Implementação Paralela do Algoritmo de Geração de Chaves do Esquema de Assinatura Digital XMSS (eXtended Merkle Signature Scheme)

Thales Vilas Bôas Araújo[1], **Jairo Panetta**[1]

[1]Divisão de Ciência da Computação (IEC)
Instituto Tecnológico de Aeronáutica (ITA)
Praça Marechal Eduardo Gomes, 50, Vila das Acácias – 12.228-900
São José dos Campos – SP – Brazil

thalestvba@gmail.com, panetta@gmail.com

Abstract. *Hash Based Signature Schemes are gaining attention due to their believed quantum resistance. Their signing and verification times are comparable to those of algorithms in use today, but their key's generation time is much greater. To speed-up the execution time of key generation algorithms, this paper introduces and analises two parallel MIMD implementations of the hash based signature scheme XMSS (eXtended Merkle Signature Scheme).*

Resumo. *Algoritmos de assinatura baseados em hash estão ganhando atenção devido a sua apregoada resistência a algoritmos quânticos. Apesar dos tempos de confecção e verificação de assinatura serem compatíveis com os tempos dos algoritmos atualmente em uso, o tempo de geração das chaves é ordens de grandeza superior. Este artigo apresenta e analisa o desempenho de duas implementações de paralelismo MIMD (Multiple Instruction Multiple Data) para acelerar o tempo de execução do algoritmo de geração de chaves do esquema de assinatura digital baseado em hash XMSS (eXtended Merkle Signature Scheme).*

1. Introdução

Esquemas de assinatura digital baseados em hash estão ganhando atenção devido a sua apregoada resistência a algoritmos quânticos [Hülsing et al. 2015a]. Dentre tais esquemas está o eXtendend Merkle Signature Scheme (XMSS), apresentado em 2011 [Buchmann et al. 2011] e que em 2018 teve sua documentação de implementação descrita pelo Internet Research Task Force (IRTF) [Hülsing et al. 2018]. Há algoritmos desse tipo participando da recente concorrência do National Institute for Standards and Tecnology (NIST) para padronização de esquemas resistentes a algoritmos quânticos [NIST 2016].

A segurança de um esquema de assinatura baseado em hash é garantida por um requisito simples: basta que a função de hash utilizada seja segura [Butin 2017]. O funcionamento do esquema é independente das características internas da função de hash escolhida, portanto tal função pode ser alterada sem a necessidade de modificações nos demais componentes do esquema de assinatura.

Diferente dos esquemas de assinatura digital tradicionais, em que uma única chave secreta é utilizada para confeccionar a assinatura de todas as mensagens, nos esquemas

de assinatura baseados em hash cada chave secreta só pode ser utilizada para assinar uma única mensagem. O usuário deve definir de antemão a quantidade de chaves secretas que deseja, confeccionar todas as chaves e então computar sua chave pública, que o identifica e autentica todas as assinaturas geradas com as chaves secretas.

Apesar dos tempos de confecção e verificação da assinatura dos esquemas de assinaturas baseados em hash serem compatíveis com os tempos dos algoritmos em uso atualmente, o tempo necessário para gerar a chave pública é muito maior. O tempo necessário para gerar $2^{20} = 1.048.576$ chaves de assinatura e calcular a chave pública correspondente do algoritmo XMSS é da ordem de milhares de segundos, enquanto para o algoritmo ECDSA o tempo é da ordem de milissegundos.

A computação de todas chaves é inviável em dispositivos que possuam baixo poder computacional, como sistemas embarcados e sensores [Pereira et al. 2016]. Nesse caso é mais eficiente que tais chaves sejam computadas a priori em um computador que possua maior capacidade computacional e em seguida inseridas no dispositivo em questão.

A otimização de código para dispositivos embarcados foi estudada em [Hülsing et al. 2015b], [Pereira 2015] e [Wang et al. 2018], dentre outros. Há também otimizações em relação ao tamanho da assinatura, como em [Zheng et al. 2018]. A aplicação de paralelismo em esquemas de assinatura baseados em hash foi tema do trabalho [de Oliveira and López 2015], que utilizou instruções vetoriais presentes em processadores Intel com microarquitetura Haswell para reduzir o tempo de execução dos algoritmos de geração de chaves, confecção e verificação de assinatura de esquemas de assinatura baseados em hash. Mais recentemente, o artigo [de Oliveira et al. 2017] expandiu essa implementação, utilizando também instruções vetoriais presentes na microarquitetura Skylake.

Esse artigo apresenta duas implementações de paralelismo MIMD (Multiple Instruction Multiple Data) utilizando diretivas OpenMP para reduzir o tempo necessário para a geração da chave pública do algoritmo XMSS. A seção 2 apresenta os conceitos iniciais necessários para a compreensão do esquema XMSS, o qual é descrito na seção 3. As implementações paralelas do algoritmo de geração da chave pública são apresentadas na seção 4. A seção 5 contém os resultados e a discussão dos experimentos. A conclusão é apresentada na seção 6.

2. Conceitos Iniciais

2.1. Funções de Hash

Uma função de hash é um mapeamento calculado de maneira eficiente, ou seja, em tempo polinomial em relação ao tamanho da saída, n, tal que $h : \{0,1\}^* \rightarrow \{0,1\}^n$ com $n \in \mathbb{N}$, também representado por $y = h(x)$. Além disso, há outras outras características necessárias para funções de hash utilizadas em criptografia.

A primeira característica é denominada resistência à pre-imagem: dado qualquer valor de saída y da função h, é difícil encontrar um valor de entrada x tal que $y = h(x)$. A segunda característica é denominada resistência à segunda pré-imagem: dado qualquer valor de entrada x da função h, é difícil encontrar um valor de entrada x' tal que $x' \neq x$ e $h(x') = h(x)$. A terceira característica é a resistência à colisões. Uma colisão ocorre

quando dois valores de entrada distintos, x' e x'' geram o mesmo valor de saída, ou seja: $h(x') = h(x'')$ e $x' \neq x''$. Como funções de hash têm o conjunto domínio maior do que o conjunto imagem, o mapeamento não pode ser injetor e portanto colisões são inevitáveis. A propriedade de resistência à colisão avalia o quão difícil é encontrar dois valores quaisquer que produzem uma colisão.

Difícil nos casos acima significa que um algoritmo probabilístico de tempo polinomial têm probabilidade de sucesso irrisória. Probabilidade irrisória significa que a chance de sucesso é menor que qualquer fração polinomial para determinado parâmetro, ou seja: $Pr[A] < \mathcal{O}(1/n^c)$ e $n, c \in \mathbb{N}$ e $c > 0$ [Bellare 2002].

Considerando um adversário que possui apenas um computador clássico, é possível obter pré-imagens realizando $\mathcal{O}(2^n)$ avaliações da função de hash. Utilizando algoritmos quânticos, a complexidade da busca é reduzida para $\mathcal{O}(2^{n/2})$ pelo algoritmo de Groover [Grover 1996].

O ataque de Yugal [Yuval 1979] possibilita encontrar colisões com probabilidade de 50% ao serem avaliados $\mathcal{O}(2^{n/2})$ valores de entrada da função de hash. Utilizando algoritmos quânticos, é possível fazê-lo com complexidade $\mathcal{O}(2^{n/3})$ [Brassard et al. 1998].

Tais valores representam o limite superior da complexidade de algoritmos genéricos que podem ser aplicados para encontrar pré-imagens e colisões em qualquer função de hash. Funções de hash específicas podem ter vulnerabilidades que resultem em algoritmos com menor complexidade. Até o momento existem algoritmos capazes de produzir colisões para as funções de hash MD-5 [Wang et al. 2004] e SHA-1 [Stevens et al. 2017] em tempo menor que $\mathcal{O}(2^n)$, entretanto não há ataques para obter pré imagens para tais funções de hash que possuam complexidade menor que algoritmos genéricos genéricos.

Em funções de hash utilizadas em criptografia, a resistência a colisão implica resistência a segunda pré imagem, entretanto o inverso não é verdadeiro [Stallings 2014].

Uma família de funções de hash, F_n, é um conjunto de funções de hash f_k em que cada valor k define um mapeamento distinto dos demais, tal que $F_n = \{f_k : \{0,1\}^* \rightarrow \{0,1\}^n | k \in \{0,1\}^n\}$, conforme definido em [Buchmann et al. 2011].

2.2. Assinaturas Baseadas em Hash

Um dos primeiros esquemas de assinatura que utiliza apenas funções de hash foi proposto em [Lamport 1979]. Diferentemente dos esquemas existentes até então, em que a assinatura era confeccionada para a mensagem como um todo, o esquema de Lamport consiste em assinar cada um dos bits da mensagem de maneira individual e independente.

O esquema consiste em escolher aleatoriamente duas chaves de assinatura para cada bit da mensagem a ser assinada, uma para cada possível valor do bit (0 ou 1). As chaves de verificação são calculadas aplicando uma função de hash f em cada uma das chaves de assinatura. Para confeccionar uma assinatura, o signatário percorre os bits da mensagem e seleciona a chave correspondente ao valor do bit. O conjunto de chaves selecionadas forma a assinatura da mensagem. Para verificar essa assinatura, o destinatário da mensagem aplica a função f nas chaves recebidas na assinatura. O resultado obtido é comparando com a chave de verificação armazenada em uma entidade terceira confiável. Caso todos os valores obtidos pelo destinatário forem iguais aos valores da chave de

verificação a assinatura é aceita. Caso isso não ocorra em um ou mais bits da mensagem, a assinatura é recusada.

Cada conjunto de chaves de assinatura só pode ser utilizado uma única vez, pois parte da informação secreta do esquema, a chave de assinatura, é revelada no processo. Por isso, tal esquema é denominado assinatura de única vez - OTS (One Time Signature).

No esquema de Winternitz (WOTS - Winternitz One Time Signatrue) [Merkle 1979] são assinados conjuntos de bits de tamanho $\log_2 w$. O valor w é denominado parâmetro de Winternitz. O conjunto de bits é denominado *bloco*. Para cada bloco é escolhida aleatoriamente uma chave de assinatura inicial, x_i. Para cada um dos w valores possíveis desse bloco, a chave utilizada para assinar o bloco é calculada aplicando uma função de hash f sucessivas vezes na chave inicial x_i. A quantidade de aplicações sucessivas de f equivale ao valor do bloco interpretado como um inteiro positivo. Seja M_i o valor do i-ésimo bloco e σ_i o valor da assinatura do mesmo, então $\sigma_i = f^{M_i}(x_i)$, em que o expoente representa a quantidade de aplicações sucessivas da função. A chave de verificação do i-ésimo bloco, representado por y_i, corresponde à chave de assinatura do bloco com maior valor possível, ou seja $y_i = f^{w-1}(x_i)$. Seja m o tamanho da mensagem, então a quantidade de chaves de assinatura necessária para a mensagem é dada por $l_1 = \lceil \frac{m}{\log_2 w} \rceil$.

O esquema WOTS necessita de um valor de conferência, C, para impedir que a mensagem e a assinatura sejam alteradas. Tal valor é calculado por $C = \sum_{i=1}^{l_1}(w-1-M_i)$ e também é assinado da mesma forma que a mensagem. A quantidade de chaves de assinatura para o valor de conferência é $l_2 = \lfloor \frac{\log_2(l_1(w-1))}{\log_2 w} \rfloor$. A quantidade de chaves necessárias no esquema é dada por $l = l_1 + l_2$.

Assim como a assinatura Lamport, cada chave de assinatura só pode ser utilizada uma única vez (OTS).

2.3. Árvore de Merkle

Merkle [Merkle 1979] propôs utilizar árvores binárias e funções de hash para possibilitar a autenticação de várias chaves OTS e a identificação do usuário utilizando apenas uma chave pública. Dessa forma, o usuário gera um determinado número de chaves OTS de assinatura, representadas por X, e de verificação, representadas Y, e em seguida computa a chave pública a partir das chaves de verificação.

Seja $Y = \{y_i\}$ um conjunto de chaves de verificação para um OTS. O primeiro passo do algoritmo de Merkle é comprimir as chaves de verificação em apenas uma chave. Isso é feito concatenando as chaves de verificação e submetendo o resultado a uma função de hash. Seja L_i a compressão da chave de verificação do i-ésimo OTS e f uma função de hash, então $L_i = f(y_1||y_2||...||y_l)$.

O resultado dessa operação será armazenado nas folhas da árvore de Merkle, conforme a figura 1. Para calcular o nível acima das folhas da árvore, dados n_p o nó pai, n_e e n_d respectivamente os filhos direito e esquerdo, então $n_p = f(n_e||n_d)$, $f : \{0,1\}^{2n} \rightarrow \{0,1\}^n$. Esse processo prossegue até que seja computada a raiz da árvore, que é a chave pública responsável por autenticar todas as chaves OTS que participaram do processo.

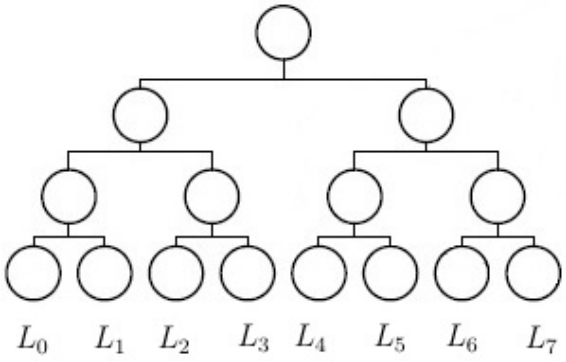

Figura 1. Árvore de Merkle com 8 OTS (adaptado de [Buchmann et al. 2011])

3. eXtended Merkle Signature Scheme - XMSS

O esquema de assinatura XMSS [Buchmann et al. 2011] utiliza uma adaptação do esquema WOTS como esquema OTS, uma árvore de Merkle para computar a chave pública e uma árvore binária para computar as folhas da árvore de Merkle a partir das chaves de verificação de cada OTS. A figura 2 apresenta esses elementos do esquema XMSS.

3.1. Geração de Chaves de Assinatura

Em cada OTS, as chaves de assinatura devem ser independentes. Uma forma de obtê-las é selecioná-las de maneira aleatória em uma distribuição uniforme. Entretanto nesse caso seria necessário armazenar todas as chaves.

Uma forma de reduzir a quantidade de chaves armazenadas é gerar de maneira aleatória apenas um valor inicial, denominado *semente*, e utilizá-lo com entrada para uma função pseudo aleatória, PRF, que computa os valores da chave de assinatura. A função pseudo aleatória deve ser tal que o conjunto de chaves geradas seja indistinguível de um conjunto de valores escolhidos de maneira aleatória em uma distribuição normal.

Na implementação, cada OTS possui a respectiva semente. Em cada OTS, as chaves de assinatura são calculadas utilizando a semente e o índice da chave de assinatura que está sendo gerada. Seja l a quantidade de chaves de assinatura e x_i a i-ésima chave de assinatura, então $x_i = PRF(i, semente)$ para $i = (1, ..., l)$. Na prática os elementos de entrada são concatenados e submetidos à função SHA-2: $x_i = PRF(i, semente) =$ SHA-2$(i||semente)$.

3.2. WOTS+

[Hevia and Micciancio 2002] e [Dods et al. 2005] avaliaram a segurança dos esquemas de assinatura baseados em hash e concluíram que a função de hash utilizada nos esquemas deve ser resistente à colisões, resultando em segurança $\mathcal{O}(2^{n/2})$. Isso implica que o tamanho de saída da função de hash deve ser o dobro do parâmetro de segurança que se deseja atingir. Outra opção seria a utilização de uma família de funções pseudo aleatórias, o que resulta em segurança $\mathcal{O}(2^n)$.

O esquema de assinatura WTOS+ [Hülsing 2017] atinge fator de segurança $\mathcal{O}(2^n)$ utilizando uma família de funções de hash F_n que possui resistência a segunda pré imagem. Esse tipo de função pode ser construída utilizando apenas funções de mão única

[Rompel 1990] e são condição necessária e suficiente para a existência de esquemas de assinatura digital [Naor and Yung 1989]. Para eliminar a necessidade de resistência a colisão, cada execução de uma função $f_k \in F_n$ é aleatorizada a partir de um conjunto de valores aleatórios $r = (r_1, r_2, ..., r_w)$. O padrão adotado em [Hülsing et al. 2018] utiliza o esquema WOTS+, em detrimento daquele previsto em [Buchmann et al. 2011].

A função utilizada para obter a assinatura e a chave de verificação é dada por $f_k^i(x, r) = f_k(f_k^{i-1}(x, r) \oplus r_i)$ e $f_k^0(x, r) = x$, em que i representa a quantidade de aplicações da função, x representa a chave de assinatura inicial do conjunto de bits, $r = (r_1, r_2, ..., r_w)$ representa os valores utilizados para aleatorizar os valores de entrada da função f_k tais que $r_i \in \{0,1\}^n$, e $f_k : \{0,1\}^n \to \{0,1\}^n, k \in \{0,1\}^n$.

3.3. Árvore-L e Árvore de Merkle

Na árvore de Merkle as folhas de cada OTS são obtidas pela concatenação das chaves de verificação, que em seguida são submetidas a uma função de hash. O esquema XMSS utiliza uma árvore binária denominada árvore-L para obter as folhas da árvore de Merkle de cada OTS.

Tal árvore é semelhante a uma árvore de Merkle, com a diferença que as folhas são as chaves de verificação do esquema WOTS+. A maneira de calcular os níveis superiores da árvore é a mesma da árvore Merkle original: o nó pai é obtido concatenando os nós filhos e submetendo o resultado a uma função de hash. Como no esquema WOTS+, é utilizada uma família de funções de hash com resistência à segunda pré imagem, G_n, e cada chamada de uma função g_k é aleatorizada por um conjunto de valores r. Tal construção foi inicialmente proposta em [Bellare and Rogaway 1997], e evita a necessidade de resistência à colisões, que resultaria na mesma diferença de segurança do esquema que no caso WOTS+.

Sejam n_p, n_e, n_d respectivamente os nós pai, filho esquerdo e filho direito, $g_k \in G_n | g_k : \{0,1\}^{2n} \to \{0,1\}^n$, então $n_p(n_e, n_d, r) = g_k(n_e \oplus r_e || n_d \oplus r_d)$, em que os valores de r são utilizados para aleatorizar cada chamada da função g_k, $r_i \in \{0,1\}^n$.

Como o número de chaves de assinatura de cada OTS não necessariamente é uma potência de 2, a árvore não é binária completa, e por isso é denominada árvore-L. Um nó que não possui "irmão direito", ou seja, não pode ser submetido ao processo descrito acima, é elevado para níveis superiores até que possua um irmão esquerdo e é possível computar seu nó pai.

A raiz de cada árvore-L corresponde à uma folha da árvore Merkle. A quantidade de chaves OTS, e portanto a quantidade de folhas da árvore de Merkle, é dada por 2^h, em que h é a altura da árvore de Merkle. Assim como na árvore-L, é utilizada uma família de funções de hash G_n, e cada chamada de uma função $g_k \in G_n$ é aleatorizada por um valor conjunto de valores $r = (r_1, r_2...)$. Isso também é feito para que não seja necessário utilizar uma função de hash resistente à colisões. A raiz da árvore Merkle é a chave pública do usuário e será utilizada para identificá-lo e autenticar todas suas chaves OTS.

4. Implementação Paralela

As implementações paralelas do algoritmo de geração de chaves foram realizadas por meio de diretivas OpenMP em código escrito pelos autores baseado no RFC-8391

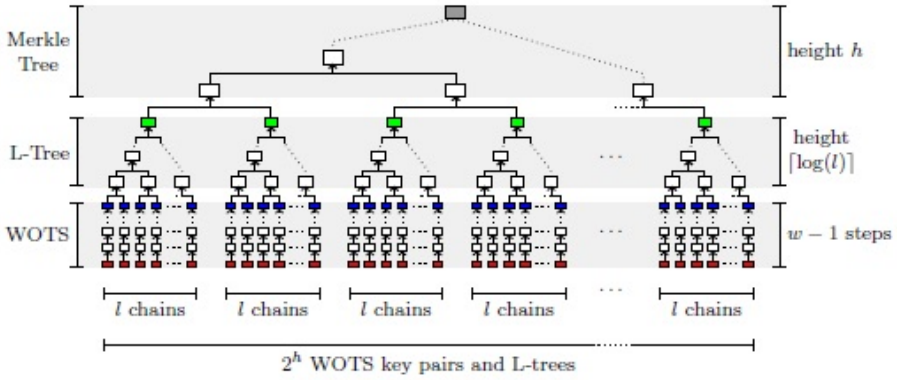

Figura 2. Esquema XMSS [Wang et al. 2018]

[Hülsing et al. 2018]. A função de hash SHA-2 utilizada é proveniente da biblioteca OpenSSL.

A primeira fase do algoritmo de geração da chave pública é a obtenção da semente em cada OTS. Posteriormente é realizada a expansão nos l valores de chave de assinatura, dado por $x_i = PRF(i, semente)$ com $i = 1, ...l$. Como os cálculos de x_i são independentes entre si, esta primeira fase pode ser executada em paralelo.

Dentro de cada OTS a chave de verificação só depende da chave de assinatura inicial. Como as chaves iniciais são independentes entre si, a aplicação da função de hash w vezes em cada chave pode ser feita em paralelo.

Na árvore-L de cada OTS, cada nó do nível $\alpha + 1$ da árvore depende de seus dois filhos do nível α. Todos os nós do mesmo nível são independentes entre si e podem ser calculados em paralelo se os nós do nível inferior já estiverem sido computados. A ideia da implementação paralela é a mesma adotada para somar em paralelo os elementos de um vetor [Daniel and Steele 1986], algoritmo paralelo conhecido como “cascade sum” . A mesma ideia é aplicada para implementar paralelismo na árvore de Merkle. Na árvore-L, os nós “excedentes” se comparados a uma árvore binária completa são computados após o cálculo em paralelo dos demais nós desse nível.

A primeira implementação paralela consiste em utilizar os entes paralelos, threads nesse caso, dentro de cada OTS. Tal caso será denominado “Paralelismo Winternitz”, referenciado também por P_W, pois o paralelismo está entre cada conjunto de w bits. Uma thread é responsável pela geração da chave de assinatura de um determinado índice e em seguida computa sua chave de verificação. Após isso, a thread prossegue para outro índice, caso exista alguma chave ainda não computada. Quando todas as chaves de verificação estão calculadas, as threads realizam a computação da árvore-L conforme descrito anteriormente. Após terminarem a árvore-L do respectivo OTS, as threads prosseguem para o próximo OTS, realizando o mesmo procedimento.

A segunda implementação paralela do algoritmo consiste em cada thread realizar a computação de um OTS, desde a computação das chaves de assinatura a partir da semente até a raiz da árvore-L daquele OTS. Esse caso será denominado ‘’Paralelismo OTS”, referenciado por P_O.

Em ambos casos a árvore de Merkle é computada após a computação completa

dos 2^h OTS.

4.1. Parâmetros Utilizados

Os parâmetros a serem definidos para instanciar um esquema XMSS [Buchmann et al. 2011] são: o parâmetro de segurança $n \in \mathbb{N}$, o parâmetro de Winternitz $w \in \mathbb{N}$ e $w > 1$, o comprimento em bits da mensagem a ser assinada $m \in \mathbb{N}$ com $m > 1$, as famílias de funções F_n e G_n e a altura da árvore de Merkle h, que define a quantidade de OTS, dada por 2^h.

O RFC-8391 [Hülsing et al. 2018] apresenta um conjunto de parâmetros tidos como necessários: $n = 256$, $w = 16$, $m = 256$, F_n e G_n= SHA-2 256, com opção de $h = 10, 16$ e 20. O valor de l, a quantidade de conjuntos de bits em cada OTS, obtido a partir de tais parâmetros é 67. Os experimentos foram realizados com esses parâmetros, e $h = 20$. Para implementar $f_k \in F_n$ e $g_k \in G_n$ com SHA-2 a chave k é concatenada ao(s) valor(es) de entrada da função: $f_k(x) = \text{SHA-2}(k||x)$ e $g_k(a, b) = \text{SHA-2}(k||a||b)$.

5. Experimentos, Resultados e Avaliação

Realizamos um conjunto de experimentos para avaliar o impacto das duas formas de paralelismo no tempo de execução do algoritmo de geração de chaves. Todos os experimentos trataram o mesmo problema, definido pelos parâmetros reportados na seção 4.1. Cada experimento consistiu em executar uma implementação paralela utilizando uma quantidade de threads. Medimos o tempo de execução (*wall time*) de cada experimento. Repetimos cada experimento dez vezes e reportamos o tempo médio das dez execuções.

Os experimentos foram realizados em um nó computacional do Parque Computacional de Alto Desempenho (PCAD) da Universidade Federal do Rio Grande do Sul (URFGS), contendo dois processadores Intel Xeon E5-2699 v4 de 2,2 GHz e 256 GB de RAM. Os compiladores utilizados foram o GNU Compiler Collection (GCC) versão 7.3.0-27 e o Intel C++ Compiler (ICC) versão 19.0.3.199. Em ambos compiladores foram utilizadas as chaves de compilação -O3 e -march=broadwell.

O nó computacional utilizado possui 44 núcleos físicos. Quando a quantidade de threads utilizada excede o número de núcleos físicos disponíveis, a tecnologia Hyper-Threading da Intel aproveita ciclos ociosos na execução de uma thread para executar outra thread no mesmo núcleo físico. Com essa tecnologia, o nó computacional passa a ter 88 núcleos, sendo 44 físicos e 44 virtuais. Os núcleos virtuais são utilizados apenas quando todos os núcleos físicos já estão em uso.

A figura 3 reporta o tempo de execução das duas implementações paralelas em função do número de threads e do compilador utilizado. A diferença no tempo de execução de P_O utilizando GCC e ICC foi irrisória, portanto apenas o tempo de execução com GCC será apresentado nos gráficos.

É possível verificar a partir da figura 3 que o paralelismo traz ganhos consideráveis pela redução do tempo de execução, mas não é possível quantificar esse ganho. Uma métrica clássica para quantificar ganhos de paralelismo é o *speed-up*, definido como a razão entre o tempo de execução sequencial e o tempo de execução utilizando p threads, ou seja, $S_p = t_1/t_p$, em que t_p é o tempo de execução com p threads. O tempo de execução sequencial utilizado foi o menor obtido entre os compiladores avaliados. Outra métrica clássica de paralelismo é a *eficiência*, definida por $E_p = S_p/p$, ou seja, quão

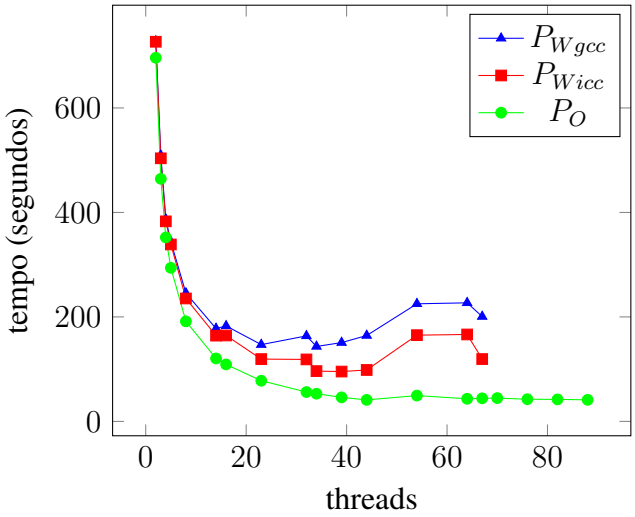

Figura 3. Tempo de Execução em Função do Número de Threads

eficientemente as p threads são utilizadas. A figura 4 reporta o speed-up (gráfico a esquerda) e a eficiência (gráfico a direita) de P_O e P_W em função do número de threads e dos compiladores utilizados.

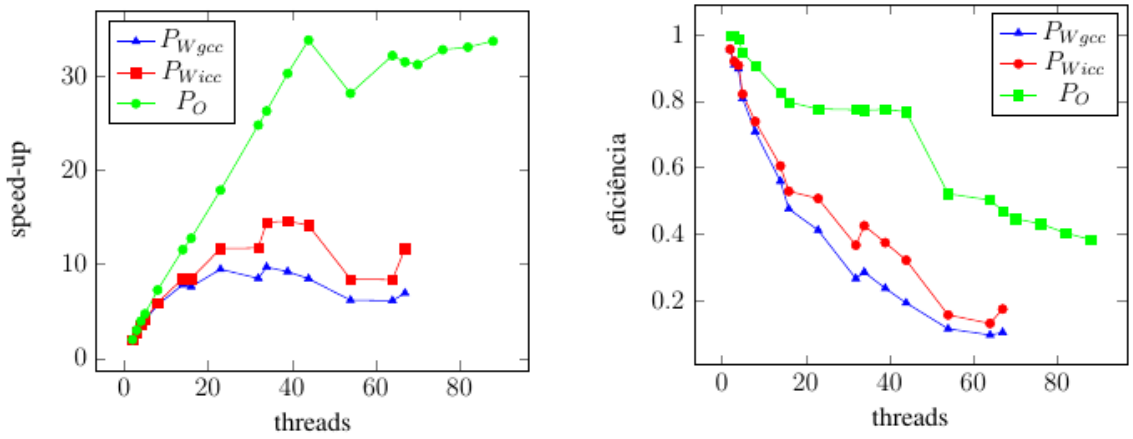

Figura 4. Speed up (esq) e eficiência (dir) em Função do Número de Threads

5.1. Paralelismo Winternitz

Durante a execução em paralelo da árvore-L no Paralelismo Winternitz, à medida que a árvore chega a níveis mais altos, há menos nós a serem computados e portanto a quantidade de threads ociosas aumenta durante a execução do algoritmo. Quando há aumento na quantidade de threads, os níveis iniciais são calculados de maneira mais rápida, entretanto o nível em que estas threads ficaram ociosas na árvore-L é mais baixo, resultando no decréscimo de eficiência apontado pelo gráfico à direita na figura 4, sem o estabelecimento de um patamar em que a eficiência permanece constante.

Em termos de complexidade paralela, o tempo de execução da árvore é $\mathcal{O}(\log n)$. Esse valor é atingido quando cada nível é executado em apenas uma execução paralela, ou seja, com 32 threads. Qualquer quantidade de threads acima desse valor resulta em

threads ociosas já no primeiro nível de execução. Ainda assim observamos que tais casos proporcionam aceleração do algoritmo.

O menor tempo de execução (figura 3) foi de 95.38 segundos, obtido com 39 threads e compilador ICC, resultando em um speed-up (à esquerda na figura 4) de 14.56. Utilizando GCC o menor tempo de execução foi de 143 segundos, obtido com 34 threads, speed-up de 9.68. Tal diferença ocorreu porque o ICC identificou melhores oportunidades de otimização do código em relação ao compilador GCC. A partir desses picos de desempenho, o tempo de execução do código gerado por ambos compiladores apresentou pequeno aumento enquanto as threads permaneceram na faixa de nós físicos, caracterizando um patamar no tempo de execução e speed-up.

Ao utilizar mais de 44 threads o tempo de execução aumenta consideravelmente. De 64 para 67 threads há uma nova redução no tempo de execução, mas não a ponto de retornar ao patamar do melhor tempo de execução.

A eficiência decrescente (à direita na figura 4) apresenta um ponto de inflexão com uso de 34 threads em ambos compiladores. Após o esgotamento dos núcleos físicos a queda na eficiência é ainda mais acentuada.

Com o Paralelismo Winternitz não há razão para utilizar mais que 67 threads para os parâmetros em questão, pois tal valor equivale à quantidade de chaves em cada OTS. Dessa forma, utilizar mais do que 67 threads resultaria apenas em threads ociosas já no compto da chave de assinatura, resultando em tempo de execução devido ao gerenciamento das threads. Entretanto, foi verificado na prática que a limitação começa em uma quantidade de threads ainda menor.

O compilador ICC gerou código mais eficiente que o compilador GCC em todos os experimentos realizados. No caso de utilização de apenas 2 threads a diferença foi irrisória, mas a diferença foi acima de 20% em todos os casos em que foram utilizadas 23 ou mais threads. A maior diferença foi de 68%, utilizando 67 threads.

5.2. Paralelismo OTS

O Paralelismo OTS teve desempenho e eficiência melhores que o Paralelismo Winternitz em todos os casos avaliados. Sua implementação não é afetada pelas limitações da árvore-L presentes em P_W. O tempo de execução obtido com o código de ambos compiladores apresentou diferenças irrisórias, a maior delas de 3%. Portanto analisaremos P_O considerando a compilação GCC, que obteve melhor desempenho na maioria dos experimentos.

O speed-up cresceu linearmente durante a faixa de utilização de núcleos físicos. A eficiência nessa faixa estabiliza em aproximadamente 78%. O menor tempo de execução foi obtido com a quantidade máxima de threads utilizando somente núcleos físicos, 44 threads: 41.15 segundos, speed-up de aproximadamente 33.75 e eficiência de 76.7%.

Na faixa de utilização de núcleos virtuais ocorreu inicialmente aumento considerável do tempo de execução, o que resultou em uma queda acentuada na eficiência. Após 67 threads ocorre o estabelecimento de um patamar de eficiência entre 46% e 40%. Isso permite que o tempo de execução se aproxime daquele registrado com 44 threads. Utilizando 76, 82 ou 88 threads a diferença para o menor tempo de todos os experimentos é menor que 1 segundo, sendo que no último caso é de apenas 0.12 segundo.

Utilizando de 76 a 88 threads ocorre uma nova queda na eficiência, indicando que os núcleos físicos e virtuais estão próximos ao esgotamento de sua capacidade, o que potencialmente resultaria em uma queda acentuada de desempenho caso fosse possível adicionar ainda mais threads.

6. Conclusão

Os esquemas de assinatura baseados em hash possuem sua segurança baseada em funções largamente estudadas e compreendidas [Gorbenko et al. 2018]. Entretanto não é adequado empregá-los em todos os casos de uso de assinatura digital [Butin et al. 2017]. O tamanho das chaves os torna impraticáveis para aplicações em que a mensagem possui tamanho próximo ou até mesmo menor que o tamanho da assinatura.

Em aplicações específicas, a simplicidade de implementação e a velocidade de operação tornam os esquemas baseados em hash bastante úteis. Outro fator importante é que a segurança desses esquemas já está bem caracterizada, enquanto os demais esquemas resistentes a adversários que possuam computadores quânticos ainda estão em desenvolvimento.

Os dispositivos que mais se beneficiarão das características das assinaturas baseadas em hash têm pouca capacidade computacional, de forma que é inviável gerar uma grande quantidade de chaves em tais dispositivos, sendo mais eficiente realizar tal operação em recursos que possuam maior capacidade computacional.

Esse artigo mostrou que a utilização de paralelismo MIMD com diretivas OpenMP é capaz de reduzir o tempo de geração das chaves em até quase 34 vezes, resultando em tempo de execução inferior a 1 minuto para gerar uma quantidade de chaves superior a 1 milhão.

Os experimentos demonstraram que a utilização de paralelismo requer avaliação minuciosa da quantidade de threads a serem empregadas, pois foi verificado que em algumas faixas o aumento dos entes paralelos não resultou em redução do tempo de execução. A eficiência do código, considerando apenas tempo de execução e quantidade de threads empregadas, foi consideravelmente maior sem a utilização de threads virtuais. A utilização de Hyper Threading não possibilitou melhora no tempo de execução do algoritmo, apenas conseguiu equiparar o melhor desempenho sem a utilização de tal tecnologia.

Outro fator que que deve ser avaliado na implementação de paralelismo é a capacidade de aproveitamento dos recursos proporcionados pela arquitetura. O Paralelismo OTS aproveita de maneira eficiente a grande quantidade de entes paralelos disponíveis na arquitetura e no nó computacional utilizados. O mesmo não ocorre utilizando o Paralelismo Winternitz. Essa diferença teve impacto direto na redução do tempo de execução, sendo patente que o Paralelismo OTS apresentou melhor desempenho do que o Paralelismo Winternitz nos experimentos realizados.

Outros esquemas de assinatura baseados em hash que também possuam vários OTS combinados através de árvores de Merkle, como a variação multi-árvore do algoritmo XMSS, denominado XMSS^{MT} [Hülsing et al. 2017], e o esquema SPHINCS [Bernstein et al. 2015] podem se beneficiar dessa forma de implementação de paralelismo, mesmo que a quantidade de árvores seja diferente.

Referências

Bellare, M. (2002). A note on negligible functions. *J. Cryptology*, 15(4):271–284.

Bellare, M. and Rogaway, P. (1997). Collision-resistant hashing: Towards making uowhfs practical. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 470–484.

Bernstein, D. J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., and Wilcox-O'Hearn, Z. (2015). SPHINCS: practical stateless hash-based signatures. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, pages 368–397.

Brassard, G., Høyer, P., and Tapp, A. (1998). Quantum cryptanalysis of hash and claw-free functions. In *LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings*, pages 163–169.

Buchmann, J. A., Dahmen, E., and Hülsing, A. (2011). XMSS - A practical forward secure signature scheme based on minimal security assumptions. *IACR Cryptology ePrint Archive*, 2011:484.

Butin, D. (2017). Hash-based signatures: State of play. *IEEE Security & Privacy*, 15(4):37–43.

Butin, D., Walde, J., and Buchmann, J. A. (2017). Post-quantum authentication in openssl with hash-based signatures. In *Tenth International Conference on Mobile Computing and Ubiquitous Network, ICMU 2017, Toyama, Japan, October 3-5, 2017*, pages 1–6.

Daniel, H. W. and Steele, Jr., G. L. (1986). Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183.

de Oliveira, A. K. D. S. and López, J. (2015). An efficient software implementation of the hash-based signature scheme MSS and its variants. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, pages 366–383.

de Oliveira, A. K. D. S., Lopez, J., and Cabral, R. (2017). High performance of hash-based signature schemes. *International Journal of Advanced Computer Science and Applications*, 8:421–432.

Dods, C., Smart, N. P., and Stam, M. (2005). Hash based digital signature schemes. In *Cryptography and Coding, 10th IMA International Conference, Cirencester, UK, December 19-21, 2005, Proceedings*, pages 96–115.

Gorbenko, Y. I., Melnik, T. V., and Gorbenko, I. D. (2018). Analysis of potential post-quantum schemes of hash-based digital signatures. *Telecommunications and Radio Engineering*, 77:603–626.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219.

Hevia, A. and Micciancio, D. (2002). The provable security of graph-based one-time signatures and extensions to algebraic signature schemes. In *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, pages 379–396.

Hülsing, A. (2017). WOTS+ - shorter signatures for hash-based signature schemes. *IACR Cryptology ePrint Archive*, 2017:965.

Hülsing, A., Butin, D., Gazdag, S., Rijneveld, J., and Mohaisen, A. (2018). XMSS: extended merkle signature scheme. *RFC*, 8391:1–74.

Hülsing, A., Gazdag, S.-L., Butin, D., and Buchmann, J. (2015a). Hash-based signatures: An outline for a new standard. In *Workshop on Cybersecurity in a Post-Quantum World, NIST*.

Hülsing, A., Rausch, L., and Buchmann, J. A. (2017). Optimal parameters for XMSS^{MT}. *IACR Cryptology ePrint Archive*, 2017:966.

Hülsing, A., Rijneveld, J., and Schwabe, P. (2015b). Armed SPHINCS - computing a 41kb signature in 16kb of RAM. *IACR Cryptology ePrint Archive*, 2015:1042.

Lamport, L. (1979). Constructing digital signatures from a one-way function. Technical report, SRI International Computer Science Laboratory.

Merkle, R. (1979). *Secrecy, authentication and public key systems / A certified digital signature*. PhD thesis, Dept. of Electrical Engineering, Stanford University.

Naor, M. and Yung, M. (1989). Universal one-way hash functions and their cryptographic applications. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 33–43, New York, NY, USA. ACM.

NIST (2016). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Announcement.

Pereira, G. C. C. F. (2015). *Assinaturas Digitais Pós Quanticas Multivariadas e Baseadas em Hash*. PhD thesis, Departamento de Engenharia de Computação e Sistemas Digitais, Escola Politécnica da Universidade de São Paulo.

Pereira, G. C. C. F., Puodzius, C., and Barreto, P. S. L. M. (2016). Shorter hash-based signatures. *Journal of Systems and Software*, 116:95–100.

Rompel, J. (1990). One-way functions are necessary and sufficient for secure signatures. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 387–394.

Stallings, W. (2014). *Criptografia e Segurança de Redes - Princípios e Práticas (6. ed.* Pearson Education.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full SHA-1. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 570–596.

Wang, W., Jungk, B., W´alde, J., Deng, S., Gupta, N., Szefer, J., and Niederhagen, R. (2018). XMSS and embedded systems - XMSS hardware accelerators for RISC-V. *IACR Cryptology ePrint Archive*, 2018:1225.

Wang, X., Feng, D., Lai, X., and Yu, H. (2004). Collisions for hash functions md4, md5, HAVAL-128 and RIPEMD. *IACR Cryptology ePrint Archive*, 2004:199.

Yuval, G. (1979). How to swindle rabin. *Cryptologia*, 3(3):187–191.

Zheng, A. Y. C. L., Ferraz, L. T. D., and Jr., M. A. S. (2018). A clipping technique for shorter hash-based signatures. In *Anais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 167–180.