XXX Congresso de Iniciação Científica UNICAMP

Avaliação de requisitos mínimos de processamento e memória em esquemas de criptografia pós-quântica

Palavras-chave: Criptografia Pós-Quântica, reticulados, criptografia assimétrica

Autores:

George Gigilas Junior [IC - Unicamp]

Prof. Dr. Marco Aurélio Amaral Henriques (orientador) [FEEC - Unicamp]

INTRODUÇÃO

Com o iminente advento dos computadores quânticos, diversos problemas mais complexos poderão ser resolvidos com seu enorme potencial computacional. Ao mesmo tempo que eles terão papel importante para diversas pesquisas, a segurança de inúmeras aplicações na Internet estão sob ameaça, por conta do impacto da computação quântica nos esquemas de criptografia de chave pública utilizados atualmente. Nesse contexto, o instituto americano National Institute of Standards and Technology (NIST) iniciou, em 2016, um processo de padronização de algoritmos pós-quânticos, tanto de criptografia assimétrica, quanto de assinatura digital [1]. Os finalistas da terceira rodada de avaliações foram anunciados em 2021. Como três dos quatro algoritmos de troca de chaves são baseados em problemas com reticulados, a probabilidade era alta de que pelo menos um algoritmo desse tipo fosse escolhido.

Em contrapartida, as chaves desses algoritmos são bastante grandes, comparadas com as chaves dos esquemas atuais, o que demanda um consumo elevado de memória e de ciclos de clock durante as operações com elas. Porém, além da implementação em computadores tradicionais, também é importante que os dispositivos mais restritos, como os de IoT (Internet of Things), também estejam protegidos, especialmente com o uso cada vez mais frequente dessa tecnologia no dia-a-dia. Tais dispositivos são mais restritos e possuem menor poder computacional, o que traz dificuldade para a implementação desses algoritmos. Diante disso, esta pesquisa busca avaliar os requisitos mínimos de memória e processamento do algoritmo SABER [2], um dos finalistas baseados em reticulados do processo do NIST, assim como de suas versões alternativas, o LightSABER e o FireSABER. Para tanto, foram implementadas e avaliadas técnicas de otimização propostas na literatura, especialmente voltadas à redução do consumo de memória de pilha, recurso mais escasso em dispositivos IoT.

SABER E SUAS VARIAÇÕES

Baseado no problema Module-LWR, o SABER é um mecanismo de encapsulamento de chaves (*Key Encapsulation Mechanism* ou KEM) IND-CCA2 seguro. Quanto ao conceito IND-CCA2 seguro, isso quer dizer que nenhum adversário é capaz de distinguir pares de texto cifrado baseados nas mensagens que ele cifrou, mesmo com acesso às funções de cifração e de decifração do mecanismo [3]. Já o problema Module-LWR, consiste em um problema de reticulados alternativo ao problema do Learning with Errors (LWE), em que as entradas são polinômios (ao invés de inteiros) pertencentes a um anel quociente e os erros são calculados de forma determinística. Por sua vez, o LWE corresponde a resolver equações matriciais, que representam a combinação linear de vetores, acrescida de ruídos (chamados de erros) [4]. A Figura 1 esquematiza o funcionamento do LWE, sendo que a matriz **A** e o vetor **b** formam a chave pública, o vetor **s** corresponde à chave privada, o vetor **e** corresponde aos ruídos e o inteiro *p* define o conjunto sobre o qual serão realizadas as operações.

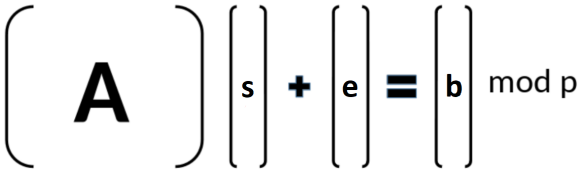

Figura 1. Esquematização do LWE.

O SABER possui as operações de geração de chaves, encapsulamento e desencapsulamento (assim como todo KEM), sendo elas derivadas da versão IND-CPA do SABER (que contém geração de chaves, cifração e decifração), utilizando uma versão pós-quântica da transformada Fujisaki-Okamoto [5]. Uma característica diferencial do SABER é que todas as reduções modulares são feitas sobre potências de dois, para facilitar as operações aritméticas e diminuir a complexidade computacional do algoritmo.

Para possibilitar um nível de flexibilização de acordo com o nível de segurança e de complexidade adequado, foram feitas duas outras versões do SABER: o LightSABER e o FireSABER. Ambas possuem a mesma estrutura que o SABER, mudando apenas alguns parâmetros. 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. Um dos parâmetros de maior impacto no desempenho do algoritmo é o número de polinômios total envolvido nas operações. Por exemplo, a quantidade de polinômios do vetor secreto varia de 2 no LightSABER, para 3 no SABER e 4 no FireSABER, enquanto a matriz pública, para cada versão, possui o quadrado do número de polinômios do vetor secreto correspondente. Sendo assim, o tamanho das chaves e o nível de segurança varia para cada versão: o LightSABER possui chave privada de 1.568 bytes, chave pública de 672 bytes e nível de segurança quântica de 115; o SABER tem chave privada de 2.304 bytes, chave pública de 992 bytes e nível de segurança quântica de 180; por fim, o FireSABER possui chave privada de 3.040 bytes, chave pública de 1.312 bytes e nível de segurança quântica de 245.

OTIMIZAÇÕES PROPOSTAS

Tendo em vista que o SABER utiliza vetores e uma matriz grandes, há um grande consumo de memória para armazenar, acessar e para realizar operações aritméticas entre elas. Por esse motivo, em ambientes computacionais restritos, a dificuldade de implementá-lo é a quantidade de memória consumida pelo algoritmo, o que estimulou o desenvolvimento de diversas otimizações de memória, as quais serviram de base para as que foram implementadas neste trabalho. Mais especificamente, as otimizações visam a redução da quantidade de memória de pilha, memória extra alocada estaticamente durante a execução do programa. Vale ressaltar que a memória de programa não tem sido problema para as implementações, por conta da quantidade disponível mais do que suficiente (até mesmo em ambientes restritos), o que faz com que ela não seja alvo de otimizações

Nas três funções principais do KEM, a multiplicação de polinômios é invocada várias vezes, sendo que ela é a operação que requer o maior consumo de memória de pilha de todo o esquema. Diferente 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, impossibilitando a utilização da forma mais eficiente de se fazer tais operações, a Number Theoretic Transform (NTT) [6]. Por isso, o SABER realiza a multiplicação de polinômios a partir de uma combinação dos algoritmos de Toom-Cook [7] e de Karatsuba [7][8], a alternativa compatível com ele mais eficiente. Porém, esses algoritmos consomem O(n) de memória de pilha - por serem recursivos e não serem *in-place* (isto é, utilizam memória extra) - , recurso bastante limitado em ambientes restritos. Diante disso, em troca de um número maior de ciclos de *clock*, foi proposto por Karmakar et. al [9] uma versão *in-place* do algoritmo de Karatsuba que consome apenas O(log n) de memória extra de pilha.

Adicionalmente, Karmakar et. al propõem otimizações para reduzir o uso de memória na geração da matriz **A** e do vetor secreto **s**, a partir de uma estratégia *just-in-time*. 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. Dessa

forma, elas não ocupam espaço de memória de pilha a mais do que o necessário, evitando um overflow de memória. Como Karmakar, A. et. al só implementaram suas otimizações propostas no SABER, foi necessário adaptá-las aos códigos do LightSABER e do FireSABER.

Analisando otimizações propostas para microcontroladores ARM Cortex-M4 [10], concluímos que, por não dependerem de instruções específicas dessa arquitetura, era possível trazê-las para o Cortex-M0+. Sendo assim, implementamos duas dessas otimizações propostas. A primeira consiste em utilizar apenas 4 bits para codificar os coeficientes dos polinômios do vetor secreto **s**, ao invés de 13 bits. De acordo com a literatura [10], para esquemas que não utilizam o NTT, essa mudança não impacta negativamente a sua segurança, sendo possível economizar memória sem comprometer a segurança e sem modificar drasticamente o desempenho de velocidade. Na verdade, como agora dois coeficientes podem ser representados por um único byte, as funções de empacotamento (responsáveis por tornar mais compacta a representação dos polinômios) ficam até mais simples. Por fim, os autores do artigo também desenvolveram uma versão *in-place* (sem gasto adicional de memória) da verificação do texto cifrado após a decifração, o que reduz o consumo da memória.

IMPLEMENTAÇÃO EM AMBIENTE RESTRITO

O algoritmo com as otimizações descritas foi implementado na placa de desenvolvimento FRDM-KL25Z, que possui 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. Os testes foram todos realizados através da IDE MCUXpresso, desenvolvida pela NXP, que tem uma ferramenta integrada para medição de uso de memória de pilha. Essa IDE possui um compilador embutido GNU Arm Embedded Toolchain 2021.07. Para contagem de ciclos, utilizou-se a interface CMSIS (Cortex Micro-Controller Software Interface and Standard), que conta com recursos para isso.

Com o intuito de determinar a flag mais vantajosa para cada algoritmo, foram feitos testes nas três versões do algoritmo e com diversas flags de otimização, medindo o consumo de memória de pilha e o número de ciclos de CPU. Após análise dos resultados, notou-se que a flag -O1 é a que apresenta o melhor custo-benefício para o LightSABER e que a flag -O2 é a mais adequada para compilação do SABER e do FireSABER. Apesar de ser a flag que mais deveria otimizar a velocidade de execução, a flag -O3 não mostrou resultados tão favoráveis, o que indica que ela deve ser evitada ou utilizada com cautela.

Tabela 1. 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	4.495.576	5.940.149	6.930.342
LightSABER (-O1)	Memória (kB)	3,36 (-19%)	3,46 (-8%)	3,49 (-7%)
	Ciclos	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	7.743.499 (+72%)	9.630.721 (+62%)	10.973.725 (+58%)

A Tabela 1 contém os valores de memória de pilha e de ciclos de *clock* da execução dos três algoritmos, para as respectivas melhores flags de compilação. Analisando esses valores, observa-se que o consumo de memória de pilha se manteve baixo e não variou tanto para as três versões do algoritmo. Porém, o número de ciclos de clock variou bastante entre as versões, o que é esperado devido ao maior número de operações. A avaliação do número de ciclos de clock e do consumo de memória de pilha foi feita separadamente, para cada uma das funções do KEM, visto que alguns

contextos podem realizar uma função com mais frequência do que as demais. Ressalta-se novamente que 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 isso a Tabela 1 destaca esse tipo de memória.

Com o propósito de comparar os resultados obtidos com resultados oficiais, buscaram-se valores de referências no site oficial do algoritmo e no PQM4 [11], sendo este último um *framework* responsável por bibliotecas, avaliação e testes com algoritmos de criptografia pós-quântica. Porém, não foram encontrados dados para as variações do SABER no Cortex-M0, o que impossibilitou a comparação dos resultados das mesmas. Esses dados comparativos estão presentes na Tabela 2 e apontam uma melhora considerável das otimizações feitas neste trabalho, com relação à versão estabelecida na literatura para Cortex-M0 [9], especialmente em memória. Por não terem sido implementadas na versão para Cortex-M0, isso provavelmente está atrelado às otimizações trazidas da implementação para Cortex-M4 [10]. 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. Uma outra mudança foi o aumento da frequência de clock, mas isso não afeta a comparação pois foi medido o número de ciclos de clock por operação.

Tabela 2. Resultados dos melhores testes do SABER comparados com os valores de referência.

Função		SABER em M0 [9]	SABER em M4 *speed/memory* [1] [11]	SABER em M4 *speed* [2] [11]	Este trabalho
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%)

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

Por fim, a Tabela 2 indica que tanto a versão para Cortex-M4 que otimiza velocidade, quanto a que otimiza memória e velocidade, possuem melhor desempenho, especialmente em ciclos de clock, o que é esperado devido ao maior poder computacional dessa arquitetura. Mesmo assim, os resultados deste trabalho são importantes pois tornam a execução em Cortex-M0+ mais eficiente e mais econômica, chegando a consumir 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.

CONCLUSÕES E TRABALHOS FUTUROS

Com a implementação e adaptação de otimizações no SABER (e suas variações) propostas na literatura, conseguimos torná-los mais eficientes na plataforma ARM Cortex-M0+, com a versão mais segura (IND-CCA2) desses algoritmos. Outro aspecto importante, principalmente para diminuir o número de ciclos de clock, foi o estudo das flags de otimização de compilação, que mostrou que a flag -O3 não produziu os melhores resultados. Mais especificamente, observamos que a flag -O1 foi a mais adequada para o LightSABER, enquanto a flag -O2 foi a melhor para o SABER e para o FireSABER.

Com relação à versão de segurança média do algoritmo, conseguimos reduzir de 6% a 8% o número de ciclos de clock e de 18% a 39% o consumo de memória, comparando com os resultados disponíveis para Cortex-M0 na literatura [9]. Considerando as diferenças entre o SABER e suas

variações, é esperado que as reduções sejam similares e proporcionais nas variações. Apesar de a implementação em Cortex-M0+ apresentar um pior desempenho do que a implementação em Cortex-M4 (o que é esperado dado o poder computacional superior do Cortex-M4), destacamos que o consumo de memória no Cortex-M0+ é similar ao da versão em Cortex-M4 otimizado para memória, chegando a ser inferior quando a implementação em Cortex-M4 está otimizada para velocidade.

Propomos, como trabalhos futuros, a otimização mais detalhada da versão do algoritmo desenvolvida neste trabalho em nível de linguagem assembly, o que poderia produzir resultados ainda melhores com relação aos ciclos de clock. Além disso, dando continuidade ao que fizemos, podem ser pesquisadas e trazidas para o Cortex-M0 otimizações feitas para outras plataformas, com o intuito de se aproximar ainda mais dos requisitos mínimos de memória e processamento. Dessa vez, poderiam ser trazidas otimizações de velocidade que não aumentem significativamente o uso de memória, visto que o consumo da mesma já foi bastante reduzido.

REFERÊNCIAS

[1] 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

[2] 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

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

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

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

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

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

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

[9] 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.

[10] 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.

[11] 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.