Impacto da otimização de funções hash no desempenho do algoritmo de assinatura digital pós-quântica CRYSTALS-Dilithium

Rodrigo Duarte de Meneses, Marco Aurélio Amaral Henriques

[1]Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas (Unicamp) – Campinas, SP – Brasil

r197962@dac.unicamp.br, maah@unicamp.br

Resumo. *O algoritmo de assinatura digital CRYSTALS-Dilithium é um dos algoritmos pós-quânticos recentemente padronizados pelo NIST. A partir de estudos na literatura indicando um overhead computacional associado às funções hash utilizadas no algoritmo, otimizamos sua implementação com a adoção da função hash TurboSHAKE. Tal otimização conferiu maior velocidade para o Dilithium em todos os níveis de segurança, sem alterar o uso de memória RAM.*

Abstract. *The CRYSTALS-Dilithium digital signature algorithm is one of the post-quantum algorithms recently standardized by NIST. From studies in the literature indicating a computational overhead associated with the hash functions used in the algorithm, we optimized its implementation with the adoption of the TurboSHAKE hash function. Such optimization conferred a higher speed to Dilithium at all levels of security, without changing the use of RAM.*

1. Introdução

Com o advento da computação digital, a tarefa de autenticação de informações se tornou indispensável para a segurança dos protocolos de comunicação. Por este motivo, a assinatura e verificação de dados digitais configura uma importante área de estudo para criptografia e compreende os chamados algoritmos de assinatura digital. Atualmente, de acordo com os padrões vigentes de criptografia assimétrica (ou de chave pública), os criptossistemas mais populares para esta aplicação são o RSA (Rivest-Shamir-Adleman) e ECDSA (*Elliptic Curve Digital Signature Algorithm*) [Stallings 2015], que têm sua segurança associada à dificuldade de fatorar (em tempo polinomial) inteiros grandes e calcular o logaritmo discreto de um dado valor em uma dada base, respectivamente.

A partir dos desenvolvimentos recentes da computação quântica e da criação do algoritmo de Shor [Shor 1997], esses problemas poderão ser resolvidos em tempo polinominal por um computador quântico suficientemente robusto. Logo, a segurança antes atribuída aos criptossistemas mais utilizados é colocada em risco, comprometendo grande parte das aplicações em segurança de dados. O projeto de esquemas criptográficos resistentes aos ataques de um computador quântico define a área da criptografia pós-quântica.

Nesse contexto, o *National Institute of Standards and Technology* (NIST) anunciou, em 2016, um processo de padronização para algoritmos pós-quânticos. Em 2022, após 6 anos de estudo, o NIST padronizou três algortimos de assinatura digital: FALCON, CRYSTALS-Dilithium e SPHINCS+ [NIST 2022]. Mais especificamente, o NIST indica a utilização do Dilithium como algoritmo primário, apontando o FALCON como

uma alternativa para aplicações que exijam menores tamanhos de assinatura. O algoritmo SPHINCS+, apesar de padronizado, ficou como uma opção de reserva, visto que é considerado seguro, mas demanda muitos recursos computacionais.

Neste trabalho, exploramos os resultados disponíveis na literatura [Aumasson 2019] que apontam para um *overhead* computacional associado às funções de hash criptográfico da família SHA-3, amplamente utilizadas no Dilithium para expansão de parâmetros internos como matrizes e vetores de polinômios. A substituição das funções SHAKE pelas versões com menor número de rodadas TurboSHAKE promete uma aceleração em *software* do algoritmo sem comprometer sua segurança [Bertoni et al. 2023]. São apresentados e discutidos os resultados preliminares dos impactos observados para as principais funções do Dilithium a partir dessa otimização.

2. CRYSTALS-Dilithium, MLWE e reticulados

Para compreender o algoritmo utilizado e seus mecanismos matemáticos, descrevemos brevemente seu funcionamento e os problemas matemáticos envolvidos em sua segurança [Lyubashevsky et al. 2021].

2.1. Reticulados e MLWE

O problema matemático sobre o qual o Dilithium adquire sua segurança é o problema *Module Learning With Errors* (MLWE), definido sobre um tipo de estrutura algébrica abstrata chamada reticulado. Um reticulado pode ser entendido como um espaço vetorial discreto, composto por um conjunto S de pontos distribuídos em um espaço n-dimensional. Cada ponto em S pode ser expresso em termos de uma base de n vetores em S linearmente independentes. Naturalmente, existem diferentes bases possíveis para um dado reticulado, permitindo a definição de bases boas ou ruins – dependendo do quão trivial é expressar os pontos em S em termos de uma combinação linear dos vetores desta base.

A partir dessas propriedades, é possível definir um número de problemas sobre essas estruturas algébricas que são de interesse particular para algoritmos de criptografia, como o problema LWE. Este problema consiste na resolução de equações matriciais, análogas à combinação linear de vetores em um reticulado S, com a inclusão de um ruído que impossibilita a solução do sistema de forma trivial utilizando o método de eliminação de Gauss-Jordan. Assim, dado $\mathbb{Z}_q$ um anel de inteiros módulo q e $\mathbb{Z}_q^n$ um conjunto de vetores n-dimensionais sobre $\mathbb{Z}_q$, o problema consiste na dificuldade de se distinguir um vetor qualquer $\mathbf{t} \in \mathbb{Z}_q^n$ de um vetor da forma $\mathbf{t} = \mathbf{A} \cdot \mathbf{s_1} + \mathbf{s_2}$, onde $\mathbf{s_1}$ e $\mathbf{s_2}$ pertencem a $\mathbb{Z}_q^n$ e $\mathbf{A}$ é uma matriz com elementos em $\mathbb{Z}_q^n$.

O problema MLWE é uma generalização do LWE em que tomamos um anel polinomial $\mathbb{Z}_q[X]/(X^n + 1)$ ao invés do anel de inteiros módulo q. Esses problemas são considerados de difícil resolução mesmo para um computador quântico; portanto, sua aplicação em algoritmos pós-quânticos é justificável. No Dilithium, os parâmetros de construção do anel polinomial são fixos, e todas as operações são realizadas sobre o anel $R = \mathbb{Z}_q[X]/(X^{256} + 1)$, com $q = 2^{23} - 2^{13} + 1$, para todos os níveis de segurança.

2.2. CRYSTALS-Dilithium

O CRYSTALS-Dilithium é um algoritmo de assinatura digital da suíte CRYSTALS (*Cryptographic Suite for Algebraic Lattices*), com design baseado no esquema *Fiat-Shamir with Aborts*, consistindo em três funções principais – geração de chaves (Gen), assinatura

digital (Sign) e verificação (Verify), cujos pseudocódigos (sem a compressão da chave pública) estão apresentados nos Algoritmos 1, 2 e 3, respectivamente. Durante todo este estudo foi utilizada a implementação do Dilithium disponível no seu repositório oficial – correspondente à versão submetida ao *Round* 3 do processo de padronização do NIST.

Algorithm 1 Pseudocódigo de Gen

1: $\mathbf{A} \leftarrow R_q^{k \times \ell}$
2: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$
3: $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
4: **return** $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$

Na função Gen, utiliza-se um gerador de números pseudo-aleatórios baseado em uma *seed* aleatória de 256 bits, que é expandida através de uma XOF (*eXtendable Output Function*) para geração dos elementos da matriz pública $\mathbf{A}$ e dos vetores privados $\mathbf{s_1}$ e $\mathbf{s_2}$. Todos esses elementos são polinômios em R. Essa etapa é indicada nas linhas 1 e 2 do algoritmo apresentado no Alg. 1. Como o armazenamento da matriz pública $\mathbf{A}$ é um processo custoso em termos de memória, na prática armazena-se somente a *seed* e gera-se a matriz $\mathbf{A}$ sob demanda.

Algorithm 2 Pseudocódigo de Sign(sk, M)

1: $\mathbf{z} := \perp$
2: **while** $\mathbf{z} = \perp$ **do**
3: $\quad \mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$
4: $\quad \mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
5: $\quad c \in B_\tau := \mathbf{H}(M \parallel \mathbf{w}_1)$
6: $\quad \mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
7: $\quad$ **if** $(\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta)$ or $(\|\text{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta)$ **then** $\mathbf{z} := \perp$
8: $\quad$ **end if**
9: **end while**
10: **return** $\sigma = (\mathbf{z}, c)$

Na função Sign, é gerado um vetor de mascaramento $\mathbf{y}$ (linha 3, Alg. 2), utilizado para geração da assinatura. O assinador computa $\mathbf{A}\mathbf{y}$ e define $\mathbf{w}_1$ como os "bits de alta ordem" dos coeficientes desse vetor. O desafio c (*challenge*) é gerado a partir do hash da mensagem M a ser assinada, concatenada à $\mathbf{w}_1$. Em seguida é produzida a potencial assinatura $\mathbf{z}$ utilizando a chave privada sk. Caso $\mathbf{z}$ já fosse entregue como saída, existiria a possibilidade de vazamento da chave privada. Por esse motivo, inclui-se uma verificação chamada de *rejection sampling* que garante que a assinatura não revelará informações sobre os parâmetros privados. Por fim, a saída é dada como o conjunto do desafio c e a assinatura $\mathbf{z}$ após o *rejection sampling* não ter detectado nenhum problema de vazamento.

Algorithm 3 Pseudocódigo de Verify$(pk, M, \sigma = (\mathbf{z}, c))$

1: $\mathbf{w}_1' := \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$
2: **return** $[\|z\|_\infty < \gamma_1 - \beta]$ **and** $[c = \mathbf{H}(M \parallel \mathbf{w}_1')]$

Na função Verify, indicada no Alg. 3, iniciamos calculando $\mathbf{w}_1'$ como os bits de alta ordem dos resultado $\mathbf{A}\mathbf{z} - c\mathbf{t}$. Deve-se destacar o fato de que para que haja a

correta verificação da assinatura, devemos ter $\mathbf{w}_1' = \mathbf{w}_1$ e portanto HighBits($\mathbf{Az} - c\mathbf{t}$) = HighBits($\mathbf{Ay}$). Importante notar que $\mathbf{Az} - c\mathbf{t} = \mathbf{Ay} - c\mathbf{s}_2$ e que a condição para assinatura válida é HighBits($\mathbf{Az} - c\mathbf{t}$) = HighBits($\mathbf{Ay} - c\mathbf{s}_2$), onde o tamanho reduzido do vetor de ruídos $\mathbf{s_2}$ é essencial para a corretude da etapa de verificação.

3. SHAKE e TurboSHAKE

As duas principais operações realizadas pelo Dilithium são expansão de parâmetros aleatórios através de XOFs, e multiplicação dentro do anel polinomial R, para as quais utiliza-se a NTT (*Number Theoretic Transform*). Assim, a variação do nível de segurança do Dilithium consiste em variar o número de operações realizadas sobre o anel polinomial e as expansões por XOF, o que confere grande flexibilidade ao algoritmo.

O Dilithium submete-se à norma FIPS 202, padronização do NIST sobre funções hash criptográficas da família SHA-3. O algoritmo faz uso do SHAKE-128 e SHAKE-256 como XOF para expansão de parâmetros a partir de uma *seed* de entrada. No entanto, conforme os resultados recentes da literatura, há um *overhead* computacional associado a essas funções, de forma que implementações otimizadas passaram a surgir como uma alternativa que não comprometa o nível de segurança [Aumasson 2019].

Uma das alternativas mais interessantes é o TurboSHAKE [Bertoni et al. 2023], uma implementação com rodadas reduzidas (12 rodadas, em comparação às 24 rodadas utilizadas no SHAKE tradicional) previamente utilizada no algoritmo KangarooTwelve, que implementa a permutação Keccak-p[1600, 12] com maior velocidade comparada às funções SHA-3 e SHAKE. Utilizando da mesma arquitetura em esponja (*sponge architecture*) do SHAKE, o TurboSHAKE possui também versões de 128 e 256 bits de segurança, permitindo a substituição do SHAKE-128 e SHAKE-256 por suas respectivas versões TurboSHAKE sem diminuição do nível de segurança. Essa mudança busca uma aceleração em *software* do algoritmo através economia de processamento associada às XOFs. É importante ressaltar que a otimização proposta deve considerar que SHAKE e TurboSHAKE não produzem a mesma saída para um dado conjunto de entrada; assim, a migração para sistemas já operacionais deve ser feita com maior cautela.

4. Metodologia experimental

Realizamos a medição de ciclos de CPU e picos de RAM para cada uma das três funções principais do Dilithium; Gen, Sign e Verify. Comparamos também o impacto da substituição das XOFs SHAKE-128 e SHAKE-256 por suas respectivas versões TurboSHAKE-128 e TurboSHAKE-256, denotando por v.0 e v.1 as versões padrão e otimizada do algoritmo, respectivamente. Utilizamos a implementação do TurboSHAKE conforme descrita e apresentada em [Bertoni et al. 2023]. Foram realizadas 1500 iterações para obtenção dos dados indicados, incluindo a geração de chaves, assinatura e verificação para uma mensagem aleatória de 59 bytes.

O Dilithium utiliza três níveis de segurança prescritos pelo NIST (níveis 2, 3 e 5), consistindo em diferentes conjuntos de parâmetros definidos na documentação oficial [Lyubashevsky et al. 2021] em ordem crescente de segurança. Realizamos as medições para todos os níveis de segurança disponíveis, a fim de entender melhor como a otimização das funções de hash se relaciona com os parâmetros do Dilithium.

Para realização das medidas foi utilizado um computador com processador Intel

Core i5-8265U, CPU 1.6 GHz, OS Ubuntu 20.04.6 LTS e memória RAM de 8 Gb. Utilizamos o *software* Massif, disponível no programa de avaliação Valgrind 3.15.0, como ferramenta para medição e monitoramento da utilização de memória RAM durante a execução de cada função. Para a medição de ciclos de CPU, utilizamos o programa perf, fazendo o cálculo da média e da mediana de ciclos para cada uma das funções do Dilithium. Escolhemos apresentar também as medianas dada a presença de alguns *outliers* (em particular na função Sign) que eram esperados dada a utilização de *rejection sampling*.

5. Resultados e discussões

Os resultados obtidos são apresentados nas Tabelas 1, 2 e 3, para os picos de RAM observados para cada função do Dilithium, e os valores de média e mediana de ciclos de CPU consumidos para as versões padrão (v.0) e otimizada com TurboSHAKE (v.1).

A partir da otimização implementada, conforme apresentado na Tabela 1, foi possível constatar que não houve impacto no consumo de memória entre as versões testadas. Isso já era esperado, uma vez que o algoritmo de hash otimizado tem a mesma estrutura do original. Nas Tabelas 2e 3 percebemos a variação percentual do número de ciclos de CPU para v.1 calculado em relação ao resultado obtido para v.0, indicando uma diminuição no consumo de CPU para todos os níveis de segurança, o que se justifica pela grande utilização de funções hash no Dilithium, especialmente na geração de chaves e verificação de assinaturas. Os dados indicam também uma maior economia em termos de ciclos de CPU para os níveis de segurança mais altos, uma vez que o Dilithium trabalha com vetores e matrizes de maior dimensão nesses níveis, exigindo mais das funções hash sendo otimizadas. Apesar dos números de ciclos apresentados aqui serem ligeiramente superiores aos mostrados no site oficial do projeto Dilithium [Lyubashevsky et al. 2021] devido às diferenças entre processadores, entendemos ser razoável inferir que também serão obtidos os mesmos níveis de redução de ciclos pela mudança da funções hash em outras implementações.

Tabela 1. Uso máximo de RAM (bytes) para níveis de segurança 2, 3 e 5

Pico de RAM para CRYSTALS-Dilithium v.0/v.1			
Nível de segurança	Gen	Sign	Verify
Dilithium 2	42.488 B	59.648 B	43.936 B
Dilithium 3	67.128 B	90.272 B	68.344 B
Dilithium 5	105.496 B	134.960 B	106.448 B

Tabela 2. Média de ciclos de CPU para cada função do Dilithium

Média de ciclos de CPU para o CRYSTALS-Dilithium			
Nível de segurança	Função	v.0	v.1 (variação %)
Dilithium 2	Gen	418.522	325.304 (-22.3%)
	Sign	1.844.839	1.674.893 (-9.2%)
	Verify	447.904	362.306 (-19.1%)
Dilithium 3	Gen	740.820	566.839 (-23.4%)
	Sign	2.781.223	2.525.570 (-9.2%)
	Verify	705.726	556.641 (-21.1%)
Dilithium 5	Gen	1.144.005	842.012 (-26.4%)
	Sign	3.523.184	3.255.138 (-7.6%)
	Verify	1.173.812	899.562 (-23.3%)

Tabela 3. Mediana de ciclos de CPU para cada função do Dilithium

Média de ciclos de CPU para o CRYSTALS-Dilithium			
Nível de segurança	Função	v.0	v.1 (variação %)
Dilithium 2	Gen	411.942	321.624 (-21.9%)
	Sign	1.490.671	1.279.631 (-14.2%)
	Verify	446.843	361.137 (-19.2%)
Dilithium 3	Gen	724.807	562.345 (-22.4%)
	Sign	2.230.749	2.080.019 (-6.8%)
	Verify	704.122	553.911 (-21.3%)
Dilithium 5	Gen	1.119.261	836.480 (-25.3%)
	Sign	3.026.118	2.665.533 (-11.9%)
	Verify	1.168.729	897.902 (-23.2%)

6. Conclusões e trabalhos futuros

A otimização das funções de hash utilizadas no algoritmo de assinatura pós-quântica CRYSTALS-Dilithium causou um impacto significativo no seu desempenho, sem diminuir sua segurança. Como trabalhos futuros, temos a otimização do custoso cálculo de NTT, buscando um compromisso entre memória e processamento. Adicionalmente, focando na implementação em dispositivos restritos, devemos buscar reduções no uso de memória RAM sem comprometer sua segurança.

Referências

Aumasson, J. P. (2019). Too much crypto. Cryptology ePrint Archive, Paper 2019/1492. https://eprint.iacr.org/2019/1492.

Bertoni, G., Daemen, J., Hoffert, S., Peeters, M., Assche, G. V., Keer, R. V., and Viguier, B. (2023). Turboshake. Cryptology ePrint Archive, Paper 2023/342. https://eprint.iacr.org/2023/342.

Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., and Stehlé, D. (2021). Crystals-dilithium: Algorithm specification and supporting documentation. https://pq-crystals.org/dilithium/.

NIST (2022). Nist announces first four quantum-resistant cryptographic algorithms. https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms. Acessado em 22/08/2023.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing.

Stallings, W. (2015). *Criptografia e Segurança de Redes: Princípios e Práticas*. Pearson.