Implementação e avaliação do algoritmo pós-quântico ML-KEM em *smart cards*

Fernando A. Penido[1], Caio Teixeira[2], Rodrigo D. Meneses[3], Marco A. Henriques[4]

Departamento de Computação e Automação (DCA) -
Faculdade de Engenharia Elétrica e de Computação (FEEC) -
Universidade Estadual de Campinas (UNICAMP)

{f281180[1], r197962[3]}@dac.unicamp.br, caio@dca.fee.unicamp.br[2], maah@unicamp.br[4]

Abstract. *This paper implements and evaluates the post-quantum key encapsulation mechanism ML-KEM on the Java Card platform, specifically targeting SIM and eSIM cards. To achieve this goal, on-the-fly vector generation and global buffers are used to address memory and performance restrictions in these environments. The achieved memory footprint enables the deployment of ML-KEM on resource-constrained cards with less than 7 kB of RAM, further driving the adoption of post-quantum cryptography in mobile networks.*

Resumo. *Este artigo implementa e avalia o algoritmo pós-quântico de encapsulamento de chaves ML-KEM na plataforma Java Card, especificamente visando SIMs e eSIMs como as plataformas finais. Para tal fim, são utilizadas técnicas de otimização como buffers globais e geração de vetores sob demanda para contornar limitações de memória e processamento nesses ambientes restritos. O consumo de memória alcançado permite implementar o ML-KEM em cartões com recursos restritos e menos de 7 kB de RAM, impulsionando ainda mais a adoção da criptografia pós-quântica em redes móveis.*

1. Introdução

Em 1994, Peter Shor publicou um artigo descrevendo um algoritmo para computadores quânticos que possibilita resolver os problemas da fatoração de inteiros grandes e do cálculo do logaritmo discreto sobre curvas elípticas em tempo polinomial [Shor 1994]. Em consequência, os algoritmos de chave pública tradicionais – como o RSA e a criptografia de curvas elípticas (ECC) – que possuem segurança baseada nas dificuldades desses problemas se tornarão vulneráveis a ataques baseados no algoritmo de Shor quando surgir um computador quântico suficientemente grande para executá-lo. Por isso, surgiu a necessidade de desenvolver algoritmos criptográficos seguros contra ataques de computadores quânticos, o que hoje é chamado de Criptografia Pós-Quântica (PQC).

Devido a técnicas como "Harvest Now, Decrypt Later" [Christof Paar et al. 2024], na qual um atacante guarda os dados criptografados transmitidos atualmente e posteriormente os decifra quando houver avanços significativos na tecnologia – como a criação de

O presente trabalho foi realizado com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq (Brasil), por meio de bolsa de Iniciação Científica (PIBIC). Nele foi utilizada a IA Gemini 3.1 Pro para auxílio na revisão de texto e depuração do código.

um computador quântico criptograficamente relevante (CQCR) –, o desenvolvimento de algoritmos pós-quânticos não é algo para o futuro, mas urgente [Moody et al. 2024].

Devido à vulnerabilidade dos algoritmos de chave pública atuais, o órgão de padronização americano *National Institute of Standards and Technology* (NIST) promoveu uma competição aberta para submissão de algoritmos pós-quânticos [NIST 2016], avaliados ao longo de quatro rodadas de seleção. Dentre os escolhidos, os únicos de encapsulamento de chaves foram o *Module-Lattice-Based Key-Encapsulation Mechanism* (ML-KEM) [NIST 2024b] e o *Hamming Quasi-Cyclic* (HQC) [Philippe Gaborit et al. 2025]. Porém, devido ao grande custo de memória e à baixa performance do HQC, o ML-KEM é recomendado para aplicações em geral, com o HQC como um substituto caso algoritmos baseados em reticulados sejam quebrados no futuro.

Contudo, o ML-KEM possui chaves pública e privada significativamente maiores que as empregadas em mecanismos tradicionais de troca de chaves, como o Elliptic Curve Diffie-Hellman. A otimização da implementação do algoritmo pós-quântico é, portanto, essencial para permitir seu uso em ambientes computacionalmente restritos. Em específico, protocolos de troca de chaves dentro de módulos de identidades precisam se adequar às limitações da plataforma de execução. Esse é o caso em cartões SIM e seus equivalentes embarcados, os eSIM, utilizados para identificar assinantes em redes móveis. A execução das operações criptográficas nesses módulos é recomendável devido à disponibilidade de coprocessadores especializados e de mecanismos de resistência à violação física, que favorecem o armazenamento seguro de chaves efêmeras e de longo prazo [NIST 2024a].

Nesse contexto, implementamos e avaliamos o algoritmo ML-KEM na plataforma Java Card, usando técnicas de otimização de memória para viabilizar sua execução em módulos de identidades. Os resultados obtidos revelam que tal implementação é viável, mesmo com as limitações de memória e processamento dos *smart cards*.

O restante do artigo é organizado da seguinte forma: a Seção 2 revisa a literatura pertinente ao escopo de algoritmos pós-quânticos em dispositivos com recursos limitados; a Seção 3 introduz os conceitos fundamentais para as otimizações propostas; a Seção 4 analisa os métodos de otimização; a Seção 5 apresenta os resultados das otimizações; e a Seção 6 conclui o trabalho e apresenta possíveis próximos passos.

2. Trabalhos relacionados

A literatura referente à implementação de algoritmos de criptografia pós-quântica em ambientes restritos é vasta; porém, implementações específicas para *smart cards* e módulos de identidade ainda são limitadas. O principal trabalho a respeito da implementação do ML-KEM em Java Card é o publicado por [Albrecht et al. 2018], que propõe uma implementação do Kyber (versão submetida ao NIST que originou o ML-KEM) em *smart cards* utilizando aceleradores de RSA e ECC para reduzir o tempo de execução do algoritmo. Porém, o código do trabalho não foi disponibilizado para o público, ainda que sejam descritos pseudocódigos de certas otimizações. Além desse, [Yuan et al. 2021] descrevem a implementação de um esquema criptográfico de chave pública baseado em *ring-LWE* na plataforma Java Card, implementando técnicas de otimização de memória e processamento para multiplicação modular de polinômios.

A fim de evidenciar os desafios relacionados à plataforma Java Card, é desta-

cado o estudo de [van der Laan et al. 2018]. Embora esteja fora do escopo de reticulados, ele implementa o algoritmo eXtended Merkle Signature Scheme (XMSS), também pós-quântico, em Java Card e com código aberto; porém, são reportados altos tempos de processamento para assinatura ($\approx$ 33 s) e para geração de chaves ($\approx$ 60 s).

No escopo da otimização do ML-KEM, [Hou et al. 2026] aplicam múltiplas técnicas de redução de consumo de memória, como geração de vetores sob demanda e reutilização de memória alocada para *buffers* intermediários. Ademais, [Botros et al. 2019] introduzem operações de compressão de polinômios e métodos de multiplicação que reduzem significativamente a utilização da memória RAM. Técnicas similares também foram empregadas para o Module-Lattice-Based Digital Signature Algorithm (ML-DSA) por [Meneses et al. 2024].

Além dos trabalhos acadêmicos, destacam-se implementações comerciais como a realizada pela FEITIAN Technologies. A partir de um sistema operacional próprio (FEITIAN Java Card Operating System) e de hardware dedicado, é possível executar algoritmos de criptografia pós-quântica [FEITIAN Technologies 2026]. Porém, essa aplicação ainda se diferencia deste trabalho ao apresentar uma implementação nativa em hardware, ao contrário da implementação em software proposta neste artigo.

2.1. Contribuição da pesquisa

Este trabalho apresenta, segundo o conhecimento dos autores, a primeira implementação em código aberto[1], completa e otimizada, do algoritmo pós-quântico ML-KEM na plataforma Java Card, contribuindo assim para a melhoria da segurança no uso de *smart cards*, especialmente cartões SIM/eSIM. A execução do algoritmo foi possibilitada pela utilização de técnicas de otimização de memória, como a geração de vetores sob demanda e a reutilização de *buffers* globais. Os resultados obtidos neste trabalho podem servir como referência sobre os requisitos para a execução do ML-KEM em um *smart card* ou SIM/eSIM, orientando os fabricantes sobre como viabilizar a segurança pós-quântica na troca de chaves, cuja adoção se torna cada vez mais necessária.

3. Conceitos Fundamentais

3.1. Criptografia baseada em reticulados

A criptografia baseada em reticulados, cuja segurança depende da dificuldade de resolver alguns problemas matemáticos sobre reticulados, é uma das vertentes mais promissoras no cenário de criptografia pós-quântica. Dentre os cinco algoritmos selecionados para padronização pelo NIST, três são baseados em reticulados.

Um reticulado no espaço real m-dimensional $\mathbb{R}^m$ é o conjunto de todas combinações lineares com coeficientes inteiros de n vetores linearmente independentes $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$, chamados conjuntamente de base do reticulado; ou seja, $\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \{\sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z}\}$ [Micciancio and Goldwasser 2002]. Um ponto no reticulado é, então, uma combinação linear específica desses vetores linearmente independentes.

Um dos principais problemas em reticulados, do ponto de vista da criptografia, é o problema *Learning With Errors* (LWE). Este problema consiste em recuperar um vetor que foi deslocado do seu ponto original no reticulado por um erro arbitrário mas limitado,

[1] https://github.com/regras/mlkem-on-simcard

o que dificulta a descoberta do vetor sem erro, especialmente em espaços com muitas dimensões. O problema pode ser estruturado como aquele de achar o vetor secreto $\mathbf{s}$ dada a matriz base do reticulado $\mathbf{A}$ e um vetor $\mathbf{t}$ tal que $\mathbf{A} \cdot \mathbf{s} + \mathbf{e} \equiv \mathbf{t} \pmod{q}$, sendo $\mathbf{e}$ o vetor secreto de erros [Micciancio and Regev 2009]. Há variantes mais adequadas desse problema para o propósito computacional criptográfico. Dentre elas, a mais utilizada é Module Learning With Errors (MLWE), na qual é empregado o anel polinomial $R_q = \mathbb{Z}_q[X]/(X^n+1)$ no lugar de $\mathbb{Z}_q$ como coordenadas de um vetor, a fim de reduzir o tamanho de chaves e o tempo computacional.

3.2. ML-KEM

O ML-KEM, inicialmente denominado CRYSTALS-Kyber, é um algoritmo baseado no problema MLWE e padronizado pelo NIST em 2024 por meio do FIPS 203 [NIST 2024b]. Ele é um mecanismo de encapsulamento de chaves (KEM), ou seja, tem como propósito estabelecer uma chave simétrica segura em um canal público, e possui três conjuntos de parâmetros distintos correspondentes aos níveis de segurança 2, 3 e 5 da escala do NIST. O algoritmo consiste em três funções principais: geração de chaves, encapsulamento e desencapsulamento. O algoritmo é descrito em linhas gerais a seguir:

Geração de chaves: inicia-se com a geração das sementes ρ e σ a partir de uma semente de 32 bytes concatenada com o parâmetro k, correspondente à dimensão dos vetores e matrizes. Em seguida, são gerados a matriz $\hat{\mathbf{A}}$, representação da matriz $\mathbf{A}$ transformada para o domínio da *Number Theoretic Transform* (NTT), o vetor secreto $\hat{\mathbf{s}}$ e o vetor de erro $\hat{\mathbf{e}}$ para calcular o vetor $\hat{\mathbf{t}} = \hat{\mathbf{A}} \cdot \hat{\mathbf{s}} + \hat{\mathbf{e}}$. Por fim, retorna-se a chave de encapsulamento ek, baseada na chave pública ek_{PKE} (formada pela concatenação de $\hat{\mathbf{t}}$ com ρ), e a chave de desencapsulamento dk, baseada na chave privada dk_{PKE} (vetor $\hat{\mathbf{s}}$) [NIST 2024b]. Na Seção 4.1, será apresentada a versão equivalente otimizada para ambientes restritos realizada por este trabalho.

Encapsulamento: o encapsulamento do ML-KEM é composto internamente por uma cifração, a qual executa a regeração da matriz $\hat{\mathbf{A}}$, e geração dos vetores $\hat{\mathbf{y}}$ e $\mathbf{e_1}$ além de um polinômio e_2. O algoritmo gera um valor aleatório interno, do qual deriva a chave simétrica compartilhada K, e o cifra para produzir o texto cifrado c [NIST 2024b].

Desencapsulamento: o desencapsulamento executa uma decifração interna que envolve a utilização da chave privada tal que, combinada com c, é possível recuperar a chave secreta K [NIST 2024b]. Devido à transformada de Fujisaki-Okamoto, é necessário verificar a consistência do texto cifrado ao cifrar novamente a mensagem m.

Caso corretamente implementado, o algoritmo produz uma chave compartilhada idêntica de 256 bits para o destinatário e o remetente, permitindo a comunicação utilizando algoritmos simétricos, como o AES [Dworkin et al. 2001].

4. Metodologia

A implementação do ML-KEM é feita na plataforma Java Card, uma versão modificada do Java para ambientes restritos de *smart cards*, que executa o programa por meio da Java Card Virtual Machine (JCVM). A execução é feita a partir de *applets*, que constituem as aplicações a serem executadas no ambiente do módulo de identidade. Comparada ao Java e a outras linguagens de propósito geral, há múltiplas limitações devido ao

ambiente-alvo restrito, incluindo a falta de suporte nativo para tipos como *long*, vetores multidimensionais e mais limitações para tamanhos de arquivos CAP (*Converted applet*).

Outra característica do Java Card é o suporte opcional a inteiros do tipo *int* e à coleta de lixo (*garbage collection*) automática. Embora existam diferenças entre fabricantes, cartões SIM e eSIMs geralmente utilizam arquiteturas de pelo menos 32 bits, como a ARM Cortex-M3. Por essa razão, assume-se, neste projeto, que o cartão ofereça suporte ao tipo *int*.

A alocação de memória é feita exclusivamente dentro do método `install` (função que instancia um *applet*), tanto para alocações na memória Flash quanto para alocações na RAM, onde os *buffers* descritos na Seção 4.1 são criados e depois reutilizados para outros propósitos. Essa prática evita quaisquer alocações dinâmicas e permite a técnica de reutilização de *buffers*.

Para codificar e testar a implementação, foi utilizado o simulador jCardSim que, por possuir a base em Java, traz algumas características específicas ao ambiente de teste como maiores espaços de memória. Além disso, o simulador dispensa a necessidade de compilação e manipulação de arquivos CAP e contorna a falta de funções-padrão de Java de apoio ao desenvolvedor (como `System.out.println`).

Em relação à implementação do algoritmo, foram utilizadas como referências o código-fonte da biblioteca criptográfica BouncyCastle [The Legion of the Bouncy Castle 2026], a implementação de Fisher (escrita em Java e baseada no código-referência do Kyber na linguagem de programação Go) [Fisher 2022], e, para as funções de eXtendable Output Function (XOF), como o SHAKE, foi adaptada uma implementação em Java Card realizada pela empresa ThothTrust [ThothTrust Private Limited 2022]. O código original se referia ao Keccak-F1600 de 256 bits de saída, e foi modificado para cobrir as funções principais de SHAKE: `reset`, `absorb`, `squeeze`.

A implementação desenvolvida por essa pesquisa utilizou vetores teste disponibilizados pelo NIST (*Known Answer Tests*, ou KATs) para verificar a corretude do algoritmo, e garantiu que a implementação produz a saída esperada dada uma entrada previamente estabelecida [NIST 2024b].

4.1. Otimizações

A plataforma Java Card possui restrições relacionadas ao ambiente de execução. Para viabilizar o algoritmo, com auxílio de certas técnicas de otimização mencionadas na Seção 2, foram realizadas as seguintes otimizações:

Reúso de *Buffers* Globais: consiste na criação de *buffers* globais dentro do *applet* de tamanho fixo, alocados na RAM e na Flash. Essa técnica permite um maior controle sobre o uso de memória, facilitando a coleta de lixo e a otimização no uso de memória por meio da reutilização de *buffers* em diferentes contextos. Conforme estabelecido pelo FIPS 203 [NIST 2024b], todo *buffer* que armazena dados sensíveis é zerado assim que essas informações não são mais necessárias. Essa medida previne o vazamento de valores intermediários para um atacante. Ademais, devido às restrições de memória do cartão e ao *buffer* compartilhado, instâncias paralelas não seriam possíveis. Caso seja necessária uma execução

paralela, haveria a necessidade de outras alocações de memória ou controles de concorrência.

Geração de vetores sob demanda: vetores de polinômios como $\hat{\mathbf{e}}$, $\hat{\mathbf{s}}$ e a matriz $\hat{\mathbf{A}}$ consomem memória com crescimento linear ou quadrático em relação ao parâmetro k. Porém, a geração desses vetores, conforme determinado pelo padrão, é feita polinômio por polinômio e as operações de multiplicação e adição também podem ser realizadas por etapas, utilizando um elemento por vez. Portanto, não é necessário armazenar os vetores por completo, somente os 256 coeficientes do tipo *short* (2 *bytes)* relativos ao polinômio que forma um elemento do vetor.

Simplificação de vetores: devido à falta de suporte para vetores multidimensionais (matrizes), vetores bidimensionais e tridimensionais foram substituídos por vetores unidimensionais. Dado que a geração de vetores sob demanda trata os elementos um a um, essa técnica é utilizada principalmente na simplificação da chave privada dk. Para acessar um índice como $dk[i][j]$, foram utilizados *offsets* de forma que o acesso a $dk[i][j]$ seja equivalente a $dk[(j * n) + i]$, onde n é o tamanho do polinômio que forma um elemento do vetor.

Redução da sobrecarga de classes e objetos: houve a necessidade de adaptar a orientação a objetos nativa de Java para reduzir as múltiplas interfaces e classes para uma única classe, a fim de evitar problemas relativos a limites de memória em conjunto com a falta de coleta de lixo (deixando objetos se acumularem na memória) [van der Laan et al. 2018].

A otimização do algoritmo de geração de chaves é ilustrada no pseudocódigo apresentado no Algoritmo 1, em que $\mathbb{B}$ é o conjunto de 256 inteiros de 8 bits sem sinal (bytes), $\mathbb{S}$ é o conjunto de 2^{16} inteiros de 16 bits sem sinal e S^k é o conjunto de vetores com k elementos do conjunto S.

A ideia principal do algoritmo é gerar polinômios quando necessário, por meio de funções como SamplePolyCBD e SampleNTT. Elas realizam operações de multiplicação e adição sobre o polinômio e armazenam os resultados no local desejado, em específico, na chave privada packedDK. Nota-se também a reutilização de *buffers*, como o bufNoise que armazena os polinômios tanto do vetor $\hat{\mathbf{e}}$ quanto do vetor $\hat{\mathbf{s}}$. Ademais, há somente vetores puros, sem qualquer abstração de objetos por trás, e acessados por meio de *offsets* que determinam o índice desejado para cada vetor. Como visto, cada técnica de otimização é fundamental para a execução do código em um ambiente restrito.

Por fim, as implementações do encapsulamento e do desencapsulamento, seguem o mesmo padrão lógico utilizado pela geração de chaves e, portanto, seus pseudocódigos foram suprimidos por questões de espaço.

5. Resultados e discussões

A estrutura de alocação de memória usada neste trabalho para o algoritmo ML-KEM pode ser visualizada na Tabela 1. Objetos que são utilizados para cálculos internos, como as funções de *hash* e XOF, além dos *buffers* para polinômios, são alocados na RAM. Já objetos que são de utilização recorrente, que é o caso da chave de desencapsulamento e da chave simétrica secreta acordada, são armazenados na Flash. É importante ressaltar que não foi necessário alocar um *buffer* específico para a chave pública, uma vez que ela está contida na chave privada, conforme especificado no padrão:

Algoritmo 1 Função de geração de chaves otimizada

Entrada: paramsK $\in \{2,3,4\}$, originalSeed $\in \mathbb{B}^{64}$, bufNoise $\in \mathbb{S}^{256}$, bufPolyTemp $\in \mathbb{S}^{256}$, bufMatrix $\in \mathbb{S}^{256}$
Saída: packedDK $\in \mathbb{B}^{768k+96}$ (Chave Privada contendo dk_{PKE}, ek_{PKE} e semente)

seedBuf $\leftarrow$ G(originalSeed, paramsSymBytes) ▷ seedBuf $= (\rho, \sigma)$
$N \leftarrow 0$
for $i \leftarrow 0$ **to** paramsK $- 1$ **do** ▷ **Etapa 1: Geração e Empacotamento de** $\hat{\mathbf{s}}$
  bufNoise $\leftarrow$ NTT(SamplePolyCBD$_{\eta_1}$(PRF$_{\eta_1}(\sigma, N)$))
  offset $\leftarrow i \times$ encodedPolySize ▷ encodedPolySize = 384
  packedDK[offset] $\leftarrow$ ByteEncode$_{12}$(bufNoise)
  $N \leftarrow N + 1$
end for
$N \leftarrow$ paramsK
for $i \leftarrow 0$ **to** paramsK $- 1$ **do** ▷ **Etapa 2: Geração da Matriz** $\hat{\mathbf{A}}$ **e Cálculo de** $\hat{\mathbf{t}}$
  bufPolyTemp $\leftarrow \mathbf{0}$ ▷ Atua como acumulador da soma para a linha i
  for $j \leftarrow 0$ **to** paramsK $- 1$ **do**
    bufMatrix $\leftarrow$ SampleNTT(seedBuf$||i||j$) ▷ Recupera $\hat{\mathbf{A}}[i][j]$
    offset $\leftarrow j \times$ encodedPolySize
    bufNoise $\leftarrow$ ByteDecode$_{12}$(packedDK[offset]) ▷ Recupera $\hat{\mathbf{s}}[j]$
    bufPolyTemp $\leftarrow$ bufPolyTemp $+$ bufMatrix $\circ$ bufNoise ▷ $\hat{\mathbf{A}}[i][j] \circ \hat{\mathbf{s}}[i]$
  end for
  bufNoise $\leftarrow$ NTT(SamplePolyCBD$_{\eta_1}$(PRF$_{\eta_1}(\sigma, N)$)) ▷ Gera $\hat{\mathbf{e}}[i]$ sobre o ruído
  $N \leftarrow N + 1$
  bufPolyTemp $\leftarrow$ bufPolyTemp $+$ bufNoise ▷ Adiciona $\hat{\mathbf{e}}[i]$ ao acumulador
  offset $\leftarrow$ (paramsK $+ i$) $\times$ encodedPolySize
  packedDK[offset] $\leftarrow$ ByteEncode$_{12}$(bufPolyTemp) ▷ Guarda $\hat{\mathbf{t}}[i]$
end for
offset $\leftarrow 2 \times$ paramsK $\times$ encodedPolySize ▷ **Etapa 3: Anexar Semente Final**
packedDK[offset] $\leftarrow \rho$

$dk = (dk_{PKE}||ek||H(ek)||z)$. Tanto a função SHAKE quanto a função SHA3 resultam de uma implementação derivada da empresa ThothTrust [ThothTrust Private Limited 2022].

Na geração de chaves do ML-KEM, faz-se uso de todos os objetos listados na Tabela 1, exceto o texto cifrado e a chave secreta. Portanto, para o maior nível de segurança, especificamente o ML-KEM-1024, foram alocados 2,68 kB na RAM e 3,20 kB na Flash, totalizando um consumo de memória de até 6,25 kB.

No encapsulamento, conforme o princípio de reutilização de *buffers*, aproveitam-se as alocações realizadas na geração de chaves. Ademais, há uma alocação adicional devido ao texto cifrado e chave secreta, resultando em um consumo de memória de 3,20 kB na Flash e até 4,25 kB de RAM para o ML-KEM-1024.

Conforme abordado na Seção 3.2, a Transformada de Fujisaki-Okamoto requer que o texto cifrado seja verificado por meio de uma nova cifração. Em consequência, há a necessidade de alocar memória suficiente para reter tanto o texto cifrado recebido quanto o recém-gerado. Ao todo, para essa etapa, são necessários 3,20 kB alocados na Flash e até

Tabela 1. Alocação de memória em bytes usada para o algoritmo ML-KEM

Estrutura / Componente	Tipo de Memória	Tamanho Alocado (B)
Buffers de polinômios	RAM	1.548
Buffer de amostragem por rejeição	RAM	672
Buffer de sementes	RAM	64
SHAKE-128 e SHAKE-256	RAM	407
Chave de Desencapsulamento (dk)*	Flash	1.632 / 2.400 / 3.168
Chave Secreta	Flash	32
Texto Cifrado*	RAM	768 / 1.088 / 1.568

* Varia conforme os níveis de segurança 1, 3 e 5 do ML-KEM.

5,82 kB alocados na RAM, dependendo do nível de segurança. Pode-se consultar todos os valores de alocação de memória na Tabela 2, que apresenta os valores de consumo de memória para cada uma das principais rotinas do algoritmo ML-KEM.

Conforme a Tabela 3, é possível, por meio dos métodos descritos na Seção 4.1, reduzir em média 76% da memória utilizada para a execução de todas as principais funções do algoritmo em comparação a uma implementação padrão do ML-KEM, na qual é alocada memória para cada objeto separadamente, sem qualquer otimização. Por essa tabela, é possível visualizar o impacto que as otimizações realizadas por este trabalho afetam o consumo de memória do algoritmo.

Tabela 2. Uso de memória em bytes do ML-KEM neste trabalho, discriminado entre RAM e Flash.

Nível	Operação	RAM (B)	Flash (B)	Total (B)
ML-KEM-512	Geração de Chaves	2.679	1.632	4.311
	Encapsulamento	3.447	1.664	5.111
	Desencapsulamento	4.215	1.664	5.879
ML-KEM-768	Geração de Chaves	2.679	2.400	5.079
	Encapsulamento	3.767	2.432	6.199
	Desencapsulamento	4.855	2.432	7.287
ML-KEM-1024	Geração de Chaves	2.679	3.168	5.847
	Encapsulamento	4.247	3.200	7.447
	Desencapsulamento	5.815	3.200	9.015

A partir da geração de vetores sob demanda, matrizes e vetores conseguem ser significativamente reduzidos. A matriz $\hat{\mathbf{A}}$, e vetores $\hat{\mathbf{s}}$ e $\hat{\mathbf{e}}$, na versão otimizada, consomem somente 1.536 B de espaço total, comparados aos 7.680 B que seriam utilizados se todos elementos fossem armazenados diretamente. Além disso, a partir da reutilização dos *buffers* globais, houve reduções de 100% para certos vetores devido à reutilização de uma memória alocada previamente para outro objeto, não havendo necessidade de um espaço adicional. Destaca-se ainda que não foi necessária nenhuma alocação dinâmica, garantindo um uso controlado da memória.

Para uma contextualização prática, utilizando como referência o cartão SMAOT500B 5G UICC SIM Card, de 64/128 kB de memória Flash e 6 kB de memória RAM de arquitetura CORTUS APS3cd de 32 bits, é possível realizar todas as operações

Tabela 3. Comparação de consumo de memória dentre as alocações de *buffer* feitas entre a implementação padrão e a otimizada para o conjunto de parâmetros ML-KEM-768.

Fase	Variáveis	Padr. (B)	Otim. (B)	Redução	Método
Geração	Matrizes: $\hat{\mathbf{A}}$	4.608	512	88,9%	Sob demanda
	Vetores de polinômios: $\mathbf{s}, \mathbf{e}$	3.072	1.024	66,7%	Sob demanda
	Vetores e chave pública: $\hat{\mathbf{t}}, ek$	2.720	0	100,0%	Reúso de *buffer*
	Semente e chave privada: d, dk	2.464	2.464	0,0%	-
	Instâncias de funções hash: SHAKE e SHA3	1.479	1.479	0,0%	-
	Subtotal Geração	**14.343**	**5.479**	**61,8%**	
Encapsulamento	Matrizes e vetores: $\hat{\mathbf{A}}, \mathbf{y}, \mathbf{e_1}, \mathbf{e_2}, u, \mu, v$	10.752	0	100,0%	Reúso de *buffer*
	Texto cifrado: c	1.088	1.088	0,0%	-
	Chave secreta: K	32	32	0,0%	-
	Subtotal Cifração	**11.872**	**1.120**	**90,6%**	
Desencapsulamento	Vetores e texto cifrado: $c, u, v, \mathbf{s}, w, m$	4.672	0	100,0%	Reúso de *buffer*
	Texto cifrado: c'	1.088	1.088	0,0%	-
	Subtotal Decifração	**5.760**	**1.088**	**81,1%**	
	Uso total	**32.519**	**7.687**	**76,0%**	

do ML-KEM. Caso seja desejado executar outros *applets* simultaneamente, é possível realocar certos vetores de menor uso para a Flash, como o texto cifrado. Tal estratégia contornaria o problema de espaço, apesar da perda de desempenho relacionado a operações de leitura e escrita mais lentas, e exigiria o cuidado com o limite de escritas da Flash (em torno de 2 milhões para esse cartão específico), protegendo a memória de desgaste físico.

A respeito dos custos computacionais, devido às otimizações feitas para reduzir a memória, ocorre um *tradeoff* sobre o desempenho do algoritmo. A Tabela 4 compara o tempo de execução entre esse trabalho e a implementação realizada pelo Fisher (KyberJCE) [Fisher 2022], na qual nota-se o aumento significativo do tempo de execução das principais rotinas do ML-KEM. Esse atraso decorre, principalmente, do recálculo de polinômios gerados sob demanda e da falta de suporte para o tipo *long*, que é usado extensivamente por funções de hash e XOF como SHAKE e SHA3. Como a geração de polinômios é baseada na geração de números pseudoaleatórios por meio da função SHAKE, percebe-se um aumento do tempo de execução de 726,25% para a geração de polinômios da matriz (de 24,679 μs para 203,912 μs). As medições foram realizadas em notebook Intel i5-13420H a partir do Java Microbenchmarking Harness (JMH). Apesar da discrepância nos tempos de execução em comparação ao KyberJCE, é importante considerar as diferenças dos ambientes- alvo de cada implementação – visto que o KyberJCE foi otimizado para a linguagem Java.

Tabela 4. Comparativo de tempo de execução entre KyberJCE e Este Trabalho

Nível	Operação	KyberJCE (ms/op)	Este Trabalho (ms/op)	Razão
ML-KEM-512	Geração de Chaves	0,17 ± 0,01	1,41 ± 0,15	8,67
	Encapsulamento	0,07 ± 0,01	1,53 ± 0,31	21,33
	Desencapsulamento	0,08 ± 0,01	1,85 ± 0,20	24,08
ML-KEM-768	Geração de Chaves	0,20 ± 0,01	2,26 ± 0,05	11,48
	Encapsulamento	0,10 ± 0,01	2,37 ± 0,49	23,00
	Desencapsulamento	0,12 ± 0,03	2,76 ± 0,11	23,59
ML-KEM-1024	Geração de Chaves	0,25 ± 0,03	3,60 ± 0,30	14,48
	Encapsulamento	0,17 ± 0,08	3,89 ± 0,16	22,45
	Desencapsulamento	0,16 ± 0,02	4,66 ± 0,42	28,94

6. Conclusão e Trabalhos Futuros

Concluímos, a partir dos resultados apresentados, a viabilidade espacial da implementação do algoritmo ML-KEM no contexto de *smart cards*, como observado na comparação com os requisitos de armazenamento de um cartão SMAOT500B 5G UICC SIM Card. Porém, para a execução apropriada, sem sobrecarregar a memória RAM ou Flash, seria interessante que *smart cards* possuíssem maior armazenamento. Ademais, o desempenho do algoritmo mostrou-se inferior quando comparado a outras implementações em Java, o que demanda novas otimizações em relação ao seu tempo de execução. Apesar dos desafios apresentados pela plataforma Java Card, como a falta de suporte a vetores multidimensionais e à coleta de lixo, a execução desse algoritmo foi possível ao utilizar as seguintes técnicas de otimização: reutilização de *buffers* globais, planificação de vetores, geração de vetores sob demanda e redução de classes.

Esta implementação representa um passo importante na viabilização da transição para algoritmos pós-quânticos que se fará obrigatória até 2035, especialmente considerando cartões SIM e eSIMs, que deverão utilizar o algoritmo para se autenticarem com a operadora de forma segura. Como possíveis avanços deste trabalho, propõe-se a implementação desse algoritmo utilizando apenas os tipos *byte* e *short*, permitindo uma maior compatibilidade com plataformas de *smart card* mais limitadas. Ademais, seria importante avaliar o desempenho do algoritmo quando combinado com um algoritmo tradicional em protocolos híbridos. Esta avaliação resultaria em uma melhor compreensão sobre seu comportamento dentro de cenários mais realistas durante o período de transição. Por fim, testes práticos utilizando cartões físicos (ou cartões virtuais executados dentro de eSIMs) estão sendo feitos para avaliar com maior precisão a viabilidade do algoritmo em cenários reais.

Referências

Albrecht, M. R., Hanser, C., Hoeller, A., Pöppelmann, T., Virdia, F., and Wallner, A. (2018). Implementing rlwe-based schemes using an rsa co-processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019:169–208.

Botros, L., Kannwischer, M. J., and Schwabe, P. (2019). Memory-efficient high-speed implementation of kyber on cortex-m4. In Buchmann, J., Nitaj, A., and Rachidi, T., editors, *Progress in Cryptology – AFRICACRYPT 2019*, pages 209–228, Cham. Springer International Publishing.

Christof Paar et al. (2024). *Understanding Cryptography*. Springer.

Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, Jr., J. (2001). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (FIPS) 197, National Institute of Standards and Technology, Gaithersburg, MD. `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=901427`.

FEITIAN Technologies (2026). Feitian completes post-quantum cryptography upgrade across core product lines, strengthening long-term digital security. `https://www.ftsafe.com/about/press_release/acgraphy_Upgrade_Across_Core_Product_Lines_Strengthening_Long_Term_Digital_Security`.

Fisher, S. K. (2022). Kyberjce. https://github.com/fisherstevenk.

Hou, R., Gao, Y., Liu, Y., Ming, J., and Zhou, Y. (2026). Meml-kem: A memory-efficient implementation of ml-kem for iot devices. In Liu, H., Ibrahim, S., and Rauber, T., editors, *Algorithms and Architectures for Parallel Processing*, pages 216–230, Singapore. Springer Nature Singapore.

Meneses, R., Teixeira, C., and Henriques, M. (2024). Compact memory implementations of the ml-dsa post-quantum digital signature algorithm. In *Anais Estendidos do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 233–243, Porto Alegre, RS, Brasil. SBC.

Micciancio, D. and Goldwasser, S. (2002). *Complexity of Lattice Problems: A Cryptographic Perspective*. Springer, New York, NY.

Micciancio, D. and Regev, O. (2009). *Lattice-based Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg.

Moody, D., Perlner, R., Regenscheid, A., Robinson, A., and Cooper, D. (2024). Transition to post-quantum cryptography standards. Technical report, National Institute of Standards and Technology.

NIST (2016). Request for comments on post-quantum cryptography requirements and evaluation criteria. https://csrc.nist.gov/projects/post-quantum-cryptography.

NIST (2024a). Guidelines on mobile device forensics. Technical report, National Institute of Standards and Technology, Gaithersburg, MD.

NIST (2024b). Module-lattice-based key-encapsulation mechanism standard. Technical report, National Institute of Standards and Technology. https://csrc.nist.gov/pubs/fips/203/final.

Philippe Gaborit et al. (2025). Hamming quasi-cyclic (hqc). Technical report, National Institute of Standards and Technology (NIST).

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

The Legion of the Bouncy Castle (2026). The legion of the bouncy castle cryptography apis. https://www.bouncycastle.org/.

ThothTrust Private Limited (2022). jckeccak. https://github.com/ThothTrustCom/jcKeccak.

van der Laan, E., Poll, E., Rijneveld, J., de Ruiter, J., Schwabe, P., and Verschuren, J. (2018). Is java card ready for hash-based signatures? In Inomata, A. and Yasuda, K., editors, *Advances in Information and Computer Security*, pages 127–142, Cham. Springer International Publishing.

Yuan, Y. et al. (2021). Memory-constrained implementation of lattice-based encryption scheme on standard java card platform. *IET Information Security*, 15:267–281.