Universidade Estadual de Campinas
Instituto de Computação

Everaldo Antonio Moreira Alves

Implementação em Software dos Algoritmos Pós-Quânticos Kyber e Dilithium na Plataforma ARMv8

CAMPINAS
2024

Everaldo Antonio Moreira Alves

Implementação em Software dos Algoritmos Pós-Quânticos Kyber e Dilithium na Plataforma ARMv8

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Julio César López Hernández

Este exemplar corresponde à versão final da Dissertação defendida por Everaldo Antonio Moreira Alves e orientada pelo Prof. Dr. Julio César López Hernández.

CAMPINAS
2024

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Silvania Renata de Jesus Ribeiro - CRB 8/6592

AL87i

Alves, Everaldo Antonio Moreira, 1979-
Implementação em Software dos Algoritmos Pós-Quânticos Kyber e Dilithium na Plataforma ARMv8 / Everaldo Antonio Moreira Alves. – Campinas, SP : [s.n.], 2024.

Orientador(es): Julio César López Hérnandez.
Dissertação (mestrado) – Universidade Estadual de Campinas (UNICAMP), Instituto de Computação.

1. Criptografia pós-quântica. 2. Crystals-kyber (Algoritmos criptográficos). 3. Crystals-Dilithium (Algoritmos criptográficos). I. López Hérnandez, Julio César, 1961-. II. Universidade Estadual de Campinas (UNICAMP). Instituto de Computação. III. Título.

Informações complementares

Título em outro idioma: Software implementation of the Post-Quantum Kyber and Dilithium algorithms on the ARMv8 platform
Palavras-chave em inglês:
Post-quantum cryptography
Crystals-kyber (Cryptographic algorithms)
Crystals-Dilithium (Cryptographic algorithms)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Julio César López Hérnandez [Orientador]
Routo Terada
Hilder Vitor Lima Pereira
Data de defesa: 16-12-2024
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0009-0008-8510-0719
- Currículo Lattes do autor: http://lattes.cnpq.br/4518004370626916

Universidade Estadual de Campinas
Instituto de Computação

Everaldo Antonio Moreira Alves

Implementação em Software dos Algoritmos Pós-Quânticos Kyber e Dilithium na Plataforma ARMv8

Banca Examinadora:

- Prof. Dr. Julio César López Hérnandez
  IC/UNICAMP
- Prof. Dr. Hilder Vitor Lima Pereira
  IC/UNICAMP
- Prof. Dr. Routo Terada
  IME/USP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 16 de dezembro de 2024

Agradecimentos

Inicialmente, agradeço a Deus pela sua graça e bondade que me conduziram até aqui.

À minha esposa pelo suporte incondicional e pela tolerância a minha ausência ao longo desta jornada.

Aos meus pais e irmãos, por todo o suporte e incentivo que me concederam em todas as etapas da minha vida e também durante a elaboração deste trabalho.

Ao professor Dr. Julio César López Hernández pela orientação precisa e pela disponibilidade ofertada durante todo o período de elaboração deste trabalho.

Ao Capitão-de-Fragata Vilc Queup Ruffino pelo apoio e pela orientação segura desde o processo para ingresso na UNICAMP até a conclusão do curso.

Ao doutorando Décio Luiz Gazzoni Filho, que generosamente compartilhou, de maneira solícita e sempre efetiva, sua experiência na área de criptografia, em várias ocasiões ao longo do curso.

Resumo

O rápido avanço da computação quântica impõe desafios significativos à criptografia clássica, uma vez que as capacidades emergentes põem em risco os fundamentos de segurança nos quais se baseiam as comunicações digitais. Esta crescente ameaça estimulou a evolução dos padrões criptográficos, levando ao surgimento da criptografia pós-quântica. Neste contexto, o Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST) desempenhou um papel fundamental para estabelecer uma estrutura criptográfica segura para o futuro, especificando esquemas resilientes contra ameaças quânticas a partir de um processo de seleção de algoritmos. Dentre os algoritmos selecionados, CRYSTALS-Kyber compõe o padrão como mecanismo de encapsulamento de chaves e CRYSTALS-Dilithium como esquema de assinatura digital.

Esta dissertação concentra-se na implementação em software destes algoritmos na plataforma ARMv8, uma arquitetura largamente empregada em dispositivos de computação contemporâneos. Embora trabalhos anteriores tenham explorado implementações desses algoritmos com base na documentação disponibilizada pelos autores, a recente publicação das especificações finais pelo NIST acarreta a necessidade de assegurar a conformidade com os requisitos de segurança e eficiência atualmente estabelecidos.

Nesse sentido, ao explorar o potencial da arquitetura ARMv8 mantendo o alinhamento às diretrizes mais recentes do NIST, este trabalho contribui para a área ao fornecer uma implementação otimizada para uma plataforma de computação moderna, estabelecendo uma referência para futuras pesquisas e aplicações práticas em comunicações digitais seguras. Os resultados obtidos indicam acelerações de até 2.82x, 2.56x e 2.40x para geração de chaves, encapsulamento e desencapsulamento no Kyber, respectivamente, e de 2,56x, 2,67x e 2,29x para geração de chaves, assinatura e verificação no Dilithium, em comparação com a implementação de referência disponibilizada pelos autores dos respectivos esquemas.

Abstract

The rapid advancement of quantum computing poses significant challenges to classical cryptography, as emerging capabilities threaten the security foundations on which digital communications rely. This growing threat has spurred the evolution of cryptographic standards, leading to the emergence of post-quantum cryptography. In this context, the United States National Institute of Standards and Technology (NIST) has played a fundamental role in establishing a secure cryptographic framework for the future by specifying schemes resilient to quantum threats through an algorithm selection process. Among the selected algorithms, CRYSTALS-Kyber forms the standard as a key encapsulation mechanism, and CRYSTALS-Dilithium as a digital signature scheme.

This dissertation focuses on the software implementation of these algorithms on the ARMv8 platform, a widely used architecture in contemporary computing devices. Although previous works have explored implementations of these algorithms based on documentation provided by the authors, the recent publication of the final specifications by NIST necessitates ensuring compliance with currently established security and efficiency requirements.

In this regard, by exploring the potential of the ARMv8 architecture while aligning with the latest NIST guidelines, this work contributes to the field by providing an optimized implementation for a modern computing platform, establishing a reference for future research and practical applications in secure digital communications. The results obtained indicate accelerations of up to 2.82x, 2.56x, and 2.49x for key generation, encapsulation, and decapsulation in Kyber, respectively, and 2.56x, 2.67x, and 2.29x for key generation, signing, and verification in Dilithium, compared to the reference implementation provided by the respective scheme authors.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A evolução da tecnologia digital tem elevado a criptografia a um papel central na segurança de sistemas informatizados. Aplicações que envolvem comunicações seguras, transações financeiras e armazenamento de dados sensíveis, por exemplo, dependem fundamentalmente de primitivas criptográficas.

Dividida em dois campos, a criptografia moderna pode ser classificada como simétrica ou assimétrica. A criptografia simétrica utiliza a mesma chave para cifrar e decifrar uma mensagem, sendo exemplos populares o AES (*Advanced Encryption Standard*) e o DES (*Data Encryption Standard*). Por outro lado, a criptografia assimétrica, também conhecida como criptografia de chave pública, utiliza um par de chaves, uma pública e outra privada para prover segurança à comunicação digital. As chaves são relacionadas matematicamente, mas os parâmetros são definidos de maneira a inviabilizar o cálculo da chave privada a partir da pública.

No campo da criptografia assimétrica, dois componentes são especialmente importantes para a manutenção dos pilares da segurança: mecanismos de estabelecimento de chaves e esquemas de assinaturas digitais. Um mecanismo de estabelecimento de chaves permite que duas partes interajam para definir uma chave simétrica compartilhada em um canal de comunicação público. Por outro lado, as assinaturas digitais fornecem um meio de assegurar a autenticidade e integridade de comunicações e documentos em canais inseguros. Essas assinaturas são realizadas por meio de solução de problemas computacionalmente desafiadores, utilizando a chave privada, de forma que apenas o detentor dessa chave pode criar uma assinatura válida e que qualquer parte possa verificá-la usando a chave pública correspondente.

A robustez dos algoritmos assimétricos clássicos reside na complexidade computacional de problemas como a fatoração de inteiros grandes [45] e o cálculo de logaritmos discretos [13], tidos até recentemente como intratáveis. No entanto, o avanço da computação quântica e, particularmente a publicação do algoritmo de Shor [48], revelam que esses fundamentos são frágeis diante de computadores quânticos robustos, tornando possível resolvê-los em tempo polinomial. A possibilidade de comprometer tais cifras coloca em xeque a segurança de infraestruturas críticas globalmente adotadas, catalisando a necessidade urgente de novas abordagens criptográficas.

Essa demanda conduziu ao desenvolvimento da criptografia pós-quântica, um campo emergente dedicado a explorar novos problemas computacionais que se mostram resisten-

tes contra as capacidades de computação quântica. Este novo domínio busca estabelecer algoritmos que possam ser implementados de forma segura na presença de adversários quânticos, garantindo a integridade da infraestrutura de segurança digital para as futuras gerações.

Nesse contexto, o Instituto Nacional de Padrões e Tecnologia dos Estados Unidos (NIST) lançou, em dezembro de 2016, uma iniciativa para padronizar algoritmos pós-quânticos. O concurso foi planejado para ser executado em várias rodadas, cada uma com o objetivo de filtrar e aprofundar a análise dos algoritmos propostos. A resposta foi considerável, com 69 candidatos submetidos inicialmente [33], abrangendo diversas classes de problemas matemáticos, tais como reticulados, códigos corretores de erro, sistemas multivariados e funções de resumo criptográfico. Todavia, destacando-se uma quantidade expressiva de esquemas baseados em reticulados, dentre estes o CRYSTALS[1]-Kyber e o CRYSTALS-Dilithium, o que sublinhava a confiança da comunidade científica na robustez desses problemas contra ataques quânticos.

Após avaliações rigorosas que consideraram segurança, eficiência e aplicabilidade prática, o número de candidatos foi reduzido. Na segunda rodada, 26 algoritmos continuaram na disputa [33], com foco em testes mais detalhados e ajustes com base nos comentários do NIST e da comunidade global. A terceira rodada trouxe ainda mais consolidação, com apenas 15 algoritmos avançando, indicando uma seleção cada vez mais criteriosa baseada em análises de robustez e performance.

O ano 2023 marcou de forma significativa o processo com a seleção de três algoritmos sob os novos Padrões Federais de Processamento de Informações (FIPS): Kyber foi padronizado sob o FIPS 203 como um mecanismo de encapsulamento de chaves[2] [31], enquanto Dilithium e SPHINCS+ foram escolhidos como esquemas de assinaturas digitais [36] [37] sob os FIPS 204 e 205, respectivamente. Além desses, foi selecionado o FALCON [17], um esquema para assinaturas digitais também baseado em reticulados cujo documento padronizador está em elaboração.

A padronização do SPHINCS+ sob o FIPS 205, caracteriza uma escolha motivada pela necessidade de diversificar os fundamentos matemáticas dos algoritmos pós-quânticos. SPHINCS+, sendo baseado em funções de resumo criptográfico, oferece uma alternativa segura no caso de as premissas matemáticas relacionadas aos reticulados serem eventualmente refutadas.

Dessa forma, até então, três esquemas de assinaturas digitais, se destacaram no processo de padronização cada um deles com importantes particularidades:

- Dilithium, conhecido por sua implementação e compreensão simplificadas, possui tempos de geração de chaves excepcionalmente rápidos, tornando-o ideal para aplicações que exigem autenticação rápida e eficiente. Apesar de suas vantagens, Dilithium pode ter tamanhos de assinatura ligeiramente maiores que o FALCON, o que pode gerar impactos em cenários de comunicação restrita.

- FALCON, projetado para minimizar o custo de comunicação através de assinatu-

[1]CRYSTALS – Cryptographic Suite for Algebraic Lattices – um pacote submetido ao esforço de padronização pós-quântica do NIST.

[2]Um tipo particular de mecanismo de estabelecimento de chaves.

ras extremamente compactas, o que é essencial em ambientes como servidores de certificados e aplicações que demandam intensa troca de dados. No entanto, essa eficiência é acompanhada por uma complexidade aumentada e tempos de geração de chaves mais longos, o que pode ser uma desvantagem em dispositivos com recursos limitados.

- SPHINCS+, utilizando funções de resumo criptográfico para segurança, oferece uma abordagem robusta que é independente dos problemas específicos associados a reticulados. No entanto, o esquema tende a apresentar tamanhos de assinatura significativamente maiores, o que pode ser desafiador em contextos que apresentem restrições associadas à largura de banda ou ao armazenamento.

As tabelas a seguir proporcionam uma visão comparativa desses esquemas em termos de tamanho de chave pública e assinatura (em bytes). Os dados correspondem às categorias de segurança 2 e 5 para as quais os três esquemas possuem variantes.

Esquema	**Chave Pública**	**Assinatura**
CRYSTALS-Dilithium	1312	2420
FALCON	897	690
SPHINCS+	32	17088

Tabela 1.1: Tamanho da chave pública e da assinatura (em bytes) para a categoria de segurança 2.

Esquema	**Chave Pública**	**Assinatura**
CRYSTALS-Dilithium	2592	4595
FALCON	1793	1280
SPHINCS+	64	49856

Tabela 1.2: Tamanho da chave pública e da assinatura (em bytes) para a categoria de segurança 5.

Cabe ressaltar que, dadas as incertezas envolvidas no processo de estimar a robustez dos criptossistemas pós-quânticos, atribuíveis principalmente à possibilidade de emergência de novos ataques criptoanalíticos, bem como à capacidade restrita de antecipar características operacionais de futuros computadores quânticos, o NIST adotou uma estratégia, delineada em sua chamada de propostas [30], que consiste na definição de cinco categorias de segurança equivalentes à dificuldade de comprometer determinadas primitivas criptográficas clássicas. As categorias são baseadas na resistência comparável a ataques contra cifras de bloco e funções de resumo criptográfico específicas: as categorias 1,3 e 5 estão relacionadas à resistência comparável a criptoanálise de força bruta para uma cifra de bloco com chave de 128, 192 e 256 bits respectivamente. Por outro lado, as categorias 2 e 4 correspondem às exigências de segurança de buscas de colisão para funções de resumo criptográfico de 256 e 384 bits respectivamente.

A despeito da continuidade do processo de padronização, em sua quarta e última rodada prevista para encerrar em dezembro de 2024, CRYSTALS-Kyber e CRYSTALS-Dilithium foram selecionados como algoritmos primários por ambos apresentarem forte

segurança, excelente desempenho e pela expectativa de que atendam adequadamente a maioria das aplicações [35].

1.1 Contribuições

Esta dissertação apresenta uma implementação otimizada dos algoritmos Kyber e Dilithium para plataforma ARMv8, uma arquitetura que se destaca tanto em eficiência energética quanto em capacidade de processamento. As otimizações foram projetadas para tirar proveito dos recursos avançados da arquitetura.

As técnicas e recursos utilizados permitiram alcançar uma melhoria de performance de até 2.57x em relação ao código de referência para o processo de encapsulamento do Kyber.

Adicionalmente, foram exploradas melhorias no processo de assinatura do Dilithium, onde se atingiu um aumento de desempenho de até 2.67x em comparação com a implementação de referência.

As implementações descritas nesta dissertação foram norteadas pelas especificações mais recentes do NIST para os algoritmos criptográficos pós-quânticos primários. Dessa forma, o código pode servir como referência para novas implementações na mesma arquitetura.

Além disso, as discussões sobre os aspectos de implementação dos esquemas, nas quais foram identificados os processos de maior custo computacional e as estratégias de mitigação aplicadas, pavimentam o caminho para pesquisas futuras voltadas para otimização dos algoritmos na plataforma ARMv8.

1.2 Trabalhos Relacionados

Os algoritmos Kyber e Dilithium, que compõe o projeto CRYSTALS, já foram implementados em várias plataformas, tais como x86, RISC-V e ARM. No entanto, dada a presença massiva de dispositivos baseados em ARM em segmentos críticos como dispositivos móveis, servidores e sistemas embarcados, a exploração de otimizações especializadas nesta arquitetura se torna cada vez mais relevante.

Nesse sentido, Ortiz et al. [39], tendo como alvo a arquitetura ARMv8-A, propuseram uma otimização do Kyber na qual alcançaram aceleração de até 1.28x para o processo de geração de chaves, 1.34x para o encapsulamento e 1.32x para o desencapsulamento em relação ao código de referência. Os autores utilizaram instruções NEON para acelerar a distribuição binomial centrada e a multiplicação de polinômios.

Sanal et al. [46], ao utilizar instruções vetoriais para NTT, multiplicação, adição e subtração polinomial, atingiram acelerações de até 1.72×, 1.88× e 2.29× para geração de chaves, encapsulamento e desencapsulamento do Kyber respectivamente, tendo com alvo a arquitetura ARMv8-A.

Já Nguyen e Gaj [32] apresentaram uma implementação otimizada do Kyber para ARMv8 na qual alcançaram melhorias de até 2.45x para o encapsulamento e de até 3.04x

no desencapsulamento em relação ao código de referência. Foram utilizadas instruções NEON e minimizadas as chamadas à redução de Barret na inversão da NTT.

Em 2022, Smith e Jones [51] apresentaram melhorias na implementação do Kyber em dispositivos móveis utilizando o conjunto de instruções SIMD NEON. Este estudo mostrou ganhos de desempenho de até 2.16x em relação à implementação de referência. Para alcançar esses resultados, os autores reestruturaram o código-fonte do Kyber para explorar de maneira mais eficiente as operações paralelas proporcionadas pelo NEON.

No mesmo ano, Kim et al. [25] propuseram uma implementação do Dilithium para ARMv8 na qual se concentraram principalmente em otimizar os cálculos que envolvem a NTT, atingindo melhorias de performance de até 43.83% no processo de geração de chaves, 113.25% no processo de assinatura e 41.92% no processo de verificação em relação ao código de referência.

Por fim, Becker et al. [5], ao empregarem técnicas particulares para redução de Barret, NTT e multiplicações polinomiais implementadas com instruções vetoriais, apresentaram ganhos de até 3.05x para geração de chaves, 3.58x para assinatura e 3.30x para verificação do Dilithium em relação ao código de referência, tendo como arquitetura alvo ARMv8-A.

1.3 Organização desta Dissertação

No Capítulo 2 são estabelecidos os elementos fundamentais necessários para o entendimento do conteúdo dos capítulos subsequentes. Isso inclui a notação utilizada em todo o documento, o embasamento matemático necessário à compreensão dos algoritmos criptográficos discutidos e os conceitos básicos de segurança.

Os Capítulos 3 e 4 são dedicados à exploração do mecanismo de encapsulamento de chaves CRYSTALS-Kyber e do esquema de assinatura digital CRYSTALS-Dilithium respectivamente. São detalhados os algoritmos empregados e as particularidades dos esquemas.

O Capítulo 5 é dedicado a apresentação das técnicas de otimização aplicadas a ambos os esquemas criptográficos e o Capítulo 6 à apresentação dos resultados alcançados.

Finalmente, o Capítulo 7 conclui a dissertação com um resumo das contribuições e direções de potenciais pesquisa futuras.

Capítulo 2

Conceitos Elementares

2.1 Fundamentos Matemáticos

Notações. R_q representa o anel $\mathbb{Z}_q[X]/(X^n+1)$. Esse anel é composto por polinômios da forma $f(X) = f_0 + f_1X + \cdots + f_{n-1}X^{n-1}$, onde $f_j \in \mathbb{Z}_q$ para todo j, equipado com adição e multiplicação módulo $X^n + 1$.

A imagem de R_q sob a Transformada da Teoria dos Números (NTT) é representada por Tq. Seus elementos são chamados de representações NTT de polinômios em R_q. A operação de multiplicação em Tq é denotada pelo símbolo $\circ$.

Um elemento $v \in (\mathbb{Z}_q^{256})^3$ será um vetor de comprimento três cujas entradas $v[0]$, $v[1]$ e $v[2]$ são elas próprias elementos de $\mathbb{Z}_q^{256}$ (ou seja, vetores). Pode-se pensar que cada uma dessas entradas representa um polinômio em R_q, de modo que v em si representa um elemento do módulo R_q^3.

Para representar matrizes e vetores cujas entradas são elementos de R_q, serão utilizadas letras em negrito (por exemplo, $\mathbf{v}$ para vetores e $\mathbf{A}$ para matrizes). Para representar matrizes e vetores cujas entradas são elementos de T_q, será empregado um "chapéu"(por exemplo, $\hat{\boldsymbol{v}}$ e $\hat{\boldsymbol{A}}$).

A norma euclidiana do vetor u é denotada por $\|\mathbf{u}\|$. O termo $\mod \pm q$ indica que os números são reduzidos de modo que os resultados sejam simetricamente distribuídos em torno de zero. A operação é realizada de tal forma que o resultado fique dentro do intervalo $[-q/2, q/2]$ se q for par, ou $[-\frac{q-1}{2}, \frac{q-1}{2}]$ se q for ímpar.

A implementação de $mod \pm q$ normalmente envolve uma etapa de redução modular tradicional seguida por uma condição para ajustar valores que caem fora do intervalo centrado, semelhante aos ajustes que discutiremos para a função de redução de Barrett.

2.1.1 Reticulados

A estrutura algébrica denominada reticulados, juntamente com os problemas computacionais difíceis baseados neles, são essenciais para o entendimento dos algoritmos tratados nesta dissertação. Dessa forma, esta seção oferece definições formais e as propriedades fundamentais dessa estrutura, preparando o alicerce para as discussões dos capítulos subsequentes.

Definição 1 (Reticulado)**.**
Um reticulado Λ é definido a partir de um conjunto B de n vetores linearmente independentes no espaço $\mathbb{R}^n$, expressos como:

$$B = \{b_1, b_2, \ldots, b_n\}, \quad b_i \in \mathbb{R}^n.$$

O reticulado Λ consiste no conjunto de todas as combinações lineares inteiras desses vetores. O conjunto B é conhecido como a base do reticulado, e pode ser representado por uma matriz $m \times n$ que contém os vetores de B como colunas. Formalmente, o reticulado Λ pode ser descrito, em termos da definição matricial de sua base B, como:

$$\Lambda = \{Bz : z \in \mathbb{Z}^n\}.$$

Os reticulados desempenham um papel importante na criptografia moderna, especialmente em esquemas projetados para oferecer segurança contra potenciais computadores quânticos. Essa estrutura atraiu a atenção devido ao fato de que os projetos para construções criptográficas eram acompanhados por provas de segurança baseadas em instâncias de pior caso [11], o que fundamenta de forma robusta sua aplicação prática.

2.1.2 Problemas Difíceis em Reticulados

Problemas difíceis são o alicerce sobre o qual a segurança dos esquemas criptográficos é construída, pois garantem que certas operações, embora factíveis para um usuário legítimo, sejam proibitivamente custosas sob o ponto de vista de um adversário.

Dentre os problemas difíceis baseados em reticulados, destacam-se o Problema do Vetor Mais Próximo (CVP) e o Problema do Vetor Mais Curto (SVP), para os quais não se conhecem soluções eficientes em tempo polinomial, tornando-os fortes candidatos para fundamentar algoritmos criptográficos robustos.

Definição 2 (Problema do Vetor Mais Próximo (CVP))**.**
Considere um reticulado $\Lambda \subset \mathbb{R}^n$ e um vetor $\mathbf{t} \in \mathbb{R}^n$. O problema do vetor mais próximo (CVP) consiste em encontrar um vetor $\mathbf{v} \in \Lambda$ tal que a distância euclidiana $\|\mathbf{v} - \mathbf{t}\|$ seja mínima, ou seja, $\|\mathbf{v} - \mathbf{t}\| = \min_{\mathbf{u} \in \Lambda} \|\mathbf{u} - \mathbf{t}\|$.

Definição 3 (Problema do Vetor Mais Curto (SVP))**.**
Dado um reticulado $\Lambda \subset \mathbb{R}^n$, o problema do vetor mais curto (SVP) busca determinar um vetor não-nulo $\mathbf{v} \in \Lambda$ cuja norma euclidiana seja a menor possível entre os vetores do reticulado. Isto é, $\|\mathbf{v}\| = \min_{\mathbf{u} \in \Lambda \setminus \{0\}} \|\mathbf{u}\|$.

A complexidade do SVP, particularmente em altas dimensões, dificulta sua aplicação direta em esquemas criptográficos práticos. Isso conduziu à introdução do problema γ-SVP, uma aproximação do SVP, onde o objetivo é encontrar um vetor cujo comprimento é até γ vezes o comprimento do vetor mais curto. Formalmente, o γ-SVP pode ser definido como:

Definição 4 (Problema do Vetor Mais Curto com Aproximação (γ-SVP))**.**
Dado um reticulado Λ e um fator de aproximação γ, o objetivo é encontrar um vetor $\mathbf{v} \in \Lambda$ tal que $0 < \|\mathbf{v}\| \leq \gamma \cdot \lambda_1(\Lambda)$, onde $\lambda_1(\Lambda)$ é a norma do vetor mais curto de Λ.

A partir do γ-SVP, foi introduzido por Regev em 2005 [44], o problema de Aprendizado com Erros (LWE). Esta construção foi considerada relativamente simples de aplicar em esquemas criptográficos e assintoticamente ao menos tão forte quanto outros problemas de pior caso de reticulados [40]. O LWE é definido da seguinte forma:

Definição 5 (Problema Aprendizado com Erros (LWE))**.**
Dado um conjunto de amostras $(\mathbf{a}_i, b_i)$ *geradas por* $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \bmod q$*, com* $\mathbf{s}$ *sendo um vetor secreto e* e_i *um erro pequeno, o desafio é recuperar* $\mathbf{s}$*.*

A introdução de um pequeno erro (e é pequeno em relação a q) é fundamental para garantir a segurança, uma vez que serve para mascarar a estrutura linear do problema, tornando a tarefa de distinguir entre um conjunto de amostras lineares legítimas e amostras aleatórias praticamente equivalente à dificuldade de resolver problemas de reticulados em seu pior caso.

É possível robustecer o emprego do LWE na criptografia de reticulados com auxílio da estrutura algébrica módulo. Um módulo generaliza a noção de espaço vetorial, substituindo os escalares por um anel. Essas estruturas são convenientes, pois permitem que operações com vetores e matrizes de polinômios sejam realizadas usando as regras e notações convencionais de álgebra linear, como a adição de vetores, multiplicação por escalares e produto escalar.

Definição 6 (Módulo)**.**
Seja $R_q = \mathbb{F}_q[x]/(x^n + 1)$*. Para qualquer inteiro* $k \geq 1$*, o conjunto*

$$R_q^k = \left\{ \begin{bmatrix} p_1 \\ \vdots \\ p_k \end{bmatrix} : p_i \in R_q \right\},$$

é um módulo de dimensão k *sobre* R_q*. Os elementos dos módulos, convenientemente chamados de vetores, seguem as operações definidas pela álgebra linear sobre o anel* R_q*, o que permite manipulações algébricas sofisticadas.*

Em 2009, Chris Peikert expandiu significativamente o escopo do LWE ao introduzir variações que utilizam módulos sobre anéis de polinômios, culminando na formulação do problema de Aprendizado com Erros em Módulos (MLWE). Esta generalização não só amplia a estrutura algébrica do problema original, mas também aprimora sua eficiência e escalabilidade, tornando-a ainda mais pertinente para aplicações criptográficas [28].

O MLWE pode ser formalmente definido como segue:

Definição 7 (Problema Aprendizado com Erros em Módulos (MLWE))**.**
Seja $R = \mathbb{Z}_q[x]/\langle f(x) \rangle$ *um anel de polinômios quociente, onde* $f(x)$ *é um polinômio mônico e* q *um inteiro positivo que define o módulo. Dado um vetor secreto* $s \in R^k$*, uma matriz pública* $A \in R^{m \times k}$ *e um vetor de erros* $e \in R^m$ *cujas entradas seguem uma distribuição de ruído sobre* R*, o desafio é recuperar* s *a partir de amostras da forma* $(A, As + e)$*.*

Neste problema, m, k, e l são parâmetros que afetam a segurança do esquema, e a dificuldade de resolver o MLWE depende fortemente da escolha de R, da distribuição do ruído, e das dimensões da matriz.

Um outro problema importante para o contexto dessa dissertação é o Problema da Solução de Inteiros Curtos (SIS), que desafia o adversário a encontrar um vetor não-trivial de inteiros pequenos $\mathbf{x}$ tal que, quando multiplicado por uma matriz A geradora de um reticulado, resulte no vetor nulo sobre um módulo q, ou seja, $A\mathbf{x} \equiv \mathbf{0} \bmod q$. Esta tarefa é computacionalmente difícil devido às propriedades intrínsecas dos reticulados, onde a complexidade de encontrar tal solução escala exponencialmente com o aumento das dimensões dessa estrutura.

Definição 8 (Problema da Solução de Inteiros Curtos (SIS))**.**
Dado uma matriz $A \in \mathbb{Z}_q^{m \times n}$ e um vetor $\mathbf{u} \in \mathbb{Z}_q^m$, encontrar um vetor não-trivial $\mathbf{x} \in \mathbb{Z}^n$ tal que $A\mathbf{x} = \mathbf{u} \bmod q$ e $\|\mathbf{x}\|$ é pequena (tipicamente norma ℓ_2 ou ℓ_∞ pequena).

O problema é considerado difícil porque requer a identificação de um vetor $\mathbf{x}$ que não só satisfaz a equação linear módulo q mas também possui norma pequena, o que dificulta a solução devido às restrições combinatórias e algébricas.

Assim como ocorre no LWE, quando o módulo $\mathbb{Z}_q^n$ é substituído por um módulo sobre um anel maior que $\mathbb{Z}_q$ (como R_q), o problema resultante é chamado de MSIS (Solução de Inteiros Curtos em Módulos).

2.1.3 Reduções Modulares

A implementação eficiente de operações matemáticas é fundamental para garantir tanto a segurança quanto o desempenho dos sistemas criptográficos. Nesse sentido, torna-se relevante o emprego de técnicas para acelerar operações frequentes como as reduções modulares. Dentre as formas eficientes de realizar tais operações estão a Redução de Barrett e a Redução de Montgomery.

A redução de Barrett evita a divisão direta, computacionalmente custosa, usando uma aproximação baseada em um valor pré-computado μ, que é tipicamente selecionado como $\lfloor 2^{2t}/q \rfloor$ para algum t maior que o número de bits de q.

Definição 9 (Redução de Barrett)**.**
Seja x um inteiro grande do qual desejamos calcular $x \bmod q$. A redução é realizada através da seguinte operação:

$$r = x - \left(\left\lfloor \frac{x \cdot \mu}{2^{2t}} \right\rfloor \cdot q \right),$$

onde μ é uma constante pré-computada, tipicamente $\mu = \left\lfloor \frac{2^{2t}}{q} \right\rfloor$ para um t escolhido tal que 2^t seja maior ou igual ao maior valor de x esperado na operação.

Este resultado pode precisar de um ajuste final para garantir que $0 \leq r < q$, sendo ajustado por $r = r - q$ se $r \geq q$.

A redução de Montgomery é um método eficiente para realizar multiplicações modulares, especialmente útil em aplicações criptográficas. O método consiste em substituir as divisões convencionais, que são computacionalmente custosas, por operações de multiplicação e deslocamento de bits que têm menor custo computacional. Formalmente, o procedimento é descrito como segue:

Definição 10 (Redução de Montgomery)**.**
Seja q um número primo e R uma potência de dois escolhida de modo que $R > q$. Definimos R^{-1} como o inverso de R módulo q, o qual satisfaz a relação: $R \cdot R^{-1} \equiv 1 \pmod{q}$.

Para um dado inteiro x, a representação em Montgomery de x é definida por $\overline{x} = x \cdot R \bmod q$. Considere dois inteiros x e y, ambos representados na forma de Montgomery. A redução de Montgomery do produto $x \cdot y$ em relação a R e q é executada pela função M, que é definida por:

$$M(x \cdot y) = (x \cdot y \cdot R^{-1} \mod q).$$

Esta operação efetua a redução do produto $x \cdot y$ para sua forma normal enquanto elimina o fator R adicional introduzido pela representação em Montgomery. Isso permite que as operações subsequentes usem números na forma padrão sem a necessidade de conversões adicionais fora da representação de Montgomery.

A redução é eficiente porque R é uma potência de 2, o que permite que as divisões sejam implementadas como deslocamentos de bits à direita, uma operação muito mais rápida em hardware do que a divisão aritmética tradicional.

2.1.4 Transformada da Teoria dos Números (NTT)

A Transformada da Teoria dos Números (NTT) é uma adaptação da Transformada Discreta de Fourier (DFT) para o domínio dos números inteiros. A DFT transforma um vetor de números complexos ou reais em um vetor de números complexos representando as frequências do sinal original. A NTT, por outro lado, opera em um domínio finito, geralmente um corpo finito $\mathbb{Z}/p\mathbb{Z}$ onde p é um número primo que define a característica do corpo.

A NTT viabiliza a conversão do processo de multiplicação de polinômios, que é computacionalmente custoso, em multiplicações ponto a ponto, que podem ser executadas de maneira mais eficiente do que no método *Schoolbook*, a forma mais simples de multiplicação polinomial, cujas complexidades de espaço e tempo são respectivamente $O(n)$ e $O(n^2)$ conforme equação 2.1.

$$C(x) = \sum_{k=0}^{2n-2} c_k x^k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j} \tag{2.1}$$

Por outro lado, para realizar processo análogo empregando a NTT são necessárias as seguintes etapas:

1. **Transformação:** Os coeficientes do polinômio são transformados do domínio dos coeficientes para o domínio da transformada utilizando a NTT.

2. **Multiplicação ponto a ponto:** Cada coeficiente transformado do primeiro polinômio é multiplicado pelo coeficiente correspondente do segundo polinômio transformado, realizando-se assim uma multiplicação ponto a ponto eficiente no domínio da transformada.

3. **Interpolação:** A inversa da NTT (INTT) é aplicada para transformar os resultados de volta para o domínio dos coeficientes.

Esse processo reduz a complexidade computacional da multiplicação de polinômios de $O(n^2)$ para $O(n \log n)$, assumindo que a NTT e sua inversa são implementadas eficientemente. Nesse contexto, a multiplicação de dois polinômios, a e b, é realizada da seguinte forma:

$$c = a \times b = \text{INTT}(\text{NTT}(a) \circ \text{NTT}(b)) \tag{2.2}$$

Raízes Primitivas da Unidade e Estrutura da NTT

Para que a NTT seja aplicável, é necessário que exista uma raiz primitiva n-ésima da unidade no corpo $\mathbb{Z}/p\mathbb{Z}$, onde p é um número primo que satisfaz $p \equiv 1 \bmod 2n$. Esta condição garante a existência de n raízes distintas da unidade, essenciais para a transformada. Um número ζ que satisfaz $\zeta^n \equiv 1 \bmod p$ e $\zeta^k \not\equiv 1 \bmod p$ para todo $0 < k < n$ é usado para construir as bases da NTT, facilitando a conversão da convolução de polinômios em multiplicações ponto a ponto.

Definição 11 (Transformada da Teoria dos Números (NTT))**.**
Seja p um número primo tal que $p \equiv 1 \bmod 2n$. Esta condição garante a existência de uma n-ésima raiz primitiva da unidade $\zeta \in \mathbb{Z}/p\mathbb{Z}$, ou seja, um elemento que satisfaz:

$$\zeta^n \equiv 1 \bmod p \quad e \quad \zeta^k \not\equiv 1 \bmod p, \forall\, 0 < k < n.$$

*A **Transformada da Teoria dos Números (NTT)** de um vetor de coeficientes $a = (a_0, a_1, \ldots, a_{n-1}) \in (\mathbb{Z}/p\mathbb{Z})^n$ é definida como o vetor:*

$$NTT_p(a) = (A_0, A_1, \ldots, A_{n-1}), \quad onde \quad A_k = \sum_{j=0}^{n-1} a_j \zeta^{jk} \bmod p,\ k = 0, 1, \ldots, n-1.$$

Sendo que:

- *ζ é a raiz primitiva n-ésima da unidade em $\mathbb{Z}/p\mathbb{Z}$;*
- *A_k representa a avaliação do polinômio $a(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ no ponto ζ^k no corpo finito.*

A NTT transforma o vetor a do domínio dos coeficientes para um vetor de avaliações no domínio das raízes primitivas. O processo inverso, conhecido como a Inversa da NTT (INTT), permite retornar ao domínio original dos coeficientes.

Considerações de Segurança e Eficiência

Uma implementação direta da NTT de um polinômio a no anel $\mathbb{F}_q[x]/(x^n + 1)$ iniciaria com a seleção de uma raiz da unidade ω de ordem $2n$. Esta raiz facilita a decomposição do polinômio em componentes menores que correspondem a valores de a avaliados nas potências sucessivas de ω.

No entanto, para aplicações criptográficas, onde a eficiência e a segurança são de extrema importância, métodos mais rápidos, adaptados de algoritmos da Transformada

Rápida de Fourier (FFT), como Cooley-Tukey e Gentleman-Sande, são geralmente preferidos. Estes algoritmos permitem uma decomposição mais eficiente do polinômio a em componentes que podem ser tratados de forma mais eficaz ao longo das subárvores da árvore de decomposição de $x^n + 1$, utilizando ω em um corpo finito.

As definições desses métodos para o cálculo da NTT e de sua inversa (INTT) são apresentadas a seguir:

Definição 12 (NTT usando o Algoritmo de Cooley-Tukey)**.**
Este método divide a NTT de um vetor de n elementos em transformadas menores de forma recursiva, facilitando o cálculo em corpos finitos onde n é uma potência de 2. Formalmente, se n é uma potência de 2, a transformação pode ser dividida em duas partes de tamanho $n/2$:

$$NTT(a)[k] = \sum_{j=0}^{n/2-1} \left(a[j] + \omega^k a[j + n/2]\right) \omega^{2jk} \bmod q,$$

onde ω é uma raiz primitiva n-ésima da unidade no corpo $\mathbb{F}_q$.

Definição 13 (INTT usando o Algoritmo de Gentleman-Sande)**.**
Este método reorganiza o vetor de entrada antes de aplicar transformadas menores, otimizando a execução da INTT para recuperação de polinômios no domínio dos coeficientes. A transformação inversa é dividida em duas partes de tamanho $n/2$ de forma recursiva, similar ao método Cooley-Tukey, mas aplicada de trás para frente. Formalmente, a INTT pode ser expressa como:

$$INTT(A)[k] = \frac{1}{n} \sum_{j=0}^{n/2-1} \left(A[j] + \omega^{-k} A[j + n/2]\right) \omega^{-2jk} \bmod q,$$

onde ω^{-1} é a inversa da raiz n-ésima da unidade utilizada na NTT, garantindo que os resultados sejam ajustados corretamente ao corpo $\mathbb{F}_q$.

2.2 Segurança

Esta seção descreve os conceitos e termos essenciais relacionados à segurança de algoritmos criptográficos empregados nos capítulos subsequentes da dissertação.

2.2.1 Segurança em Algoritmos Assimétricos

O nível de segurança de um algoritmo criptográfico é uma medida quantitativa que descreve a resistência do algoritmo contra ataques adversos. Ele é frequentemente expresso em bits e refere-se ao número de operações (geralmente binárias) que um adversário precisaria realizar para comprometer o algoritmo. Um nível de segurança de 128 bits, por exemplo, implica que são necessárias aproximadamente 2^{128} operações para comprometer o esquema criptográfico, o que é considerado inviável com a tecnologia atual. O tamanho da chave em um algoritmo assimétrico é um dos principais fatores que determina seu nível

de segurança; algoritmos com chaves maiores geralmente oferecem maior segurança, mas também maior consumo de recursos computacionais.

Outra definição de segurança diz respeito às ameaças e às garantias que um algoritmo criptográfico oferece contra elas. Para o contexto dessa dissertação, três dessas definições são particularmente importantes: IND-CPA (indistinguibilidade sob ataques de texto claro escolhido), IND-CCA2 (indistinguibilidade sob ataques de texto cifrado escolhido adaptativo), e SUF-CMA (segurança contra falsificação existencial forte sob ataque de mensagem escolhida). Enquanto as duas primeiras dizem respeito a análise de esquemas de criptografia de chave pública e mecanismos de encapsulamento de chaves, a última está relacionada a avaliação da segurança de esquemas de assinatura digital.

Indistinguibilidade sob Ataques de Texto Claro Escolhido (IND-CPA)

A segurança de um esquema de criptografia de chave pública (PKE) é definida em termos de indistinguibilidade sob ataques de texto claro escolhido. Formalmente, a segurança em termos de indistinguibilidade é apresentada como um jogo criptográfico [49, 6], onde um criptossistema é considerado seguro, se nenhum adversário pode vencer o jogo com uma probabilidade significativamente maior do que a de um palpite aleatório.

Seja $\mathcal{A}$ um adversário probabilístico de tempo polinomial, que atua em duas etapas e tem como objetivo vencer o jogo IND-CPA de PKE, descrito a seguir. Na primeira etapa, $\mathcal{A}$ tem acesso a um oráculo de encriptação `Enc()` para encriptar um número arbitrário (polinomialmente limitado) de mensagens de sua escolha. Na segunda etapa, $\mathcal{A}$ submete duas mensagens distintas e novas m_0, m_1, e recebe uma das mensagens encriptada, c_b. O objetivo do adversário é decidir qual mensagem m_b está encriptada em um dado criptograma:

$$
\begin{aligned}
&\text{Jogo IND-CPA}^{\mathcal{A}}_{\text{PKE}}\text{:} \\
&(pk, sk) \leftarrow \text{KeyGen}() \\
&b \leftarrow \{0, 1\} \\
&(m_0, m_1) \leftarrow \mathcal{A}(pk) \\
&c_b \leftarrow \text{Enc}(pk, m_b) \\
&b' \leftarrow \mathcal{A}^{\text{Enc}()}(pk, c_b) \\
&\text{return } b \stackrel{?}{=} b'.
\end{aligned}
$$

Um esquema PKE é considerado seguro sob IND-CPA, se para todos os adversários eficientes $\mathcal{A}$ existe uma função negligenciável $\text{negl}(n)$ do parâmetro de segurança n, tal que a vantagem de $\mathcal{A}$ em vencer o jogo IND-CPA PKE é dada por:

$$
\text{Adv}_{\text{IND-CPA PKE}}(\mathcal{A}) = \left| \Pr[\text{IND-CPA}^{\mathcal{A}}_{\text{PKE}} = 1] - \frac{1}{2} \right| < \text{negl}(n).
$$

Indistinguibilidade sob Ataques de Texto Cifrado Escolhido Adaptativo (IND-CCA2)

A noção padrão de segurança para um Mecanismo de Encapsulamento de Chaves (KEM) é a indistinguibilidade sob ataques de texto cifrado escolhido adaptativo [41]. De maneira similar ao IND-CPA, um adversário tem acesso a um oráculo de encapsulamento `Encaps()` durante todo o ataque, permitindo que encapsule um número arbitrário de chaves de sua escolha. Além disso, o atacante tem acesso contínuo a um oráculo de desencapsulamento `Decaps()`. A segurança IND-CCA proporciona garantias de segurança mais fortes em comparação com o IND-CPA e é formalizada no seguinte jogo:

$$\begin{aligned}
&\text{Jogo IND-CCA2}^{\mathcal{A}}_{\text{KEM}}\text{:}\\
&(pk, sk) \leftarrow \text{KeyGen}()\\
&b \leftarrow \{0,1\}\\
&(K'_1, (c', K'_0)) \leftarrow \text{Encaps}(pk)\\
&b' \leftarrow \mathcal{A}^{\text{Decaps}()}(pk, c', K'_b)\\
&\text{return } b \stackrel{?}{=} b'.
\end{aligned}$$

Um KEM é considerado seguro sob IND-CCA2, se para todos os adversários eficientes $\mathcal{A}$ a probabilidade de vencer o jogo $\text{IND-CCA2}^{\mathcal{A}}_{\text{KEM}}$ é negligenciável. Mais precisamente, dado alguma função negligenciável $\text{negl}(n)$ do parâmetro de segurança n:

$$\text{Adv}_{\text{IND-CCA2 KEM}}(\mathcal{A}) = \left|\Pr[\text{IND-CCA2}^{\mathcal{A}}_{\text{KEM}} = 1] - \frac{1}{2}\right| < \text{negl}(n).$$

Segurança Contra Falsificação Existencial Forte sob Ataque de Mensagem Escolhida (SUF-CMA)

É a forma mais rigorosa de segurança para esquemas de assinatura digital, exigindo que um adversário, mesmo após observar assinaturas válidas para um número arbitrário de mensagens escolhidas, não consiga forjar uma nova assinatura para qualquer par de mensagem-assinatura e, adicionalmente, não consiga alterar qualquer assinatura existente de forma que continue a ser verificada como válida. Esta propriedade adicional visa prevenir o que é conhecido como *mauling* de assinaturas, onde uma assinatura legítima é transformada em outra também válida, mas maliciosamente alterada. O jogo criptográfico que formaliza essa propriedade é descrito da seguinte forma:

$$\begin{aligned}
(pk, sk) &\leftarrow \text{KeyGen}()\\
\{(m_i, \sigma_i)\} &\leftarrow \mathcal{A}^{\text{Sign}()}(sk, \text{queries})\\
(m^*, \sigma^*) &\leftarrow \mathcal{A}\\
&\text{return Verify}(pk, m^*, \sigma^*) \stackrel{?}{=} \text{true and } (m^*, \sigma^*) \notin \{(m_i, \sigma_i)\}
\end{aligned}$$

Um esquema de assinatura é considerado seguro sob SUF-CMA se nenhum adversário

eficiente conseguir produzir uma nova assinatura (m^*, σ^*) que seja verificada com sucesso, com exceção de uma probabilidade negligenciável, definida por uma função $\text{negl}(n)$. Formalmente, a vantagem do adversário $\mathcal{A}$ em forjar uma assinatura é definida como:

$$\text{Adv}_{\text{SUF-CMA}}(\mathcal{A}) = \Pr[\text{Verify}(pk, m^*, \sigma^*) = \text{true}] < \text{negl}(n).$$

Transformada de Fujisaki-Okamoto

No contexto de desenvolvimento de esquemas de criptografia de chave pública, almeja-se atingir a segurança IND-CCA2. Esta forma de segurança é considerada a mais robusta, assegurando por implicação o cumprimento de noções de segurança menos rigorosas. No entanto, a demonstração de que um esquema atende ao padrão IND-CCA2 apresenta desafios significativos, superiores aos envolvidos na verificação da segurança IND-CPA, como discutido por Hofheinz et al [20]. Diante dessa complexidade, diversas transformações foram propostas para elevar esquemas de criptografia de chave pública que possuem inicialmente noções de segurança mais fracas até o nível de segurança IND-CCA2.

A Transformada de Fujisaki e Okamoto [18] é uma dessas técnicas e funciona, essencialmente, encapsulando o texto claro com um valor aleatório e aplicando uma função de resumo criptográfico em conjunto com a chave pública para garantir que as alterações no texto cifrado resultem em saídas imprevisíveis.

Em termos formais, se considerarmos que $\mathcal{E}(m)$ é o algoritmo de encriptação que depende de um texto claro m e uma chave pública pk, e $\mathcal{D}(c)$ é o algoritmo de decriptação que depende de um texto cifrado c e uma chave privada sk, a transformada de Fujisaki-Okamoto modifica $\mathcal{E}$ e $\mathcal{D}$ para incluir componentes aleatórios e hashes de modo que:

$$\mathcal{E}'(m) = (\mathcal{E}(m, r), \mathcal{H}(\mathcal{E}(m, r), pk))$$

$$\mathcal{D}'(c) = \begin{cases} \mathcal{D}(c, sk) & \text{se } \mathcal{H}(c, pk) \text{ for válido} \\ \text{detecta erro} & \text{caso contrário;} \end{cases}$$

onde r é um valor aleatório e $\mathcal{H}$ é uma função de resumo criptográfico que combina partes do texto cifrado e da chave pública para verificar a integridade e autenticidade do texto cifrado. Em caso de detecção de erro durante a decriptação, o algoritmo gera uma chave aleatória e continua o processo de forma indistinguível, garantindo que um atacante não possa obter qualquer vantagem ou informação adicional nessa situação.

Essa técnica tem sido amplamente adotada em esquemas de criptografia modernos, incluindo os esquemas para encapsulamento de chaves submetidos ao NIST no processo de padronização de algoritmos pós-quânticos [19]. No esquema Kyber, a transformada de Fujisaki-Okamoto é implementada nos processos de encapsulamento e desencapsulamento que serão abordados no Capítulo 3.

2.2.2 Ataques de canais laterais

Um ataque de canal lateral é uma técnica que explora informações colaterais vazadas durante a execução de um algoritmo criptográfico. Essas informações podem incluir con-

sumo de energia, emissões eletromagnéticas, variações no tempo de execução e até mesmo sons emitidos pelo dispositivo. De acordo com Kocher et al. [26], ataques de canal lateral podem fornecer aos adversários dados críticos para comprometer sistemas criptográficos que, teoricamente, seriam seguros.

Nos últimos anos, as pesquisas demonstrando como os ataques de canal lateral afetam a criptografia pós-quântica têm se expandido. Os ataques de tempo, inicialmente demonstrados como aplicáveis à esquemas baseados em reticulados por Silverman e Whyte [50], foram posteriormente explorados nos estudos de [12] [15]. Ataques baseados em cache foram identificados como exploráveis em [10]. Além disso, ataques de análise de potência em esquemas baseados em reticulados foram documentados em uma lista crescente de trabalhos [53] [42] [22].

Em 2022, Azouaoui et al. [2], afirmaram que a transformada de Okamoto, presente em muitos esquemas de troca de chaves pós-quânticos adiciona resistência adequada contra um adversário de caixa preta, porém não leva em conta vazamentos durante seu cálculo.

Posteriormente, em 2023, Cheng et al. [43] publicaram em seu artigo um ataque de canal lateral denominado GoFetch. Este ataque pode extrair chaves secretas de implementações criptográficas de tempo constante por meio de pré-buscadores dependentes de memória de dados (DMPs) presentes em muitas CPUs da Apple. Os autores demonstraram que o GoFetch tem potencial para ameaçar implementações do OpenSSL Diffie-Hellman Key Exchange, Go RSA decryption, bem como do CRYSTALS-Kyber e do CRYSTALS-Dilithium.

No ano seguinte, Berstein [7] publicou um artigo onde demonstra um ataque ao algoritmo Kyber, no qual se exploram as variações no tempo de desencapsulamento no código de referência oficial na versão de novembro de 2023.

Diante desse cenário, é essencial implementar contramedidas adequadas. As técnicas para se contrapor a esses ataques variam, mas geralmente incluem abordagens como a ofuscação do processo, onde artifícios são usadas para mascarar as operações criptográficas e tornar os vazamentos de informação menos úteis para um atacante, e o cuidado para que operações distintas consumam quantidades similares de energia, dificultando a análise para fins maliciosos. Estratégias de hardware também são empregadas, como a adição de circuitos projetados para equalizar o consumo de energia ou para gerar ruídos que ofuscam os sinais úteis que poderiam ser explorados para fins escusos [21].

Contramedidas

A implementação de algoritmos criptográficos deve levar em conta a necessidade de mitigação de canais laterais. Para abordar esse desafio, podem ser aplicadas as seguintes técnicas:

a) Mascaramento:

Técnicas de mascaramento podem ser empregadas para ocultar as informações sensíveis durante a execução de operações críticas, tornando mais difícil para um adversário inferir dados secretos através de canais laterais. Essa contramedida envolve a aleatorização dos cálculos e a mistura de valores intermediários para impedir ataques baseados em análise de potência ou tempo.

b) Minimização de Variações de Tempo de Execução: A minimização das variações no tempo de execução pode ser alcançada através de técnicas como a padronização dos ciclos de execução e a utilização de algoritmos de tempo constante. Isso garante que operações criptográficas sensíveis tenham um comportamento uniforme, dificultando a exploração de canais laterais para obter informações secretas. Uma das formas de por em prática essa contramedida está no emprego de técnicas de programação sem desvios condicionais. A aplicação correta dessas técnicas é altamente dependente do algoritmo em análise, mas uma generalização útil é calcular os dois ramos do desvio condicional simultaneamente e utilizar operações mascaradas para selecionar o valor correto apenas ao final da execução [27]. A ideia é ilustrada pelo trecho de código abaixo para seleção entre dois valores em tempo constante, dependendo de um bit de condição:

```
unsigned select(unsigned a, unsigned b, unsigned bit) {
   /* -0 = 0, -1 = 0xFF....FF */
   unsigned mask = - bit;
   unsigned ret = mask & (a^b);
   ret = ret^a;
   return ret;
}
```

Listagem 2.1: Trecho de código para escolha em tempo constante entre dois valores com base em um bit.

Capítulo 3

CRYSTALS-Kyber

Um Mecanismo de Encapsulamento de Chaves (KEM) consiste em um conjunto de algoritmos que podem ser utilizados para estabelecer uma chave secreta compartilhada entre duas partes que se comunicam em um canal público. Esta chave secreta compartilhada pode então ser usada para criptografia simétrica.

Os algoritmos que compõem um KEM são o algoritmo de geração de chaves (*KeyGen*), o algoritmo de encapsulamento (*Encaps*) e o algoritmo de Desencapsulamento (*Decaps*). Além disso, um KEM possui um conjunto de parâmetros relacionados ao compromisso entre segurança e eficiência.

A Figura 3.1 ilustra o processo onde Alice e Bob desejam estabelecer uma chave simétrica para comunicação sigilosa. Alice começa executando o algoritmo *KeyGen* para gerar uma chave de encapsulamento (pública) e uma chave de desencapsulamento (privada). Após obter a chave de encapsulamento de Alice, Bob executa o algoritmo *Encaps*; isso produz a cópia de Bob (K) da chave secreta compartilhada junto com um criptograma associado. Bob envia o criptograma para Alice, e Alice completa o processo executando o algoritmo *Decaps* usando sua chave de desencapsulamento e o criptograma; esta etapa produz a cópia de Alice (K') da chave secreta compartilhada.

O CRYSTALS-Kyber é um KEM cuja segurança se baseia na dificuldade do problema MLWE. Este problema propõe a mesma tarefa que o LWE, mas com $\mathbb{Z}_q^n$ substituído pelo módulo R_q^k, construído ao tomar o produto cartesiano k-vezes de um certo anel polinomial R_q para algum inteiro $k > 1$. Em particular, o segredo é um elemento x do módulo R_q^k.

Em um alto nível, a construção do Kyber ocorre em duas etapas. Primeiro, a ideia mencionada acima é usada para construir um esquema de criptografia de chave pública (Kyber.CPAPKE) a partir do problema MLWE. Posteriormente, esse esquema é convertido em um mecanismo de encapsulamento de chave (Kyber.CCAKEM) usando uma variante da transformada de Fujisaki-Okamoto.

Para atender às diretrizes estabelecidas pelo NIST, o Kyber oferece três conjuntos de parâmetros: Kyber512, Kyber768 e Kyber1024, projetados para serem equivalentes às categorias de segurança 1,3 e 5. Esses conjuntos de parâmetros são apresentados na Tabela 3.1.

- n é o grau máximo dos polinômios utilizados no esquema. A razão pela qual $n = 256$ se deve ao objetivo de encapsular chaves com 256 bits de entropia (ou seja, usar um

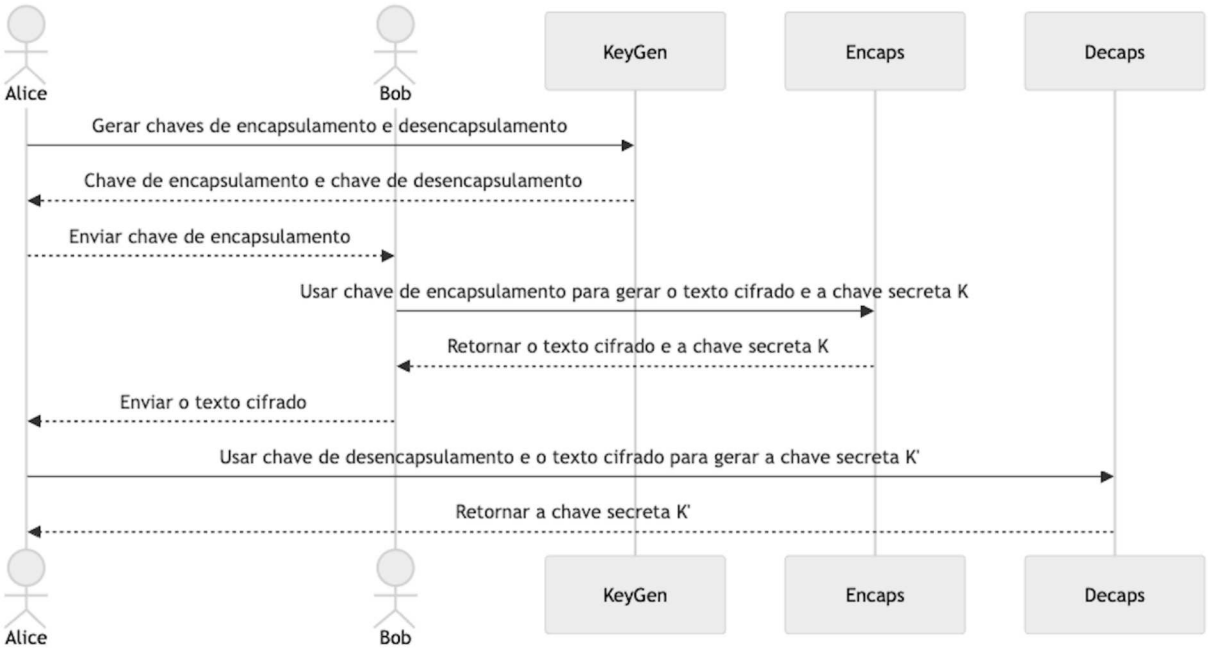

Figura 3.1: Diagrama de Sequência de um mecanismo de estabelecimento de chaves usando um KEM

Variante	**Categoria de Segurança**	n	q	k	η_1	η_2	d_u, d_v
Kyber512	1	256	3329	2	3	2	(10,4)
Kyber768	3	256	3329	3	2	2	(10,4)
Kyber1024	5	256	3329	4	2	2	(11,5)

Tabela 3.1: Conjunto de parâmetros do Kyber e respectivas correspondências aos níveis de segurança do NIST

tamanho de texto claro de 256 bits no algoritmo de encriptação). Valores menores de n exigiriam codificar múltiplos bits de chave em um coeficiente polinomial, o que requer níveis de ruído mais baixos e, portanto, reduz a segurança. Valores maiores de n reduziriam a capacidade de escalar facilmente a segurança através do parâmetro k.

- q é o valor utilizado para operações de módulo. Foi definido como um pequeno primo que satisfaz $n \mid (q-1)$; isso é necessário para possibilitar a multiplicação rápida baseada em NTT. Há dois primos menores para os quais essa propriedade se mantém, a saber, 257 e 769. No entanto, para esses primos não seria possível alcançar uma probabilidade de falha negligenciável necessária para a segurança CCA, sendo então escolhido o próximo maior, ou seja, $q = 3329\ (2^8 \cdot 13 + 1)$.

- k é selecionado para fixar a dimensão do reticulado como um múltiplo de n. Alterar k é o principal mecanismo do Kyber para escalar a segurança (e como consequência, a eficiência) para diferentes níveis.

- o parâmetro η_1 define o ruído de s e e no algoritmo de geração de chaves e de r

no algoritmo encriptação. O parâmetro η_2 define o ruído de e_1 e e_2 no algoritmo encriptação.

- d_u e d_v são parâmetros utilizados como entradas para as funções de compressão e serialização, bem como para os respectivos processos inversos.

A versão submetida à terceira rodada do processo de padronização do NIST foi utilizada para o estabelecimento do padrão FIPS 203 intitulado *Module-Lattice-based Key-Encapsulation Mechanism* (ML-KEM). Esse padrão organiza os algoritmos em algoritmos auxiliares, algoritmos do esquema de criptografia de chave pública (K-PKE), algoritmos internos e algoritmos do Mecanismo de Encapsulamento de Chaves (ML-KEM).

3.1 Algoritmos Auxiliares

3.1.1 Funções Criptográficas

Os algoritmos que compõe o ML-KEM requerem o uso de várias funções criptográficas. Cada uma delas deve ser instanciada por meio de uma função de resumo criptográfico aprovada pelo NIST ou uma função de saída extensível (XOF), as quais são descritas em detalhes no FIPS 202 [34].

Uma função de resumo criptográfico é um algoritmo que mapeia dados de comprimento variável para dados de comprimento fixo. Por outro lado, uma XOF é especialmente projetada para produzir saídas de comprimento variável.

Há apenas poucas funções de saída extensíveis descritas na literatura. As mais conhecidas, que também cunharam o termo XOF, são as funções SHAKE baseadas em Keccak [9]. As funções SHAKE são particularmente importantes, uma vez que permitem ajustes finos do tamanho de saída, adaptando-se assim a diversos requisitos de segurança sem comprometer a integridade criptográfica. Além disso, SHAKE é projetada para funcionar também como uma função pseudo-aleatória (PRF), que é um tipo de função elaborada para emular a aleatoriedade de tal forma que seja computacionalmente inviável distinguir sua saída de uma sequência verdadeiramente aleatória. Assim, as funções da família Keccak são adotadas como única primitiva a ser empregada no ML-KEM.

A decisão de confiar em apenas uma primitiva subjacente para todas essas funções se baseia na possibilidade de reduzir o tamanho do código em plataformas embarcadas e reduzir preocupações de ataques que explorem fraquezas em uma entre várias primitivas simétricas [1].

No contexto deste trabalho tais funções criptográficas são utilizadas da seguinte forma:

Função pseudo-aleatória (PRF). A função PRF toma um parâmetro $\eta \in \{2, 3\}$, uma entrada de 32 bytes e uma entrada de 1 byte. Ela produz uma saída de $(64 \cdot \eta)$-bytes. Será denotada por PRF: $\{2, 3\} \times B^{32} \times B \to B^{64\eta}$, e será instanciada como

$$PRF_\eta(s, b) := SHAKE256(s \| b, 64 \cdot \eta),$$

Função de saída extensível (XOF). O padrão emprega SHAKE128 como XOF. É adotada uma abordagem de API[1] incremental que permite a execução de operações criptográficas em etapas sucessivas, gerenciando o estado entre essas chamadas. Um *wrapper* XOF é utilizado para encapsular esta API, fornecendo uma interface simplificada para as funções subjacentes. A utilização de um *wrapper* garante que a manipulação dos dados seja feita de maneira consistente e segura, uma vez que uma XOF, como o SHAKE128, pode gerar uma quantidade arbitrária de dados de saída. Além disso, o *wrapper* facilita a implementação de mudanças ou atualizações no algoritmo subjacente sem impactar o código que utiliza a API, proporcionando uma camada adicional de abstração que pode aumentar a segurança e a manutenabilidade do software. Tal flexibilidade, bem como a segurança proporcionadas por essa arquitetura é descrita detalhadamente em [24].

A interface do *wrapper* XOF é descrita a seguir:

1. XOF.Init() = SHAKE128.Init().
   Inicializa um "contexto" XOF *ctx*.

2. XOF.Absorb(ctx, str) = SHAKE128.Absorb(ctx, str).
   Injeta dados para serem usados na fase de "absorção" do SHAKE128 e atualiza o contexto.

3. XOF.Squeeze(ctx, ℓ) = SHAKE128.Squeeze(ctx, $8 \cdot \ell$).
   Extrai ℓ bytes de saída produzidos durante a fase de "espremimento" do SHAKE128 e atualiza o contexto.

Note que XOF.Squeeze requer que o comprimento da entrada seja especificado em bytes. Isso está de acordo com a convenção de que todas as funções *wrapper* tratam entradas e saídas como vetores de bytes e medem os comprimentos de todos esses vetores em termos de bytes.

Três funções de resumo criptográfico. A especificação também faz uso de três instâncias de funções de resumo criptográfico H, J e G, da seguinte forma:

As funções H e J recebem cada uma uma entrada de comprimento variável e produzem uma saída de 32 bytes. São denotadas por $H : B^* \to B^{32}$ e $J : B^* \to B^{32}$, respectivamente, e serão instanciadas como:

$$H(s) := \text{SHA3-256}(s) \quad \text{e} \quad J(s) := \text{SHAKE256}(s, 32)$$

onde $s \in B^*$.

A função G recebe uma entrada de comprimento variável e produz duas saídas de 32 bytes. Será denotada por $G : B^* \to B^{32} \times B^{32}$ As duas saídas de G serão denotadas por $(a, b) \leftarrow G(c)$, onde $a, b \in B^{32}$, $c \in B^*$, e $G(c) = a \| b$. A função G será instanciada como:

$$G(c) := \text{SHA3-512}(c).$$

[1] API (Interface de Programação de Aplicações): é um conjunto de regras e especificações que facilitam a interação entre diferentes componentes de software, permitindo a comunicação e compartilhamento de funcionalidades de forma eficiente.

3.1.2 Conversão e Compressão

No domínio dos sistemas criptográficos modernos, o tratamento eficiente dos formatos de dados é um fator importante para a segurança e para a eficiência. A especificação ML-KEM provê uma estrutura detalhada para converter e comprimir dados, garantindo uma gestão robusta e segura ao longo dos processos que compõem o esquema. Nesse contexto, têm papel relevante as funções de conversão, compressão e codificação de dados cujos processos acompanhados dos respectivos métodos de reversão são resumidos a seguir.

Conversão entre Bits e Bytes

As transições entre arranjos de bits e bytes são essenciais para formatar dados de modo a maximizar a compatibilidade e a eficiência durante a transmissão e armazenamento. A operação `BitsToBytes` segmenta um arranjo de bits de maneira que cada segmento represente um byte em ordem *little-endian*[2]. Por outro lado, `BytesToBits` realiza a operação oposta para reverter os dados ao seu formato original após recepção ou durante processos de decodificação.

Compressão e Descompressão

A compressão de dados, realizada pelo algoritmo `Compress_d`, é projetada para reduzir o espaço de armazenamento e a largura de banda necessária para transmissões, comprimindo inteiros módulo q para inteiros em um domínio menor $\mathbb{Z}_{2^d}$. A função de descompressão, `Decompress_d`, é projetada para restaurar os dados ao seu formato original, permitindo a recuperação após a transmissão ou armazenamento em formato comprimido. Portanto, `Compress_d(Decompress_d(y)) = y` $\forall\ y \in \mathbb{Z}_q$ e $\forall\ d < 12$.

Codificação e Decodificação

As operações de codificação e decodificação, `ByteEncode_d` e `ByteDecode_d`, tem por objetivo viabilizar a serialização e desserialização de arranjos de inteiros módulo m, adequando o formato dos dados para armazenamento ou transmissão e vice-versa. Estas operações asseguram que cada inteiro seja representado de forma precisa e compacta. Especificamente, `ByteEncode_d` serializa um vetor de d-bits inteiros em um vetor de $32 \cdot$ d bytes. Essa expansão durante a serialização assegura que os coeficientes sejam processados de maneira uniforme, contribuindo para inviabilizar ataques que exploram a variação na representação dos dados.

3.1.3 Amostragem

Tradicionalmente, análises teóricas sobre esquemas que se fundamentam no problema LWE consideram o uso de ruído gaussiano, quer seja arredondado ou discreto. Como resultado, muitas implementações iniciais também amostraram ruído de uma distribuição gaussiana discreta, que se revelou ou bastante ineficiente ou vulnerável a ataques de tempo. Tal fato, motiva o uso de distribuições de ruído que possam ser amostradas de forma fácil,

[2]Little-endian é um formato de armazenamento de dados em sistemas de computação onde o byte menos significativo é armazenado no endereço de memória mais baixo e o byte mais significativo é armazenado no endereço de memória mais alto.

eficiente e segura. Uma das opções é a amostragem binomial centrada, que é conhecida por sua eficiência computacional e por ser menos suscetível a ataques de canal lateral. Dessa forma, o ruído no ML-KEM é amostrado de uma distribuição binomial centrada B_η para $\eta = 2$ ou $\eta = 3$.

Define-se B_η como:

$$B_\eta = \sum_{i=1}^{\eta} (b_i - b_{\eta+i}),$$

onde b_i são amostras independentes e uniformemente aleatórias com probabilidade $= \frac{1}{2}$.

Então, cada coeficiente de um polinômio em R_q é amostrado de forma independente a partir dessa distribuição, usando a função `SamplePolyCBD`, conforme descrito no Algoritmo 1.

Algoritmo 1 SamplePolyCBDη(B)

Entrada: vetor de bytes $B \in \mathbb{B}^{64\eta}$.
Saída: vetor $f \in Z_q^{256}$. ▷ os coeficientes do polinômio amostrado

1: $b \leftarrow \text{BytesToBits}(B)$
2: **for** $i = 0$ até 255 **do**
3: $\quad x \leftarrow \sum_{j=0}^{\eta-1} b[2i\eta + j]$
4: $\quad y \leftarrow \sum_{j=0}^{\eta-1} b[2i\eta + \eta + j]$
5: $\quad f[i] \leftarrow (x - y) \mod q$ ▷ $f \in \mathbb{Z}_q^{256}$
6: **end for**
7: **return** f

Além da amostragem de elementos por meio da distribuição binomial centrada, o ML-KEM também utiliza a amostragem direta de polinômios no domínio Tq empregando o Algoritmo 2 que descreve a função `SampleNTT`.

Essa função é utilizada para gerar os elementos da matriz $\hat{\mathbf{A}}$ e, dessa forma, proporciona maior eficiência ao esquema, uma vez que viabiliza operações futuras diretamente no domínio NTT.

A intuição por trás de `SampleNTT` é que, se o fluxo de bytes de entrada for estatisticamente próximo de um fluxo aleatório uniforme, então o polinômio de saída será estatisticamente próximo de um elemento aleatório uniforme de T_q tendo em vista que a NTT é bijetiva e, portanto, mapeia polinômios com coeficientes aleatórios uniformes para polinômios com coeficientes aleatórios uniformes no domínio transformado.

3.1.4 Algoritmos NTT

No contexto do ML-KEM, a NTT é usada para melhorar a eficiência da multiplicação no anel R_q. Lembre-se de que R_q é o anel $\mathbb{Z}_q[X]/(X^n + 1)$, que é naturalmente isomórfico a outro anel, denotado por T_q. Este último é uma soma direta de 128 extensões quadráticas de $\mathbb{Z}_q$. Assim, a NTT é um isomorfismo computacionalmente eficiente entre esses dois anéis. Ao receber um polinômio $f \in R_q$, a NTT produz um elemento $\hat{f} := \text{NTT}(f)$ do anel T_q, onde $\hat{f}$ é chamado de representação NTT de f. A propriedade de isomorfismo implica que $f \times_{R_q} g = \text{NTT}^{-1}(\hat{f} \times_{T_q} \hat{g})$, onde $\times_{R_q}$ e $\times_{T_q}$ denotam a multiplicação em R_q e

Algoritmo 2 SampleNTT(B)

Entrada: vetor de bytes $B \in \mathbb{B}^{34}$. ▷ uma semente de 32 bytes com dois índices
Saída: vetor $\hat{a} \in \mathbb{Z}_q^{256}$. ▷ os coeficientes da NTT de um polinômio

$ctx \leftarrow$ XOF.Init()
$ctx \leftarrow$ XOF.Absorb(ctx, B) ▷ insere o vetor de bytes dado no XOF
$j \leftarrow 0$
while $j < 256$ **do**
    $(ctx, C) \leftarrow$ XOF.Squeeze($ctx, 3$) ▷ obtém um novo vetor de 3 bytes C do XOF
    $d1 \leftarrow C[0] + 256 \cdot (C[1] \mod 16)$ ▷ $0 \leq d1 < 2^{12}$
    $d2 \leftarrow \lfloor C[1]/16 \rfloor + 16 \cdot C[2]$ ▷ $0 \leq d2 < 2^{12}$
    if $d1 < q$ **then**
        $\hat{a}[j] \leftarrow d1$
        $j \leftarrow j + 1$
    end if
    if $d2 < q$ and $j < 256$ **then**
        $\hat{a}[j] \leftarrow d2$
        $j \leftarrow j + 1$
    end if
end while
return $\hat{a}$

T_q, respectivamente. Além disso, como T_q é um produto de 128 anéis, cada um consistindo de polinômios de grau um, a operação $\times_{T_q}$ é muito mais eficiente que a operação $\times_{R_q}$. Por essas razões, a NTT é considerada parte integral do ML-KEM e não apenas uma otimização.

Uma vez que o primo q é definido como $3329 = 2^8 \cdot 13 + 1$ e $n = 256$, existem 128 raízes primitivas da unidade de 256-ésima ordem e nenhuma raiz primitiva da unidade de 512-ésima ordem em $\mathbb{Z}_q$. Note que $\zeta = 17 \in \mathbb{Z}_q$ é uma raiz primitiva da unidade de 256-ésima ordem módulo q. Assim, $\zeta^{128} \equiv -1$, o que é fundamental para a divisão do polinômio em fatores de grau 2.

Se considerarmos BitRev7(i) como o inteiro obtido pela reversão dos bits do valor inteiro sem sinal de 7 bits que corresponde ao inteiro de entrada $i \in \{0, ..., 127\}$, podemos reorganizar os índices para melhorar a simetria e a uniformidade dos coeficientes. Essa estruturação não apenas facilita a implementação eficiente da NTT, mas também melhora a paralelização e a eficiência do cache durante a computação. Especificamente, o polinômio $X^{256} + 1$ é fatorado em 128 polinômios de grau 2 módulo q conforme:

$$X^{256} + 1 = \prod_{k=0}^{127} (X^2 - \zeta^{2\text{BitRev7}(k)+1}).$$

Portanto, $R_q := \mathbb{Z}_q[X]/(X^{256} + 1)$ é isomórfico a uma soma direta de 128 corpos de extensão quadrática de $\mathbb{Z}_q$, denotado T_q. Especificamente, este anel tem a estrutura

$$T_q := \bigoplus_{k=0}^{127} \mathbb{Z}_q[X]/(X^2 - \zeta^{2\text{BitRev7}(k)+1})$$

Assim, a representação NTT $\hat{f} \in T_q$ de um polinômio $f \in R_q$ é o vetor que consiste nos resíduos de grau um correspondentes:

$$\hat{f} := (f \mod (X^2 - \zeta^{2\text{BitRev7}(0)+1}), ..., f \mod (X^2 - \zeta^{2\text{BitRev7}(127)+1})).$$

Nos Algoritmos 3 e 4 a seguir, $\hat{f}$ é armazenado como um vetor de 256 inteiros módulo q. Especificamente,

$$f \mod (X^2 - \zeta^{2\text{BitRev7}(i)+1}) = \hat{f}[2i] + \hat{f}[2i+1]X$$

para i de 0 a 127.

Algoritmo 3 NTT(f)

Entrada: vetor $f \in \mathbb{Z}_q^{256}$. ▷ os coeficientes do polinômio de entrada
Saída: vetor $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ os coeficientes do polinômio de entrada no domínio da NTT

$\hat{f} \leftarrow f$ ▷ computará a NTT in-place numa cópia do vetor de entrada
$k \leftarrow 1$
for len $\leftarrow 128$ **to** 2 **do**
  for start $\leftarrow 0$ **to** 256 **by** len/2 **do**
    $zeta \leftarrow \zeta^{\text{BitRev7}(k)} \mod q$
    $k \leftarrow k + 1$
    for $j \leftarrow$ start **to** start + len − 1 **do**
      $t \leftarrow zeta \cdot \hat{f}[j + \text{len}]$ ▷ passos 8-10 feitos módulo q
      $\hat{f}[j + \text{len}] \leftarrow \hat{f}[j] - t$
      $\hat{f}[j] \leftarrow \hat{f}[j] + t$
    end for
  end for
end for
return $\hat{f}$

A adição e a multiplicação por escalar de elementos de T_q são operações diretas: elas podem ser realizadas utilizando operações aritméticas coordenada a coordenada nos vetores de coeficientes. Por outro lado, a multiplicação em T_q consiste na multiplicação independente de cada uma das 128 coordenadas com respeito ao módulo quadrático dessa coordenada. Especificamente, a coordenada i-ésima em T_q do produto $\hat{h} = \hat{f} \times_{T_q} \hat{g}$ é determinada pelo cálculo:

$$\hat{h}[2i+1]X = (\hat{f}[2i] + \hat{f}[2i+1]X)(\hat{g}[2i] + \hat{g}[2i+1]X) \mod (X^2 - \zeta^{2\text{BitRev7}(i)+1} + \hat{h}[2i]).$$

Assim, pode-se computar o produto de dois elementos de T_q usando o algoritmo `MultiplyNTTs` (Algoritmo 5). Note que `MultiplyNTTs` utiliza `BaseCaseMultiply` (Algoritmo 6) como uma sub-rotina.

Algoritmo 4 $\text{NTT}^{-1}(\hat{f})$

Entrada: vetor $\hat{f} \in \mathbb{Z}_q^{256}$. ▷ os coeficientes no domínio da NTT
Saída: vetor $f \in \mathbb{Z}_q^{256}$. ▷ os coeficientes originais

$f \leftarrow \hat{f}$ ▷ computará in-place numa cópia do vetor de entrada
$k \leftarrow 127$
for len $\leftarrow 2$ **to** 128 **by** $2 \cdot$ len **do**
  for start $\leftarrow 0$ **to** 255 **by** $2 \cdot$ len **do**
    $zeta \leftarrow \zeta^{\text{BitRev7}(k)} \mod q$
    $k \leftarrow k - 1$
    for $j \leftarrow$ start **to** start + len $- 1$ **do**
      $t \leftarrow f[j]$
      $f[j] \leftarrow (f[j] + f[j + \text{len}]) \mod q$ ▷ passos 9-10 feitos módulo q
      $f[j + \text{len}] \leftarrow zeta \cdot (f[j + \text{len}] - t) \mod q$
    end for
  end for
end for
$f \leftarrow f \cdot 3303 \mod q$ ▷ multiplica cada entrada por $3303 \equiv 128^{-1} \mod q$
return f

Algoritmo 5 MultiplyNTTs

Entrada: dois vetores $\hat{f}, \hat{g} \in \mathbb{Z}_q^{256}$. ▷ os coeficientes de duas representações NTT
Saída: um vetor $\hat{h} \in \mathbb{Z}_q^{256}$. ▷ os coeficientes do produto das entradas

for $i \leftarrow 0$ **to** 127 **do**
  $(\hat{h}[2i], \hat{h}[2i+1]) \leftarrow \text{BaseCaseMultiply}(\hat{f}[2i], \hat{f}[2i+1], \hat{g}[2i], \hat{g}[2i+1], \zeta^{\text{BitRev7}(2i)+1})$
end for
return $\hat{h}$

Algoritmo 6 BaseCaseMultiply

Entrada: $a_0, a_1, b_0, b_1 \in \mathbb{Z}_q$. ▷ os coeficientes de $a_0 + a_1X$ e $b_0 + b_1X$
Entrada: $\gamma \in \mathbb{Z}_q$. ▷ o módulo é $X^2 - \gamma$
Saída: $c_0, c_1 \in \mathbb{Z}_q$. ▷ os coeficientes do produto dos dois polinômios

$c_0 \leftarrow a_0 \cdot b_0 + a_1 \cdot b_1 \cdot \gamma$ ▷ passos 1-2 feitos módulo q
$c_1 \leftarrow a_0 \cdot b_1 + a_1 \cdot b_0$
return c_0, c_1

3.2 Algoritmos do Componente *Public Key Encription* (K-PKE)

Os algoritmos que compõem o K-PKE são invocados como sub-rotinas do ML-KEM e, dessa forma, herdam deste o conjunto de parâmetros selecionados ao instanciar o esquema. Adicionalmente, tiram proveito do mecanismo de validação de entradas incorporado no ML-KEM, assegurando uma integração harmoniosa e eficiente.

Importante ressaltar que o K-PKE não é aprovado para uso independente, restringindo-se a integrar o padrão estabelecido pelo NIST discutido neste capítulo.

O K-PKE é constituído pelos processos de geração de chaves, encriptação e decriptação conforme especificado nos Algoritmos 7, 8 e 9 respectivamente.

Algoritmo 7 K-PKE.KeyGen

Entrada: $d \in \mathbb{B}^{32}$.
Saída: chave de encriptação $ek_{PKE} \in \mathbb{B}^{384k+32}$.
Saída: chave de decriptação $dk_{PKE} \in \mathbb{B}^{384k}$.

$(\rho, \sigma) \leftarrow G(d\|k)$ ▷ expande para duas sementes pseudo-aleatórias de 32 bytes
$N \leftarrow 0$
for $i = 0$ **to** $k-1$ **do** ▷ gera a matriz $\hat{\boldsymbol{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
  for $j = 0$ to $k-1$ **do**
    $\hat{\boldsymbol{A}}[i,j] \leftarrow \text{SampleNTT}(\rho\|j\|i)$ ▷ j e i são os bytes 33 e 34 da entrada
  end for
end for
for $i = 0$ **to** $k-1$ **do** ▷ gera $s \in (\mathbb{Z}_q^{256})^k$
  $s[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$ ▷ $s[i] \in \mathbb{Z}_q^{256}$ amostrado de CBD
  $N \leftarrow N + 1$
end for
for $i = 0$ to $k-1$ **do**
  $e[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$ ▷ $e[i] \in \mathbb{Z}_q^{256}$ amostrado de CBD
  $N \leftarrow N + 1$
end for
$\hat{s} \leftarrow \text{NTT}(s)$ ▷ NTT é executado k vezes (uma para cada coordenada de s)
$\hat{e} \leftarrow \text{NTT}(e)$ ▷ NTT é executado k vezes
$\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{A}} \circ \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}$ ▷ sistema linear ruidoso no domínio NTT
$ek_{PKE} \leftarrow \text{ByteEncode}_{12}(\hat{t})\|\rho$ ▷ ByteEncode_{12} é executado k vezes; inclui semente para $\hat{\boldsymbol{A}}$
$dk_{PKE} \leftarrow \text{ByteEncode}_{12}(\hat{s})$ ▷ ByteEncode_{12} é executado k vezes
return (ek_{PKE}, dk_{PKE})

No algoritmo de geração de chaves, a escolha do conjunto de parâmetros afeta o comprimento do segredo s (através do parâmetro k) e, consequentemente, os tamanhos do vetor de ruído e e da matriz pseudo-aleatória $\hat{\boldsymbol{A}}$. A escolha do conjunto de parâmetros também afeta a distribuição do ruído (através do parâmetro η_1) usada para amostrar as entradas de s e e.

A matriz $\hat{\boldsymbol{A}}$ contém os coeficientes das equações lineares, enquanto o vetor t representa o resultado dessas equações após a introdução do ruído e. O vetor t é calculado como

$t = As + e$. Este conjunto de elementos forma a chave de encriptação que é concatenada a uma semente de 32 bytes. Essa semente viabiliza que $\hat{\boldsymbol{A}}$ seja recalculada de forma determinística e, assim, o tamanho da chave pode ser reduzido. Além disso, essa semente pública, é expandida em uma matriz de inteiros módulo 3329: no total, 2 x 512 inteiros para o ML-KEM-512, 3 x 768 inteiros para o ML-KEM-768 ou 4 x 1024 inteiros para o ML-KEM-1024, ou seja, de acordo com o parâmetro k.

Por outro lado, a chave de decriptação é o vetor s de comprimento k, composto por elementos de R_q. De forma simplificada, $s \in R_q^k$ é o vetor de variáveis secretas. Este vetor é utilizado para resolver as equações lineares ruidosas da chave de encriptação, permitindo a recuperação da mensagem original.

A chave de encriptação do K-PKE servirá como a chave de encapsulamento do ML-KEM e, portanto, pode ser tornada pública. Entretanto, a chave de decriptação e a aleatoriedade do `K-PKE.KeyGen` devem permanecer privadas, pois podem ser usadas para realizar o desencapsulamento no ML-KEM.

Algoritmo 8 K-PKE.Encrypt(ek_{PKE}, m, r)

Entrada: chave de encriptação $ek_{PKE} \in \mathbb{B}^{384k+32}$.
Entrada: mensagem m $\in \mathbb{B}^{32}$.
Entrada: aleatoriedade r $\in \mathbb{B}^{32}$.
Saída: texto cifrado c $\in \mathbb{B}^{32(du+dv)}$.

$N \leftarrow 0$
$\hat{t} \leftarrow \text{ByteDecode}_{12}(ek_{PKE}[0 : 384k])$
$\rho \leftarrow ek_{PKE}[384k : 384k + 32]$ ▷ Extrai a semente de 32 bytes de ek_{PKE}
for $i = 0$ **to** $k - 1$ **do** ▷ Regenera a matriz $\hat{\boldsymbol{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$
    for $j = 0$ **to** $k - 1$ **do**
        $\hat{\boldsymbol{A}}[i, j] \leftarrow \text{SampleNTT}(\rho \| j \| i)$
    end for
end for
for $i = 0$ **to** $k - 1$ **do** ▷ Gera y $\in (\mathbb{Z}_q^{256})^k$
    $y[i] \leftarrow \text{SamplePolyCBD}\eta_1(\text{PRF}\eta_1(r, N))$ ▷ $y[i] \in \mathbb{Z}_q^{256}$ amostrado de CBD
    $N \leftarrow N + 1$
end for
for $i = 0$ **to** $k - 1$ **do** ▷ Gera $e_1 \in (\mathbb{Z}_q^{256})^k$
    $e_1[i] \leftarrow \text{SamplePolyCBD}\eta_2(\text{PRF}\eta_2(r, N))$ ▷ $e_1[i] \in \mathbb{Z}_q^{256}$ amostrado de CBD
    $N \leftarrow N + 1$
end for
$e_2 \leftarrow \text{SamplePolyCBD}\eta_2(\text{PRF}\eta_2(r, N))$ ▷ Amostra $e_2 \in \mathbb{Z}_q^{256}$ de CBD
$\hat{\boldsymbol{y}} \leftarrow \text{NTT}(\boldsymbol{y})$ ▷ NTT é executado k vezes
$\boldsymbol{u} \leftarrow \text{NTT}^{-1}(\hat{\boldsymbol{A}}^\top \circ \hat{\boldsymbol{y}}) + \boldsymbol{e_1}$ ▷ NTT^{-1} é executado k vezes
$\mu \leftarrow \text{Decompress}_1(\text{ByteDecode}_1(m))$
$v \leftarrow \text{NTT}^{-1}(\hat{\boldsymbol{t}}^\top \circ \hat{\boldsymbol{y}}) + e_2 + \mu$ ▷ Codifica a mensagem de texto claro m no polinômio v
$c_1 \leftarrow \text{ByteEncode}_{du}(\text{Compress}_{du}(\boldsymbol{u}))$ ▷ ByteEncode_{du} é executado k vezes
$c_2 \leftarrow \text{ByteEncode}_{dv}(\text{Compress}_{dv}(v))$
return $c \leftarrow (c_1 \| c_2)$

No algoritmo de encriptação (Algoritmo 8), procede-se inicialmente a extração do vetor t e da semente associada à chave. Em seguida, a semente é expandida para reconstruir

a matriz $\hat{A}$, replicando o procedimento utilizado durante a geração de chaves. A correta derivação de t e $\hat{A}$ de uma chave de encriptação gerada pelo `K-PKE.KeyGen` assegura que estes elementos sejam consistentes com os valores originariamente estabelecidos.

Esse algoritmo também faz uso de um vetor $y \in R_q^k$ e de termos de ruído $e_1 \in R_q^k$ e $e_2 \in R_q$, obtidos a partir da distribuição binomial centrada, com a pseudo-aleatoriedade derivada de r empregando uma PRF. A subsequente equação ruidosa é formulada como $(\boldsymbol{u}, v) \leftarrow (A^\top y + e_1, t^\top y + e_2)$, e a mensagem m é incorporada ajustando-se o termo $t^\top y + e_2$ com a codificação μ. Conclui-se o processo com a compressão e serialização do par $(\boldsymbol{u}, v)$ em bytes, culminando na produção do criptograma.

Algoritmo 9 K-PKE.Decrypt(dk_{PKE}, c)

Entrada: chave privada $dk_{PKE} \in \mathbb{B}^{384k}$.
Entrada: texto cifrado $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Saída: mensagem $m \in \mathbb{B}^{32}$.

1: $c_1 \leftarrow c[0 : 32d_u k]$
2: $c_2 \leftarrow c[32d_u k : 32(d_u k + d_v)]$
3: $\mathbf{u'} \leftarrow \text{Decompress}_{d_u}(\text{ByteDecode}_{d_u}(c_1))$ ▷ ByteDecode_{d_u} executado k vezes
4: $v' \leftarrow \text{Decompress}_{d_v}(\text{ByteDecode}_{d_v}(c_2))$
5: $\hat{\mathbf{s}} \leftarrow \text{ByteDecode}_{12}(dk_{PKE})$
6: $w \leftarrow v' - \text{NTT}^{-1}(\hat{\boldsymbol{s}}^\top \circ \text{NTT}(\boldsymbol{u'}))$ ▷ NTT executado k vezes e NTT^{-1} uma
7: $m \leftarrow \text{ByteEncode}_1(\text{Compress}_1(w))$ ▷ Decodifica o texto claro m do polinômio v
8: **return** m

Por fim, o procedimento para a decriptação exposto no algoritmo `K-PKE.Decrypt` (Algoritmo 9) inicia com a reconstituição da equação perturbada $(\boldsymbol{u}, v)$, que é a base do criptograma c. Neste contexto, interpreta-se $\boldsymbol{u}$ como os coeficientes da equação e v, o termo constante. É importante recordar que a chave de decriptação, dk_{PKE}, incorpora o vetor de variáveis secretas s. Utilizando-se desta chave, o algoritmo de decriptação estabelece o termo constante real $v' = s^\top u'$ e procede ao cálculo de $v' - v$. O processo de decriptação é finalizado com a extração da mensagem clara m a partir de $v' - v$.

3.3 Algoritmos Internos

A transição do rascunho para a versão final do FIPS 203 trouxe uma mudança na arquitetura do ML-KEM, introduzindo três algoritmos internos: `ML-KEM.KeyGen_internal`, `ML-KEM.Encaps_internal` e `ML-KEM.Decaps_internal`, respectivamente Algoritmos 10, 11 e 12. Estes algoritmos atuam como interfaces intermediárias para os algoritmos principais de geração de chaves, encapsulamento e desencapsulamento, melhorando aspectos tanto do ponto de vista da segurança quanto da engenharia de software.

Do ponto de vista da segurança, a nova estrutura facilita a verificabilidade e testabilidade do sistema. Ao definir interfaces claras, cada uma dessas operações pode ser testada e validada de forma independente. Isso se alinha com os requisitos do Programa de Validação de Algoritmo Criptográfico do NIST, onde a capacidade de isolar e testar componentes individuais é essencial para a certificação. Além disso, como esses algoritmos

internos são determinísticos, suas saídas podem ser previsivelmente comparadas contra valores esperados, simplificando o processo de verificação formal e análise de corretude.

Em termos de engenharia de software, a modularidade introduzida facilita a manutenção e a evolução do código. Ao isolar a lógica complexa e detalhada em subrotinas internas, é possível modificar ou otimizar os componentes de forma independente, sem impacto direto nas interfaces externas. Isso é particularmente importante, uma vez que a flexibilidade para ajustar componentes internos, em resposta a novos ataques ou descobertas, é imperioso para a longevidade do esquema.

Uma das mudanças mais importantes foi a introdução de verificações de erros e a separação de responsabilidades dentro do processo de geração de chaves. Na versão final (Algoritmo 10), a geração de chaves envolve duas fontes de entropia (d e z), ambas verificadas quanto à sua integridade antes das etapas subsequentes. Essa verificação prévia assegura que o sistema não continue a execução com valores inválidos, um passo essencial para prevenir falhas de segurança que poderiam surgir de entradas corrompidas ou geradores de entropia não especificados.

Além disso, a introdução de `ML-KEM.KeyGen_internal` isola a complexidade da geração de chaves, protegendo a interface externa contra manipulações e reduzindo a superfície de ataque.

Algoritmo 10 ML-KEM.KeyGen_internal(d, z)

Entrada: aleatoriedade $d \in \mathbb{B}^{32}$.
Entrada: aleatoriedade $z \in \mathbb{B}^{32}$.
Saída: chave de encapsulamento $ek \in \mathbb{B}^{384k+32}$.
Saída: chave de desencapsulamento $dk \in \mathbb{B}^{768k+96}$.
1: $(ek_{PKE}, dk_{PKE}) \leftarrow$ K-PKE.KeyGen(d) ▷ executa a geração de chaves para K-PKE
2: $ek \leftarrow ek_{PKE}$ ▷ A chave de encapsulamento é apenas a chave de encriptação PKE
3: $dk \leftarrow (dk_{PKE} \| ek \| H(ek) \| z)$ ▷ A chave de desencapsulamento KEM inclui a chave de decriptação PKE
4: **return** (ek, dk)

O algoritmo `ML-KEM.Encaps_internal` (Algoritmo 11) apresenta etapas para assegurar a confidencialidade durante a produção de uma chave secreta compartilhada e de um criptograma, ambos derivados de um valor aleatório m e da chave de encapsulamento ek. Isso é alcançado por meio da aplicação sequencial das funções de resumo criptográfico G e H. Especificamente, H é utilizada para processar a chave de encapsulamento ek, transformando-a em uma entrada padronizada para G, que então combina esse resultado com m para gerar a chave secreta K e a aleatoriedade r. Esta metodologia fortalece a integridade e a imprevisibilidade dos dados.

Por fim, o algoritmo `ML-KEM.Decaps_internal` (Algoritmo 12) realiza o processo de desencapsulamento da chave compartilhada com mecanismos de segurança para lidar com possíveis adulterações ou ataques. Inicialmente, são extraídas as partes necessárias da chave de desencapsulamento para realizar a decriptação do texto cifrado.

Em seguida, a função G é empregada para gerar um valor aleatório r' a partir da combinação da mensagem decriptada m' e do hash da chave de encapsulamento h, que é utilizado na reencriptação da mensagem para verificar a consistência do texto cifrado.

Algoritmo 11 ML-KEM.Encaps_internal(ek, m)

Entrada: chave de encapsulamento $ek \in \mathbb{B}^{384k+32}$.
Entrada: aleatoriedade $m \in \mathbb{B}^{32}$.
Saída: chave secreta compartilhada $K \in \mathbb{B}^{32}$.
Saída: texto cifrado $c \in \mathbb{B}^{32(d_u k+d_v)}$.
1: $(K, r) \leftarrow G(m\|H(ek))$ ▷ deriva a chave secreta compartilhada K e a aleatoriedade r
2: $c \leftarrow$ K-PKE.Encrypt(ek, m, r) ▷ encripta m usando K-PKE com a aleatoriedade r
3: **return** (K, c)

Esse procedimento é empregado para proteção contra ataques que tentam explorar vulnerabilidades na previsibilidade do comportamento do algoritmo.

A função J, por sua vez, é usada para derivar a chave compartilhada final $\bar{K}$ a partir da combinação do texto cifrado e do valor aleatório z. Essa estratégia garante que, mesmo na falha da correspondência dos textos cifrados, a chave compartilhada gerada seja imprevisível, fortalecendo a robustez do sistema contra ataques de substituição de texto cifrado.

Um outro mecanismo importante para o desencapsulamento é o mecanismo de rejeição implícita. Ele permite manejar discrepâncias entre o texto cifrado recebido e o texto cifrado recriado sem expor vulnerabilidades. Se os textos cifrados não coincidirem, o algoritmo ajusta a chave compartilhada para um hash do texto cifrado e do valor aleatório z, ao invés de falhar abertamente. Este procedimento impede que um adversário obtenha informações sobre a chave ou o estado interno do sistema através de tentativas de manipulação de entrada.

Tais aprimoramentos na arquitetura do esquema ML-KEM, não só aumentam a segurança e a testabilidade como garantem a adaptabilidade frente às evoluções tecnológicas e de ameaças, viabilizando a manutenção da eficácia do esquema ao longo do tempo.

Algoritmo 12 ML-KEM.Decaps_internal(dk, c)

Entrada: chave de desencapsulamento $dk \in \mathbb{B}^{768k+96}$.
Entrada: texto cifrado $c \in \mathbb{B}^{32(d_u k+d_v)}$.
Saída: chave secreta compartilhada $K \in \mathbb{B}^{32}$.
1: $dk_{PKE} \leftarrow \text{dk}[0 : 384k]$ ▷ extrai (da chave de desencapsulamento KEM) a chave de decriptação PKE
2: $ek_{PKE} \leftarrow \text{dk}[384k : 768k + 32]$ ▷ extrai a chave de encriptação PKE
3: $h \leftarrow \text{dk}[768k + 32 : 768k + 64]$ ▷ extrai o hash da chave de encriptação PKE
4: $z \leftarrow \text{dk}[768k + 64 : 768k + 96]$ ▷ extrai o valor de rejeição implícita
5: $m' \leftarrow$ K-PKE.Decrypt(dk_{PKE}, c) ▷ decripta o texto cifrado
6: $(K', r') \leftarrow G(m'\|h)$
7: $\bar{K} \leftarrow J(z\|c)$
8: $c' \leftarrow$ K-PKE.Encrypt(ek_{PKE}, m', r') ▷ reencripta usando a aleatoriedade derivada r'
9: **if** $c \neq c'$ **then**
10: $K' \leftarrow \bar{K}$ ▷ se os textos cifrados não coincidem, "rejeitar implicitamente"
11: **end if**
12: **return** K'

3.4 Algoritmos ML-KEM

Os algoritmos principais do esquema ML-KEM são: Geração de Chaves (`ML-KEM.KeyGen`), Encapsulamento (`ML-KEM.Encaps`) e Desencapsulamento (`ML-KEM.Decaps`). Cada um desses é apresentado a seguir conforme Algoritmos 13, 14 e 15 respectivamente.

Algoritmo 13 ML-KEM.KeyGen()

Saída : chave de encapsulamento $ek \in \mathbb{B}^{384k+32}$
Saída : chave de desencapsulamento $dk \in \mathbb{B}^{768k+96}$
1: $d \leftarrow \mathbb{B}^{32}$ ▷ d são 32 bytes aleatórios
2: $z \leftarrow \mathbb{B}^{32}$ ▷ z são 32 bytes aleatórios
3: **if** $d ==$ NULL or $z ==$ NULL **then**
4: **return** $\perp$ ▷ retorna uma indicação de erro se a geração de bits aleatórios falhar
5: **end if**
6: $(ek, dk) \leftarrow$ ML-KEM.KeyGen_internal(d, z) ▷ executa o algoritmo interno
7: **return** (ek, dk)

O algoritmo de geração de chaves `ML-KEM.KeyGen` produz uma chave de encapsulamento e uma chave de desencapsulamento. Enquanto a chave de encapsulamento pode ser tornada pública, a chave de desencapsulamento deve permanecer privada. A chave de encapsulamento ML-KEM é simplesmente a chave de encriptação do K-PKE. A chave de desencapsulamento ML-KEM é composta pela chave de decriptação do K-PKE, a chave de encapsulamento, um hash da chave de encapsulamento e um valor pseudo-aleatório de 32 bytes. Este valor será utilizado no mecanismo de rejeição implícita do algoritmo de desencapsulamento (Algoritmo 15). As sementes (d, z), geradas nos passos 1 e 2, podem ser armazenadas para expansão posterior usando `ML-KEM.KeyGen_internal`. Como essas sementes podem ser usadas para computar a chave de desencapsulamento, elas são consideradas dados sensíveis e devem ser tratadas com as mesmas salvaguardas aplicadas à chave de desencapsulamento, conforme especificado em [38].

Algoritmo 14 ML-KEM.Encaps(ek)

Entrada validada : chave de encapsulamento $ek \in \mathbb{B}^{384k+32}$.
Saída : chave secreta compartilhada $K \in \mathbb{B}^{32}$.
Saída : texto cifrado $c \in \mathbb{B}^{32(duk+dv)}$.
1: $m \leftarrow \mathbb{B}^{32}$ ▷ m são 32 bytes aleatórios
2: **if** $m ==$ NULL **then**
3: **return** $\perp$ ▷ retorna uma indicação de erro se a geração de bits aleatórios falhar
4: **end if**
5: $(K, c) \leftarrow$ ML-KEM.Encaps_internal(ek, m) ▷ executa o algoritmo interno
6: **return** (K, c)

O algoritmo de encapsulamento `ML-KEM.Encaps`, fundamenta-se no mecanismo de encriptação descrito no Algoritmo 8 para gerar o texto cifrado e a chave compartilhada. Neste processo, Bob, ao utilizar `ML-KEM.Encaps`, aceita uma chave de encapsulamento como entrada e invoca o algoritmo interno correspondente. Este, por sua vez, realiza a chamada de `K-PKE.Encrypt` para gerar um texto cifrado e a chave secreta compartilhada.

A validação da entrada é crítica para assegurar a integridade e a segurança das operações. As verificações incluem a conformidade do tamanho do vetor de bytes da chave de encapsulamento com os parâmetros especificados e a consistência dos dados após operações de codificação e decodificação.

Algoritmo 15 ML-KEM.Decaps(c, dk)

Entrada validada: texto cifrado $c \in \mathbb{B}^{32(d_u k+d_v)}$.
Entrada validada: chave de desencapsulamento $dk \in \mathbb{B}^{768k+96}$.
Saída: chave compartilhada $K \in \mathbb{B}^{32}$.
1: $K' \leftarrow$ ML-KEM.Decaps_internal(dk, c)
2: **return** K'

O processo de desencapsulamento especificado no Algoritmo 15 emprega seu correspondente interno com entradas validadas para produzir a chave compartilhada. A validação do texto cifrado deve ser realizada a cada execução do algoritmo. Por outro lado, para a chave de desencapsulamento, é possível assumir a validade caso seja obedecido o procedimento definido em [38], que detalha métodos específicos para a garantia de segurança de um KEM.

Embora extremamente improvável, é possível que ocorra falha no processo no qual Bob e Alice tentam estabelecer uma chave compartilhada mesmo que ambos estejam agindo honestamente e não haja interferência adversarial. Isto é, Alice e Bob empregando, respectivamente, os Algoritmos 15 e 14 geram saídas distintas. A probabilidade de que tal falha ocorra é apresentada na Tabela 3.2.

Variante	**Taxa de falha de desencapsulamento**
ML-KEM-512	$2^{-138.8}$
ML-KEM-768	$2^{-164.8}$
ML-KEM-1024	$2^{-174.8}$

Tabela 3.2: Taxa de falha de desencapsulamento ML-KEM

Além dessa consideração, é essencial conhecer também os tamanhos das chaves e do texto cifrado para seleção adequada do compromisso entre segurança e performance de acordo com os diferentes cenários de aplicação. A Tabela 3.3 reúne essas informações para as três variantes do esquema.

	Encaps	**Decaps**	**Chave compartilhada**	**Texto cifrado**
ML-KEM-512	800	1632	32	768
ML-KEM-768	1184	2400	32	1088
ML-KEM-1024	1568	3168	32	1568

Tabela 3.3: Tamanhos das chaves e do texto cifrado (bytes)

Capítulo 4

CRYSTALS-Dilithium

Um esquema de assinatura digital oferece um análogo criptográfico às assinaturas manuscritas que, de fato, fornece garantias de segurança substancialmente mais robustas. Atualmente reconhecidas como juridicamente vinculativas em diversos países, as assinaturas digitais emergem como uma ferramenta poderosa, sendo empregadas na certificação de contratos ou no notariado de documentos para autenticação de indivíduos ou corporações e, ainda, como elementos essenciais de protocolos mais complexos [23]. Em linhas gerais, um esquema de assinatura digital envolve três etapas essenciais:

1. **Geração de Chaves**: Nesta fase, é gerado um par de chaves matematicamente relacionadas, consistindo de uma chave pública e uma privada. A chave pública pode ser divulgada amplamente, enquanto a chave privada deve ser mantida em sigilo pelo emissor.

2. **Assinatura**: O emissor utiliza sua chave privada para assinar a mensagem. Essa assinatura é realizada de modo a vincular a mensagem à chave privada de forma segura, utilizando processos que garantem a integridade e a autenticidade da mensagem.

3. **Verificação**: Qualquer parte que receba a mensagem e a respectiva assinatura pode verificar a autenticidade utilizando a chave pública do emissor. Este processo confirma que a mensagem foi assinada pela chave privada correspondente à chave pública e não sofreu alteração.

A implementação prática dessas etapas pode ser facilitada pelo uso de protocolos criptográficos interativos ou não interativos.

Um protocolo interativo envolve várias rodadas de comunicação entre um provador e um verificador. O provador inicia enviando um compromisso, uma representação cifrada de uma informação. O verificador então propõe um desafio, solicitando ao provador que demonstre conhecimento da informação cifrada através de uma resposta que valida o compromisso sem revelar a informação subjacente. O sucesso do protocolo depende da precisão da resposta do provador aos desafios do verificador.

Em contraste, um protocolo não interativo elimina a necessidade de qualquer interação direta entre o provador e o verificador durante o processo de verificação. Ao invés disso,

o provador gera uma única mensagem que contém todas as informações necessárias para que o verificador possa validar a afirmação.

A transformação de um protocolo interativo em um não interativo permite que, em um esquema de assinatura, o processo de verificação possa ser realizado sem a presença contínua do signatário. Essa transformação é essencial para a implementação prática, pois simplifica significativamente a verificação e expande a aplicabilidade em situações nas quais a comunicação direta com o signatário não é viável

Uma das formas de realizar essa transformação é empregando a heurística de *Fiat-Shamir* [29], que substitui o desafio gerado interativamente por um valor determinístico, derivado de um hash do compromisso e da mensagem a ser assinada. Este processo captura a essência do protocolo interativo — compromisso, desafio e resposta — integrando essa lógica em uma função autônoma.

Diversos esquemas de assinatura digital pós-quânticos adotam esses princípios, dentre eles, o CRYSTALS-Dilithium cuja segurança se baseia no problema MLWE e em uma variante do problema MSIS [16]. O esquema compartilha características com a proposta de Schnorr [47], uma vez que ambos se baseiam na ideia de demonstrar conhecimento de uma chave privada sem revelá-la.

A versão 3.1 do CRYSTALS-Dilithium foi utilizada pelo NIST para estabelecer o padrão FIPS 204 - Module-Lattice-Based Digital Signature Standard (ML-DSA) cujos parâmetros são apresentados na próxima seção. O esquema possui três variantes, a saber ML-DSA-44, ML-DSA-65 e ML-DSA-87 voltadas para atender os níveis de segurança equivalentes às categorias 2, 3 e 5 respectivamente.

4.1 Conjunto de Parâmetros

A Tabela 4.1 apresenta os três conjuntos de parâmetros do ML-DSA, cada um especificando valores utilizados nos algoritmos de geração de chaves, assinatura e verificação. Os nomes dos conjuntos de parâmetros têm o formato ML-DSA-kℓ, onde k,ℓ são as dimensões da matriz $\hat{\boldsymbol{A}}$, que faz parte da chave pública.

- q - o parâmetro q representa o módulo usado em várias operações do esquema. Para todas as variantes, o valor de q é 8380417 ($2^{23} - 2^{13} + 1$), definindo o corpo finito sobre o qual os polinômios e outras estruturas são estabelecidas.

- d - indica o número de bits menos significativos que são descartados de t, um parâmetro relacionado ao tamanho da chave pública.

- τ - determina o número de coeficientes que são definidos como ± 1 no polinômio c, usado na geração de assinaturas.

- λ - mede a resistência à colisão do polinômio modificado $\tilde{c}$, representando a força contra ataques que buscam encontrar duas mensagens diferentes que resultem na mesma assinatura.

- γ_1 - define o intervalo de valores para os coeficientes do vetor y, usado na operação de assinatura.

Parâmetros	**ML-DSA-44**	**ML-DSA-65**	**ML-DSA-87**
q - módulo	8380417	8380417	8380417
d - # de bits descartados de t	13	13	13
τ - # de ± 1 em polinômio c	39	49	60
λ - força de colisão de $\tilde{c}$	128	192	256
γ_1 - intervalo de coeficiente de y	2^{17}	2^{19}	2^{19}
γ_2 - intervalo de arredondamento de baixa ordem	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
(k, ℓ) - dimensões de $\boldsymbol{A}$	(4,4)	(6,5)	(8,7)
η - intervalo da chave privada	2	4	2
$\beta = \tau \cdot \eta$	78	196	120
ω - máx # de 1's na dica h	80	55	75
Entropia do desafio $\log \tau + \tau$	192	225	257
Repetições	4.25	5.1	3.85
Categoria de segurança	2	3	5

Tabela 4.1: Conjunto de Parâmetros do ML-DSA

- γ_2 **-** especifica a faixa de arredondamento a ser utilizada.
- (k, ℓ) - estes parâmetros definem as dimensões da matriz $\boldsymbol{A}$ usada nas operações de geração de chaves, assinatura e verificação.
- η **-** indica o intervalo de valores para a chave privada.
- β **-** representa uma métrica que combina o número de coeficientes específicos τ no polinômio e a amplitude dos valores η associados à chave privada. Essa combinação impacta diretamente a robustez e a segurança das assinaturas geradas.
- ω **-** define o número máximo de bits 1 na dica h, sendo usado para auxiliar na verificação da assinatura.
- *entropia do desafio* - quantifica a aleatoriedade e a complexidade associadas à geração de assinaturas.
- *repetições* - número de repetições esperado no laço principal do algoritmo de assinatura.
- *categoria de segurança* - classifica o nível de segurança associado com base nas categorias definidas pelo NIST em sua chamada original de propostas [30]. Mais especificamente, alega-se que os recursos computacionais necessários para comprometer o ML-DSA são maiores ou iguais aos necessários para quebrar um cifrador de bloco ou função de hash equivalente, quando esses recursos são estimados usando um modelo de computação realista.

Os tamanhos das chaves e das assinaturas correspondentes a cada um dos conjuntos de parâmetros apresentados anteriormente estão descritos na Tabela 4.2, facilitando a comparação entre as diferentes configurações e suas respectivas implicações na eficiência e segurança do sistema.

	ML-DSA-44	**ML-DSA-65**	**ML-DSA-87**
Chave Privada	2528	4000	4864
Chave Pública	1312	1952	2592
Tamanho da Assinatura	2420	3293	4595

Tabela 4.2: Tamanhos (em bytes) das chaves e assinaturas do ML-DSA

As etapas de construção do esquema, que serão apresentadas na próxima seção, nos permitem visualizar os conceitos discutidos até o momento e a aplicação do problema MLWE em resposta aos desafios impostos por adversários com capacidades quânticas.

4.2 Construção do ML-DSA

A construção do ML-DSA é baseada em um protocolo semelhante ao de Schnorr no qual um provador que conhece $\boldsymbol{A} \in \mathbb{Z}_q^{KxL}, \boldsymbol{S_1} \in \mathbb{Z}_q^{Lxn}$ e $\boldsymbol{S_2} \in \mathbb{Z}_q^{Kxn}$ demonstra conhecimento dessas matrizes a um verificador que conhece $\boldsymbol{A}$ e $\boldsymbol{T} \in \mathbb{Z}_q^{Kxn}$. O protocolo é estruturado inicialmente da seguinte forma:

1. **Compromisso:** O provador gera um vetor $y \in \mathbb{Z}_q^L$ com coeficientes pequenos e envia um compromisso ao verificador. Este compromisso é calculado como $w_{\text{Approx}} = Ay + y_2$, onde $y_2 \in \mathbb{Z}_q^K$ é um vetor com coeficientes pequenos. Este cálculo aponta para o fundamento do esquema na dificuldade do problema MLWE.

2. **Desafio:** Após receber o compromisso, o verificador envia de volta ao provador um vetor $c \in \mathbb{Z}_q^n$ com coeficientes pequenos. Este vetor serve como desafio, exigindo que o provador demonstre conhecimento da matriz $\boldsymbol{S_1}$ sem revelar diretamente as informações sensíveis.

3. **Resposta:** O provador calcula a resposta $z = y + S_1 c$ e a envia de volta ao verificador. O verificador então confirma a validade da resposta verificando se $Az - Tc \approx w_{\text{Approx}}$, onde $\boldsymbol{T} = \boldsymbol{AS_1} + \boldsymbol{S_2}$. A precisão da resposta indica que o provador conhece as matrizes envolvidas e pode manipulá-las corretamente conforme o desafio recebido.

No entanto, ao converter esse protocolo interativo em um esquema de assinatura, surgem questões de viés, onde a resposta z é enviesada em uma direção relacionada ao valor privado $\boldsymbol{S_1}$, e o termo $r = w_{\text{Approx}} - Az + Tc$ é enviesado em uma direção relacionada ao valor privado $\boldsymbol{S_2}$.

Para corrigir esses vieses e transformar o protocolo em um esquema de assinatura seguro é realizada uma modificação na etapa do desafio, que passa a utilizar um hash do compromisso concatenado à mensagem, e é incluído um procedimento de rejeição por meio

do qual o signatário persegue um valor adequado de z; se os coeficientes de z estiverem fora de uma faixa especificada, o processo de assinatura é abortado e o assinante começa novamente a partir de um novo valor de y. Essas verificações são análogas às realizadas na Linha 21 do Algoritmo 20. Nesse contexto, a chave pública é $(\boldsymbol{A}, \boldsymbol{T})$, e a chave privada é $(\boldsymbol{S_1}, \boldsymbol{S_2})$.

Por razões de segurança ou eficiência, o padrão incorpora várias modificações e ajustes que são adicionados a este quadro básico. Tais ajustes incluem procedimentos para reduzir o tamanho das chaves e da assinatura, bem como para a codificação, conversão e compressão dos dados, os quais serão abordados ao longo das descrições dos algoritmos nas próximas seções.

As funções de assinatura, verificação e geração de chaves podem ser divididas em componentes externos e internos. Esta divisão simplifica as APIs e os testes do Programa de Validação de Algoritmos Criptográficos do NIST. Os componentes externos são responsáveis por gerar aleatoriedade e realizar várias verificações antes de invocar suas contrapartes internas. Já os componentes internos são determinísticos e operam sob a premissa de que os componentes externos não encontraram condições de erro.

4.3 Algoritmos Externos

Os componentes externos do ML-DSA (Algoritmos 16, 17 e 18), que incluem as funções de geração de chaves, assinatura e verificação, atuam como camadas de abstração que simplificam a arquitetura. Essas interfaces gerenciam as tarefas de geração de aleatoriedade e de validação de entradas, garantindo que apenas dados confiáveis sejam transmitidos para os componentes internos. Ademais, as interfaces externas não apenas reforçam a segurança contra falhas e ataques potenciais, mas também promovem a clareza, a manutenibilidade e a simplicidade para o processo de auditoria do código.

4.3.1 Geração de Chaves

O algoritmo de geração de chaves `ML-DSA.KeyGen` não recebe nenhuma entrada e tem como saída uma chave pública (pk) e uma chave privada (sk), ambas codificadas como strings de bytes. A geração dessas chaves é fundamentalmente ancorada no uso de um Gerador de Bits Aleatórios (RBG) aprovado pelo NIST [3] [52] [4] para gerar uma semente aleatória de 256 bits (32 bytes), que é fornecida como entrada para `ML-DSA.KeyGen_internal` (Algoritmo 19), responsável por produzir as chaves.

Algoritmo 16 ML-DSA.KeyGen()

Saída: $pk \in \mathbb{B}^{32+32k(bitlen(q-1)-d)}$ e $sk \in \mathbb{B}^{32+32+64+32\cdot((l+k)\cdot bitlen(2\eta)+dk)}$.

1: $\xi \leftarrow \mathbb{B}^{32}$ ▷ escolhe uma semente aleatória
2: **if** $\xi =$ NULL **then**
3: **return** $\perp$ ▷ retorna uma indicação de erro se a geração de bits aleatórios falhar
4: **end if**
5: **return** ML-DSA.KeyGen_internal(ξ)

4.3.2 Assinatura

O algoritmo `ML-DSA.Sign` é responsável pelo processo de assinatura e recebe como entradas uma chave privada sk, uma mensagem M, e uma string de contexto ctx, que serve para especificar o domínio ou a finalidade da assinatura, garantindo que ela seja válida apenas no ambiente esperado e protegendo contra ambiguidades que poderiam ser exploradas por adversários. O comprimento da string de contexto ctx é verificado no início do algoritmo; se exceder 255 bytes, um erro ($\perp$) será retornado. Isso é necessário para evitar que dados de contexto excessivamente grandes sejam processados, o que poderia comprometer a segurança ou a eficiência da assinatura.

Em seguida, um valor aleatório (rnd) de 32 bytes é gerado para ser usado como um *nonce*[1]. Esse valor aleatório é fundamental para garantir que cada assinatura seja única, mesmo que a mesma mensagem seja assinada várias vezes. Na versão determinística opcional do algoritmo[2], esse valor aleatório é substituído por uma sequência de zeros de 32 bytes. Se a geração de bits aleatórios falhar, o algoritmo retorna um erro, assegurando que não haja assinaturas comprometidas por falta de entropia.

Após essas verificações e preparações, o algoritmo concatena um byte de valor zero, o comprimento do contexto, a string de contexto ctx, e a mensagem M em uma nova string M'. O byte de valor zero, gerado por `IntegerToBytes(0, 1)`, atua como uma espécie de marcador de início para indicar o começo da seção de dados relacionada ao contexto. Essa prática ajuda a prevenir ambiguidades e assegura uma separação clara dos domínios de dados na construção da assinatura. A string M' é então usada, junto com a chave privada e o valor aleatório, na função interna (`ML-DSA.Sign_internal`) para gerar a assinatura σ. A assinatura gerada σ é então retornada, concluindo o processo.

Algoritmo 17 ML-DSA.Sign(sk, M, ctx)

Entrada: chave privada $sk \in \mathbb{B}^{32+32+64+32\cdot((l+k)\cdot bitlen(2\eta)+dk)}$.
Entrada: mensagem $M \in \{0,1\}^*$.
Entrada: string de contexto ctx (string de bytes com no máximo 255 bytes).
Saída: Assinatura $\sigma \in \mathbb{B}^{\lambda/4+l\cdot 32\cdot(1+bitlen(\gamma_1-1))+\omega+k}$.

1: **if** $|ctx| > 255$ **then**
2: **return** $\perp$ ▷ retorna uma indicação de erro se a string de contexto for muito longa
3: **end if**
4: $rnd \leftarrow \mathbb{B}^{32}$ ▷ para a variante determinística opcional, substitua $rnd \leftarrow \{0\}^{32}$
5: **if** $rnd =$ NULL **then**
6: **return** $\perp$ ▷ retorna uma indicação de erro se a geração de bits aleatórios falhar
7: **end if**
8: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1)$ $\|$ IntegerToBytes$(|ctx|,1)$ $\|$ ctx) $\|$ M
9: $\sigma \leftarrow$ ML-DSA.Sign_internal(sk, M', rnd)
10: **return** σ

[1]Um *nonce* (do inglês "number used once") é um número ou valor arbitrário que é utilizado apenas uma vez em um processo criptográfico, garantindo que cada execução do processo seja única.

[2]Uma versão disponibilizada para plataformas onde a geração de valores aleatórios é inviável, embora o uso dessa versão aumente o risco de ataques por canal lateral.

4.3.3 Verificação

O algoritmo `ML-DSA.Verify` é utilizado para verificar a autenticidade de uma assinatura digital σ associada a uma mensagem M, utilizando uma chave pública pk e uma string de contexto ctx. Inicialmente, verifica-se o comprimento da string de contexto ctx; se o comprimento exceder 255 bytes, o algoritmo retorna um erro ($\perp$). Esta verificação inicial é importante para evitar o processamento de dados de contexto que possam comprometer a integridade do algoritmo ou causar falhas operacionais.

Após garantir que a string de contexto ctx seja válida, o algoritmo constrói uma nova mensagem M' que combina o contexto e a mensagem original de forma segura. O processo começa com a adição de um byte de valor zero, gerado por `IntegerToBytes`, que atua como um marcador de início para separar claramente os componentes de entrada. Em seguida, o comprimento do contexto, representado como um byte por `IntegerToBytes(|ctx|, 1)`, é concatenado ao próprio contexto ctx e à mensagem original M. A função `BytesToBits` é então utilizada para converter a concatenação resultante de bytes em uma sequência de bits, assegurando uma representação consistente.

Finalmente, o algoritmo chama a função interna `ML-DSA.Verify_internal`, que realiza a verificação efetiva da assinatura σ usando a chave pública pk e a mensagem modificada M'. Esta função verifica se a assinatura é válida para os dados fornecidos, considerando o contexto específico, e retorna um valor booleano: verdadeiro se a assinatura for válida e falso caso contrário.

Algoritmo 18 ML-DSA.Verify(pk, M, σ, ctx)

Entrada: chave pública $pk \in \mathbb{B}^{32+32k(bitlen(q-1)-d)}$.
Entrada: mensagem $M \in \{0,1\}^*$.
Entrada: assinatura $\sigma \in \mathbb{B}^{\lambda/4+l\cdot 32\cdot(1+bitlen(\gamma_1-1))+\omega+k}$.
Entrada: string de contexto ctx (string de bytes com no máximo 255 bytes).
Saída: Booleano.
1: **if** $|ctx| > 255$ **then**
2: **return** $\perp$ ▷ retorna uma indicação de erro se a string de contexto for muito longa
3: **end if**
4: $M' \leftarrow$ BytesToBits(IntegerToBytes$(0,1)$ ‖ IntegerToBytes$(|ctx|,1)$ ‖ ctx) ‖ M
5: **return** ML-DSA.Verify_internal(pk, M', σ)

4.4 Algoritmos Internos

Esta seção descreve as funções internas para geração de chaves, assinatura e verificação, as quais produzem suas saídas de forma determinística a partir das entradas validadas de suas contrapartes externas.

4.4.1 Geração de Chaves - ML-DSA Interno

O algoritmo `ML-DSA_internal.KeyGen` (Algoritmo 19) é responsável pela geração das chaves pública e privada. O processo tem início com a expansão da semente aleatória

de 32 bytes, ξ por meio da XOF (SHAKE256), denotada como H. Essa expansão gera três valores críticos: uma semente pública ρ, uma semente privada ρ', e uma semente adicional K utilizada no processo de assinatura. A semente pública ρ é usada para pseudo-aleatoriamente amostrar uma matriz polinomial $\hat{\boldsymbol{A}} \in R_q^{k \times \ell}$ no domínio NTT.

O algoritmo então prossegue para a amostragem dos vetores polinomiais secretos $s_1 \in R_q^{\ell}$ e $s_2 \in R_q^{k}$ usando a semente privada ρ'. Estes vetores são amostrados de forma que seus coeficientes sejam pequenos, ou seja, estejam dentro de um intervalo limitado $[-\eta, \eta]$, o que é fundamental para manter a eficiência e a segurança do esquema. A principal operação criptográfica ocorre ao computar o vetor público $t = As_1 + s_2$. Essa construção está diretamente relacionada ao problema MLWE. Especificamente, recuperar s_1 e s_2, que compõem a chave privada, a partir de A e t, é considerado computacionalmente inviável.

Para otimização, t é comprimido por meio da função `Power2Round`, que descarta os d bits menos significativos de cada coeficiente de t, resultando em dois componentes: t_1, o vetor polinomial comprimido, e t_0, que mantém os bits descartados. Essa compressão reduz o tamanho da chave pública sem comprometer a segurança, pois os bits de ordem inferior podem ser recuperados a partir de múltiplas assinaturas, mas não são necessários para a validação. Finalmente, o algoritmo constrói a chave pública pk como uma codificação da semente pública ρ e do vetor polinomial comprimido t_1. A chave privada sk inclui a semente pública ρ, a semente privada adicional K para uso durante a assinatura, um hash de 64 bytes da chave pública para segurança adicional durante a assinatura, os vetores polinomiais secretos s_1 e s_2, e o vetor polinomial t_0 que contém os bits menos significativos descartados em t_1.

Algoritmo 19 ML-DSA.KeyGen_internal(ξ)

Entrada: Semente $\xi \in \mathbb{B}^{32}$
Saída: Chave pública $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$,
Chave privada $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}(2\eta)+dk)}$

1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow H(\xi \parallel \text{IntegerToBytes}(k, 1) \parallel \text{IntegerToBytes}(\ell, 1), 128)$ ▷ Expande a semente
2: $A \leftarrow \text{ExpandA}(\rho)$ ▷ A é gerada e armazenada na representação NTT como $\hat{A}$
3: $(s_1, s_2) \leftarrow \text{ExpandS}(\rho')$
4: $t \leftarrow \text{NTT}^{-1}(A \circ \text{NTT}(s_1)) + s_2$ ▷ Calcula $t = As_1 + s_2$
5: $(t_1, t_0) \leftarrow \text{Power2Round}(t)$ ▷ Compressão de t
6: $pk \leftarrow \text{pkEncode}(\rho, t_1)$
7: $tr \leftarrow H(pk, 64)$
8: $sk \leftarrow \text{skEncode}(\rho, K, tr, s_1, s_2, t_0)$ ▷ K e tr são utilizados na assinatura
9: **return** (pk, sk)

4.4.2 Assinatura - ML-DSA Interno

O algoritmo `ML-DSA.Sign_internal` (Algoritmo 20) é um processo determinístico que tem como entrada uma chave privada sk, uma mensagem formatada M', e um valor de aleatoriedade específico rnd. Na fase de compromisso, a chave privada sk é decomposta em seus componentes constituintes $(\rho, K, tr, s_1, s_2, t_0)$ usando `skDecode`. As transformadas

dos vetores secretos s_1, s_2, e t_0 são calculadas usando a NTT visando multiplicações polinomiais eficientes. A matriz A, gerada a partir da semente pública ρ pela função `ExpandA`, é armazenada em sua forma NTT como $\hat{A}$. O algoritmo calcula um representante da mensagem μ aplicando a função de resumo criptográfico H ao hash da chave pública tr concatenado à mensagem formatada M'. Uma semente privada ρ'' é derivada da semente K, do valor de aleatoriedade rnd, e de μ.

Usando ρ'', um vetor polinomial aleatório y é amostrado da distribuição R_q^ℓ pela função `ExpandMask`, garantindo que cada coeficiente de y esteja no intervalo $[-\gamma_1 + 1, \gamma_1]$. Este vetor é então transformado para o domínio NTT para calcular $w = \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(y))$, e o compromisso w_1 é derivado aplicando `HighBits` em w, isolando seus bits mais significativos.

Na fase de desafio, o algoritmo gera um valor aleatório c usando `SampleInBall`, que amostra um polinômio cujos coeficientes possuem valores no conjunto $\{-1, 0, 1\}$ com peso de Hamming $\tau \leq 64$, a partir do hash do compromisso concatenado com μ e os bits significativos de w_1. Este polinômio c, representado como um polinômio em R_q, é transformado para o domínio NTT como $\hat{c}$. A multiplicação de $\hat{c}$ com os vetores secretos $\hat{s}_1$ e $\hat{s}_2$ no domínio NTT produz $\langle\langle cs_1 \rangle\rangle$ e $\langle\langle cs_2 \rangle\rangle$, respectivamente. Esta notação $\langle\langle \cdot \rangle\rangle$ indica que o resultado dessas multiplicações é armazenado como variáveis temporárias, permitindo seu uso repetido sem necessidade de recomputação. A estrutura matemática dessas operações assegura que a resposta a esse desafio dependa do conhecimento dos vetores secretos, reforçando a segurança contra ataques de forja existencial e seletiva.

Na fase de resposta, $z = y + \langle\langle cs_1 \rangle\rangle$ é calculada, e o ajuste residual $r_0 = \text{LowBits}(w - \langle\langle cs_2 \rangle\rangle)$ é computado para alinhar o compromisso e a resposta. Verificações de validade asseguram que $||z||_\infty$ e $||r_0||_\infty$ não excedam os limites de segurança $\gamma_1 - \beta$ e $\gamma_2 - \beta$. Se essas condições não forem atendidas, a assinatura é rejeitada, mantendo a robustez contra falhas de segurança. A função `MakeHint` é usada para calcular h, uma dica binária indicando se a adição de $-\langle\langle ct_0 \rangle\rangle$ a $w - \langle\langle cs_2 \rangle\rangle + \langle\langle ct_0 \rangle\rangle$ altera os bits mais significativos dos coeficientes mantidos após a compressão realizada pela função `Power2Round`. Esse mecanismo, além de assegurar a integridade da assinatura, permite a a recuperação correta dos dados.

Se o número de 1s em h for maior que ω ou outras condições falharem, a assinatura é rejeitada. Finalmente, a assinatura σ é codificada com `sigEncode` e retornada, encapsulando os valores de compromisso e resposta c, $z \mod \pm q$, e h, garantindo a integridade e autenticidade da assinatura.

4.4.3 Verificação - ML-DSA Interno

O algoritmo `ML-DSA.Verify_internal` (Algoritmo 21) é responsável por verificar a validade de uma assinatura digital σ para uma mensagem formatada M', utilizando a chave pública pk. O objetivo principal deste algoritmo é assegurar a autenticidade e integridade da assinatura, confirmando que ela foi gerada pelo detentor da chave privada correspondente. O processo inicia-se com a decodificação da chave pública pk e da assinatura σ através das funções `pkDecode` e `sigDecode`. A função `pkDecode` extrai a semente pública ρ e o vetor comprimido t_1, enquanto `sigDecode` recupera o hash de compromisso $\tilde{c}$, a resposta z, e a dica h. Se a dica h for inválida ($h = \perp$), o algoritmo retorna imediatamente

Algoritmo 20 ML-DSA.Sign_internal(sk, M', rnd)

Entrada: Chave privada $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen}(2\eta)+dk)}$, mensagem formatada $M' \in \{0,1\}^*$, e valor de aleatoriedade por mensagem ou variável dummy $rnd \in \mathbb{B}^{32}$.
Saída: Assinatura $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$.

$(\rho, K, tr, s_1, s_2, t_0) \leftarrow \text{skDecode}(sk)$
$\hat{s}_1 \leftarrow \text{NTT}(s_1)$
$\hat{s}_2 \leftarrow \text{NTT}(s_2)$
$\hat{t}_0 \leftarrow \text{NTT}(t_0)$
$\hat{A} \leftarrow \text{ExpandA}(\rho)$ ▷ A é gerada e armazenada na representação NTT como $\hat{A}$
$\mu \leftarrow H(\text{BytesToBits}(tr) \parallel M', 64)$ ▷ Representação da mensagem, opcionalmente calculada em outro módulo criptográfico
$\rho'' \leftarrow H(K \parallel rnd \parallel \mu, 64)$ ▷ Computa a semente aleatória privada
$\kappa \leftarrow 0$ ▷ Inicializa o contador κ
$(z, h) \leftarrow \perp$
while $(z, h) = \perp$ **do** ▷ Loop de amostragem de rejeição
    $y \in R_q^\ell \leftarrow \text{ExpandMask}(\rho'', \kappa)$
    $w \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(y))$
    $w_1 \leftarrow \text{HighBits}(w)$ ▷ Compromisso do assinante
    $\tilde{c} \leftarrow H(\mu \parallel \text{w1Encode}(w_1), \lambda/4)$ ▷ Hash do compromisso
    $c \in R_q \leftarrow \text{SampleInBall}(\tilde{c})$ ▷ Desafio do verificador
    $\hat{c} \leftarrow \text{NTT}(c)$
    $\langle cs_1 \rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{s}_1)$
    $\langle cs_2 \rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{s}_2)$
    $z \leftarrow y + \langle cs_1 \rangle$ ▷ Resposta do assinante
    $r_0 \leftarrow \text{LowBits}(w - \langle cs_2 \rangle)$
    if $||z||_\infty \geq \gamma_1 - \beta$ ou $||r_0||_\infty \geq \gamma_2 - \beta$ **then**
        $(z, h) \leftarrow \perp$ ▷ Verificações de validade
    else
        $\langle ct_0 \rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{t}_0)$
        $h \leftarrow \text{MakeHint}(-\langle ct_0 \rangle, w - \langle cs_2 \rangle + \langle ct_0 \rangle)$ ▷ Dica do assinante
        if $||\langle ct_0 \rangle||_\infty \geq \gamma_2$ ou o número de 1s em h é maior que ω **then**
            $(z, h) \leftarrow \perp$
        end if
    end if
    $\kappa \leftarrow \kappa + \ell$ ▷ Incrementa o contador
end while
$\sigma \leftarrow \text{sigEncode}(c, \tilde{z} \mod \pm q, h)$
return σ

false, indicando uma assinatura inválida. Em seguida, a matriz pública A é gerada na representação NTT como $\hat{A}$ usando ρ, permitindo cálculos eficientes.

Na sequência, o algoritmo calcula o representante da mensagem μ utilizando um hash da concatenação do hash da chave pública tr e a mensagem formatada M'. A função `SampleInBall` é empregada para transformar o hash de compromisso $\tilde{c}$ em um polinômio c em R_q, com coeficientes limitados a $\{-1, 0, 1\}$ e peso de Hamming $\tau \leq 64$, garantindo um desafio imprevisível e seguro. O valor intermediário w' é então calculado de forma eficiente, combinando a resposta assinada z, o desafio c, e o compromisso público t_1 para formar um novo compromisso.

Finalmente, a verificação envolve a função `UseHint` para reconstruir o compromisso do assinante w'_1 usando a dica h e o valor w'. Um novo hash de compromisso $\tilde{c}'$ é calculado e deve coincidir com o hash de compromisso original $\tilde{c}$. Para que a assinatura seja considerada válida, duas condições devem ser atendidas: a norma infinita $||z||_\infty$ deve ser menor que $\gamma_1 - \beta$, e $\tilde{c}'$ deve ser igual a $\tilde{c}$. Se ambas as condições forem satisfeitas, o algoritmo retorna **true**, indicando uma assinatura válida; caso contrário, retorna **false**. Este processo garante que a assinatura digital mantém tanto a autenticidade do assinante quanto a integridade da mensagem.

Algoritmo 21 ML-DSA.Verify_internal(pk, M', σ)

Entrada: Chave pública $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$, mensagem formatada $M' \in \{0,1\}^*$, assinatura $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$.
Saída: Booleano.

1: $(\rho, t_1) \leftarrow \text{pkDecode}(pk)$
2: $(\tilde{c}, z, h) \leftarrow \text{sigDecode}(\sigma)$ ▷ Decodifica o hash de compromisso $\tilde{c}$, a resposta z, e a dica h
3: **if** $h = \perp$ **then**
4: **return false** ▷ Retorna falso se a dica não foi codificada corretamente
5: **end if**
6: $\hat{A} \leftarrow \text{ExpandA}(\rho)$ ▷ A é gerada e armazenada na representação NTT como $\hat{A}$
7: $tr \leftarrow H(pk, 64)$
8: $\mu \leftarrow H(\text{BytesToBits}(tr) \parallel M', 64)$ ▷ Representante da mensagem, opcionalmente calculada em outro módulo criptográfico
9: $c \in R_q \leftarrow \text{SampleInBall}(\tilde{c})$ ▷ Calcula o desafio do verificador a partir de $\tilde{c}$
10: $w' \leftarrow \text{NTT}^{-1}(\hat{A} \circ \text{NTT}(z) - \text{NTT}(c) \circ \text{NTT}(t_1 \cdot 2^d))$ ▷ $w' = Az - ct_1 \cdot 2^d$
11: $w'_1 \leftarrow \text{UseHint}(h, w')$ ▷ Reconstrói o compromisso do assinante
12: $\tilde{c}' \leftarrow H(\mu \parallel \text{w1Encode}(w'_1), \lambda/4)$ ▷ Gera um novo hash de compromisso que deve coincidir com $\tilde{c}$
13: **return** $[[||z||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$ ▷ Retorna verdadeiro se ambos os critérios de verificação forem satisfeitos

4.5 Funções Auxiliares

Dado o elevado número de algoritmos que especificam os procedimentos auxiliares para o esquema ML-DSA, optou-se por resumir nesta seção a finalidade de cada um deles,

agrupando-os por categorias funcionais. A descrição completa de cada algoritmo pode ser consultada no Apêndice B.

Geração de Polinômios

- `ExpandA`: A função `ExpandA` utiliza o SHAKE-128 para gerar os polinômios da matriz pública $A \in R_q^{k \times \ell}$. Isso é feito expandindo a semente $\rho \in \{0,1\}^{256}$ junto com valores de nonce de 16 bits. A amostragem por rejeição é apoiada pela função `RejNTTPoly`.

- `ExpandS`: Utiliza SHAKE-256 para gerar os vetores de polinômios secretos s_1 e $s_2 \in S^\ell \times S^k$. Para cada polinômio, a semente ρ e um nonce de 16 bits são passados para o SHAKE-256 e submetidos a uma amostragem por rejeição no intervalo $\{-\eta, \eta\}$ com apoio da função `RejBoundedPoly`.

- `ExpandMask`: A função `ExpandMask()` é usada para gerar um vetor de polinômios com coeficientes no intervalo $[-\gamma_1 + 1, \gamma_1]$.

- `SampleInBall`: Durante a geração e verificação de assinaturas, esta função é usada para gerar um polinômio com exatamente τ coeficientes definidos como $+1$ ou -1, enquanto os coeficientes restantes são zero.

Aritmética de Polinômios

- As multiplicações de polinômios são realizadas usando a `NTT` e sua `inversa`. As operações no domínio T_q são realizadas pelos algoritmos `AddNTT` para adição, `MultiplyNTT` para multiplicação e `AddVectorNTT` para soma de dois vetores. O produto de um escalar por um vetor é obtido por meio de `ScalarVectorNTT`. Por fim, uma vez que as multiplicações modulares da forma $a \cdot b \mod q$ são computacionalmente intensivas, a técnica de Montgomery é usada com auxílio da função `Montgomery_Reduce`.

Funções de Resumo Criptográfico

- **G() e H()**: SHAKE-128 é usado na função G e SHAKE-256 na função H.

Power2Round

- A função $\texttt{Power2Round}_q$ recebe um elemento $r = r_1 \cdot 2^d + r_0$ e retorna r_0 e r_1, onde $r_0 = r \mod \pm 2^d$ e $r_1 = \frac{r - r_0}{2^d}$. Esta função é usada para reduzir a precisão de um valor de forma controlada, preservando componentes de alta e baixa precisão para futuras operações.

Decomposição e Funções Relacionadas

- Seja α um divisor de $q - 1$. A função `Decompose` é definida da mesma maneira que `Power2Round`, com α substituindo 2^d. As funções `HighBits` e `LowBits` retornam r_1 e r_0, respectivamente, a partir da saída de `Decompose`. A função `MakeHint` utiliza

`HighBits` para produzir uma dica h. A função `UseHint` usa a dica h produzida por `MakeHint` para recuperar os bits mais significativos.

Conversão de Tipos de Dados

- A função `IntegerToBits` converte um número inteiro não negativo em uma string de bits de comprimento α, permitindo a representação eficiente de números em formato binário, enquanto a função `BitsToInteger` realiza a conversão inversa. Para manipulação de dados em interfaces de baixo nível, as funções `BitsToBytes` e `BytesToBits` são utilizadas; a primeira converte uma string de bits em uma string de bytes, agrupando cada conjunto de 8 bits, e a segunda realiza o processo inverso. Além disso, para a geração de coeficientes de polinômios dentro de limites específicos, a função `CoefFromThreeBytes` é utilizada para derivar um coeficiente a partir de três bytes, verificando se o valor está no intervalo módulo q. Complementarmente, a função `CoefFromHalfByte` gera um coeficiente de polinômio a partir de meio byte, com base no parâmetro η.

Codificação e Decodificação

- As funções de codificação e decodificação tem por objetivo a serialização e recuperação eficientes de dados, assegurando que as informações possam ser transmitidas e armazenadas de forma compacta e precisa. As funções `SimpleBitPack` e `BitPack` são utilizadas para codificar vetores de polinômios em strings de bytes; a primeira realiza a codificação de maneira direta, enquanto a segunda considera os limites superiores e inferiores dos coeficientes. Para a decodificação, as funções `SimpleBitUnpack` e `BitUnpack` convertem strings de bytes de volta para polinômios, sendo que a última assegura que os dados recuperados respeitem os limites de coeficientes especificados. Adicionalmente, as funções `HintBitPack` e `HintBitUnpack` lidam com a codificação e decodificação de vetores de dicas em strings de bytes, registrando e restaurando as posições de coeficientes não-zero, o que evita a transmissão de dados redundantes e auxilia na correção de erros.

  A serialização das chaves ocorre por meio dos processos especificados nos algoritmos `pkEncode` e `skEncode`. Já os algoritmos `pkDecode` e `skEncode` são empregados para desserialização.

  A serialização das assinaturas é realizada pela função `sigEncode` e a operação inversa pela função `sigDecode`. Durante o processo de verificação, a função `sigDecode` pode lidar com entradas provenientes de fontes não confiáveis, sendo necessário usar a função `BitUnpack` para garantir que os valores retornados estejam sempre no intervalo correto. Por fim, `w1Encode` codifica um vetor de polinômios $\mathbf{w}_1 \in R^k$ em uma string de bytes.

Capítulo 5

Implementação e otimizações

A arquitetura de computadores forma a base tecnológica que define a estrutura funcional e a organização de um sistema computacional, influenciando diretamente o desempenho, a eficiência energética e a capacidade de execução de instruções. No cenário atual, duas abordagens se destacam: a Intel x86, amplamente reconhecida por seu uso em computadores pessoais, e a arquitetura ARM (*Advanced RISC Machine*), que ganhou notoriedade em dispositivos móveis devido à sua eficiência energética. A arquitetura x86 segue o paradigma CISC (*Complex Instruction Set Computing*) para aumentar a eficiência no nível do compilador com instruções complexas e de múltiplos ciclos. Em contraste, a ARM adota o conceito RISC (*Reduced Instruction Set Computing*), que simplifica as operações, permitindo que cada instrução seja executada em um único ciclo de relógio, o que é particularmente vantajoso para aplicações que requerem baixo consumo de energia e alto desempenho.

Nos últimos anos, a arquitetura ARM tem experimentado um crescimento significativo na indústria, indo além dos dispositivos móveis e sendo adotada de forma crescente em supercomputadores e centros de dados. Essa ampla adoção torna a arquitetura uma escolha estratégica por propiciar o alcance de diversos segmentos tecnológicos.

5.1 Arquitetura Advanced RISC Machine (ARMv8)

Embora a versão mais recente da arquitetura seja a ARMv9, ARMv8 continua sendo a mais difundida, devido à sua ampla adoção e maturidade tecnológica. Além disso, ARMv8 oferece um conjunto versátil de capacidades de computação, a maioria das quais é compatível com a nova versão, o que implica que otimizações desenvolvidas para ARMv8 também beneficiam dispositivos baseados na ARMv9.

A ARMv8 é dividida em três perfis específicos: *Application profile* (A), voltado para a maioria das necessidades de computação; *Microcontroller profile* (M), direcionado para sistemas embarcados com restrições de energia; e *Real-time profile* (R), focado em ambientes críticos de segurança e tempo real. Neste trabalho, foi adotado o perfil A (ARMv8-A) devido à sua aplicabilidade diversificada em diferentes tipos de dispositivos, incluindo computadores pessoais de alto desempenho, servidores, dispositivos móveis e sistemas embarcados avançados.

No perfil ARMv8-A, a arquitetura suporta dois modos de execução distintos: AArch64, que utiliza registradores de 64 bits e o conjunto de instruções A64, introduzido com o ARMv8; e AArch32, um modo de registrador de 32 bits que usa o conjunto de instruções A32, proporcionando compatibilidade com ARMv7 e sistemas anteriores. A introdução da capacidade de 64 bits com ARMv8 representou um avanço significativo em relação às arquiteturas anteriores, permitindo um maior espaço de endereçamento e um conjunto de instruções mais eficiente. Nesta dissertação, apenas o modo de execução AArch64 será utilizado devido à sua importância para implementações de alto desempenho em dispositivos modernos.

Uma característica singular da arquitetura ARMv8-A é sua flexibilidade em termos de paradigmas de execução. Ela não especifica um modelo fixo, permitindo implementações tanto de execução em ordem[1] quanto fora de ordem[2]. Processadores como o Cortex-A53 e Cortex-A35, por exemplo, utilizam a execução em ordem, adequada para aplicações de baixo consumo energético e dispositivos móveis. Por outro lado, processadores mais avançados, como os da série ARM Cortex-A7x e aqueles da linha Apple M (M1, M2 e M3), implementam a execução fora de ordem, uma abordagem que favorece o alto desempenho e a eficiência em cargas de trabalho mais complexas, como processamento intensivo de dados e aprendizado de máquina.

Adicionalmente, a arquitetura ARMv8-A inclui o suporte ao SIMD (*Single Instruction, Multiple Data*) avançado, conhecido como NEON, que permite o processamento simultâneo de múltiplos dados com uma única instrução. Isso é particularmente vantajoso em implementações criptográficas, onde operações matemáticas intensivas podem ser executadas em paralelo. Além disso, a presença de instruções criptográficas dedicadas, como AES (*Advanced Encryption Standard*) e SHA (*Secure Hash Algorithm*), também pode melhorar a eficiência, permitindo que operações críticas sejam executadas diretamente no hardware, minimizando a carga da CPU e aumentando a segurança ao limitar a exposição de dados sensíveis.

Na próxima seção, serão detalhadas as particularidades e instruções específicas da arquitetura ARMv8-A úteis para a otimização das implementações dos esquemas ML-KEM e ML-DSA.

Características da ARMv8

O modo AArch64 utiliza o conjunto de instruções A64, que é baseado em uma arquitetura *load-store*, onde operações de leitura e escrita na memória são separadas das operações de processamento. Esta abordagem permite um controle mais preciso sobre o fluxo de dados, aumentando a eficiência computacional. O banco de registradores AArch64 inclui 31 registradores de propósito geral, cada um com 64 bits de largura, além de três registradores especiais: o registrador zero, o ponteiro de pilha e o contador de programa. Este conjunto de registradores é complementado pelo banco de registradores NEON, utilizado

[1]Execução em ordem (*in-order execution*) refere-se à execução de instruções na ordem em que aparecem no programa, tipicamente beneficiando processadores que priorizam simplicidade e eficiência energética.

[2]Execução fora de ordem (*out-of-order execution*) permite que o processador execute instruções baseando-se na disponibilidade de recursos, não necessariamente seguindo a ordem do código, o que pode aumentar o desempenho ao utilizar melhor as unidades de execução disponíveis.

pela unidade ALU SIMD Avançada, que executa operações vetorizadas e de ponto flutuante. O banco NEON possui 32 registradores de 128 bits, que podem ser interpretados de diferentes formas, como bytes (8 bits), *words* (16 bits), *doublewords* (32 bits) e *quadwords* (64 bits), proporcionando uma flexibilidade que viabiliza uma significativa eficiência em operações vetoriais. Embora haja uma latência associada ao carregamento de valores nos registradores NEON, devido à necessidade de recarregamento da memória ou migração dos registradores escalares, o ganho de velocidade no processamento de dados vetorizados frequentemente supera este custo.

5.1.1 Instruções de Interesse para Otimizações

Diversas instruções específicas são particularmente úteis para otimizações no contexto deste trabalho:

- **Carregamento/armazenamento de múltiplas estruturas (LD1, LD2, LD3, LD4, ST1, ST2, ST3, ST4)**: Essas instruções permitem o carregamento eficiente de dados de blocos contíguos de memória diretamente para um a quatro registradores NEON, seguindo um padrão entrelaçado. Isso é particularmente útil para operações que envolvem vetores ou matrizes, onde a organização intercalada dos dados é benéfica para o paralelismo e para a reutilização de dados.

  Por exemplo, suponha que os dados na memória sejam inteiros de 32 bits, numerados de 1 a 16. Utilizando a instrução `LD4` com quatro registradores NEON (`r1`, `r2`, `r3` e `r4`), os valores seriam carregados da seguinte forma: `r1` = (1, 5, 9, 13), `r2` = (2, 6, 10, 14), `r3` = (3, 7, 11, 15) e `r4` = (4, 8, 12, 16).

  Esse padrão entrelaçado permite que os dados sejam organizados diretamente em registradores de maneira alinhada às operações subsequentes, reduzindo a necessidade de reorganizações explícitas e economizando ciclos de clock.

  De forma análoga, as instruções de armazenamento `ST1`, `ST2`, `ST3` e `ST4` realizam o armazenamento eficiente de dados de um a quatro registradores NEON em um bloco contíguo de memória, preservando o padrão entrelaçado.

- **Deslocamento de bits (VSHRQ)**: A instrução VSHRQ (*vector shift right*) realiza um deslocamento lógico à direita em múltiplos elementos de vetores inteiros, com largura de 8, 16, 32 ou 64 bits, de forma paralela. Os bits deslocados para a direita são descartados, e os bits superiores (à esquerda) são preenchidos com zeros.

  Essa operação é utilizada para divisões rápidas por potências de dois, normalização de dados e manipulação de bits em algoritmos criptográficos e de processamento de sinais por exemplo.

  A sintaxe geral da instrução é: `VSHRQ{type} {qd}, {qn}, #{imm}`

  - `type`: Define a largura dos elementos (`8`, `16`, `32` ou `64` bits).
  - `qd`: Registrador de destino (*vector register*).
  - `qn`: Registrador fonte contendo os dados de entrada.

  - `#imm`: Valor imediato que especifica o número de bits a ser deslocado.

  Por exemplo, o comando `VSHRQ.U32 q0, q1, #2` desloca logicamente os elementos de 32 bits contidos no registrador `q1` para a direita por 2 posições e armazena o resultado no registrador `q0`.

- **Instruções Aritméticas (VMULL, VMUL, VMLA, VADD, VSUB)**: Essas instruções realizam operações vetoriais de multiplicação, soma e subtração em blocos de dados. São particularmente importantes para a multiplicação de polinômios e para os cálculos da NTT. A instrução VMULL (*vector multiply long*) é usada para multiplicações de 16 bits com extensão do resultado para 32 bits ($(\mathbb{Z}_{2^{16}})^2 \longrightarrow \mathbb{Z}_{2^{32}}$) ou para multiplicações de 32 bits com extensão para 64 bits ($(\mathbb{Z}_{2^{32}})^2 \longrightarrow \mathbb{Z}_{2^{64}}$), garantindo a precisão necessária em operações como a redução de Montgomery. VMUL (*vector multiply*) é utilizada para multiplicações diretas, como as realizadas na redução de Barrett, onde se requer uma multiplicação vetorial eficiente e rápida sem expansão de bits. VMLA (*vector multiply-accumulate*) realiza a multiplicação de vetores e acumula o resultado em um vetor de destino ($d = (ab) + c$), sendo utilizada em operações que exigem a combinação de multiplicações e somas, tais como as operações realizadas na redução de Barrett. VADD e VSUB realizam somas e subtrações vetoriais, sendo importantes para os cálculos da NTT e em etapas de multiplicação de polinômios, onde os coeficientes precisam ser combinados e ajustados de maneira vetorizada.

- **Replicação de dados (VDUP)**: A instrução `VDUP` (*Vector Duplicate*) replica um valor escalar em todos os elementos de um vetor, preenchendo o vetor com cópias idênticas desse valor. Essa operação é fundamental para preparar vetores para operações paralelas, como multiplicações ou somas que envolvem um valor constante.

5.2 Aspectos Críticos de Implementação

Implementar uma versão otimizada de algoritmos criptográficos para uma plataforma específica gera desafios substanciais. Converter as descrições teóricas em código eficiente requer uma compreensão das características da arquitetura de hardware e das especificidades do compilador, além de habilidades para explorar instruções avançadas e paralelismo adequados a cada situação. Neste contexto, alguns aspectos podem ser destacados:

Multiplicações Polinomiais com Transformadas NTT

A implementação eficiente da NTT e das multiplicações polinomiais é um aspecto fundamental, pois essas operações custam juntas o equivalente a cerca de 43% do processo de assinatura e cerca de 29% do processo de encapsulamento. Para implementar as operações NTT e INTT de forma otimizada, foram utilizadas as instruções aritméticas citadas na seção 5.1.1, que permitem realizar multiplicações, adições e subtrações vetorizadas e que foram projetadas para serem executadas em tempo constante. Essas instruções ajudam a processar várias operações de multiplicação de coeficientes de uma só vez, reduzindo

de forma significativa a latência. A implementação foi estruturada para usufruir do potencial ofertado pelos registradores NEON, que possuem 128 bits de largura, permitindo operações simultâneas em oito inteiros de 16 bits ou quatro de 32 bits para o ML-KEM e ML-DSA respectivamente.

Alinhamento de Memória e Uso de Cache

Outro aspecto técnico envolve o alinhamento de memória e o gerenciamento do cache L1 no Apple M1 e no Apple M2. As rotinas de ambos os esquemas manipulam grandes vetores e matrizes, exigindo um acesso eficiente à memória para minimizar atrasos causados por falhas de cache. O alinhamento dos dados em memória e o uso eficiente do cache são particularmente importantes para as operações que realizam múltiplos acessos a posições de memória adjacentes tais como os demandados pela NTT. A implementação foi ajustada para que os dados estivessem alinhados de maneira a aproveitar ao máximo a largura de banda do cache L1, reduzindo os custos de acesso e garantindo uma execução mais rápida e uniforme. Para isso, foi empregada a diretiva `__attribute__((aligned()))`, que faz parte das extensões suportadas pelos compiladores GCC e Clang em C/C++, assegurando que cada instância da estrutura que representa um polinômio estivesse alinhada de maneira otimizada para operações vetoriais. Além disso, a instrução `VLD1` é utilizada para carregar os dados de forma contígua nos registradores NEON, aproveitando ao máximo a largura de banda do cache e garantindo um acesso eficiente.

Otimização na Geração da Matriz A

A geração da matriz A representa um dos componentes mais custosos, sendo necessária tanto na geração de chaves quanto em operações subsequentes dos esquemas. Este processo envolve chamadas extensivas às funções `SHAKE` para derivar os coeficientes pseudoaleatórios que compõem a matriz, tornando-se um gargalo significativo em termos de tempo de execução. Portanto, a otimização desta etapa é essencial para melhorar a performance geral dos algoritmos.

5.3 Ataques de Canal Lateral

Ataques de canal lateral representam uma ameaça crítica para a implementação de algoritmos criptográficos. Esses ataques exploram vazamentos de informações, como tempo de execução, consumo de energia ou emissões eletromagnéticas, para recuperar chaves secretas ou outros dados confidenciais.

Uma implementação robusta precisa não apenas garantir a correção e a eficiência dos cálculos, mas também se proteger contra esse tipo de ameaça. No contexto dos algoritmos foco deste trabalho, foram identificadas operações críticas que manipulam segredos e implementadas contramedidas para assegurar a segurança dos dados sensíveis ao longo das operações dos esquemas.

Para o algoritmo ML-KEM, as operações aritméticas que envolvem os coeficientes dos polinômios secretos s e e durante o encapsulamento e desencapsulamento são particularmente suscetíveis a ataques de canal lateral. Essa operações podem revelar informações

sobre os valores secretos através de vazamentos de temporização ou consumo de energia. Para mitigar esse risco, foram utilizadas instruções projetadas para gerar execução em tempo constante para essas operações, evitando variações de desempenho que possam expor padrões que sejam úteis ao atacante. Tal medida também foi adotada para os cálculos da NTT e sua inversa a fim de reduzir as chances de vazamentos associados a diferenciais de execução.

No caso do ML-DSA, a operação $t = As_1 + s_2$, por exemplo, é particularmente vulnerável a ataques de canal lateral, dado que os vetores secretos s_1 e s_2 podem influenciar o tempo de execução e o consumo energético de maneira detectável. Essa multiplicação polinomial precisa ser cuidadosamente implementada para mitigar os riscos de vazamentos associados. Para alcançar tal objetivo, foi adotada uma abordagem que utiliza instruções projetadas para execução em tempo constante, mantendo a uniformidade independente dos valores em s_1 e s_2, aumentando a robustez da implementação.

Ainda nesse contexto, recentemente, duas publicações expuseram vulnerabilidades presentes nos esquemas alvo deste trabalho que afetavam muitas bibliotecas disponíveis na *web*, a saber o trabalho de Bernstein et al.[8] e o trabalho de Chen et al. [43].

Berstein investigou como operações de divisão que dependem de dados secretos podem introduzir variações de temporização, expondo informações sensíveis sobre as chaves privadas. O estudo demonstrou que a divisão de coeficientes por constantes pode resultar em tempos de execução que oscilam conforme os valores de entrada, representando um risco de vazamento de informações. Além disso, o trabalho apontou a presença de operações de divisão direta, utilizando o operador (`/`), tanto na implementação de referência quanto em bibliotecas amplamente acessíveis na *web*.

Um dos exemplos mencionados no artigo é a compressão de polinômios que usa divisões como a seguinte:

```
t = (((t << 1) + KYBER_Q/2) / KYBER_Q) & 1;
```

esse trecho é usado na compressão de coeficientes e pode introduzir variações no tempo de execução que revelem detalhes sobre o texto cifrado recomposto. Se um atacante conseguir medir essas variações, ele pode inferir informações críticas sobre a chave secreta, tornando importante implementar essas operações de maneira constante.

Para mitigar esse tipo de vulnerabilidade, as divisões dependentes de dados secretos podem ser substituídas por operações alternativas, como multiplicações e shifts que sejam executados de forma uniforme e em tempo constante. Dessa forma, o código se torna mais resistente a ataques de temporização.

Por outro lado, Chen introduz uma técnica inovadora para realizar ataques de canal lateral em algoritmos pós-quânticos, incluindo o Kyber, analisando sinais temporais no domínio da frequência em diferentes estágios da computação.

A técnica explora como pequenas variações no tempo de execução de instruções criptográficas podem ser amplificadas e analisadas no domínio da frequência, revelando padrões que estão correlacionados com a chave secreta. Conforme afirmam os autores, implementações criptográficas para a arquitetura ARMv8, como o Apple M1, são vulneráveis a esse tipo de ataque devido às variações nas operações aritméticas complexas, como multiplicações de polinômios. O código a seguir é vulnerável:

```
void poly_multiply(poly *r, const poly *a, const poly *b) {
    for (int i = 0; i < N; i++) {
        r->coeffs[i] = a->coeffs[i] * b->coeffs[i];
    }
}
```

Listagem 5.1: Trecho de código vulnerável à ataques de tempo.

Para mitigar esse tipo de ataque, devem ser implementadas técnicas de mascaramento e operações de tempo constante. O mascaramento é uma técnica de proteção criptográfica que utiliza valores aleatórios, chamados máscaras, para ocultar os dados sensíveis durante os cálculos. Essas máscaras são geradas dinamicamente e aplicadas de forma independente a cada variável sensível, garantindo que os dados reais não sejam manipulados diretamente durante as operações. Como resultado, mesmo que o atacante tenha acesso a vazamentos de dados intermediários, essas informações não revelarão os dados sensíveis originais.

Além disso, as operações que envolvem dados confidenciais devem ser implementadas de maneira que o tempo de execução seja independente dos valores das entradas. Dessa forma, além de adotar estratégias específicas para evitar padrões detectáveis em acessos à memória e operações condicionais, faz-se necessário selecionar cuidadosamente as instruções a serem empregadas, buscando aquelas projetadas para execução em tempo constante. Tal medida, reduz a possibilidade de variações temporais que possam ser capturadas e analisadas por um atacante.

Uma abordagem mais global para mitigar ataques de canal lateral envolve o uso do bit *Data Independent Timing* (DIT) presente na arquitetura ARMv8, que pode ser ativado diretamente nos dispositivos usados neste trabalho. O bit DIT instrui o processador a manter o tempo de execução constante em operações sensíveis já projetadas para este comportamento, como comparações e transferências de dados. Isso assegura maior previsibilidade na execução dessas operações, ajudando a reduzir vazamentos temporais em rotinas críticas.

O código a seguir demonstra como ativar o bit DIT nos dispositivos da Apple usados neste trabalho, garantindo que as operações subsequentes sejam menos suscetíveis a variações de tempo:

```
void set_dit_bit() {
    uint64_t dit = 1 << 24;  // Ativa o bit DIT
    // Configura o registrador especial
    asm volatile("msr s3_3_c4_c2_5, %0" : : "r"(dit));

    dit = 0;

    // Verifica se o bit foi ativado
    asm volatile("mrs %0, s3_3_c4_c2_5" : "=r"(dit));

    if (dit != 1 << 24) {
        exit(1);  // Sai se a configuracao falhar
    }
}
```

Listagem 5.2: Trecho de código para ativação do bit DIT na plataforma ARMv8.

Este código usa instruções *inline* de assembly para configurar o bit DIT no registrador especial, garantindo que as instruções projetadas para serem executadas em tempo constante se comportem da maneira prevista. Abaixo, a forma como esse recurso foi usado:

```
#ifdef USE_FEAT_DIT
    set_dit_bit();
#endif
```

Ao usar a macro `USE_FEAT_DIT`, o bit DIT é ativado no início das operações críticas, assegurando que o tempo de execução seja constante, reduzindo o risco de ataques de canal lateral.

A utilização do bit DIT fornece uma mitigação mais global contra ataques de temporização, tornando-a uma técnica eficaz para as plataformas da família Apple Silicon, sem comprometer o desempenho geral do sistema.

5.4 Otimizações

Para orientar as escolhas sobre as rotinas que deveriam ser prioritárias durante o processo de otimização, foi realizado um perfilamento de nossa implementação inicial em C para identificar gargalos e rotinas custosas em termos de ciclos de *clock* para cada esquema. As Tabelas 5.1 a 5.6 apresentam os resultados desse perfilamento em termos percentuais. O enfoque dessas tabelas não é fornecer uma lista exaustiva de todas as funções, mas identificar as rotinas críticas com maior impacto no desempenho, que podem se beneficiar de otimizações específicas e, consequentemente, proporcionar aumento da eficiência global dos esquemas. Esses resultados foram obtidos utilizando a ferramenta Valgrind na versão

3.15.0 sobre a plataforma Apple M1, um *System on a Chip* (SoC) desenvolvido pela Apple, baseado na arquitetura ARMv8-A lançado em novembro de 2020. O M1 possui um *design* híbrido com 8 núcleos de CPU, divididos entre: 4 núcleos de alto desempenho (*Firestorm*), projetados para tarefas intensivas em processamento, com clock máximo de até 3,2 GHz; e 4 núcleos de alta eficiência (*Icestorm*), otimizados para reduzir o consumo de energia em tarefas menos exigentes, operando em frequências mais baixas. Cada núcleo de alto desempenho possui 192 KB de cache L1 de instrução e 128 KB de cache L1 de dados, enquanto os núcleos de alta eficiência têm 128 KB de cache L1 de instrução e 64 KB de cache L1 de dados. O processador também inclui um cache L2 compartilhado de 12 MB para os núcleos *Firestorm* e 4 MB para os núcleos *Icestorm*.

5.4.1 ML-KEM

Função	**KeyGen (%)**	**Encaps (%)**	**Decaps (%)**
KeccakF1600_StatePermute	35.3%	26.3%	19.9%
NTT	18.4%	7.4%	11.0%
BaseMul	26.4%	29.8%	31.0%
InvNTT	-	11.7%	11.6%
Gen_Matrix	4.7%	3.6%	2.8%
Poly_Reduce	7.6%	12.4%	13.1%
Total	96.7%	94.4%	95.0%

Tabela 5.1: Perfilamento das Funções no ML-KEM-512.

Função	**KeyGen (%)**	**Encaps (%)**	**Decaps (%)**
KeccakF1600_StatePermute	33.0%	27.6%	21.8%
NTT	16.4%	7.0%	10.9%
BaseMul	30.3%	32.0%	33.6%
InvNTT	-	9.9%	9.7%
Gen_Matrix	6.3%	5.1%	4.0%
Poly_Reduce	6.7%	10.3%	10.9%
Total	94.7%	94.2%	95.2%

Tabela 5.2: Perfilamento das Funções no ML-KEM-768.

Função	KeyGen (%)	Encaps (%)	Decaps (%)
KeccakF1600_StatePermute	34.0%	29.4%	24.0%
NTT	14.2%	6.3%	10.0%
BaseMul	32.1%	33.7%	35.0%
InvNTT	-	8.5%	8.2%
Gen_Matrix	7.8%	6.7%	5.4%
Poly_Reduce	5.7%	8.8%	9.2%
Total	94.9%	94.5%	95.6%

Tabela 5.3: Perfilamento das Funções no ML-KEM-1024.

5.4.2 ML-DSA

Função	KeyGen (%)	Sign (%)	Verify (%)
KeccakF1600_StatePermute	45.4%	15.9%	35.0%
InvNTT_Tomont	8.4%	25.1%	15.5%
NTT	7.2%	11.8%	10.5%
Poly_Pointwise_Montgomery	14.9%	29.4%	22.0%
Poly_Uniform	7.8%	1.1%	6.0%
Poly_Reduce	0.8%	1.8%	1.5%
Poly_Caddq	0.6%	0.6%	0.6%
Poly_Add	0.7%	1.0%	1.0%
Total	85.8%	86.7%	92.1%

Tabela 5.4: Perfilamento das funções no ML-DSA-44.

Função	KeyGen (%)	Sign (%)	Verify (%)
KeccakF1600_StatePermute	49.0%	15.4%	27.4%
InvNTT_Tomont	7.5%	25.4%	17.8%
NTT	5.5%	10.7%	10.9%
Poly_Pointwise_Montgomery	14.4%	31.5%	25.2%
Poly_Uniform	8.7%	1.9%	4.0%
Poly_Reduce	1.2%	1.8%	1.3%
Poly_Caddq	0.5%	0.7%	0.6%
Poly_Add	0.9%	1.2%	0.9%
Total	87.7%	88.6%	87.8%

Tabela 5.5: Perfilamento das funções no ML-DSA-65.

Função	KeyGen (%)	Sign (%)	Verify (%)
KeccakF1600_StatePermute	51.5%	15.8%	30.0%
InvNTT_Tomont	6.1%	24.3%	15.0%
NTT	4.7%	9.7%	10.5%
Poly_Pointwise_Montgomery	14.8%	33.3%	25.0%
Poly_Uniform	9.9%	1.2%	5.5%
Poly_Reduce	0.9%	1.8%	1.5%
Poly_Caddq	0.4%	0.6%	0.4%
Poly_Add	1.1%	1.6%	1.2%
Total	89.4%	88.3%	89.1%

Tabela 5.6: Perfilamento das funções no ML-DSA-87.

Para otimizar as rotinas críticas, a principal técnica utilizada foi a vetorização de laços. Por meio dela, foi possível aplicar as instruções SIMD para reduzir o número de iterações em rotinas importantes como a NTT e sua inversa, a multiplicação de polinômios, as reduções modulares e a geração da matriz A, que juntas custam até 68% no ML-KEM e 60% no ML-DSA. Além da vetorização de laços, outras técnicas foram aplicadas para otimizar o desempenho das operações criptográficas. O *loop unrolling* foi implementado em funções de amostragem, como `cbd2` e `cbd3`, para reduzir o overhead das operações de controle, permitindo um processamento mais eficiente dos coeficientes. Além disso, embora o código de referência também utilize tabelas pré-computadas para armazenar as raízes primitivas da unidade empregadas nos cálculos da NTT e INTT, este trabalho propõe uma implementação aprimorada do uso dessas tabelas ao combinar técnicas de alinhamento de memória e vetorização com instruções NEON específicas para a arquitetura ARMv8. Essas estratégias combinadas ajudaram a alcançar um desempenho otimizado, garantindo que as operações críticas de cada esquema fossem executadas de forma eficiente conforme detalhado a seguir.

Transformadas NTT e INTT

Utilizar NTT por si só já traz benefícios consideráveis ao esquema. Podemos constatar tal fato por meio da análise da operação mais intensa no processo de geração das chaves (`Algoritmo 19`), que é a multiplicação $w = As_1$ onde $A \in R_q^{kx\ell}$ e $s_1 \in \tilde{S}_{\gamma_1}^{\ell}$. Se o processo de multiplicação tradicional fosse empregado, seria necessário realizar kxℓ multiplicações de polinômios cada uma delas requerendo n^2 multiplicações em $\mathbb{Z}_q$ (onde $n = 256$). Isso seria o equivalente a $\approx 2^{22}$ multiplicações em $\mathbb{Z}_q$ considerando o ML-DSA(8,7). Por outro lado, com o emprego da NTT é possível reduzir as multiplicações para $\approx 2^{15}$, tendo em vista que seria necessário converter s_1 (`NTT(`s_1`)`), calcular w e, finalmente, realizar `INTT(`w`)`. As operações de transformação de domínio são realizadas em ℓx$nlogn$ e o cálculo de w em $k\ell n$ multiplicações em $\mathbb{Z}_q$. Uma aceleração de fator 128.

Todavia, faz-se necessário implementar de forma eficiente para que o potencial ganho teórico realmente se traduza em um processo de menor custo computacional. A estratégia adotada teve como pilares os cálculos *in-place*, o uso de tabela de consulta para as raízes

primitivas da unidade e o emprego de reduções otimizadas, culminando em significativa economia de ciclos.

Em relação às reduções, um dos trechos de impacto relevante é a multiplicação que envolve as raízes primitivas da unidade (linha 8 do `Algoritmo 3`). Essa operação é crítica nos cálculos da NTT, pois exige sucessivas multiplicações modulares entre os coeficientes do polinômio e as raízes primitivas pré-computadas. Para otimizar essa operação, foi implementada a redução de Montgomery vetorizada de maneira a processar 8 coeficientes paralelamente conforme código apresentado na Listagem 5.3.

```
int16x8_t montgomery_reduce_neon_8(int32x4x2_t prod) {
    int32x4_t low = prod.val[0];
    int32x4_t high = prod.val[1];

    // Passo 1: m = (int16_t)a
    int16x4_t m_low = vmovn_s32(low);
    int16x4_t m_high = vmovn_s32(high);

    // Passo 2: t = m * QINV   (QINV = q^-1 mod 2^16)
    int32x4_t t_low = vmull_s16(m_low, vdup_n_s16(QINV));
    int32x4_t t_high= vmull_s16(m_high, vdup_n_s16(QINV));

    // Passo 3: m = (int16_t)t
    m_low = vmovn_s32(t_low);
    m_high = vmovn_s32(t_high);

    // Passo 4: u = m * KYBER_Q
    int32x4_t u_low = vmull_s16(m_low, vdup_n_s16(KYBER_Q));
    int32x4_t u_high= vmull_s16(m_high, vdup_n_s16(KYBER_Q));

    // Passo 5: t = a - u
    t_low = vsubq_s32(low, u_low);
    t_high = vsubq_s32(high, u_high);

    // Passo 6: t >>= 16
    t_low = vshrq_n_s32(t_low, 16);
    t_high = vshrq_n_s32(t_high, 16);

    // Passo 7: Retornar (int16_t)t
    int16x4_t result_low = vmovn_s32(t_low);
    int16x4_t result_high = vmovn_s32(t_high);

    return vcombine_s16(result_low, result_high);
}
```

Listagem 5.3: Código para Redução de Montgomery.

Essa redução vetorizada viabilizou maior eficiência ao processo da NTT. Durante as iterações com $len \geq 8$, os coeficientes foram carregados em blocos de 8 inteiros de 16 bits no registrador NEON. Para multiplicar esses coeficientes pelas raízes primitivas da unidade, os valores de zeta foram replicados com a instrução `VDUP`, possibilitando operações paralelas sobre múltiplos coeficientes.

Em seguida, as operações de soma e subtração também são processadas em paralelo

sobre esses blocos vetorizados, gerando ainda mais eficiência ao processo.

Essa integração estratégica das otimizações reduziu o número de iterações necessárias para processar cada polinômio de 256 (processamento escalar) para 32 (processamento em blocos de 8), resultando em uma diminuição significativa no número de ciclos de clock. Como consequência, foi possível alcançar uma aceleração de até 4× na execução da NTT no ML-KEM, quando comparada à implementação de referência.

Multiplicação de Polinômios

A multiplicação de polinômios no contexto dos esquemas tratados neste trabalho ocorre no anel $\mathbb{Z}_q[X]/(X^n + 1)$, onde $n = 256$ e q é um primo específico. Para acelerar essa operação, é utilizada a NTT, que permite a multiplicação de polinômios ponto a ponto no domínio da frequência. No caso do ML-DSA, o valor de q viabiliza a existência de uma raiz de ordem 512, permitindo a multiplicação ponto a ponto direta dos coeficientes correspondentes. No entanto, no caso do ML-KEM, o valor de q não permite a existência de uma raiz 512-ésima, e os cálculos são feitos levando em conta a relação $X^2 = \zeta$ como visto na Seção 3.1.4.

Essa particularidade exige que a multiplicação de polinômios seja realizada de forma estruturada, como mostrado no `Algoritmo 6`, que descreve a multiplicação de dois polinômios de grau 1 no anel $\mathbb{Z}_q[X]/(X^2 - \zeta)$.

Na multiplicação de dois polinômios $a(X) = a_0 + a_1X$ e $b(X) = b_0 + b_1X$, temos:

$$a(X) \cdot b(X) = a_0b_0 + (a_0b_1 + a_1b_0)X + a_1b_1X^2 \tag{5.1}$$

Levando em conta o anel $\mathbb{Z}_q[X]/(X^2 - \zeta)$, o termo X^2 é substituído por ζ, resultando em:

$$r(X) = (a_0b_0 + a_1b_1\zeta) + (a_0b_1 + a_1b_0)X \tag{5.2}$$

Para aproveitar a capacidade de paralelismo dos registradores NEON, a multiplicação de polinômios foi reorganizada para processar blocos de 8 coeficientes. Esse processamento vetorizado permite a realização de múltiplas operações simultâneas, reduzindo drasticamente o número de ciclos de clock em relação ao código de referência. A implementação foi estruturada da seguinte forma:

- **Multiplicações Diretas:** Os coeficientes ímpares e pares dos polinômios de entrada são carregados em vetores de 16 bits e multiplicados em paralelo usando a instrução `VMULL`. Além disso, a multiplicação do termo a_1b_1 por ζ ($a_1b_1\zeta$) também é realizada nessa etapa para garantir a correta redução do termo quadrático. O valor de ζ é carregado em vetores NEON por meio da instrução `VDUP`, que replica o valor de ζ para todos os elementos do vetor, garantindo a continuidade do processamento paralelo. Os produtos resultantes são reduzidos modularmente pela redução de Montgomery vetorizada.

- **Multiplicações Cruzadas:** Para garantir a correta combinação dos coeficientes lineares, são realizadas multiplicações cruzadas entre os coeficientes pares de um polinômio e os ímpares do outro, resultando nos termos a_0b_1 e a_1b_0.

- **Redução de Montgomery Vetorizada:** Todos os resultados intermediários, tanto das multiplicações diretas quanto das cruzadas, são reduzidos por meio da redução de Montgomery vetorizada, otimizando o processamento modular.

- **Combinação dos Resultados:** Os resultados das multiplicações diretas (incluindo o termo $a_1 b_1 \zeta$) e cruzadas são somados usando a instrução `VADD`, consolidando o resultado final do polinômio. O vetor resultante de 8 elementos é então armazenado de volta com `VST1`, garantindo acesso eficiente à memória.

Essa abordagem, viabiliza o carregamento e armazenamento contíguo de dados, o que maximiza o aproveitamento da hierarquia de memória. Tal estratégia resultou em uma otimização significativa da multiplicação de polinômios com aceleração de até 4.80x para o ML-KEM.

No caso do ML-DSA, a combinação de um módulo primo maior e o suporte para raízes primitivas de ordem adequada permite que cada coeficiente dos polinômios transformados seja multiplicado diretamente pelo seu correspondente, sem a necessidade de operações adicionais para combinar termos. A abordagem consiste nos seguintes aspectos:

- **Processamento Vetorizado em Blocos de 8 Coeficientes:** Os polinômios são processados em blocos de 8 coeficientes, utilizando a instrução `VLD1Q_S32_x2` para carregar de forma eficiente dois vetores de 4 elementos de 32 bits cada. Esse carregamento vetorizado reduz a quantidade de acessos à memória, aumentando a taxa de transferência de dados e garantindo o alinhamento adequado.

- **Multiplicação Ponto a Ponto com Paralelismo de Dados:** As multiplicações são realizadas com as instruções `VMULL`, que permitem multiplicar pares de coeficientes de 32 bits e produzir resultados de 64 bits sem perda de precisão. Assim, múltiplos produtos são processados simultaneamente, maximizando o uso dos registradores NEON e acelerando a os cálculos.

- **Redução de Montgomery Vetorizada:** Para garantir que os resultados das multiplicações permaneçam dentro do domínio $\mathbb{Z}_q$, os produtos de 64 bits são reduzidos com auxílio da redução de Montgomery vetorizada. Esse método é altamente eficiente por aproveitar a execução paralela, aplicando a redução modular de forma simultânea a vários coeficientes. Isso assegura que todos os valores resultantes estejam corretamente reduzidos módulo q, preservando as propriedades algébricas de $\mathbb{Z}_q[X]/(X^n + 1)$ e evitando sobrecargas computacionais.

- **Armazenamento Eficiente em Memória:** Os resultados reduzidos são armazenados de volta de maneira contígua na memória com a instrução `VST1`, otimizando o uso da hierarquia de cache e evitando penalidades de latência em acessos desalinhados.

Tal abordagem aproveita de forma mais direta a capacidade de execução paralela da arquitetura ARM do que a proposta no ML-KEM. Dessa forma, a combinação do processamento vetorizado com o carregamento e armazenamento eficientes resultou em uma expressiva redução no consumo de ciclos de clock e, consequentemente, em uma aceleração de até 7,0x em relação ao código de referência.

Geração da Matriz A com SHAKE2x

Inicialmente, foram exploradas bibliotecas de código otimizado para ARMv8, incluindo a XKCP e OpenSSL no intuito de empregar uma versão eficiente da função de permutação *Keccak*, que é recorrentemente utilizada e chega a custar mais de 30% do processo de geração de chaves de ambos os esquemas. Contudo, após avaliações experimentais, concluiu-se que que essas bibliotecas não resultaram em reduções significativas no tempo de execução para a geração da matriz A.

Essa limitação conduziu a uma busca por soluções com melhor aproveitamento dos recursos da plataforma alvo, especialmente pela função `KeccakF1600_StatePermute` que é responsável por aplicar a permutação de 1600 bits no estado interno do algoritmo Keccak, sendo o núcleo das funções de resumo criptográfico da família SHA-3 e das funções de expansão SHAKE. Essa permutação executa uma sequência estruturada de operações lógicas, incluindo rotações circulares, deslocamentos de bits, operações XOR e combinações não lineares que juntas geram custo computacional relevante. Essas etapas são projetadas para garantir uma difusão completa dos bits de entrada e uma alta resistência a ataques, como análise diferencial e ataques de colisão. Portanto, encontrar uma versão otimizada desta função torna-se necessário para produzir uma implementação performática do ML-KEM e do ML-DSA.

Assim, foi adotada a solução para `KeccakF1600_StatePermute` disponível na biblioteca ARMED KECCAK que faz uso de instruções mais eficientes da arquitetura ARMv8 e possui *benchmarks* documentados com emprego de ambos os dispositivos usados neste trabalho.

Além disso, foi otimizada a função de rejeição uniforme, permitindo o processamento simultâneo de 4 coeficientes polinomiais por iteração. Esses coeficientes representam amostras uniformemente distribuídas no anel $\mathbb{Z}_q$, fundamentais para a correta geração dos elementos da matriz A. Tal modificação minimizou o número de iterações necessárias para validar e gerar os coeficientes dentro do intervalo aceitável.

Complementando essa abordagem, foi implementada a função `SHAKE128_2x`, capaz de processar duas instâncias de SHAKE128 simultaneamente, explorando o paralelismo de dados com as instruções NEON. Esse processamento paralelo de dois estados independentes permitiu a geração simultânea de dois conjuntos de coeficientes da matriz A, resultando em uma aceleração de até 2.72x no ML-KEM e de até 2.56x no ML-DSA.

Capítulo 6

Resultados

Os experimentos para avaliação do desempenho da implementação otimizada foram realizados com auxílio do Google Benchmark na versão 1.9.0. A ferramenta faz uso de iterações de aquecimento (*warm-up iterations*) para estabilizar a execução antes das medições principais, tornando possível desconsiderar ruídos momentâneos, e realiza iterações automáticas até atingir uma precisão estatística aceitável.

Os dispositivos empregados foram o MacBook Air com o chip Apple M1 (8GB RAM) e o MacBook Air com chip Apple M2 (8GB RAM), que possuem uma arquitetura ARMv8 com suporte a instruções NEON. O compilador utilizado foi o Clang 16 e o sistema operacional o MacOS Sonoma 14.3 no M1 e o MacOS Sonoma 14.6 Sonoma no M2. Os resultados são apresentados nas Tabelas 6.1 a 6.4 e incluem o número de ciclos de CPU necessários para as três operações principais dos algoritmos ML-KEM e ML-DSA: geração de chaves, encapsulamento e desencapsulamento para o primeiro algoritmo; e geração de chaves, assinatura e verificação para o segundo. Para maximizar a eficiência dos algoritmos implementados, o código foi compilado utilizando as seguintes opções:

- `-O3`: Para otimizações agressivas, incluindo desdobramento de laços, o que permite reduzir a sobrecarga de controle e aumentar a paralelização.
- `-fno-omit-frame-pointer`: Para facilitar a coleta de informações detalhadas de *profiling*, permitindo uma análise mais precisa do uso de ciclos de *clock*.
- `-march=armv8-a+simd`: Para habilitar o uso de instruções NEON, permitindo operações SIMD.

O código está disponível em `https://github.com/everaldoalves/ML-KEM` e `https://github.com/everaldoalves/ML-DSA`.

Análise Comparativa de Desempenho

As Tabelas 6.1 e 6.2 resumem os resultados das medições para cada conjunto de parâmetros do ML-KEM, enquanto as Tabelas 6.3 e 6.4 apresentam os dados obtidos para o ML-DSA. Foi realizada uma comparação das implementações de referência [1] [14] com as implementações produzidas neste trabalho e calculada a melhoria com base nos ciclos:

$$\texttt{Aceleração} = (\texttt{Ciclos Referência}/\texttt{Ciclos Otimizada}).$$

Resultados mais detalhados com os dados relativos aos subalgoritmos mais relevantes podem ser encontrados no Apêndice A .

Apple M1				
Variante	**Algoritmo**	**Impl. Ref.**	**Este Trabalho**	**Aceleração (x)**
ML-KEM-512	KeyGen	430	166	2.59
	Encaps	510	204	2.50
	Decaps	660	275	2.40
ML-KEM-768	KeyGen	743	336	2.21
	Encaps	812	350	2.32
	Decaps	1018	489	2.08
ML-KEM-1024	KeyGen	1171	415	2.82
	Encaps	1194	466	2.56
	Decaps	1473	618	2.38

Tabela 6.1: Contagens de ciclos no Apple M1 para os três níveis de segurança do ML-KEM em comparação com [1]

Apple M2				
Variante	**Algoritmo**	**Impl. Ref.**	**Este Trabalho**	**Aceleração (x)**
ML-KEM-512	KeyGen	413	163	2.53
	Encaps	479	192	2.49
	Decaps	612	263	2.33
ML-KEM-768	KeyGen	701	305	2.29
	Encaps	759	342	2.21
	Decaps	948	459	2.07
ML-KEM-1024	KeyGen	1094	395	2.77
	Encaps	1134	441	2.57
	Decaps	1388	587	2.36

Tabela 6.2: Contagens de ciclos no Apple M2 para os três níveis de segurança do ML-KEM em comparação com [1]

Apple M1				
Variante	**Algoritmo**	**Impl. Ref.**	**Este Trabalho**	**Aceleração (x)**
ML-DSA-44	KeyGen	1282	613	2.09
	Sign	5937	2518	2.36
	Verify	1418	659	2.15
ML-DSA-65	KeyGen	2553	997	2.56
	Sign	10964	4110	2.67
	Verify	2472	1081	2.29
ML-DSA-87	KeyGen	3515	1646	2.14
	Sign	12290	4739	2.59
	Verify	3708	1666	2.23

Tabela 6.3: Contagens de ciclos no Apple M1 para os três níveis de segurança do ML-DSA em comparação com [14]

Apple M2				
Variante	**Algoritmo**	**Impl. Ref.**	**Este Trabalho**	**Aceleração (x)**
ML-DSA-44	KeyGen	1134	565	2.00
	Sign	5348	2333	2.29
	Verify	1252	610	2.05
ML-DSA-65	KeyGen	2298	926	2.48
	Sign	10044	3801	2.64
	Verify	2222	1008	2.20
ML-DSA-87	KeyGen	3115	1550	2.00
	Sign	10338	4490	2.30
	Verify	3286	1570	2.09

Tabela 6.4: Contagens de ciclos no Apple M2 para os três níveis de segurança do ML-DSA em comparação com [14]

A análise dos resultados de *benchmark* dos esquemas revelou aumento de desempenho em todas as variantes. Embora haja pequenas diferenças, os ganhos são consistentes em ambos os dispositivos utilizados, evidenciando que o código foi otimizado de forma satisfatória. Houve melhorias significativas no total de ciclos após a otimização, especialmente nas funções de geração da matriz A, NTT e inversa, bem como na multiplicação de polinômios que desempenham um papel importante na eficiência geral dos algoritmos.

A geração da matriz A é um processo custoso necessário às três etapas principais dos esquemas. Com a aceleração dessa rotina, foi possível reduzir significativamente o tempo total de execução dos algoritmos. Essa melhoria foi atingida pela refatoração das rotinas para gerar dois polinômios de forma paralela em proveito à versão otimizada das funções `shake`.

A NTT e sua inversa, implementadas nas funções `poly_ntt` e `poly_invntt_tomont`, também mostraram-se impactantes para a eficiência global. A NTT é amplamente utilizada em ambos os esquemas para acelerar operações de multiplicação de polinômios, e as otimizações nessas rotinas resultaram em uma redução de até 4x nos ciclos de clock necessários no ML-KEM e de até 3.33x no ML-DSA.

Além disso, as funções de multiplicação ponto a ponto foram notoriamente otimizadas, com ganhos de até 7x como o alcançado em `poly_pointwise_montgomery`. Esta função é particularmente relevante em operações de verificação de assinaturas e encapsulamento de chaves, onde pequenas otimizações podem se traduzir em grandes melhorias no desempenho global.

Esses resultados ratificam que as otimizações realizadas nas funções de maior impacto computacional não apenas aceleraram o processo individualmente, mas tiveram um efeito cascata que beneficiou todo o desempenho do esquema.

Por fim, os experimentos realizados com o compilador CLANG foram repetidos empregando o compilador GCC na versão 14.2. Em média, o CLANG se mostrou aproximadamente 2,5% mais rápido, tornando-o a melhor escolha para arquitetura alvo.

Capítulo 7

Conclusões

Este trabalho apresentou uma implementação otimizada dos algoritmos Kyber e Dilithium, padronizados pelo NIST como ML-KEM e ML-DSA, tendo como alvo a arquitetura ARMv8-A. Os resultados obtidos mostram acelerações significativas em relação às implementações de referência, com reduções expressivas no número de ciclos de clock necessários para as operações de geração de chaves, encapsulamento, desencapsulamento, assinatura e verificação. Essas otimizações foram alcançadas por meio da exploração dos recursos disponíveis na arquitetura alvo, especialmente as instruções NEON.

No entanto, apesar dos avanços obtidos, ainda existem desafios e oportunidades de aprimoramento que podem ser explorados futuramente:

Amostragem com CBD: O suporte da arquitetura ARMv8-A para vetorização pode ser explorado de forma ainda mais abrangente. Futuras otimizações podem focar em maximizar essas capacidades, especialmente durante o processo de amostragem usando distribuição binomial centrada, o que beneficiará tanto o processo de geração de chaves quanto o processo de encapsulamento.

NTT: Embora a implementação da NTT e sua inversa já tenha proporcionado acelerações significativas de até 4x, outras técnicas podem ser exploradas para extrair desempenho adicional, como o uso de assembly para um controle mais refinado sobre o potencial do hardware e a exploração de paralelismo de tarefas de forma granular, utilizando bibliotecas como OpenMP, para otimizar a distribuição do cálculo da NTT entre múltiplos núcleos da CPU.

Integração com criptografia baseada em hardware: O uso de instruções de criptografia dedicadas no ARMv8, como AES, pode ser explorado para melhorar a eficiência de operações auxiliares em ambos os esquemas tratados nesta dissertação. O AES-CTR pode ser implementado como um gerador de números aleatórios seguro, conforme especificado na SP 800-90A, para a geração de bytes aleatórios de forma compatível com as diretrizes de segurança do NIST. Essa abordagem permite aproveitar o suporte nativo de hardware para AES, o que pode resultar em melhorias de performance sem comprometer a conformidade com os padrões estabelecidos.

Em conclusão, este trabalho abre portas para uma série de otimizações futuras que podem melhorar ainda mais o desempenho dos algoritmos na plataforma ARMv8, contribuindo para o uso mais eficiente dos recursos computacionais que empregam tal arquitetura.

Referências Bibliográficas

[1] Robert Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. Third-round submission to the NIST's post-quantum cryptography standardization process, 2022.

[2] Melissa Azouaoui, Yulia Kuzovkova, Tobias Schneider, and Christine van Vredendaal. Post-quantum authenticated encryption against chosen-ciphertext side-channel attacks. Cryptology ePrint Archive, Paper 2022/916, 2022. `https://eprint.iacr.org/2022/916`.

[3] Elaine B. Barker and John M. Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication SP 800-90A, Rev. 1, National Institute of Standards and Technology (NIST), Gaithersburg, MD, June 2015.

[4] Elaine B. Barker, John M. Kelsey, Kerry McKay, Allen Roginsky, and Meltem Sönmez Turan. Recommendation for random bit generator (rbg) constructions. NIST Special Publication SP 800-90C (Third Public Draft), National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2022.

[5] 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. Cryptology ePrint Archive, Paper 2021/986, 2021.

[6] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, St. Petersburg, Russia, May 28 - June 1 2006. Springer.

[7] Daniel J. Bernstein. Analyzing the complexity of reference post-quantum software: the case of lattice-based kems. Cryptology ePrint Archive, Paper 2023/1924, 2023. `https://eprint.iacr.org/2023/1924`.

[8] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent

division timings in kyber implementations. Cryptology ePrint Archive, Paper 2024/1049, 2024.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference. Submission to the NIST SHA-3 competition, 2011.

[10] Nina Bindel, Johannes Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquín García-Alfaro, editors, *Foundations and Practice of Security - 10th International Symposium, FPS 2017*, volume 10723 of *Lecture Notes in Computer Science*, pages 225–241, Nancy, France, Oct 2017. Springer.

[11] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 2018.

[12] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - a cache attack on the bliss lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference*, volume 9813 of *Lecture Notes in Computer Science*, pages 323–345, Santa Barbara, CA, USA, August 2016. Springer, Springer.

[13] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[14] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium – algorithm specifications and supporting documentation (version 3.1). Specification document (update from February 2021), feb 2021. Version 3.1.

[15] Thomas Espitau, Pierre-Alain Fouque, Benoit Gerard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures – exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. Cryptology ePrint Archive, Paper 2017/505, 2017.

[16] Jieyu Zheng et al. Optimized vectorization implementation of crystals-dilithium, 2023.

[17] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. *Specification v1.2*, January 2020.

[18] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference*, pages 537–554, Santa Barbara, California, USA, Aug 1999. Springer. [S.l.].

[19] Qian Guo, Thomas Johansson, and Alexander Nilsson. A key-recovery timing attack on post-quantum primitives using the fujisaki-okamoto transformation and its application on FrodoKEM. Cryptology ePrint Archive, Paper 2020/743, 2020. `https://eprint.iacr.org/2020/743`.

[20] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *Theory of Cryptography: 15th International Conference, TCC 2017*, volume Part I, pages 341–371, Baltimore, MD, USA, Nov 2017. Springer. [S.l.].

[21] Weng Hong. Proteção contra ataques de canal lateral. *Journal of Cryptographic Security*, 15(3):123–135, 2021.

[22] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. Power analysis on ntru prime. *Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):123–151, 2019.

[23] Jonathan Katz. *Digital Signatures*. Springer, 2010.

[24] John Kelsey, Shu-jen Chang, and Ray Perlner. SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash. NIST Special Publication 800-185, National Institute of Standards and Technology, Gaithersburg, MD, 2016.

[25] Youngbeom Kim, Jingyo Song, Taek-Young Youn, and Seog Chung Seo. Crystals-dilithium on armv8. *Security and Communication Networks*, 2022.

[26] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[27] Lucas Z. Ladeira, Erick N. Nascimento, João Paulo F. Ventura, Ricardo Dahab, Diego F. Aranha, and Julio C. Lopez Hernandez. Canais laterais em criptografia simétrica e de curvas elípticas: ataques e contramedidas. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSEG)*, 2016.

[28] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2012.

[29] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2009.

[30] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. National Institute of Standards and Technology, 2016.

[31] National Institute of Standards and Technology. Module-lattice-based key encapsulation mechanism standard. Federal Information Processing Standards Publication (FIPS), 2023. NIST FIPS 203 ipd.

[32] Duc Tri Nguyen and Kris Gaj. Optimized software implementations of crystals-kyber, ntru, and saber using neon-based special instructions of armv8. *Journal of Cryptographic Engineering*, 11(4):291–306, 2021.

[33] National Institute of Standards and Technology. Post-quantum cryptography | csrc - nist computer security resource center. `https://csrc.nist.gov/Projects/post-quantum-cryptography/workshops-and-timeline`. Acesso em 3 de maio de 2024.

[34] National Institute of Standards and Technology. Sha-3 standard: Permutation-based hash and extendable-output functions. `https://csrc.nist.gov/pubs/fips/202/ipd`, 2015. Accesso em 25 de junho de 2024.

[35] National Institute of Standards and Technology. Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates, 2022.

[36] National Institute of Standards and Technology. Module-lattice-based digital signature standard. `https://csrc.nist.gov/pubs/fips/204/ipd`, 2023. Accesso em 25 de maio de 2024.

[37] National Institute of Standards and Technology. Stateless hash-based digital signature standard. `https://csrc.nist.gov/pubs/fips/205/ipd`, 2023. Accesso em 25 de maio de 2024.

[38] National Institute of Standards and Technology. Recommendations for key-encapsulation mechanisms. Special Publication 800-227, National Institute of Standards and Technology, Gaithersburg, MD, 2024. Available online.

[39] Jheyne Ortiz, Félix Carvalho Rodrigues, Décio Gazzoni Filho, Caio Teixeira, Julio López, and Ricardo Dahab. Evaluation of crystals-kyber and saber on the armv8 architecture. In *Anais do XXII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 372–377, Porto Alegre, RS, Brasil, 2022. SBC.

[40] Chris Peikert. Public-key cryptosystems from the worst-case shortest vector problem: extended abstract. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 333–342. ACM Press, May/June 2009. 1, 2.

[41] Charles Rackoff and Daniel R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference*, volume 576

of *Lecture Notes in Computer Science*, pages 433–444, Santa Barbara, California, USA, Aug 1991. Springer.

[42] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. Generic side-channel attacks on cca-secure lattice-based pke and kems. *Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):307–335, 2020.

[43] Pratyay Ravi, Mehedi Alam, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. GoFetch: Attacking post-quantum cryptography with time-to-frequency domain side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):537–562, 2022.

[44] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. Association for Computing Machinery.

[45] 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, 1978.

[46] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Kyber on arm64: Compact implementations of kyber on 64-bit arm cortex-a processors. In Joaquin Garcia-Alfaro, Shujun Li, Radha Poovendran, Hervé Debar, and Moti Yung, editors, *Security and Privacy in Communication Networks*, pages 424–440, Cham, 2021. Springer International Publishing.

[47] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[48] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE, 1994.

[49] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. IACR Cryptology ePrint Archive, 2004. Page 332.

[50] Joseph H. Silverman and William Whyte. Timing attacks on ntruencrypt via variation in the number of hash calls. pages 208–224, 2007.

[51] J. Smith and R. Jones. Optimization of kyber on mobile platforms using simd neon. *Cryptology ePrint Archive*, (2022/5678), 2022.

[52] Meltem Sönmez Turan, Elaine B. Barker, John M. Kelsey, Kerry A. McKay, Mary L. Baish, and Mike Boyle. Recommendation for the entropy sources used for random bit generation. NIST Special Publication SP 800-90B, National Institute of Standards and Technology (NIST), Gaithersburg, MD, January 2018.

[53] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David Oswald. Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: The case study of kyber. Cryptology ePrint Archive, Report 2020/912, 2020. https://eprint.iacr.org/2020/912.

Apêndice A

Resultados Experimentais

A.1 ML-KEM

ML-KEM-512						
Função	**M1**			**M2**		
	Referência	**Otimizada**	**Aceleração(x)**	**Referência**	**Otimizada**	**Aceleração(x)**
gen_matrix	85	38	2.24	80	34	2.35
poly_ntt	24	7	3.43	22	8	2.75
poly_invntt_tomont	38	10	3.80	36	9	4.00
polyvec_basemul_acc_montgomery	33	7	4.71	30	7	4.29
crypto_kem_keypair	430	166	2.59	413	163	2.53
crypto_kem_enc	510	204	2.50	479	192	2.49
crypto_kem_dec	660	275	2.40	612	263	2.33

Tabela A.1: Comparação de desempenho (em ciclos) entre a implementação de referência e a otimizada do ML-KEM-512 nos processadores Apple M1 e M2.

ML-KEM-768						
Função	**M1**			**M2**		
	Referência	**Otimizada**	**Aceleração(x)**	**Referência**	**Otimizada**	**Aceleração(x)**
gen_matrix	191	90	2.12	180	80	2.25
poly_ntt	24	7	3.43	22	8	2.75
poly_invntt_tomont	38	10	3.80	36	9	4.00
polyvec_basemul_acc_montgomery	48	10	4.80	44	10	4.40
crypto_kem_keypair	743	336	2.21	701	305	2.29
crypto_kem_enc	812	350	2.32	759	342	2.21
crypto_kem_dec	1018	489	2.08	948	459	2.07

Tabela A.2: Comparação de desempenho (em ciclos) entre a implementação de referência e a otimizada do ML-KEM-768 nos processadores Apple M1 e M2.

ML-KEM-1024						
Função	**M1**			**M2**		
	Referência	**Otimizada**	**Aceleração(x)**	**Referência**	**Otimizada**	**Aceleração(x)**
gen_matrix	342	155	2.21	322	141	2.28
poly_ntt	24	8	3.00	22	8	2.75
poly_invntt_tomont	38	10	3.80	36	9	4.00
polyvec_basemul_acc_montgomery	63	14	4.50	59	13	4.54
crypto_kem_keypair	1171	415	2.82	1094	395	2.77
crypto_kem_enc	1194	466	2.56	1134	441	2.57
crypto_kem_dec	1473	618	2.38	1388	587	2.36

Tabela A.3: Comparação de desempenho (em ciclos) entre a implementação de referência e a otimizada do ML-KEM-1024 nos processadores Apple M1 e M2.

A.2 ML-DSA

ML-DSA-44						
Função	**M1**			**M2**		
	Referência	**Otimizada**	**Aceleração(x)**	**Referência**	**Otimizada**	**Aceleração(x)**
polyvec_matrix_expand	552	212	2.60	461	214	2.15
poly_ntt	29	14	2.07	26	12	2.17
poly_invntt_tomont	36	15	2.40	32	12	2.67
poly_pointwise_montgomery	5	1	5.00	5	1	5.00
crypto_sign_keypair	1282	613	2.09	1134	565	2.01
crypto_sign_signature	5937	2518	2.36	5348	2333	2.29
crypto_sign_verify	1418	659	2.15	1252	610	2.05

Tabela A.4: Comparação de desempenho (em ciclos) entre a implementação de referência e a otimizada do ML-DSA-44 nos processadores Apple M1 e M2.

ML-DSA-65						
Função	**M1**			**M2**		
	Referência	**Otimizada**	**Aceleração(x)**	**Referência**	**Otimizada**	**Aceleração(x)**
polyvec_matrix_expand	1035	480	2.16	887	447	1.98
poly_ntt	36	14	2.57	33	12	2.75
poly_invntt_tomont	45	15	3.00	40	12	3.33
poly_pointwise_montgomery	7	1	7.00	7	1	7.00
crypto_sign_keypair	2553	997	2.56	2298	926	2.48
crypto_sign_signature	10964	4110	2.67	10044	3801	2.64
crypto_sign_verify	2472	1081	2.29	2222	1008	2.20

Tabela A.5: Comparação de desempenho (em ciclos) entre a implementação de referência e a otimizada do ML-DSA-65 nos processadores Apple M1 e M2.

ML-DSA-87						
Função	**M1**			**M2**		
	Referência	**Otimizada**	**Aceleração(x)**	**Referência**	**Otimizada**	**Aceleração(x)**
polyvec_matrix_expand	1933	851	2.27	1638	802	2.04
poly_ntt	29	14	2.07	27	12	2.25
poly_invntt_tomont	35	15	2.33	32	13	2.46
poly_pointwise_montgomery	5	1	5.00	5	1	5.00
crypto_sign_keypair	3515	1646	2.14	3115	1550	2.01
crypto_sign_signature	12290	4739	2.59	10338	4490	2.30
crypto_sign_verify	3708	1666	2.23	3286	1570	2.09

Tabela A.6: Comparação de desempenho (em ciclos) entre a implementação de referência e a otimizada do ML-DSA-87 nos processadores Apple M1 e M2.

Apêndice B

Algoritmos Auxiliares do ML-DSA

Algoritmo 22 IntegerToBits(x, α)

Entrada: Um número inteiro não negativo x e um número inteiro positivo α.
Saída: Uma string de bits y de comprimento α.

1: **for** i de 0 até $\alpha - 1$ **do**
2: $\quad y[i] \leftarrow x \mod 2$
3: $\quad x \leftarrow \lfloor x/2 \rfloor$
4: **end for**
5: **return** y

Algoritmo 23 BitsToInteger(y)

Entrada: Uma string de bits y de comprimento α.
Saída: Um número inteiro não negativo x.

1: $x \leftarrow 0$
2: **for** i de 1 até α **do**
3: $\quad x \leftarrow 2x + y[\alpha - i]$
4: **end for**
5: **return** x

Algoritmo 24 BitsToBytes(y)

Entrada: Uma string de bits y de comprimento c.
Saída: Uma string de bytes z.

1: $z \leftarrow 0^{\lceil c/8 \rceil}$
2: **for** i de 0 até $c - 1$ **do**
3: $\quad z[\lfloor i/8 \rfloor] \leftarrow z[\lfloor i/8 \rfloor] + y[i] \cdot 2^{(i \mod 8)}$
4: **end for**
5: **return** z

Algoritmo 25 BytesToBits(z)

Entrada: Uma string de bytes z de comprimento d.
Saída: Uma string de bits y.

for i de 0 até $d - 1$ **do**
    for j de 0 até 7 **do**
        $y[8i + j] \leftarrow z[i] \mod 2$
        $z[i] \leftarrow \lfloor z[i]/2 \rfloor$
    end for
end for
return y

Algoritmo 26 CoefFromThreeBytes(b0, b1, b2)

Entrada: Bytes $b0$, $b1$, $b2$.
Saída: Um inteiro módulo q ou $\perp$.

if $b2 > 127$ **then**
    $b2 \leftarrow b2 - 128$
end if
$z \leftarrow 65536 \cdot b2 + 256 \cdot b1 + b0$
if $z < q$ **then**
    return z
else
    return $\perp$
end if

Algoritmo 27 CoefFromHalfByte(b)

Entrada: Inteiro $b \in \{0, 1, \ldots, 15\}$.
Saída: Um inteiro entre $-\eta$ e η, ou $\perp$.

if $\eta = 2$ and $b < 15$ **then**
    return $2 - (b \mod 5)$
else if $\eta = 4$ and $b < 9$ **then**
    return $4 - b$
else
    return $\perp$
end if

Algoritmo 28 SimpleBitPack(w, b)

Entrada: $b \in \mathbb{N}$ e $w \in \mathbb{R}$ tal que os coeficientes de w estão todos em $[0, b]$.
Saída: Uma string de bytes de comprimento $32 \cdot \text{bitlen } b$.

$z \leftarrow ()$
for i de 0 até 255 **do**
    $z \leftarrow z \,\|\, \text{IntegerToBits}(w_i, \text{bitlen } b)$
end for
return BitsToBytes(z)

Algoritmo 29 BitPack(w, a, b)

Entrada: $a, b \in \mathbb{N}$ e $w \in \mathbb{R}$ tal que os coeficientes de w estão todos em $[-a, b]$.
Saída: Uma string de bytes de comprimento $32 \cdot \text{bitlen}\ (a + b)$.

$z \leftarrow ()$
for i de 0 até 255 **do**
    $z \leftarrow z \parallel \text{IntegerToBits}(b - w_i, \text{bitlen}\ (a + b))$
end for
return BitsToBytes(z)

Algoritmo 30 SimpleBitUnpack(v, b)

Entrada: $b \in \mathbb{N}$ e uma string de bytes v de comprimento $32 \cdot \text{bitlen}\ b$.
Saída: Um polinômio $w \in \mathbb{R}$, com coeficientes em $[0, 2^c - 1]$, onde $c = \text{bitlen}\ b$.

$c \leftarrow \text{bitlen}\ b$
$z \leftarrow \text{BytesToBits}(v)$
for i de 0 até 255 **do**
    $w_i \leftarrow \text{BitsToInteger}((z[ic], z[ic + 1], \ldots, z[ic + c - 1]), c)$
end for
return w

Algoritmo 31 BitUnpack(v, a, b)

Entrada: $a, b \in \mathbb{N}$ e uma string de bytes v de comprimento $32 \cdot \text{bitlen}\ (a + b)$.
Saída: Um polinômio $w \in \mathbb{R}$, com coeficientes em $[b - 2^c + 1, b]$, onde $c = \text{bitlen}\ (a + b)$.

$c \leftarrow \text{bitlen}\ (a + b)$
$z \leftarrow \text{BytesToBits}(v)$
for i de 0 até 255 **do**
    $w_i \leftarrow b - \text{BitsToInteger}((z[ic], z[ic + 1], \ldots, z[ic + c - 1]), c)$
end for
return w

Algoritmo 32 HintBitPack(h)

Entrada: Um vetor polinomial $h \in \mathbb{R}_2^k$ tal que no máximo ω dos coeficientes em h são iguais a 1.
Saída: Uma string de bytes y de comprimento $\omega + k$.

$y \in B^{\omega+k} \leftarrow 0^{\omega+k}$ ▷ Inicializa y com zeros
Index $\leftarrow 0$
for i de 0 até $k - 1$ **do**
    for j de 0 até 255 **do**
        if $h[i][j] = 1$ **then**
            $y[\text{Index}] \leftarrow j$ ▷ Armazena as localizações dos coeficientes não-zero em $h[i]$
            Index $\leftarrow$ Index $+ 1$
        end if
    end for
    $y[\omega + i] \leftarrow$ Index ▷ Armazena o valor de Index após processar $h[i]$
end for
return y

Algoritmo 33 HintBitUnpack(y)

Entrada: Uma string de bytes y de comprimento $\omega + k$.
Saída: Um vetor polinomial $h \in \mathbb{R}_2^k$ ou $\perp$.

$h \in \mathbb{R}_2^k \leftarrow 0$ ▷ Inicializa h com zeros
Index $\leftarrow 0$
for i de 0 até $k - 1$ **do**
    if $y[\omega + i] <$ Index or $y[\omega + i] > \omega$ **then**
        return $\perp$ ▷ Verifica se os índices são válidos
    end if
    while Index $< y[\omega + i]$ **do**
        $h[i][y[\text{Index}]] \leftarrow 1$
        Index $\leftarrow$ Index $+ 1$
    end while
end for
while Index $< \omega$ **do**
    if $y[\text{Index}] = 0$ **then**
        return $\perp$ ▷ Verifica se todos os índices restantes são válidos
    end if
    Index $\leftarrow$ Index $+ 1$
end while
return h

Algoritmo 34 pkEncode($\rho, \mathbf{t}_1$)

Entrada: $\rho \in \mathbb{B}^{32}$, $\mathbf{t}_1 \in R^k$ com coeficientes em $[0, 2^{\text{bitlen}(q-1)-d} - 1]$.
Saída: Chave pública $pk \in \mathbb{B}^{32+32k\cdot\text{bitlen}(q-1)-d}$.

$pk \leftarrow \rho$
for i de 0 até $k - 1$ **do**
    $pk \leftarrow pk \parallel \text{SimpleBitPack}(\mathbf{t}_1[i], 2^{\text{bitlen}(q-1)-d} - 1)$
end for
return pk

Algoritmo 35 pkDecode(pk)

Entrada: Chave pública $pk \in \mathbb{B}^{32+32k\cdot(\text{bitlen}(q-1)-d)}$.
Saída: $\rho \in \mathbb{B}^{32}$, $\mathbf{t}_1 \in R^k$ com coeficientes em $[0, 2^{\text{bitlen}(q-1)-d} - 1]$.

$(\rho, z_0, \ldots, z_{k-1}) \in \mathbb{B}^{32} \times (\mathbb{B}^{32\cdot(\text{bitlen}(q-1)-d)})^k \leftarrow pk$
for i de 0 até $k - 1$ **do**
    $\mathbf{t}_1[i] \leftarrow \text{SimpleBitUnpack}(z_i, 2^{\text{bitlen}(q-1)-d} - 1)$ ▷ Está sempre no intervalo correto
end for
return $(\rho, \mathbf{t}_1)$

Algoritmo 36 skEncode$(\rho, K, t_r, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

Entrada: $\rho \in \mathbb{B}^{32}$, $K \in \mathbb{B}^{32}$, $t_r \in \mathbb{B}^{64}$, $\mathbf{s}_1 \in R^{\ell}$ com coeficientes em $[-\eta, \eta]$, $\mathbf{s}_2 \in R^{k}$ com coeficientes em $[-\eta, \eta]$, $\mathbf{t}_0 \in R^{k}$ com coeficientes em $[-2^{d-1}+1, 2^{d-1}]$.
Saída: Chave privada $sk \in \mathbb{B}^{32+32+64+32\cdot((k+\ell)\cdot \text{bitlen}(2\eta)+dk)}$.

$sk \leftarrow \rho \parallel K \parallel t_r$
for i de 0 até $\ell - 1$ **do**
    $sk \leftarrow sk \parallel \text{BitPack}(\mathbf{s}_1[i], \eta, \eta)$
end for
for i de 0 até $k - 1$ **do**
    $sk \leftarrow sk \parallel \text{BitPack}(\mathbf{s}_2[i], \eta, \eta)$
end for
for i de 0 até $k - 1$ **do**
    $sk \leftarrow sk \parallel \text{BitPack}(\mathbf{t}_0[i], 2^{d-1} - 1, 2^{d-1})$
end for
return sk

Algoritmo 37 skDecode(sk)

Entrada: Chave privada $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen}(2\eta)+dk)}$.
Saída: $\rho \in \mathbb{B}^{32}$, $K \in \mathbb{B}^{32}$, $t_r \in \mathbb{B}^{64}$, $\mathbf{s}_1 \in R^{\ell}$, $\mathbf{s}_2 \in R^{k}$, $\mathbf{t}_0 \in R^{k}$ com coeficientes em $[-2^{d-1}+1, 2^{d-1}]$.

$(\rho, K, t_r, y_0, \ldots, y_{\ell-1}, z_0, \ldots, z_{k-1}, w_0, \ldots, w_{k-1}) \in \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{64} \times (\mathbb{B}^{32\cdot \text{bitlen}(2\eta)})^{\ell} \times (\mathbb{B}^{32\cdot \text{bitlen}(2\eta)})^{k} \times (\mathbb{B}^{32d})^{k} \leftarrow sk$
for i de 0 até $\ell - 1$ **do**
    $\mathbf{s}_1[i] \leftarrow \text{BitUnpack}(y_i, \eta, \eta)$ ▷ Pode estar fora de $[-\eta, \eta]$ se a entrada estiver malformada
end for
for i de 0 até $k - 1$ **do**
    $\mathbf{s}_2[i] \leftarrow \text{BitUnpack}(z_i, \eta, \eta)$ ▷ Pode estar fora de $[-\eta, \eta]$ se a entrada estiver malformada
end for
for i de 0 até $k - 1$ **do**
    $\mathbf{t}_0[i] \leftarrow \text{BitUnpack}(w_i, 2^{d-1} - 1, 2^{d-1})$ ▷ Está sempre no intervalo correto
end for
return $(\rho, K, t_r, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

Algoritmo 38 sigEncode$(\tilde{c}, \mathbf{z}, \mathbf{h})$

Entrada: $\tilde{c} \in \mathbb{B}^{\lambda/4}$, $\mathbf{z} \in R^{\ell}$ com coeficientes em $[-\gamma_1 + 1, \gamma_1]$, $\mathbf{h} \in R^{k}$.
Saída: Assinatura $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$.

$\sigma \leftarrow \tilde{c}$
for i de 0 até $\ell - 1$ **do**
    $\sigma \leftarrow \sigma \parallel \text{BitPack}(\mathbf{z}[i], \gamma_1 - 1, \gamma_1)$
end for
$\sigma \leftarrow \sigma \parallel \text{HintBitPack}(\mathbf{h})$
return σ

Algoritmo 39 sigDecode(σ)

Entrada: Assinatura $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$.
Saída: $\tilde{c} \in \mathbb{B}^{\lambda/4}$, $\mathbf{z} \in R^{\ell}$ com coeficientes em $[-\gamma_1+1, \gamma_1]$, $\mathbf{h} \in R^{k}$, ou $\perp$.
1: $(\tilde{c}, x_0, \ldots, x_{\ell-1}, y) \in \mathbb{B}^{\lambda/4} \times (\mathbb{B}^{32\cdot(1+\text{bitlen}(\gamma_1-1))})^{\ell} \times \mathbb{B}^{\omega+k} \leftarrow \sigma$
2: **for** i de 0 até $\ell - 1$ **do**
3: $\quad \mathbf{z}[i] \leftarrow \text{BitUnpack}(x_i, \gamma_1 - 1, \gamma_1)$ ▷ Está no intervalo correto, pois γ_1 é uma potência de 2
4: **end for**
5: $\mathbf{h} \leftarrow \text{HintBitUnpack}(y)$
6: **return** $(\tilde{c}, \mathbf{z}, \mathbf{h})$

Algoritmo 40 w1Encode($\mathbf{w}_1$)

Entrada: $\mathbf{w}_1 \in R^{k}$ cujas coordenadas polinomiais têm coeficientes em $[0, \frac{q-1}{2\gamma_2} - 1]$.
Saída: Uma representação em string de bytes $\tilde{\mathbf{w}}_1 \in \mathbb{B}^{32k\cdot\text{bitlen}(\frac{q-1}{2\gamma_2}-1)}$.
1: $\tilde{\mathbf{w}}_1 \leftarrow ()$
2: **for** i de 0 até $k - 1$ **do**
3: $\quad \tilde{\mathbf{w}}_1 \leftarrow \tilde{\mathbf{w}}_1 \parallel \text{SimpleBitPack}(\mathbf{w}_1[i], \frac{q-1}{2\gamma_2} - 1)$
4: **end for**
5: **return** $\tilde{\mathbf{w}}_1$

Algoritmo 41 SampleInBall(ρ)

Entrada: Uma semente $\rho \in \mathbb{B}^{\lambda/4}$.
Saída: Um polinômio $c \in R$ com coeficientes em $\{-1, 0, 1\}$ e peso de Hamming $\tau \leq 64$.
1: $c \leftarrow 0$
2: $\text{ctx} \leftarrow H.\text{Init}()$
3: $\text{ctx} \leftarrow H.\text{Absorb}(\text{ctx}, \rho)$
4: $(\text{ctx}, s) \leftarrow H.\text{Squeeze}(\text{ctx}, 8)$
5: $h \leftarrow \text{BytesToBits}(s)$ ▷ h é uma string de bits de comprimento 64
6: **for** i de $256 - \tau$ até 255 **do**
7: $\quad (\text{ctx}, j) \leftarrow H.\text{Squeeze}(\text{ctx}, 1)$
8: $\quad$ **while** $j > i$ **do** ▷ amostragem por rejeição em $\{0, \ldots, i\}$
9: $\quad\quad (\text{ctx}, j) \leftarrow H.\text{Squeeze}(\text{ctx}, 1)$
10: $\quad$ **end while** ▷ j é um byte pseudo-aleatório que é $\leq i$
11: $\quad c_i \leftarrow c_j$
12: $\quad c_j \leftarrow (-1)^{h[i+\tau-256]}$
13: **end for**
14: **return** c

Algoritmo 42 RejBoundedPoly(ρ)

Entrada: Uma semente $\rho \in \mathbb{B}^{66}$.
Saída: Um polinômio $a \in R$ com coeficientes em $[-\eta, \eta]$.

$j \leftarrow 0$
ctx $\leftarrow H$.Init()
ctx $\leftarrow H$.Absorb(ctx, ρ)
while $j < 256$ **do**
    $z \leftarrow H$.Squeeze(ctx, 1)
    $z_0 \leftarrow$ CoeffFromHalfByte($z \mod 16$)
    $z_1 \leftarrow$ CoeffFromHalfByte($\lfloor z/16 \rfloor$)
    if $z_0 \neq \perp$ **then**
        $a_j \leftarrow z_0$
        $j \leftarrow j + 1$
    end if
    if $z_1 \neq \perp$ **and** $j < 256$ **then**
        $a_j \leftarrow z_1$
        $j \leftarrow j + 1$
    end if
end while
return a

Algoritmo 43 RejNTTPoly(ρ)

Entrada: Uma semente $\rho \in \mathbb{B}^{34}$.
Saída: Um elemento $\hat{a} \in T_q$.

$j \leftarrow 0$
ctx $\leftarrow G$.Init()
ctx $\leftarrow G$.Absorb(ctx, ρ)
while $j < 256$ **do**
    (ctx, s) $\leftarrow G$.Squeeze(ctx, 3)
    $\hat{a}[j] \leftarrow$ CoeffFromThreeBytes($s[0], s[1], s[2]$)
    if $\hat{a}[j] \neq \perp$ **then**
        $j \leftarrow j + 1$
    end if
end while
return $\hat{a}$

Algoritmo 44 ExpandA(ρ)

Entrada: Uma semente $\rho \in \mathbb{B}^{32}$.
Saída: Matriz $\hat{A} \in (T_q)^{k \times \ell}$.

for r de 0 até $k - 1$ **do**
    for s de 0 até $\ell - 1$ **do**
        $\rho' \leftarrow \rho \parallel$ IntegerToBytes(s, 1) $\parallel$ IntegerToBytes(r, 1)
        $\hat{A}[r, s] \leftarrow$ RejNTTPoly(ρ') ▷ A semente ρ' depende de s e r
    end for
end for
return $\hat{A}$

Algoritmo 45 ExpandS(ρ)

Entrada: Uma semente $\rho \in \mathbb{B}^{64}$.
Saída: Vetores $s_1 \in R^\ell$ e $s_2 \in R^k$.
1: **for** r de 0 até $\ell - 1$ **do**
2: $\quad s_1[r] \leftarrow \text{RejBoundedPoly}(\rho \parallel \text{IntegerToBytes}(r, 2))$ ▷ A semente depende de r
3: **end for**
4: **for** r de 0 até $k - 1$ **do**
5: $\quad s_2[r] \leftarrow \text{RejBoundedPoly}(\rho \parallel \text{IntegerToBytes}(r + \ell, 2))$ ▷ A semente depende de $r + \ell$
6: **end for**
7: **return** (s_1, s_2)

Algoritmo 46 ExpandMask(ρ, μ)

Entrada: Uma semente $\rho \in \mathbb{B}^{64}$ e um número inteiro não negativo μ.
Saída: Vetor $y \in R^\ell$.
1: $c \leftarrow 1 + \text{bitlen}(\gamma_1 - 1)$ ▷ γ_1 é sempre uma potência de 2
2: **for** r de 0 até $\ell - 1$ **do**
3: $\quad \rho' \leftarrow \rho \parallel \text{IntegerToBytes}(\mu + r, 2)$
4: $\quad v \leftarrow H(\rho', 32c)$ ▷ A semente depende de $\mu + r$
5: $\quad y[r] \leftarrow \text{BitUnpack}(v, \gamma_1 - 1, \gamma_1)$
6: **end for**
7: **return** y

Algoritmo 47 Power2Round(r)

Entrada: $r \in \mathbb{Z}_q$.
Saída: Inteiros (r_1, r_0).
1: $r^+ \leftarrow r \mod q$
2: $r_0 \leftarrow r^+ \mod \pm 2^d$
3: **return** $\left(\frac{r^+ - r_0}{2^d}, r_0\right)$

Algoritmo 48 Decompose(r)

Entrada: $r \in \mathbb{Z}_q$.
Saída: Inteiros (r_1, r_0).
1: $r^+ \leftarrow r \mod q$
2: $r_0 \leftarrow r^+ \mod \pm (2\gamma_2)$
3: **if** $r^+ - r_0 = q - 1$ **then**
4: $\quad r_1 \leftarrow 0$
5: $\quad r_0 \leftarrow r_0 - 1$
6: **else**
7: $\quad r_1 \leftarrow \frac{r^+ - r_0}{2\gamma_2}$
8: **end if**
9: **return** (r_1, r_0)

Algoritmo 49 HighBits(r)

Entrada: $r \in \mathbb{Z}_q$.
Saída: Inteiro r_1.
1: $(r_1, r_0) \leftarrow$ Decompose(r)
2: **return** r_1

Algoritmo 50 LowBits(r)

Entrada: $r \in \mathbb{Z}_q$.
Saída: Inteiro r_0.
1: $(r_1, r_0) \leftarrow$ Decompose(r)
2: **return** r_0

Algoritmo 51 MakeHint(z, r)

Entrada: $z, r \in \mathbb{Z}_q$.
Saída: Valor booleano.
1: $r_1 \leftarrow$ HighBits(r)
2: $v_1 \leftarrow$ HighBits($r + z$)
3: **return** $[[r_1 \neq v_1]]$

Algoritmo 52 UseHint(h, r)

Entrada: Valor booleano h, $r \in \mathbb{Z}_q$.
Saída: $r_1 \in \mathbb{Z}$ com $0 \leq r_1 \leq \frac{q-1}{2\gamma_2}$.
1: $m \leftarrow \frac{q-1}{2\gamma_2}$
2: $(r_1, r_0) \leftarrow$ Decompose(r)
3: **if** $h = 1$ **and** $r_0 > 0$ **then**
4: **return** $(r_1 + 1) \mod m$
5: **else if** $h = 1$ **and** $r_0 \leq 0$ **then**
6: **return** $(r_1 - 1) \mod m$
7: **end if**
8: **return** r_1

Algoritmo 53 NTT(w)

Entrada: polinômio $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

Entrada: polinômio $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

1: **for** j de 0 até 255 **do**
2: $\quad \hat{w}[j] \leftarrow w_j$ ▷ Inicializa $\hat{w}$ com os coeficientes de w
3: **end for**
4: $k \leftarrow 0$
5: len $\leftarrow 128$
6: **while** len ≥ 1 **do**
7: $\quad$ start $\leftarrow 0$
8: $\quad$ **while** start < 256 **do**
9: $\quad\quad k \leftarrow k + 1$
10: $\quad\quad \zeta \leftarrow \zeta^{\text{brv}(k)} \mod q$ ▷ Calcula a raiz primitiva de unidade ζ
11: $\quad\quad$ **for** j de start até start + len − 1 **do**
12: $\quad\quad\quad t \leftarrow \zeta \cdot \hat{w}[j + \text{len}]$
13: $\quad\quad\quad \hat{w}[j + \text{len}] \leftarrow \hat{w}[j] - t$
14: $\quad\quad\quad \hat{w}[j] \leftarrow \hat{w}[j] + t$
15: $\quad\quad$ **end for**
16: $\quad\quad$ start $\leftarrow$ start $+ 2 \cdot$ len
17: $\quad$ **end while**
18: $\quad$ len $\leftarrow \lfloor \text{len}/2 \rfloor$ ▷ Reduz o tamanho pela metade a cada iteração
19: **end while**
20: **return** $\hat{w}$

Algoritmo 54 $\text{NTT}^{-1}(\hat{w})$

Entrada: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$.
Saída: polinômio $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$.

for j de 0 até 255 **do**
    $w_j \leftarrow \hat{w}[j]$ ▷ Inicializa w_j com os valores de $\hat{w}$
end for
$k \leftarrow 256$
$len \leftarrow 1$
while $len < 256$ **do**
    $start \leftarrow 0$
    while $start < 256$ **do**
        $k \leftarrow k - 1$
        $\zeta \leftarrow -\zeta^{\text{brv}(k)} \mod q$ ▷ Calcula a raiz primitiva de unidade negativa
        for j de $start$ até $start + len - 1$ **do**
            $t \leftarrow w_j$
            $w_j \leftarrow t + w_{j+len}$
            $w_{j+len} \leftarrow t - w_{j+len}$
            $w_{j+len} \leftarrow \zeta \cdot w_{j+len}$
        end for
        $start \leftarrow start + 2 \cdot len$
    end while
    $len \leftarrow 2 \cdot len$
end while
$f \leftarrow 8347681$ ▷ $f = 256^{-1} \mod q$
for j de 0 até 255 **do**
    $w_j \leftarrow f \cdot w_j$ ▷ Normaliza cada coeficiente
end for
return w

Algoritmo 55 $\text{BitRev8}(m)$

Entrada: Um byte $m \in [0, 255]$.
Saída: Um byte $r \in [0, 255]$.

$b \leftarrow \text{IntegerToBits}(m, 8)$
$b \in \{0,1\}^8 \leftarrow (0, \ldots, 0)$ ▷ Inicializa b
for i de 0 até 7 **do**
    $b[i] \leftarrow b[7 - i]$
end for
$r \leftarrow \text{BitsToInteger}(b, 8)$
return r

Algoritmo 56 $\text{AddNTT}(\hat{a}, \hat{b})$

Entrada: $\hat{a}, \hat{b} \in T_q$.
Saída: $\hat{c} \in T_q$.

for i de 0 até 255 **do**
    $\hat{c}[i] \leftarrow \hat{a}[i] + \hat{b}[i]$
end for
return $\hat{c}$

Algoritmo 57 MultiplyNTT($\hat{a}, \hat{b}$)

Entrada: $\hat{a}, \hat{b} \in T_q$.
Saída: $\hat{c} \in T_q$.
1: **for** i de 0 até 255 **do**
2: $\quad \hat{c}[i] \leftarrow \hat{a}[i] \cdot \hat{b}[i]$
3: **end for**
4: **return** $\hat{c}$

Algoritmo 58 AddVectorNTT($\hat{\mathbf{v}}, \hat{\mathbf{w}}$)

Entrada: $\ell \in \mathbb{N}$, $\hat{\mathbf{v}} \in T_q^\ell$, $\hat{\mathbf{w}} \in T_q^\ell$.
Saída: $\hat{\mathbf{u}} \in T_q^\ell$.
1: **for** i de 0 até $\ell - 1$ **do**
2: $\quad \hat{\mathbf{u}}[i] \leftarrow \text{AddNTT}(\hat{\mathbf{v}}[i], \hat{\mathbf{w}}[i])$
3: **end for**
4: **return** $\hat{\mathbf{u}}$

Algoritmo 59 ScalarVectorNTT($\hat{c}, \hat{\mathbf{v}}$)

Entrada: $\hat{c} \in T_q$, $\ell \in \mathbb{N}$, $\hat{\mathbf{v}} \in T_q^\ell$.
Saída: $\hat{\mathbf{w}} \in T_q^\ell$.
1: **for** i de 0 até $\ell - 1$ **do**
2: $\quad \hat{\mathbf{w}}[i] \leftarrow \text{MultiplyNTT}(\hat{c}, \hat{\mathbf{v}}[i])$
3: **end for**
4: **return** $\hat{\mathbf{w}}$

Algoritmo 60 Montgomery_Reduce(a)

Entrada: Inteiro a tal que $-2^{31}q \leq a \leq 2^{31}q$.
Saída: $r \equiv a \cdot 2^{-32} \mod q$ tal que $-q < r < q$.
1: $QINV \leftarrow 58728449$ ▷ O inverso de q módulo 2^{32}
2: $t \leftarrow ((a \mod 2^{32}) \cdot QINV) \mod 2^{32}$ ▷ Calcula t
3: $r \leftarrow (a - t \cdot q)/2^{32}$ ▷ Calcula r reduzindo a
4: **return** r