for several other implementations. The difficulty with this type of comparison is that the platforms' capabilities drastically vary, and thus, the cycle counts do not immediately allow conclusions about the implementations' quality. We include results from the reference implementation optimized for Intel AVX2 on a Skylake CPU from [LDK+22], from an optimized implementation on the Arm Cortex-A72 offering Neon vector extensions from [BHK+22], from an implementation for the Arm Cortex-M4 from [AHKS22], as well as two HW/SW co-design implementations from [KSFS23] on the PULPino platform and [ZXXH22] on the RocketCore platform. In addition to that, we consider the results from [Tur23] and [SOSK23] that also target OpenTitan and OTBN.

5.2.1 Profiling

Figure 5.1 shows that the amount of cycles spent on polynomial arithmetic has drastically reduced compared to Figure 3.6. We reduce the cycle count for polynomial arithmetic by 35, 38, and 39 percentage points for key generation, signing, and verification, respectively, compared to our baseline implementation on OTBN. We deem this observation as one indicator for the effectiveness of our extensions to the architecture. Tables 5.1 to 5.3 give a more detailed insight, presenting the accumulated cycle count of Dilithium's sub-routines. Basing on this data, we will provide more detailed analysis and comparison to other implementations in the following.



(a) Key generation.

(b) Verification.

(c) Signing.

Figure 5.1: Dilithium2 cycle count profiling on OTBN*, median values over 10 000 iterations. Groups with less than 1% not displayed. Percentages may not add up to 100% due to rounding.

Table 5.1: DILITHIUM2 key generation. Median number of instructions, stalls, total cycles, and number of calls for subroutines on OTBN*, 10 000 iterations.

Function	#Instructions	#Stalls	Total Cycles	#Calls
poly_uniform	45 671	13 983	59 654	16
poly_uniform_eta	22 259	7169	29 428	8
SHAKE	432	10 330	10 762	0
polyeta_pack_dilithium	4776	344	5120	8
poly_pointwise_acc_dilithium	2364	1548	3912	12
intt_dilithium	2808	692	3500	4
ntt_dilithium	2780	692	3472	4
polyt0_pack_dilithium	2756	316	3072	4
polyt1_pack_dilithium	2584	300	2884	4
poly_power2round_dilithium	932	392	1324	4
poly_caddq_dilithium	672	264	936	4
poly_pointwise_dilithium	532	388	920	4
poly_add_dilithium	532	388	920	4
keccak_send_message	496	249	745	50
key_pair_dilithium	533	140	673	1
main	16	3	19	1

Polynomial Multiplication

From Table 5.4, we can obtain a comparison of the cycle counts of the NTT, INTT, and pointwise multiplication. The per-function cycle counts for our implementations can be computed form Tables 3.2 to 3.4 and 5.1 to 5.3 by dividing the accumulated total runtime by the number of calls.

We can observe that the performance with our vector extensions is almost 10 times better than without on OTBN. We expected a result like this as we can process eight coefficients at once instead of one. An additional speed-up is incurred by the modular multiplication also completing in a single cycle, as opposed to four for our implementation of the improved Plantard multiplication in the baseline case. This effect is even more noticeable in the pointwise multiplication, where the modular multiplication using Plantard arithmetic takes five instead of four cycles.

Further, from Tables 3.2 to 3.4 and 5.1 to 5.3 we can see that the median accumulated runtime for the NTT, INTT, and pointwise multiplication (with accumulation) of our implementation on OTBN* in comparison to our baseline implementation, saves a total of 97 796, 646 893, and 143 759 cycles for key generation, signing, and verification, respectively. When compared to the baseline implementation, this alone causes a reduction of the median cycle counts for key generation, signing, and verification by 38%, 59%, and 46%, respectively.

The implementation by [Tur23], which is based on the Kronecker+ approach, is up to 42.23 times slower than our implementation on OTBN* and even slower than our

Table 5.2: Dilithium2 signing. Median number of instructions, stalls, total cycles, and number of calls for subroutines on OTBN*, 10 000 iterations.

Function	#Instructions	#Stalls	Total Cycles	#Calls
poly_uniform	45 671	13 983	59 654	16
poly_chknorm_dilithium	24 816	11 674	36 490	20
intt_dilithium	25 272	6228	31 500	36
ntt_dilithium	18 765	4671	23 436	27
poly_make_hint_dilithium	15 169	5030	20 199	4
SHAKE	617	15 565	16 182	0
poly_pointwise_acc_dilithium	7092	4644	11 736	36
poly_uniform_gamma1_dilithium	8916	1248	10 164	12
poly_pointwise_dilithium	4788	3492	8280	36
polyw1_pack_dilithium	7176	660	7836	12
decompose_dilithium	6912	384	7296	384
polyvec_encode_h_dilithium	4716	2116	6832	1
polyeta_unpack_dilithium	5112	368	5480	8
poly_challenge	2997	968	3965	3
poly_add_dilithium	2128	1552	3680	16
poly_decompose_dilithium	1932	1632	3564	12
polyz_pack_dilithium	2984	464	3448	4
polyt0_unpack_dilithium	2788	320	3108	4
sign_dilithium	2415	611	3026	1
poly_caddq_dilithium	2016	792	2808	12
poly_sub_dilithium	1056	776	1832	8
keccak_send_message	724	363	1087	69
main	19	4	23	1

baseline implementation on OTBN, which suggests that the Kronecker+ approach, as applied in [Tur23], is not favorable for the implementation of Dilithium on OTBN.

On the other hand, the performance of [SOSK23] is closer to what we achieve on OTBN*. While their NTT and INTT implementations do not use vectorization and should consequently grant us a speed-up of a factor of eight, they deploy a dedicated circuit for computing the butterfly operations in a single cycle, while this takes three cycles per eight coefficients on OTBN*.

Considering the implementation using Intel's AVX2 vector extension on a Skylake CPU, we can note even higher performance than on OTBN*, which we specifically extended to perform well for Dilithium's polynomial multiplication. The reason for this observation is the fact that Intel Skylake is a super-scalar architecture that allows performing multiple operations in parallel. For example, [Int23, Figure 2-8] shows that there are two execution units capable of vector multiplication, three for vector ALU operations, and a whole separate set of execution units for memory accesses. This, in combination with the

Table 5.3: DILITHIUM2 verification. Median number of instructions, stalls, total cycles, and number of calls for subroutines on OTBN*, 10 000 iterations.

Function	#Instructions	#Stalls	Total Cycles	#Calls
poly_uniform	45 671	13 983	59 654	16
poly_use_hint_dilithium	11 000	4930	15 930	4
SHAKE	393	8877	9270	0
ntt_dilithium	6255	1557	7812	9
poly_chknorm_dilithium	5200	2448	7648	4
poly_pointwise_acc_dilithium	2364	1548	3912	12
intt_dilithium	2808	692	3500	4
polyz_unpack_dilithium	2948	468	3416	4
polyt1_unpack_dilithium	2648	304	2952	4
polyw1_pack_dilithium	2392	220	2612	4
decompose_dilithium	2304	128	2432	128
poly_pointwise_dilithium	1064	776	1840	8
polyvec_decode_h_dilithium	1404	345	1749	1
verify_dilithium	1162	448	1610	1
poly_challenge	993	320	1313	1
poly_caddq_dilithium	672	264	936	4
poly_sub_dilithium	528	388	916	4
keccak_send_message	434	218	652	38
main	21	4	25	1

Out-of-Order (OoO) execution capabilities of the Intel platform, allows more than one instruction to retire in every clock cycle. OTBN was specifically designed to avoid these types of optimizations, among others, for security reasons, as outlined in Section 2.8.1.

When compared to the implementation on the Arm Cortex-A72 using Neon vector extensions, we see a speed-up for the NTT and INTT of a factor between 2.58 and 3.22. This result also matches our expectation: On Arm Neon, the width of the vector registers is half, meaning 128 bits, when compared to OTBN, which explains a factor of two. Additionally, one vectorized modular multiplication using Barrett multiplication from [BHK+22] takes three cycles instead of one for our `bn.mulvm`. Note that we give no number for the pointwise multiplication as [BHK+22] does not give results for individual pointwise multiplications but for a series of multiplications, additions, and a final reduction, and thus, is hardly comparable to other implementations.

As expected, the implementation on the Arm Cortex-M4 is slower by a factor greater than eight due the lack of vectorization [AHKS22]. In addition to this, the modular multiplication also takes three cycles, as opposed to one on OTBN*.

In contrast, the implementation from [ZXXH22] offers drastically higher performance than any of the other ones listed. They achieve this performance by using 32 dedicated butterfly units for their NTT and INTT computation, which at the same time comes

at a significant hardware cost [ZXXH22]. In [KSFS23], cycle counts for the polynomial multiplication-related functions are not explicitly stated.

Table 5.4: Benchmarks for polynomial multiplication related functions. All numbers given refer to cycles.

Platform	NTT		INTT		Pointwise	
OTBN* (This work)	868	(×1.00)	875	(×1.00)	230	(×1.00)
OTBN (This work)	8203	(×9.45)	8698	(×9.94)	2552	(×11.10)
OTBN [Tur23]	10 763	(×12.40)	13 943	(×15.93)	9714	(×42.23)
OTBN [SOSK23][a]	1972	(×2.27)	2244	(×2.56)	768	(×3.34)
Skylake [LDK+22]	532	(×0.61)	504	(×0.58)	99	(×0.43)
Cortex-A72 [BHK+22]	2241	(×2.58)	2821	(×3.22)	—	—
Cortex-M4 [AHKS22]	8093	(×9.32)	8415	(×9.62)	1955	(×8.50)
HW/SW [KSFS23]	—	—	—	—	—	—
HW/SW [ZXXH22]	32	(×0.04)	32	(×0.04)	32	(×0.14)

[a] Modified variant of OTBN.

In Section 4.2.1, we mention that it may be a strong assumption that the `bn.mulvm` instruction would execute in a single cycle. As we require four cycles for the multiplication with a constant using Plantard multiplication on OTBN, we will consider the impact on the NTT of `bn.mulvm` also taking four cycles to complete. One NTT requires a total of 1024 multiplications, which amounts to $\frac{1024}{8} = 128$ `bn.mulvm` instructions, increasing the cycle count of the NTT on OTBN* to 1252. Still, this implies a ×6.55 speed-up compared to our implementation on OTBN.

Rounding

Next, we consider the impact of our optimization to the rounding functions, where we provide vectorized implementations for the decomposition and Power2Round functions. For the decomposition during the signing, we get a speed-up factor of 8.16 saving 52 224 cycles, making up 4.75 % of the original total runtime. While we also get a speed-up of similar size for the vectorized Power2Round, the overall saving is almost negligible due to its minor impact on the runtime of the full scheme.

Sampling

Our optimization to the sampling of coefficients between $[-\eta, \eta]$ seems worthwhile, as we achieve a reduction of the runtime for this type of sampling in the key generation by 23.14 %, which amounts to 6.96 % for the whole key generation on OTBN*.

5.2.2 Full Scheme

The performance results for the full scheme, also in comparison to the data from Table 3.5, can be obtained from Table 5.5. With the hardware design for OTBN operating at 100 MHz, the median runtime for one key generation is 1.27 ms, for one signature 2.74 ms, and for one verification is 1.28 ms on OTBN*.

We can see that, compared to the implementation on OTBN, we achieve moderate speed-ups of 2.02× and 2.46× for key generation and verification, respectively. At the same time, the signing has a more considerable speed-up of a factor of 4.02. As we have seen in Section 5.2.1, the larger speed-up in the signature generation is due to it requiring far more polynomial multiplications than the key generation and verification.

The most closely related implementation is the one on OpenTitan published in [SOSK23], which uses both OpenTitan's Ibex core and a modified version of the OTBN core. In their implementation, the authors of [SOSK23] deploy a special ALU geared towards modular arithmetic, as well as instructions to accelerate the SHAKE computation. The latter allows for the computation of one round of Keccak in 40 cycles. This results in just below one million cycles for signature verification, about eight times as much as on OTBN*. We see three main arguments for this major difference compared to our work: First, the KMAC core that we interface to from OTBN takes only four cycles per Keccak round instead of 40. Second, our added instructions allow for better performance for the polynomial arithmetic functions, as detailed in Section 5.2.1. Moreover, we also use them to optimize other parts of the scheme, like the rounding, sampling, and bit-packing. Third, [SOSK23] relies on the Ibex core for unpacking the signature, the public key, and the hashing of the message to μ, incurring additional overhead for invoking the execution on OTBN and passing of data.

Interestingly, our implementation's performance on OTBN* is very close to the performance on Intel Skylake with AVX2 vector instructions. While both architectures offer equally sized wide registers, the reasons for the full scheme performing similarly are fundamentally different. When profiling the AVX2 implementation's Dilithium2 signing routine, we can note that about 50% of the runtime is spent on computing SHAKE, while on OTBN and OTBN*, the interface to the KMAC core drastically reduces the time spent on SHAKE. On the other hand, the AVX2 implementation offers faster polynomial arithmetic, superscalar execution, and a vectorized variant of the rejection sampling, enabled by rather complex vector instructions of the AVX2 vector extension. Benchmarks for a Kyber implementation in the Jasmin language from [ABB+23, Table 1] have shown that the vectorized rejection sampling dramatically improves Kyber's performance on an Intel CPU with AVX2.

While the performance on the Arm Cortex-A72 using the Neon vector extension is in the same order of magnitude as our implementation on OTBN*, it cannot match the performance due to lack of hardware acceleration for SHAKE, as well as worse performance for polynomial arithmetic, as analyzed in Section 5.2.1.

The reasons for the Arm Cortex-M4 performing much worse are mainly based on its lack of vector instructions and an accelerator for SHAKE.

The implementations from [KSFS23] and [ZXXH22] also use hardware acceleration for

the SHAKE computations and, thus, are more closely comparable to our architecture in this sense. While the data from Table 5.4 would have hinted that the implementation from [ZXXH22] would be vastly more efficient than our work on OTBN*, their speed-up regarding the whole scheme is far smaller than for the polynomial arithmetic alone. Compared to [KSFS23], our implementation on OTBN* is faster by a factor of 4.66–6.96, which we trace back to the vectorized approach in our implementation, compared to [KSFS23], using a generic butterfly unit.

Table 5.5: Full scheme benchmarks. All numbers given refer to cycles. Median result was selected, if given. 10 000 iterations for our measurements.

Operation	Platform	Key Gen.		Sign		Verify	
Mean	OTBN	257 891	(×2.02)	1 421 552	(×4.36)	315 588	(×2.46)
	OTBN*	127 356	(×1.00)	326 308	(×1.00)	128 196	(×1.00)
Median	OTBN*	127 353	(×1.00)	273 755	(×1.00)	128 195	(×1.00)
	OTBN	257 888	(×2.02)	1 100 386	(×4.02)	315 586	(×2.46)
	OpenTitan [SOSK23][a]	—	—	—	—	997 722	(×7.78)
	Skylake [LDK+22]	124 031	(×0.97)	259 172	(×0.95)	118 412	(×0.92)
	Cortex-A72 [BHK+22]	269 724	(×2.12)	649 230	(×2.37)	272 824	(×2.13)
	Cortex-M4 [AHKS22]	1 596 000	(×12.53)	4 093 000	(×14.95)	1 572 000	(×12.26)
	HW/SW [KSFS23]	593 403	(×4.66)	1 905 872	(×6.96)	651 217	(×5.08)
	HW/SW [ZXXH22]	45 800	(×0.36)	175 100	(×0.64)	89 800	(×0.70)
Std. Dev.	OTBN	100		951 060		151	
	OTBN*	102		161 306		151	

[a] Including modified variant of OTBN, parts of the execution on Ibex Core.

In the following, we want to make a similar argument as in Section 5.2.1 concerning the number of cycles `bn.mulvm` takes. From instruction histogram data, we learn that the median number of times `bn.mulvm` is executed in our implementation on OTBN* is 2112, 11 712, 2624 for the key generation, signing, and verification, respectively. Again, assuming four cycles per `bn.mulvm` instruction would result in a total median cycle count of 133 689, 308 891, 136 067 for the key generation, signing, and verification, respectively. We deem this increase of the cycle counts as modest; thus, the constraints on the actual hardware realization could be relaxed.

5.3 Security on OTBN*

As described in Section 3.5.3, we take great care to obtain a constant-time implementation of Dilithium that does not expose timing side-channel leakage. Again, we do not implement any kind of protection against power or EM-based side-channel attacks or even fault attacks that go beyond the protections offered by OTBN's hardware itself.

As we do not provide hardware implementations for our extensions to the architecture, we have no influence on the timing properties of the instructions and assume them to be

constant-time. However, we note that this is an important aspect that needs to be taken into account when constructing a hardware implementation for our proposal.

6 Conclusion & Outlook

In this thesis, we have studied the implementation of Dilithium on a slightly modified variant of the OTBN architecture and established a baseline for the cost in terms of cycle counts. We have carefully optimized the polynomial multiplication-related parts to get an impression of the instruction set's capabilities. In addition, we propose several extensions to the instruction set that improve the performance of lattice-based cryptography schemes. Applying these extensions to our baseline implementation of Dilithium allows for a speed-up of factor 9.45, 9.94, and 11.10 for the NTT, INTT, and pointwise multiplication, respectively. Regarding the full scheme, our extensions grant speed-ups of a factor of 2.02, 4.02, and 2.46 for key generation, signing, and verification, respectively, compared to our baseline implementation.

From our analysis in Section 5.2, it has become clear that the most beneficial additions to the instruction set are the instructions aiding the computation of the polynomial multiplication. In comparison, the value added by `bn.shv` concerning lattice-based cryptography specifically is relatively small, while it still may be useful for a broader range of applications.

Thus, we think adding our proposed instructions to OTBN is a worthwhile extension, greatly aiding with the computation of lattice-based cryptography.

Our implementation of polynomial arithmetic still has room for improvement in future work. In the baseline implementation, we can apply the better accumulation strategy for the matrix-vector product presented in [CHK+21]. This strategy allows the trade of modular multiplications in the pointwise multiplication for additional memory. We estimate a reduction of the cycle count for the computation of the matrix-vector product by $\approx 20\%$, while this technique will require 1024 B more memory.

An optimization that can be applied to both the baseline and the implementation using our extensions is the deployment of "small" NTTs for "small" products. As described in [AHKS22], the products of $c\boldsymbol{s}_1$ and $c\boldsymbol{s}_2$ are bounded by $\tau\eta$ in size, and thus, these multiplications can be regarded as computations in $\mathbb{Z}_{q'}$ for $q' > 2\tau\eta$ [CHK+21, Section 2.4.6]. This fact would, for example, allow setting q' to the Fermat number 257 or to Kyber's prime 3329, such that we can use the `.16H` variants of the vector instructions and thus compute twice the amount of data at once. For the baseline implementation, it could be explored whether implementing FNTs with $q' = 257$ yields a benefit.

The room for improvement especially extends to the memory requirements for our implementations. As the amount of memory on OTBN in its current state is very limited, applying optimizations in this direction is crucial. Thus, integrating approaches from [BRS22] could be considered.

Further, exploring more extensions to the instruction set could be of interest, especially regarding instructions allowing vectorizing the rejection sampling. This technique,

introduced in [GS16], is also applied in the reference implementation using Intel's AVX2 vector extension. Although the results are not exactly comparable, for an AVX2-optimized implementation of Kyber in [ABB+23], the vectorized sampling halved the runtime of the full scheme when compared to an AVX2-optimized implementation without this type of sampling.

In addition to that, it could be considered whether a more general form of a permutation instruction would fit our use case.

Moreover, analyzing the additional hardware resources required by our proposed extensions would be of great interest to improve comparability to related work and assess the feasibility in practice. This would further allow us to put the value of our proposed instructions in relation to their cost.

With Dilithium and Kyber sharing several structural similarities, it would be logical to consider an implementation of Kyber on OTBN. It could be explored whether additional or different extensions to the instruction set would be beneficial and if further challenges concerning the architecture arise.

In the long run, it would be desirable to have a masked implementation of Dilithium (and Kyber) on OTBN, especially considering the fact that the KMAC core already implements a masked Keccak operation.

Finally, it would be of great value to (re)implement the schemes in the Jasmin language [ABB+17]. We think that being able to reason about an implementation's safety, correctness, and security is crucial for cryptographic implementations, especially when targeting an RoT like OpenTitan.

Bibliography

[ABB+17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1807–1823, New York, NY, USA, October 2017. Association for Computing Machinery. https://acmccs.github.io/papers/p1807-almeidaA.pdf.

[ABB+23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber: Episode IV: Implementation correctness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):164–193, June 2023. https://tches.iacr.org/index.php/TCHES/article/view/10960.

[ABCG20] Erdem Alkim, Yusuf A. Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M} LWE schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 336–357, June 2020. https://tches.iacr.org/index.php/TCHES/article/view/8593.

[Abd21] Amin Abdulrahman. *Constant-Time NTTs for NTT-Unfriendly Rings on Cortex-M3.* Bachelor's Thesis, Ruhr-Universität Bochum, Bochum, June 2021.

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum Key Exchange—A New Hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, August 2016. USENIX Association. https://eprint.iacr.org/2015/1092.

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA Extensions for Finite Field Arithmetic: Accelerating Kyber and NewHope on RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 219–242, June 2020. https://tches.iacr.org/index.php/TCHES/article/view/8589.

[AHH+19] Martin R. Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based Schemes Using an RSA Co-Processor. *IACR Transactions on Cryptographic Hardware*

and Embedded Systems, pages 169–208, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7338.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security*, Lecture Notes in Computer Science, pages 853–871, Cham, 2022. Springer International Publishing. https://eprint.iacr.org/2022/112.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, July 1996. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/237814.237838.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, Lecture Notes in Computer Science, pages 311–323, Berlin, Heidelberg, 1987. Springer. https://link.springer.com/chapter/10.1007/3-540-47721-7_24.

[BGD+04] Johannes Buchmann, Luis Carlos Coronado García, Martin Döring, Daniela Engelbert, Christoph Ludwig, Raphael Overbeck, Arthur Schmidt, Ulrich Vollmer, and Ralf-Philipp Weinmann. Post-Quantum Signatures. *Cryptology ePrint Archive, Paper 2004/297*, 2004. http://eprint.iacr.org/2004/297.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and Higher-Order Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 173–214, August 2021. https://tches.iacr.org/index.php/TCHES/article/view/9064.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, January 2012. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/2090236.2090262.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. In Johannes Buchmann, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2019*, pages 209–228, Cham, 2019. Springer International Publishing. https://eprint.iacr.org/2019/489.

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. Dilithium for Memory Constrained Devices. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology - AFRICACRYPT 2022*, Lecture Notes in Computer Science, pages 217–235, Cham, 2022. Springer Nature Switzerland. https://eprint.iacr.org/2022/323.

[BRv22] Joppe W. Bos, Joost Renes, and Christine van Vredendaal. Post-Quantum cryptography with contemporary Co-Processors: Beyond kronecker, Schönhage-Strassen & nussbaumer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3683–3697, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/bos.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, February 2021. https://tches.iacr.org/index.php/TCHES/article/view/8791.

[CP11] William D. Casper and Stephen M. Papa. Root of Trust. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 1057–1060. Springer US, Boston, MA, 2011. https://doi.org/10.1007/978-1-4419-5906-5_789.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965. https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf.

[Deo18] Amit Deo. Module-LWE vs. Ring-LWE? http://malb.io/discrete-subgroup/slides/2018-01-15-deo.pdf, January 2018.

[Dil23] Dilithium. https://github.com/pq-crystals/dilithium/tree/3e9b9f1412f6c7435dbeb4e10692ea58f181ee51, November 2023.

[Ehr19] Stina Ehrensvard. The YubiKey as the WebAuthn Root of Trust. https://www.yubico.com/blog/the-yubikey-as-the-webauthn-root-of-trust/, April 2019. (accessed 2023-12-19).

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 414–460, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9303.

[FS87] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, Lecture Notes in Computer Science, pages 186–194, Berlin, Heidelberg, 1987. Springer. https://link.springer.com/content/pdf/10.1007/3-540-47721-7_12.pdf.

[FSS20] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 239–280, August 2020. https://tches.iacr.org/index.php/TCHES/article/view/8683.

[Gau66] Carl Friedrich Gauss. Theoria Interpolationis Methodo Nova Tractata. *Nachlass*, (3):265–330, 1866. https://gdz.sub.uni-goettingen.de/download/pdf/PPN235999628/PPN235999628.pdf.

[Gir91] Marc Girault. An identity-based identification scheme based on discrete logarithms modulo a composite number. In Ivan Bjerre Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, Lecture Notes in Computer Science, pages 481–486, Berlin, Heidelberg, 1991. Springer. https://link.springer.com/content/pdf/10.1007/3-540-46877-3_44.pdf.

[GJCJ23] Naina Gupta, Arpan Jati, Anupam Chattopadhyay, and Gautam Jha. Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(8):3234–3243, August 2023. https://ieeexplore.ieee.org/document/10129249.

[GKOS18] Tim Güneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of Lattice-Based Signature Schemes in Embedded Systems. *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 385–388, 2018. https://ieeexplore.ieee.org/document/8617969.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Amber Sprenkels. Compact Dilithium Implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, December 2020. https://tches.iacr.org/index.php/TCHES/article/view/8725.

[GS66] 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. https://doi.org/10.1145/1464291.1464352.

[GS16] Shay Gueron and Fabian Schlieker. Speeding up R-LWE Post-quantum Key Exchange. In Billy Bob Brumley and Juha Röning, editors, *Secure IT Systems*, Lecture Notes in Computer Science, pages 187–198, Cham, 2016. Springer International Publishing. https://eprint.iacr.org/2016/467.

[Har09] David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, October 2009. https://www.sciencedirect.com/science/article/pii/S0747717109001059.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved Plantard Arithmetic for Lattice-based Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 614–636, August 2022. https://tches.iacr.org/index.php/TCHES/article/view/9833.

[Int23] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, August 2023. https://cdrdv2.intel.com/v1/dl/getContent/671488.

[Jus11] Mike Just. Schnorr Identification Protocol. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 1083–1083. Springer US, Boston, MA, 2011. https://doi.org/10.1007/978-1-4419-5906-5_95.

[KA20] Emre Karabulut and Aydin Aysu. RANTT: A RISC-V Architecture Extension for the Number Theoretic Transform. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 26–32, August 2020. https://ieeexplore.ieee.org/document/9221619.

[Kan22] Matthias J. Kannwischer. *Polynomial Multiplication for Post-Quantum Cryptography*. PhD thesis, 2022. https://kannwischer.eu/thesis/.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019. https://ieeexplore.ieee.org/document/8835233.

[KL21] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security. CRC Press/Taylor & Francis Group, 3rd edition edition, 2021.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. https://www.ams.org/mcom/1987-48-177/S0025-5718-1987-0866109-5/.

[Kog19] Dima Kogan. Lecture 5: Proofs of Knowledge, Schnorr's protocol, NIZK. https://crypto.stanford.edu/cs355/19sp/lec5.pdf, 2019.

[Kro82] L. Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. (Abdruck einer Festschrift zu Herrn E. E. Kummers Doctor-Jubiläum, 10. September 1881.). *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882. https://eudml.org/doc/148487.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. Pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4. Technical report, 2019. https://eprint.iacr.org/2019/844.

[KSFS23] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-Quantum Signatures on RISC-V with Hardware Acceleration. *ACM Transactions on Embedded Computing Systems*, January 2023. https://dl.acm.org/doi/10.1145/3579092.

[Kyb23] Kyber. https://github.com/pq-crystals/kyber/tree/a621b8dde405cc507cbcfc5f794570a4f98d69cc, December 2023.

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Lepoint Tancrède, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[LHP+23] Douwei Lei, Debiao He, Cong Peng, Min Luo, Zhe Liu, and Xinyi Huang. Faster Implementation of Ideal Lattice-Based Cryptography Using AVX512. *ACM Transactions on Embedded Computing Systems*, 22(5):83:1–83:18, September 2023. https://dl.acm.org/doi/10.1145/3609223.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security*, pages 124–139, Cham, 2016. Springer International Publishing. https://eprint.iacr.org/2016/504.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In Henri Gilbert, editor, *Advances*

in Cryptology – EUROCRYPT 2010, Lecture Notes in Computer Science, pages 1–23, Berlin, Heidelberg, 2010. Springer. https://eprint.iacr.org/2012/230.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, June 2015. https://doi.org/10.1007/s10623-014-9938-4.

[LS19] Vadim Lyubashevsky and Gregor Seiler. NTTRU: Truly Fast NTRU Using NTT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):180–201, May 2019. https://tches.iacr.org/index.php/TCHES/article/view/8293.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, Lecture Notes in Computer Science, pages 598–616, Berlin, Heidelberg, 2009. Springer. https://www.iacr.org/archive/asiacrypt2009/59120596/59120596.pdf.

[MBB+23] Konstantina Miteloudi, Joppe W. Bos, Olivier Bronchain, Björn Fay, and Joost Renes. PQ.V.ALU.E: Post-Quantum RISC-V Custom ALU Extensions on Dilithium and Kyber. *Smart Card Research & Advanced Application Conference (CARDIS) (to appear)*, 2023. https://eprint.iacr.org/2023/1505.

[Mil86] Victor S. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, Lecture Notes in Computer Science, pages 417–426, Berlin, Heidelberg, 1986. Springer.

[MKvOV96] Alfred J. Menezes, Jonathan Katz, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985. https://www.ams.org/mcom/1985-44-170/S0025-5718-1985-0777282-X/.

[MP22] Michele Mosca and Marco Piani. Quantum Threat Timeline. Technical report, Global Risk Institute, December 2022. https://globalriskinstitute.org/mp-files/2022-quantum-threat-timeline-report-dec.pdf/.

[Nat] National Institute of Standards and Technology. Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography.

[NDMZ+21] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A RISC-V Post Quantum Cryptography Instruction Set Extension for Number Theoretic Transform to Speed-Up CRYSTALS Algorithms. *IEEE Access*, 9:150798–150808, 2021. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9605604.

[NG21] Duc Tri Nguyen and Kris Gaj. Optimized Software Implementations of CRYSTALS-Kyber, NTRU, and Saber Using NEON-Based Special Instructions of ARMv8. 2021. https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/nguyen-optimized-software-gmu-pqc2021.pdf.

[NIS22] NIST. Selected Algorithms 2022 - Post-Quantum Cryptography. https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022, 2022. (accessed 2023-12-17).

[NIS23a] NIST. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (Draft). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf, August 2023.

[NIS23b] NIST. FIPS 204: Module-Lattice-Based Digital Signature Standard (Draft). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.ipd.pdf, August 2023.

[NIS23c] NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard (Draft). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.ipd.pdf, August 2023.

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms*, volume 2 of *Springer Series in Information Sciences*. Springer-Verlag Berlin Heidelberg, 1982. https://www.springer.com/gp/book/9783540118251.

[Ope23] OpenTitan. lowRISC, November 2023. https://github.com/lowRISC/opentitan.

[Pei16] Chris Peikert. A decade of lattice cryptography. 10:283–424, 2016. http://dx.doi.org/10.1561/0400000074.

[Pla21] Thomas Plantard. Efficient Word Size Modular Arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, July 2021. https://thomas-plantard.github.io/pdf/Plantard21.pdf.

[Pol71] John M. Pollard. The fast Fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971. https://www.ams.org/journals/mcom/1971-25-114/S0025-5718-1971-0301966-0/S0025-5718-1971-0301966-0.pdf.

[Pöp16] Thomas Pöppelmann. *Efficient Implementation of Ideal Lattice-Based Cryptography*. Doctoralthesis, Ruhr-Universität Bochum, Universitätsbibliothek, 2016. https://hss-opus.ub.ruhr-uni-bochum.de/opus4/frontdoor/index/index/docId/4917.

[Pop23] Giacomo Pope. CRYSTALS-Dilithium Python Implementation. https://github.com/GiacomoPope/dilithium-py, November 2023.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, September 2009. https://dl.acm.org/doi/10.1145/1568318.1568324.

[Reg10] Oded Regev. The Learning with Errors Problem (Invited Survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204, June 2010. https://ieeexplore.ieee.org/document/5497885.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. https://dl.acm.org/doi/10.1145/359340.359342.

[Saa23a] Markku-Juhani O. Saarinen. Benchmarking RISC-V Post-Quantum Crypto. https://mjos.fi/doc/20231108-rvsummit-pqc.pdf, November 2023.

[Saa23b] Markku-Juhani O. Saarinen. RISC-V Cryptography and Hardware Security. https://mjos.fi/doc/20230629-rvi-tech-session.pdf, June 2023.

[Sch82] Arnold Schönhage. Asymptotically fast algorithms for the numerical muitiplication and division of polynomials with complex coefficients. In Jacques Calmet, editor, *Computer Algebra*, Lecture Notes in Computer Science, pages 3–15, Berlin, Heidelberg, 1982. Springer. https://link.springer.com/chapter/10.1007/3-540-11607-9_1.

[Sch90] Claus P. Schnorr. Efficient Identification and Signatures for Smart Cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, Lecture Notes in Computer Science, pages 239–252, New York, NY, 1990. Springer. https://link.springer.com/chapter/10.1007/0-387-34805-0_22.

[Sch91] Claus P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991. https://d-nb.info/1156214580/34.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. *Cryptology ePrint Archive, Paper 2018/039*, page 39, 2018. http://eprint.iacr.org/2018/039.

[Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. https://www.computer.org/csdl/pds/api/csdl/proceedings/download-article/12OmNqNXErh/pdf.

[SOSK23] Tobias Stelzer, Felix Oberhansl, Jonas Schupp, and Patrick Karl. Enabling Lattice-Based Post-Quantum Cryptography on the OpenTitan Platform. In *Proceedings of the 2023 Workshop on Attacks and Solutions in Hardware Security*, ASHES '23, pages 51–60, New York, NY, USA, November 2023. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3605769.3623993.

[Ste22] Tobias Stelzer. Post-Quantum Cryptography Arithmetic Logic Unit for the OpenTitan Big Number Accelerator. Master's thesis, Technical University of Munich, Munich, October 2022.

[Tea23a] OpenTitan Team. Datasheet - OpenTitan Documentation. https://opentitan.org/book/hw/top_earlgrey/doc/specification.html, 2023. (accessed 2023-12-10).

[Tea23b] OpenTitan Team. OpenTitan's RTL Freeze - Leveraging Transparency to Create Trustworthy Computing · lowRISC: Collaborative open silicon engineering. https://lowrisc.org/blog/2023/06/opentitans-rtl-freeze-leveraging-transparency-to-create-trustworthy-computing/, June 2023. (accessed 2023-11-13).

[Tit17] Titan in depth: Security in plaintext. https://cloud.google.com/blog/products/identity-security/titan-in-depth-security-in-plaintext?hl=en, August 2017. (accessed 2023-10-24).

[Tur23] Horia Turcuman. Speeding-up Post-Quantum Cryptography on an RSA Co-Processor. Master's thesis, Technical University of Munich, Munich, September 2023. https://github.com/horiaionut/kroneker-plus-on-otbn/blob/5576d7b035f5fe55a7199987ea05613d4aa913e7/paper.pdf.

[Win96] Franz Winkler. *Polynomial Algorithms in Computer Algebra*. Texts & Monographs in Symbolic Computation. Springer-Verlag Wien, 1996. https://www.springer.com/gp/book/9783211827598.

[Xin18] Xiaowen Xin. Titan M makes Pixel 3 our most secure phone yet. https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/, October 2018. (accessed 2023-12-19).

[ZHL+21] Zhen Zhou, Debiao He, Zhe Liu, Min Luo, and Kim-Kwang R. Choo. A Software/Hardware Co-Design of Crystals-Dilithium Signature Scheme.

ACM Transactions on Reconfigurable Technology and Systems, 14(2):11:1–11:21, June 2021. https://dl.acm.org/doi/10.1145/3447812.

[ZHS+22] Jieyu Zheng, Feng He, Shiyu Shen, Chenxi Xue, and Yunlei Zhao. Parallel Small Polynomial Multiplication for Dilithium: A Faster Design and Implementation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, pages 304–317, New York, NY, USA, December 2022. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3564625.3564629.

[ZXXH22] Yifan Zhao, Ruiqi Xie, Guozhu Xin, and Jun Han. A High-Performance Domain-Specific Processor With Matrix Extension of RISC-V for Module-LWE Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(7):2871–2884, July 2022. https://ieeexplore.ieee.org/document/9748063.

[ZZS+23] Jieyu Zheng, Haoliang Zhu, Zhenyu Song, Zheng Wang, and Yunlei Zhao. Optimized Vectorization Implementation of CRYSTALS-Dilithium. http://arxiv.org/abs/2306.01989, October 2023.

A Reproducing the Results

This chapter explains how to set up the software toolchain and reproduce the results from Chapter 5. Our recommended setup is based on Docker[1], and thus, in the following we assume Docker and Docker Compose to be installed. We provide all files related to this thesis in a repository on GitHub[2]. The repository contains the following:

- A submodule `opentitan`, which is a fork of the official OpenTitan repository[3] containing our modifications to the simulator, the instruction set, as well as our code for Dilithium, and the setup for tests and benchmarks. The code related to Dilithium can be found inside `./opentitan/sw/otbn/crypto/handwritten` as well as `./opentitan/sw/otbn/crypto/tests`.
- A `Dockerfile` that can be used to build the container for reproducing our results. In the build process, the aforementioned submodule is cloned into the container, and the relevant dependencies from the OpenTitan project are installed.
- A file `docker-compose.yml` that acts as a convenience wrapper, mounting a host directory `./dilithium_benchmarks` to a directory inside the container where the sqlite database holding the results will be stored.
- A directory `dilithium_benchmarks` that contains an exemplary database and a script `Evaluation.py` that can be used to extract the information from the sqlite database in a meaningful way.

Next, in Listing A.1, we will describe what commands need to be executed in order to acquire the source code and set up the container.

```
# Acquire the source code
git clone --recursive https://github.com/dop-amin/dilithium-on-opentitan-thesis.git
cd dilithium-on-opentitan-thesis
# Optionally: Checkout the commit as of writing this
git checkout 87eeffd7e7fff7a2c145411034b29737d7a909d9
# Build the image
docker build -t dilithium-on-opentitan-image .
# Start the container
docker compose up -d dilithium-on-opentitan
# Attach to the container
docker attach dilithium-on-opentitan
```

Listing A.1: Instructions for setup of the test environment.

[1] https://www.docker.com/
[2] https://github.com/dop-amin/dilithium-on-opentitan-thesis.git
[3] https://github.com/lowRISC/opentitan

Once the steps from Listing A.1 have been completed, a shell inside the container should have been spawned inside the directory `opentitan`.

The tests can be run and evaluated as described in Listing A.2. For the Python script, option `-f` determines the database file to evaluate, `-i` defines the IDs from the database to evaluate, and `-o` optionally the output file to write the result to. As this directory is mapped from the container to the host, the results will remain accessible after quitting the container. The database entity-relationship scheme can be obtained from Figure A.1.

```
# Inside the container's shell
# Run a first test, may take some time as bazel sets itself up in this step
./bazelisk.sh test --cache_test_results=no --sandbox_writable_path="/home/ubuntu/
    dilithium_benchmarks" //sw/otbn/crypto/tests:dilithium_key_pair_bench
cd ../dilithium_benchmarks
# Run the evaluation script
python3 Evaluation.py -i 1 2 3 4 5 6 -f dilithium_bench_example.db -o my_result.txt

```

Listing A.2: Instructions for reproduction of our results.

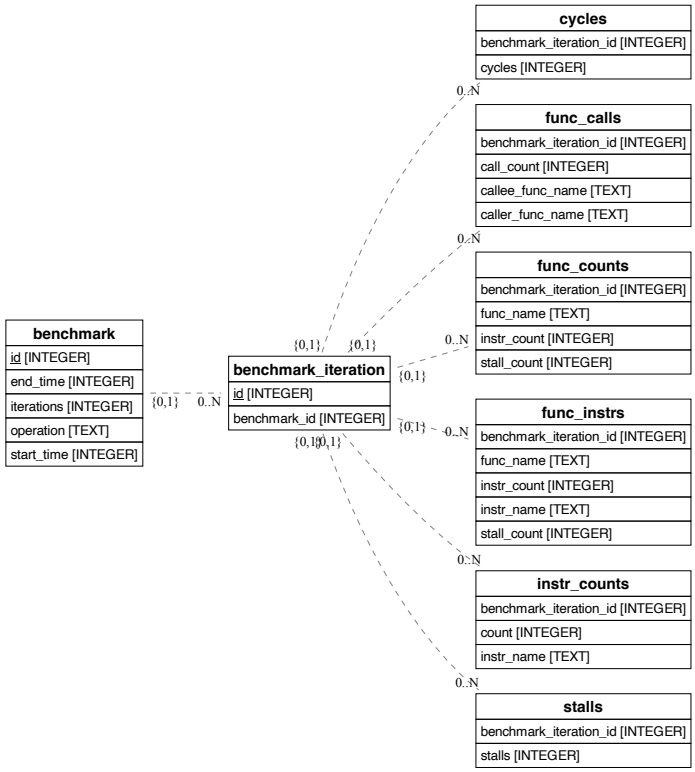


Figure A.1: Entity-relationship diagram for the database storing benchmark results.

We denote the tests using `//sw/otbn/crypto/tests:dilithium_{key_pair,sign,verify}{,_base}_bench`, meaning there are separate tests for the key gener-

ation, signing, and verification. Appending `_base` to the operation selects the test that does not make use of our modifications to the instruction set.

By default, the number of tests to be run is set to two, however, this can be varied in the file `./sw/otbn/crypto/tests/dilithiumpy_bench_otbn/bench_dilithium.py` by setting `ITERATIONS` to the number of desired iterations. `NPROC` can be used to define how many processes will work on the given number of iterations in parallel. Note that it may be required to increase the timeout of the test using `--test_timeout` for a large number of iterations.

When using Docker is not desired, a local setup is also possible by following the instructions by the OpenTitan Team[4] and replicating the additional steps from the Dockerfile manually. Additionally, the option `--sandbox_writable_path` needs to be set matching `DATABASE_PATH` in `./sw/otbn/crypto/tests/dilithiumpy_bench_otbn/bench_dilithium.py` such that the result from the sandbox can be written to the host file system. The script `Evaluation.py` requires the package `tabulate` to print the tables.

[4] `https://opentitan.org/book/doc/getting_started/index.html`

B Additional Data

Table B.1: Mapping of functions to groups for the profiling.

Group	Functions
Poly. Arith.	intt_base_dilithium, ntt_base_dilithium, poly_pointwise_acc_base_dilithium, poly_pointwise_base_dilithium, poly_caddq_base_dilithium, poly_sub_base_dilithium, poly_add_base_dilithium, poly_add_pseudovec_base_dilithium, poly_pointwise_acc_dilithium, intt_dilithium, ntt_dilithium, poly_pointwise_dilithium, poly_add_dilithium, poly_sub_dilithium
Reduction	poly_reduce32_pos_dilithium, poly_reduce32_dilithium, poly_reduce32_short_dilithium, decompose_dilithium
Sampling	poly_uniform_base_dilithium, poly_uniform_gamma1_base_dilithium, poly_challenge, poly_chknorm_base_dilithium, poly_uniform_eta_base_dilithium, poly_uniform, poly_uniform_eta, poly_chknorm_dilithium, poly_uniform_gamma1_dilithium
Rounding	decompose_base_dilithium, poly_make_hint_dilithium, poly_decompose_dilithium, poly_use_hint_dilithium, poly_power2round_base_dilithium, poly_power2round_dilithium
Packing	polyw1_pack_dilithium, polyeta_unpack_base_dilithium, polyvec_encode_h_dilithium, polyz_pack_base_dilithium, polyt0_unpack_base_dilithium, polyz_unpack_base_dilithium, polyt1_unpack_dilithium, polyvec_decode_h_dilithium, polyeta_pack_dilithium, polyt0_pack_base_dilithium, polyt1_pack_dilithium, polyt0_pack_dilithium, polyz_unpack_dilithium, polyeta_unpack_dilithium, polyz_pack_dilithium, polyt0_unpack_dilithium
SHAKE	SHAKE, keccak_send_message
Other	main, sign_base_dilithium, verify_base_dilithium, key_pair_base_dilithium, key_pair_dilithium, verify_dilithium, sign_dilithium