Avaliação de requisitos de processamento e memória do algoritmo de criptografia pós-quântica SABER em plataforma ARM Cortex-M0+

George Gigilas Junior[1], Marco A. A. Henriques[1]

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

georgejuniorg@yahoo.com.br, maah@unicamp.br

Abstract. *SABER is a CCA-secure lattice based post-quantum key encapsulation scheme. This scheme was among the finalists in the third round of the NIST post-quantum cryptography standardization process. In this work, we adapt SABER and evaluate its execution on ARM Cortex-M0+ microcontroller. We also applied optimizations suggested in the literature on SABER, on LightSABER, a faster but less secure version, and on FireSABER, a slower but more secure version. These optimizations reduce memory usage in exchange for a higher CPU cycle count, making these algorithms viable for resource-constrained environments.*

Resumo. *O SABER é um esquema de encapsulamento de chaves CCA-seguro pós-quântico baseado em reticulados e finalista na terceira rodada padronização de algoritmos de criptografia pós-quântica do NIST. Neste trabalho, avaliamos sua execução em um microcontrolador ARM Cortex-M0+ e aplicamos otimizações sugeridas na literatura, adaptando-as para execução no SABER, no LightSABER, versão mais rápida, mas com menor segurança, e no FireSABER, versão mais lenta e mais segura. Essas otimizações diminuem o uso de memória, em troca de um número maior de ciclos de CPU, tornando a execução dos algoritmos viável em ambientes restritos.*

1. Introdução

O desenvolvimento dos computadores quânticos promete trazer um potencial computacional gigantesco, capaz de solucionar problemas mais complexos e mais rapidamente. Apesar de abrir portas para pesquisas importantes para a humanidade, esse potencial também causará um grande impacto nos esquemas de criptografia assimétrica utilizados atualmente. Tais esquemas se baseiam nos problemas de logaritmo discreto e de fatoração de números inteiros grandes, que são problemas que até hoje não foram resolvidos em tempo hábil em computadores tradicionais. Porém, já existem algoritmos para computadores quânticos que os resolvem em tempo polinomial - o algoritmo de Shor [1] e o algoritmo de Proost e Zalka [2] - o que põe em risco a segurança de inúmeras aplicações na Internet.

Diante desse cenário, o órgão americano National Institute of Standards and Technology (NIST) deu início, em 2016, a um processo de padronização de algoritmos pós-quânticos de criptografia assimétrica (troca de chaves) e de assinatura digital [3]. Tal processo consiste em três (ou mais) rodadas nas quais os algoritmos propostos são avaliados e selecionados para as próximas etapas. Em 2021, o NIST anunciou quatro algoritmos de troca de chaves como finalistas da terceira rodada de avaliações: Classic McEliece [4], CRYSTALS-Kyber [5], NTRU [6] e SABER [7]. Os três últimos

algoritmos citados são baseados em problemas sobre reticulados, o que tornava provável que pelo menos um algoritmo baseado em reticulados fosse escolhido.

Entretanto, apesar de se mostrarem promissores, esses algoritmos demandam alto poder computacional, seja por memória ou por processamento, como pode ser visto nos resultados obtidos pelo PQM4, *framework* responsável por bibliotecas, avaliação e testes com algoritmos de criptografia pós-quântica [8]. Isso acontece porque as chaves desses esquemas são consideravelmente maiores do que as chaves dos esquemas tradicionais, o que torna as operações mais custosas. Embora essa característica não seja, em geral, problemática para os computadores comuns, ela traz dificuldades para a implementação desses algoritmos em ambientes computacionais restritos, frequentemente utilizados para tecnologias IoT (Internet of Things).

Com a popularização de dispositivos inteligentes no dia-a-dia das pessoas, a adaptação dos algoritmos de criptografia pós-quântica para sistemas restritos se tornou um desafio a ser vencido, já que é importante que esses dispositivos também estejam protegidos contra ataques de computadores quânticos. Sendo assim, este artigo apresenta e avalia técnicas de otimização propostas na literatura para o algoritmo SABER, candidato finalista da terceira rodada do processo de padronização do NIST, com o intuito de conhecer seus requisitos de memória e processamento em ambientes computacionais restritos. Além disso, este trabalho também implementa e avalia a adaptação dessas otimizações para o LightSABER, versão mais rápida e menos segura do SABER, e para o FireSABER, versão mais lenta e mais segura. Por fim, também são avaliados os impactos de diferentes níveis de otimização durante a compilação dos códigos fonte, a fim de determinar o melhor deles para cada versão do algoritmo.

2. Reticulados, LWE, LWR e Module-LWR

Um reticulado em $\mathbb{R}^n$ é um subgrupo discreto de $\mathbb{R}^n$ definido por um conjunto S de pontos distribuídos em um espaço *n*-dimensional, tal que todos os pontos pertençam a um espaço definido e tenham sido gerados a partir de uma base dada em S [9]. Por sua vez, uma base é definida por um conjunto de *n* vetores em S - dados pelo segmento que liga dois pontos de S -, centrados na mesma origem e linearmente independentes. A partir desta definição, conclui-se que existem diversas bases para representar um mesmo reticulado, elas se classificam em: bases boas, aquelas cujos vetores tenham maior ortogonalidade, e bases ruins, que são bases cujos vetores tenham menor ortogonalidade. A Figura 1 ilustra um exemplo de reticulado bidimensional e duas de suas bases, uma boa ($\mathbf{u_1}$ e $\mathbf{u_2}$) e uma ruim ($\mathbf{v_1}$ e $\mathbf{v_2}$).

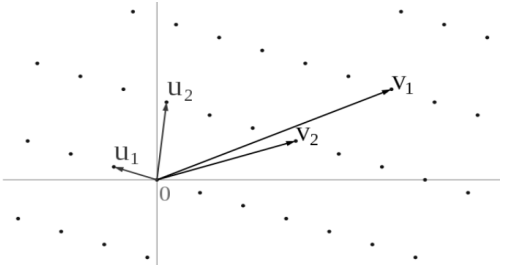

Figura 1. Um reticulado bidimensional e duas de suas bases distintas.

A partir desse conceito de reticulados, existem diversos problemas matemáticos propostos, como o Shortest Vector Problem (SVP) e o Closest Vector Problem (CVP) [10]. O primeiro consiste em encontrar o menor vetor (vetor de menor norma) não nulo em um reticulado S, dados o reticulado e a sua base. Já o segundo, consiste em encontrar o vetor **u**, pertencente a S, mais próximo do vetor **v** (não necessariamente pertencente a S), dados o vetor **v**, o reticulado e sua base. O vetor mais próximo corresponde ao vetor **u** que possui coeficientes mais próximos dos coeficientes de **v**. Tais problemas são difíceis de resolver para reticulados com muitas dimensões e com bases suficientemente ruins, o que é interessante para a criptografia. Alguns algoritmos, como o GGH (Goldreich-Goldwasser-Halevi) [11], utilizam uma base ruim para geração da chave pública e uma base boa para geração da chave privada.

No entanto, a maioria dos algoritmos de criptografia assimétrica baseados em reticulados utiliza o problema Learning with Errors (LWE) [12] e suas variações. O problema consiste em resolver equações matriciais, que representam a combinação linear de vetores, acrescida de ruídos (chamados de erros). A importância desses ruídos é de impedir a utilização do método de eliminação de Gauss-Jordan (ou qualquer outro método com objetivos semelhantes) para resolver o sistema de equações rapidamente, o que garante a dificuldade desse problema. A Eq. 1 é a base para o LWE, sendo que a matriz $\mathbf{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{l \times 1})$ - em que $\mathcal{U}$ corresponde a distribuição uniforme e $\mathbb{Z}_q$ corresponde ao anel de números inteiros módulo um inteiro q - e o vetor **b** formam a chave pública (pk), o vetor $\mathbf{s} \leftarrow \ _{\mu}(\mathbb{Z}_q^{l \times 1})$ - em que $_{\mu}$ corresponde à distribuição binomial centralizada com parâmetro μ - corresponde à chave privada (sk), o vetor $\mathbf{e} \leftarrow \ _{\mu}(\mathbb{Z}_q)$ corresponde aos ruídos e o inteiro p define o valor sobre o qual serão realizadas as operações modulares em cada elemento da matriz e dos vetores.

$$\mathbf{A} \times \mathbf{s} + \mathbf{e} = \mathbf{b} \qquad \textbf{(Eq. 1)}$$

Uma variação importante do problema do LWE é o Learning with Rounding (LWR) [13], cuja diferença está na geração dos ruídos. Ao invés de gerar os ruídos de forma aleatória, o LWR gera os ruídos de forma determinística. Isso se dá a partir do arredondamento da operação de multiplicação do produto entre a matriz pública **A** e o vetor secreto **s**, pela razão entre p e q, dois inteiros primos entre si. Dessa forma, o LWR diminui a dependência da geração de números aleatórios, em relação ao LWE.

Ainda sobre o LWR, destaca-se sua variação Module-LWR (baseada no problema do Module-LWE [14]) utilizada no algoritmo SABER. A estrutura do problema é a mesma, porém, ao invés de as entradas dos vetores **s** e **b** e da matriz **A** serem inteiros pertencentes ao conjunto $\mathbb{Z}_q$, elas são polinômios pertencentes ao anel quociente $R_q = \mathbb{Z}_q/(x^n-1)$ (sendo n um inteiro que define o maior grau possível para os polinômios deste conjunto). As vantagens dessa abordagem são a diminuição da complexidade computacional do algoritmo, a diminuição dos bits a serem compartilhados e a proteção contra ataques à estrutura de anel dos Ring-LWE/LWR (outra variação do problema). Além disso, essa abordagem permite uma flexibilidade para aumentar o nível de segurança sem mudar a aritmética por trás.

3. Segurança e indistinguibilidade

Um dos critérios básicos de segurança para esquemas de criptografia assimétrica é a indistinguibilidade (IND) [15], que indica que o adversário (modelado como uma máquina de Turing probabilística que executa códigos em tempo polinomial) é incapaz de distinguir pares de texto cifrado baseados nas mensagens que ele cifrou. Essa propriedade é definida a partir de um jogo. Dadas a chave pública, duas mensagens m_0 e m_1 e uma função de cifração E, o adversário envia m_0 e m_1 para o desafiante (que controla E) e uma delas é escolhida aleatoriamente para ser retornada cifrada. O esquema é considerado indistinguível se o adversário (mesmo conhecendo m_0 e m_1) não consegue adivinhar, com probabilidade maior que 50%, qual mensagem foi retornada cifrada. Existem três tipos de segurança baseados no critério IND [15]: IND-CPA (*INDistinguishability under Chosen Plaintext Attack*), IND-CCA (*INDistinguishability under Chosen Ciphertext Attack*) e IND-CCA2 (*INDistinguishability under adaptive Chosen Ciphertext Attack*). Esses tipos são definidos a partir de variações no jogo descrito anteriormente, que introduzem novos mecanismos para o adversário.

O primeiro tipo, IND-CPA, significa indistinguibilidade sob ataque de texto em claro escolhido. Nele, o adversário pode fazer quantas cifrações desejar antes e depois de submeter a mensagem para o desafiante (com exceção da cifração de m_0 e m_1), e analisar o texto cifrado resultante. Já o segundo tipo, significa indistinguibilidade sob ataque de texto cifrado escolhido não-adaptativo. Nessa versão do jogo, o adversário possui acesso a uma função de decifração, podendo usá-la quantas vezes quiser, desde que seja antes de enviar a mensagem a ser cifrada. Por fim, o terceiro tipo significa indistinguibilidade sob ataque de texto cifrado escolhido adaptativo. Nele o adversário pode utilizar a função de decifração mesmo depois de ter recebido a mensagem do desafiante. Vale ressaltar que o adversário não pode decifrar o texto cifrado recebido.

Para todos os três tipos descritos, o adversário não pode possuir vantagem não desprezível no jogo. Esses tipos de segurança foram citados em ordem crescente de segurança, pois um algoritmo IND-CCA2 seguro também é IND-CCA seguro e IND-CPA seguro. O SABER é um esquema IND-CCA2 seguro, assim como os outros finalistas da terceira rodada do processo do NIST, tendo o maior nível de segurança no critério IND. Na seção seguinte, será discutida a construção do SABER e como ele oferece esse tipo de segurança.

4. SABER, LightSABER e FireSABER

O SABER é um mecanismo de encapsulamento de chaves (*Key Encapsulation Mechanism* ou KEM) IND-CCA2 seguro, baseado no problema Module-LWR, como citado anteriormente. Como todo KEM, ele possui as operações de geração de chaves (*key generation*), encapsulamento (*encapsulation*) e desencapsulamento (*decapsulation*). Essas operações são obtidas a partir de uma versão pós-quântica da transformada Fujisaki-Okamoto [16] aplicada sobre a versão IND-CPA do SABER. Esta, por sua vez, contém as operações de geração de chaves, cifração (*encryption*) e decifração (*decryption*), que estão descritas nos Algoritmos 1, 2 e 3, respectivamente. Um detalhe importante do SABER é que todas as reduções modulares são feitas sobre

potências de dois, o que facilita as operações aritméticas e diminui a complexidade computacional do algoritmo.

Na geração de chaves, é criada uma semente (ou *seed*) aleatória de 256 bits que é utilizada para geração da matriz pública **A** de dimensão l x l, através da função de *hash* SHAKE-128. O vetor secreto **s** possui dimensão l e é gerado por meio de uma distribuição binomial centralizada β_μ com parâmetro μ. Já o vetor público **b** é obtido através da seleção de bits do resultado da multiplicação da matriz **A** pelo vetor **s** acrescido de um vetor **h** (que emula o arredondamento típico do LWR). Essa seleção é feita por uma função chamada `bits(x,i,j)` que tem como parâmetros um valor x, uma posição i e o número j de bits que se deseja selecionar ($j \leq i$). Quando utilizada, ela retorna j bits consecutivos do valor x, começando de um menos significativo ((i-j+1)-ésimo bit) e indo até um mais significativo (i-ésimo bit). Para efeito dessa função, as posições dos bits são contadas de 1 (bit menos significativo) até o número total de bits do valor (indicando o bit mais significativo) Sendo assim, `bits(x,i,i)` equivale a x mod 2^i e `bits(x,i,j)` equivale a $x/2^{i-j}$ mod 2^j, operações recorrentes no algoritmo. Vale ressaltar que a chave pública é composta pelo vetor **b** e pela semente da matriz **A**, evitando a inclusão de toda a matriz na mesma.

Algoritmo 1. Operação de geração de chaves [7].

Algorithm 1: `Saber.KeyGen()`

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} \leftarrow \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
$\boldsymbol{s} \leftarrow \beta_\mu(R_q^{l \times 1})$
$\boldsymbol{b} = \mathtt{bits}(\boldsymbol{A}\boldsymbol{s} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
return $(pk := (\boldsymbol{b}, seed_{\boldsymbol{A}}), sk := \boldsymbol{s})$

Algoritmo 2. Operação de cifração [7].

Algorithm 2: `Saber.Enc`$(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in \mathcal{M}; r)$

$\boldsymbol{A} \leftarrow \mathtt{gen}(\mathrm{seed}_{\boldsymbol{A}}) \in R_q^{l \times l}$
$\boldsymbol{s'} \leftarrow \beta_\mu(R_q^{l \times 1})$
$\boldsymbol{b'} = \mathtt{bits}(\boldsymbol{A}^T\boldsymbol{s'} + \boldsymbol{h}, \epsilon_q, \epsilon_p) \in R_p^{l \times 1}$
$v' = \boldsymbol{b}^T\mathtt{bits}(\boldsymbol{s'}, \epsilon_p, \epsilon_p) + h_1 \in R_p$
$c_m = \mathtt{bits}(v' + 2^{\epsilon_p - 1}m, \epsilon_p, \epsilon_t + 1) \in R_{2t}$
return $c := (c_m, \boldsymbol{b'})$

Algoritmo 3. Operação de decifração [7].

Algorithm 3: `Saber.Dec`$(sk = \boldsymbol{s}, c_m, \boldsymbol{b'})$

$v = \boldsymbol{b'}^T\mathtt{bits}(\boldsymbol{s}, \epsilon_p, \epsilon_p) + h_1 \in R_p$
$m' = \mathtt{bits}(v - 2^{\epsilon_p - \epsilon_t - 1}c_m + h_2, \epsilon_p, 1) \in R_2$
return m'

Por sua vez, a cifração é composta por operações semelhantes às da geração de chaves. Há também a multiplicação do vetor **b** (recebido via chave pública da outra parte) pelo vetor secreto próprio **s'** com arredondamento (feito através da adição do

polinômio h_1 e da seleção de bits), e um mecanismo de reconciliação que diminui consideravelmente a chance de falha do algoritmo, a ponto de ser desprezível (a descrição do mecanismo foi suprimida por questões de espaço, mas pode ser vista em uma das referências [17]). A decifração é simples e suas operações são semelhantes às da cifração. Ambas operações são feitas para mensagens m de 32 bytes, correspondentes a chaves simétricas de 256 bits que serão usadas para o tráfego seguro de maior volume de dados com algoritmos simétricos. Os parâmetros c_m, t e h_2 estão relacionados ao mecanismo de reconciliação. Para todas as operações, $\varepsilon_p = 10$, $\varepsilon_q = 13$ e $\varepsilon_t = 3$ são constantes que expressam o número de bits das constantes p, q e t, respectivamente. Além disso, os parâmetros que aparecem em mais de um Algoritmo e têm o mesmo nome são os mesmos. O parâmetro l corresponde ao número de polinômios por linha (e por coluna) da matriz **A** e ao número de polinômios dos vetores **s** e **b**.

Tabela 1. Configurações das versões de SABER [7].

Características	LightSABER	SABER	FireSABER
Categoria de segurança	1	3	5
Probabilidade de falha	2^{-120}	2^{-136}	2^{-165}
Nível de segurança clássica (*primal* / *dual attack*)	126 / 126	199 / 198	270 / 270
Nível de segurança quântica (*primal* / *dual attack*)	115 / 115	181 / 180	246 / 245
Parâmetro t	2^2	2^3	2^5
Parâmetro μ	10	8	6
Parâmetro l	2	3	4
Tamanho da chave pública (bytes)	672	992	1.312
Tamanho da chave privada (bytes)	1.568	2.304	3.040
Tamanho do texto cifrado (bytes)	736	1.088	1.472

O SABER possui algumas características bem importantes para seu funcionamento correto e seguro. Dentre elas, está o parâmetro n, que corresponde ao grau do polinômio no denominador do anel quociente do Module-LWR, definido como 256 (suficiente para garantir a segurança), com coeficientes de 13 bits. Outros valores importantes são $p = 2^{10}$ e $q = 2^{13}$, potências de dois suficientemente grandes para garantir a segurança e facilitar as reduções modulares. Por fim, destaca-se o uso da norma FIPS 202 [18], que define funções *hash* criptográficas SHA-3 padronizadas pelo NIST.

A partir dessa construção, os autores do SABER criaram duas outras versões: o LightSABER e o FireSABER. A primeira, é uma versão mais simples e com grau de segurança menor, o que a torna menos custosa computacionalmente. A segunda é uma versão mais robusta e mais segura, com custo computacional maior. Ambas possuem a mesma estrutura que o SABER, mudando apenas alguns parâmetros, em especial o parâmetro μ da distribuição binomial centralizada, o parâmetro t relacionado ao mecanismo de reconciliação, e o parâmetro l. As diferenças entre as três versões do algoritmo podem ser vistas na Tabela 1.

5. Otimizações propostas na literatura para otimização de memória

Como visto na seção anterior, o SABER é um algoritmo que depende de vetores e de uma matriz grandes, especialmente porque cada polinômio possui 256 coeficientes de 13 bits cada. Essa característica faz com que haja um grande consumo de memória, tanto para armazenar e acessar essas estruturas, quanto para realizar operações aritméticas entre elas. Por conta disso, para ambientes computacionais restritos, o maior desafio da implementação do SABER corresponde à quantidade de memória consumida pelo algoritmo. Portanto, já foram propostas diversas otimizações de memória, as quais serviram de base para as que foram implementadas neste trabalho. Mais especificamente, as otimizações são feitas para reduzir a quantidade de memória de pilha, memória extra alocada estaticamente durante a execução do programa, já que o algoritmo não faz alocações dinâmicas e as variáveis globais possuem tamanho fixo.

Em todo o esquema, a função que utiliza mais espaço na memória de pilha é a de multiplicação de polinômios, que é invocada várias vezes em qualquer operação de multiplicação entre vetores e entre matriz e vetor. Ao contrário da maioria dos algoritmos que utilizam polinômios, a aritmética modular do SABER não possui como módulo um número primo, e sim uma potência de dois. Sendo assim, não é possível utilizar a Number Theoretic Transform (NTT) [19], que é a forma mais eficiente de se fazer tais operações. Por esse motivo, o SABER realiza a multiplicação de polinômios a partir de uma combinação dos algoritmos de Toom-Cook [20] e de Karatsuba [20][21], que mostrou ser a alternativa compatível com ele mais eficiente.

No entanto, esses algoritmos são recursivos e não são *in-place* (isto é, utilizam memória extra), o que faz com que haja um grande consumo de memória de pilha, da ordem de O(n) (sendo n o número de coeficientes dos polinômios que serão multiplicados), recurso bastante limitado em ambientes restritos. Para contornar esse problema, foi proposto por Karmakar et. al [22] uma versão do algoritmo de Karatsuba que é *in-place* e mais eficiente em memória, em troca de um número maior de ciclos de clock. Essa versão, juntamente com o método tradicional de multiplicação de polinômios, substitui o algoritmo que combina os métodos de Toom-Cook e de Karatsuba, consumindo apenas O(log n) de memória extra de pilha.

Além disso, Karmakar et. al propõem otimizações acerca da geração da matriz $\mathbf{A}$ e da geração do vetor secreto $\mathbf{s}$. Como parte do esforço para reduzir o uso de memória, o artigo sugere uma estratégia *just-in-time* para geração dessas duas estruturas. Em outras palavras, quando elas forem utilizadas para alguma operação, ao invés de gerar a matriz ou o vetor inteiro de uma só vez, os polinômios são gerados um por vez, reaproveitando seu espaço na memória. Sendo assim, elas não ocupam espaço de memória de pilha a

mais do que o necessário, o que evita um overflow de memória. No caso da matriz, um polinômio é gerado, multiplicado e, em seguida, o espaço ocupado por ele é reutilizado. Já no caso do vetor, o polinômio é gerado, copiado para um vetor e, por fim, o espaço dele na memória, naquela função, é reaproveitado. Neste último caso, note que o vetor secreto não chega a ser gerado: o que acontece é que os polinômios são gerados um por vez e, então, copiados para um outro vetor que pertence a outra função. O efeito que isso gera é que a função de geração do vetor secreto não precisa ocupar mais memória, se um outro vetor com o mesmo propósito já alocou o espaço necessário.

Ambas otimizações se baseiam no fato de que a função interna ao SHAKE-128, a keccak_squeezeblocks(), produz 168 bytes por vez [18]. Essa função é utilizada para construção da matriz **A**, a partir da semente, e para construção dos coeficientes dos polinômios **s**, a partir de uma outra semente e da distribuição binomial centralizada. A partir da estratégia *just-in-time*, pode-se concluir que, como vários polinômios vão ser construídos um por vez, é importante que o uso da função keccak_squeezeblocks() seja otimizado. Porém, a geração de cada polinômio requer um número de bytes que não é múltiplo de 168, o que faz com que exista uma quantidade de bytes que sobram entre duas chamadas dessa função. A implementação original do SABER possui um buffer grande para guardar todos os bytes necessários para a matriz e um para o vetor, mas isso contrasta com a ideia da estratégia. Logo, para possibilitar que esta seja eficiente, constrói-se um buffer menor, que possui espaço suficiente apenas para guardar os bytes que sobraram da chamada anterior concatenados com os novos bytes gerados. Sendo assim, é possível implementar a estratégia de forma que economize memória.

Além das otimizações descritas até aqui, também foram adaptadas duas outras desenvolvidas para microcontroladores ARM Cortex-M4 [23]. A primeira delas consiste em utilizar apenas 4 bits para codificar os coeficientes dos polinômios do vetor secreto **s**, ao invés de 13 bits. Para esquemas que não utilizam o NTT, essa mudança não causa nenhum impacto negativo na sua segurança [23]. Portanto, é possível economizar memória sem comprometer a segurança e sem modificar drasticamente o desempenho de velocidade. Aliás, funções de empacotamento ficam mais simples, pois dois coeficientes podem ser representados por um único byte. Por fim, os autores do artigo também desenvolveram uma verificação *in-place* do texto cifrado durante o desencapsulamento, o que também otimiza o uso da memória. Vale destacar que ambas as otimizações não dependem de instruções específicas do microcontrolador utilizado, o que permite que elas sejam replicadas para outras arquiteturas.

6. Adaptações para LightSABER e FireSABER

Karmakar, A., et. al [22] só implementaram suas otimizações propostas no SABER. Por isso, foi necessário adaptá-las aos códigos do LightSABER e do FireSABER. A primeira adaptação necessária foi na parte de geração do texto cifrado, invocada durante a operação de cifração para empacotar o valor cifrado de m (c_m). Esse empacotamento é feito de forma diferente para cada valor do parâmetro t, o que faz com que sejam necessárias três funções de empacotamento. Sendo assim, o código foi adaptado para realizar esse passo de acordo com os parâmetros escolhidos. De forma análoga, a etapa de extração bits é feita durante a operação de decifração para desempacotar e decifrar a mensagem a partir de c_m. Como o desempacotamento reverte

o que foi feito no empacotamento, esse processo também é diferente para cada valor de *t* e o código foi adaptado da mesma forma.

Por fim, a última alteração necessária foi na função GenSecret(), responsável pela construção do vetor secreto. Como visto anteriormente, foi utilizada uma estratégia *just-in-time* durante esse processo, criando um polinômio por vez. Uma vez que cada versão do SABER possui um valor diferente para o parâmetro *l*, cada uma delas possui um número de polinômios diferente no vetor **s**. Sendo assim, foi feita uma versão dessa função para cada valor de *l*. O SABER possui $l = 3$, o que significa que seu vetor secreto inclui três polinômios. Para o LightSABER em que $l = 2$, basta adaptar a função do SABER e não construir o terceiro polinômio. Quanto ao FireSABER, cujo vetor secreto conta com 4 polinômios, foi estendida a função do SABER para que fossem gerados os bytes necessários para geração do quarto polinômio.

7. Implementação em ambiente restrito

O algoritmo resultante das otimizações descritas foi implementado na placa de desenvolvimento FRDM-KL25Z, um ambiente computacional restrito. A placa conta com uma unidade de microcontrolador (MCU) KL25Z128 (processador ARM Cortex-M0+), 128 kB de memória flash, 16 kB de memória SRAM e 48 MHz de frequência de clock. A memória *flash* (não volátil) armazena o código do programa (seção .text) e os dados iniciais (seção .data), enquanto a memória SRAM (volátil) armazena, durante a execução, informações temporárias como a pilha e as seções .bss (Block Starting Symbol) - que contém espaço alocado estaticamente para variáveis - e .data. A seção de memória *heap* não é utilizada no algoritmo trabalhado, por este não fazer uso de alocação dinâmica. Todos os testes foram realizados utilizando a IDE MCUXpresso, desenvolvida pela NXP, que conta com uma ferramenta para medição de uso de memória de pilha. Essa ferramenta possui um compilador embutido gcc versão 10.3.1 20210824 (release) (GNU Arm Embedded Toolchain versão 10.3-2021.10). Deve ser notado que o algoritmo utiliza parte da memória SRAM da placa para variáveis globais, o que foi contabilizado a partir da soma da quantidade de memória que cada uma delas consumiu. Por esse motivo, apesar de a placa possuir 16 kB de memória SRAM, parte dela não está disponível para pilha pelo fato de que as variáveis globais ocupam um espaço considerável. Para contagem de ciclos, utilizou-se a interface CMSIS (Cortex Micro-Controller Software Interface and Standard), que conta com ferramentas para isso.

Foram executados testes nas três versões do algoritmo, já que todas mostraram ser compatíveis com a placa. Além disso, foram feitos vários testes para diversas flags de otimização na compilação, sendo elas: -O0, -Os, -O1, -O2 e -O3. A primeira corresponde à compilação sem otimização, a segunda otimiza, principalmente, a memória de programa, também reduzindo os ciclos de clock, e as três últimas indicam níveis crescentes de otimização de ciclos de clock. Após vários testes, constatamos que a flag -O1 foi a mais adequada para o LightSABER e a flag -O2, a melhor para o SABER e para o FireSABER, quanto à economia de memória e de ciclos de clock. Os dados desses testes foram suprimidos por questões de espaço.

Os valores resultantes das melhores flags de otimização para cada algoritmo se encontram na Tabela 2. A partir dela, pode-se notar que o consumo de memória de pilha

ficou abaixo de 40% da memória total disponível, e se manteve relativamente próximo para as três versões do algoritmo. No entanto, o número de ciclos de clock variou bastante entre as versões. Optou-se por avaliar o número de ciclos de clock e o consumo de memória de pilha para cada uma das funções do KEM separadamente, visto que alguns contextos podem realizar uma função com mais frequência do que as demais. Como explicado anteriormente, as otimizações propostas na literatura visam diminuir o uso de memória de pilha, recurso mais escasso que a memória de programa e cujo consumo está atrelado à alocação de memória estática. Por esse motivo, a Tabela 2 destaca esse tipo de memória.

Tabela 2. Consumo de memória de pilha e ciclos de clock dos algoritmos nas melhores configurações de compilação.

Algoritmo		Geração de chaves (variação %)	Encapsulamento (variação %)	Desencapsulamento (variação %)
SABER (-O2)	Memória (kB)	4,13	3,75	3,77
	Ciclos de clock	4.495.576	5.940.149	6.930.342
LightSABER (-O1)	Memória (kB)	3,36 (-19%)	3,46 (-8%)	3,49 (-7%)
	Ciclos de clock	2.172.163 (-52%)	3.157.820 (-47%)	3.815.365 (-45%)
FireSABER (-O2)	Memória (kB)	4,88 (+18%)	4,00 (+7%)	4,02 (+7%)
	Ciclos de clock	7.743.499 (+72%)	9.630.721 (+62%)	10.973.725 (+58%)

Os resultados dispostos na Tabela 2 indicam que a operação de desencapsulamento é a mais custosa, em número de ciclos de clock, dentre as três operações do KEM, o que é esperado dada sua natureza. Porém, nota-se que, ao contrário das outras duas versões, a geração de chaves no LightSABER não é a operação que consome mais memória de pilha. Isso possivelmente se dá pelo fato de que o LightSABER possui um parâmetro μ maior do que as outras versões, por questões de segurança, o que significa que as funções de encapsulamento e desencapsulamento devem operar com mais bits nas funções relacionadas ao mecanismo de reconciliação.

Quanto ao gasto de memória com variáveis globais, existem dois tipos: variáveis inicializadas, que se localizam na seção .data da memória; e variáveis não inicializadas, que são armazenados na seção .bss. Esses dados se encontram na Tabela 3, juntamente com o uso observado das duas seções de memória citadas. Em relação a eles, destaca-se que o gasto teórico foi calculado a partir das variáveis globais não inicializadas

declaradas explicitamente nos arquivos (tipo .c) do código fonte. Esses valores diferem do espaço efetivamente utilizado na seção .bss, provavelmente por conta de dados que a própria IDE inseriu no código para inicialização e controle. Ademais, não foram levados em conta nos valores teóricos os gastos relacionados às possíveis variáveis globais presentes no arquivo tipo .s contendo código em linguagem *assembly* de funções *hash* (FIPS 202), por conta da dificuldade de contabilizá-las. O mesmo vale para a seção .data, cujo uso não havia sido previsto nas análises teóricas.

Por fim, a Tabela 3 também inclui o consumo de memória relacionado ao programa em si, localizado na seção .text da memória. Esses valores foram obtidos através da compilação com a melhor flag de otimização para cada versão do algoritmo. Como o programa ocupou menos da metade do espaço disponível (128 kB), esses valores não são tão relevantes e, por esse motivo, eles não foram levados em conta no processo de decisão da melhor flag para cada algoritmo.

Tabela 3. Consumo de memória relacionado às variáveis globais e de programa.

Grandeza	LightSABER	SABER	FireSABER
Valor teórico (.bss)	3,04 kB	4,45 kB	5,89 kB
Seção .bss	5,03 kB	6,44 kB	7,88 kB
Seção .data	2,48 kB	2,48 kB	2,48 kB
Seção .text	55,04 kB	54,53 kB	54,70 kB

Na Tabela 4 estão dispostos os resultados de consumo de memória e de ciclos de clock obtidos para o algoritmo SABER comparados com os valores de referência disponibilizados no site oficial do algoritmo e pelo PQM4 [8]. Não foi possível fazer comparações para as outras versões (LightSABER e FireSABER), pois os dados referentes às mesmas para o Cortex-M0 não foram encontrados. Os resultados obtidos ficaram estáveis, isto é, não variaram com outras execuções do algoritmo.

Com base na Tabela 4, conclui-se que as otimizações feitas neste trabalho para o SABER apresentaram resultados consideravelmente melhores do que a versão para Cortex-M0 estabelecida na literatura [22], especialmente em memória, como observado na variação percentual. Acredita-se que o principal motivo para essa diferença sejam as otimizações inspiradas na implementação para Cortex-M4 [23]. Dentre as diferenças entre as arquiteturas ARM Cortex-M0 e Cortex-M0+, destaca-se a mudança de um pipeline de três estágios para dois estágios, o que pode produzir impactos no número de ciclos de clock.[1] Outra mudança foi o aumento da frequência de clock, porém isso não afeta a comparação pois foi medido o número de ciclos de clock por operação.

Ademais, também se pode observar que ambas as versões para Cortex-M4 - uma que otimiza velocidade e outra que otimiza memória e velocidade - apresentam resultados melhores, especialmente em ciclos de clock, o que é de se esperar dado o

[1] A documentação do processador ARM Cortex-M0+ pode ser encontrada no link: https://developer.arm.com/documentation/ddi0484/b/

poder computacional maior dessa arquitetura. Entretanto, os resultados deste trabalho são importantes pois tornam a execução em Cortex-M0+ mais eficiente e mais econômica, consumindo menos memória do que a implementação focada em velocidade do Cortex-M4 e se aproximando bastante ao consumo de memória da versão para Cortex-M4 focada em otimizar esse recurso.

Tabela 4. Resultados dos melhores testes do SABER comparados com os valores de referência.

Função		SABER em M0 [22]	SABER em M4 *speed/memory*[2] [8] (variação %)	SABER em M4 *speed*[3] [8] (variação %)	Este trabalho (variação %)
Key generation	Memória (kB)	5,03	3,79 (-25%)	6,64 (+32%)	4,13 (-18%)
	Ciclos (mil)	4.786	820 (-83%)	645 (-87%)	4.495 (-6%)
Encapsulation	Memória (kB)	5,12	3,18 (-38%)	7,32 (+43%)	3,75 (-27%)
	Ciclos (mil)	6.328	1.059 (-83%)	851 (-87%)	5.940 (-6%)
Decapsulation	Memória (kB)	6,22	3,19 (-49%)	7,32 (+18%)	3,77 (-39%)
	Ciclos (mil)	7.509	1.038 (-86%)	774 (-90%)	6.930 (-8%)

8. Conclusão e trabalhos futuros

A partir de estudos de otimizações feitas na literatura, foi possível tornar o SABER e suas variações mais eficientes na plataforma ARM Cortex-M0+. Os resultados foram produzidos a partir da versão IND-CCA2 do SABER, que é a mais segura. Além disso, foi determinado qual flag de otimização de compilação é a mais adequada para cada um dos algoritmos: a flag -O1 é a que produz melhores resultados para execução do LightSABER e a flag -O2 é a que se mostrou melhor para o SABER e para o FireSABER.

Os resultados alcançados neste trabalho foram melhores que os existentes na literatura para Cortex-M0, tanto em ciclos de clock como em consumo de memória. No primeiro caso, a redução foi de 6 a 8% e, no segundo caso, variou entre 18 e 39%, considerando apenas a versão de segurança média do algoritmo. Pelas diferenças entre

[2] Versão dedicada a otimização de memória de pilha, levando em conta a velocidade.
[3] Versão dedicada a otimização de velocidade de execução.

SABER e suas variações LightSABER e FireSABER, é esperado que reduções similares ocorram nestes casos também. Em comparação com resultados da literatura para o mais avançado Cortex M4, as implementações em Cortex M0 e M0+ acabam tendo um pior desempenho. Entretanto, deve ser notado que as otimizações aplicadas ao SABER em Cortex M0+ resultaram em consumo de memória similar àquele da versão em Cortex M4 (otimizado para diminuir consumo de memória) e até mesmo inferior, quando este último tem seu código otimizado para velocidade.

Para trabalhos futuros, sugerimos avaliar a possibilidade de otimizar a versão do algoritmo desenvolvida neste trabalho em nível de programação em linguagem assembly, o que poderia produzir resultados ainda melhores tanto em consumo de memória quanto de tempo. Ademais, podem ser pesquisadas e trazidas, para essa arquitetura, eventuais otimizações feitas para outras plataformas, visando melhorar ainda mais a relação custo-benefício de executar o algoritmo no Cortex-M0+. Como o consumo de memória de pilha foi reduzido em relação a implementações anteriores, seria interessante focar em otimizações de velocidade que não aumentem o uso de memória.

9. Referências

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

[2] Proos, J., Zalka, C. (2003), "Shor's discrete logarithm quantum algorithm for elliptic curves", eprint arXiv:quant-ph/0301141.

[3] National Institute of Standard and Technology – NIST (2016), "Request for Comments on Post-Quantum Cryptography Requirements and Evaluation Criteria", Notice 81 FR 50686, p. 50686-50687. https://csrc.nist.gov/projects/post-quantum-cryptography

[4] Bernstein, D., Chou, T., et. al (2017), "Classic McEliece: conservative code-based cryptography, 'Supporting Documentation'". https://classic.mceliece.org/nist.html.

[5] Bos, J., Ducas, L., Kiltz, et. al (2018), "CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM", 2018 IEEE European Symposium on Security and Privacy, EuroS&P.

[6] Hoffstein, J., Pipher, J. & Silverman, J. (1996), "NTRU: A new high-speed public key cryptosystem", Manuscript circulated at CRYPTO 1996 rump session.

[7] D'Anvers, JP., Karmakar, A., et. al (2018), "Saber: Module-LWR Based Key Exchange, CPA-Secure Encryption and CCA-Secure KEM", In: Joux A., Nitaj A., Rachidi T. (eds) Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings. Lecture Notes in Computer Science 10831, Springer 2018, ISBN 978-3-319-89338-9. https://doi.org/10.1007/978-3-319-89339-6_16

[8] Kannwischer, M. et. al (2019), "PQM4: Post-quantum crypto library for the ARM Cortex-M4", https://github.com/mupq/pqm4. Último acesso em: 27-06-2022.

[9] Conway, J., Sloane, N. (1999), “Sphere Packings, Lattices and Groups”, Grundlehren der Mathematischen Wissenschaften, 290 (3rd ed.), Berlin, New York: Springer-Verlag, ISBN 978-0-387-98585-5, MR0920369.

[10] Micciancio, D., Regev, O. (2008), “Lattice-based cryptography”, CSE Department, University of California, San Diego.

[11] Goldreich, O., Goldwasser, S. & Halevi, S. (1997), “Public-key cryptosystems from lattice reduction problems”, Lecture Notes in Computer Science, vol. 1294.

[12] Regev, O. (2005), “The Learning with Errors Problem”, Courant Institute of Mathematical Sciences, New York University.

[13] Banerjee, A., Peikert, C. & Rosen, A. (2012), “Pseudorandom Functions and Lattices”, in: Pointcheval, D., Johansson, T. (eds) Advances in Cryptology – EUROCRYPT 2012. EUROCRYPT 2012. Lecture Notes in Computer Science, vol 7237. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-29011-4_42

[14] Langlois, A., Stehlé, D. (2015), “Worst-case to average-case reductions for module lattices”, Designs, Codes and Cryptography, 75(3):565–599

[15] Bogdanov, D. (2005), “IND-CCA2 secure cryptosystems”, University of Tartu.

[16] Hofhein, D., Hövelmanns, K. & Kiltz, E. (2017), “A Modular Analysis of the Fujisaki-Okamoto Transformation”, Karlsruhe Institute of Technology.

[17] Tolhuizen, L., et. al (2017), “Improved key-reconciliation method”, IACR Cryptol.

[18] May, W. (2015), “SHA-3 Standard: Permutation-Based and Extendable-Output Functions”, Federal Information Processing Standards Publication, National Institute of Standards and Technology.

[19] Hedge, S., Nagapadma, R. (2019), “Number Theoretic Transform for Fast Digital Computation”, Department of Electronic and Communication Engineering, NIE Institute of Technology.

[20] Crandall, R., Pomerance, C. (2005), “Prime Numbers – A Computational Perspective”, Second Edition, Springer, Section 9.5.1: Karatsuba and Toom–Cook methods, p.473.

[21] Karatsuba, A., Ofman, Y. (1963), “Multiplication of multidigit numbers on automata”, Sov Phys Dokl 7:595–596.

[22] Karmakar, A., et. al (2018), “Saber on ARM. CCA-secure module lattice-based key encapsulation on ARM”, In Transactions in Cryptographic Hardware and Embedded Systems.

[23] Mera, J., Karmakar, A. & Verbauwhede, I. (2020), “Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography”, In Transactions in Cryptographic Hardware and Embedded Systems.