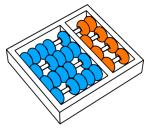
Universidade Estadual de Campinas
Instituto de Computação



Ana Karina Dourado Salina de Oliveira

Implementação Eficiente em Software do Esquema de Assinatura de Merkle e suas Variantes

CAMPINAS
2017

Ana Karina Dourado Salina de Oliveira

Implementação Eficiente em Software do Esquema de Assinatura de Merkle e suas Variantes

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Julio César López Hernández

Este exemplar corresponde à versão final da Tese defendida por Ana Karina Dourado Salina de Oliveira e orientada pelo Prof. Dr. Julio César López Hernández.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

OL4i

Oliveira, Ana Karina Dourado Salina de, 1974-
Implementação eficiente em software do esquema de assinatura de Merkle e suas variantes / Ana Karina Dourado Salina de Oliveira. – Campinas, SP : [s.n.], 2017.

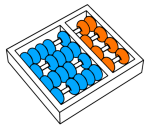
Orientador: Julio César López Hernández.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia de dados (Computação). 2. Criptografia pós-quântica. 3. Criptografia de chaves públicas. 4. Merkle, Assinaturas de. 5. Funções hash. 6. Assinaturas digitais. I. López Hernández, Julio César,1961-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Efficient software implementation of the Merkle signature scheme and its variants
Palavras-chave em inglês:
Data encryption (Computer science)
Post-quantum cryptography
Public key cryptography
Merkle signatures
Hash functions
Digital signatures
Área de concentração: Ciência da Computação
Titulação: Doutora em Ciência da Computação
Banca examinadora:
Julio César López Hernández [Orientador]
Ricardo Felipe Custódio
Marcos Antonio Simplicio Junior
Ricardo Dahab
Marco Aurélio Henrique
Data de defesa: 07-04-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Ana Karina Dourado Salina de Oliveira

Implementação Eficiente em Software do Esquema de Assinatura de Merkle e suas Variantes

Banca Examinadora:

- Prof. Dr. Julio César López Hernández
  IC/UNICAMP
- Prof. Dr. Ricardo Felipe Custódio
  INE/CTC/UFSC
- Prof. Dr. Marcos Antonio Simplicio Junior
  LARC/USP -São Paulo
- Prof. Dr. Ricardo Dahab
  IC/UNICAMP
- Prof. Dr. Marco Aurélio Henriques
  FEEC/UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 22 de março de 2017

Agradecimentos

Eu agradeço a Deus pela oportunidade de realizar mais este trabalho.
Agradeço a minha família e em especial aos meus filhos Henrique Salina de Oliveira e Kamila Salina de Oliveira pela compreensão e ao meu marido Henry Tamashiro de Oliveira pelo apoio e paciência durante todos esses anos do doutorado.
Aos meus pais Edgar e Maria Eliza e sogros Hugo e Margarida pela torcida e incentivo.
Ao professor Dr. Julio López pela orientação neste trabalho.
Um agradecimento especial aos amigos da FACOM pelo apoio e contribuições valiosas neste trabalho.
As minhas amigas pelo suporte dado durante as idas à Campinas.
Aos colegas deste doutorado, que compartilhamos horas de estudo, em especial ao Armando Faz-Hernández por suas contribuições, Roberto Cabral pelo trabalho em conjunto e Amanda Cristina pela amizade.
Ao Pedro M. S. Eleutério por suas revisões em inglês.
A minha prima Luana por seu apoio em Campinas.
A minha tia Marialice por suas correções em português.
Ao prof. Dr. Johannes Buchmann e Dr. Andreas Hülsing pelas indicações e referências.
A todos que direta ou indiretamente apoiaram a realização deste doutorado.

Resumo

Um esquema de assinatura digital é uma importante ferramenta para a criptografia de chave pública e uma tecnologia essencial para prover autenticidade, integridade e irretratabilidade. Assinaturas digitais são utilizadas em muitas aplicações práticas, tais como: *eCommerce*, *eGovernment*, distribuição de software, autenticação de usuários, dentre outras aplicações.

Os esquemas de assinaturas digitais mais utilizados nos dias de hoje são o RSA [50], o DSA [15], o ECDSA [30] e o EdDSA [12]. Estes esquemas não são imunes aos computadores quânticos, pois sua segurança depende da dificuldade de fatorar grandes números inteiros e calcular logaritmos discretos [5].

O esquema de assinatura digital de Merkle [39] e suas variantes são baseados em funções de resumo e são considerados resistentes aos computadores quânticos. Estes esquemas são candidatos promissores a esquemas de assinaturas digitais com segurança quântica e têm sido objeto de esforços para padronização.

Neste trabalho, apresenta-se uma implementação otimizada em software de duas propostas de padrão dos esquemas de assinatura baseados em funções de resumo utilizando um conjunto de instruções AVX2 nos processadores Haswell e Skylake. A implementação utiliza várias técnicas de otimização para acelerar a função de resumo subjacente, resultando em um melhor desempenho nos algoritmos de geração de chaves, assinatura e verificação. A implementação utilizou os parâmetros selecionados pelas propostas do LMS [37] e do XMSS [23]. Os desempenhos das implementações em software foram medidos em um processador Haswell (Intel Core i7-4770K de 3,4 GHz) e Skylake (Intel Core i7-6700K de 4,2 GHz).

Em particular, a operação de assinatura do esquema XMSS, usando SHA-256 no processador Skylake, pode ser calculada em 3.884.005 ciclos (1.031 assinaturas por segundo) para o nível de segurança de 128 bits (contra ataques quânticos), usando uma árvore de altura 60 com 12 camadas. Para o LMS, a mesma operação requer 1.287.778 ciclos (3.115 assinaturas por segundo). As chaves de assinaturas do XMSS e do LMS têm o mesmo tamanho, porém, no LMS, a chave privada é menor do que no XMSS, enquanto a chave pública é maior do que no XMSS.

Os resultados da implementação em software indicam que ambas as propostas para assinaturas digitais baseadas em funções de resumo, LMS e XMSS, têm alto desempenho quando são implementadas utilizando o conjunto de instruções vetoriais nos processadores atuais da Intel.

Abstract

A digital signature scheme is a relevant tool for public key cryptography and an essential technology to provide authenticity, integrity, and non-repudiation of data. Digital signatures are used in many practical applications, such as eCommerce, eGovernment, software distribution, user authentication, and various other applications.

The digital signature algorithms that are used in practice today are RSA [50], DSA [15], ECDSA [30] and EdDSA [12]. They are not quantum immune since their security relies on the difficulty of factoring large composite integers and computing discrete logarithms [5].

The Merkle digital signature scheme [39] (and its variants) is based on hash functions and is considered secure against quantum computing. Merkle-based schemes are promising candidates for providing digital signature schemes in the quantum scenario, and for that reason, these schemes have been proposed for standardization.

In this work, we present an optimized software implementation of two standard proposals of hash-based signatures using a set of AVX2 instructions on Haswell and Skylake processors. The implementation uses various optimization techniques to speed up the underlying hash function evaluation, and consequently, the algorithms for key generation, signature, and verification process. The implementation uses the parameters specified in the proposals LMS [37] and XMSS [23]. The performance results of the software implementation were measured in a modern Haswell processor (Intel Core i7-4770K 3.4 GHz) and Skylake (Intel Core i7-6700K 4.2 GHz).

In particular, the signature operation in XMSS scheme, using SHA-256 in the Skylake processor, can be calculated in 3,884,005 cycles (1,031 signatures per second) for a 128-bit security level (against quantum attacks) using a tree of height 60 and 12 layers. For the LMS, signature operation requires 1,287,778 cycles (3,115 signatures per second). Both XMSS and LMS signature keys are equal in size, however, in the LMS, the private key is smaller than XMSS, while the public key is larger than XMSS.

The results of the software implementation indicate that both proposals LMS and XMSS for digital signature based on hash functions have high performance when implemented using the vector instruction set on recent Intel processors.

Lista de Figuras

Lista de Tabelas

Lista de Abreviações e Siglas

AVX	*Advanced Vector Extensions*
CMAC	*Cipher-based Message Authentication Code*
CMSS	*An Improved Merkle Signature Scheme*
EUCMA	*Existentially Unforgeable under Adaptively Chosen Message Attacks*
GMSS	*Merkle Signatures with Virtually Unlimited Signature Capacity*
HMAC	*keyed-Hash Message Authentication Code*
HSS	*Hierarchical Signature System*
KDF	*Key Derivation Function*
LDOTS	*Lamport-Diffie One-Time Signature*
LDWM	*Lamport-Diffie-Winternitz-Merkle*
LMS	*Leighton e Micali Signature*
MAC	*Message Authentication Code*
MMX	*Multimedia eXtensions*
MSS	*Merkle Signature Scheme*
NIST	*National Institute of Standards and Technology*
NSA	*National Security Agency*
OTS	*One-Time Signature*
PRF	*Pseudo Random Function*
SIMD	*Single Instruction Multiple Data*
SSE	*Streaming SIMD Extensions*
WOTS	*Winternitz One-Time Signature*
WOTS$^+$	*Variants of the Winternitz One Time Signature Scheme*
XMSS	*Extended Merkle Signature Scheme*
XMSS-MT	*Multi-Tree variant of XMSS*
XOF	*Extendable Output Functions*

Sumário

Capítulo 1

Introdução

No final do século XX, a criptografia passou a ser muito mais do que a arte de escrever ou solucionar códigos. A criptografia passou a lidar com problemas de autenticação de mensagens, assinaturas digitais, protocolos de troca de chaves secretas, protocolos de autenticação, leilões eletrônicos, eleições eletrônicas, dinheiro digital, entre outros. De fato, pode-se dizer que a criptografia moderna se preocupa com problemas que podem surgir em qualquer computação distribuída passível de sofrer um ataque, seja ele interno ou externo. Segundo [32], pode-se dizer que a criptografia é a ciência que estuda técnicas de segurança da informação digital, de transações e de computação distribuída. Usuários dependem da criptografia toda vez que eles acessam um *website* seguro. Métodos de criptografia são utilizados para executar controle de acesso em sistemas operacionais multi-usuário, assim como métodos de proteção de software empregam criptografia, autenticação e outras ferramentas para evitar cópias indevidas [32].

Por muitos anos, todas as codificações e cifradores dependiam de chaves secretas. Na criptografia de chave secreta (ou criptografia simétrica), duas pessoas que querem se comunicar, precisam compartilhar uma chave privada. Algoritmos de criptografia simétrica proveem, além do sigilo, os códigos de autenticação de mensagem (*Message Authentication Code*–MAC), que garantem a integridade e a autenticidade da mensagem [32]. Um dos problemas da criptografia simétrica é que chaves secretas não podem simplesmente ser enviadas por meio de um canal inseguro, porque um adversário pode observá-las no caminho. O compartilhamento das chaves secretas pode ser feito através de um canal seguro, ou utilizando um serviço confiável de troca de mensagens, ou entregue pessoalmente.

A criptografia de chave pública utiliza um par de chaves: uma pública, que é publicamente conhecida, e uma privada, que é conhecida apenas pelo proprietário. Um dos principais objetivos da criptografia de chave pública é permitir que duas pessoas troquem informações confidenciais por meio de um canal inseguro [5]. A criptografia de chave pública provê, além do sigilo, a assinatura digital das mensagens [32]. Em 1978, os pesquisadores Ron Rivest, Adi Shamir e Leonard Adleman (RSA) [50], propuseram o primeiro esquema prático de assinatura digital utilizando criptografia de chave pública.

Esquemas de assinatura digital estão entre as primitivas criptográficas mais utilizadas na prática [32] e é uma importante ferramenta da criptografia pública e uma tecnologia essencial para prover autenticidade, integridade e irretratabilidade. Atualmente, alguns dos esquemas de assinaturas digitais mais utilizados são o RSA [50], o DSA [15], o ECDSA

[30] e o EdDSA [12]. Esses esquemas têm sua segurança baseada na dificuldade de fatorar grandes números inteiros (RSA) ou calcular logaritmos discretos (DSA, ECDSA, EdDSA). Em 1994, Shor [54] introduziu um algoritmo quântico para fatorar inteiros e calcular logaritmos discretos em tempo polinomial, usando operações simples em um computador quântico. O surgimento dos computadores quânticos poderá causar um problema para a segurança baseada nos esquemas de criptografia mais utilizados na atualidade, pois com a implementação do algoritmo de Shor, existirá a possibilidade de quebrar a segurança de esquemas de assinatura digital como o RSA, DSA, ECDSA e EdDSA.

A pergunta que surge é: quais criptossistemas são resistentes aos computadores quânticos? Quando um bom sistema substituto for encontrado, ainda haverá questões relativas a logística de mudança de todos os criptossistemas em uso. Além disso, a mais sensível informação cifrada deve ficar segura mesmo com os computadores quânticos. Estes dados devem ser protegidos com resistência a criptossistemas quânticos [5]. A criptografia pós-quântica tem focado na construção de criptossistemas que permanecerão seguros mesmo na presença de computadores quânticos. Pesquisas relativas à criptografia pós-quântica têm ganhado relevância e diversos esquemas voltaram a ser estudados, ao mesmo tempo que surgiram novos esquemas, tais como: Esquemas de assinatura digital baseados em funções de resumo [40], criptografia baseada em código [48] e criptografia baseada em reticulados (*lattice*).

O esquema de assinatura *one-time* (*One-Time Signature*–OTS), proposto por Lamport [33], garante a segurança da assinatura se um par de chaves for utilizado para assinar uma única mensagem. Esquemas OTS são inadequados para a maioria das situações práticas, pois cada par de chaves só pode ser utilizado uma vez e para assinar uma única mensagem.

Merkle [40] propôs um esquema de assinatura *N-Time* que pode ser utilizado para assinar múltiplas mensagens. O esquema de Merkle (*Merkle Signature Scheme*–MSS) transforma a validade de um número arbitrário e fixo de chaves de verificação OTS para a validade de uma única chave pública, que é a raiz de uma árvore binária. Os esquemas OTS e MSS são construídos a partir de uma função de resumo e, por este motivo, são classificados como esquemas de assinatura baseados em funções de resumo.

O esquema de assinatura de Merkle estendido (*Extended Merkle Signature Scheme*–XMSS) [2] também é um esquema de assinatura baseado em funções de resumo. Este esquema tem assinaturas de tamanho menor em relação ao esquema de Merkle. Em [2], foi provado que o esquema XMSS é seguro, baseado em algumas propriedades da função de resumo subjacente. Mais especificamente, não é necessário que a função de resumo criptográfica seja resistente a colisão para garantir a segurança do XMSS, podendo substituir este requisito pela resistência a segunda pré-imagem [23].

Estes esquemas, OTS, MSS e XMSS, geralmente apresentam chaves públicas e privadas de tamanho pequeno, bem como uma rápida geração de assinatura e verificação, mas possuem grandes assinaturas e uma geração de chaves relativamente lenta [23]. O esquema XMSS-MT é uma variante multi-árvore (Multi-Tree) que melhora o tempo de geração de chaves e aumenta o número de assinaturas que podem ser geradas. Os esquemas XMSS e XMSS-MT são adequados para implementações compactas e são naturalmente resistentes à maioria dos tipos de ataques de canal lateral [23].

Independentemente da efetiva realização da computação quântica, as organizações de padronização estão encorajando a transição para a criptografia pós-quântica, isto é, esquemas criptográficos nos quais não há indícios de vulnerabilidade aos ataques de computadores quânticos [38]. Estão em andamento esforços de padronização pelo Instituto Nacional de Padrões e Tecnologia (NIST), que está aceitando submissões para algoritmos criptográficos de chave pública resistentes aos computadores quânticos [47].

As assinaturas baseadas em funções de resumo estão atualmente em processo de padronização. Há duas propostas de padrão das variantes do esquema de Merkle: Em [37], é descrito o esquema de Leighton e Micali (*Leighton e Micali Signature*–LMS), que é uma adaptação do esquema de assinatura one-time Lamport-Diffie-Winternitz-Merkle (LDWM) [39] e, em [23], é descrito o XMSS [2].

Assim, são necessárias implementações em software eficientes e seguras que comprovem a viabilidade da utilização desses esquemas. Uma das formas, é utilizar recursos específicos das máquinas, como o conjunto de instruções vetoriais AVX2 (*Advanced Vector Extensions*) [28], para melhorar o desempenho. Estas máquinas possuem um conjunto de instruções SIMD (*Single Instruction Multiple Data*) que permite executar uma única instrução sobre um conjunto de dados, explorando o paralelismo de dados. Como as funções de resumo realizam muitos embaralhamentos de dados, elas podem se beneficiar do uso de vetores de instruções para executar operações em paralelo utilizando SIMD. Desta forma, a implementação dos esquemas de assinatura baseados em funções de resumo, também é favorecida utilizando o conjunto de instruções vetoriais AVX2.

1.1 Contribuições

Como visto, a criptografia pós-quântica é um tema de muita relevância atualmente e não há muitas pesquisas com foco na implementação em software dos esquemas de assinatura digital pós-quântica baseados em funções de resumo. Este trabalho pretende preencher essa lacuna e tem como principais contribuições:

- A tese apresenta novas técnicas computacionais para a implementação eficiente em software de esquemas de assinatura digital baseados em funções de resumo. Em particular é mostrado como acelerar o desempenho de duas propostas padrão LMS e XMSS utilizando o conjunto de instruções vetoriais AVX2 nas microarquiteturas Haswell e Skylake.

- O trabalho descreve uma implementação em software dos esquema de assinatura digital LMS e XMSS para diferentes níveis de segurança nos processadores Haswell e Skylake. Adicionalmente, é apresentado uma análise comparativa dos esquema de assinatura digital implementados.

1.2 Metodologia

Para atingir os objetivos propostos e as contribuições deste trabalho, a metodologia utilizada envolve as seguintes etapas:

- Análise detalhada do esquema de assinatura de Merkle e suas variantes.
- Análise dos esquemas LMS e XMSS.
- Análise do conjunto de instruções vetoriais das microarquiteturas Haswell e Skylake e das portas de execução disponíveis nestes processadores, orientando para a implementação dos esquemas.
- Otimizações em software aplicadas às funções de resumo SHA2 e SHA3, que se beneficiam das microarquiteturas utilizadas.
- Otimizações aplicadas às funções do LMS e do XMSS.
- Implementações em software das duas propostas de padrão escritas na linguagem C.
- Comparação de segurança e desempenho das duas propostas de padrão.

1.3 Trabalhos Relacionados

Winternitz [39] propôs uma melhora no esquema de assinatura OTS de Lamport [33], reduzindo o tamanho das chaves públicas e privadas. A desvantagem dos esquemas de assinatura OTS é que um par de chaves só pode ser utilizado para assinar um único documento.

O MSS [39] introduziu o esquema de assinatura *N-Time*. Este esquema propôs a construção de uma árvore binária na qual as folhas da árvore são resumos das chaves públicas OTS, e a raiz da árvore é a chave pública. Merkle apresentou também o primeiro algoritmo para calcular o caminho de autenticação (*Tree Traversal*), que é utilizado para validar a chave pública.

Em 2003, foi apresentado um algoritmo variante para calcular os caminhos de autenticação (*Fractal Merkle Tree Representation and Traversal*) [29]. Este algoritmo eficiente gera uma sequência de folhas com seus caminhos de autenticação associados melhorando tanto os requisitos de espaço e tempo de execução em relação aos algoritmos conhecidos.

Szydlo [56] propôs em 2004 uma técnica para o algoritmo *Tree Traversal* de Merkle (Merkle *Tree Traversal in Log Space and Time*), que requer somente espaço e tempo logarítmicos. Buchmann et al. [4] descreveram um algoritmo eficiente que calcula os caminhos de autenticação baseado no trabalho de Szydlo.

Algumas variantes do esquema MSS propostas foram: o CMSS [7], o GMSS [3], o XMSS [2] e o XMSS-MT [26]. O CMSS [7] (um aperfeiçoamento no esquema de assinatura de Merkle), constrói duas árvores encadeadas reduzindo o tempo de geração das chaves e da assinatura, e permite assinar até 2^{40} documentos. O CMSS reduz o tamanho da chave privada com a utilização de um gerador pseudoaleatório (PRNG). Em 2007, o GMSS [3] (O esquema de Merkle com capacidade de assinaturas virtualmente ilimitada), permitiu um significante aumento no número de documentos a serem assinados com um par de chaves e reduziu o tamanho e o tempo de geração de assinatura.

Em 2011 foi introduzido o XMSS, um esquema de assinatura *forward-secure* com requisitos mínimos de segurança. Este esquema utiliza uma versão modificada do esquema de Winternitz (WOTS$^+$) [25], que é existencialmente infalsificável sobre ataque de mensagem escolhida quando instanciado com uma família de funções pseudoaleatórias. Posteriormente, foi introduzido o esquema XMSS-MT [26] que permite assinar um grande número de documentos.

Em 2014, McGrew e Curcio [36] publicaram um Internet-Draft que descreve um esquema de assinatura baseado em funções de resumo. Este sistema, especifica o esquema de assinatura *one-time* baseado no trabalho de Lamport [33], Diffie [11], Winternitz [39] e Merkle [39] (LDWM). Nesta proposta, é apresentada uma moderna análise de segurança desses algoritmos. As versões mais atuais passaram a se chamar LMS. Estas versões seguem as versões anteriores, de Lamport, Diffie, Winternitz, e Merkle, porém tiveram adaptações conforme o trabalho de Leighton e Micali [34].

Em 2015, Hülsing et al. [22] propuseram um Internet-Draft que especifica o XMSS: *Extended hash-based signatures* com o esquema WOTS$^+$, uma única árvore (XMSS) e multi-árvores (XMSS-MT).

No artigo [10], de minha co-autoria, descreve-se uma implementação otimizada em software do esquema MSS e suas variantes GMSS, XMSS e XMSS-MT. Para acelerar os algoritmos de geração de chaves, assinatura e verificação foi utilizado um subconjunto de instruções vetoriais AVX2 em um processador Haswell da Intel.

Em 2015, foi introduzido o SPHINCS [1], um esquema de assinatura baseado em funções de resumo que foi projetado para fornecer segurança de 128 bits, mesmo contra atacantes equipados com computadores quânticos. Os esquemas típicos de assinatura baseados em funções de resumo são *statefull*, porque precisam registrar informações, chamadas de estado, após cada assinatura. O SHPHINCS é um esquema de assinatura *stateless*, que eliminou a necessidade de manter o estado atual da assinatura, melhorando assim a segurança, pois este esquema de assinatura não precisa registrar informações após cada assinatura. Porém, o SPHINCS tem assinaturas muito grandes, que podem torná-lo inviável em alguns cenários.

Em 2016, Geovandro et al. [49] propuseram um esquema de assinatura baseado em funções de resumo que está relacionado com o esquema de Rohde et al. (REDBP) [51]. A diferença do esquema REDBP é que o esquema proposto possui assinaturas de tamanho menor e assinaturas mais rápidas. Esse esquema propõe utilizar um valor aleatório juntamente com a mensagem como entrada para a função de resumo, permitindo reduzir o tamanho da saída da função de resumo de $2n$ para n bits. Esta redução é baseada no requisito de uma função de resumo resistente à pré-imagem ao invés de uma função resistente a colisão.

Em 2016, o trabalho [38] analisou o gerenciamento de estados nos esquemas de assinaturas *N-Time* baseados em funções de resumo e propôs um esquema híbrido *stateless/stateful* que visa a proteção contra cópias não intencionais do estado da chave privada, tem a vantagem do método *stateless* e tem assinaturas de tamanho menor do que o SPHINCS.

1.4 Organização da Tese

O restante desta tese está dividida nos seguintes capítulos:

- Capítulo 2 apresenta as primitivas do esquema de Merkle e suas variantes, tais como as propriedades das funções de resumo, construções, algoritmos das funções SHA2 e SHA3, usos das funções de resumos e segurança dessas funções.

- Capítulo 3 descreve os conceitos do esquema MSS e de suas variantes (CMSS, GMSS, XMSS e XMSS-MT). Também apresenta os esquemas de assinatura one-time de Lamport e de Winternitz, o algoritmo eficiente do caminho de autenticação que foi implementado neste trabalho e a segurança desses esquemas.

- Capítulo 4 aborda as propostas de padrão das variantes do esquema de Merkle (LMS e XMSS) que foram implementadas neste trabalho. Esse capítulo também apresenta os algoritmos de geração de chaves, assinatura e verificação do LMS e do XMSS e a segurança desses esquemas.

- Capítulo 5 ilustra a microarquitetura (Haswell e Skylake) e as otimizações realizadas neste trabalho. Esse capítulo também aborda as otimizações em software realizadas nas funções de resumo para estas microarquiteturas e as otimizações nas funções do LMS e do XMSS.

- Capítulo 6 apresenta os resultados da implementação do LMS e do XMSS para os níveis de segurança de 128 e 256 bits com as funções de resumo SHA2 e SHA3. Além disso, mostra os parâmetros utilizados na implementação desses esquemas. No final, apresenta uma análise dos resultados e uma comparação das duas propostas de padrão.

- Capítulo 7 apresenta as conclusões deste trabalho.

Capítulo 2

Primitivas do Esquema de Merkle e suas variantes

Neste capítulo são apresentadas as principais primitivas utilizadas neste trabalho para implementar o esquema de assinatura digital de Merkle e suas variantes. Este capítulo está organizado como segue: Seção 2.1, apresenta as propriedades das funções de resumo criptográfico. Nas Seções 2.2 e 2.3 são descritas as construções e os algoritmos de resumo padrão do NIST SHA2 e SHA3 implementados neste trabalho. A Seção 2.4, mostra algumas aplicações das funções de resumo. A Seção 2.5 sumariza a segurança das funções de resumo SHA2 e SHA3.

2.1 Funções de Resumo

As funções de resumo surgiram como uma importante ferramenta para a criptografia e são usadas em aplicações de segurança, tais como: esquemas de assinatura digital, identificação de dados, derivação de chaves, entre outros. Funções de resumo são uma importante primitiva criptográfica e podem ser usadas para resolver uma grande variedade de problemas que envolvem autenticação e integridade. De acordo com o padrão NIST (*National Institute of Standards and Technology*) [46], o projeto das funções de resumo deve ser baseado nos seguintes requisitos: facilidade de implementação e utilização, eficiência e resistência aos ataques cripto-analiticos conhecidos. Uma função de resumo $H : \{0,1\}^* \longrightarrow \{0,1\}^l$ mapeia uma mensagem M, que é uma cadeia binária de tamanho finito e arbitrário para um resumo r que é uma cadeia binária de tamanho fixo l bits. Deste modo, $r = H(M)$.

2.1.1 Propriedades das Funções de Resumo

Algumas propriedades das funções de resumo são: resistência à pré-imagem, resistência à segunda pré-imagem e resistência à colisão [14].

1. *Resistência à pré-imagem:* dada uma função $H : \{0,1\}^* \longrightarrow \{0,1\}^l$ e um resumo r, deve ser computacionalmente inviável encontrar uma mensagem M tal que $H(M) = r$.

2. *Resistência à segunda pré-imagem:* dada uma função $H : \{0,1\}^* \longrightarrow \{0,1\}^l$ e uma mensagem M, deve ser computacionalmente inviável encontrar uma mensagem $M' \neq M$, tal que $H(M) = H(M')$.

3. *Resistência à colisão:* dada uma função $H : \{0,1\}^* \longrightarrow \{0,1\}^l$, deve ser computacionalmente inviável encontrar M e M' tal que $M \neq M'$ e $H(M) = H(M')$ [14].

Resistência à colisões é uma propriedade que torna as funções de resumo adequadas para uso em esquemas de assinatura. Toda função de resumo tem um número infinito de colisões, pois um número infinito de mensagens de entrada produzem um número finito de resumos. Assim, uma função de resumo nunca é livre de colisões. A exigência é apenas que, embora existam colisões, elas não possam ser computacionalmente encontradas.

Conforme especificado pelo NIST [43] no concurso do SHA3, a função de resumo deve atender a mínimos requisitos de aceitabilidade. Esses requisitos têm como base os seguintes fatores: segurança, eficiência computacional, requisitos de memória, adequação de hardware e software, simplicidade, flexibilidade e requisitos de licença.

2.2 Construções das Funções de Resumo

A seguir, são descritas as construções das funções de resumo utilizadas neste trabalho. Primeiro, na Seção 2.2.1, é descrita a construção Merkle-Dangard [9], que é utilizada pela função de resumo SHA2. Em sequência, na Seção 2.2.2, é apresentada a construção esponja [17], base da função de resumo SHA3.

2.2.1 Construção Merkle-Dangard

Construção Merkle-Damgard é um método de construir uma função de resumo resistente à colisão [32]. Esse método permite a conversão de qualquer função de compressão de tamanho fixo resistente a colisão para uma função de resumo, mantendo a propriedade de resistência à colisão. Nesta construção, uma mensagem M é dividida em k blocos $B^1, \ldots, B^k$ (sendo o bloco de tamanho fixo l bits), com um preenchimento (*padding*) se necessário. Se o tamanho da mensagem não for múltiplo de l, então a mensagem tem que ser preenchida com um número apropriado de bits. Conforme provado no Teorema 4.14 em [32], se a função de compressão é resistente à colisão, então a função de resumo também é resistente a colisão.

A Figura 2.1 apresenta a construção Merkle-Dangard. O primeiro bloco Bl_1, juntamente com um valor inicial IV, é passado como entrada para a função de compressão f_c, retornando o resultado do encadeamento $CV^1 = f_c(IV, B^1)$. Então, CV^1 agora é o novo vetor inicial e é passado como entrada junto com B^2 para f_c produzindo o segundo vetor de encadeamento $CV^2 = f_c(CV^1, B^2)$. Este encadeamento vai até B^k, e CV^{k-1}, gerando $CV^k = f_c(CV^{k-1}, B^k)$. Assim, a saída da função de resumo é $H(M) = CV^k$.

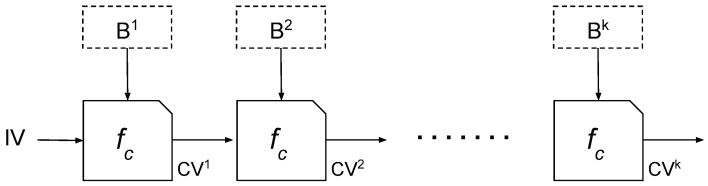


Figura 2.1: Construção Merkle-Dangard

2.2.2 Construção Esponja

A construção esponja foi proposta por Bertoni, Daemen, Peeters e Van Assche em 2007 [17]. Essa construção é baseada em uma permutação f_p de tamanho fixo e uma regra de preenchimento *pad* que constrói uma função, denominada função esponja que tem uma entrada com tamanho variado e a saída tem um tamanho arbitrário. A função esponja imita o comportamento de um oráculo aleatório, exceto pelo fato de que na construção esponja podem ocorrer colisões internas, uma vez que ela possui uma espaço de memória finito [17].

A analogia com uma esponja é que a função absorve um número arbitrário de bits de entrada para o seu estado, depois um número arbitrário de bits de saída são espremidos para fora de seu estado. Conforme especificado em [17], uma função esponja pode ser usada como uma função de resumo de l bits, truncando a saída da função.

A função esponja ($SPONGE[f_p, pad, r]$), definida em [17], tem três componentes:

- uma função de permutação, denotada por f_p,
- um parâmetro taxa de bits, denotado por r, e
- uma regra de preenchimento, denotado por *pad*.

A função f_p mapeia uma cadeia de tamanho fixo, denotada por b, para uma cadeia do mesmo tamanho. O parâmetro b é chamado largura de f_p. As funções de resumo SHA3, são exemplos de construção esponja, onde a função base f_p é invertível, isto é, uma permutação, embora a construção esponja não requeira que f_p seja invertível. A taxa r é um inteiro positivo que é estritamente menor do que a largura b. A capacidade, denotada por c, é o inteiro positivo $b - r$. Assim, $b = r + c$.

A regra de preenchimento, *pad*, é uma função que produz *padding*, ou seja, uma cadeia com um tamanho adequado para anexar a outra cadeia. Dentro da construção esponja, *padding* é acrescentado à mensagem para assegurar que pode ser dividida em uma sequência de cadeias de r bits. Em geral, dado um inteiro positivo x e um número inteiro não negativo y, a saída $pad(x, y)$ é uma cadeia com a propriedade onde $y + len(pad(x, y))$ é um positivo múltiplo de x.

A Figura 2.2 ilustra a construção esponja. Inicialmente, a mensagem de entrada é preenchida com o *pad* e dividida em blocos de r bits. Em seguida, os b bits do estado são inicializados com zero. A construção esponja executa duas fases: absorção e extração.

Na fase de absorção é aplicado uma operação XOR entre um bloco de r bits da mensagem de entrada com os primeiros r bits do estado atual. Então, o estado é atualizado por meio de uma permutação f_p. Esse processo é aplicado para todos os blocos da mensagem. Na fase de extração os primeiros r bits do estado são usados como saída. Se o tamanho do resumo (y bits) for maior que o tamanho do bloco (r bits), então o estado é processado novamente pela permutação f_p e os r bits retornados são concatenados com os r bits retornados anteriormente. Esse processo é repetido até que os y bits requeridos sejam extraídos.

Os últimos c bits do estado nunca são afetados diretamente pelos blocos de entrada na fase de absorção e nem extraídos na fase de extração. O parâmetro c determina a segurança atingível pela construção [17].

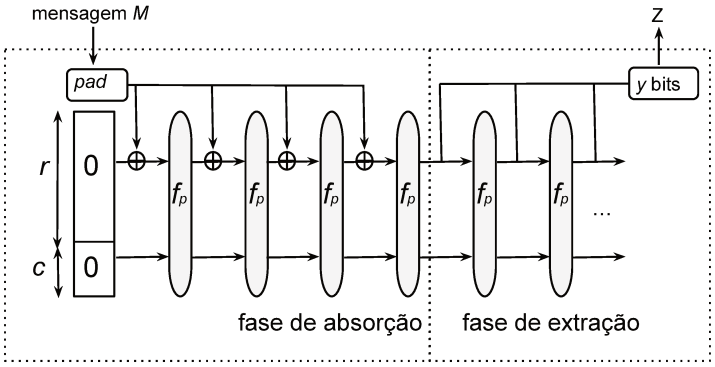


Figura 2.2: Construção Esponja

O Algoritmo 1 especifica a função $SPONGE[f_p, pad, r]$. Este algoritmo recebe (M, l), onde M é a mensagem de entrada e y é o tamanho em bits escolhido para ser retornado pelo Algoritmo 1. A função $len()$ retorna o comprimento em bits de uma cadeia. Em princípio, a saída pode ser considerada como uma cadeia infinita, cuja execução é interrompida após produzir o número de bits desejado.

Uma função esponja aleatória é uma instância da construção esponja com a permutação f_p escolhida aleatoriamente dentre um conjunto de permutações sobre b bits. Em [17], foi provado que uma função de esponja aleatória é tão forte quanto um oráculo aleatório, exceto os efeitos induzidos pela memória finita. Uma esponja aleatória pode servir como um modelo de referência para expressar provas de segurança para funções de resumo iteradas e cifras de fluxo [18]. A probabilidade de sucesso de um ataque depende não só da natureza do ataque, mas também dos parâmetros da função esponja aleatória que forem escolhidos, isto é, da sua capacidade, da taxa de bits e se é uma permutação aleatória ou uma transformação aleatória.

2.3 Algoritmos de Funções de Resumo

A primeira versão do algoritmo de função de resumo designado pela NSA e padronizado pelo NIST foi o SHA0. Segundo especificado pelo NIST [42], a NSA (*National Security Agency*) encontrou uma fraqueza no *SHA0* e publicou uma versão melhorada, o SHA1. O

Algorithm 1 Função $SPONGE[f_p, pad, r](M, l)$ [17]

Entrada: a mensagem M, o tamanho da saída y, o tamanho b e a taxa r.
Saída: cadeia de tamanho y.

$B = M||pad(r, len(M))$;//$B = B^0||\ldots||B^{k-1}$;
$k = len(B)/r$;
$c = b - r$;
$S = 0^b$;
for $(i = 0; i \leq k - 1; i++)$ **do**
    $S = f_p(S \oplus (B^i||0^c))$;
end for
$Z = NULL$;
$Z = Z||Trunc_r(S)$;
while $(l > |Z|)$ **do**
    $S = f_p(S)$;
    $Z = Z||Trunc_r(S)$;
end while
return $Trunc_d(Z)$;

SHA1 é uma função de resumo de 160 bits e também é considerado inseguro. A diferença do SHA0 para o SHA1 é a adição de uma rotação de um bit para recorrência linear. Um dos problemas de segurança do SHA1 é o resumo de tamanho pequeno de 160 bits, pois colisões contra qualquer função de resumo de 160 bits podem ser encontradas em somente 2^{80} passos [14]. Em [35], foi demonstrado ataques práticos de colisão para a função SHA1.

Em 2001, o NIST publicou um novo padrão de projeto contendo três novas funções de resumo e em 2004, o NIST incluiu 4 novas funções de resumo referenciadas como a família SHA2, que tem 224, 256, 384 e 512 bits. Em 2012, Keccak foi o algoritmo vencedor da competição NIST para o novo algoritmo de resumo criptográfico SHA3.

2.3.1 Algoritmo SHA-256

Nesta seção é descrito o algoritmo SHA-256 com as principais funções conforme descrito em [42]. Os algoritmos SHA-224, SHA-256, SHA-384, SHA-512 usam funções similares, porém as funções do SHA-224 e do SHA-256 têm tamanho de palavra de 32 bits, enquanto as funções do SHA-384 e do SHA-512 têm tamanho de palavra de 64 bits.

Cada algoritmo pode ser descrito em duas etapas: pré-processamento e cálculo do resumo da mensagem. O pré-processamento consiste de três passos: preencher uma mensagem, dividir a mensagem em blocos de l bits e definir os valores de inicialização (H_j) para serem usados no cálculo do resumo, para $0 \leq j \leq 7$.

Preenchimento

O objetivo do preenchimento (*padding*) é garantir que os blocos da mensagem sejam múltiplos de 512 ou 1024 bits, dependendo do algoritmo. Suponha que o tamanho da mensagem, M, é l bits. Por exemplo, no SHA-256, acrescente o bit "1" ao final da mensagem, seguido por b bits zero, onde b é a menor solução não negativa para a equação $l + 1 + b \equiv 448 \bmod 512$. A mensagem e seu *padding* deve ser dividida em k blocos de l

bits, para $l \in \{512, 1024\}$.

Inicialização

Antes de iniciar o cálculo do resumo, valores iniciais são guardados em H_j, para $0 \leq j \leq 7$. As palavras em H_j dependem do tamanho do resumo da mensagem. No SHA-256, os valores de H_j são oito palavras de 32 bits em hexadecimal:

$H_0 =$ 0x6a09e667
$H_1 =$ 0xbb67ae85
$H_2 =$ 0x3c6ef372
$H_3 =$ 0xa54ff53a
$H_4 =$ 0x510e527f
$H_5 =$ 0x9b05688c
$H_6 =$ 0x1f83d9ab
$H_7 =$ 0x5be0cd19

Funções Lógicas

O SHA-224 e o SHA-256 usam seis funções lógicas, onde cada uma opera sobre 3 palavras de 32 bits x,y,e z e produzem como saída uma palavra de 32 bits. Essas funções são:

$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
$\sum_0^{256}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
$\sum_1^{256}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
$\sigma_0^{256}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus ROTR^3(x)$
$\sigma_1^{256}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus ROTR^{10}(x)$
$ROTR^n(x) \approx ROTL^{v-n}(x)$
$ROTR^n(x) = (x \gg n) \vee (x \ll v - n)$
$ROTL^n(x) = (x \ll n) \vee (x \gg v - n)$

No SHA-256, uma mensagem é dividida em k blocos de 512 bits. Algoritmo 2 descreve a função de resumo SHA-256. Cada bloco B^i da mensagem M, para $1 \leq i \leq k$, é processado pela função SHA-256. O cálculo do resumo é gerado a partir de um escalonamento desses blocos, que são iterativamente processados pelas funções lógicas até gerar o resumo da mensagem.

2.3.2 Algoritmo SHA3

O KECCAK [41] foi o algoritmo vencedor da competição do NIST para a escolha do SHA3. A família SHA3 consiste de quatro funções de resumo criptográficas: SHA3-224, SHA3-256, SHA3-384 e SHA3-512; e duas funções de resumo com saída variável (*Extendable Output Functions*–XOF), chamadas SHAKE128 e SHAKE256. Os sufixos "128" e "256" indicam a segurança dessas duas funções, em contraste das outras funções de resumo, nas quais o sufixo indica o tamanho do resumo. O XOF é uma alternativa para

Algorithm 2 Função de resumo SHA-256 [42]

Entrada: a mensagem M, valores iniciais de $H_0, \ldots, H_7$.
Saída: valor do resumo da mensagem $H_0, \ldots, H_7$.

k=quantidade de blocos da mensagem M;
for $(i = 1; i < k; i = i + 1)$ **do**
  for $(t = 0; t < 63; t = t + 1)$ **do**
    if $(t \leq 15)$ **then**
      $W_t = B_t^i$;
    else
      $W_t = \sigma_1^{256}(W_{t-2}) + W_{t-7} + \sigma_0^{256}(W_{t-15}) + W_{t-16}$;
    end if
  end for
  Inicialize as oito variáveis:
    $a = H_0$;
    $b = H_1$;
    $c = H_2$;
    $d = H_3$;
    $e = H_4$;
    $f = H_5$;
    $g = H_6$;
    $h = H_7$;
  for $(t = 0; t < 63; t = t + 1)$ **do**
    $T_1 = h + \sum_1^{256}(e) + Ch(e, f, g) + K_t^{256} + W_t$;
    $T_2 = \sum_0^{256}(a) + Maj(a, b, c)$;
    $h = g$;
    $g = f$;
    $f = e$;
    $e = d + T_1$;
    $d = c$;
    $c = b$;
    $b = a$;
    $a = T_1 + T_2$;
  end for
    $H_0 = a + H_0$;
    $H_1 = b + H_1$;
    $H_2 = c + H_2$;
    $H_3 = d + H_3$;
    $H_4 = e + H_4$;
    $H_5 = f + H_5$;
    $H_6 = g + H_6$;
    $H_7 = h + H_7$;
end for

construções que necessitam de várias chamadas de uma função de resumo e/ou requerem o truncamento dos bits de saída para produzir um tamanho de resumo não padrão. Essas seis funções baseiam-se na construção esponja e têm como base o algoritmo Keccak, que é apresentado a seguir.

Permutação Keccak-p

Uma permutação Keccak-p que possui n_r rodadas e trabalha sobre um estado S de b bits, é denotada por $Keccak - p[b, n_r]$; a permutação é definida para quaisquer $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ e qualquer inteiro positivo n_r. Uma rodada da permutação $Keccak - p$ consiste em uma sequência de cinco transformações, chamadas de etapas de mapeamento.

O estado interno do algoritmo pode ser organizado em uma matriz $5 \times 5 \times v$, onde $v = b/25$ pode ser visto como o tamanho das palavras do estado em bits. As 25 palavras de v bits de um estado $S[x, y] = s_{5x+y}$ para $x \geq 0$, $y < 5$ e s_i de $0 \leq i < 25$, conforme a Matriz 2.1:

$$S = \begin{bmatrix} s0 & s1 & s2 & s3 & s4 \\ s5 & s6 & s7 & s8 & s9 \\ s10 & s11 & s12 & s13 & s14 \\ s15 & s16 & s17 & s18 & s19 \\ s20 & s21 & s22 & s23 & s24 \end{bmatrix} \tag{2.1}$$

Etapas de Mapeamento

As cinco etapas de mapeamento usadas durante uma rodada da função $Keccak - p[b, n_r]$ são denotadas por θ, ρ, π, χ e ι [8]. O algoritmo dessas etapas tem como entrada um estado S e como saída o estado atualizado. A etapa de mapeamento ι é a única que possui uma entrada adicional, um inteiro i_r chamado índice da rodada.

O tamanho do estado é um parâmetro que está omitido da notação, pois b sempre é especificado quando as etapas de mapeamento são chamadas. Para simplificar a notação, todas as operações lógicas e de rotação usadas nas etapas de mapeamento são aplicadas sobre palavras de tamanho v.

Na etapa de mapeamento ρ, cada palavra do estado é rotacionada uma quantidade fixa de r_i bits. Na Matriz 2.2, $R[x, y] = r_{5x+y}$ para $x \geq 0$, $y < 5$, cada elemento r_i representa a quantidade de bits que a palavra s_i será rotada.

$$R = \begin{bmatrix} 0 & 1 & 62 & 28 & 27 \\ 36 & 44 & 6 & 55 & 20 \\ 3 & 10 & 43 & 25 & 39 \\ 41 & 45 & 15 & 21 & 8 \\ 18 & 2 & 61 & 56 & 14 \end{bmatrix} \tag{2.2}$$

Dado um estado S e o índice de uma rodada i_r, uma função Rnd é a transformação

que resulta na aplicação das etapas de mapeamento θ, ρ, π, χ e ι, da seguinte forma:

$$Rnd(S, i_r) = \iota(\chi(\pi(\rho(\theta(S)))), i_r). \tag{2.3}$$

A permutação$Keccak - p[b, n_r]$ consiste de n_r iterações de Rnd. A função Rnd é apresentada no Algoritmo 3.

Algorithm 3 Função Rnd(S) [8]

Entrada: estado S, índice da rodada i_r.
Saída: estado S.

//Etapa de mapeamento θ
$C[y] = S[0, y] \oplus S[5, y] \oplus S[10, y] \oplus S[15, y] \oplus S[20, y]$ para $0 \leq y < 5$;
$D[x] = C[(x - 1) \bmod 5] \oplus (C[(x + 1) \bmod 5] \lll 1)$ para $0 \leq x < 5$;
$S[x, y] = S[x, y] \oplus D[x]$ para $0 \leq x, y < 5$;
//Etapa de mapeamento ρ
$S[x, y] = (S[x, y] \lll R[x, y])$ para $0 \leq x, y \leq 4$;
//Etapa de mapeamento π
$S'[(2y + 3x) \bmod 5, x] = S[x, y]$ para $0 \leq x, y \leq 4$;
//Etapa de mapeamento χ
for ($x \geq 0$ AND $y < 5$) **do**
    $T = (S'[x, (y + 2) \bmod 5] \wedge (\neg S'[x, (y + 1) \bmod 5])$;
    $S[x, y] = S'[x, y] \oplus T$;
end for
//Etapa de mapeamento ι
$S[0, 0] = S[0, 0] \oplus rc(i_r)$;
return S;

Família SHA3/SHAKE

As quatro funções de resumo SHA3 são definidas a partir da aplicação da função $Keccak[c]$ sobre uma mensagem M concatenada com dois bits e a especificação do tamanho de saída, como pode ser visto a seguir:

- SHA3-224(M) = Keccak[448]$(M||01, 224)$.
- SHA3-256(M) = Keccak[512]$(M||01, 256)$.
- SHA3-384(M) = Keccak[768]$(M||01, 384)$.
- SHA3-512(M) = Keccak[1024]$(M||01, 512)$.

Em cada caso, a capacidade sempre é o dobro do tamanho da saída; os dois bits adicionados à mensagem (01) servem para dar suporte à separação de domínio, por exemplo, distinguir as mensagens processadas pela função de resumo SHA3 e pelas funções com saída extensível (XOF).

As duas XOFs do SHA3 são chamadas de SHAKE128 e SHAKE256. Os valores 128 e 256 indicam a segurança dessas funções, em contraste com os valores dos sufixos para as funções de resumo, que indicam o tamanho do resumo.

Os algoritmos das funções de resumo SHAKE128 e SHAKE256 são os primeiros XOFs que o NIST [41] padronizou e são definidas a partir da função $Keccak[c]$. Essas funções recebem uma mensagem M concatenada com quatro bits e com a especificação do tamanho de saída d, como é apresentado a seguir:

- SHAKE128(M) = Keccak[256]$(M||1111, d)$.
- SHAKE256(M) = Keccak[512]$(M||1111, d)$.

2.4 Usos das Funções de Resumo

A seguir, são descritas algumas recomendações de segurança para uso das funções de resumo em aplicações, conforme publicadas no NIST [46] e que são utilizadas na implementação deste trabalho.

2.4.1 Resumo Truncado da Mensagem

Algumas aplicações podem requerer um valor do resumo da mensagem menor do que o fornecido pela função de resumo. Em tais casos, pode-se usar um subconjunto dos bits produzidos pela função de resumo [46].

Conforme especificado pelo NIST [46], se λ é o tamanho desejado em bits do resumo da mensagem, então o resumo truncado da mensagem pode ser usado se obedecer aos seguintes requisitos:

1. O tamanho do bloco de saída da função de resumo deve ser maior do que λ.
2. Selecionar os λ bits mais à esquerda do resumo completo l e descartar os demais.
3. Se a resistência à colisão é necessária, λ deve ter a resistência à colisão pelo menos duas vezes maior do que resistência à colisão s necessária (isto é, $\lambda \geq 2s$).

Truncar o resumo de mensagens pode impactar na segurança de uma aplicação. A resistência à colisão é reduzida de $l/2$ para $\lambda/2$ (em bits). A resistência à pré-imagem esperada é de λ bits e não l bits, independentemente da função de resumo utilizada. A resistência à segunda pré-imagem esperada depende do tamanho da mensagem.

A proposta do LMS [37], propõe o uso do SHA256-16 como um dos parâmetros para o esquema, que representa a função de resumo SHA-256 com o resumo final truncado retornando os 16 bytes mais à esquerda.

2.4.2 Assinaturas Digitais

A função de resumo é usada para mapear uma mensagem de qualquer tamanho para um resumo de tamanho fixo [46]. Em um esquema de assinatura digital, este resumo de mensagem é assinado por um algoritmo de assinatura. No esquema de assinatura de Merkle [39], por exemplo, a construção das funções para geração de chaves, assinatura e verificação do esquema, é baseada em funções de resumo.

Uma assinatura digital é usada para verificar quem assinou a mensagem e se a mensagem não foi alterada depois de assinada. Quando duas mensagens diferentes têm o mesmo resumo de mensagem, então uma colisão foi encontrada. Se isso acontecer, então uma assinatura digital não garante a autenticidade da mensagem assinada, porque as duas mensagens poderiam ser consideradas válidas. Portanto, uma função de resumo usada em assinaturas digitais requer resistência à colisão.

Em algumas situações específicas, resumos de mensagem truncadas são utilizados na geração de assinaturas digitais, como por exemplo na proposta de padrão do LMS. Em tais casos, a segurança depende do tamanho desses valores truncados, bem como da função de resumo utilizada.

Para aplicações de assinatura digital, a segurança do esquema de assinatura que usa uma função de resumo é, geralmente, a resistência à colisão da função de resumo e a segurança fornecida pelo algoritmo de assinatura. Quando a técnica de resumo aleatório, conforme descrito na próxima seção, é aplicada à mensagem antes de calcular o valor de resumo, a segurança pode ser maior do que a resistência à colisão da função de resumo.

Valor de Resumo Aleatório para Assinaturas Digitais

De acordo com o NIST [46], um *resumo aleatório* é uma técnica que embaralha as mensagens que serão utilizadas por uma função de resumo criptográfica durante a geração de assinaturas digitais. A mensagem terá uma assinatura digital diferente cada vez que é assinada, usando uma cadeia aleatória diferente. A intenção desse método é fortalecer a resistência à colisões fornecidas pelas funções de resumo criptográficas em aplicativos de assinatura digital, sem quaisquer alterações nas funções de resumo e nos algoritmos de assinatura digital.

Neste processo de geração do *resumo aleatório*, uma cadeia aleatória é utilizada para embaralhar uma mensagem M. A mensagem M é codificada com preenchimento de um único bit "1", seguido de zero ou mais bits "0". A cadeia aleatória pode ser adicionada a mensagem ou pode-se utilizar a função "ou exclusivo" ($\oplus$), para embaralhar a mensagem com a cadeia aleatória.

Quando se utiliza essa técnica, a função de resumo aleatória pode suportar um maior nível de segurança em relação a uma potencial vulnerabilidade, oferecendo maior proteção contra um ataque de colisão [46]. As propostas do LMS e do XMSS aplicam esta técnica nas versões mais recentes para aumentar o nível de segurança.

2.4.3 Códigos de Autenticação de Mensagem

Segundo definido pelo NIST [45], os códigos de autenticação de mensagem (MAC) fornecem autenticação e integridade dos dados. Esses códigos são construídos no contexto da criptografia simétrica.

Dois tipos de primitivas para construir MACs, que foram aprovados pelo NIST [45], são baseados em:

1. algoritmos de cifras de bloco, por exemplo o algoritmo do CMAC (*Cipher-based Message Authentication Code*) ;

2. funções de resumo, por exemplo o algoritmo do HMAC (*keyed-Hash Message Authentication Code*).

O algoritmo HMAC requer o uso de uma chave secreta que é compartilhada entre o emissor e o receptor da mensagem. A segurança proporcionada pelo algoritmo HMAC depende do tamanho da chave HMAC, da função de resumo utilizada e do tamanho da saída do código de autenticação da mensagem.

A proposta do XMSS aplicou a técnica HMAC para implementar um gerador de números pseudoaleatórios a partir da primeira versão. A partir da quarta versão, a proposta aplica a função de resumo diretamente para gerar os valores pseudoaleatórios, não utilizando mais o HMAC.

2.4.4 Funções de Derivação de Chaves

As funções de resumo podem ser utilizadas como blocos de construção de funções de derivação de chaves (*Key Derivation Function*–KDFs) [45]. As KDFs podem usar funções de resumo direta ou indiretamente em uma construção HMAC. As funções de derivação podem ser utilizadas para gerar chaves secretas.

As KDFs construídas a partir de funções de resumo especificados em [45] utilizam uma construção HMAC e podem ser utilizadas para derivar a chave secreta de uma chave existente, ou como um passo de expansão de chave no método de derivação de chave.

De acordo com o NIST [45], se uma chave derivada destina-se a fornecer uma segurança de b bits, então a segurança depende:

- do tamanho das chaves utilizadas,
- da resistência à pré-imagem da função de resumo, e
- do tamanho da chave derivada em bits.

2.4.5 Geração de Números Pseudoaleatórios

Um gerador de números pseudoaleatórios é um algoritmo que gera uma sequência de bits, que possuem algumas das propriedades dos números aleatórios. Esses geradores utilizam uma função pseudoaleatória (*Pseudo Random Function*–PRF) que pode ser construída a partir de funções de resumo ou de algoritmos de cifras de bloco [45]. Uma PRF é o elemento básico na construção de uma função de derivação de chave.

A função PRF descrita em [2], recebe como entrada uma semente $Seed$ de l bits e uma mensagem M de l bits. A semente $Seed$ é gerada a partir de um gerador aleatório seguro. A função devolve um valor pseudoaleatório x, que é calculado da seguinte forma $x = PRF(Seed, M)$. A função de resumo utilizada pela PRF deve ser selecionada de modo que possa proporcionar uma segurança que atenda ou exceda a segurança mínima necessária dos bits gerados.

Alguns geradores podem ser encontrados em [44]. As propostas de padrão do LMS [37] e do XMSS [23], especificam um gerador de números pseudoaleatório que é construído a partir de funções de resumo.

2.5 Segurança das Funções de Resumo

A segurança de uma função de resumo é determinada por sua resistência à colisão, resistência à pré-imagem ou resistência à segunda pré-imagem. Se uma aplicação requer mais de uma propriedade da função de resumo, então a segurança da aplicação considera a propriedade de segurança mais fraca.

2.5.1 Ataque de Aniversário

Este ataque implica um tamanho de saída mínimo necessário para uma função de resumo ser segura contra adversários que executam em um determinado tempo [32]. Para uma função de resumo H:$\{0,1\}^* \rightarrow \{0,1\}^l$, o ataque funciona da seguinte forma: Escolha q valores aleatórios distintos $\{x_1, \ldots, x_q \in \{0,1\}^{2l}\}$, compute $y_i = H(x_i)$, e verifique se existem dois valores iguais de y_i.

Se $q > 2^l$, então a probabilidade desse algoritmo encontrar uma colisão é 1. Porém, para valores menores que q, é mais difícil calcular a probabilidade exata. Então, neste caso, H é tratado como uma função aleatória. Ou seja, para cada i, assume-se que o valor $y_i = H(x_i)$ é uniformemente distribuído em $\{0,1\}^l$ e é independente de qualquer um dos valores de saída anteriores $\{y_i\}_{j<i}$ para distintos $\{x_i\}$.

Se escolher valores $\{y_1, \ldots, y_q \in \{0,1\}^l\}$ uniformemente ao acaso, qual é a probabilidade de existirem diferentes valores de i, j, onde $y_i = y_j$? Este problema de encontrar colisões é conhecido como o problema do ataque de aniversário [32]. Quando $q = \Theta(2^{l/2})$, a probabilidade de uma colisão é de aproximadamente $1/2$.

2.5.2 Nível de Segurança

Se o tamanho da saída de uma função de resumo é de l bits, então o ataque de aniversário encontra uma colisão com alta probabilidade executando $O(q) = O(2^{l/2})$ avaliações da função de resumo. A colisão pode ser encontrada no tempo $O(l \times 2^{l/2})$. Então, para a função de resumo ser resistente à ataques para encontrar colisão que executam em tempo T, o tamanho da saída da função de resumo precisa ser pelo menos $2logT$ bits [32].

Como exemplo, se uma função de resumo tem tamanho de saída de 128 bits, é possível encontrar uma colisão executando 2^{64} passos. Entretanto, se o tamanho da saída é de 256 bits, então são necessários 2^{128} passos para encontrar uma colisão. Assim, a existência de ataques genéricos de aniversário exige que qualquer função de resumo resistente à colisão precisa ter um tamanho de saída maior do que 128 bits [32].

2.5.3 Resumo da Segurança

De acordo com o NIST [46], a resistência à colisão esperada de qualquer função de resumo aprovada é, de um modo geral, metade do tamanho do valor do seu resumo.

A segurança dos algoritmos: SHA-224, SHA-256, SHA-384, e SHA-512 é apresentada em [46]. A resistência à segunda pré-imagem para SHA-384, SHA-512/224 e SHA-512/256 não depende apenas das próprias funções, mas também do tamanho das mensagens. Se

o tamanho das mensagens é pequeno, então a resistência à segunda pré-imagem é praticamente a mesma que a sua resistência à pré-imagem.

As quatro funções de resumo SHA-3 são alternativas à funções da família SHA-2, provendo um nível de segurança pelo menos igual ao da família SHA-2 contra os ataque à colisões, à primeira pré-imagem e à segunda pré-imagem [41].

As duas funções SHA-3 com saída variável (SHAKE128 e SHAKE256) foram projetadas para serem resistentes à colisão, à primeira pré-imagem e à segunda pré-imagem, e a outros ataque que sejam resistentes por uma função aleatória de mesmo tamanho, até o nível de segurança de 128 bits para o SHAKE128 e 256 bits para o SHAKE256. Essas funções, assim como uma função aleatória, não conseguem prover um nível de segurança maior que $d/2$ bits contra ataques baseados em colisões e d bits contra ataques de pré-imagem e de segunda pré-imagem

A segurança das funções de resumo SHA2 e SHA3 encontram-se resumidas na Tabela 2.1. Para o nível de segurança contra ataques à segunda pré-imagem em uma mensagem M, a função $L(M)$ é definida como $\lceil log_2(len(M)/B) \rceil$, onde B é o tamanho do bloco da função, por exemplo, 512 bits para SHA-1, SHA-224 e SHA-256 e 1024 bits para o SHA-512.

Tabela 2.1: Segurança das funções de resumo SHA2 e SHA3 [46]

		Segurança em Bits		
Função	**Tamanho da Saída**	**Colisão**	**Pré-imagem**	2^a **Pré-imagem**
SHA-224	224	112	224	min(224,256-$L(M)$)
SHA-256	256	128	256	256-$L(M)$
SHA-384	384	192	384	384
SHA-512	512	256	512	512-$L(M)$
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	min(d/2, 128)	$\geq$ min(d, 128)	min(d, 128)
SHAKE256	d	min(d/2, 256)	$\geq$ min(d, 256)	min(d, 256)

Capítulo 3

Esquema de Merkle e suas variantes

Neste capítulo são descritos os conceitos básicos do Esquema de Assinatura Digital de Merkle e suas variantes abordados neste trabalho. Este capítulo está organizado como segue: Seção 3.1 descreve as características dos esquemas de assinatura. Na Seção 3.2, são apresentadas as características dos esquemas OTS, o esquema OTS de Lamport-Diffie, o esquema OTS de Winternitz e a modificação do esquema de Winternitz. A Seção 3.3 mostra o esquema de Merkle e seus algoritmos. A Seção 3.4, descreve um algoritmo eficiente para calcular os caminhos de autenticação. As Seções 3.5 e 3.6 apresentam as variantes de Merkle que propõem a construção de multi árvore: CMSS e GMSS. As Seções 3.7 e 3.8 descrevem o XMSS e o XMSS-MT. Por fim, na Seção 3.9, é apresentada a segurança desses esquemas.

3.1 Esquemas de Assinatura Digital

A assinatura digital fornece uma prova inegável de que a mensagem foi assinada pelo detentor da chave privada. Para isto, uma assinatura digital deve ter as seguintes propriedades: autenticidade, integridade e não-repúdio.

Os esquemas de assinatura digital abordados neste trabalho são baseados em funções de resumo. Esquemas de assinatura digital baseados em funções de resumo tornaram-se populares com o trabalho de Merkle em 1979 [39]. Esquemas de assinatura baseados em funções de resumo beneficiam-se das propriedades dessas funções para gerar esquemas de assinatura seguros.

Um esquema de assinatura digital `SIGN` é uma tripla de algoritmos (`KG`; `SIG`; `VER`) onde, `KG` é um algoritmo de geração de chaves, `SIG` é um algoritmo de assinatura e `VER` é um algoritmo de verificação de assinatura.

Seja `PK` a chave pública e `SK` a chave privada. A tripla de algoritmos que compõem um esquema de assinatura digital satisfazem as seguintes propriedades:

1. O algoritmo de geração de chaves `KG` recebe como entrada um parâmetro de segurança 1^l e produz um par de chaves (sk, pk), onde sk é a chave privada e pk é a chave pública.

2. O algoritmo de geração de assinatura `SIG` recebe como entrada uma mensagem

$M \in \{0,1\}^*$ e uma chave privada sk e produz uma assinatura sig, denotada por $sig \leftarrow \texttt{SIG}(sk, M)$.

3. O algoritmo de verificação de assinatura `VER` recebe como entrada uma mensagem M, uma assinatura sig de M e uma chave pública pk e devolve 1 se a assinatura for válida ou 0 se assinatura for inválida.

A Figura 3.1 representa um esquema de assinatura digital. Um esquema de assinatura digital permite que um emissor assine uma mensagem M com uma chave privada (sk), de forma que uma outra parte que conhece a chave pública pk, pode verificar que a mensagem não foi modificada.

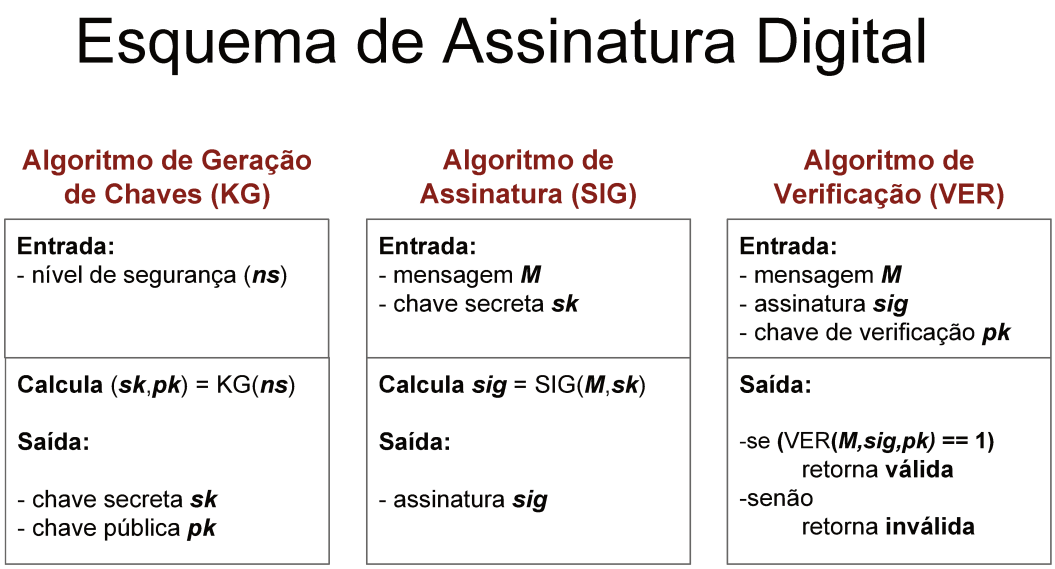


Figura 3.1: Assinatura Digital

3.2 Esquema de Assinatura One-Time

Um esquema OTS é um esquema que utiliza um par de chaves diferente para cada assinatura. Esse esquema é considerado seguro se uma chave privada (sk) é usada para assinar uma única mensagem. Assinaturas One-Time apareceram inicialmente nos trabalhos de Lamport [33]. As OTSs são assinaturas digitais baseadas em funções de resumo. A segurança desses esquemas depende da segurança da função de resumo utilizada, assim se uma função de resumo torna-se insegura, esta função pode ser trocada por qualquer outra função de resumo segura.

3.2.1 O Esquema LDOTS

Lamport e Diffie criaram um esquema de assinatura one-time Lamport-Diffie (*Lamport-Diffie One-Time Signature*–LDOTS) em 1979. Seja l (um inteiro positivo) o parâmetro de segurança do esquema LDOTS. O esquema LDOTS utiliza uma função unidirecional

$$f : \{0,1\}^l \to \{0,1\}^l,$$

e uma função de resumo

$$g : \{0,1\}^* \to \{0,1\}^l.$$

Geração do par de chaves do esquema LDOTS.

A chave privada de assinatura sk do esquema LDOTS é escolhida aleatoriamente e consiste de $2l$ cadeias binárias de tamanho l. A chave privada de assinatura é escolhida aleatoriamente

$$sk = (x[0][0], x[1][0], \ldots, x[l-1][0], x[0][1], x[1][1], \ldots, x[l-1][1]) \in \{0,1\}^{(l,2l)}.$$

A chave de verificação pk do esquema LDOTS é

$$pk = (y[0][0], y[1][0], \ldots, y[l-1][0], y[0][1], y[1][1], \ldots, y[l-1][1]) \in \{0,1\}^{(l,2l)},$$

onde $y[i][j] = f(x[i][j])$, $0 \leq i \leq l-1$, $j = \{0,1\}$.

Geração de Assinatura do esquema LDOTS.

Dada uma mensagem M e a chave privada $x[i][j]$ com $0 \leq i \leq l-1$ e $j = \{0,1\}$, a assinatura da menagem M é calculada da seguinte forma, primeiro calcula o resumo de uma mensagem $r = g(M) = (m[0], \ldots, m[l-1])$ com $m[i] \in \{0,1\}$. Então compara, se $m[i] = 0$ então $sig[i] = x[i][0]$ caso contrário, $sig[i] = x[i][1]$. A assinatura do esquema LDOTS é

$$sigOts = (sig[0], sig[1], \ldots, sig[l-1]) = (x[0][m[0]], x[1][m[1]], \ldots, x[l-1][m[l-1]]).$$

Verificação de Assinatura do esquema LDOTS.

Para verificar a assinatura $sigOts = (sig[0], sig[1], \ldots, sig[l-1])$, da mensagem M, primeiro calcula o resumo da mensagem $r = g(M) = (m[0], \ldots, m[l-1])$. Então, calcula $f(sig[i])$ para cada elemento $0 \leq i \leq l-1$.

Se $m[i] = 0$ então compara $f(sig[i]) = y[i][0]$ caso contrário, se $m[i] = 1$ então compara $f(sig[i])$=$y[i][1]$. Ao final checa se

$$(f(sig[0]), \ldots, f(sig[l-1])) = (y[0][m[0]], \ldots, y[l-1][m[l-1]]).$$

Se todos os valores são iguais, então a assinatura é válida, caso contrário é inválida.

A Figura 3.2 apresenta um exemplo do esquema LDOTS. Neste exemplo, a função unidirecional é $f(x) = (x + 1)\ mod\ 2^l$ e o tamanho da saída da função é $l = 4$ bits. As chaves sk e pk e a assinatura tem $2l$ elementos com tamanho de l bits.

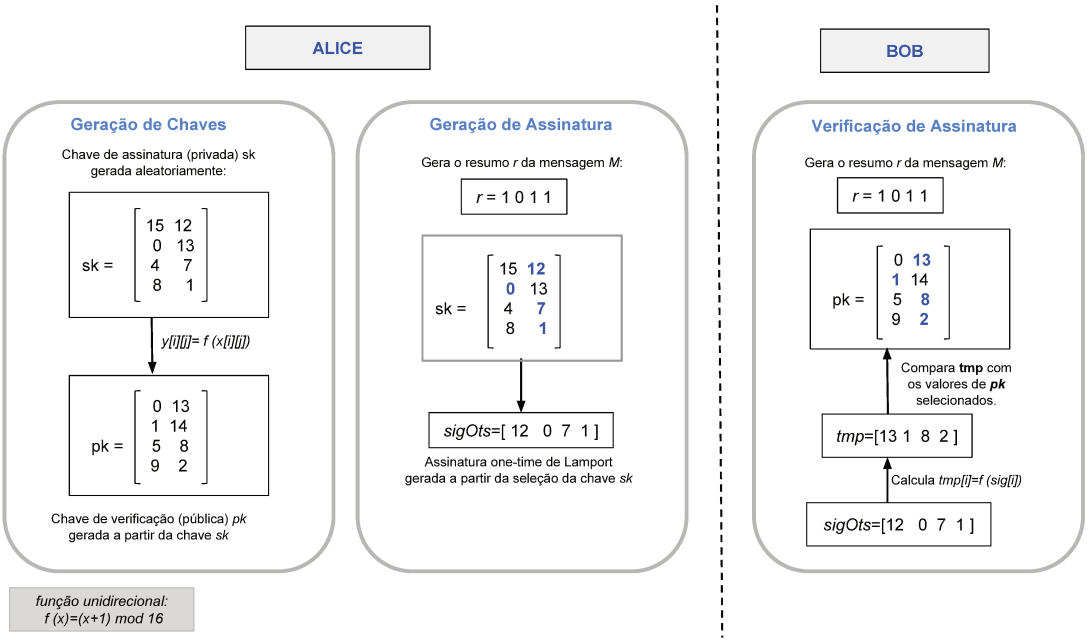


Figura 3.2: Esquema de Lamport

3.2.2 O Esquema WOTS

O Esquema de Assinatura de Winternitz (*Winternitz One-Time Signature*–WOTS) [39] é uma modificação do esquema LDOTS [33]. O esquema WOTS usa um parâmetro w que é o número de bits a serem assinados simultaneamente. Este esquema produz assinatura de tamanho menor do que a assinatura de Lamport, mas aumenta o número de avaliações da função unidirecional de 1 para $2^w - 1$, para cada elemento da chave privada de assinatura. O parâmetro w determina o valor de t, que é o número de elementos da assinatura. Quanto maior o w, menor o tamanho da assinatura e maior o tempo para geração e verificação de assinatura.

O esquema WOTS usa uma função unidirecional $f : \{0,1\}^l \to \{0,1\}^l$ e uma função de resumo criptográfica resistente à colisão $g : \{0,1\}^* \to \{0,1\}^l$, onde l é um inteiro positivo. A função de encadeamento f^e calcula e iterações de f sobre a entrada $x \in \{0,1\}^l$ para $e \in \mathbb{N}$. A função de encadeamento é definida recursivamente como:

$$f^e(x) = \begin{cases} x & \text{se } e = 0; \\ f(f^{e-1}(x)) & \text{se } e > 0. \end{cases}$$

Geração do par de chaves do esquema WOTS

Dado um parâmetro Winternitz $w \in \mathbb{N}$, o algoritmo usa $t = t_1 + t_2$, onde $t_1 = \lceil l/w \rceil$, $t_2 = \lceil (\lfloor log_2 t_1 \rfloor + 1 + w)/w \rceil$. A chave privada $sk = (k[0], ..., k[t-1])$ é um vetor de

tamanho t, onde cada elemento $k[i]$ é uma cadeia de l bits e só pode ser utilizada para assinar somente uma mensagem. A chave de verificação pk é calculada como:

$$pk = (u[0], ..., u[t-1]) \qquad onde \qquad u[i] = f^{2^w-1}(k[i]).$$

Geração de assinatura do esquema WOTS

Dada uma mensagem M e a chave privada sk, a assinatura da menagem M é calculada da seguinte forma, primeiro calcula $r = g(M)$, onde r é dividido em t_1 blocos binários de tamanho w, resultando em $r = (m[0]||...||m[t_1 - 1])$. Os $m[i]$ blocos são representados como inteiros em $\{0, 1, ..., 2^w - 1\}$.

A soma de verificação é utilizado para assegurar que qualquer tentativa de falsificação de uma assinatura seja detectada. A soma de verificação é calculado como $c = \sum_{i=0}^{t_1-1}(2^w - m[i])$. A cadeia c pode ser dividida em t_2 blocos: $c = (c[0]|| \ldots ||c[t_2 - 1])$ de tamanho w. Seja $b = r||c$ a concatenação da cadeia r com a cadeia c. Então,

$$b = (b[0]||b[1]|| \ldots ||b[t-1]) = (m[0]|| \ldots ||m[t_1 - 1]||c[0]|| \ldots ||c[t_2 - 1]).$$

O valor de b pode ser visto como uma sequência de valores de tamanho w bits. Esses valores determinam o número de vezes que a função f é aplicada a cada elemento de k. Logo, a assinatura é calculada como:

$$sigOts = (sig[0], ..., sig[t-1]) = (f^{b[0]}(k[0]), \ldots, f^{b[t-1]}(k[t-1])).$$

Verificação de Assinatura do esquema WOTS

Para verificar a assinatura $sigOts = (sig[0], ..., sig[t-1])$ da mensagem M o verificador deve completar a série de aplicações da função f utilizando os w bits valores do resumo da mensagem e da soma de verificação correspondente. Então, calcula-se $(b[0], ..., b[t-1])$ da mesma forma como na geração de assinatura.

Depois, completa-se as aplicações da função:

$$sig' = (tmp[0], \ldots, tmp[t-1]) = (f^{2^w-1-b[0]}(sig[0]), \ldots, f^{2^w-1-b[t-1]}(sig[t-1])).$$

Se $tmp[i] = u[i]$ para $i = \{0, 1, ..., t-1\}$, então a assinatura é válida. Caso contrário, é inválida.

A Figura 3.3 apresenta um exemplo do esquema WOTS com $w = 4$ para os algoritmos de geração de chaves, assinatura e verificação.

3.2.3 O Esquema WOTS$^+$

Hülsing [25] propôs o esquema WOTS$^+$, que é uma modificação do esquema WOTS. O esquema WOTS$^+$ usa uma família de funções de resumo $F = \{f_K : \{0,1\}^l \to \{0,1\}^l | K \in \{0,1\}^l\}$. Neste esquema, a função de encadeamento f^e usa vetores de máscara de bits para aumentar a segurança do esquema.

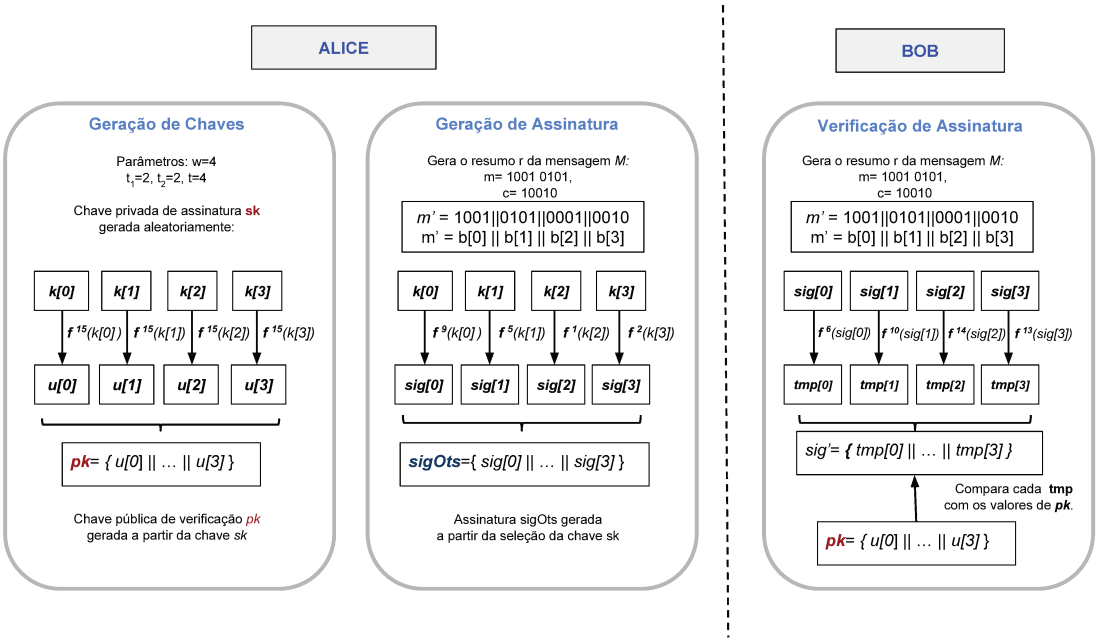


Figura 3.3: Esquema de Winternitz

Seja $f_K \in F$, então a função de encadeamento f^e calcula e iterações de f_K sobre a entrada $x \in \{0,1\}^l$ e um vetor de máscara de bits $bm = (bm[1], ..., bm[w-1])$ onde $e \in \mathbb{N}$. A função de encadeamento é definida recursivamente como:

$$f^e(x, bm) = \begin{cases} x & \text{if } e = 0; \\ f_K(f^{e-1}(x, bm) \oplus bm[e]) & \text{if } e > 0. \end{cases}$$

O esquema WOTS$^+$ é parametrizado por um parâmetro l e o parâmetro de Winternitz $w \in \mathbb{N}$, para $w > 1$. Os parâmetros l e w são usados para calcular: $t_1 = \lceil (l)/(log_2(w)) \rceil$, e $t_2 = \lfloor (log_2(t_1(w-1)))/(log_2(w)) \rfloor + 1$, $t = t_1 + t_2$. Esses parâmetros são públicos.

Geração do Par de Chaves do esquema WOTS$^+$

As chaves privadas $sk = (k[0], \ldots, k[t-1])$, podem ser calculadas por um gerador aleatório ou por um gerador pseudoaleatório. A chave pública de verificação é:

$$pk = (u[0], \ldots, u[t-1]) = (f^{w-1}(k[0], bm), \ldots, f^{w-1}(k[t-1], bm)).$$

Geração de Assinatura do esquema WOTS$^+$

O algoritmo de geração de assinatura primeiro calcula o resumo da mensagem $r = g(M)$ de l bits. O resumo é então representado em base w: $r = g(M) = (m[0], \ldots, m[t_1 - 1])$, $m[i] \in \{0, ..., w-1\}$. Depois, o valor da soma de verificação $C = \sum_{i=0}^{t_1-1}(w - 1 - m[i])$ (tamanho t_2) também é representado em base w e e adicionado a r, resultando em $b = (b[0], ..., b[t-1])$. O tamanho da assinatura e da chave privada são t cadeias de l bits. A assinatura é:

$$sigOts = (sig[0], ..., sig[t-1]) = (f^{b[0]}(k[0], bm), \ldots, f^{b[t-1]}(k[t-1], bm)).$$

Verificação de Assinatura do esquema WOTS$^+$

Para checar a assinatura, o verificador calcula os valores $b = (b[0], ..., b[t-1])$ como na assinatura. Se

$$(f^{w-1-b[0]}(sig[0], bm), \ldots, f^{w-1-b[t-1]}(sig[t-1], bm)) = (u[0], \ldots, u[t-1]),$$

então a assinatura é válida.

3.3 Esquema de Merkle (MSS)

Os esquemas de assinatura one-time são inadequados para a maioria das aplicações práticas porque cada par de chaves só pode ser usado para uma única assinatura [6]. Merkle [39] propôs uma solução para este problema. O Esquema de Assinatura Digital proposto por Merkle (MSS) é um método para assinar um grande (mas fixo) número de mensagens. Este método reduz a validade de um número arbitrário (mas fixo) de chaves de verificação one-time para a validade de uma única chave pública: a raiz de uma árvore binária. O esquema de Merkle trabalha com uma função de resumo criptográfica $g : \{0,1\}^* \longrightarrow \{0,1\}^l$ e com um esquema de assinatura one-time. O MSS é flexível por permitir que seja escolhido qualquer esquema one-time seguro.

O MSS constrói uma árvore binária onde cada chave privada de assinatura e verificação one-time está associada a uma folha da árvore. Cada folha contém o valor do resumo da chave de verificação one-time. A Figura 3.4 mostra a construção da árvore binária de Merkle. O valor contido na raiz da árvore é a chave pública do MSS. Uma árvore com altura h e 2^h folhas terá 2^h pares de chaves públicas e privadas OTS. Denota-se o nó i, na altura j por $no[i][j]$ para $i = \{0, \ldots, 2^h - 1\}$ e $j = \{0, \ldots, h-1\}$.

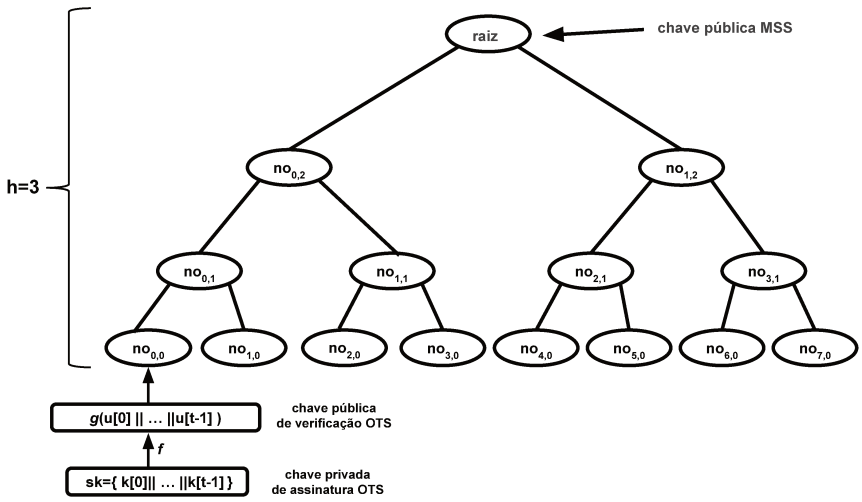


Figura 3.4: Construção da árvore de Merkle adaptada de [6]

3.3.1 Geração da Chave Privada do MSS

Uma chave privada do MSS consiste de 2^h chaves privadas OTS. O índice do nó é inicializado com zero para gerar as chaves privadas [39]. Uma PRF pode ser utilizada para gerar os elementos da chave privada. Utilizar a PRF na geração das chaves privadas permite reduzir o custo de armazenamento das chaves privadas, pois somente a semente para a PRF será armazenada. As chaves podem ser recuperadas quando necessário a partir da semente utilizada.

Para cada folha da árvore do MSS, os t elementos da chave secreta da OTS são calculados como $k[i] = PRF(Seed, i)$ $(0 \leq i \leq t-1)$. $Seed$ é uma semente gerada aleatoriamente e i é o índice da chave k. As chaves pk são calculada através da chamada ao algoritmo de geração de chaves OTS.

3.3.2 Geração da Chave Pública do MSS

A chave pública do MSS (pub) está associada a 2^h pares de chaves OTS. O Algoritmo 4 (treehash), descreve a geração da chave pública (pub). O algoritmo recebe como entrada o índice inicial da folha s e a altura da árvore h. Cada folha da árvore, recebe o resultado da aplicação da função de resumo na concatenação dos valores da chave de verificação $g(u[0]||...||u[t-1])$. Cada nó intermediário recebe o resultado da aplicação da função de resumo na concatenação dos valores dos nós de seus dois filhos, esquerdo e direito.

O algoritmo usa um vetor de estrutura $vetorNo$, que armazena um nó intermediário em cada altura e uma estrutura auxiliar $noAtual$. As estruturas $vetorNo$ e $noAtual$ são compostas de:

- um valor do resumo $hash$;
- a altura do nó $altura$;
- o índice do nó $indNo$.

Quando um novo nó $noAtual$ é calculado, o algoritmo verifica se o último nó do vetor $vetorNo$ tem altura igual do $noAtual$. Se as alturas forem iguais, os dois nós são concatenados, e um novo nó será calculado através do resumo da concatenação desses nós. O algoritmo termina quando a raiz da árvore é encontrada.

3.3.3 Caminho de Autenticação

Para que o emissor possa prover a autenticidade de cada assinatura, ele precisa incluir um caminho de autenticação na assinatura. Na Figura 3.5 observa-se a ordem em que os nós da árvore foram gerados através da execução do Algoritmo 4.

Os nós em cinza representam o primeiro caminho de autenticação salvo durante a geração da chave pública do MSS. Esse caminho de autenticação é composto pelos valores dos nós em cada altura, onde esses nós são os irmãos dos nós no caminho de autenticação que ligam a folha até a raiz do MSS. O uso do caminho de autenticação reduz a necessidade de enviar toda a árvore para o receptor.

Algorithm 4 Geração da chave pública do MSS (treehash) [39]

Entrada: a altura h, índice do nó inicial s.
Saída: a chave pública pub.

$indvet = 0$; //índice do vetor $vetorNo$
for ($indSig = s$; $indSig < 2^h$;$indSig = indSig + 1$) **do**
  $noAtual.altura = 0$; //altura de in
  $noAtual.indNo = indSig$; //índice do nó
  gerar sk e pk para folha $indSig$;
  $noAtual.hash = g(u[0]||...||u[t-1])$;
  while (($noAtual.altura == vetorNo[indvet-1].altura$) AND ($indvet! = 0$)) **do**
    $indvet--$;
    $tmp = vetorNo[indvet].hash$;
    $noAtual.hash$=$g(tmp||noAtual.hash)$;
    $noAtual.altura++$;
    $noAtual.indNo = (noAtual.indNo)/2$;
  end while
  $vetorNo[indvet]$=$noAtual$;
  $indvet++$;
end for
return $pub = vetorNo[indvet]$;

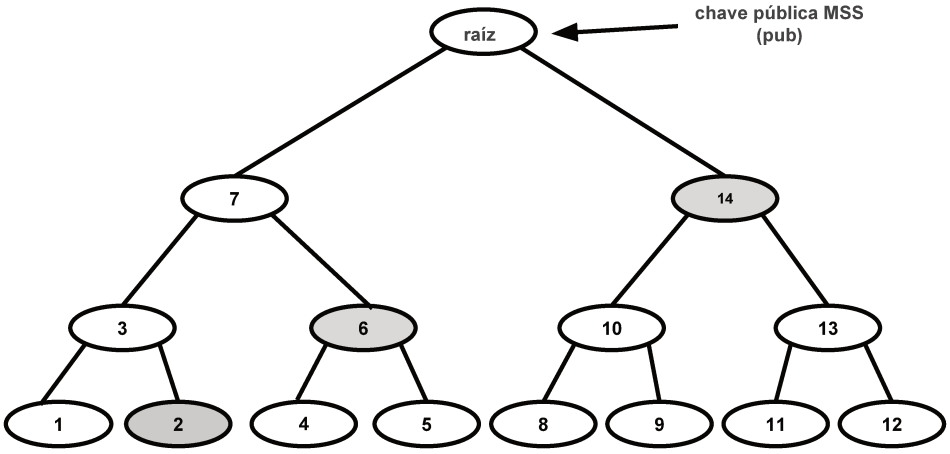


Figura 3.5: Geração da chave pública da árvore de Merkle adaptada de [6]

O pseudocódigo no Algoritmo 5, pode ser executado para guardar o primeiro caminho de autenticação $Aut = \{aut[0], ..., aut[h-1]\}$, toda vez que um nó *noAtual* for gerado pelo Algoritmo 4. O primeiro caminho de autenticação (aut) é salvo em toda altura j (para $0 \leq j \leq h$), quando o índice do nó for igual a 1. Um vetor auxiliar (aux) guarda os valores dos nós esquerdos no caminho de autenticação que serão utilizados nas próximas assinaturas.

Algorithm 5 Geração do primeiro caminho de autenticação [39]

Entrada: o nó gerado *noAtual*.
Saída: os vetores aut e aux.

```
i = noAtual.indNo;
j = noAtual.altura;
if i == 1 then
    aut[j] = no[i][j];
else
    if i == 0 then
        aux[j] = no[i][j];
    end if
end if
```

3.3.4 Geração de Assinatura do MSS

O MSS permite a geração de 2^h assinaturas, para uma árvore de altura h. A geração de assinatura consiste de dois passos: primeiro, a assinatura $sigOts$ do resumo da mensagem $r = g(M)$ é gerada utilizando um esquema OTS com a respectiva chave privada sk, correspondente a uma folha s. No segundo passo, o caminho de autenticação para a próxima folha da árvore é preparado. Esse caminho é calculado de forma eficiente, pois somente computa os nós do caminho de autenticação que mudam na próxima assinatura.

O Algoritmo 6 mostra os dois passos do processo de geração de uma assinatura. O Algoritmo recebe como entrada a altura da árvore e a chave privada sk e retorna a assinatura de Merkle $sig = (s, sigOts, pk, (aut[0], ..., aut[h-1]))$.

A assinatura do MSS contém o índice do nó s, a assinatura OTS $sigOts$, a chave de verificação pk e o caminho de autenticação aut.

O Algoritmo 6 atualiza os nós do caminho de autenticação através da chamada ao pseudocódigo ($treehash(indNo, j)$) do Algoritmo 4, passando como entrada o índice do nó inicial $indNo$ e altura j selecionados. Depois de 2^j execuções, o valor do nó na altura j está pronto e é armazenado no vetor $aux[j]$.

3.3.5 Verificação de Assinatura do MSS

De acordo com o método em [6], o processo de verificação de assinatura consiste de duas etapas: na primeira etapa, a assinatura $sigOts$ é verificada utilizando-se a chave de verificação one-time pk e o respectivo esquema one-time. Na segunda etapa, é preciso validar a chave pública da árvore de Merkle. Para isso, o receptor calcula o caminho de autenticação, construindo o caminho $(p[0], ..., p[h])$ do nó atual $h = 0$ até a raiz. O

Algorithm 6 Geração de assinatura do MSS [39]

Entrada: a altura h, a chave sk.
Saída: a assinatura sig.

```
for (i = 0, i < 2^h, i++) do
    o Algoritmo OTS gera sigOts utilizando a chave sk;
    for (j = 0, j < h, j++) do
        if (i + 1)/(2^j) == 0 then
            aut[j] = aux[j];
            indNo = (i + 1 + 2^j) ⊕ 2^j;
            aux[j] = treehash(indNo, j);
        end if
    end for
    return sig = (s, sigOts, pk, (aut[0], ..., aut[h − 1]));
end for
```

índice do nó i é utilizado para decidir em qual ordem o caminho de autenticação será reconstruído. Inicialmente, para a folha de índice i, $p[0] = g(pk)$. Para $j = 1, 2, \ldots, h$ executa-se a condição a seguir para calcular o valor resumo da sequência dos nós em cada altura:

$$p[j] = \begin{cases} g(aut[j-1] \| p[j-1]) & \text{if } \lfloor i/(2^{j-1}) \rfloor \equiv 1 \bmod 2; \\ g(p[j-1] \| aut[j-1]) & \text{caso contrário.} \end{cases}$$

Ao final, compara-se o valor de $p[j]$ com a chave pública conhecida pub. Se o valor for igual, a assinatura é válida.

3.4 Algoritmo BDS

Em 2008 [4], foi proposto o Algoritmo BDS, que calcula os caminhos de autenticação de forma mais eficiente. O maior custo na geração de uma assinatura Merkle é determinado pelo tempo gasto para calcular o caminho de autenticação. O cálculo do caminho de autenticação depende dos nós a serem atualizados no caminho para cada assinatura, assim para algumas assinaturas esse cálculo é lento e para outras pode ser muito rápido.

Como no algoritmo de Szydlo [56], o algoritmo BDS possui duas estratégias diferentes para calcular os nós de autenticação, dependendo se o nó filho é um nó esquerdo ou direito. A diferença do algoritmo de Szydlo é que são agendados somente o cálculo das folhas e não dos nós internos da árvore. A ideia do algoritmo é equilibrar o número de folhas que são calculadas em cada execução do algoritmo, uma vez que este cálculo é um dos mais caros na geração da árvore. Os nós internos são calculados quando necessário, pois este cálculo é insignificante em comparação com os custos para se gerar um nó folha da árvore binária. O algoritmo BDS mantém um estado (*state*) que armazena as estruturas de dados atualizadas.

Estruturas de dados

O Algoritmo 7 utiliza as seguintes estruturas de dados para uma árvore de altura h:

- $aut[i]$, $i = 0, ..., h-1$: vetor de nós que armazena o caminho de autenticação atual.
- $retain$: único nó de autenticação direito na altura $h-2$.
- $stack$: pilha de nós com as operações *empilha* e *desempilha*.
- $keep[h]$, $h = 0, ..., h-2$: vetor que armazena determinados nós para o cálculo eficiente dos nós de autenticação esquerdos.
- $treehash[j]$, $j = 0, ..., h-3$: instâncias do algoritmo *treehash* que compartilham a pilha *stack*. Cada campo da estrutura treehash tem os seguintes métodos:

  $treehash[j].no$. Armazena um único nó cauda que é guardado na execução Algoritmo 4 (treehash).

  $treehash[j].initialize(s)$. Este método marca a folha dessa instância que será iniciada em treehash.

  $treehash[j].height$. Esta entrada marca a menor altura da instância $treehash[j]$ a ser calculada.

  $treehash[j].update(s)$. Esse método executa Algoritmo 4, a partir da folha marcada em $treehash[j].initialize(s)$ para a altura marcada em $treehash[j].height$.

Inicialização do Algoritmo

A inicialização do algoritmo é feita durante a geração do par de chaves do MSS, quando é armazenado o caminho de autenticação para a primeira folha ($s = 0$): $aut[i] = no[i][1]$, $i = 0, ..., h-1$.

Para $j = 0, ..., h-3$, são armazenados os nós *treehash*: $treehash[j].no = no[j][3]$, e o próximo nó de autenticação direito na altura $h-2$: $retain = no[h-2][3]$.

Versão Simplificada do Algoritmo

No Algoritmo 7 é descrita a versão simplificada descrita por [4] do algoritmo BDS. O algoritmo recebe como entrada o índice do nó atual $s \in \{0, ..., 2^h - 2\}$, a altura da árvore de Merkle $h \geq 2$, onde h deve ser par, e o estado *state* do algoritmo com as estrutura de dados (aut, $keep$, $retain$, e $treehash$), que foram atualizadas na assinatura anterior ou na inicialização. O algoritmo calcula o caminho de autenticação para a próxima folha $s+1$ e atualiza as estruturas de dados do *state*.

3.5 O CMSS

O CMSS [7] é um aperfeiçoamento do MSS (*An Improved Merkle Signature Scheme*), é uma variante do MSS que permite aumentar a quantidade de assinaturas de 2^{20} para 2^{40}, diminuindo os tempos de geração de chaves e de geração de assinatura.

No CMSS são construídas duas árvores, uma subárvore (no nível mais baixo) e uma árvore principal (no nível mais alto), cada uma com $2^{h/2}$ folhas, para altura $h/2$. Para

Algorithm 7 Atualização do caminho de Autenticação [4]

Entrada: folha s, altura $h \geq 2$, e o estado $state$.
Saída: caminho de autenticação para a folha $s+1$.

if (s%2==0) **then**//s é um nó esquerdo
  $Tau = 0$;
else
  $Tau = max\{h : 2^h | (s+1)\}$;
end if
//Se o pai da folha s na altura $(Tau+1)$ é um nó esquerdo
if ($((s/2^{Tau+1})\%2 == 0)$ AND $(Tau < (h-1))$) **then**
  $keep[Tau] = aut[Tau]$;
end if
if $(Tau == 0)$ **then**
  $aut[0] = Leafcalc(s)$;
else
  $aut[Tau] = g(aut[Tau-1]||keep[Tau-1])$;
  for $(i = 0; i <= (Tau-1); i++)$ **do**
    if $(i <= (h-3))$ **then**
      $aut[i] = treehash[i]$;
    elseif($i == h-2$)
      $aut[i] = retain[i]$;
    end if
  end for
  //*treehash* instâncias devem ser inicializadas
  for $(i = 0; i <= (Tau-1); i++)$ **do**
    $indNo = (s + 1 + 3 * (1 << i))$;
    if $(indNo < 2^h)$ **then**
      $treehash[i].initialize(indNo)$;
    end if
  end for
end if
//atualiza $(h/2-1)$ instâncias *treehash*
for $(l = 0; l < (h/2-1); l++)$ **do**
  $x = min\{i : treehash[i].height = min_{j=0,\ldots,h-3} treehash[j].height\}$;
  //as instâncias *treehash* com índice x recebem uma atualização:
  $treehash[x].update()$;
end for
return $aut[0], \ldots, aut[h-1]$;

gerar 2^h chaves de assinatura, duas árvores com altura $2^{h/2}$ são geradas no CMSS. Diferentemente, no MSS, uma única árvore com 2^h nós é construída, levando a um tempo de execução bem maior.

CMSS tem as seguintes características: usa o esquema de assinatura one-time de Winternitz, usa o algoritmo de Szydlo [55] para calcular os caminhos de autenticação e uma PRF [44] para reduzir o tamanho da chave privada.

A figura 3.6 apresenta a construção do CMSS. A chave pública CMSS é a raiz da árvore principal. Os dados são assinados com as folhas da subárvore, mas a raiz dessa subárvore não é chave pública. Esta raiz é autenticada por uma assinatura do MSS referente a uma folha da árvore principal. Após as 2^h primeiras assinaturas serem geradas, uma nova subárvore é construída e utilizada para gerar as próximas 2^h assinaturas. A subárvore atual é chamada de subárvore existente e a próxima subárvore é chamada de subárvore desejada.

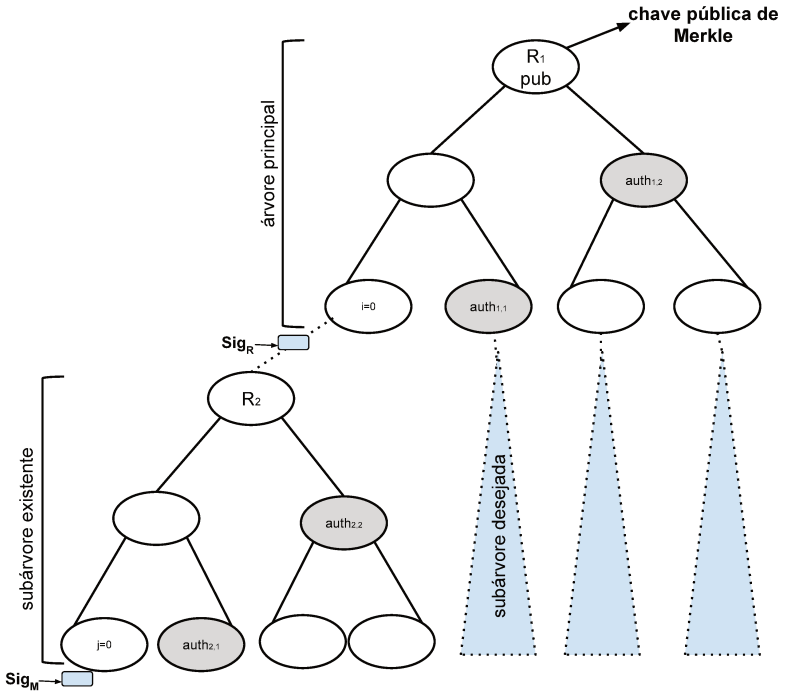


Figura 3.6: A construção do CMSS adaptada de [7]

3.5.1 Geração de Chaves do CMSS

Para gerar o par de chaves, o algoritmo de geração de chaves do MSS é executado duas vezes. Inicialmente, a primeira subárvore existente e seu primeiro caminho de autenticação são gerados. Então, a árvore principal e seu primeiro caminho de autenticação também são calculados. A chave privada do CMSS possui dois índices que são os índices das folhas das subárvores principal e existente (i e j), a raiz da subárvore existente ($R[2]$), três sementes para a PRF, três caminhos de autenticação, e três vetores de caminho de autenticação (da árvore principal, da subárvore existente e da subárvore desejada respectivamente).

3.5.2 Geração de Assinatura do CMSS

A geração de uma assinatura do CMSS é realizada em quatro partes. Primeiro, a assinatura $sigOts[M]$ do resumo da mensagem $r = g(M)$ é calculada usando uma chave sk da folha da subárvore existente e o respectivo esquema WOTS. A seguir, a assinatura $sigOts[R]$ da raiz da subárvore é calculada usando uma chave sk da folha árvore principal. Então, a próxima subárvore (subárvore desejada) é parcialmente construída, onde um novo nó folha é gerado. Finalmente, as chaves privadas das duas subárvores são atualizadas para a próxima assinatura juntamente com o caminho de autenticação de cada subárvore. Cada vez que uma nova assinatura do CMSS é gerada, a assinatura da raiz da subárvore atual é recalculada.

3.5.3 Verificação de Assinatura do CMSS

A verificação de assinatura do CMSS é realizada em dois passos. Primeiro, os dois caminhos de autenticação (da subárvore existente e da subárvore principal) validados. Então a validade das duas assinaturas one-time $sigOts[M]$ e $sigOts[R]$ são verificadas. Isto dobra o tempo de verificação da assinatura por ter que verificar a validade de duas subárvores.

3.6 O GMSS

O esquema de Merkle com capacidade de assinaturas virtualmente ilimitada (*Merkle Signatures with Virtually Unlimited Signature Capacity*–GMSS) foi publicado em 2007 [3]. No GMSS, os autores propõem uma alteração no esquema de assinatura Merkle que permite assinar 2^{80} mensagens utilizando um único par de chaves. O GMSS foi utilizado para projetar um protocolo cliente servidor para servidores web usando *SSL/TLS* que minimiza a latência e aumenta a resistência à ataques de negação de serviço.

O GMSS consiste de uma árvore com d camadas (subárvores), onde cada subárvore é uma árvore de Merkle. Subárvores em camadas diferentes podem ter alturas diferentes. Seja h_i a altura de uma subárvore na camada i para $i = 1, \ldots, d$. Denota-se $\tau_{1,0}$ a única árvore na camada 1, no nível mais acima. As $2^{h_1+\ldots+h_{i-1}}$ árvores de Merkle nas camadas $i = 2, \ldots, d$ são denotadas por $\tau_{i,j}, j = 0, \ldots, 2^{h_1+\ldots+h_{i-1}} - 1$ numerando da esquerda para a direita.

O número total de assinaturas que podem ser gerados com o GMSS é $2^{h_1+\ldots+h_d}$. A Figura 3.7 ilustra a construção do GMSS. Uma árvore na camada i é pai da árvore na camada $i + 1$. A raiz de uma árvore filha é assinada com uma folha (chave privada de assinatura one-time) da sua árvore pai. $Raiz[\tau]$ denota a raiz da árvore τ. $sigOts[\tau]$ denota a assinatura one-time da $Raiz[\tau]$, que é gerado usando a folha l do pai de τ. Para cada assinatura existe um único caminho de autenticação que é composto de todos os caminhos de autenticação em cada subárvore Merkle. O GMSS usa o esquema $WOTS$ [39]. Diferentes camadas podem ter valores de parâmetros de Winternitz diferentes. Assim, o conjunto de parâmetros do GMSS é: $P = (d, (h1, \ldots, h_d), (w_1, \ldots, w_d))$.

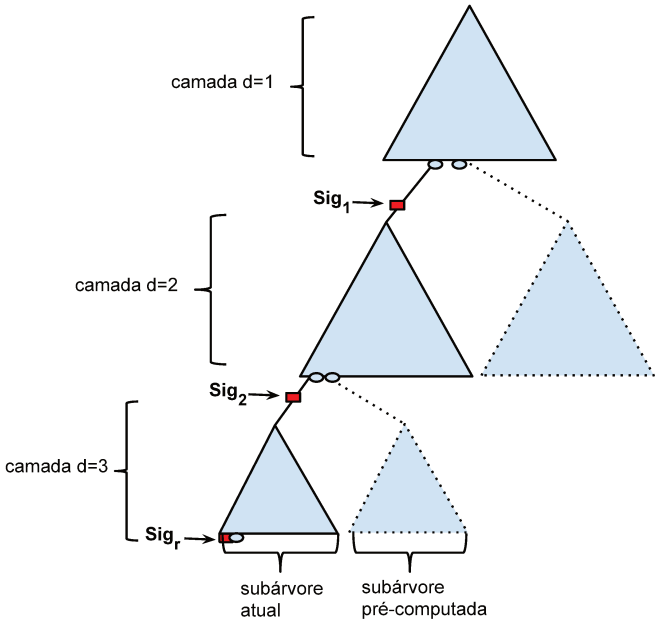


Figura 3.7: Construção do GMSS adaptada de [3]

3.6.1 Geração de Chaves do GMSS

Para cada subárvore, o algoritmo de geração de chaves do esquema WOTS gera as chaves de assinatura e o Algoritmo treehash (Algoritmo 4), calcula as raízes das árvore $Raiz[\tau_{i,0}]$, $i = 1, \ldots, d$. Os primeiros caminhos de autenticação de cada subárvore são salvos durante a geração das raízes.

As assinaturas $sigOts[\tau]$ das d subárvores Merkle, que serão usadas para a primeira assinatura, são calculadas. Como os valores mudam com menos frequência para as camadas superiores, a precomputação pode ser distribuída por várias etapas, resultando em uma melhora significativa da velocidade de assinatura e permitindo escolher grandes parâmetros w de Winternitz, que resulta em assinaturas de tamanho menor. Para garantir chaves privadas de tamanhos pequenos, foi utilizado um Algoritmo PRF, onde somente a semente precisa ser armazenada.

3.6.2 Geração de Assinatura do GMSS

A geração de assinatura do GMSS consiste dos seguintes passos. Primeiro o resumo da mensagem $r = g(M)$ é assinado utilizando a folha atual da subárvore na camada mais profunda(d), resultando em $sigOts[r]$. Depois os caminhos de autenticação em cada camada são atualizados. O caminho de autenticação na camada d sempre é atualizado. Nas camadas $i = d-1, \ldots, 1$ os caminhos de autenticação somente serão atualizados se a folha s da subárvore na camada $i+1$ for igual a $s_{i+1} = 2^{h_{i+1}} - 1$.

A assinatura do GMSS $sig = (i, sigOts[r], Aut[d], sigOts[1], Aut[1], \ldots, sigOts[d-1], Aut[d-1])$, consiste do índice das folhas s, das assinaturas one-time $sigOts[r]$ e $sigOts[i]$ onde $i = 1, ..., d-1$ e dos caminhos de autenticação $Aut[\tau_{i,j_i}, s_i]$ para $i = 1, ..., d$ e $j = 0, ..., 2^{h_1+\ldots+h_{i-1}} - 1$. A geração de assinatura pode ser dividida em duas partes.

A primeira parte, *online*, calcula $sigOts[r]$. A segunda parte, *offline*, pré-calcula os caminhos de autenticação e assinaturas *one-time* das raízes necessárias para as próximas assinaturas.

A cada assinatura, uma próxima subárvore em cada camada i é parcialmente construída, para $i = 2, ..., d$. Neste processo, uma folha das próximas subárvores são geradas e guardadas. Quando a raiz $Raiz[i, 1]$ de cada subárvore for gerada, o algoritmo do esquema WOTS de geração de assinatura gera a assinatura $sigOts[i, 1]$ dessa raiz. Quando a última assinatura da subárvore atual for utilizada, a subárvore atual "reencarna" com a próxima subárvore que já estará pronta. A próxima subárvore será, então, novamente reconstruída.

3.6.3 Verificação de Assinatura do GMSS

O processo de verificação de *GMSS* é essencialmente o mesmo que para o *MSS* e para o *CMSS*. Primeiro, verifica-se a assinatura one-time $sigOts[r]$ da mensagem M usando o esquema one-time de Winternitz. Segundo, o verificador valida as chaves públicas das raízes de todas as subárvores, construindo o caminho $(p[0], ..., p[h])$ da folha até a raiz como no MSS.

3.7 O XMSS

O XMSS [2] é uma modificação do MSS que produz assinaturas de tamanho menor comparado com os esquemas anteriores baseados no MSS. Este esquema usa o esquema WOTS$^+$ descrito na Seção 3.2.3. Especialmente, não é necessário que a função de resumo criptográfica seja resistente à colisão para a segurança do XMSS. Os autores do XMSS demonstraram que este esquema é *forward secure* e eficiente quando instanciado com duas famílias de funções: uma família de funções de resumo resistente à segunda pré-imagem G e outra família de funções pseudoaleatória F, onde

$$G = \{g : \{0,1\}^{2l} \to \{0,1\}^l | K \in \{0,1\}^l\},$$

$$F = \{f : \{0,1\}^l \to \{0,1\}^l | K \in \{0,1\}^l\}.$$

Os parâmetros do XMSS são: $h \in \mathbb{N}$, a altura da árvore; $l \in \mathbb{N}$, tamanho da saída da função de resumo em bits; $w \in \mathbb{N}(w > 1)$, o parâmetro de Winternitz.

Construção da Árvore Binária do XMSS

A árvore binária construída no XMSS é uma modificação da árvore binária construída no MSS. Uma árvore de altura h tem $h + 1$ níveis. Os nós no nível j são $no[i][j]$, para $0 < j \leq h$ e $0 \leq i < 2^{h-j}$ [2].

A função de resumo g é aplicada na árvore do XMSS com o vetor de máscara de bits (bm). Estes vetores de máscara de bits são escolhidos aleatoriamente ou pseudoaleatoriamente, onde $bm[2i + 2j]$ é usado no nó esquerdo e $bm[2i + 2j + 1]$ é usado no nó direito. Os vetores de máscara de bits são a principal diferença das outras árvores de Merkle, e permitem substituir a necessidade de famílias de funções de resumo resistentes a colisão

por uma família de funções de resumo resistentes à segunda pré-imagem. Essa modificação diminui o requisito do tamanho da saída da função de resumo, diminuindo assim o tamanho das chaves do esquema. Os nós são calculados como:

$$no[i][j] = g((no[2i][j-1] \oplus bm[2i+2j])||(no[2i+1][j-1] \oplus bm[2i+2j+1])).$$

3.7.1 Geração das chaves do XMSS

As chaves privadas podem ser geradas utilizando um gerador pseudoaleatório para reduzir os custos de armazenamento. Para geração da chave pública do XMSS, uma árvore binária do XMSS é construída. Para gerar as folhas da árvore do XMSS, uma árvore Ltree é construída. A Figura 3.8 mostra a construção da árvore do XMSS e da árvore Ltree. A máscara de bits esquerda é $bm\_0 = bm[2i + 2j]$ e a máscara de bits direita é $bm\_1 = bm[2i + 2j + 1]$, o nó esquerdo é $noEsq = no[2i][j - 1]$ e o nó direito é $noDir = no[2i + 1][j - 1]$, para para $0 \leq i < 2^{h-j}$ e $0 < j \leq h$.

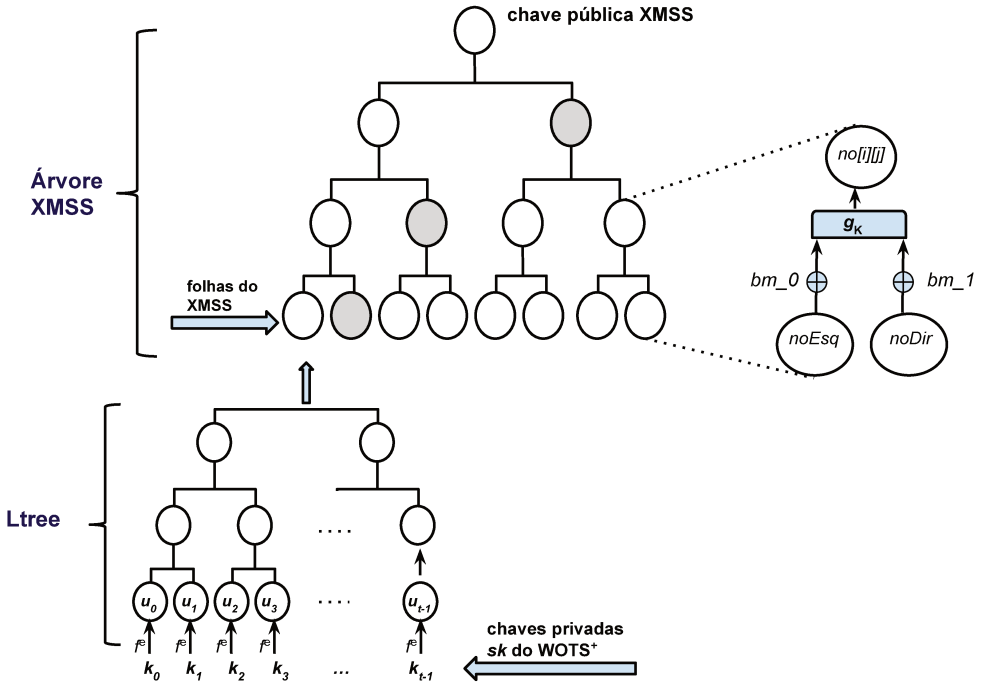


Figura 3.8: Geração das chaves do XMSS adaptada de [2]

Árvore Ltree

A Ltree é uma árvore binária não balanceada, onde os nós internos são calculados como na árvore do XMSS. A Ltree usa um vetor de máscara de bits da mesma forma que a árvore do XMSS. As t folhas da Ltree são os valores das chaves públicas de verificação do esquema WOTS$^+$ $(u[0], \ldots, u[t-1])$. Se t não é uma potência de 2, então não há folhas suficientes para construir uma árvore binária. Então, um nó que não tem irmão

direito será elevado ao nível superior da Ltree até se tornar o irmão direito de outro nó. O Algoritmo 8 descreve os passos para a geração de uma folha da árvore do XMSS.

Algorithm 8 Geração da árvore Ltree [22]

Entrada: parâmetro t, a chave de verificação OTS pk, o bitmask bm.
Saída: uma folha da árvore do XMSS $u[0]$.

```
t' = t;
while (t' > 1) do
    for (i = 0;i < ⌊(t'/2)⌋;i = i + 1) do
        Gerar bm_0 e bm_1 aleatoriamente ou pseudoaleatoriamente;
        u[i] = g((u[2i] ⊕ bm_0)||(u[2i + 1] ⊕ bm_1));
    end for
    if (t'%2 == 1) then
        u[⌊(t'/2)⌋] = u[t' − 1];
    end if
    t' = ⌈t'/2⌉;
end while
return u[0];
```

Geração da Chave Pública do XMSS

A chave pública do XMSS é a raiz da árvore do XMSS e consiste da raiz e dos vetores de máscara de bits bm. O Algoritmo de geração da chave pública gera 2^h chaves de assinatura e verificação do esquema WOTS$^+$. Para cada par de chaves do esquema WOTS$^+$ gerados, o Algoritmo 8 é chamado para gerar as folhas da árvore do XMSS. As folhas geradas, são guardadas como no algoritmo treehash (Algoritmo 4). A diferença está na geração dos nós, onde vetores de máscara de bits são utilizados. Como descrito nesta seção, cada nó interno é calculado como: $no[i][j] = g((no[2i][j-1] \oplus bm[2i+2j])||(no[2i+1][j-1] \oplus bm[2i+2j+1]))$, para $0 < j \leq h$ e $0 \leq i < 2^{h-j}$.

3.7.2 Geração de Assinatura do XMSS

A geração de assinatura do XMSS consiste de dois passos. Primeiro a assinatura do resumo da mensagem $r = g(M)$ é gerada usando uma chave privada do esquema WOTS$^+$ correspondente a uma folha da árvore do XMSS. No segundo passo, a folha da árvore é atualizada em uma unidade e o caminho de autenticação é atualizado para a próxima assinatura. A assinatura $sig = (i, sigOts, pk, (aut[0], ..., aut[h-1]))$ consiste do índice do nó i, da assinatura $sigOts$, da chave de verificação pk e do caminho de autenticação aut.

3.7.3 Verificação de Assinatura do XMSS

A verificação de assinatura também consiste dois passos. No primeiro passo, a assinatura $sigOts$ é verificada utilizando-se a chave de verificação do esquema WOTS$^+$. No segundo passo a chave pública da árvore do XMSS precisa ser validada. Então o receptor calcula o caminho de autenticação construindo o caminho da folha até a raiz, como na verificação

do MSS. A diferença do MSS é que, na reconstrução do caminho, os vetores de máscara de bits são utilizados.

3.8 XMSS-MT-Multi Árvore do XMSS

O XMSS-MT [26] permite assinar um grande, mas fixo, número de mensagens e resolve o problema de geração lenta das chaves. O XMSS-MT é um esquema de assinatura resistente a um ataque de mensagem escolhida (*Existentially Unforgeable under Adaptively Chosen Message Attacks*–EUCMA). O XMSS-MT é considerado seguro quando for instanciado com uma família de funções de resumo resistente à segunda pré-imagem e por uma família de funções pseudoaleatórias. O XMSS-MT usa muitas camadas de árvores do XMSS [3].

A Figura 3.9 representa a construção da árvore do XMSS-MT. O número de camadas é $d \in \mathbb{N}$ onde $1 \leq cm < d$. As folhas da árvore na camada mais baixa $cm = d$ são usadas para assinar as mensagens. As árvores das camadas de $cm = 1, \ldots, d-1$, são usadas para assinar as raízes das árvores na camada abaixo. A chave pública é a raiz da árvore na camada $cm = 1$. Considerando a altura total h com d camadas, então cada subárvore tem altura h/d. Todas as árvores do XMSS tem a mesma altura e o mesmo parâmetro de Winternitz. O XMSS-MT usa o algoritmo do esquema WOTS$^+$ como no XMSS.

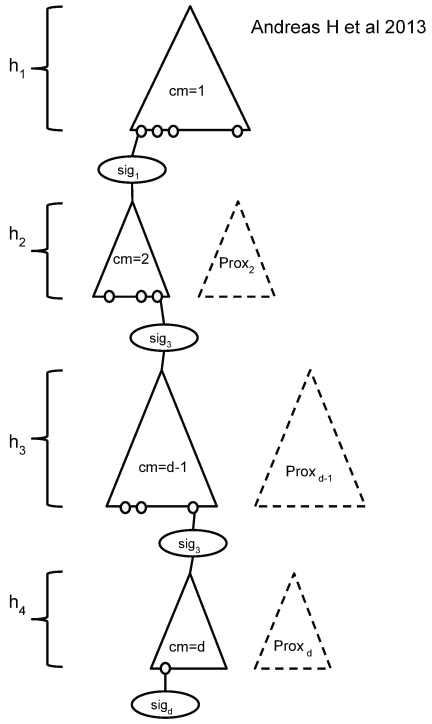


Figura 3.9: Construção do XMSS-MT adaptada de [3]

3.8.1 Geração de Chaves do XMSS-MT

Primeiro os vetores de máscara de bits e as sementes para os geradores pseudoaleatório são gerados aleatoriamente. Os mesmos vetores de máscara de bits são usados em todas

as camadas. Para cada subárvore, o algoritmo de geração de chaves do esquema WOTS$^+$ gera as chaves de assinatura. Então, o Algoritmo de Geração da Chave Pública do XMSS calcula as raízes das árvore $Raiz[\tau_{cm,0}]$, $cm = 1, \ldots, d$.

Os primeiros caminhos de autenticação de cada subárvore $Aut[\tau_{cm,0}, 0]$ são salvos durante a geração das raízes. Quando as raízes para as camadas cm, onde $cm = 2, \ldots, d$ são calculadas, elas são assinadas com o primeiro par de chaves da árvore na camada anterior $cm - 1$. A chave secreta consiste das sementes para a PRF, do vetor com o Algoritmo do Caminho de autenticação e da assinatura das raízes. A chave pública consiste dos vetores de máscara de bits e da chave pública que é a $Raiz$ na camada $cm = 1$.

3.8.2 Geração de Assinatura do XMSS-MT

Para gerar a assinatura do resumo da mensagem $r = g(M)$, o algoritmo recebe como entrada a mensagem M, o índice $indSig$ e a chave secreta sk. A geração de assinatura consiste de duas fases. Primeiro a assinatura do resumo da mensagem $sigOts[r]$ é calculada como no XMSS. A seguir, a chave secreta é atualizada e o algoritmo do caminho de autenticação atualiza os caminhos de autenticação nas camadas necessárias. Além disso, nas camadas $cm = 2, \ldots, d$, quando uma folha é usada, uma nova folha na próxima subárvore é gerada e guardada, como no GMSS. A assinatura é

$$Sig = (indSig, sigOts[r], Aut[d], sigOts[1], Aut[1], \ldots, sigOts[d-1], Aut[d-1]).$$

Um caso especial ocorre se ($indSig$ mod 2^h) = $(2^h - 1)$. Neste caso, o último par de chaves da árvore atual foi usado e uma nova subárvore é necessária para a próxima assinatura. A nova subárvore ($Prox$) já está pronta, porque ela foi gerada em cada assinatura. Então, a árvore atual "reencarna" como $Prox$ e uma nova subárvore será construída. A assinatura da nova raiz também é gerada. O mesmo acontece para todas as camadas j, onde ($indSig$ mod 2^{h_j}) = $(2^{h_j} - 1)$.

3.8.3 Verificação de Assinatura do XMSS-MT

O processo de verificação é semelhante ao do XMSS. Inicialmente, a folha atual da árvore do XMSS, em cada camada, é recuperada a partir da assinatura. Então, para verificar a assinatura das raízes, as folhas recuperadas são utilizadas junto com o respectivo caminho de autenticação para gerar os valores das raízes em cada camada. A raiz da subárvore na camada $cm = 1$ é comparada com a chave pública. Se os valores forem iguais, então a assinatura será válida.

3.9 Segurança

Nesta seção são descritos os aspectos de segurança dos esquemas de assinatura de Merkle e suas variantes.

3.9.1 Segurança do MSS

Os autores em [6] estimaram o nível de segurança do MSS com o esquema LDOTS para uma função de resumo com saída de tamanho l bits. Seja $b \in \mathbb{N}$, o nível de segurança. Este nível de segurança pode ser calculado como t/ϵ onde t é o tempo de execução de um falsificador e ϵ é sua probabilidade de sucesso. Seja K, um conjunto finito, $h \in \mathbb{N}$, t_{cr}, t_{ots}, ϵ_{cr}, $\epsilon_{ots} \in \mathbb{R} > 0$. Seja $G = g_k : \{\{0,1\}^* \to \{0,1\}^l | k \in K\}$ é uma família de funções de resumo (t_{cr}, ϵ_{cr}) resistente a colisão e (t_{ow}, ϵ_{ow}) resistente a pré-imagem. A seguir, é descrito como determinar o nível de segurança do MSS.

Segurança para Computadores Clássicos

A segurança do esquema de Merkle para computadores clássicos, que foi definida em [6], considerou apenas ataques genéricos contra a resistência à pré-imagem e contra a resistência à colisão. Esses ataques genéricos consideram as buscas exaustivas e o ataque de aniversário. Para uma função de resumo com saída de tamanho l bits, um ataque de aniversário que inspeciona $2^{l/2}$ valores do resumo tem uma probabilidade de sucesso de aproximadamente $1/2$. Uma pesquisa exaustiva por uma pré-imagem que inspeciona $2^{l/2}$ valores aleatórios, tem probabilidade $1/2^{l/2}$. Portanto, a família de funções de resumo é $(2^{l/2}, 1/2)$ resistente à colisão e $(2^{l/2}, 1/2^{l/2})$ resistente à pré-imagem. Assim, o nível de segurança do esquema de assinatura de Merkle combinado com o esquema LDOTS é pelo menos

$$b = l/2 - 1,$$

se a altura da árvore de Merkle é no máximo $h \leq l/3$ e o tamanho da saída da função de resumo é pelo menos $l \geq 87$.

Segurança para Computadores Quânticos

Para a segurança do esquema de Merkle usando computadores quânticos, o algoritmo de Grover [20] pode ser usado para realizar ataques genéricos contra a resistência a colisão e resistência à pré-imagem. O algoritmo de Grover requer $2^{l/3}$ avaliações da função de resumo para encontrar uma colisão com probabilidade pelo menos $1/2$. Então, a função de resumo é assumida ser $(2^{l/3}, 1/2^{l/3})$ resistente à pré-imagem. Assim, o nível de segurança do MSS combinado com o esquema LDOTS é pelo menos

$$b = l/3 - 1,$$

se a altura dá árvore de Merkle é no máximo $h \leq l/4$ e o tamanho da saída da função de resumo é pelo menos $l \geq 196$.

3.9.2 Segurança do XMSS

Nesta seção é descrita a segurança do XMSS, conforme provado em [21]. Inicialmente serão apresentadas as características de segurança do XMSS. Ao final, é descrito o nível de segurança do XMSS e da versão multi árvore do XMSS-MT.

O XMSS é um esquema de assinatura *stateful* conforme [21]. Este tipo de esquema não é coberto pela definição de padrão de esquemas de assinatura digital. Para que o esquema seguisse o padrão, os autores seguiram a definição dos esquemas de assinatura com chave em evolução. Em um esquema com chave em evolução, o tempo de vida de um par de chaves é dividido em vários períodos de tempo (T). Enquanto a chave pública (pk) é fixa, as chaves secretas são diferentes ($sk_0, ..., sk_{T-1}$), uma para cada período de tempo. Um esquema de assinatura de chave em evolução contém um algoritmo de atualização de chave que é chamado no final de cada período de tempo para atualizar a chave secreta. O final de um período de tempo pode ser um dia, ou qualquer outra coisa, onde um período termina depois de assinar uma mensagem e o algoritmo de atualização da chave é automaticamente chamado. Em contraste com um esquema de assinatura comum, o algoritmo de geração de chaves de um esquema de assinatura de chave em evolução toma como uma entrada adicional o número máximo de períodos (T) e emite a chave pública (pk) e a primeira chave secreta (sk_0). Uma assinatura de uma mensagem (sig, i) neste esquema, contém o índice i do período da chave secreta utilizada. Esta assinatura (sig, i) é válida somente por período de tempo i para a chave pública pk. O XMSS é um esquema de assinatura de chave em evolução com $T = 2^H$ para $H \in \mathbb{N}$, onde o algoritmo de atualização da chave é chamado automaticamente pelo algoritmo de assinatura depois de cada assinatura.

Definição 4.1 [21]: Um esquema de assinatura com chave em evolução é uma quádrupla de algoritmos probabilísticos em tempo polinomial $\mathtt{KES} = (\mathtt{KG}, \mathtt{KUpd}, \mathtt{SIG}, \mathtt{VER})$, onde KG é o algoritmo de geração de chaves, KUpd é o algoritmo de atualização de chave, SIG é o algoritmo de geração de assinatura e VER é o algoritmo de verificação de assinatura.

O esquema é parametrizado por um parâmetro de segurança $l \in \mathbb{N}$ e pelo número de períodos de tempo $p \in \mathbb{N}$, $p = poly(l)$ e opera sobre o seguinte conjunto finito: O espaço da chave secreta $KS = KS_0 x \dots x KS_{p-1}$ consistindo de p conjuntos, o espaço da chave pública KP, o espaço da mensagem M, e o espaço da assinatura Σ. O tempo de execução dos algoritmos é polinomial em l e os algoritmos são definidos como:

$\mathtt{KG}(1^{\mathtt{l}}, \mathtt{p})$: O algoritmo de geração de chave que tem como entrada o parâmetro de segurança $l \in \mathbb{N}$ em unário e o número de períodos de tempo $p \in \mathbb{N}$, e gera como saída uma chave privada de assinatura privada inicial $sk_0 \in \mathcal{KS}_0$ e uma chave pública de verificação $pk \in \mathcal{KP}$.

$\mathtt{KUpd}(\mathtt{sk}, \mathtt{i})$: O algoritmo de atualização da chave que tem como entrada um índice $i \in \mathbb{N}$ e uma chave privada de assinatura $sk \in \mathcal{KS}$, e gera como saída a chave privada de assinatura $sk' \in \mathcal{KS}_{i+1}$ para o próximo período de tempo, se $i < p - 1$ e $sk \in \mathcal{KS}_i$. Se $i \geq p - 1$, então gera uma cadeia vazia. Em todos os outros casos, retorna falha.

$\mathtt{SIG}(\mathtt{sk}, \mathtt{M}, \mathtt{i})$: O algoritmo de assinatura que tem como entrada uma chave privada de assinatura $sk \in \mathcal{KS}$, uma mensagem $M \in \mathcal{M}$, e um índice $i \in \mathbb{N}$ e gera como saída a assinatura $(sig, i) \in \Sigma$ da mensagem M, se $i < T$ e $sk \in \mathcal{KS}_i$. Caso contrário, retorna falha.

$\mathtt{VER}(\mathtt{pk}, \mathtt{M}, (\mathtt{sig}, \mathtt{i}))$: O algoritmo de verificação, que tem como entrada de uma chave pública $pk \in \mathcal{KP}$, uma mensagem $M \in \mathcal{M}$, e uma assinatura $(sig, i) \in \Sigma$ e produz 1 se, e somente se, (sig, i) é uma assinatura válida de M sobre a chave pública pk para período de tempo i e 0 caso contrário.

$\mathtt{KUpd}(\mathtt{sk_0})^{\mathtt{i}} = \mathtt{KUpd}(\ldots \mathtt{KUpd}(\mathtt{sk_0}, 0)\ldots, \mathtt{i}-1)$ denota o cálculo da chave para o período de tempo i a partir de sk_0. A seguinte condição deve ser mantida: Para todo $\mathtt{M} \in \mathcal{M}$, $(\mathtt{pk}, \mathtt{sk_0}) \longleftarrow \mathtt{KG}(1^{\mathtt{l}}, \mathtt{p})$, e $i < p$: $\mathtt{VER}(\mathtt{M}, (\mathtt{SIG}(\mathtt{M}, \mathtt{KUpd}(\mathtt{sk_0})^{\mathtt{i}}), \mathtt{i}), \mathtt{pk}) = 1$.

Todo esquema de assinatura digital é um esquema de assinatura de chave em evolução com apenas um período (p) e um algoritmo de atualização de chave que sempre retorna uma cadeia vazia [21]. O XMSS é um esquema de assinatura de chave em evolução com $p = 2^h$ para uma árvore de altura h, onde $h \in \mathbb{N}$. O algoritmo de atualização de chave do XMSS consiste em incrementar o índice i da chave secreta e executar o Algoritmo BDS para preparar o próximo caminho de autenticação. Isso é feito depois de cada assinatura.

Conforme definido em [19], um esquema EUCMA é um esquema de assinatura que é resistente a um ataque de mensagem escolhida. Esta definição foi adaptada para o XMSS, onde a chave privada de assinatura muda após cada assinatura. No *Teorema 4.4 [21]* foi provado o seguinte: se G é uma família de funções de resumo resistentes à segunda pré-imagem e F uma família de funções pseudoaleatórias, então o XMSS é EUCMA.

O XMSS usa um gerador de chaves pseudoaleatório construído a partir de uma função unidirecional. Como demonstrado em [52], a existência de uma função unidirecional é a condição necessária para a existência de um esquema de assinatura digital seguro. Em [52], também é apresentado a construção de uma família de funções de resumo resistentes a colisão a partir de uma função unidirecional. Essas construções implicam que existe uma instância segura do XMSS se existe algum esquema de assinatura digital seguro.

Conforme especificado em [21], um esquema de assinatura é *forward secure* se, mesmo depois que uma chave privada de assinatura for comprometida, todas as assinaturas, que foram criadas antes dessa chave comprometida, permanecem válidas. De um ponto de vista baseado em ataque, isso significa que se um invasor descobre uma chave secreta sk_i, ele não pode ser capaz de falsificar uma assinatura de uma chave secreta sk_j, para $j < i$.

Para tornar o XMSS *forward secure* o autor em [21] usa uma função PRF para gerar as sementes das chaves secretas do esquema WOTS. No *Teorema 4.9 [21]* foi provado o seguinte: se G é uma família de funções de resumo resistente à segunda pré-imagem e F uma família de funções pseudoaleatórias, então o XMSS com uma função PRF é um esquema de assinatura *forward secure*.

O XMSS-MT pode ser visto como uma árvore de certificação usando o XMSS. Como cada subárvore é uma árvore do XMSS, que é *forward secure* e EUCMA, então a composição dessas subárvore usando a técnica de encadeamento implica que o XMSS-MT é um esquema de assinatura *forward secure* e EUCMA [26].

Nível de Segurança do XMSS

O autor do XMSS [21] mostra como calcular o nível de segurança do XMSS e do XMSS-MT para um dado conjunto de parâmetros. o XMSS tem nível de segurança b se um ataque bem sucedido sobre o esquema exige $\approx 2^{b-1}$ avaliações de uma função de resumo ou da cifra de bloco. Se o XMSS for combinado com o esquema WOTS$^+$, é preciso considerar os ataques genéricos, os ataques contra a função unidirecional e ataques contra a indetectabilidade.

Sendo l, o tamanho da saída da função de resumo, h a altura da árvore, δ o número

da camada (nível) e w e t parâmetros de Winternitz, então para alcançar um nível de segurança b, os autores calculam:

- Para XMSS, os parâmetros w,t,l,h devem satisfazer a seguinte relação:

$$b > l - h - 1 - max\{h + 3, log(t2w^2 + w)\}.$$

- Para o XMSS-MT, os parâmetros w,t,l,h,d devem satisfazer a seguinte relação:

$$b > \min_{0 \leq \delta \leq d-1} \left\{ l - h_\delta - logd - (\sum_{j=\delta+1}^{d-1} h_j) - 1 - max\{h_\delta + 3, log(t_\delta 2w_\delta^2 + w)\} \right\}.$$

Capítulo 4

Propostas de Padrão

Neste capítulo são descritos duas propostas de padrões de esquemas de assinatura baseados em funções de resumo que foram implementados neste trabalho. Primeiro, na Seção 4.1, descreve-se a proposta de padrão do esquema LMS [37], que é uma variante do esquema de Merkle. Na Seção 4.2, é descrito a proposta de padrão baseada no XMSS [23].

4.1 Proposta de Padrão do esquema LMS

A primeira versão da proposta do esquema LMS denominada LDWM [37], foi submetida em 2014 pelos autores D. McGrew e M. Curcio. Esta proposta descreve especifica o esquema de assinatura *one-time* baseado no trabalho de Lamport [33], Diffie [11], Winternitz [39] e Merkle [39] (LDWM).

A terceira proposta denominada LMS, foi submetida por D. McGrew, M. Curcio e S. Fluhrer e segue as versões anteriores do LDWM porém teve adaptações conforme o trabalho de Leighton e Micali. [34]. A idéia de Leighton e Micali é utilizar cadeias aleatórias para randomizar o resumo da mensagem conforme descrito na Seção 2.4.2. Também é definido um esquema de assinatura de dois níveis(duas camadas de subárvores).

Na quarta versão, publicada em 2016, a ordem dos parâmetros de entrada da função de resumo foi alterada com o objetivo de melhorar a segurança. A mensagem utilizada pela função de resumo foi movida da posição inicial para a posição final e uma cadeia de segurança passou para a posição inicial. O esquema de assinatura hierárquico (*Hierarchical Signature System*–HSS) foi generalizado podendo usar mais de dois níveis.

A quinta versão também publicada em 2016 especificou que, no método HSS, o valor do identificador I (que faz parte da cadeira de segurança) é calculado a partir do valor I da árvore pai. Diferente das versões anteriores onde o valor de I é extraído da chave pública (o que significava que todas as árvores do esquema LMS em um determinado nível e a chave pública usavam o mesmo valor de I). O tamanho do identificador I foi modificado com base no conjunto de parâmetros, permitindo uma implementação significativamente mais rápida. Foi incluído ao conjunto de parâmetros árvore de altura $h = 15$, passando a ter as seguintes opções de valores $h \in \{5, 10, 15, 20\}$.

A seguir, na Seção 4.1.4, é descrito o esquema LM-OTS e na Seção 4.1.5 é descrito o esquema LMS.

4.1.1 Cadeias de Segurança

O esquema LMS [37] introduziu cadeias de valores aleatórios (cadeias de segurança) que são adicionadas a mensagem antes de aplicar a função de resumo. O objetivo destas cadeias é melhorar a segurança contra ataques sobre várias aplicações da função de resumo, porque as funções do esquema proposto executam várias aplicações da função de resumo degradando assim a segurança.

A Figura 4.1 representa o formato geral da mensagem resultante após adicionar as cadeias de segurança cadeia1 e cadeia2 à mensagem M.

Figura 4.1: Formato da mensagem com as cadeias de segurança

As cadeias de segurança, *cadeia*1 e *cadeia*2, estão compostas por um ou mais campos dependendo das funções que são utilizadas:

- I - um identificado (valor aleatório) do par de chaves pública/privada do esquema LMS definido com um tamanho de 64 bytes.

- D - um parâmetro de separação de domínio que possui um único byte e assume os seguintes valores:
  $DITER = 0x00$ nas iterações da função F dos algoritmos do LM-OTS.
  $DPUBL = 0x01$ na aplicação da função $Fpub$ para calcular o resumo da concatenação de todos os valores de pk.
  $DMESG = 0x02$ na aplicação da função $Fmsg$ para calcular o resumo da mensagem nos algoritmos LM-OTS.
  $DFOLH = 0x03$ na aplicação da função $Ffolh$ para gerar um nó folha da árvore.
  $DINTR = 0x04$ na aplicação da função G para calcular um nó interno da árvore.

- C - um valor aleatório de n bytes que é incluído com a mensagem na aplicação da função $Fmsg$ para calcular o resumo da mensagem.

- r - um valor de 4 bytes associado a um nó da árvore que é usado para calcular esse nó.

- q - uma cadeia de 4 bytes associada como o índice da folha da árvore do esquema LMS.

- i - um valor de 2 bytes associado com o índice do elemento da chave privada do LM-OTS, na qual a função de resumo está sendo aplicada.

- j - um valor de 1 byte que representa o número da iteração da aplicação da função F no elemento da chave privada do LM-OTS.

4.1.2 Funções e representações

A representação X_y especifica a uma cadeia binária de X com tamanho fixo de y bytes (*big-endian*), para X e y inteiros não negativos ($y > 0$). Por exemplo, 1_1 representa o valor [00000001] e 3_2 representa o valor [0000000000000011].

A função $bytes(S, i, j)$ representa o intervalo do $i-$ésimo byte até $j-$ésimo byte da cadeia S. Dada uma cadeia de bytes S e um índice i, então $bytes(S, i)$ representa o $i-$ésimo byte de S. Por exemplo, se $S = 0x02040608$, então $bytes(S, 0)$ é $0x02$ e $bytes(S, 1, 2)$ retorna $0x0406$.

A função $coef(S, i, w)$ é utilizada para representar uma cadeia de bytes como uma cadeia de inteiros sem sinal de w bits. A função é definida por: se S é uma cadeia, i um inteiro positivo, e w um membro do conjunto $\{1, 2, 4, 8\}$, então $coef(S, i, w)$ é o $i-$ésimo valor w, se S é interpretado como uma sequência de valores de w bits.

4.1.3 Funções de Resumo Chaveadas

O formato da entrada para uma função de resumo (H) tem a propriedade de que cada invocação dessa função terá uma entrada que é distinta de todas as outras, com alta probabilidade. Isto ocorre porque a proposta do esquema LMS randomiza cada chamada da função de resumo. Esta propriedade é importante para provar a segurança no modelo de oráculos aleatórios. Os formatos são utilizados durante a geração de chaves, assinatura e verificação.

O formato das funções do esquema LMS podem ter como parâmetros: uma árvore de altura h, uma sequência de caracteres $S = \{I||q\}$, e os parâmetros I, q, r e D conforme definidos na Seção 4.1.1. O parâmetro *tmp* corresponde ao valor de *sig* no processo de verificação ou ao valor de *sk* na geração de chaves e na assinatura. O padrão recomenda usar como função H a função de resumo SHA-256. Os formato das funções do esquema LMS são descritas a seguir.

- $F(S, tmp, i, j) = H(X)$, onde $X = \{S||tmp||i_{16}||j_8)||DITER\}$.
- $G(I, noEsq, noDir, r) = H(Y)$, onde $Y = \{I||noEsq||noDir||r_{32}||DINTR\}$.
- $Fpub(S, pk) = H(Z)$, onde $Z = \{S||pk||DPUBL\}$.
- $Fmsg(S, C, M) = H(W)$, onde $W = \{S||C||M||DMESG\}$.
- $Ffolh(I, noFolha, r) = H(V)$, onde $V = \{I||noFolha||r_{32}||DFOLH\}$.

4.1.4 Esquema LM-OTS

O Esquema de Assinatura *One-Time* Leighton e Micali (LM-OTS) é usado para validar a autenticidade de uma mensagem através da associação de uma chave privada com a chave pública compartilhada [37]. Para facilitar o uso dessas assinaturas em um esquema de assinatura *N-Time*, foi introduzido um parâmetro q, que representa o índice da folha da árvore binária. Quando o esquema LM-OTS é usado fora de um esquema de assinatura *N-Time*, este parâmetro recebe o valor zero.

O esquema LM-OTS tem os seguintes parâmetros: n, o número de bytes da saída da função de resumo; w, o parâmetro de Winternitz ($w \in \{1, 2, 4, 8\}$); p, o número de elementos do vetor de n bytes que compõe a assinatura do LM-OTS e ls, o número de bits de deslocamento a esquerda usados na função de soma de verificação CKSM.

Geração da Chave Privada LM-OTS

A chave privada sk podem ser geradas por um gerador aleatório ou por um método pseudoaleatório. O algoritmo recebe como entrada o identificador I, os parâmetros p e q e gera sk. A sequência S de caracteres de segurança tem tamanho tmS bytes, onde $tmS = 68$ bytes e $I = 64$ bytes [37]. Cada chave sk só deve ser usada para assinar uma única mensagem. O Algoritmo 9 mostra o pseudocódigo para gerar uma chave privada.

Algorithm 9 Geração da chave privada do LM-OTS [37]

Entrada: o identificador I, os parâmetros p e q
Saída: a chave privada sk

```
tipo32 = parâmetros do Algoritmo;
S = {I||q};
for ( i = 0; i < p; i = i + 1 ) do
    gera as chaves k[i] aleatoriamente ou pseudoaleatoriamente;
end for
return sk={tipo32||S||k[0]||k[1]||...||k[p − 1]};
```

Geração da Chave Pública LM-OTS

A chave pública do LM-OTS (pk), é gerada a partir da chave privada sk [37]. A função F é executada $2^w - 1$ vezes em cada elemento de sk para gerar os elementos da chave pública (pk). Ao final a função $Fpub$ é aplicada na concatenação de todos os valores de pk resultantes. As funções F e $Fpub$ utilizam a sequência de segurança $S = \{I||q\}$ de tamanho tmS bytes.

O Algoritmo 10 mostra o pseudocódigo para geração da chave pública do LM-OTS. Cada par de chaves pública/privada é associado com um único identificador I, gerado durante a geração do par de chaves. O parâmetro q (índice da folha) é uma entrada para o algoritmo. O Algoritmo retorna a chave pública pk=$\{tipo_{32}||S||K\}$.

Função Soma de Verificação

Uma função de soma de verificação (CKSM) é utilizada para assegurar que uma tentativa de falsificação, que manipula os elementos de alguma assinatura, seja detectada [37]. A função CKSM tem como entrada S, uma cadeia de bytes e retorna o valor sum. A função CKSM é definida no Algoritmo 31 no Apêndice.

Geração de Assinatura do LM-OTS

O processo de assinatura tem os seguintes passos. Primeiro é gerada a assinatura da mensagem M utilizando a função $Fmsg$. Para gerar este resumo, a mensagem é concatenada

Algorithm 10 Geração da chave pública do LM-OTS [37]

Entrada: a chave privada sk e os parâmetro p e w.
Saída: a chave pública pk.

$tipo_{32}$ = parâmetros do Algoritmo;
Obtenha k e S da chave privada sk;
for ($i = 0; i < p; i = i + 1$) **do**
  $tmp = k[i]$;
  for ($j = 0; j < 2^w - 1; j = j + 1$) **do**
    $tmp = F(S, tmp, i, j)$;
  end for
  $u[i] = tmp$;
end for
$K = Fpub(S, u)$;
return pk=$\{tipo_{32}||S||K\}$;

com a sequência de segurança $S = \{I||q\}$ de tamanho tmS bytes, um valor aleatório C e o $DMESG$.

O resultado do resumo da mensagem é assinado da seguinte forma [37]. A variável a, recebe o coeficiente correspondente a posição i, calculado a partir do resultado do resumo da mensagem Q. A variável a é uma sequência de valores de w bits. Cada valor w, determina o número de vezes que a função F é aplicada no elemento k da chave privada.

O pseudocódigo para este procedimento é mostrado no Algoritmo 11. O identificador I e a folha q, são os mesmos usados na Seção 4.1.4. As saídas da função F são concatenadas e retornadas como a assinatura $sigOts$.

Algorithm 11 Geração de Assinatura do LM-OTS [37]

Entrada: a mensagem M, a chave privada sk e os parâmetros p e w.
Saída: a assinatura $sigOts$.

$tipo_{32}$ = parâmetros do Algoritmo;
calcular a cadeia de segurança C aleatoriamente;
Obtenha k e S de sk;
$Q = Fmsg(S, C, M)$;
for ($i = 0$; $i < p$; $i = i + 1$) **do**
  $a = coef(Q||Cksm(Q), i, w)$;
  $tmp = k[i]$;
  for ($j = 0$; $j < a$; $j = j + 1$) **do**
    $tmp = F(S, tmp, i, j)$;
  end for
  $sig[i] = tmp$;
end for
return $sigOts = \{tipo_{32}||C||sig[0]||sig[1]||...||sig[p-1]\}$;

Verificação de Assinatura do LM-OTS

Com o objetivo de verificar a assinatura de uma mensagem (sig), o receptor deve calcular uma série de aplicações de F usando os valores do resumo da mensagem e da soma de verificação [37]. Esse valores são armazenados e concatenados, como na geração de chaves

para aplicação da função de resumo. O resultado da função $Fpub$ é comparado com a chave pública $pkLMOTS$. O pseudocódigo do processo de Verificação de Assinatura do LM-OTS é apresentado no Algoritmo 12.

Algorithm 12 Verificação de Assinatura do LM-OTS [37]

Entrada: a mensagem M, a assinatura $sigOts$, a chave pública pk, e os parâmetros p e w.
Saída: resultado da verificação.

```
if (o tipo de pk ≠ do tipo de sig) then
    return INVÁLIDO;
end if
if (o tamanho de sigOts ≠ ((8 + n).(p + 1)) bytes) then
    return INVÁLIDO;
end if
if ( o tamanho de pk ≠ (4 + tmS + n) bytes) then
    return INVÁLIDO;
end if
Obtenha C e sig de sigOts;
Obtenha S e K de pk;
Q = Fmsg(S, C, M);
for (i = 0; i < p; i = i + 1) do
    a = (2^w − 1) − coef(Q||Cksm(Q), i, w);
    tmp = sig[i];
    for (j = a; j < 2^w − 1; j = j + 1) do
        tmp = F(S, tmp, i, j);
    end for
    z[i] = tmp;
end for
Kc = Fpub(S, z);
if (Kc == K) then
    return VÁLIDO;
else
    return INVÁLIDO;
end if
```

4.1.5 O Esquema LMS

O esquema LMS [37] pode assinar uma quantidade grande mas fixa de mensagens. Esse esquema usa dois componentes criptográficos: um esquema de assinatura *one-time* e uma função de resumo. Cada par de chave pública/privada do esquema LMS é associada com uma árvore binária, conforme a árvore de Merkle.

O esquema LMS tem os seguintes parâmetros: h, altura da árvore (número de níveis - 1) e n, número de bytes associado a cada nó. Há 2^h folhas na árvore. Cada folha da árvore contém o valor da chave pública de um par de chaves do LM-OTS. Cada nó interior é calculado através da aplicação da função de resumo à concatenação dos valores de seus nós filhos. A chave pública do esquema LMS é o valor da raiz da árvore.

Cada nó da árvore é associado com um índice inteiro denotado como $indNo$. O índice do nó raiz é 1. Um nó com índice i, tem o seu filho esquerdo com índice $2.i$, e

o direito $2.i + 1$. Cada nó da árvore terá um índice único, e as folhas terão os índices: $2^h, (2^h)+1, (2^h)+2, \ldots, (2^h)+(2^h)-1$. O $j-$ésimo nó no nível L tem índice de nó $2^L + j$.

Chave Privada do esquema LMS

Uma chave privada do esquema LMS consiste de 2^h chaves privadas de assinatura *one-time* e o índice da folha da próxima chave privada LM-OTS que ainda não foi utilizada [37]. Um elemento da chave privada (sk) é gerado usando o Algoritmo 13. O índice da folha q é inicializado com zero quando a chave privada do esquema LMS é criada.

Algorithm 13 Geração da chave privada do esquema LMS [37]

Entrada: altura h, o identificador I, os parâmetros p e q.
Saída: a chave privada SK.

```
tipo32 = parâmetros do Algoritmo;
S = {I||q};
for ( i = 0; i < 2^h; i = i + 1 ) do
    sk[i] = chave privada do LM-OTS;
end for
return SK={tipo32||S||sk[0]||sk[1]||...||sk[2^h − 1]};
```

Geração da Chave Pública do esquema LMS

A chave pública PK do esquema LMS, é associada com as i chaves públicas/privadas do LM-OTS, para i de 0 até $2^h - 1$ [37]. Cada folha da árvore do LMS recebe uma chave pública do LM-OTS calculada como $pk[i] = \{tipo_{32}||S||K\}$. Cada nó é associado com uma cadeia de n bytes. O Algoritmo 14 descreve o processo de geração da chave pública do LMS.

Geração de Assinatura do esquema LMS

A assinatura do esquema LMS consiste: de um código ($tipo_{32}$) indicando os parâmetros do esquema LMS, do índice da folha q associado com a assinatura do LM-OTS, da assinatura do LM-OTS, e de um vetor de valores aut, que são o caminho de autenticação [37]. A assinatura pode ser representada como

$$Sig = \{tipo_{32}||q_{32}||sigOts||aut[0]||aut[1]||...||aut[h-1]\}.$$

Para calcular a assinatura da mensagem com a chave privada do esquema LMS, o emissor primeiro computa a assinatura do LM-OTS da mensagem, usando a chave privada do LM-OTS correspondente à próxima folha que ainda não foi utilizada. Então, o índice da folha da chave privada do esquema LMS deve ser incrementado e o caminho de autenticação também deve ser atualizado. Para atualizar o caminho de autenticação, pode-se utilizar qualquer algoritmo conhecido na literatura, como o algoritmo BDS citado no Capítulo 3, Seção 3.4.

Algorithm 14 Geração da chave pública do esquema LMS [37]

Entrada: a altura h, o nó inicial s, o identificador I, os parâmetros p, q e r.
Saída: chave pública PK.

```
indvet = 0; //índice do vetor vetorNo
for (indSig = s; indSig < 2^h; indSig = indSig + 1) do
    noAtual.altura = 0; //altura de in
    noAtual.indNo = indSig; //índice do nó
    gerar sk e pk para folha indSig;
    noAtual.hash = Folh(I, pk, r);
    while ((noAtual.altura == vetorNo[indvet − 1].altura) AND (indvet! = 0)) do
        indvet − −;
        tmp = vetorNo[indvet].hash;
        noAtual.hash=G(I, tmp, noAtual.hash, r);
        noAtual.altura + +;
        noAtual.indNo = (noAtual.indNo)/2;
    end while
    vetorNo[indvet]=noAtual;
    indvet + +;
end for
pub = vetorNo[indvet];
return PK = {tipo_32||I||pub};
```

Verificação de Assinatura do esquema LMS

A verificação do esquema LMS consiste de dois passos. No primeiro, o algoritmo de verificação LM-OTS é executado para verificar a assinatura $sigOts$ [37]. No segundo passo, o caminho de autenticação (aut) é utilizado para validar a chave pública pub. O Algoritmo 15, descreve o processo de verificação de assinatura do esquema LMS. O Algoritmo retorna que a assinatura é válida, se o valor da raiz for igual a chave pública do esquema LMS, caso contrário, retorna que a assinatura é inválida.

4.1.6 Esquema de Assinatura Hierárquico

O HSS possui d camadas de subárvores ($2 \leq d \leq 8$). Esse esquema pode ser utilizado para minimizar o tempo de geração da chave pública [37]. Cada camada possui uma subárvore do esquema LMS com altura h.

Este esquema usa uma estrutura $hssInfo$, que contém os parâmetros do Algoritmo do esquema LMS e os parâmetros do Algoritmo do LM-OTS.

A seguinte notação é usada, onde $1 \leq i \leq d$:

- $prv[i]$: é a chave privada da camada i,
- $pub[i]$: é a chave pública da camada i,
- $sig[i]$: é a assinatura da camada i.

Algorithm 15 Verificação de Assinatura do esquema LMS [37]

Entrada: a assinatura Sig e a chave pública PK.
Saída: resultado da verificação.

Obtenha *tipoPub*, I e *pub* da chave pública PK;
if (tamanho de PK $\neq (4 + LenI + m)$ bytes) **then**
  return INVÁLIDO;
end if
Obtenha *tipoSig*, q, *sigOts* e *aut* da assinatura Sig;
if (tipoPub $\neq$ tipoSig) **then**
  return INVÁLIDO;
end if
Calcular Kc no Algoritmo 12;
$i = 0$;
$indNo = 2^h + q$;
$tmp = Ffolh(I, Kc, indNo)$;
while $(indNo > 1)$ **do**
  if ($indNo$ é impar) **then**
    $tmp = G(I, aut[i], tmp, (indNo/2))$;
  else
    $tmp = G(I, tmp, aut[i], (indNo/2))$;
  end if
  $indNo = indNo/2$;
  $i = i + 1$;
end while
if $(tmp = pub)$ **then**
  return VÁLIDO;
else
  return INVÁLIDO;
end if

Geração de Chaves do HSS

Para gerar um par de chaves pública/privada do HSS, pares de chaves públicas/privadas do esquema LMS ($pub[i]/prv[i]$) são gerados, em cada nível $1 \leq i \leq d$ [37]. Estes pares de chaves devem ser gerados de forma independente, e cada par de chaves deve usar um identificador distinto I. A chave pública do HSS é a chave pública do primeiro nível ($pub[1]$), seguido da estruturas $hssInfo$ dos níveis restantes $1 \leq i \leq (d-1)$.

A chave privada do HSS consiste de $\{prv[1]||\ldots||prv[d]\}$. Os valores do primeiro nível ($pub[1]$ e $prv[1]$) não mudam, porém os valores de $pub[i]$ e $prv[i]$, para $i > 1$, mudam de acordo com o algoritmo de geração de assinatura.

Geração de Assinatura do HSS

Conforme descrito em [37], para assinar uma mensagem utilizando com a chave privada do HSS, os seguintes passos são realizados: A mensagem é assinada com a chave privada da camada (d-1) ($prv[d-1]$), resultando em $Sig[d-1]$. A assinatura das chaves públicas do esquema LMS, das subárvores dos níveis de 2 a d, podem ser geradas quando estas subárvores são construídas na geração do par de chaves. A chave pública do nível i, $pub[i]$ é assinada com a chave privada do nível anterior ($prv[i-1]$), resultando em $Sig[i]$.

A assinatura do HSS consiste de $d-1$ assinaturas do LMS e é descrita como:

$$sigLMS[i] = \{Sig[i]||pub[i+1]\},\ para\ 0 \leq i \leq (d-2).$$

O número de elementos de $sigLMS$ é indicado pelo valor $(d-1)$ e aparece nos 4 bytes iniciais da assinatura. A assinatura do HSS é:

$$\{(d-1)_{32}||sigLMS[0]||...||sigLMS[d-2]||sigLMS[d-1]\}.$$

Há 2^h pares de chaves do LM-OTS em cada nível. Se o último índice da chave privada do nível $(d-1)$ for utilizada ($prv[d-1]$), então inicia-se um processo de regeneração da árvore do esquema LMS naquele nível. Neste processo, novos pares de chaves pública/privada do esquema LMS ($pub[d-1]/prv[d-1]$) são geradas nesse nível. A nova chave pública $pub[d-1]$ é assinada com a chave privada do nível anterior ($prv[d-2]$), resultando em $Sig[d-2]$. Se $prv[d-2]$ também for a última chave do nível $(d-2)$, então o processo de regeneração é aplicado a esse nível, e assim sucessivamente até o nível 2.

Verificação de Assinatura do HSS

Para verificar a assinatura usando a chave pública do HSS, os seguintes passos são realizados:

- Para $i = d$, a assinatura $sig[i]$ da mensagem M é verificada usando a chave pública $pub[i]$.
- Para $i = d-1$ até 1, assinatura $sig[i]$ da raiz $pub[i+1]$ é verificada utilizando a chave pública $pub[i]$.

Se a verificação falhar em algum nível, a verificação retorna INVÁLIDO, caso contrário, retorna VÁLIDO.

4.1.7 Segurança do esquema LMS

A proposta do esquema LMS [37] prova a segurança no modelo oráculo aleatório. Para isto, o esquema introduziu "cadeias de segurança" que é distinta para cada chamada da função de resumo, atenuando os ataques sobre múltiplas invocações da função de resumo.

De acordo com a análise em [31], se o esquema LDWM executa N instâncias da função de resumo (com tamanho de saída de n bytes) ao mesmo tempo, então a segurança degrada linearmente em N. Da mesma forma, se a função F é aplicada em todas as t instâncias do esquema para gerar as chaves pk, então o nível de segurança degrada linearmente em $t \times N$. Considerando a aplicação de F^e, para q/e valores aleatórios. Seja uma entrada x, tal que $F^e(x) = y_j^i$ para algum valor de i e j. A probabilidade de que o x seja encontrado por um falsificador é aproximadamente $(q/e) \times p \times N \times (2^{-8n})$. Então, para garantir k bits de segurança, o tamanho da mensagem deve ser pelo menos $k + \texttt{log}N + \texttt{log}p - \texttt{log}e$ bits.

A escolha da saída da função de resumo (n) tem um forte efeito sobre a segurança do sistema. Conforme especificado em [37], o valor de n é determinado pelas funções selecionadas para uso como parte do algoritmo LM-OTS. Como demonstrado por Katz [31] no Teorema 8, LM-OTS é seguro no modelo oráculo aleatório: para qualquer adversário atacando arbitrariamente muitas instâncias de assinaturas *one-time* e fazendo no máximo q consultas aos valores de resumo, a probabilidade do adversário falsificar uma assinatura é no máximo $q2^{(1-8n)}$.

Para um nível de segurança de 128 bits, supondo que não haja computadores quânticos, pode-se usar uma função de resumo com saída $n = 32$ (256 bits), truncada em $n = 16$ (128 bits). Para um nível de segurança de 128 bits, considerando os computadores quânticos, a segurança é equivalente à raiz quadrada do tamanho do domínio dessa função, sendo necessário usar uma função de resumo com saída $n = 32$. O esquema LMS [37] define o uso da função SHA-256 para $n = 32$ e SHA-256-16 para $n = 16$. A função SHA-256-16 é a função SHA-256 com sua saída final truncada para retornar os 16 bytes mais à esquerda.

O esquema LMS, requer que o emissor mantenha o estado da assinatura atual para assegurar que nenhum componente da assinatura seja usado mais de uma vez. Assim, depois que uma assinatura é gerada, o valor da chave privada terá que ser atualizado.

4.2 Proposta de Padrão do XMSS

Em 2015, Hülsing et al. [22] propuseram a primeira versão de um Internet-Draft que especifica o XMSS: *Extended hash-based signatures* com o esquema de assinatura *one-time* WOTS$^+$, uma única árvore (XMSS) e multi-árvores (XMSS-MT).

Em Janeiro de 2016, a segunda versão apresenta as seguintes alterações: adicionou as funções de resumo chaveadas para melhorar a segurança. Além disso, foi introduzido um parâmetro de endereço para ser utilizado pelas funções de resumo. Os valores dos vetores de máscara de bits passam a ser gerados pseudoaleatoriamente, onde somente é

necessário armazenar uma pequena semente de n bytes. A função de resumo SHA-3 foi substituída pela função de resumo SHA-2, porque o SHA-2 apresenta melhores tempos de execução e tem uso mais generalizado. O conjuntos de parâmetros baseados em AES foram removidos.

Em Maio de 2016, a quarta versão da proposta tem as seguintes alterações: endereços têm agora tamanho de 32 bytes. Alguns elementos de endereço foram aumentados, especialmente o endereço da árvore, que agora tem 64 bits. A função ChaCha20 não é mais utilizada e o conjuntos de parâmetros SHA-256 são agora obrigatórios, enquanto para o SHA-512, são opcionais. A função SHA3 (SHAKE128 e SHAKE256) foi adicionada como opcional. Em Julho de 2016, a sexta versão modifica o tamanho dos campos do endereço para simplificar a implementação.

4.2.1 Esquema de Endereço da Função de Resumo

O XMSS [23] randomiza cada chamada da função de resumo. Isto significa que, além do resumo da mensagem inicial M, cada chamada da função de resumo se utilizam valores aleatórios diferentes. Estes valores são gerados a partir de uma função pseudoaleatória que recebe como entrada uma semente $Seed$ de n bytes e um parâmetro de 32 bytes denominado endereço $Adrs$. A função pseudoaleatória devolve um valor de n bytes, onde n é o parâmetro de segurança.

A Figura 4.2 representa os formatos dos endereços $Adrs$ conforme especificado em [23]. Existem três tipos diferentes de endereços: um usado em esquemas OTS ($AdrsOts$), um usado na construção das Ltrees $AdrsLtree$ e um usado na construção da árvore principal de Merkle $AdrsTree$. Todos os campos dentro desses endereços são codificados como valores inteiros sem sinal.

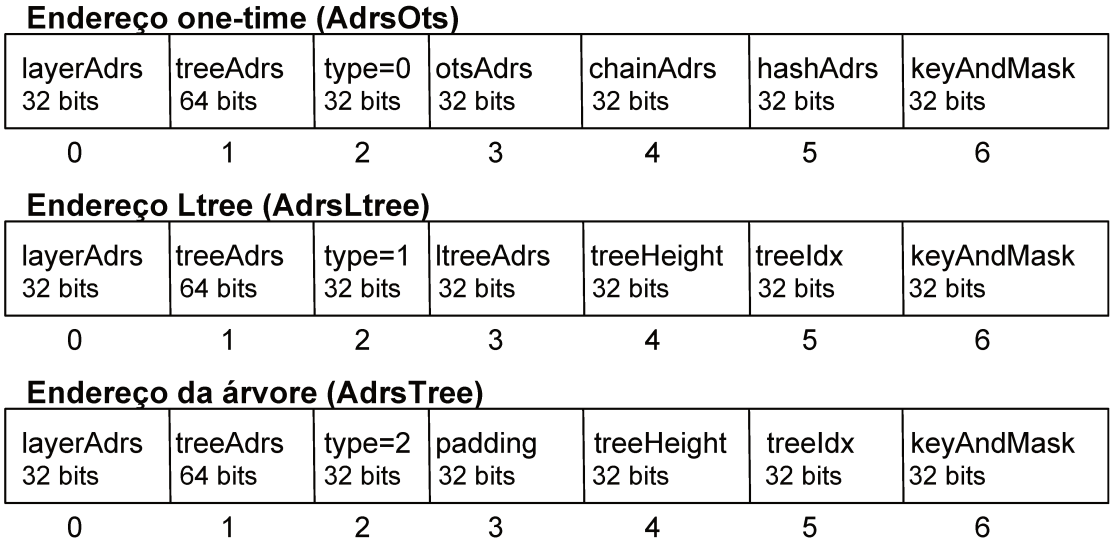


Figura 4.2: Formato dos endereços

4.2.2 Função baseW

A função $baseW(X, w, outLen)$ considera uma cadeia de bytes como uma cadeia de números de base w, ou seja, inteiros pertencentes ao conjunto: $\{0, ..., w-1\}$. A correspondência é definida como: Se X tem len_X bytes e w é um membro do conjunto $\{4, 16\}$, então a função $baseW(X, w, outLen)$ retorna um vetor de $outLen$ posições com valores inteiros entre 0 e $w - 1$. O tamanho de $outLen$ é deve ser menor ou igual a $8 * len_X/log_2(w)$. O Algoritmo 32 correspondente a função $baseW()$, é descrito no Apêndice.

4.2.3 Funções de Resumo Chaveadas

As funções de resumo chaveadas usam uma função de resumo (H). Recebem como entrada, uma cadeia c de n bytes, uma chave Key (gerada por uma função pseudoaleatória cujas entrada são uma semente pública $Seed$ e um endereço $Adrs$), e a mensagem M.

As funções de resumo chaveadas utilizadas são definidas como:

- $F(Key, M) = H(X)$, onde $X = \{0_n||Key||M\}$,
- $G(Key, M) = H(Y)$ onde $Y = \{1_n||Key||M\}$,
- $Fmsg(Key, M) = H(Z)$, onde $Z = \{2_n||Key||M\}$,
- $PRF(Key, M) = H(W)$, onde $W = \{3_n||Key||M\}$.

4.2.4 Esquema WOTS$^+$

Hülsing [25] propôs WOTS$^+$, uma modificação do WOTS que usa uma função de encadeamento F^e começando de uma entrada aleatória. No XMSS [23], o autor descreve WOTS$^+$, uma versão semelhante a especificada em [25]. O algoritmo WOTS$^+$ usa uma função de resumo criptográfica chaveada F. A função F recebe e retorna bytes de tamanho n usando chaves de tamanho n bytes. Além disso, o WOTS$^+$ usa uma função pseudoaleatória PRF. PRF toma como entrada uma chave de n bytes e um índice de 32 bytes e gera saídas pseudoaleatórias de tamanho n.

WOTS$^+$ usa os parâmetros n, e w (inteiros positivos). Esses parâmetros são:

- n : o tamanho da mensagem e dos elementos das chaves secreta, da chave pública e da assinatura em bytes.
- w : o parâmetro de Winternitz, pertencente ao conjunto $\{4, 16\}$.

O valor de n é determinado pela função de resumo criptográfica usada em WOTS$^+$. A função de resumo é escolhida para garantir um nível de segurança adequado. Estes parâmetros são conhecidos publicamente.

Os parâmetros n e w são usados para calcular t (o número de elementos da cadeia de n bytes da chave secreta, da chave pública e da assinatura do WOTS$^+$). Os parâmetros são computados como: $t = t1 + t2$, onde:

$$t1 = \lceil n/(log_2(w)) \rceil \quad e \quad t2 = \lfloor (log_2(t1(w-1)))/(log_2(w)) \rfloor + 1.$$

Função de Encadeamento

A função de encadeamento (Algoritmo 16) calcula uma iteração da função F sobre uma entrada de n bytes usando as saídas da função PRF [23]. Em cada iteração, a função PRF gera uma chave Key e uma máscara bm que são utilizadas pela função F. $Adrs$ é um endereço OTS de 32 bytes e $Seed$ é uma cadeia de n bytes. A função de encadeamento recebe como entrada uma cadeia X de n bytes, um índice inicial i, um número de passos s, bem como $Adrs$ e $Seed$. A função de encadeamento retorna o resultado final após s iterações da função F.

Algorithm 16 Função de Encadeamento (*chain*)[23]

Entrada: cadeia X, índice inicial i, número de passos s, $Seed$, $Adrs$.
Saída: resultado da função F.

```
if (s == 0) then return X;
end if
tmp = X;
for (nr = i; nr < s; nr++) do
    Adrs[5] = nr; //hashAdrs=nr
    Adrs[6] = 0; //KeyAndMask=0
    Key = PRF(Seed, Adrs);
    Adrs[6] = 1; //KeyAndMask=1
    bm = PRF(Seed, Adrs);
    tmp = F(Key, tmp XOR bm);
end for
return tmp;
```

Geração de Chaves do WOTS$^+$

Um par de chaves do WOTS$^+$ consiste de uma cadeia com t elementos de tamanho w [23]. O algoritmo WOTSgenSK tem como entrada uma semente S. O algoritmo chama a função PRF para calcular as chaves privadas $sk = \{k[0]||\ldots||k[t-1]\}$. Os elementos de sk são calculados como $k[i] = PRF(S, i_{32})$. A semente S deve ser diferente da semente $Seed$, que é utilizada para randomizar a função de resumo.

O Algoritmo de geração da chave pública, denominado WOTSgenPK (Algoritmo 17), descreve o procedimento para gerar a chave pública pk. O algoritmo recebe os valores de sk que são passados para a função de encadeamento. A chave pública serão os valores devolvidos pela função *chain* (Algoritmo 16).

Algorithm 17 WOTSgenPK[23]

Entrada: a chave privada sk, valores de $Seed$ e $Adrs$.
Saída: a chave pública pk.

```
for (i = 0; i < t; i++) do
    Adrs[4] = i; //chainAdrs=i
    u[i] = chain(k[i], 0, w - 1, Seed, Adrs);
end for
return pk = {u[0]||...||u[t-1]};
```

Geração de Assinatura do WOTS$^+$

A assinatura do WOTS$^+$ é um vetor com t elementos de tamanho n bytes. A assinatura do WOTS$^+$ é gerada por mapear uma mensagem para t valores inteiros entre 0 e $w-1$. A mensagem é transformada em $t1$ valores na base w usando a função $baseW$ (Algoritmo 32). A soma de verificação é computada e o valor de $t2$ é gerado [23].

O Algoritmo 18 descreve o procedimento para a geração de uma assinatura, onde M é a mensagem e $sigOts$ é a assinatura. O Algoritmo $WOTSsig$ tem como entrada um endereço $Adrs$ e uma semente $Seed$. Cada um dos valores inteiros retornados pela função $basew$, será utilizado para definir o número de aplicações da função F nos elementos de sk. A assinatura é formada pela concatenação dos valores retornado em sig.

Algorithm 18 $WOTSsig$[23]

Entrada: a chave privada sk, a mensagem M, valores de $Seed$ e $Adrs$.
Saída: a assinatura $sigOts$.

$csum = 0$;
$msg = baseW(M, w, t1)$;// converte a mensagem para a base w
for $(i = 0; i < t1;\ i++)$ **do**
  $csum = csum + w - 1 - msg[i]$;// calcula a soma de verificação
end for
$csum = csum << (8 - ((t2.log(w))\%8))$;// converte csum para a base w
$t2\_bytes = \lceil (t2.log(w))/8 \rceil$;
$msg = \{msg||baseW(csum_{t2\_bytes}, w, t2)\}$;
for ($i = 0; i < t;$ i++) **do**
  $Adrs[4] = i$; //chainAdrs=i
  $sig[i] = chain(k[i], 0, msg[i], Seed, Adrs)$;
end for
return $sigOts = \{sig[0]|| \ldots ||sig[t-1]\}$;

Verificação de Assinatura do WOTS$^+$

Para verificar uma assinatura $sigOts$ da mensagem M, o verificador calcula o valor de chave pública WOTS$^+$ a partir da assinatura [23]. O verificado completa os cálculos da função de encadeamento a partir dos valores da assinatura, usando os valores gerados na $baseW$. Esta etapa, chamada de $WOTSpkFromSig$, é descrita no Algoritmo 19. O resultado de $WOTSpkFromSig$ é então comparado com a chave pública. Se os valores forem iguais, a assinatura será aceita. Caso contrário, a assinatura deve ser rejeitada.

4.2.5 O XMSS

O XMSS usa quatro componentes criptográficos: o esquema WOTS$^+$, duas funções de resumo criptográficas adicionais G e $Fmsg$, e uma função pseudoaleatória PRF. Os parâmetros do XMSS são:

- w, o parâmetro de Winternitz;
- n, o tamanho em bytes do resumo da mensagem e também de cada nó;

Algorithm 19 $WOTSpkFromSig$[23]

Entrada: a mensagem M, a assinatura $sigOts$, os valores $Adrs$ e $Seed$, os parâmetros w, t, $t1$ e $t2$.
Saída: a chave pública temporária tmp_pk.

$csum = 0$;
$msg = baseW(M, w, t1)$;// converte a mensagem para a base w
for $(i = 0; i < t1;\ i++)$ **do** // computa a soma de verificação
  $csum = csum + w - 1 - msg[i]$;
end for
$csum = csum << (8 - ((t2.log(w))\%8))$;// converte csum para a base w
$t2\_bytes = \lceil (t2.log(w))/8 \rceil$;
$msg = \{msg||baseW(csum_{t2\_bytes}, w, t2)\}$;
for $(\ i = 0; i < t;\ i++\ )$ **do**
  $Adrs[4] = i$; //chainAdrs=i
  $tmp\_pk[i] = chain(sig[i], msg[i], w - 1 - msg[i], Seed, Adrs)$;
end for
return tem_pk;

- h, a altura da árvore (número de níveis - 1).

Uma árvore binária é construída para gerar a chave pública. A árvore do XMSS é uma modificação da árvore de Merkle. Cada par de chaves pública/privada do XMSS é associada com uma folha da árvore binária, onda cada nó contém um valor de n bytes. Cada folha da árvore é construída com a chave pública do WOTS$^+$. Cada nó não folha é calculado pela concatenação dos valores dos nós de seus filhos, calculando o XOR com um vetor de máscara de bits antes de aplicar a função de resumo. Os vetores de máscara de bits e as chaves para a função de resumo G são gerados de uma semente (pública) que é parte da chave pública usando uma função pseudoaleatória PRF. O valor correspondente a raiz da árvore do XMSS junto com a semente, forma a chave pública do XMSS.

Função de Resumo Aleatório da Árvore (RandHash)

Em [23] foi introduzido a função $RandHash(noEsq, noDir, Seed, Adrs)$ que randomiza o resumo na árvore. Os valores $noEsq$ e $noDir$ representam os nós esquerdo e direito da árvore. A Função $RandHash$ (Algoritmo 20) chama PRF para gerar a chave Key e os vetores de máscara de bits de n bytes $bm0$, $bm1$. Esses valores são usados para gerar um nó pai da árvore, a partir de seus filhos esquerdo (noEsq) e direito (noDir).

Geração da Chave Privada do XMSS

As chaves privadas podem ser geradas utilizando um gerador pseudoaleatório para reduzir os custos de armazenamento. Uma chave privada SK do XMSS contém 2^h chaves privadas WOTS$^+$, o índice da folha $indSig$, uma chave $skPrf$, uma chave Key, o valor da raiz de n bytes, uma semente pública $Seed$ de n bytes.

Os valores: $Seed$, Key e $skPrf$ são cadeias de n bytes geradas aleatoriamente. A semente $Seed$ é utilizada para gerar o vetor de máscara de bits e as chaves da função

Algorithm 20 Função RandHash[23]

Entrada: os nós $noEsq$ e $noDir$; os valores $Seed$ e $Adrs$.
Saída: o valor do resumo randomizado de n bytes.

$csum = 0$;
$Adrs[6] = 0$;//KeyAndMask=0
$Key = PRF(Seed, Adrs)$;
$Adrs[6] = 1$;//KeyAndMask=1
$bm0 = PRF(Seed, Adrs)$;
$Adrs[6] = 2$;//KeyAndMask=2
$bm1 = PRF(Seed, Adrs)$;
return $G(Key, (noEsq\ XOR\ bm0)||(noDir\ XOR\ bm1))$;

de resumo. A chave Key é usada para gerar valores pseudoaleatórios para randomizar o resumo da mensagem. A geração da raiz é descrita na seção de geração da chave pública.

Árvore Ltree

Para gerar as folhas da árvore do XMSS, uma árvore Ltree é construída. As chaves públicas de verificação WOTS$^+$ $(u[0], \ldots, u[t-1])$ são as folhas da Ltree. A Ltree é uma árvore binária de valores de resumos não balanceada, onde os nós internos são calculados como na árvore do XMSS. Esses nós internos são calculados pela concatenação dos valores de seus nós filhos, calculando um XOR com os vetores de máscara de bits e em seguida aplicando a função de resumo no resultado. Os vetores de máscara de bits (bm) são escolhidos aleatoriamente. Esses vetores de máscara de bits são a principal diferença das outras árvores de Merkle, e permitem substituir a necessidade de famílias de funções de resumo resistentes à colisão por uma família de funções de resumo resistentes à segunda pré-imagem.

Se t não é uma potência de 2, então não há folhas suficientes para construir uma árvore binária. Então, um nó que não tem irmão direito, é elevado ao nível superior até que ele se torne irmão direito de outro nó. O Algoritmo 21 descreve o algoritmo da Ltree.

Árvore Binária do XMSS

Para geração da chave pública do XMSS, uma árvore binária do XMSS é construída. A árvore binária do XMSS é uma modificação da árvore MSS.

A seguir, o Algoritmo da árvore do XMSS (Algoritmo 22) constrói uma árvore binária. Este algoritmo recebe uma chave secreta SK, (que inclui a semente $Seed$), o índice do nó inicial s, uma altura t e um endereço $Adrs$. O Algoritmo 22 faz chamadas ao Algoritmo 8 começando pela folha mais a esquerda. Os nós internos são calculados através na execução da sub-rotina treeHash (Algoritmo 22). Uma pilha recebe os valores dos nós internos que são calculados utilizando as operações empilha e desempilha. O algoritmo retorna o nó raiz ($root$) para altura especificada (t).

Algorithm 21 Ltree [23]

Entrada: a chave de verificação pk, os valores $Seed$ e $Adrs$.
Saída: uma folha da árvore do XMSS $u[0]$.

$l = t$;
$Adrs[4] = 0$;//treeHeight=0
while $(l > 1)$ **do**
  for $(i = 0; i < \lfloor l/2 \rfloor$; i++) **do**
    $Adrs[5] = i$;//treeIndex=i
    $u[i] = RandHash(u[2i], u[2i+1], Seed, Adrs)$;
  end for
  if $(l\%2 == 1)$ **then**
    $u[\lfloor l/2 \rfloor] = u[l-1]$;
  end if
  $l = \lceil l/2 \rceil$;
  $Adrs[4] = Adrs[4] + 1$;//treeHeight=treeHeight+1
end while
return $u[0]$;

Algorithm 22 Algoritmo da árvore do XMSS (*treehash*) [23]

Entrada: a chave privada SK, o índice inicial s, a altura t e o valor $Adrs$.
Saída: o nó raiz de n bytes.

if $(s\%(1 << t) \neq 0)$ **then return** -1;
end if
for $(i = 0; i < 2^t$; i++) **do**
  Obtenha $Seed$ de SK;
  $Adrs[2] = 0$; //tipo = endereço OTS
  $Adrs[3] = s + i$;//OTSAddress=s+i
  $pk = WOTSgenPK(SK[s+i], Seed, Adrs)$;
  $Adrs[2] = 1$; //tipo = endereço da Ltree
  $Adrs[3] = s + i$; //lteeAddress=s+i
  $no = Ltree(pk, Seed, Adrs)$;
  $Adrs[2] = 2$; //tipo = endereço da árvore
  $Adrs[4] = 0$; //treeHeight=0
  $Adrs[5] = i + s$; //treeIndex=i+s
  while (O nó no topo da pilha tem a mesma altura que no) **do**
    $Adrs[5] = (Adrs[5] - 1)/2$;//(treeIndex=treeIndex-1)/2
    $no = RandHash(Stack.pop(), no, Seed, Adrs)$;
    $Adrs[4] = Adrs[4] + 1$; //treeHeight++
  end while
  $Stack.push(no)$;
end for
return $Stack.pop()$;

Geração das Chaves do XMSS

O Algoritmo $XMSSKeyGen$ (Algoritmo 23) descreve o processo de geração do par de chaves do XMSS. As chaves privadas podem ser geradas usando um gerador pseudoaleatório para reduzir os custos de armazenamento, conforme descrito na Seção 4.2.5. A chave pública é gerada chamando o Algoritmo 22. A chave pública do XMSS PK consiste da raiz da árvore binária e da semente $Seed$, que também são armazenados em SK. Os endereços $Adrs$ são definidos inicialmente com zero, porque no XMSS há somente uma árvore principal.

Algorithm 23 $XMSSKeyGen$ [23]

Entrada: a altura h.
Saída: a chave secreta SK, a chave pública do XMSS PK.

```
Gerar SeedSK aleatoriamente;
for (i = 0; i < 2^h; i++ ) do
    Calcular S[i] = PRF(SeedSK, i);
    skWOTS[i] = WOTSgenSK(S[i]);
end for
Gerar skPrf aleatoriamente;
Gerar Seed aleatoriamente;
root = treeHash(skWOTS, 0, h, Adrs);
indSig = i;
SK = {indSig||SeedSK||skPrf||root||Seed};
PK = {root||Seed};
return {SK||PK};
```

Geração de Assinatura do XMSS

A geração de assinatura do XMSS consiste de dois algoritmos. Primeiro, $XMSSsign$ (Algoritmo 25) gera um resumo randomizado da mensagem M' usando um valor aleatório r, o índice $indSig$ do par de chaves do WOTS$^+$, e uma chave que é o valor da raiz da chave pública. O Algoritmo $XMSSsign$ chama o procedimento treeSig e atualiza a chave secreta. O procedimento treeSig (Algoritmo 24), calcula a assinatura do WOTS$^+$ do resumo da mensagem de n bytes usando a chave privada WOTS$^+$. Então, o caminho de autenticação é calculado e o índice da chave secreta é atualizada em uma unidade.

Os nós do caminho de autenticação podem ser calculados de alguma forma. O primeiro caminho de autenticação é gerado durante a execução do Algoritmo 23, na função treeHash. Para gerar o próximo caminho de autenticação, nós usamos o Algoritmo BDS [4] que é uma modificação do Algoritmo clássico do caminho de autenticação proposto por Merkle [39].

O Algoritmo 24 tem como entrada a mensagem M', a chave secreta do XMSS SK, e o endereço $Adrs$. Ele retorna a concatenação da assinatura $sigOts$ e o caminho de autenticação aut.

O Algoritmo $XMSSsign$ (Algoritmo 25), calcula a assinatura da mensagem M e uma atualização da chave secreta SK. $XMSSsign$ recebe como entrada uma mensagem

Algorithm 24 treeSig [23]

Entrada: uma mensagem M', a chave secreta SK e o valor $Adrs$.
Saída: a assinatura Sig.

Obtenha $Seed$ e $indSig$ de SK;
$aut = buildAuth(SK, indSig, Adrs)$;
$Adrs[2] = 0$; //tipo = endereço OTS
$Adrs[3] = indSig$; //otsAdrs=indSig
$sigOts = WOTS\_sign(SK[indSig], M', Seed, Adrs)$;
$Sig = \{sigOts||aut\}$;
return Sig;

M de tamanho arbitrário, e uma chave secreta do XMSS SK. Ele retorna a concatenação da chave secreta atualizada SK e a assinatura Sig.

Algorithm 25 $XMSSsign$ [23]

Entrada: a mensagem M, a chave secreta SK.
Saída: a chave secreta SK atualizada e a assinatura Sig.

Obtenha $indSig$, $skPrf$ e $root$ de SK;
$r = PRF(skPrf, indSig)$;
$M' = Fmsg((r||root||indSig_n), M)$;
$Sig = \{indSig||r||treeSig(M', SK, Adrs)\}$;
$SK.indSig = indSig + 1$;
return $\{SK||Sig\}$;

Verificação de Assinatura do XMSS

A verificação de assinatura também foi dividida em dois passos. Primeiro, para verificar a assinatura Sig, o Algoritmo 27, $XMSSverify$, calcula o resumo da mensagem (M') como na assinatura. O Algoritmo $XMSSrootFromSig$ (Algoritmo 26) tem como entrada o índice $indSig$, a assinatura $sigOts$, um caminho de autenticação aut, um resumo da mensagem M', a semente $Seed$, e o endereço $Adrs$. $XMSSrootFromSig$ retorna o valor da raiz da árvore de n bytes.

No segundo passo, $XMSSverify$ (Algoritmo 27) chama $XMSSrootFromSig$ para validar a chave pública da árvore do XMSS. Então, a chave pública $pkOts$ é calculada a partir da assinatura do WOTS$^+$, e é usada para calcular a folha usando a Ltree. A folha junto com o índice $indSig$ e o caminho de autenticação aut são usados para calcular a raiz da árvore $no[0]$.

O Algoritmo 27, $XMSSverify$, recebe a assinatura Sig do XMSS, a mensagem M, e a chave pública PK do XMSS. $XMSSverify$ retorna VÀLIDO se, e somente se, o valor de $no[0]$ é o mesmo valor da chave pública do XMSS. Caso contrário retorna INVÀLIDO.

4.2.6 Esquema de Assinatura Hierárquico - XMSS-MT

O XMSS Multi Árvore (XMSS-MT) [26] permite assinar um grande mas fixo número de mensagens e resolve o problema de geração lenta das chaves. O XMSS-MT é seguro quando instanciado com uma família de funções de resumo resistente à segunda pré-imagem G_n

Algorithm 26 $XMSSrootFromSig$ [23]

Entrada: a assinatura Sig, a mensagem M' e os valores $Seed$ e $Adrs$.
Saída: valor da raiz $no[0]$.

$Adrs[2] = 0$; // tipo = endereço OTS
$Adrs[3] = indSig$; //otsAdrs = indSig
$pkOts = WOTSpkFromSig(sigOts, M', Seed, Adrs)$;
$Adrs[2] = 1$; //tipo = endereço da Ltree
$Adrs[3] = indSig$; //lTreeAdrs=indSig
$no[0] = Ltree(pkOts, Seed, Adrs)$;
$Adrs[2] = 2$; // tipo = endereço da árvore
$Adrs[5] = indSig$;//TreeIndex=indSig
Obtenha aut de Sig;
for ($k = 0; k < h$; k++) **do**
  $Adrs[4] = k$;//treeHeight=k
  if ($(\lfloor indSig/(2^k) \rfloor \% 2) == 0$) **then**
    $Adrs[5] = Adrs[5]/2$; //TreeIndex=TreeIndex/2
    $no[1] = RandHash(no[0], aut[k], Seed, Adrs)$;
  else
    $Adrs[5] = (Adrs[5] - 1)/2$;//TreeIndex=(TreeIndex−1)/2;
    $no[1] = RandHash(aut[k], no[0], Seed, Adrs)$;
  end if
  $no[0] = no[1]$;
end for
return $no[0]$;

Algorithm 27 $XMSSverify$ [23]

Entrada: assinatura Sig, a mensagem M, a chave pública PK.
Saída: resultado da verificação.

Obtenha $root$ e $Seed$ de PK;
$Adrs = 0_{32}$;
$M' = Fmsg((r||root||indSig_n), M)$;
$no = XMSSrootFromSig(indSig, sigOts, aut, M', Seed, Adrs)$;
if ($no == root$) **then**
  return VÁLIDO;
else
  return INVÁLIDO;
end if

e outra família de funções pseudoaleatória F_n, como no XMSS. O XMSS-MT usa muitas camadas de árvores do XMSS como no GMSS [3].

O número de camadas é $d \in N$ onde $0 \leq i < d-1$. As folhas da árvore na camada mais baixa $i = d$ são usadas para assinar as mensagens. A árvore no topo e nas camadas intermediárias para $i = 1, \ldots, d-1$, são usadas para assinar as raízes das árvores na camada abaixo. A chave pública é a raiz da camada $i = 1$. Considerando a altura total h com d camadas, então cada subárvore tem altura $h_{sub} = h/d$. Todas as árvores do XMSS tem a mesma altura e o mesmo parâmetro de Winternitz. O XMSS-MT usa o algoritmo WOTS$^+$ como no XMSS.

Considere uma árvore do XMSS-MT de altura total h que tem d camadas de árvores do XMSS de altura h/d. Então a camada $d-1$ contém uma árvore do XMSS, a camada $d-2$ contém $2^{(h/d)}$ árvores do XMSS, e assim por diante.

Geração de Chaves do XMSS-MT

Uma chave privada $skMT$ do XMSS-MT consiste de uma chave privada reduzida do XMSS para cada árvore do XMSS. Essas chaves privadas reduzidas contém a chave privada WOTS$^+$ correspondente ao par de chaves do XMSS e uma função chave pseudoaleatória. O índice, a semente pública e o nó raiz não fazem parte da chave privada reduzida. Em vez disso, $skMT$ contém uma única chave $skPrf$, um único índice $indMT$ de $\lceil (h/8) \rceil$ bytes, uma semente $Seed$ de n bytes, e uma única raiz $ROOT$ que é a raiz da árvore na camada superior. O índice é um valor global para todos os pares de chaves do WOTS$^+$ de todas as árvores do XMSS na camada 0. Este índice é inicializado com 0 e armazena o índice da última chave do WOTS$^+$ usada na camada inferior, isto é, um índice entre 0 e $2^h - 1$.

O Algoritmo 33 no Apêndice, ilustra a geração de chaves do XMSS-MT. Para cada subárvore, o Algoritmo 33 faz chamadas ao Algoritmo 23 (treehash), para calcular as raízes das subárvores. Esse processo é repetido até encontrar a chave pública que é a raiz da árvore do topo.

Geração da Assinatura do XMSS-MT

Para gerar a assinatura do resumo da mensagem M, o Algoritmo $XMSSMTsign$, recebe como entrada a mensagem M e chave secreta $skMT$. A assinatura da mensagem é assinada utilizando o par de chaves do WOTS$^+$ da camada mais baixa.

O Algoritmo 34 no Apêndice descreve o processo de assinatura do XMSS-MT. Para cada camada, calcula-se o caminho de autenticação e a raiz da subárvore. Cada raiz na camada cm é assinada com a folha da árvore da camada $cm-1$, conforme descrito pelos algoritmos dos esquemas Multi Árvores. Este processo é repetido até a árvore do topo ser alcançada. A assinatura do XMSS-Mt é:

$$sigMT = \{indSig||r||sigOts_0||Auth_0||sigOts_1||Auth_1||\ldots||sigOts_{d-1}||Auth_{d-1}\}.$$

Verificação de Assinatura do XMSS-MT

O processo de verificação de assinatura do XMSS-MT é semelhante ao do XMSS, com pequenas mudanças. Primeiro é gerado o resumo da mensagem M. Então, em vez de comparar o nó da raiz calculado com um determinado valor, a assinatura de cada nó raiz é verificada. Apenas o nó raiz do nível superior é comparada com o valor da chave pública do XMSS-MT.

O Algoritmo 35 no Apêndice descreve o processo de verificação do XMSS-MT. Para cada camada, executa o Algoritmo $XMSSMTverify$.

A função $getXMSSSignature(sigMT, i)$ retorna a $i-$ésima assinatura reduzida do XMSS a partir da assinatura do XMSS-MT. O Algoritmo $XMSSMTverify$ toma como entradas uma assinatura $sigMT$ do XMSS-MT, uma mensagem M e uma chave pública $pkMT$. $XMSSMTverify$ retorna VÁLIDO se, e somente se, $sigMT$ for uma assinatura válida da mensagem M sob a chave pública $pkMT$. Caso contrário, ele retorna INVÁLIDO.

4.2.7 Segurança do XMSS

Os autores em [23], apresentam algumas considerações de segurança propostas para o XMSS. O XMSS fornece fortes garantias de segurança e é considerado seguro mesmo quando a resistência de colisão da função de resumo subjacente for quebrada. O esquema usa funções de resumo parametrizadas pelo parâmetro de segurança n, que é o tamanho do valor do resumo da mensagem. O valor de n é determinado pela função de resumo criptográfica utilizada e é escolhido para assegurar um nível adequado de segurança. Os valores do bit de segurança indicados foram estimados com base na complexidade dos ataques genéricos mais conhecidos contra a segurança da função de resumo utilizada e das funções pseudo aleatórias, assumindo adversários convencionais e quânticos.

O conjunto de parâmetros definido em [23], visa atender as aplicações mais relevantes. Considerando os computadores clássicos, os parâmetros com $n = 32$, fornecem um nível de segurança de 256 bits e os parâmetros com $n = 64$ fornecem um nível de segurança de 512 bits. Considerando ataque por computador quântico, $n = 32$ oferece segurança de 128 bits e $n = 64$ fornece segurança de 256 bits. Para as configurações $n = 32$ e $n = 64$, são indicadas as funções de resumo SHA-256 e SHA-512 e as funções de saída extensíveis SHAKE128, SHAKE256, SHA3/Keccak.

A proposta de padrão propões alguns ajustes ao XMSS. O primeiro ajuste visa impedir ataques multi-usuário/multi-alvo contra $Fmsg$. A chave pública e o índice do par de chaves tornam-se parte da chave da função de resumo. Assim, pode-se alcançar uma separação de domínio que força um atacante decidir qual valor de resumo atacar.

Além do resumo inicial da mensagem, o XMSS randomiza cada chamada da função de resumo, isso significa que, cada chamada da função de resumo usa uma chave diferente e um vetor de máscara de bits diferente. A prova da segurança é baseada na prova apresentada em [27], que considera uma compressão inicial da mensagem. Para os esquemas originais, estas provas mostram que um atacante tem de quebrar certas propriedades mínimas de segurança. Em particular, não é suficiente quebrar a resistência à colisão das funções de resumo para gerar uma falsificação. Mais especificamente, as exigências sobre

as funções utilizadas são de que F e G sejam funções de resumo chaveadas pós-quântica, multi-alvo, e resistentes à pré-imagem. A função F cumpre uma exigência adicional onde a maioria das imagens tem pelo menos duas pré-imagens. A função PRF é uma função pseudoaleatória pós-quântica, $Fmsg$ é uma função de resumo pós-quântica, multi-alvo e resistente à colisão.

A resistência à segunda pré-imagem multi-função e multi-alvo é uma extensão da resistência à segunda pré-imagem da função de resumo com chave, cobrindo o caso onde um adversário consegue encontrar uma segunda pré-imagem para um dos muitos valores. A prova em [27] divide PRF em duas funções. Quando PRF é usado para geração de chaves pseudoaleatórias ou quando e usado para a geração de aleatoriedade para o resumo randomizado da mensagem. Sempre que a função PRF é usada para gerar os vetores de máscara de bits ou chaves para a função de resumo, ele é modelado como um oráculo aleatório.

4.3 Diferenças Fundamentais entre o esquema LMS e o XMSS

Nesta seção são descritas as principais características das propostas de padrão do esquema LMS [37] e do XMSS[23]. Nós comparamos os tamanhos das chaves secretas, das chaves públicas e das assinaturas de cada esquema. Comparamos também a eficiência das funções utilizadas em relação ao tamanho da mensagem de entrada e ao número de vezes em que a função é executada.

Tamanho das Chaves dos Esquemas

A seguir, na Tabela 4.1, são descritos os tamanhos das chaves para uma função de resumo de n bytes e uma árvore de altura h. O número de elementos da assinatura OTS é denotado como p no esquema LMS e t no XMSS.

Tabela 4.1: Tamanho das chaves do esquema LMS e do XMSS em bytes.

	LMS	**XMSS**
chave secreta (SK)	$2n+4$	$4n+4$
chave pública (PK)	$3n$	$2n$
Assinatura (Sig)	$(p+1+h)n+4$	$(t+h+1)n+4$

O tamanho das chaves do esquema LMS são especificados a seguir:

- $SK = \{q||SeedSK||SeedI\}$, dado que o índice q tem 4 bytes, cada semente para gerar a chave secreta e o identificador I tem n bytes.
- $PK = \{I||no[1]\}$, contém o identificador I de $2n$ bytes e a raiz da árvore ($no[1]$) de n bytes.
- $Sig = \{q||sigOts||aut[0]||\ldots||aut[h-1]\}$, contém o índice q com 4 bytes, a assinatura do WOTS contém um valor aleatório C com n bytes e sig com pn bytes e o caminho de autenticação tem hn bytes.

Os tamanhos das chaves do XMSS são descritos a seguir:

- $SK = \{ind||SeedSK||skPrf||root||Seed\}$, contém um índice ind com 4 bytes, a chave secreta $SeedSK$ de n bytes, uma chave $skPrf$ de n bytes, a raiz $root$ de n bytes e a semente $Seed$ de n bytes.
- $PK = \{root||Seed\}$, onde root e Seed tem n bytes.
- $Sig = \{indSig||r||sigOts||aut[0]||\ldots||aut[h-1]\}$, contém o índice $indSig$ com 4 bytes, um valor aleatório r com n bytes, a assinatura do WOTS$^+$ com tn bytes e um caminho de autenticação com hn bytes.

A Tabela 4.2 mostra o tamanho das chaves dos esquemas LMS e XMSS para uma árvore com altura $h = 20$ e nível de segurança de 128 bits ($n = 32$ bytes).

Se $w = 4$ no esquema LMS e $w = 16$ no XMSS, então o tamanho da assinatura OTS é $t = 67$ para ambos esquemas.

Tabela 4.2: Comparação do tamanho das chaves do esquema LMS e do XMSS para 2^{20} assinaturas.

	LMS	**XMSS**
SK	68	132
PK	96	64
Sig	2820	2820

Observa-se que, para os parâmetros selecionados, o esquema LMS tem chave secreta menor que do XMSS. A chave pública do esquema LMS é maior que a chave pública do XMSS. As chaves de assinatura dos dois esquemas têm o mesmo tamanho.

Funções dos Esquemas

A seguir, mostramos as diferenças entre as principais funções de resumo chaveadas de n bytes dos esquemas LMS e do XMSS. Seja F utilizada na função de encadeamento e a função G utilizada para gerar os nós internos da árvore. Seja $S = \{I||q\}$, sk a chave secreta do esquema WOTS e I,r,i,q,j cadeias de segurança definidas na Seção 4.1.1.

Na proposta do esquema LMS, as funções F e G têm os seguintes tamanhos de entrada:

- $F(S, tmp, i, j) = H(X)$, onde a entrada $X = \{S||tmp||i_{16}||j_8||DITER\}$ tem $3n + 8$ bytes.
- $G(I, noEsq, noDir, r) = H(Y)$, onde a entrada $Y = \{I||noEsq||noDir||r_{32}||DINTR\}$ tem $4n + 5$ bytes

Na proposta do XMSS, as funções F e G tem os seguintes tamanhos de entrada:

- $F(Key, M) = H(X)$, onde a entrada $X = \{0_n||Key||M\}$ tem $3n$ bytes
- $G(Key, M) = H(Y)$, onde a entrada $Y = \{1_n||Key||M\}$ tem $4n$ bytes

A Tabela 4.3 resume o tamanho das mensagens de entrada para as funções de resumo dos esquemas LMS e XMSS para o nível de segurança de 128 bits ($n = 32$ bytes). Comparamos a função F que é utilizada para gerar as chaves OTS. Também comparamos a função G do esquema LMS com a do XMSS, que são utilizadas para gerar os nós internos das árvores binárias.

Tabela 4.3: Comparação do tamanho das mensagens de entrada das funções do esquema LMS e do XMSS para $n = 32$ bytes.

Padrão	**função**	**Tamanho (bytes)**
LMS	F	104
XMSS	F	96
LMS	G	133
XMSS	G	128

Observa-se que, essas funções de resumo chaveadas têm entradas com tamanho aproximado. O bloco da função de resumo do SHA2 tem tamanho $2n$ bytes, então a função F irá executar 2 blocos no esquema LMS e no XMSS. Isto ocorre porque na proposta do esquema LMS, a função F recebe $3n + 8$ bytes de entrada e na proposta do XMSS recebe $3n$ bytes. A função G do esquema LMS e do XMSS irão executar 3 blocos da função de resumo SHA2 porque no esquema LMS a função G tem $4n + 5$ bytes de entrada e no XMSS a função G tem $4n$ bytes. Desta forma, essas funções levarão o mesmo tempo de execução em cada aplicação da função F.

Outra diferença importante nos dois esquemas é o processo de geração das folhas da árvore binária. No esquema LMS, os valores das chaves públicas $pk = \{u[0]||u[1]||...||u[p-1]\}$ são concatenados e a função de resumo $Fpub$ é executada uma única vez para gerar uma folha da árvore do esquema LMS. No XMSS, uma árvore Ltree é construída para gerar as folhas da árvore principal. O objetivo da Ltree é permitir a utilização de uma família de funções de resumo resistentes à segunda pré-imagem ao invés de uma famílias de funções de resumo resistentes a colisão. Na árvore Ltree, a função de resumo G é executada $2^h - 1$ vezes (uma vez para cada nó interno) para gerar uma folha da árvore principal, gerando um *overhead* adicional neste esquema em relação ao esquema LMS.

Capítulo 5

Implementação em Software

Este capítulo apresenta as principais otimizações na implementação das propostas de padrão do esquema de Merkle realizadas neste trabalho e a microarquitetura dos computadores utilizados. Este capítulo está organizado como segue: Seção 5.1 descreve a microarquitetura dos computadores utilizados. Seção 5.2 descreve aspectos da otimização em software aplicados para estas microarquiteturas. As otimizações para a função de resumo SHA2 são apresentadas na Seção 5.3 e para a função de resumo SHA3 na Seção 5.4. A Seção 5.5 mostra as otimizações para os drafts do LMS e do XMSS.

5.1 Microarquitetura

Nesta seção, são descritos alguns detalhes da microarquitetura dos processadores (Haswell e Skylake) utilizados neste trabalho. A microarquitetura Haswell da Intel, lançada em 2013, suporta o conjunto de instruções vetoriais AVX2, que expandiu as instruções de aritmética inteira nos registradores de 128 para 256 bits. A microarquitetura Skylake, lançada em 2015, baseia-se na microarquitetura Haswell e Broadwell [28]. Esta microarquitetura melhorou a latência das instruções. A seguir, são descritos os aspectos gerais dessas microarquiteturas apresentando-se as características específicas e as instruções mais relevantes que foram utilizadas.

Operações em vetores de dados

No fim da década de 1990, os fabricantes de processadores focaram seus esforços em explorar o paralelismo de dados ao invés do paralelismo de instruções. Para isso, incorporaram unidades funcionais que são capazes de executar uma única instrução sobre um conjunto de dados. Esse processamento encaixa-se no paradigma de programação paralela, conhecido como SIMD [28].

Um dos primeiros conjuntos de instruções a implementar o paradigma SIMD foi lançado em 1997, conhecido como *Multimedia eXtensions* (MMX). O MMX adicionou registradores de 64 bits e instruções vetoriais que habilitavam o processamento de duas operações de 32 bits. Nessa época, as microarquiteturas possuíam registradores nativos de 32 bits [28].

Em 1999, a Intel lançou o *Streaming SIMD Extensions* (SSE) que incluiu oito registradores de 128 bits (XMMs). O número de registradores foi duplicado em 2000, quando o tamanho dos registradores nativos aumentou de 32 para 64. Nos anos seguintes o SSE evoluiu com o lançamento dos novos conjuntos de instruções SSE2, SSE3 e SSE4.

Em 2011, foi lançado o conjunto de instruções *Advanced Vector Extensions* (AVX), que trouxe contribuições relevantes para a arquitetura. A arquitetura incluía registradores de 256 bits, chamados YMMs, que são sobrepostos nos registradores XMMs. O AVX introduziu um novo formato de codificação que permite o uso de código de montagem de três operandos, tornando a atribuição de registradores mais flexível.

Um registrador de 128 bits (XMM) pode ser organizado como um vetor de 8 inteiros de 16-bits ou 4 inteiros de 32 bits [28]. Um registrador de 256 bits (YMM) pode ser organizado como um vetor de 8 inteiros de 32 bits ou 4 inteiros de 64 bits. O conjunto de instruções AVX2 também permite vetores de inteiros de 256 bits.

Essas operações sobre vetores são úteis quando são feitos cálculos sobre um grande conjunto de dados de modo que a mesma operação é realizada sobre múltiplos elementos de dados e a lógica do programa permite cálculos em paralelo. Algoritmos de criptografia, que fazem cálculos sobre um grande conjunto de dados, podem se beneficiar do uso de vetores de instruções para executar operações em paralelo utilizando SIMD.

O código, que é compilado para o conjunto de instruções AVX, só pode ser executado se o processador suporta AVX. Compiladores como GCC, Clang e ICC podem realizar operações sobre vetores automaticamente (sem a interferência do programador). Uma desvantagem é que é difícil predizer quando o compilador irá produzir um código vetorizado eficiente. É possível vetorizar o código explicitamente usando as funções intrínsecas. Essas funções são operações primitivas no sentido que cada função intrínseca é traduzida para uma ou mais instruções de máquina. A ideia é carregar os elementos do vetor nos registradores XMM ou YMM para então executar as operações sobre vetores.

Haswell

A microarquitetura Haswell, ou comercialmente conhecida como a família de processadores Core Intel 4ª geração, foi lançada no início de 2013 e trouxe consigo uma série de inovações para acelerar a execução dos programas. O Haswell melhora as características de instruções críticas em relação às microarquiteturas Intel anteriores chamadas Sandy Bridge e Ivy Bridge. A microarquitetura Haswell vem com algumas melhorias básicas para o desempenho em criptografia como novas instruções e melhorias na microarquitetura. As Instruções de Manipulação de Bits aumentam o desempenho do SHA e do RSA.

O aumento de 3 para 4 unidades lógicas e aritméticas no Haswell também beneficia o desempenho de operações aritméticas. O AVX2 contém novas instruções que expandem a computação de aritmética inteira nos registradores de 256 bits, pois o conjunto AVX tinha apenas instruções para a aritmética de ponto flutuante. Além disso, o AVX2 conta com instruções de permutação e combinação que permitem movimentar as palavras contidas nos registradores vetoriais, entre outras características [16].

Skylake

A microarquitetura Skylake, baseada na microarquitetura Haswell e Broadwell, foi lançada em 2015 [28]. O Skylake oferece as seguintes melhorias para o desempenho dos programas: maiores *buffers* internos, maior largura de banda do cache, melhor *branch predictor*, maior vazão, baixo consumo de energia, balanceamento da vazão e da latência de pontos flutuantes `ADD, MUL, FMA`. Uma parcela significativa das instruções SSE, AVX e de uso geral também tiveram redução de latência [28].

Instruções Relevantes

Nesta seção, são descritos aspectos que influenciam no tempo de execução das instruções. Como descrito acima, uma única instrução AVX2 pode operar oito valores de 32 bits ao mesmo tempo, em vez de apenas um valor de 32 bits. Desta forma, esta instrução permite que a abordagem SIMD processe $8X$ (8-*way*) a quantidade de dados por instrução. Porém, o ganho real quando a aplicação é executada em um SIMD 8-*way* pode ser inferior a $8X$, pois o tempo de execução do código sofre influências, tais como: latência das instruções, maior tempo de acesso a memória, dependências e cache misses [13].

Segundo Agner Fog [16], a latência de uma instrução é o atraso que a instrução gera em uma cadeia de dependência. A unidade de medida é ciclos do relógio. Quando a frequência do relógio é variada de forma dinâmica, os dados referem-se a frequência do relógio do núcleo.

A dependência está ligada à implementação. Se, para executar uma instrução, necessita-se aguardar o resultado de uma outra instrução, então tem-se uma dependência de dados. Uma dependência pode gerar gargalos que limitarão a possibilidade de paralelização de um algoritmo. Identificar as dependências e eliminá-las, quando possível, pode influenciar severamente no tempo de execução da aplicação. Cache *misses*, desalinhamentos e exceções podem aumentar a contagem de tempo de execução consideravelmente.

Outro fator que influencia no desempenho é a vazão. A vazão é o número máximo de instruções do mesmo tipo que podem ser executadas por ciclo do relógio quando os operandos de cada instrução são independentes das instruções anteriores.

Na Tabela 5.1 são destacadas algumas instruções que fazem parte do conjunto AVX2 e que são relevantes para o contexto da implementação eficiente do XMSS. Nesta tabela são apresentadas a latência, a vazão e as portas de execução no Haswell e no Skylake [16].

As portas 0, 1 e 5 citadas na Tabela 5.1, são as portas mais utilizadas, e portanto, as mais críticas na determinação da eficiência da implementação. Observa-se também nesta tabela que algumas instruções no Skylake têm menor latência e vazão, além de possuir mais portas de execução. Estes aspectos podem ser aproveitados para realizar otimizações na implementação.

5.2 Otimizações em Software

Um dos objetivos gerais deste trabalho é prover técnicas que viabilizem o uso eficiente dos conjuntos de instruções vetoriais na implementação da proposta de padrão dos esquemas

Haswell				**Porta de execução**							
	instrução vetorial	latência	vazão	0	1	2	3	4	5	6	7
LOAD/STORE	vmov	1	1/3	x	x				x		
SET	vinsert	4	1/2			x	x				
ADD	vpadd	1	1/2		x				x		
AND/OR/XOR	vpand/vpor/vpxor	1	1/3	x	x				x		
SL/SR	vpsll	1	1	x							
Skylake				**Porta de execução**							
	instrução vetorial	latência	vazão	0	1	2	3	4	5	6	7
LOAD/STORE	vmov	1	1/4	x	x				x		
SET	vinsert	3	1/2			x	x				
ADD	vpadd	1	1/3	x	x				x		
AND/OR/XOR	vpand/vpor/vpxor	1	1/3	x	x				x		
SL/SR	vpsll	1	1/2	x	x						

Tabela 5.1: Latência, vazão e portas de execução

LMS e XMSS. Como estes esquemas são baseados em funções de resumo, este trabalho mostra implementações eficientes que usam registradores de 256 bits para calcular quatro valores de resumo no SHA-512/SHAKE128 ou oito valores de resumo no SHA-256 de forma concorrente.

Os algoritmos de funções de resumo trabalham com fluxo de *buffers* de dados e têm uma quantidade muito grande de dados a serem processados. Geralmente, a necessidade de implementações de alto desempenho nestes algoritmos é grande. No Esquema de Assinatura de Merkle e suas variantes, o algoritmo da função de resumo é executado muitas vezes, sendo ainda mais relevante uma otimização de sua implementação.

5.3 Otimizações na implementação do SHA2

Nesta seção, são descritas as otimizações aplicadas ao algoritmo SHA2 que foram utilizadas na implementação dos esquemas LMS e XMSS. A primeira otimização para melhorar o desempenho dos dois esquemas calcula múltiplos valores de resumo ao mesmo tempo. Esta abordagem é denominada de otimizações *multi-buffer*, onde técnicas semelhantes aparecem no trabalho [53].

Como os *buffers* de dados são independentes entre si e possuem mensagens do mesmo tamanho, permitiu-se a paralelização em nível de dados dos algoritmos de função de resumo. O resultado do valor do resumo também é devolvido na mesma ordem em que foi enviado, facilitando sua implementação. Uma vez que os dados são carregados nos registradores, eles são processados várias vezes pela função de resumo, porque as funções dos esquemas one-time executam muitas vezes sobre os mesmos dados. Este fato, permitiu tirar proveito da microarquitetura da máquina ao executar várias iterações do algoritmo de função de resumo sobre os mesmos dados.

A segunda otimização foi implementada com base nas portas de execução dos processadores, analisando a ordem em que as instruções das funções de resumo são executadas. Foram observados aspectos que reduzem a cadeia de dependência das instruções.

5.3.1 Otimização *Multi-buffer*

Uma forma direta de usar as instruções vetoriais para otimizar a implementação da função SHA2 é processar mais de um valor de resumo por vez. As instruções vetoriais de 256 bits podem processar quatro palavras de 64 bits ou oito palavras de 32 bits usando apenas uma instrução. O algoritmo SHA-512 trabalha internamente com palavras de 64 bits e o SHA-256 com palavras de 32 bits. Logo, tirando proveito dos registradores de 256 bits, pôde-se processar quatro valores de resumo concorrentemente para o SHA-512 e oito valores de resumo para o SHA-256.

O Pseudocódigo a seguir é referente ao algoritmo do SHA-256, descrito na Seção 2.3.1. O primeiro laço controla o número de blocos (k) em que uma mensagem é dividida. Cada bloco $B[i]$ da mensagem, executa outros dois laços internos principais. Esses laços internos são executados 64 vezes para cada bloco da mensagem. Assim, ao executar os blocos de 4 ou 8 mensagens em paralelo, o tempo de execução é reduzido.

Antes de iniciar o cálculo do valor de resumo, valores iniciais H da função SHA2, são guardados nas variáveis de a,...,h. Ex: Sejam $\{M, N, O, P, Q, R, S, T\}$ oito mensagens de tamanho de 256 bits. Deseja-se gerar o valor de resumo dessas oito mensagens SHA2(M), SHA2(N),..., SHA2(T).

Algorithm 28 Pseudocódigo do Algoritmo SHA2 [42]

Entrada: a mensagem M, valores iniciais de $H_0, \ldots, H_7$.
Saída: valor do resumo da mensagem $H_0, \ldots, H_7$.

```
k=quantidade de blocos da mensagem M;
for (i = 1;i < k;i = i + 1) do
    for (t = 0;t < 63;t = t + 1) do
        if (t ≤ 15) then
            W[t] = B[i][t];
        else
            σ_0^256(W[t − 15]) = ROTR^7(W[t − 15]) ⊕ ROTR^18(W[t − 15]) ⊕ ROTR^3(W[t − 15]);
            σ_1^256(W[t − 2]) = ROTR^17(W[t − 2]) ⊕ ROTR^19(W[t − 2]) ⊕ ROTR^10(W[t − 2]);
            W[t] = σ_1^256(W[t − 2]) + W[t − 7] + σ_0^256(W[t − 15]) + W[t − 16];
        end if
    end for
    ...
    for (t = 0;t < 63;t = t + 1) do
        T_1 = h + Σ_1^256(e) + Ch(e, f, g) + K_t^256 + W[t];
        T_2 = Σ_0^256(a) + Maj(a, b, c);
        h = g;
        g = f;
        f = e;
        e = d + T_1;
        d = c;
        c = b;
        b = a;
        a = T_1 + T_2;
    end for
    ...
end for
```

A seguir, são apresentadas as otimizações aplicadas para que este cálculo seja efetuado em paralelo.

Etapa de Inicialização

O primeiro passo para a implementação foi mapear as oito mensagens (M, N, ..., T) a serem processadas nos registradores de 256 bits. O lado esquerdo da Figura 5.1 mostra cada registrador (wi) recebendo uma mensagem de 256 bits ($w0 = M, w1 = N, \ldots, w7 = T$). A instrução *vmov* foi utilizada para esta operação.

Como as mensagens tem 256 bits, particiona-se cada mensagem em um vetor com 8 palavras de 32 bits (Ex: $M = M_0, M_1, \ldots, M_7$). Para que os valores $W[t]$ no Algoritmo 28 possam ser executados ao mesmo tempo para as 8 mensagens, todas as posições t de cada mensagem deverão estar no mesmo registrador. Esse mapeamento das posições da mensagem pode ser visto no lado direito da Figura 5.1, que é a matriz transposta da matriz do lado esquerdo. Os primeiros 32 bits de cada uma das 8 mensagens foram organizados de forma que ficassem no mesmo registrador para que fossem processados ao mesmo tempo.

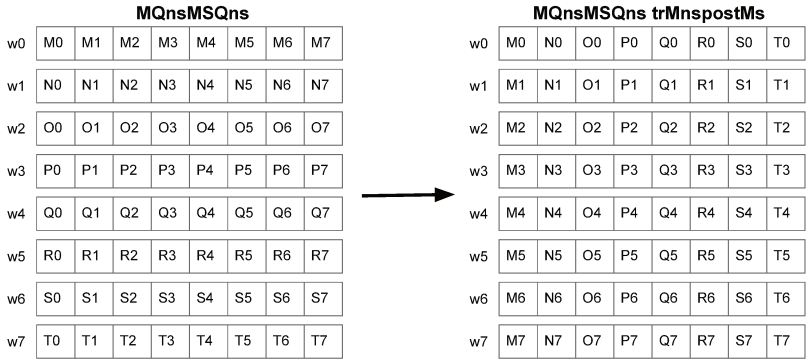


Figura 5.1: Transposição dos bits de oito mensagens de 256 bits (8x32 bits)

Após a transposição das mensagens, inicia-se então a etapa de preprocessamento, na qual o *padding* é acrescentado às mensagens. Na Figura 5.2, as etapas de preprocessamento e inicialização das variáveis H_j são demonstradas, para $0 \leq j \leq 7$. O padding e as oito variáveis do SHA2(a,...,h) foram inicializados utilizando a função *vinsert*, para atribuir o mesmo valor para as oito mensagens.

Uma vez mapeadas e inicializadas as mensagens nas variáveis de 256 bits, pode-se começar o processamento em paralelo. Por exemplo, no cálculo de $W[t]$:

$$W[t] = \sigma_1^{256}(W[t-2]) + W[t-7] + \sigma_0^{256}(W[t-15]) + W[t-16],$$

as operações com os valores de $W[t]$ para as oito mensagens podem ser efetuadas em paralelo utilizando os registradores $w[t]$ de 256 bits. Cada $W[t]$ recebe e processa valores de 32 bits. Assim, é possível computar em $w[t]$ as operações para o cálculo de 8 valores de resumo ao mesmo tempo.

Se $t = 16$, então $w[16] = \sigma_1^{256}(w[14]) + w[9] + \sigma_0^{256}(w[1]) + w[0]$.

Figura 5.2: Inicialização do SHA2

.

As operações dessa instrução são executadas para as oito mensagens ao mesmo tempo, pois os valores de $w[14]$ das oito mensagens estão no mesmo registrador. O mesmo ocorre com os valores de $w[9], w[1]$ e $w[0]$.

Quando a instrução `vpadd(`$w[0]$`,` $w[9]$`)` é executada, $W[0]$ e $W[9]$ das oito mensagens são somados ao mesmo tempo.

Por este método de paralelização, as oito mensagens $(M, N, \ldots, T)$ são executadas ao mesmo tempo pela função de resumo, conforme visualizado na Figura 5.3. Ao final, as oito variáveis do SHA2(a,...,h), são atualizadas e têm o valor do resumo das oito mensagens.

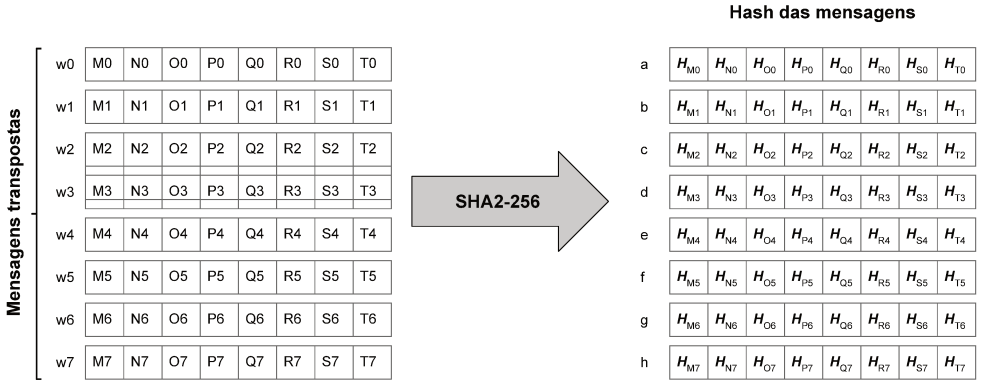


Figura 5.3: Execução do SHA2

5.3.2 Otimizações Baseadas nas Portas de Execução dos Processadores

Para implementar a segunda otimização, o código das funções que calculam o valor de resumo das mensagens foi reformulado organizando a execução das instruções a fim de melhorar o desempenho da implementação. Esta organização visa reduzir o tempo de execução eliminando diversas dependências no código do algoritmo do SHA2. Para isto, foi efetuada uma ordenação no código de forma que a execução das instruções aproveitasse ao máximo o conjunto de portas disponíveis no processador. Essa abordagem nos permite paralelizar cálculos quando não há dependência entre as instruções.

A estratégia básica, da implementação neste trabalho, foi analisar as principais funções que processam as mensagens para gerar o valor de resumo. Analisou-se, para cada instrução das funções, as portas disponíveis para executá-las, a latência, a vazão e as dependências no código. As funções mais críticas são:

- $T_1(h, e, f, g) = h + Sig1(e) + Ch(e, f, g) + K^{256} + W$
- $T_2(a, b, c) = Sig0(a) + Maj(a, b, c)$

A função T_1 do SHA-256 possui as seguintes funções lógicas:

- $Sig1(x) = Rot^6(x) \oplus Rot^{11}(x) \oplus Rot^{25}(x)$
- $Rot^n(x) = (\texttt{SL}) \vee (\texttt{SR}) = (x \ll (32 - n)) \vee (x \gg n)$
- $Ch(e, f, g) = (((f \oplus g) \wedge e) \oplus g)$

Como exemplo, a função $Sig1(x)$ faz 3 chamadas para a função de rotação $Rot^n(x)$. A função $Rot^n(x)$ executa 3 deslocamentos para a esquerda (`SL`), 3 deslocamentos para a direita(`SR`) e 3 operações (`OR`). Se cada chamada para a função de rotação $Rot^n(x)$ for executada separadamente, então as instruções dessa função também são executadas separadamente, desperdiçando as portas disponíveis no processador.

O Pseudocódigo a seguir ilustra a implementação da função $Sig1$ neste trabalho.

Como os processadores utilizados possuem 3 portas disponíveis para executar a instrução `OR`, então a função $Rot^n(x)$ foi desenrolada de forma a executar primeiramente todos os (`SL`), seguido dos (`SR`).

Ao desenrolar a função $Rotn(x)$, os 3 valores de `SL` e os 3 valores de `SR` estarão disponíveis para executar a operação `OR` para os 3 valores em paralelo. Particularmente, como o Skylake possui uma porta a mais para executar os deslocamentos, é possível executar dois deslocamentos `SL` e dois deslocamentos `SR` ao mesmo tempo.

Esta análise das funções lógicas do Algoritmo SHA2, resultou em uma implementação que visa aproveitar as portas disponíveis pelos processadores utilizados. Conseguiu-se melhorar o desempenho ao reorganizar a ordem de execução das instruções.

Como exemplo, a Figura 5.4 ilustra a sequência de execução das instruções da função $T1$ no processador Haswell, de acordo com a implementação deste trabalho. A dependência é representada de baixo para cima de acordo com a latência, a vazão e as dependências das instruções. Os nós representam as operações das instruções, os números abaixo de

Algorithm 29 Pseudocódigo da Função $Sig1$

Entrada: valor de e
Saída: valor de $Sig1$

$//Rotn(e) = (\texttt{SL}(32 - n)\ \texttt{OR}\ (\texttt{SR} \gg n))$
$t0 = \_mm256\_slli\_epi32(e, 32 - (6));$
$t1 = \_mm256\_slli\_epi32(e, 32 - (11));$
$t2 = \_mm256\_slli\_epi32(e, 32 - (25));$
$t3 = \_mm256\_srli\_epi32(e, (6));$
$t4 = \_mm256\_srli\_epi32(e, (11));$
$t5 = \_mm256\_srli\_epi32(e, (25));$
$t0 = \_mm256\_or\_si256(t0, t3);$
$t1 = \_mm256\_or\_si256(t1, t4);$
$t2 = \_mm256\_or\_si256(t2, t5);$
$//Sig1 = Rot6\ \texttt{XOR}\ Rot11\ \texttt{XOR}\ Rot25$
$Sig1 = \_mm256\_xor\_si256(t0, t1);$
$Sig1 = \_mm256\_xor\_si256(Sig1, t2);$

cada nó representam o tempo em ciclos do relógio e a disposição do diagrama mostra a dependência de cada operação. Os nós vermelhos representam as instruções com piores vazão e os nós verdes as instruções com melhor vazão.

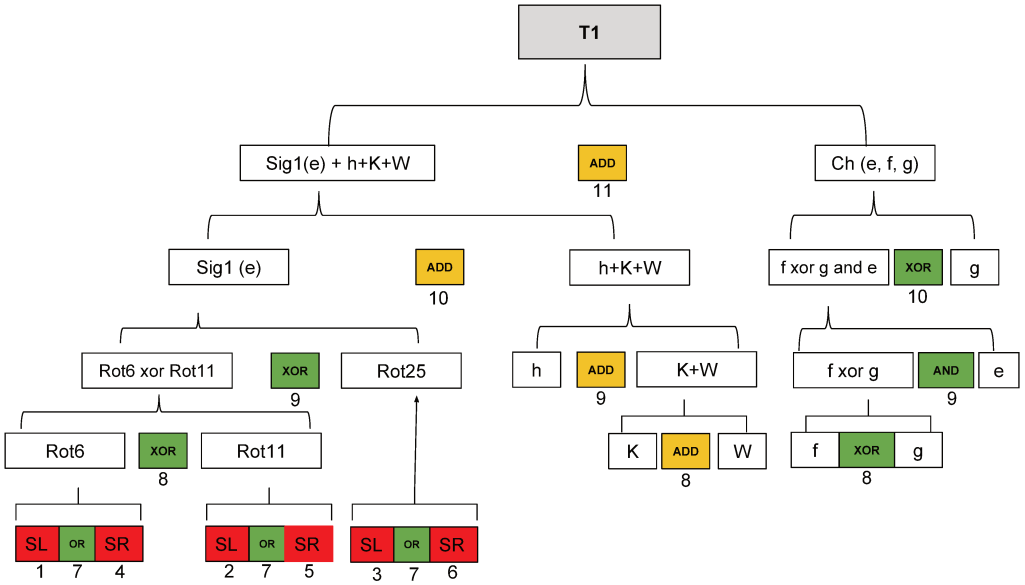


Figura 5.4: Mapa de dependência de execução da função T1 no processador Haswell

As operações de deslocamentos (SH) para a esquerda (SL) e para a direita (SR) da Figura 5.4, podem ser calculadas nos tempos 1 a 6. De acordo com a tabela 5.1 do Haswell, esta instrução tem latência de 1 ciclo e a vazão é 1. As três operações OR são representadas no tempo 7, pois a vazão dessa operação lógica é 1/3. Como a latência das operações OR é de 1 ciclo, no tempo 8 pode-se calcular as próximas instruções.

Uma latência de 1 para uma instrução de deslocamento (*shift*), significa que uma nova instrução *shift* pode iniciar a execução 1 ciclo do relógio depois daquela instrução. A vazão

1/3 para uma instrução `ADD` significa que esta instrução pode executar até 3 adições de números inteiros por ciclo do relógio, ou seja, é possível executar 3 `ADD` ao mesmo tempo.

De modo geral, a latência será de 1 se as instruções são vistas isoladamente, mas quando olha-se para uma longa cadeia de instruções da função T1, a latência total será de 11 ciclos, onde as partes mais críticas são as rotações de bits. Pode-se calcular, em média, a latência total (Lat) no Haswell da função T_1, com a seguinte fórmula:

$$Lat\_Haswell = (6 * \texttt{SH}) + (4 * \texttt{ADD})/2 + (8 * \texttt{LOGIC})/3 \approx 11\ ciclos.$$

Analisando a Latência e a vazão no processador Skylake para a função T_1, observa-se que algumas operações possuem a mesma latência, porém uma melhor vazão. A operação SH possui uma vazão de 1/2 e a operação `ADD` uma vazão de 1/3. Assim, o cálculo da latência para o processador Skylake pode ser expresso como:

$$Lat\_Skylake = (6 * \texttt{SH})/2 + (4 * \texttt{ADD})/3 + (8 * \texttt{LOGIC})/3 \approx 8\ ciclos.$$

Na Seção 6.2 é mostrado como estas otimizações melhoraram o desempenho do SHA2 no Haswell e no Skylake.

5.4 Otimizações na implementação do SHA3

A implementação do SHAKE128 foi feita conforme a implementação eficiente do SHA3, proposta em [8]. A seguir, é apresentado um resumo dessa implementação usando AVX2. Conforme descrito na Seção 2.3.2, o *Secure Hash Algorithm-3* (SHA3) é uma família de funções que foi padronizada pelo NIST em 2015[41]. Essa família consiste de quatro funções de resumo e duas funções de saída extensível (XOFs), chamadas SHAKE128 e SHAKE256.

A função de permutação usada pela família SHA3 é $KECCAK - p[1600, 24]$ e é a responsável pela eficiência do algoritmo. A função de permutação $KECCAK - p[1600, 24]$ é composta de cinco passos que são processadas 24 vezes [41]. Os passos são: o passo θ, onde é calculado um ou exclusivo (`XOR`) em cada palavra do estado; o passo ρ, onde em cada palavra do estado é rotacionada uma quantidade fixa de bits; o passo π onde as palavras do estado são permutadas; o passo χ, onde é processada uma função não-linear entre os elementos da mesma linha; e o passo ι, onde é computado um `XOR` entre o primeiro elemento do estado com um valor constante.

A função $KECCAK - p[1600, 24]$ usa um estado de 25 palavras de 64 bits. O uso das instruções $AVX/AVX2$ permite reunir quatro palavras em um mesmo registrador e processar quatro estados. Para mapear quatro estados são necessárias 25 variáveis de 256 bits. Após o mapeamento, cada uma das variáveis será composta por uma palavra de cada estado.

Para implementar o passo θ, são apenas necessários `XORs` e rotações; As instruções AVX `vpsllq` e `vpsrlq` podem ser usadas para emular instruções de rotação. Os outros quatro passos podem ser implementadas em blocos de cinco palavras. É importante processar essas palavras em conjunto para evitar um grande número de acessos de memória,

porque a arquitetura Intel tem 16 registradores de 256 bits e para esta implementação são necessários 25.

No passo ρ é necessário rotacionar uma quantidade diferente de bits em cada palavra do estado. É possível processar este passo em paralelo usando as instruções AVX2 `vpsllvq` e `vpsrlvq` para emular uma variável girar.

O passo π faz permutação das palavras do estado; Como cada palavra de cada estado foi mapeada na mesma variável, a permutação apenas muda o nome das variáveis, onde nenhuma instrução é necessária.

O passo χ é processado em paralelo usando um `XOR` e a instrução `vpandn` e o passo ι é apenas um `XOR` da primeira palavra do estado com uma constante.

5.5 Otimizações para as Funções do LMS e do XMSS

A seguir, são apresentadas as otimizações de software que foram aplicadas aos drafts do LMS [37] e do XMSS [23], onde são descritas como estas otimizações melhoraram os algoritmos de geração de chaves, assinatura e a verificação desses esquemas. Cada um desses algoritmos é baseado em funções de resumo, conforme descritas nas Seções 4.1.3 e 4.2.3. Assim, otimizar estas funções resultou em acelerar a execução desses algoritmos. No trabalho SPHINCS [1], foi citado o uso de instruções vetoriais para melhorar os tempos de execução do esquema.

As funções otimizadas foram:

- as funções de resumo chaveadas do LMS e do XMSS;
- a função F do LMS e do XMSS;
- a função PRF do LMS e do XMSS;
- a função G da Ltree do XMSS.

5.5.1 Otimização das Funções de Resumo Chaveadas

Nesta seção, são descritas as otimizações aplicadas às funções de resumo chaveadas dos esquemas LMS e XMSS. As funções de resumo chaveadas utilizadas nos algoritmos dos drafts do LMS e do XMSS, trabalham sempre com blocos de mensagens do mesmo tamanho. Então, para acelerar o tempo de cálculo das funções de resumo chaveadas, foi feita uma implementação especializada para estas funções com base no tamanho da entrada da mensagem a ser processada. Nesta implementação foram estabelecidas as quantidades de blocos em cada função de resumo, os valores atribuídos a cada bloco e o valor do *pad* da função de resumo. Os valores dos *pads* puderam ficar fixos porque os blocos dessas funções têm sempre o mesmo tamanho.

O SHA2 processa blocos de 512 bits e o SHAKE128 consegue processar até 1344 bits da mensagem por vez. Uma implementação da função F do XMSS com SHAKE128 precisa processar único bloco de mensagem. Já no SHA2, esta função precisa processar 2 blocos para gerar o valor do resumo. A Figura 5.5 mostra a otimização da função F do XMSS com pad fixo para o SHA-256.

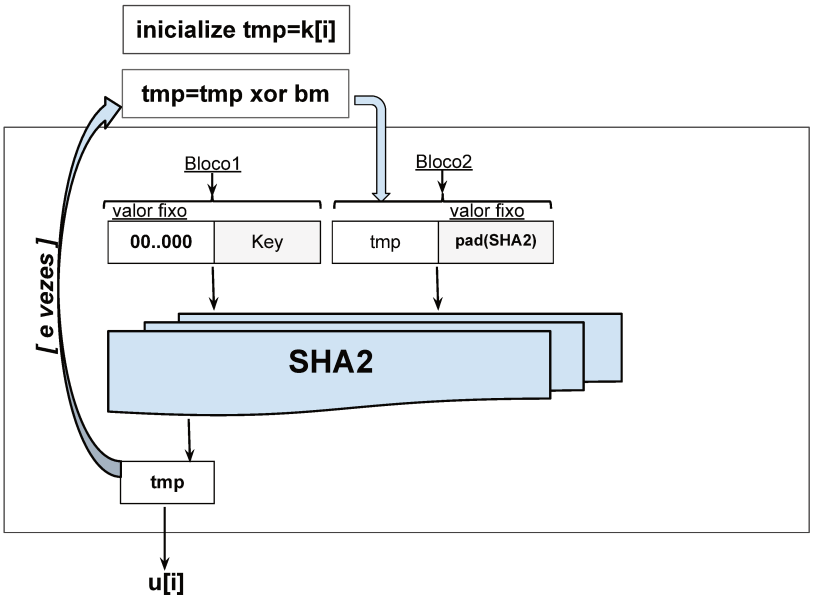


Figura 5.5: Função F no WOTS+

Para as demais funções chaveadas dos esquemas LMS e XMSS, conforme descrito na Seção 4.3, também foi criada uma implementação otimizada com valores de *pad* fixos. Como os valores de *pad* são fixos, não há necessidade de cálculo do *pad* a cada vez que a função é chamada e assim o número de operações das funções pôde ser reduzido, obtendo-se uma melhor performance.

Nesta implementação otimizada das funções, foi criada uma interface para permitir que estas funções recebam e processem blocos de mensagens de 32 bits no SHA-256 e 64 bits no SHA-512. A criação de uma implementação dessas funções com entrada, processamento e saída com valores de 32/64-bits, reduziu significativamente o tempo de processamento em relação a funções genéricas que recebem caracteres de 8-bits. Isto ocorre porque a conversão de 32/64 bits para 8 bits consome muito tempo do processador.

5.5.2 Otimização da Função F

O algoritmo da função de encadeamento envolve muitas aplicações da função F em cada elemento das chave privadas e da assinatura. Na assinatura, a técnica de paralelização também é usada para atualizar o caminho de autenticação. Assim, reduzir o tempo de execução da função F refletiu em uma melhora significativa no desempenho do LMS e do XMSS, conforme demonstrado no Capítulo 6.2.

A técnica *multi-buffer* é aplicada na função F (F_SIMD) para processar os valores da chave secreta $sk = \{k[0], \ldots, k[t-1]\}$ em paralelo. Utilizar a técnica *multi-buffer*, conforme descrito na Seção 5.3.1, para a função F com SHA-256, permitiu calcular oito instâncias da chave privada em paralelo, e com SHA-512 e SHAKE128, quatro instâncias.

A Figura 5.6 mostra a geração de oito valores da chave pública $pk = \{u[1], \ldots, u[8]\}$ do XMSS com SHA-256. Foram carregadas oito valores de $sk = \{k[1], \ldots, k[8]\}$ no registrador de 256-bits, executadas e iterações da função F e então obteve-se oito valores

de pk. Os valores de pk foram calculadas em paralelo porque sua geração é independente. Adicionalmente, foram armazenadas 8 instâncias do valor do pad em um registrador de 256, porque estes valores são utilizados várias vezes.

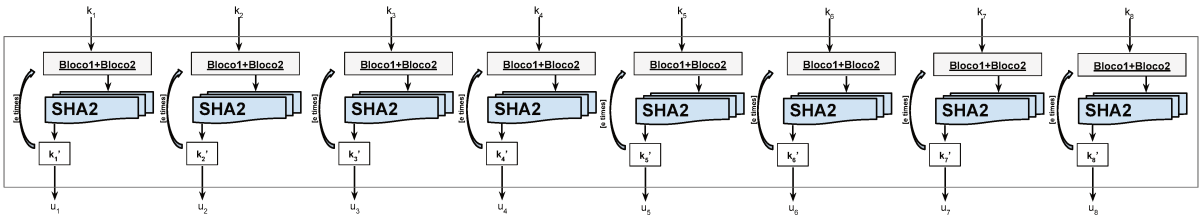


Figura 5.6: Implementação da função de encadeamento F no XMSS com SHA-256

O uso de instruções SIMD ajudou a reduzir o tempo de execução da função F, que é umas das partes computacionalmente mais caras para a geração de chaves, assinatura e verificação.

5.5.3 Otimização da Função PRF

A função pseudoaleatória (PRF) é usada para gerar os valores pseudoaleatórios no esquema XMSS. Este gerador foi implementado tanto na geração de chaves quanto na assinatura do LMS e do XMSS. A função $PRF : H(Seed||M)$ recebe como entrada um valor aleatório $Seed$ e um valor M. A função PRF_SIMD com SHA-256 que recebe 8 valores de $Seed$ e 8 valores de M, processa esses valores em paralelo e devolve 8 valores pseudoaleatórios. Essa função com SHAKE, processa 4 valores de forma paralela.

Assim, foi possível gerar os elementos da chave secreta sk de forma paralela, utilizando a função PRF otimizada. As chaves secretas são calculadas como $k[i] = PRF(Seed, i_{32})$, para $0 \leq i \leq tm$. A cadeia $Seed$ é um valor secreto, gerado aleatoriamente e é usada como semente para gerar todos os valores de sk. O valor i é concatenado com o valor de $Seed$ para gerar os valores de sk. Como não há dependência entre a geração dos elementos de sk, foi possível gerar 8 valores de sk ao mesmo tempo com SHA-256, reduzindo, assim, os tempos de execução dessas funções.

A função F na Geração de Chaves, Assinatura e Verificação

Para a geração de chaves OTS, a aplicação da otimização da função F_SIMD (*multi-buffer*) foi direta, porque na geração das chaves pk, a função F é aplicada o mesmo número de vezes em todos os elementos da chave secreta sk. Porém, para a geração da assinatura e verificação OTS, as aplicações da função F em cada elemento da assinatura dependem do resumo da mensagem M, conforme descrito nos Algoritmos 11 e 18. Assim, os elementos da chave $sig = (sig[0], \ldots, sig[tm-1])$ podem receber valores diferentes na aplicação de F.

Então, foi efetuada uma pequena alteração nos Algoritmos de assinatura e verificação one-time de forma a aplicar a função F em paralelo nos elementos da assinatura. Foi

acrescentado um algoritmo de ordenação antes da aplicação da função F. O vetor msg, que contém os valores para aplicação de F, é ordenado de acordo com o número de aplicações que a função F precisa executar. Esse valor é guardado em um vetor temporário ($tmpMsg$), que mantém a seguinte estrutura:

struct $tmpMsg$:

- ind (posição atual em msg)
- e (número de aplicações da função F)

Após a ordenação, os elementos de $tmpMsg$, que têm o mesmo valor de e para executar as aplicações de F em paralelo, foram selecionados. O Algoritmo 30, mostra a alteração nos Algoritmos 18 e 19 do XMSS para a aplicação da técnica *multi-buffer*. A mesma técnica foi aplicada aos algoritmos de assinatura e verificação LM-OTS. A constante SIMD define o número máximo de instâncias que serão executadas pela função de resumo ao mesmo tempo.

Algorithm 30 Alteração na assinatura e verificação one-time

Entrada: a mensagem M, a chave secreat sk.
Saída: assinatura sig.

```
// calcula msg como nos Algoritmos 18 e 19
msg = msg||baseW(csum_len2, w, len2);
// tmpMsg recebe os valores de msg ordenados
tmpMsg=Ordena(msg);
// executa as aplicações da função F
for ( i = 0; i < tm; i+ =SIMD) do
    max = 0; ini = i;
    //seleciona o próximo valor e
    eProx = tmpMsg[i].e;
    for ( j = 0; j < SIMD; j + +) do
        if (tmpMsg[j].e = eProx) then
            sk256 = k[j];
            Adrs[4] = j;//ChainAddress=j
            max + +; ini = ++;
        end if
    end for
    //calcula o máximo de instâncias de sig em paralelo
    tmpSig = chainSIMD(sk256, 0, proxMsg, Seed, Adrs);
    //copia para sig os valores de tmpSig na ordem original
    for ( j = 0; j < max; j + +) do
        sig[msg[ini].ind] = tmpSig[j];
        ini + +;
    end for
end for
return sig;
```

5.5.4 Otimização da Função G na Ltree

Para gerar as folhas da árvore XMSS, uma árvore Ltree do XMSS [2] é usada. Nesta seção é apresentada a otimização na geração da Ltree que melhora a performance da geração de cada folha da árvore XMSS, reduzindo assim o tempo do algoritmo de geração de chaves do XMSS.

A Ltree usa uma função de resumo ($G : H(1_n||Key||M)$) para gerar os nós internos da árvore. Então, o algoritmo da Ltree foi modificado para executar em paralelo as avaliações da função G.

A Figura 5.7 mostra a otimização na geração da Ltree com SHA-256 para w=16 e tm=67. A função gera oito nós internos ao mesmo tempo, a partir de 16 nós filhos, que são concatenados 2 a 2. Se a quantidade de nós remanescentes não é múltiplo de 16, os próximos nós internos são gerados um por um, como no algoritmo tradicional. Se tm não é uma potência de 2, então não há folhas suficientes para construir uma árvore binária. Então, um nó que não tem um irmão direito é elevado ao nível superior da Ltree até ele tornar-se o irmão direito de outro nó.

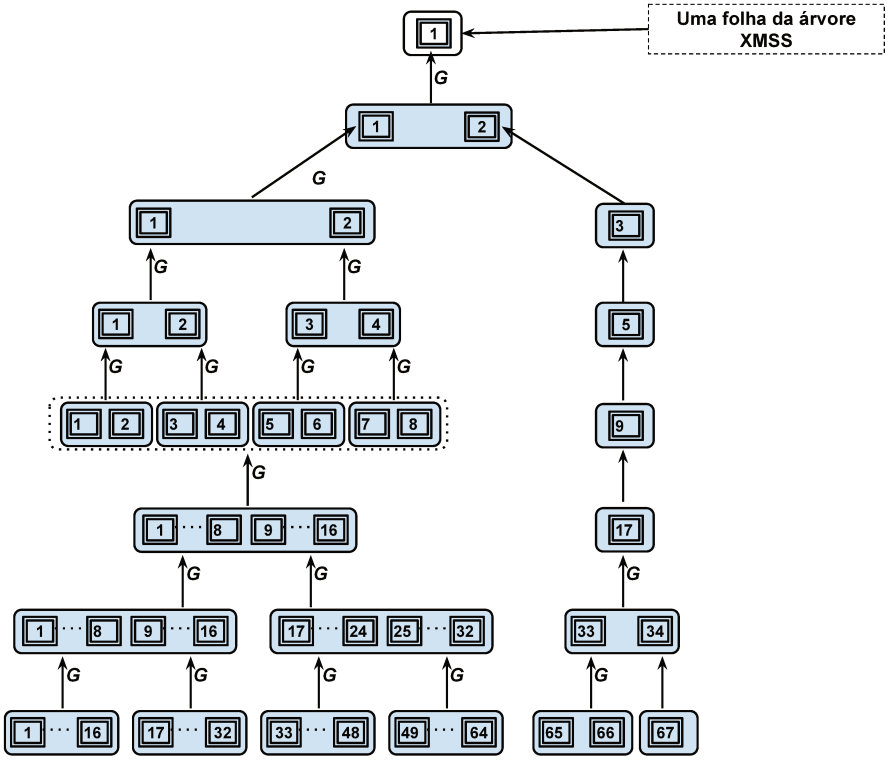


Figura 5.7: Construção da Ltree

Capítulo 6

Resultados

Neste capítulo, são apresentados os resultados do trabalho de implementação em software dos sistemas criptográficos LMS e XMSS. Os parâmetros de cada sistema são descritos na Seção 6.1 e os resultados na Seção 6.2. Ao final, a Seção 6.3 apresenta uma análise dos resultados das implementações.

6.1 Parâmetros

A seguir, são apresentados os parâmetros utilizados nas implementações do LMS e do XMSS. Cada esquema definiu um conjunto de parâmetros. Para ilustrar o desempenho desses esquemas em processadores com suporte a instruções vetoriais, selecionamos um subconjunto desses parâmetros que oferecem um nível de segurança de 128 e/ou 256 bits. Os parâmetros são: $w \in \mathbb{N}$, o parâmetro de Winternitz; $h \in \mathbb{N}$, a altura total da árvore; d, o número de camadas e n, a saída da função de resumo.

Conforme descrito nas seções 4.1.7 e 4.2.7, a saída da função de resumo escolhida influencia a segurança do sistema. Considerando computadores clássicos, o parâmetro $n = 32$ provê nível de segurança de 256 bit e $n = 64$ provê nível de segurança de 512 bit. Para obter um nível de segurança de 128 bits para computadores quânticos no XMSS, as funções SHA-256 e SHAKE128 foram selecionadas. Para obter um nível de segurança de 256 bits para computadores quânticos no XMSS, foram utilizadas as funções SHA-512 e SHAKE256.

O valor de w influencia no tempo de execução e no tamanho da assinatura. Um aumento no valor de w implica incrementar o tempo de execução das operações de assinatura, por outro lado diminui o tamanho das assinaturas. Os valores de h e d afetam o tamanho da assinatura. A saída n da função de resumo escolhida também influencia no tamanho das chaves pública, privada e de assinatura. Para o LMS selecionou-se $w = 2$ e $w = 4$ como parâmetro Winternitz e para o XMSS selecionou-se $w = 16$. A altura máxima do XMSS-MT foi $h = 60$ e o número máximo de camadas foi $d = 12$ de acordo com [23].

6.2 Resultados

Nesta seção são apresentados os resultados experimentais da implementação da funções de resumo (SHA2 e SHA3) e os resultados da implementação do LMS e do XMSS. Os resultados foram obtidos nos processadores: Sandy Bridge Core i7-2600 3.4 GHz, Haswell Core i7-4770K 3.4 GHz e Skylake Core i7-6700K de 4.2 GHz. As tecnologias Intel Turbo Boost e Intel Hyper-Threading foram desabilitadas. A implementação foi escrita em linguagem C e compilada usando o GNU C Compiler v6.2.0.

Os resultados da implementação denominados "single-buffer" utilizam os registradores nativos de 64 bits, enquanto que a implementação denominada "multi-buffer" utiliza os registradores vetoriais de 128/256 bits.

O Algoritmo BDS [4] descrito na Seção 3.4 foi utilizado para calcular os caminhos de autenticação de forma mais eficiente na assinatura. Os tempos para assinatura e verificação para $h > 20$ são calculados usando a média aritmética do primeiro milhão de assinaturas.

6.2.1 Resultados da Implementação das Funções de Resumo

A seguir, são mostrados os tempos obtidos nas implementações das funções de resumo. A Figura 6.1 mostra o desempenho das implementações da função de resumo SHA-256 utilizando registradores de 128 bits (AVX). Estas implementações processam 4 mensagens do mesmo tamanho e geram 4 resumos ao mesmo tempo. Comparamos os resultados da implementação de Gueron et al. [53] que foi executada em um processador Core i7-3770 com o desempenho da implementação neste trabalho executada em um processador Core i7-2600. Observamos que a implementação deste trabalho apresenta tempos semelhantes a implementação em [53].

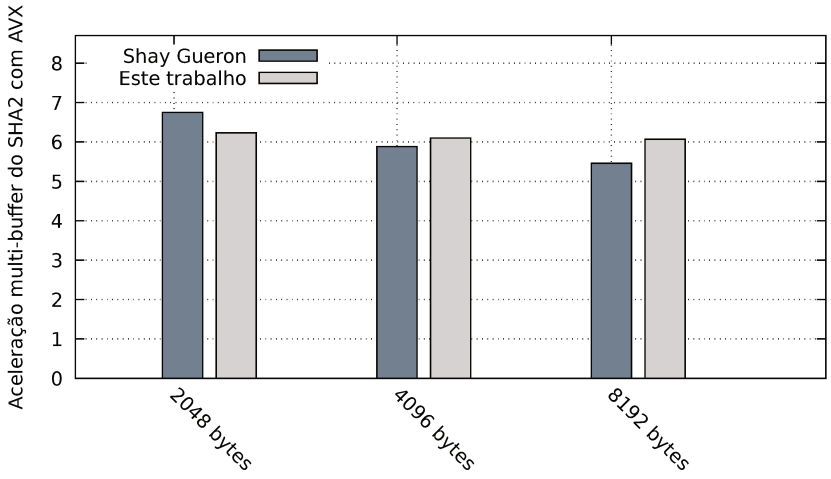


Figura 6.1: Desempenho da implementação do SHA-256 4-way

A Tabela 6.1 mostra o desempenho single-buffer (64 bits) versus multi-buffer (256 bits) em ciclos por bytes da implementação das funções de resumo (H) SHA-256 e SHAKE128

no processador Skylake. O tamanho da entrada dessas funções foi selecionado de acordo com o tamanho das funções F e G do LMS e do XMSS.

A função F processa mensagens de tamanho 104 bytes no LMS e 96 bytes no XMSS e a função G processa mensagens de tamanho de 133 bytes no LMS e 128 bytes no XMSS. Conforme descrito na seção 4.3, a função F processa dois blocos de dados (512 bits) no SHA-256 e a função G processa três blocos de dados (512 bits). No SHAKE128, as funções F e G processam somente um bloco de dados (1344 bits). Portanto, a implementação das funções F e G single-buffer com SHAKE128 tem melhores resultados do que com SHA-256 single-buffer.

Tabela 6.1: Desempenho das funções F e G com SHA-256 e SHAKE128

Tempo de Execução (ciclos/bytes)				
Função H	**Função do draft**	**Single Buffer**	**Multi Buffer**	
SHA-256	F(104 bytes)	14,64	8-way	2,98
SHAKE128	F(104 bytes)	12,37	4-way	4,73
SHA-256	G(133 bytes)	16,52	8-way	3,50
SHAKE128	G(133 bytes)	9,67	4-way	3,48

A execução das funções F e G com SHA-256 multi-buffer foi mais rápida do que com SHAKE128 multi-buffer. Os motivos são descritos a seguir: O razão da função F com SHAKE128 single-buffer é de $1,2$x em relação à implementação com SHA-256 single-buffer. A implementação (denominada 8-way) do SHA-256 processa 8 valores de resumo independentes simultaneamente enquanto a implementação (denominada 4-way) do SHAKE128 processa apenas 4 valores de resumo independentes em paralelo. A razão multi-buffer versus single-buffer da função F com SHA-256 é aproximadamente $4,6$x e com SHAKE128 é aproximadamente $2,4$x.

A Figura 6.2 ilustra o desempenho das funções F e G nos processadores Haswell e Skylake. Observa-se que a Implementação da Função F com SHA-256 apresenta um *speedup* de $4,6$x no Haswell e $4,9$x no Skylake. A Função F com SHAKE128 apresenta um *speedup* de $2,43$x no Haswell e $2,61$x no Skylake. O desempenho das funções de resumo no Skylake são melhores devido às características das microarquiteturas dos computadores apresentadas no Capítulo 5, na Tabela 5.1.

Nas seções seguintes mostra-se que o desempenho obtido na implementação multi-buffer das funções de resumo reflete no desempenho dos algoritmos de geração de chaves, assinatura e verificação do LMS e do XMSS, porque a implementação otimizada das funções destes esquemas utiliza as otimizações aplicadas as funções de resumo.

6.2.2 Resultados da Implementação do XMSS

Nesta seção, são apresentados os resultados da implementação single-buffer e multi-buffer do XMSS com SHA2 e SHAKE.

A Tabela 6.2, apresenta uma comparação entre a implementação do XMSS single-buffer deste trabalho com a implementação single-buffer apresentada no site do autor [24]. Os resultados foram obtidos no processador Haswell. Observa-se uma razão de $\approx 1,4$x

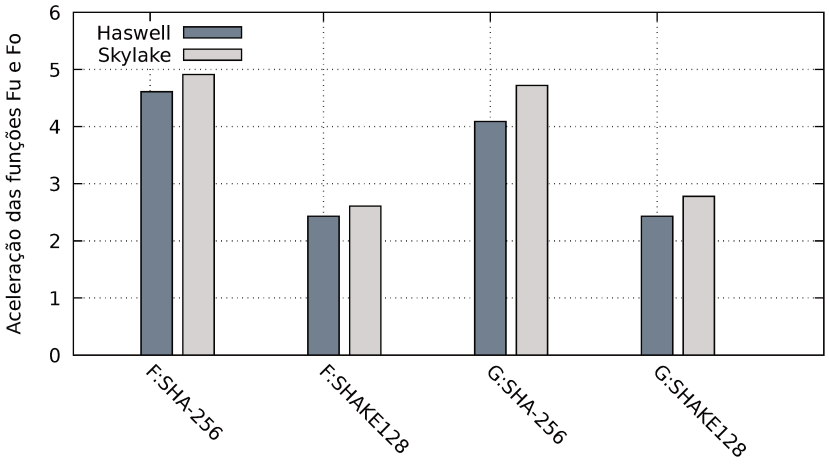


Figura 6.2: Razão Multi Buffer versus Single Buffer das funções F e G

para geração de chaves, assinatura e verificação da implementação otimizada deste trabalho em relação a implementação em [24]. Este desempenho foi devido à implementação otimizada das funções de resumo chaveadas dos esquemas, conforme descrito na Seção 5.5.1.

Tabela 6.2: Comparação da geração de chaves, assinatura e verificação entre implementação do XMSS em [24] e a implementação deste trabalho no Haswell.

Tempo de execução (ms) do XMSS com SHA-256									
	A: [24]			**B:[este trabalho]**			**Razão A/B**		
h	**KG**	**SIG**	**VER**	**KG**	**SIG**	**VER**	**KG**	**SIG**	**VER**
10	2241	9,78	1,20	1613	7,04	0,87	1,39	1,39	1,38
16	143329	16,31	1,22	103268	11,76	0,88	1,39	1,39	1,39
20	2286505	20,63	1,22	1652284	14,91	0,89	1,38	1,38	1,37

A Tabela 6.3 apresenta o desempenho do XMSS para o nível de segurança de 128 bits. São apresentados os tempos da implementação multi-buffer em relação a implementação single-buffer deste trabalho. Os tempos são medidos com as funções de resumo SHA-256 e SHAKE128.

Observa-se nesta tabela uma melhoria de desempenho decorrente da implementação do XMSS com SHA-256 multi-buffer versus single-buffer de 4, 4x, 4, 2x e 2, 4x para geração de chaves, assinatura e verificação respectivamente. O razão do XMSS com SHAKE128 multi-buffer versus single-buffer é de 2, 6x para a geração de chaves, 2, 5x para assinatura e 2, 0x para a verificação.

O desempenho multi-buffer é melhor no algoritmo de geração de chaves do que nos algoritmos de geração de assinatura e verificação, porque a função $FSIMD$ executa a mesma quantidade de vezes em todos elementos da chave secreta. Diferentemente, no processo de assinatura e verificação WOTS, foi necessária uma ordenação dos elementos da assinatura antes da aplicação da função $FSIMD$, pois a quantidade de aplicações da função depende dos bits da mensagem.

Observa-se também na Tabela 6.3 que o desempenho do XMSS single-buffer com

Tabela 6.3: Resultados no Skylake da implementação do XMSS com o parâmetro $w = 16$ e diferentes valores de h.

Tempo de execução (ms) do XMSS											
		A: Single Buffer			**B: Multi Buffer**				**Razão A/B**		
Função H	**h**	**KG**	**SIG**	**VER**	**SIMD**	**KG**	**SIG**	**VER**	**KG**	**SIG**	**VER**
SHA-256	10	1337	5,84	0,72	8-way	305	1,43	0,29	4,38	4,08	2,48
SHA-256	20	1369770	12,36	0,73	8-way	312702	2,92	0,30	4,38	4,23	2,43
SHAKE128	10	1029	4,46	0,59	4-way	400	1,78	0,29	2,57	2,51	2,03
SHAKE128	20	1056084	9,49	0,60	4-way	410818	3,74	0,30	2,57	2,54	2,00

SHAKE128 é melhor do que com SHA-256 single-buffer. A versão multi-buffer do SHA-256 apresenta melhores tempos que a versão multi-buffer do SHAKE128. Estes tempos estão de acordo com os tempos das funções de resumo da Tabela 6.1. Pois as funções F e G single-buffer apresentaram melhores resultados com SHAKE128 porque conseguem processar blocos maiores (1344 bits), enquanto a função SHA-256 processa blocos de 512 bits. A versão multi-buffer com SHA-256 tem melhores resultados por conseguir processar 8 mensagens, enquanto a versão com SHAKE128 processa somente 4 mensagens em paralelo.

Na próxima tabela 6.4, são apresentados os tempos do XMSS para o nível de segurança de 256 bits. São apresentados os tempos dos resultados do XMSS com SHA-512/SHAKE256 single-buffer versus multi-buffer.

Observa-se que o desempenho decorrente da otimização multi-buffer para geração de chaves, assinatura e verificação é, respectivamente: com SHA-512 de $2,4$ x, $2,4$x e $2,0$x; e com SHAKE256 de $3,3$ x, $3,3$x e $2,8$x.

Tabela 6.4: Resultados no Skylake do XMSS com $w = 16$ para diferentes valores de h.

Tempo de execução (ms) do XMSS										
		A: Single Buffer			**B: Multi Buffer(4-way)**			**Razão A/B**		
Função H	**h**	**KG**	**SIG**	**VER**	**KG**	**SIG**	**VER**	**KG**	**SIG**	**VER**
SHA-512	10	3458	15,12	1,82	1419	6,32	0,88	2,44	2,39	2,07
SHA-512	20	3537106	32,23	1,85	1452600	13,35	0,90	2,44	2,41	2,06
SHAKE256	10	5222	23,12	2,74	1549	6,96	0,96	3,37	3,32	2,85
SHAKE256	20	5339593	48,67	2,77	1587226	14,57	0,99	3,36	3,34	2,80

O tempos de execução do esquema XMSS com as funções SHA-512/SHAKE256 são piores do que com SHA-256/SHAKE128, mas provêm um maior nível de segurança (256 bits). Para este nível de segurança, o desempenho do XMSS com a função de resumo SHA-512 foi melhor do que com a função SHAKE256, pois neste caso, as duas funções de resumo recebem e processam blocos de 1024 bits e as etapas de processamento da função SHA3 consomem mais tempo do que as rodadas da função SHA2.

Conforme aumenta o nível de segurança para 256 bits, os tamanho das chaves do esquema também aumentam, porque o tamanho das chaves são diretamente relacionados ao tamanho da saída da função de resumo (n). A chave secreta `SK` tem $4n + 4$ bytes, a chave pública PK tem $2n$ bytes, e a assinatura `Sig` tem $(t + h + 1)n + 4$ bytes. A Tabela 6.5 apresenta o tamanho das chaves do XMSS para os níveis de segurança de 128 bits ($n = 32$) e 256 bits ($n = 64$).

Tabela 6.5: Tamanhos das chaves (bytes) do XMSS para $h = 20$ e $t = 67$.

	Nível de segurança	
Chave	**128 bits**	**256 bits**
SK	132	260
PK	64	128
Sig	2820	5636

6.2.3 Resultados da Implementação do LMS

Nesta seção são apresentados os resultados da implementação do LMS multi-buffer com SHA-256 e SHAKE256 para o nível de segurança de 128 bits. A Tabela 6.6 apresenta o desempenho da implementação do LMS multi-buffer versus single-buffer.

Tabela 6.6: Resultados no Skylake do LMS com o parâmetro $w = 4$ para diferentes valores de h.

Tempo de execução (ms) do LMS											
		A:Single Buffer			**B:Multi Buffer**				**Razão A/B**		
Função H	**h**	**KG**	**SIG**	**VER**	**SIMD**	**KG**	**SIG**	**VER**	**KG**	**SIG**	**VER**
SHA-256	10	427	1,88	0,21	8-way	100	0,47	0,09	4,27	4,00	2,33
SHA-256	20	436627	3,96	0,23	8-way	103053	0,96	0,11	4,24	4,13	2,09
SHAKE256	10	345	1,52	0,17	4-way	127	0,58	0,09	2,72	2,62	1,89
SHAKE256	20	353786	3,20	0,18	4-way	130802	1,20	0,10	2,70	2,67	1,80

Observa-se que a razão entre o desempenho da implementação do LMS multi-buffer versus single-buffer com SHA-256 e $w = 4$ para geração de chaves, assinatura e verificação é de $4,2$x, $4,1$x e $2,0$x e com SHAKE256 é de $2,7$x, $2,6$x e $1,8$x. Assim como no XMSS, o LMS single-buffer com SHAKE256 é mais rápido do que com SHA-256, e o LMS multi-buffer com SHA-256 é mais rápido do que com SHAKE256.

A tabela 6.7 apresenta os tamanhos das chaves e os tempos de execução do LMS com SHA-256 multi-buffer para os valores de $w = 2$ e $w = 4$.

Tabela 6.7: Resultados no Haswell do LMS com a função de resumo SHA-256 para diferentes valores de h e w.

LMS com SHA-256 Multi Buffer (4-way)								
			Tempo (ms)			**Tamanho (bytes)**		
h	**w**	**p**	**KG**	**SIG**	**VER**	**SK**	**PK**	**SIG**
10	2	133	60	0,27	0,05	64	96	4612
20	2	133	61987	0,56	0,06	64	96	4932
10	4	67	100	0,47	0,09	64	96	2500
20	4	67	103053	0,96	0,11	64	96	2820

Observa-se que, quanto maior o valor de w, menor o tamanho da assinatura, porém o tempo de execução na geração de chaves, assinatura e verificação aumenta porque a função F é executada $2^w - 1$ vezes em cada elemento das chaves. O tamanho da assinatura aumenta conforme aumenta a altura da árvore porque é preciso guardar o caminho de autenticação na assinatura. A chave privada mantém o mesmo valor, pois foi utilizado o

gerador GPA sugerido em [24], onde é necessário guardar somente duas sementes (uma para gerar os valores da chave privada e uma para gerar os valores do identificador I).

6.2.4 Resultados da Implementação dos Esquemas Hierárquicos

Nesta seção, são apresentados os resultados do desempenho da implementação multi-buffer dos esquemas HSS e do XMSS-MT no processador Skylake. Os parâmetros foram definidos como $w = 4$ para LMS e $w = 16$ para XMSS, então o tamanho da assinatura OTS para ambos esquemas é $t = 67$.

A Tabela 6.8 mostra os tempos de execução da implementação multi-buffer do esquema HSS com SHA-256 e SHAKE256 para o nível de segurança de 128 bits.

Tabela 6.8: Resultados do HSS multi-buffer com o parâmetro $w = 16$ para diferentes valores de h e d.

Tempo de execução (ms) HSS Multi Buffer						
Função H	**h**	**d**	**SIMD**	**KG**	**SIG**	**VER**
SHA-256	40	2	8-way	205331	1,07	0,22
SHA-256	40	4	8-way	402	0,57	0,39
SHA-256	40	8	8-way	27	0,32	0,72
SHA-256	60	6	8-way	602	0,57	0,60
SHA-256	60	12	8-way	40	0,32	1,08
SHAKE256	40	2	4-way	281462	1,44	0,23
SHAKE256	40	4	4-way	552	0,76	0,42
SHAKE256	40	8	4-way	36	0,41	0,79
SHAKE256	60	6	4-way	827	0,76	0,62
SHAKE256	40	12	4-way	55	0,41	1,14

A Tabela 6.9 apresenta os resultados de nossa implementação do XMSS-MT para o nível de segurança de 128 bits.

Tabela 6.9: Resultados do XMSS-MT multi-buffer com o parâmetro $w = 16$ para diferentes valores de h e d.

Tempo de execução (ms)XMSS-MT Multi Buffer						
Função H	**h**	**d**	**SIMD**	**KG**	**SIG**	**VER**
SHA-256	40	2	8-way	630295	3,26	0,63
SHA-256	40	4	8-way	1236	1,74	1,15
SHA-256	40	8	8-way	83	0,96	2,18
SHA-256	60	6	8-way	1883	1,75	1,76
SHA-256	60	12	8-way	124	0,96	3,32
SHAKE128	40	2	4-way	850683	4,35	0,61
SHAKE128	40	4	4-way	1667	2,28	1,14
SHAKE128	40	8	4-way	111	1,23	2,23
SHAKE128	60	6	4-way	2498	2,28	1,67
SHAKE128	60	12	4-way	167	1,23	3,33

Note que, para os sistemas hierárquicos, aumentando o número de camadas para o mesmo tamanho de altura, os tempos de execução diminuem para a geração de chaves

e assinatura, entretanto o tempo de verificação aumenta um pouco. Isto decorre das seguintes características: na geração de chaves, árvores menores precisam ser construídas, resultando em menores tempos de execução. Na assinatura, menos nós no caminhos de autenticação precisam ser recuperados, devido ao menor tamanho das subárvores e das características do algoritmo que só atualiza as subárvores necessárias. Porém, na verificação, o tempo aumenta porque todas as subárvores precisam ser reconstruídas. Para as subárvores que têm a mesma altura, o tempo de assinatura permanece constante.

Adicionalmente, o tamanho da chave secreta e da assinatura são maiores conforme aumenta o número de camadas, porque é preciso armazenar as informações para cada camada. Uma características importante é que as assinaturas hierárquicas permitem aumentar a quantidade de assinaturas sem aumentar muito os tempos de execução, principalmente mantendo estável os tempos para assinar mensagens.

6.3 Análise dos Resultados

Nesta seção são analisados os resultados com AVX2 e são comparados os esquemas LMS e XMSS.

A Figura 6.3 mostra o desempenho dos algoritmos XMSS/LMS multi-buffer versus single-buffer com AVX2. Observa-se que as implementações multi-buffer com SHA-256 têm melhor desempenho em relação a implementação single-buffer, porque permitiu executar 8 valores de resumo em paralelo enquanto o SHAKE multi-buffer só permite executar 4 valores de resumo em paralelo.

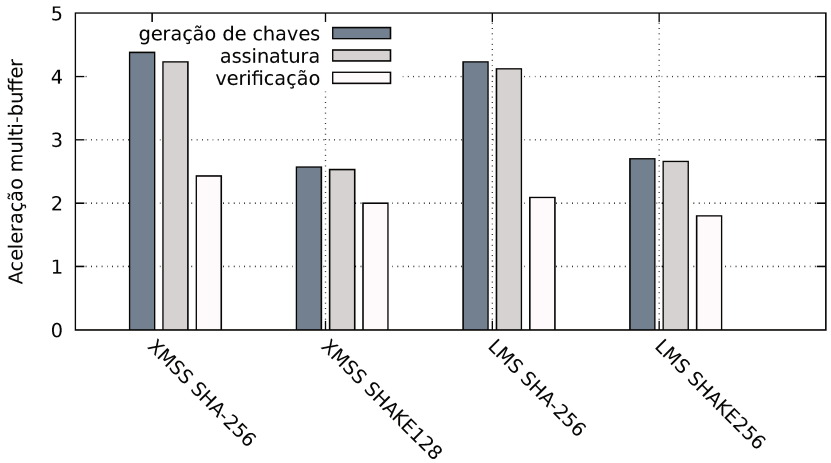


Figura 6.3: Razão Multi-buffer x Single-buffer das implementações do XMSS e do LMS

A Tabela 6.10 apresenta uma análise dos tamanhos dos códigos, implementados em linguagem C, das principais funções dos esquemas LMS e XMSS. Foi executado o comando *nm* do GNU no compilador do Linux para retornar o tamanho dos objetos de um arquivo.

Observa-se que o LMS tem tamanho de código aproximadamente $1,23$x maior do que o código do XMSS para a implementação single-buffer. Isso decorre do fato que o LMS

utiliza duas funções de resumo $Fpub$ e $Ffolh$ para a geração das folhas da árvore e o XMSS utiliza somente a Ltree.

Tabela 6.10: Comparação do tamanho dos códigos do LMS e do XMSS.

	Tamanho (bytes)	
Função	**LMS**	**XMSS**
F	9413	9664
G	14168	13996
$Fpub/Ltree$	15577	15819
$Ffolh$	9552	0

A Tabela 4.2 mostra os tempos de execução dos esquemas LMS e XMSS para uma árvore com altura $h = 20$ e nível de segurança de 128 bits ($n = 32$ bytes).

Tabela 6.11: Comparação dos tempos do LMS e do XMSS para 2^{20} assinaturas.

	Tempos (ms)		
	A: XMSS w=16	**B: LMS w=4**	**Razão A/B**
KG	410818	103053	3,99
SIG	3,74	0,96	3,89
VER	0,30	0,11	2,72

Para o mesmo nível de segurança (128 bits), utilizando a função SHA-256, observa-se na Tabela 6.11 que o LMS é $\approx 3,9$x mais rápido do que o XMSS para geração de chaves e assinatura. Isto é devido ao LMS ter menos chamadas à função de resumo, não utilizar uma árvore Ltree para geração das folhas e realizar menos operações na geração dos nós internos da árvore binária.

De acordo com os resultados apresentados, observa-se que o uso do AVX2 contribui de forma significativa na implementação das propostas de padrão para o esquema de Merkle e suas variantes. Para o nível de segurança de 128 bits, se o computador não possuir instruções AVX2, então a implementação dos esquemas LMS e XMSS com a função de resumo SHAKE single-buffer seria uma boa opção por apresentar melhores tempos de execução. Porém, se as instruções $AVX2$ estão disponíveis no computador, a implementação do LMS e XMSS multi-buffer com SHA-256 seria a melhor opção.

Capítulo 7

Conclusões

Com a evolução no desenvolvimento da construção de computadores quânticos, novos esquemas de assinatura são necessários para substituir os esquemas atuais como RSA, que não são imunes a esses computadores. Os resultados das pesquisas atuais indicam que os esquemas de criptografia baseados em funções de resumo são esquemas candidatos a substituir os esquemas atuais. Muitas melhorias já foram propostas ao esquema MSS tornando-o viável para muitas aplicações atuais.

Novas variantes surgiram para melhorar o problema de recursos de armazenamento, como o uso de geradores pseudoaleatórios, reduzindo assim o tamanho das chaves. O uso de multi-árvores, permitiu aumentar a quantidade de assinaturas e reduzir o tempo de geração de chaves e de assinatura.

Organizações governamentais de padronização (como o NIST), estão incentivando a transição para a criptografia pós-quântica e tem promovido concursos para algoritmos de criptografia de chave pública resistentes aos computadores quânticos. Atualmente, os esquemas de assinatura baseados em funções de resumo tem duas propostas de padrão: o esquema de Leighton e Micali (LMS), uma versão melhorada do esquema de assinatura única Lamport-Diffie-Winternitz-Merkle e o esquema de assinatura de Merkle estendido (XMSS). Então, o projeto e a implementação eficiente de esquemas de assinatura digital seguros e práticos são cruciais para aplicações que exigem integridade de dados e assinatura digital.

Neste cenário, este trabalho apresenta os resultados de uma implementação em software de duas propostas de padrão do esquema de Merkle (LMS e XMSS) usando AVX2 e uma análise comparativa dessas propostas. Os resultados desta pesquisa mostram que a implementação obteve melhoras significativas nos tempos de execução dos algoritmos de geração de chaves, assinatura e verificação dessas propostas. Neste processo, foi efetuado uma análise aprofundada e implementação dos esquemas de assinatura baseados em funções de resumo, apresentando os principais esquemas de assinatura one-time e N-time, como o esquema de Lamport-Diffie, o esquema de Winternitz, o esquema de Merkle e suas variantes (MSS,CMSS,GMSS,XMSS,XMSS-MT). A análise indicou que aproximadamente 90% do custo computacional desses esquemas é de aplicações da função de resumo utilizada.

Assim, este trabalho visou reduzir os custos com as função de resumo. Foram analisadas as arquiteturas dos computadores Haswell e Skylake visando explorar os recursos

disponíveis para implementar os algoritmos de forma eficiente. Foram implementadas funções de resumo, e as principais funções dos esquemas de assinatura que utilizam essas funções de resumo, utilizando técnicas que exploram o paralelismo de dados e os recursos dos computadores utilizados.

Com base nos resultados obtidos e nas análises realizadas, as implementações demonstram ser mais eficientes com o uso das instruções vetoriais AVX2. Foi ilustrado concretamente o desempenho obtido na implementação das funções de resumo SHA2 e SHA3. Também foi ilustrado que, aplicar a mesma técnica nas funções dos esquemas refletiu na melhoria de desempenho do LMS e do XMSS com SHA2 e com SHA3.

Referências Bibliográficas

[1] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. Sphincs: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, páginas 368–397. Springer, 2015.

[2] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. Xmss-a practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-Quantum Cryptography*, páginas 117–129. Springer, 2011.

[3] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In *Applied Cryptography and Network Security*, páginas 31–45. Springer, 2007.

[4] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In *International Workshop on Post-Quantum Cryptography*, páginas 63–78. Springer, 2008.

[5] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. Hash-based digital signature schemes. In *Post-Quantum Cryptography*, páginas 35–93. Springer, 2009.

[6] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. *Hash-based Digital Signature Schemes*, páginas 35–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. doi:10.1007/978-3-540-88702-7_3.

[7] Johannes Buchmann, Luis Carlos Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. Cmss–an improved merkle signature scheme. In *International Conference on Cryptology in India*, páginas 349–363. Springer, 2006.

[8] R. Cabral. Implementação eficiente das funções de resumo criptográfico: Sha-3 e quark, 2015. Disponível em: www.bibliotecadigital.unicamp.br/document/?down=000964406.

[9] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. *Merkle-Damgård Revisited: How to Construct a Hash Function*, páginas 430–448. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/11535218_26.

[10] Ana Karina D. S. de Oliveira and Julio López. An efficient software implementation of the hash-based signature scheme mss and its variants. In Kristin Lauter and

Francisco"Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, páginas 366–383. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-22174-8_20.

[11] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[12] Harold M. Edwards. A normal form for elliptic curves, 2007. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.139.567.

[13] Armando Faz-Hernández, Roberto Cabral, Diego F. Aranha, and Julio López. Implementação Eficiente e Segura de Algoritmos Criptográficos. In *Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - Minicursos*, volume XV, páginas 93–140. Sociedade Brasileira de Computação, 2015.

[14] N. Ferguson, T. Kohno, and B. Schneier. *Cryptography Engineering*. John Wiley Consumer, 2010.

[15] NIST FIPS. 186 digital signature standard, 1994. Disponível em: https://oag.ca.gov/sites/all/files/agweb/pdfs/erds1/fips_pub_07_2013.pdf.

[16] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 2016.

[17] M. Peeters G. Bertoni, J.Daemen and G. Van Assche. Sponge functions, 2007. Disponível em: http://sponge.noekeon.org/.

[18] M. Peeters G. Bertoni, J.Daemen and G. Van Assche. Cryptographic sponge functions, 2011. Disponível em: http://sponge.noekeon.org/CSF-0.1.pdf.

[19] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988. doi:10.1137/0217017.

[20] L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual Symposium on the Theory of Computing*, páginas 212–219. ACM Press, 1996.

[21] A. Hülsing. Practical forward secure signature using minimal security assumptions. In *PhD thesis*. TU Darmstadt, 2013.

[22] A. Hülsing., D. Butin, and S. Gazdag. Xmss: Extended hash-based signatures draft-xmss-00, 2015. Disponível em: https://datatracker.ietf.org/doc/draft-irtf-cfrg-xmss-hash-based-signatures/.

[23] A. Hülsing, D. Butin, S. (2016) Gazdag, and A. Mohaisen. Xmss: Extended hash-based signatures, 2016. Work in progress, Crypto Forum Research Group, Internet Draft, draft-xmss-06. Disponível em: `https://datatracker.ietf.org/doc/draft-irtf-cfrg-xmss-hash-based-signatures/`.

[24] A. Hülsing and J. Rijneveld. Implementation of xmss and xmssmt as specified in draft-huelsing-cfrg-hash-sig-xmss-06, 2016. Disponível em: `https://huelsing.wordpress.com/code`.

[25] Andreas Hülsing. W-ots+-shorter signatures for hash-based signature schemes. *Africacrypt*, 7918:173–188, 2013.

[26] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for xmss-mt. In *International Conference on Availability, Reliability, and Security*, páginas 194–208. Springer, 2013.

[27] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *Public-Key Cryptography – PKC 2016: 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*, páginas 387–416. Springer Berlin Heidelberg, 2016.

[28] INTEL. Intel 64 and ia-32 architectures optimization reference manual, 2016. Disponível em: `www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[29] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal merkle tree representation and traversal. In *Cryptographers' Track at the RSA Conference*, páginas 314–326. Springer, 2003.

[30] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001. `doi:10.1007/s102070100002`.

[31] J. Katz. Analysis of a proposed hash-based signature standard, 2015. Disponível em: `http://www.cs.umd.edu/~jkatz/papers/HashBasedSigs.pdf`.

[32] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[33] Leslie Lamport. Constructing digital signatures from a one-way function. Relatório técnico, Technical Report CSL-98, SRI International Palo Alto, 1979.

[34] Frank T Leighton and Silvio Micali. Large provably fast and secure digital signature schemes based on secure hash functions, Julho 11 1995. US Patent 5,432,852.

[35] Pierre Karpman Ange Albertini Yarik Markov Marc Stevens, Elie Bursztein. The first collision for full sha-1, 2017. Disponível em: `https://shattered.io`.

[36] D. McGrew and M. Curcio. Hash-based signatures draft-mcgrew-hash-sigs-02, 2014. Disponível em: `https://tools.ietf.org/html/draft-mcgrew-hash-sigs-00`.

[37] D. McGrew, M. Curcio, and S. Fluhrer. Hash-based signatures, 2016. Work in progress, draft-mcgrew-hash-sigs-05. Disponível em: `https://tools.ietf.org/html/draft-mcgrew-hash-sigs-05`.

[38] David McGrew, Panos Kampanakis, Scott Fluhrer, Stefan-Lukas Gazdag, Denis Butin, and Johannes Buchmann. State management for hash-based signatures, 2016.

[39] Ralph C. Merkle. Secrecy, authentication, and public key systems, 1979. Ph.D. thesis, Electrical Engineering, Stanford.

[40] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, páginas 218–238. Springer, 1989.

[41] National Institute of Standards and Technology. FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Agosto 2015. Disponível em: `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

[42] NIST. Secure Hash Standard(SHS). Disponível em: `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`.

[43] NIST. Announcing the development of new hash algorithm(s) for the revision of federal information processing standard (fips) 180–2, secure hash standard, 2007. Disponível em: `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Jan07.pdf`.

[44] NIST. Digital Signature Standard (DSS), 2007. Disponível em: `http://csrc.nist.gov/publications/fips`.

[45] NIST. Recommendation for key derivation using pseudorandom functions, 2009. Disponível em: `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf`.

[46] NIST. Recommendation for applications using approved hash algorithms, 2012. Disponível em: `http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf`.

[47] NIST. Post-quantum cryptography standardization, 2016. Disponível em: `http://csrc.nist.gov/groups/ST/post-quantum-crypto/index.html`.

[48] Raphael Overbeck and Nicolas Sendrier. *Code-based cryptography*, páginas 95–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. `doi:10.1007/978-3-540-88702-7_4`.

[49] Geovandro C.C.F. Pereira, Cassius Puodzius, and Paulo S.L.M. Barreto. Shorter hash-based signatures. *J. Syst. Softw.*, 116(C):95–100, 2016. `doi:10.1016/j.jss.2015.07.007`.

[50] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Fevereiro 1978. doi:10.1145/359340.359342.

[51] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. *Fast Hash-Based Signatures on Constrained Devices*, páginas 104–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-85893-5_8.

[52] J. Rompel. One-way functions are necessary and sufficient for secure signatures. In *In STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, páginas 387–394. ACM Press, 1990.

[53] Vlad Krasnov Shay Gueron. Simultaneous hashing of multiple messages, 2012.

[54] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, páginas 124–134. IEEE, 1994.

[55] Michael Szydlo. Merkle tree traversal in log space and time (2003 preprint version), 2003. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.3523.

[56] Michael Szydlo. *Merkle Tree Traversal in Log Space and Time*, páginas 541–554. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-24676-3_32.

Apêndice A

Algoritmos

O Algoritmo 31 descreve a função *checksum* (CKSM) do LM-OTS especificado na seção 4.1.4.

Algorithm 31 Função CKSM [37]

Entrada: cadeia S, o valor ls.
Saída: a soma de verificação (sum).

$sum = 0$;
for $(i = 0; i < u; i = i + 1)$ **do**
    $sum = sum + (2^w - 1)$ - $coef(S, i, w)$;
end for
return $(sum \ll ls)$;

O Algoritmo 32 descreve a função `baseW`() do XMSS especificado na seção 4.2.2.

Algorithm 32 Função `baseW`[23]

Entrada: a cadeia X, o parâmetro w e o valor $outlen$.
Saída: vetor $basew$.

$in = 0$;
$out = 0$;
$total = 0$;
$bits = 0$;
for $(i = 0; i < outlen;$ i++) **do**
    if ($bits == 0$) **then**
        $total = X[in]$;
        $in = in + 1$;
        $bits = bits + 8$;
    end if
    $bits = bits - lg(w)$;
    $basew[out] = (total \gg bits)$ `AND` $(w - 1)$;
    $out = out + 1$;
end for
return $basew$;

O Algoritmo 33 ilustra o processo de geração de chaves XMSS-MT especificado na seção 4.2.6.

Algorithm 33 $XMSSMTKeyGen$ [23]

Entrada: a altura h e o número de camadas d.
Saída: a chave secreta $skMT$, a chave pública $pkMT$.

$indMT = 0$;
$skMT = indMT$;
//inicializa $skPrf$ com uma cadeia aleatória de n bytes;
$skMT = skPrf$;
//inicializa Seed com uma cadeia aleatória de n bytes;
$skMT = Seed$;
Gerar a chave secreta do XMSS
$Adrs = 0_{32}$;
for ($layer = 0; layer < d$; layer++) **do**
  $Adrs[0] = layer$; // layerAdrs=layer
  for ($tree = 0; tree < (1 \ll ((d - 1 - layer).(h/d))); tree++$) **do**
    $Adrs[1] = tree$; // treeAdrs=tree
    for ($i = 0; i < 2^h; i++$) **do**
      $WOTSgenSK(wots\_sk[i])$;
    end for
    $XMSS\_SK\{skMT, SeedSK, tree, layer\}$;
  end for
end for
$SK = getXMSS\_SK(skMT, 0, d - 1)$;
$SK.Seed = Seed$;
$root = treeHash(SK, 0, h/d, Adrs)$;
$skMT = root$;
$pkMT = \{root||Seed\}$;
return $\{skMT||pkMT\}$;

O Algoritmo 34 ilustra o processo de geração de chaves XMSS-MT especificado na seção 4.2.6.

Algorithm 34 $XMSSMTsign$ [23]

Entrada: a mensagem M e a chave secreta $skMT$.
Saída: a chave secreta $skMT$ atualizada e a assinatura $sigMT$.

// Inicialização
Adrs = 0_{32};
Seed = getsmt($skMT$);
$skPrf = getskPrf(skMT)$;
$indSig$ = getIdx($skMT$);
// Atualiza $skMT$
$skMT.Ind = indSig + 1$;
// Compressão da mensagem
$r = PRF(skPrf, indSig_{32})$;
$M' = Fmsg((r||getRoot(skMT)||idxSig_n), M)$;
// Assina a mensagem
$sigMT = indSig$;
$indArv = (h - h/d)$// bits mais significativos de $indSig$;
$indFlh = (h/d)$ //bits menos significativos de $indSig$;
$SK = \{indFlh||getXMSS\_SK(skMT, indArv, 0)||skPrf||0_n||Seed\}$;
$Adrs[0] = 0$; //layerAdrs=0
$Adrs[1] = indArv$; //treeAdrs=indArv
$sigTmp = treeSig(M', SK, Adrs)$;
$sigMT = \{sigMT||r||sigTmp\}$;
for ($j = 1; j < d; j++$) **do**
    $root = treeHash(SK, 0, h/d, Adrs)$;
    $indFlh = (h/d)$ // bits menos significativos de $indArv$;
    $indArv = (h - j.(h/d))$ // bits mais significativos de $indArv$;
    SK = $\{indFlh||getXMSS\_SK(skMT, indArv, j)||skPrf||0_n||Seed\}$;
    $Adrs[0] = j$;//layerAdrs=j
    $Adrs[1] = indArv$;//treeAdrs=indArv
    $sigTmp = treeSig(root, SK, Adrs)$;
    $sigMT = \{sigMT||sigTmp\}$;
end for
return $\{skMT||sigMT\}$;

O Algoritmo 35 descreve o processo de geração de chaves XMSS-MT especificado na seção 4.2.6.

Algorithm 35 $XMSSMTverify$ [23]

Entrada: a mensagem M, a assinatura $sigMT$ e a chave pública $pkMT$.
Saída: resultado da verificação.

$indSig = sigMT.Ind$;
$Seed = pkMT.Seed$;
$Adrs = 0_{32}$;
$M' = Fmsg(getR(sigMT)||getRoot(pkMT)||indSig, M)$;
$indFlh = (h/d)$ //bits menos significativos de $indSig$;
$indArv = (h - h/d)$ //bits mais significativos de $indSig$;
$Sig' = getXMSSSignature(sigMT, 0)$;
$Adrs[0] = 0$; //layerAdrs=0
$Adrs[1] = indArv$; //treeAdrs=indArv
$no = XMSSrootFromSig(indFlh, Sig'.sigOts, Sig'.Aut, M', Seed, Adrs)$;
for $(j = 1; j < d; j++)$ **do**
    $indFlh = (h/d)$ //bits menos significativos de $indArv$;
    $indArv = (h - j.h/d)$ //bits mais significativos de $indArv$;
    $Sig' = getXMSSSignature(sigMT, j)$;
    $Adrs[0] = j$; //layerAdrs=j
    $Adrs[1] = indArv$; //treeAdrs=indArv
    $no = XMSSrootFromSig(indFlh, getSig\_ots(Sig'), Sig'.Aut, no, Seed, Adrs)$;
end for
if $(no == pkMT.root)$ **then**
    return VÁLIDO;
else
    return INVÁLIDO;
end if