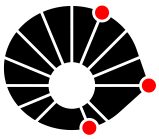

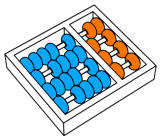

Implementação em Software do Algoritmo Pós-Quântico PERK

Gabriel C. Kinder *Julio López*

Relatório Técnico - IC-PFG-23-42
Projeto Final de Graduação
2024 - Fevereiro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Implementação em Software do Algoritmo Pós-Quântico PERK

Gabriel Costa Kinder* Julio López*

Resumo

Com os avanços recentes da computação quântica, se vê necessário a remodelação nos nossos sistemas atuais de criptografia, saindo de modelos de criptografia clássica e partindo para os modelos conhecidos como pós-quânticos, algoritmos criptográficos resistentes a ataques provenientes tanto de computadores tradicionais quanto de computadores quânticos. Porém estes algoritmos tem uma tendência de serem mais custosos em termos de recursos computacionais para serem implementados em software do que os atuais algoritmos criptográficos, portanto se cria uma necessidade ainda maior de implementações otimizadas para arquiteturas específicas. Neste projeto, além de uma explicação do algoritmo de assinatura pós-quântico PERK, é realizada uma implementação mais bem otimizada para a arquitetura ARMv8.5, e então é comparada com sua implementação de refêrencia, demonstrando seus ganhos de performance em ciclos de execução.

1 Introdução

Avanços recentes no desenvolvimento de um computador quântico e o conhecido algoritmo quântico de Shor, de tempo polinomial, para os problemas de fatoração inteira e cálculo de logaritmos discretos, são considerados uma ameaça ao futuro dos sistemas criptográficos de chave pública atuais, tais como o RSA e ECC (criptografia de curvas elípticas). Por esse motivo, uma nova área de pesquisa ganhou força na comunidade criptográfica, a chamada criptografia pós-quântica (post-quantum cryptography), que visa desenvolver novos algoritmos de chave pública baseados em problemas computacionais seguros contra computadores quânticos e convencionais, e que possa interoperar com protocolos existentes e redes.

Em 2017, o *National Institute of Standards and Technology* (NIST) [1] fez uma chamada pública para solicitar, avaliar e padronizar um ou mais algoritmos criptográficos de chave pública resistentes a computadores quânticos. De acordo com o NIST: "Pretende-se que o novo padrão de criptografia de chave pública especifique um ou mais esquemas criptográficos não secretos e divulgados publicamente para assinatura digital, encriptação de chave pública e estabelecimento de chaves, disponíveis em todo o mundo, e que sejam capazes de proteger informações confidenciais do governo no previsível futuro, inclusive após o advento dos computadores quânticos".

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

Na primeira rodada da chamada pública do NIST, foram recebidas 69 propostas; Para a segunda rodada [2] (janeiro de 2019), somente 26 propostas foram selecionadas, tendo as garantias de segurança como o critério primordial, enquanto o desempenho foi tratado como uma meta futura. Portanto, a análise da segunda rodada se centrará na avaliação do desempenho das propostas em uma ampla variedade de plataformas computacionais, incluindo grandes computadores, smartphones e também em dispositivos restritos de baixa potência. A segurança das propostas da segunda rodada é baseada em resultados teóricos provenientes das seguintes áreas da computação e matemática: reticulados, teoria de códigos, funções de resumo, curvas elípticas e isogenias, provas de conhecimento nulo, polinômios multivariados e teoria dos números.

Em 22 de Julho de 2020, foi anunciado pelo NIST a lista de 7 algoritmos aprovados na sua terceira rodada [3]. Para encriptação com chave pública e troca de chaves, os seguintes algoritmos foram selecionados: Classic-McEliece, CRYSTALS-KYBER, NTRU e Saber; para assinaturas digitais, a lista inclui CRYSTALS-Dilithium, FALCON e Rainbow. Adicionalmente, foram anunciados pelo NIST uma lista de 8 algoritmos alternativos (dos 26 candidatos aprovados na segunda rodada) que podem ser considerados no futuro para o processo de padronização.

Em Julho de 2022, foi anunciado pelo NIST 4 algoritmos para serem padronizados, além de 4 outros algoritmos que vão seguir para uma quarta rodada [4]. Os algoritmos para serem padronizados incluem 1 algoritmo de estabelecimento de chaves (CRYSTALS-KYBER) e 3 algoritmos de assinaturas digitais (CRYSTALS-Dilithium, FALCON e SPHINCS+). Os 4 algoritmos que continuarão em análise na quarta rodada são BIKE, Classic-McEliece, HQC e SIKE.

Em agosto de 2022 foi feita uma nova chamada pública [5], solicitando algoritmos pós-quânticos adicionais de assinatura digital, visto que todos os algoritmos restantes em análise tratavam-se de algoritmos de estabelecimento de chaves. Com isso foram recebidos 40 novos algoritmos para serem analizados, os quais são baseados em diferentes áreas da computação e matemática. Em específico vale destacar o alogritmo PERK, que utiliza o problema do kernel permutado [6], que é o alvo deste trabalho.

2 PERK

PERK é um esquema de assinatura digital projetado para resistir a ataques provenientes de computadores quânticos, assim como os provenientes de computadores clássicos. O esquema é construído em cima de um sistema de ZKPK (*zero-knowledge proof of knowledge*) baseado na conjectura da variante pós-quântica do Problema do Kernel Permutado (PKP) [7], em específico, de sua variante relaxada não homogênea, r-IPKP (*relaxed Inhomogenous Permuted Kernel Problem* [8]. Por sua vez, a prova de conhecimento zero é construída a partir do paradigma bem estabelecido de *MPC-in-the-head* e então é convertido para um esquema de assinatura usando a transformada de Fiat-Shamir [9].

Suas principais operações incluem a geração de números pseudo-aleatórios (PRG), utilizando SHAKE-128 ou SHAKE-256 dependendo dos parâmetros utilizados, funções de hashing resistentes a colisão, utilizando SHA3-256, SHA3-384 ou SHA3-512, operações de

permutação e sorting utilizando-se do software djbsort [10], operações em árvores de Merkle e operações aritméticas em vetores e matrizes. O algoritmo possuí 12 conjuntos de parâmetros disponíveis, alterando uso de memória, performance de diferentes partes do sistema e nível de segurança.

A sua geração de chaves depende principalmente do PRG e de algumas operações em matriz. Sua função de assinatura utiliza a chave pública e a chave privada e pode ser dividida em 5 partes, a geração dos *commitments*, o computo do primeiro desafio, a primeira resposta, o computo do segundo desafio e a segunda resposta, parte dos valores calculados são adicionados à assinatura, que vai permitir posteriormente a verificação de que quem realizou a assinatura tinha conhecimento da chave privada. A verificação da assinatura utiliza a chave pública para recalcular alguns valores gerados durante a etapa de assinatura, que só poderiam ter sido corretamente gerados utilizando-se da chave privada, de acordo com os problemas citados anteriormente, portanto verificando a autenticidade do par mensagem-assinatura.

2.1 Parâmetros

O PERK possuí um total de 7 parâmetros variáveis que alteram diferentes aspectos do alogritmo.

- λ: O principal parâmetro para escalar o nível de segurança do algoritmo. Ele altera o tamanho da *seed* que será utilizada para gerar as chaves;
- q: Valor a ser utilizado como módulo dos números inteiros utilizados no algoritmo. Em sua especificação, é sempre mantido em 1021;
- n: Define o tamanho dos vetores utilizados no problema do r-IPKP;
- m: Define o tamanho da matriz utilizado no problema do r-IPKP;
- t: Define o número de dimensões existentes no problema do r-IPKP;
- N: É o parâmetro que rege entre as variantes *short* ou *fast*, definindo o número de participantes do MPC;
- τ: Define o número de repetições do protocolo MPC.

Em sua especificação, são fornecidos 12 permutações de configurações recomendadas para estes parâmetros, como pode ser visto na Tabela 1.

2.2 Geração de Chaves

A geração de chaves do PERK se resume à amostragem de três diferentes variáveis a partir de sementes aleatoriamente geradas, duas para a chave pública e uma para a chave secreta, e ao computo de uma última variável. Para a chave secreta será amostrado pi, um vetor de tamanho n, ele é amostrado à partir da função PRG, utilizando da semente da chave secreta, além disso o valor gerado passa por uma função de permutação que também garante a não repetição de valores consecutivos.

Permutações	λ	q	n	m	t	N	τ
PERK-I-fast3	128	1021	79	35	3	32	30
PERK-I-fast5	128	1021	83	36	5	32	28
PERK-I-short3	128	1021	79	35	3	256	20
PERK-I-short5	128	1021	83	36	5	256	18
PERK-III-fast3	192	1021	112	54	3	32	46
PERK-III-fast5	192	1021	116	55	5	32	43
PERK-III-short3	192	1021	112	54	3	256	31
PERK-III-short5	192	1021	116	55	5	256	28
PERK-V-fast3	256	1021	146	75	3	32	61
PERK-V-fast5	256	1021	150	76	5	32	57
PERK-V-short3	256	1021	146	75	3	256	41
PERK-V-short5	256	1021	150	76	5	256	37

Tabela 1: Permutações de parâmetros recomendadas na especificação do PERK [6]

Inputs:
-

Ouputs:
- Secret key sk
- Public key pk

1. Sample $\mathsf{sk\_seed} \xleftarrow{\$} \{0,1\}^{\lambda}$ and $\mathsf{pk\_seed} \xleftarrow{\$} \{0,1\}^{\lambda}$
2. Sample $\pi \longleftarrow \mathsf{PRG}(\mathsf{sk\_seed})$ from $\mathcal{S}_n$
3. Sample $(\mathbf{H}, (\mathbf{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
3. For $j \in [1,t]$,
   - $\diamond$ Compute $\mathbf{y}_j = \mathbf{H}\pi[\mathbf{x}_j]$
4. Output $(\mathsf{sk}, \mathsf{pk}) = (\mathsf{sk\_seed}, (\mathsf{pk\_seed}, (\mathbf{y}_j)_{j\in[1,t]}))$

Figura 1: Algoritmo completo de geração de chaves [6]

Para a chave pública, é realizado a amostragem de h e x, h é uma matriz de tamanho $m \times n$ amostrado à partir da função PRG, utilizando-se da semente da chave pública, mantendo apenas os 10 bits menos significativos e garantindo serem menores do que q. Já x é uma matriz $t \times n$ onde seus valores são gerados de forma identica à h, porém ainda é necessário garantir, devido ao problema IPKP, que essa matriz seja linearmente independente. Então, utilizando-se dos três parâmetros amostrados, podemos computar y, que faz parte da chave pública, uma matriz de tamanho t por m, tal que $y_i = h(pi[x_i]) \quad \forall i \in [0,t]$.

Desta forma terminamos com o que será utilizado como a chave pública e a chave secreta do algoritmo. Para a chave secreta, guarda-se somente o valor de sua semente gerada aleatoriamente, para poder re-gerar os valores necessários futuramente, e para a chave pública, guarda-se o valor de sua semente aleatória e também o computo de y_i. O algoritmo completo original pode ser visto na Figura 1.

2.3 Assinatura

A assinatura do PERK contém cinco etapas, a geração das *instâncias*, a geração de dois desafios e de duas respostas. Utilizando-se das variáveis pi proveniente da chave secreta e h da chave pública são geradas τ instâncias, cada uma contendo uma árvore Merkle θ [11], N pi^*'s próprios, N vetores v's de dimensão n e N+1 vetores *commitment*, de dimensão equivalente à $\lambda/4$.

O primeiro desafio utiliza o *sal* (um número aleatório) da mensagem, a mensagem, a chave pública e todos os *commitments* das instâncias em um hash, com resultado final denominado h_1, um vetor de dimensão $\lambda/4$, que fará parte da assinatura. E utilizando-se do h_1 serão amostrados, através do PRG, τ desafios, cada um contendo t κ's, valores garantidos de serem menores do que q e diferentes de zero.

A primeira resposta realizará o computo de N+1 vetores s's, de dimensão n, para cada uma das τ instâncias, tal que $s_0 = \sum_i \kappa_i \cdot x_i \quad \forall i \in [0, t]$ e $s_i = pi_i^* \cdot s_{i-1} + v_i \quad \forall i \in [1, N+1]$.

O segundo desafio utiliza o *sal* da mensagem, a mensagem, a chave pública, todos os *commitments* das instâncias, o h_1 e todos os s's em um hash, com resultado final denominado h_2, um vetor de dimensão $\lambda/4$, que fará parte da assinatura. E utilizando-se do h_2 serão amostrados, através do PRG, τ α's, e reduzindo-os, através de uma máscara, para mante-los menor que N.

A segunda resposta computa uma *response (rsp)* para cada α, tal que:
$rsp_i = (z_1, z_2, commitment_{1,\alpha_i}) \quad \forall i \in [0, \tau]$ onde:

- $z_1 = s_{\alpha i}$
- $z_2 = (pi_i^* || \theta_i)$ se $\alpha_i \neq 1$
- $z_2 = \theta_i$ se $\alpha_i = 1$

Ao final de todas as etapas obtemos a assinatura σ tal que:
$\sigma = (salt, h_1, h_2, rsp_i) \quad \forall i \in [0, \tau]$. O algoritmo completo original pode ser visto na Figura 2.

Inputs:	**Outputs:**
- Secret key sk	- Signature σ
- Public key pk	
- Message m	

Step 1: Commitment

1. Sample $\pi \longleftarrow \mathsf{PRG}(\mathsf{sk\_seed})$ from $\mathcal{S}_n$
2. Sample $(\mathbf{H}, (\mathbf{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
3. Sample salt and master seed $(\mathsf{salt}, \mathsf{mseed}) \xleftarrow{\$} \{0,1\}^{2\lambda} \times \{0,1\}^{\lambda}$
4. Sample seeds $(\theta^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \mathsf{mseed})$ from $(\{0,1\}^{\lambda})^{\tau}$
5. For each iteration $e \in [1,\tau]$,
   - ◇ Compute $(\theta_i^{(e)})_{i\in[1,N]} \longleftarrow \mathsf{TreePRG}(\mathsf{salt}, \theta^{(e)})$
   - ◇ For each party $i \in \{N,\dots,1\}$,
     - If $i \neq 1$, sample $(\pi_i^{(e)}, \mathbf{v}_i^{(e)}) \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_i^{(e)})$ from $\mathcal{S}_n \times \mathbb{F}_q^n$
     - If $i = 1$, sample $\mathbf{v}_1^{(e)} \longleftarrow \mathsf{PRG}(\mathsf{salt}, \theta_1^{(e)})$ from $\mathbb{F}_q^n$
     - If $i \neq 1$, compute $\mathsf{cmt}_{1,i}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, i, \theta_i^{(e)})$
     - If $i = 1$, compute $\pi_1^{(e)} = (\pi_2^{(e)})^{-1} \circ \cdots \circ (\pi_N^{(e)})^{-1} \circ \pi$ and $\mathsf{cmt}_{1,1}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$
   - ◇ Compute $\mathbf{v}^{(e)} = \mathbf{v}_N^{(e)} + \sum_{i\in[1,N-1]} \pi_N^{(e)} \circ \cdots \circ \pi_{i+1}^{(e)}[\mathbf{v}_i^{(e)}]$ and $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \mathbf{Hv}^{(e)})$

Step 2: First Challenge

6. Compute $h_1 = \mathsf{H}_1(\mathsf{salt}, m, \mathsf{pk}, (\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{e\in[1,\tau],i\in[1,N]})$
7. Sample $(\kappa_j^{(e)})_{e\in[1,\tau],j\in[1,t]} \longleftarrow \mathsf{PRG}(h_1)$ from $(\mathbb{F}_q^t)^{\tau}$

Step 3: First Response

8. For each iteration $e \in [1,\tau]$,
   - ◇ Compute $\mathbf{s}_0^{(e)} = \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \mathbf{x}_j$
   - ◇ For each party $i \in [1,N]$,
     - Compute $\mathbf{s}_i^{(e)} = \pi_i^{(e)}[\mathbf{s}_{i-1}^{(e)}] + \mathbf{v}_i^{(e)}$

Step 4: Second Challenge

9. Compute $h_2 = \mathsf{H}_2(\mathsf{salt}, m, \mathsf{pk}, h_1, (\mathbf{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]})$
10. Sample $(\alpha^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1,N])^{\tau}$

Step 5: Second Response

11. For each iteration $e \in [1,\tau]$,
    - ◇ Compute $\mathbf{z}_1^{(e)} = \mathbf{s}_{\alpha}^{(e)}$
    - ◇ If $\alpha^{(e)} \neq 1$, $z_2^{(e)} = (\pi_1^{(e)} \,||\, (\theta_i^{(e)})_{i\in[1,N]\setminus\alpha^{(e)}})$
    - ◇ If $\alpha^{(e)} = 1$, $z_2^{(e)} = (\theta_i^{(e)})_{i\in[1,N]\setminus\alpha^{(e)}}$
    - ◇ Compute $\mathsf{rsp}^{(e)} = (\mathbf{z}_1^{(e)}, z_2^{(e)}, \mathsf{cmt}_{1,\alpha^{(e)}}^{(e)})$
12. Compute $\sigma = (\mathsf{salt}, h_1, h_2, (\mathsf{rsp}^{(e)})_{e\in[1,\tau]})$

Figura 2: Algoritmo completo de Assinatura [6]

Inputs:	**Outputs:**
- Public key pk - Signature σ - Message m	- Accept or Deny Signature

Step 1: Parse signature

1. Sample $(\mathbf{H}, (\mathbf{x}_j)_{j\in[1,t]}) \longleftarrow \mathsf{PRG}(\mathsf{pk\_seed})$ from $\mathbb{F}_q^{m\times n} \times (\mathbb{F}_q^n)^t$
2. Parse signature as $\sigma = (\mathsf{salt}, h_1, h_2, (\mathbf{z}_1^{(e)}, z_2^{(e)}, \mathsf{cmt}_{1,\alpha^{(e)}}^{(e)})_{e\in[1,\tau]})$
3. Recompute $(\kappa_j^{(e)})_{e\in[1,\tau],j\in[1,t]} \longleftarrow \mathsf{PRG}(h_1)$ from $(\mathbb{F}_q^t)^\tau$
4. Recompute $(\alpha^{(e)})_{e\in[1,\tau]} \longleftarrow \mathsf{PRG}(h_2)$ from $([1,N])^\tau$

Step 2: Verification

5. For each iteration $e \in [1,\tau]$,
   - ◇ Compute $\mathbf{s}_0^{(e)} = \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \mathbf{x}_j$ and $\mathbf{s}_\alpha^{(e)} = \mathbf{z}_1^{(e)}$
   - ◇ Compute $(\pi_i^{(e)}, \mathbf{v}_i^{(e)})_{i\in[1,N]\setminus\alpha}$ from $\mathbf{z}_2^{(e)}$
   - ◇ For each party $i \in [1,N] \setminus \alpha^{(e)}$,
     - If $i \neq 1$, compute $\mathsf{cmt}_{1,i}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, i, \theta_i^{(e)})$
     - If $i = 1$, compute $\mathsf{cmt}_{1,1}^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, 1, \pi_1^{(e)}, \theta_1^{(e)})$
   - ◇ For each party $i \in [1,N] \setminus \alpha^{(e)}$,
     - Compute $\mathbf{s}_i^{(e)} = \pi_i^{(e)}[\mathbf{s}_{i-1}^{(e)}] + \mathbf{v}_i^{(e)}$
   - ◇ Compute $\mathsf{cmt}_1^{(e)} = \mathsf{H}_0(\mathsf{salt}, e, \mathbf{Hs}_N^{(e)} - \sum_{j\in[1,t]} \kappa_j^{(e)} \cdot \mathbf{y}_j)$
6. Compute $\bar{h}_1 = \mathsf{H}_1(\mathsf{salt}, m, \mathsf{pk}, (\mathsf{cmt}_1^{(e)}, \mathsf{cmt}_{1,i}^{(e)})_{e\in[1,\tau],i\in[1,N]})$.
7. Compute $\bar{h}_2 = \mathsf{H}_2(\mathsf{salt}, m, \mathsf{pk}, h_1, (\mathbf{s}_i^{(e)})_{e\in[1,\tau],i\in[1,N]})$.
8. Output accept if and only if $\bar{h}_1 = h_1$ and $\bar{h}_2 = h_2$.

Figura 3: Algoritmo completo de Verificação [6]

2.4 Verificação

A etapa de verificação do PERK se resume ao processamento da assinatura recebida e então à sua verificação. Utilizando-se dos valores de h_1 e h_2 presentes na assinatura, podemos re-amostrar os valores dos κ's e dos α's, obtendo os mesmos desafios gerados na parte de assinatura. Utilizando os valor do *salt* e dos θ's presentes na resposta, podemos recomputar os v's e pi^*'s Utilizando destes valores, junto com o que temos através da chave pública, podemos também recomputar os valores dos s's. Finalmente, tendo todos estes valores, podemos novamente gerar os valores ds *commitments* utilizando-se do mesmo hash utilizado durante a assinatura.

Ao obter todos estes valores, temos o necessário para realizar o hash que obteve o valor de h_1 e então de h_2, caso estes valores sejam idênticos aos presentes na assinatura, concluímos que a assinatura é verídica e que quem realizou ela está em posse da chave secreta. O algoritmo completo de verificação pode ser visto na Figura 3.

2.5 Implementação

O algoritmo foi implementado em C e está disponível em [12], incluindo um script para sua compilação e de seu benchmark. O benchmark foi produzido para funcionar em notebooks Apple que utilizam o chip Apple M1.

2.6 Segurança

O PERK é comprovadamente EUF-CMA em um modelo de oráculo aleatório (ROM) a partir da suposição da dificuldade do problema r-IPKP e com uma quantia de repetições τ suficientes para tornar as chances de uma assinatura forjada serem negligenciáveis.

3 Arquitetura ARM

A arquitetura de processadores ARM (*Advanced RISC Machines*) é conhecida devido ao seu custo, consumo de energia e geração de calor reduzidos, tornando-os muito comuns em celulares, sistemas embarcados e outras plataformas reduzidas que dependem do uso de bateria. Porém, especialmente após o lançamento de sua arquitetura ARMv8-A, que trouxe suporte para uma série de instruções especializadas de alto desempenho [13], sua popularidade em sistemas de maior necessidade de poder de processamento cresceu.

Em específico, o conjunto de instruções NEON, instruções únicas de múltiplos dados (*SIMD*), conseguem acelerar o processamento de múltiplos dados que necessitem de operações aritméticas em comum [14], como ocorre muitas vezes durante a execução do PERK em suas aritméticas vetoriais e matriciais.

4 Otimizações

Para obter melhores resultados de performance podemos otimizar a implementação utilizando instruções voltadas para a arquitetura ARM, em específico aquelas compatíveis com o processador Apple M1.

Na implementação realizada neste projeto foi utilizado uma versão dos algoritmos SHA-3 e SHAKE, presentes em [15], compiladas especificamente para processadores ARMv8, tornando o código mais otimizado para esta plataforma em específico, como será demonstrado nas tabelas seguintes. Além disso, baseado na otimização da implementação otimizada para AVX2 fornecida, foi otimizado a função de MinMax do DJBSort em [17] utilizando código de máquina ARMv8.5, utilizando-se de selects condicionais ao invés de moves condicionais e compensando a falta de um compare lógico como o presente na arquitetura AVX2 utilizando-se de registradores de 64 bits para os números de 32 bits.

Ainda existem muitas outras formas de otimizar o código para esta plataforma, tirando maior proveito de seus registradores maiores e instruções específicas para realizar operações complexas, que poderia deixar este algoritmo ainda mais rápido.

5 Performance

5.1 Tempo de Execução

Para verificar a eficiência da implementação, foi feita uma comparação de ciclos de execução dos algoritmos de geração de chaves, assinatura e verificação. Utilizando a implementação de referência presente em [6] e a implementação realizada para este projeto presente em [12], e utilizando do mesmo sistema para realizar o benchmark e em uma mesma máquina para garantir que os resultados sejam comparáveis, obtemos os resultados presentes nas tabelas 2 e 3. Os resultados marcados como *NA* não foram executados devidos à falhas de segmentação na sua execução.

Permutações	Geração de Chaves	Assinatura	Verificação
PERK-I-fast3	63k	15.9M	7.2M
PERK-I-fast5	79k	15.6M	6.9M
PERK-I-short3	63k	84.1M	38.2M
PERK-I-short5	79k	80.1M	35.7M
PERK-III-fast3	138k	37.8M	17.3M
PERK-III-fast5	158k	36.6M	16.6M
PERK-III-short3	138k	203M	93.2M
PERK-III-short5	158k	191M	86.5M
PERK-V-fast3	227k	77.1M	37.1M
PERK-V-fast5	254k	74.7M	35.1M
PERK-V-short3	226k	*NA*	*NA*
PERK-V-short5	254k	*NA*	*NA*

Tabela 2: Resultados da performance em ciclos de execução da implementação de referência utilizado em [6], executados em um MacBook Air com o chip Apple M1 de 3.2GHz

Podemos verificar pela Tabela 4 que a geração de chaves obteve um ganho ao redor de 14%, a assinatura ao redor de 23% e a verificação ao redor de 17%.

5.2 Uso de Memória

O uso de memória do algoritmo pode ser observador à partir da Tabela 5.

5.3 Comparações

Podemos ver na Figura 4 uma comparação do custo de memória para a assinatura (eixo vertical) e para a chave pública (eixo horizontal) entre outros algoritmos de assinatura pós-quânticos participando no mesmo concurso do NIST e o algoritmo RSA, e na Figura 5 uma comparação dos ciclos de execução do processo de assinatura e de verificação destes mesmos algoritmos. Os valores utilizados nessas figuras foram obtidas de [16], utilizando métricas de códigos otimizados para AVX2 em todos os algoritmos.

Permutações	Geração de Chaves	Assinatura	Verificação
PERK-I-fast3	54k	12.1M	5.9M
PERK-I-fast5	65k	11.9M	5.7M
PERK-I-short3	54k	64.8M	31.6M
PERK-I-short5	65k	61.3M	29.4M
PERK-III-fast3	120k	28.9M	14.3M
PERK-III-fast5	136k	28.0M	13.7M
PERK-III-short3	120k	156M	77.3M
PERK-III-short5	142k	147M	71.5M
PERK-V-fast3	198k	59.0M	30.3M
PERK-V-fast5	220k	57.0M	29.1M
PERK-V-short3	199k	319M	164M
PERK-V-short5	221k	296M	151M

Tabela 3: Resultados da performance em ciclos de execução da implementação em [12], executados em um MacBook Air com o chip Apple M1 de 3.2GHz e a flag *-O3* para compilação

Permutações	Geração de Chaves	Assinatura	Verificação
PERK-I-fast3	14.3%	23.9%	18.1%
PERK-I-fast5	17.7%	23.7%	17.4%
PERK-I-short3	14.3%	22.9%	17.3%
PERK-I-short5	17.7%	23.5%	17.6%
PERK-III-fast3	13.0%	23.5%	17.3%
PERK-III-fast5	13.9%	23.5%	17.5%
PERK-III-short3	13.0%	23.2%	17.1%
PERK-III-short5	13.9%	23.0%	17.3%
PERK-V-fast3	12.8%	23.5%	18.3%
PERK-V-fast5	13.4%	23.7%	17.1%
PERK-V-short3	11.9%	*NA*	*NA*
PERK-V-short5	13.0%	*NA*	*NA*

Tabela 4: Ganhos em porcentagem entre o gasto de ciclos de execução da implementação de refêrencia e da implementação deste projeto

Como análise importante, é interessante ver que, os algoritmos que tem um melhor desempenho em todos os aspectos de memória comparado ao PERK (MIRA e SQIsign), não ganham do mesmo em performance, e aqueles que ganham em performance possuem algum ou ambos os aspectos de uso de memória piores que o PERK.

Outra vantagem do PERK é que seu principal custo de performance são chamadas para primitivas criptográficas como o SHAKE ou SHA-3, portanto qualquer otimização para estes algoritmos básicos ou ainda aceleração de hardware para os mesmos causará um grande impacto na performance do algoritmo. Outra vantagem é que, mesmo não obtendo

Permutações	Tamanho da Chave Pública	Tamanho da Chave Secreta	Tamanho da Assinatura
PERK-I-fast3	0.15kB	16B	8.35kB
PERK-I-fast5	0.24kB	16B	8.03kB
PERK-I-short3	0.15kB	16B	6.56kB
PERK-I-short5	0.24kB	16B	6.06kB
PERK-III-fast3	0.23kB	24B	18.8kB
PERK-III-fast5	0.37kB	24B	18.0kB
PERK-III-short3	0.23kB	24B	15.0kB
PERK-III-short5	0.37kB	24B	13.8kB
PERK-V-fast3	0.31kB	32B	33.3kB
PERK-V-fast5	0.51kB	32B	31.7kB
PERK-V-short3	0.31kB	32B	26.4kB
PERK-V-short5	0.51kB	32B	24.2kB

Tabela 5: Utilização de memória no algoritmo perk [6]

a mesma performance que algoritmos baseados em reticulados, ele traz uma outra opção para sua base de segurança, caso seja encontrado alguma vulnerabilidade nos anteriores.

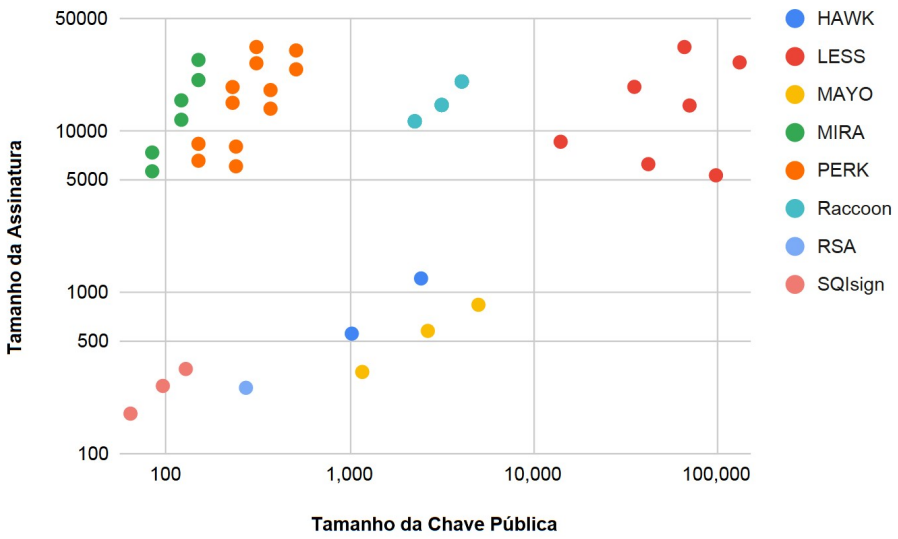

Figura 4: Comparação do custo de memória para assinatura e chave pública entre alguns algoritmos de assinatura pós-quânticos e o algoritmo RSA [16]

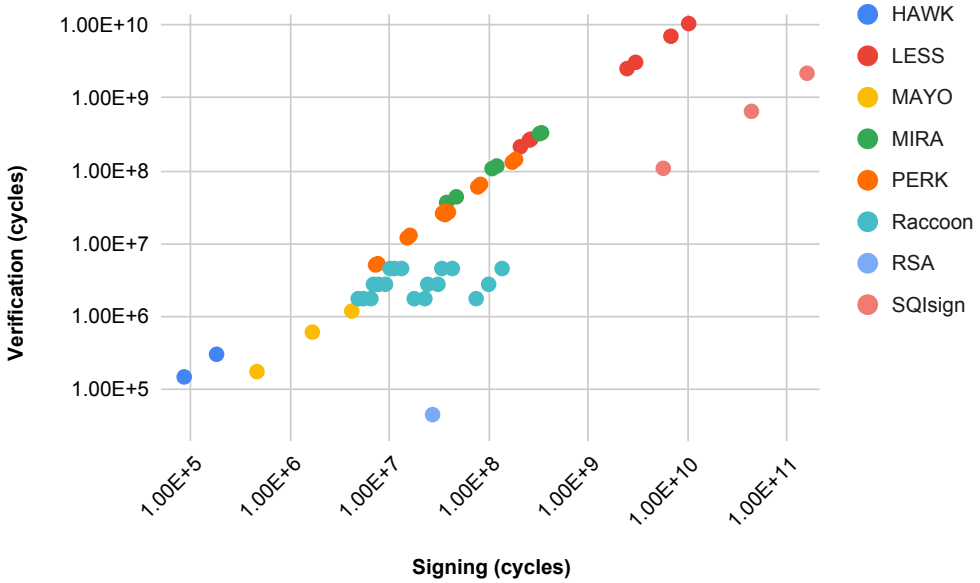

Figura 5: Comparação dos ciclos de verificação e de assinatura entre alguns algoritmos de assinatura pós-quânticos e o algoritmo RSA [16]

6 Conclusão

Este projetou se aprofundou nos conceitos do algoritmo PERK e na arquitetura ARM, incluindo uma implementação de tal algoritmo para tal arquitetura, utilizando-se apenas de uma implementação especializada de uma primitiva criptográfica, e já demonstrando resultados consideráveis de melhorias de performace quando comparado à referência.

Podemos imaginar que otimizar o algoritmo completo para tal arquitetura poderia proporcionar resultados ainda melhores em tempos de execução, desta forma tornando-o cada vez mais comparável a algoritmos de assinatura clássicos e mantendo a desvantagem principal apenas no tamanho da chave pública e da assinatura. Desta forma conseguimos demonstrar a eficiência de implementação em software de algoritmos para arquiteturas específicas, de forma a obter maior uso de suas características e instruções próprias.

Referências

[1] NIST, *Post Quantum Cryptography Standardization*, `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization` (Janeiro 2017)

[2] NIST, *NISTIR 82240, Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*, `https://csrc.nist.gov/publications/detail/nistir/8240/final` (Janeiro 2019)

[3] NIST, *NISTIR 8309, Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*, `https://www.nist.gov/publications/status-report-second-round-nist-post-\quantum-cryptography-standardization-process` (Julho 2020)

[4] NIST, *NISTIR 8413, Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*, `https://csrc.nist.gov/publications/detail/nistir/8413/final` (Julho 2022)

[5] NIST, *Post-Quantum Cryptography: Digital Signature Schemes, https://csrc.nist.gov/projects/pqc-dig-sig/standardization/call-for-proposals* (Agosto 2022)

[6] A. Najwa, et al., *PERK, https://pqc-perk.org/* (Maio 2023)

[7] A. Shamir, *An efficient identification scheme based on permuted kernels, CRYPTO'89, volume 435 do LNCS, páginas 606–609, Springer* (Agosto 1990)

[8] L. Bidoux, P. Gaborit, *Compact post-quantum signatures from proofs of knowledge leveraging structure for the PKP, SD and RSD problems, In Codes, Cryptology and Information Security (C2SI), pages 10–42. Springer* (2023)

[9] A. Fiat, A. Shamir, *How to prove yourself: Practical solutions to identification and signature problems, CRYPTO'86, volume 263 do LNCS, páginas 186–194, Springer* (Agosto 1987)

[10] D. J. Bernstein, *djbsort,* `https://sorting.cr.yp.to/` (2019)

[11] J. Katz, V. Kolesnikov, X. Wang, *Improved non-interactive zero knowledge with applications to post-quantum signatures, ACM CCS 2018, páginas 525–537, ACM Press* (Outubro 2018)

[12] G. C. Kinder, *PERK Implementation,* `https://github.com/Kinder-Eggs/PERK-Thesis` (Janeiro 2024)

[13] ARM, *ARM Discloses Technical Details Of The Next Version Of The ARM Architecture,* `https://web.archive.org/web/20190101024118/https://www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-next-version-of-the-arm-\architecture.php` (Outubro 2011)

[14] ARM, *Technologies: Arm Neon,* `https://www.arm.com/technologies/neon`

[15] XKCP, *eXtended Keccak Code Package,* `https://github.com/XKCP/XKCP` (Novembro 2023)

[16] PQShield, *Post-Quantum signatures zoo* `https://pqshield.github.io/nist-sigs-zoo/` (2023).

[17] G. C. Kinder, *DJBSort MinMax,* `https://github.com/Kinder-Eggs/PERK-Thesis/blob/main/Implementation/lib/djbsort/int32_minmax.inc` (Janeiro 2024)