Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo e implementação do algoritmo de resumo criptográfico SHA-3

Gracielle Forechi Olivier

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Diego de Freitas Aranha

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Flávio de Barros Vidal

Banca examinadora composta por:

Prof. Diego de Freitas Aranha (Orientador) — CIC/UnB
Prof. João José Costa Gondim — CIC/UnB
Prof. Pedro Antônio de Dourado Rezende — CIC/UnB

CIP — Catalogação Internacional na Publicação

Olivier, Gracielle Forechi.

Estudo e implementação do algoritmo de resumo criptográfico SHA-3 / Gracielle Forechi Olivier. Brasília : UnB, 2013.
107 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. funções de resumo criptográfico, 2. Keccak, 3. SHA-3,
4. implementação em *software*

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Estudo e implementação do algoritmo de resumo criptográfico SHA-3

Gracielle Forechi Olivier

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Diego de Freitas Aranha (Orientador)
CIC/UnB

Prof. João José Costa Gondim
CIC/UnB

Prof. Pedro Antônio de Dourado Rezende
CIC/UnB

Prof. Flávio de Barros Vidal
Coordenador do Curso de Computação — Licenciatura

Brasília, 26 de julho de 2013

Dedicatória

Dedico este trabalho à Deus que sua grandeza se fez presente em todas as etapas de construção, dando apoio e inspiração nos momentos fáceis e difíceis. Dedico também a minha família que me apoiou e proporcionou que tudo isso fosse possível.

Agradecimentos

Agradeço primeiramente aos meus familiares e amigos sem dos quais esse projeto jamais teria sido concretizado. E todos os demais que de alguma forma, seja dando suporte emocional, seja contribuindo para edificar meu conhecimento, permitiram que fosse capaz.

Resumo

A segurança computacional como campo fortemente estudado precisa sofrer mudanças constantemente, sempre buscando novas soluções e descobertas. As funções de resumo criptográfico não são diferentes nesse aspecto, sendo alvo de estudo e tentativas de quebras. No entanto, muitas aplicações dependem do uso dessas funções de resumo, pois são essenciais para provar a integridade de mensagens.

Várias das funções utilizadas como funções de resumo sofreram quebras, como MD5 e SHA-1, sobrando assim apenas a família de algoritmos SHA-2. Sendo essa a única função sem vulnerabilidades graves conhecidas, passa a ser fortemente estudada e alvo de tentativas de quebras. Com sua estrutura sendo questionada e estudada, apontou que o uso prolongado desse algoritmo deve ser cauteloso. Para isso o NIST promoveu um concurso para eleger outro algoritmo e trazer mais uma alternativa confiável de implementação de funções de resumo. Após 5 anos de concurso, a proposta vencedora *Keccak* passou a ser o novo padrão SHA-3.

Esse algoritmo faz uso do paradigma esponja, composto por duas fases de processamento. A primeira delas divide a mensagem em blocos e os absorve em estados internos. Esses estados são originados a partir de um estado sendo inicializado com zeros e, em seguida, passa a ser iterado com rodadas que possuem cinco mapeamentos, que fazem a difusão e a distribuição dos elementos nos estados. Depois de finalizados, a função passa para a fase de esmagamento, que por sua vez intercala a aplicação das funções de mapeamento até que se tenha o número de *bits* que atende o tamanho da saída no nível de segurança escolhido.

Esse trabalho faz um estudo do algoritmo e objetiva construir uma versão didática da implementação que corresponde ao *Keccak*. Essa versão é construída focando no entendimento de aspectos conceituais. Para isso, se buscará relacionar suas definições com elementos práticos são responsáveis por seu funcionamento, seguindo os modelos propostos pela literatura.

Palavras-chave: funções de resumo criptográfico, Keccak, SHA-3, implementação em *software*

Abstract

As a thoroughly studied topic, computer security must change frequently, searching for new soluctions and discoveries. Hash functions are no different in this regard, being a target of advanced study and attack attempts, since several different applications rely on them for security. Verifying the integrity of messages is perhaps the essential application of hash functions.

Many of the functions used as cryptographic hash functions suffered sucessful attacks, as in the case of MD5 and SHA-1, remaining only the SHA-2 family of algorithms as a viable option. However, being the only viable option is not healthy from a security point of view due to the concentrated attacks and the similarities between the SHA-2 and SHA-1 structures. This way, NIST advised against long-term use of the algorithm and promoted a challenge to elect the new standard, bringing a reliable alternative for the implementation of cryptographic hash functions. After five years, the winning proposal *Keccak* became the SHA-3 standard.

This algorithm follows the sponge paradigm, being composed of two processing phases. The first phase splits the message in different blocks and absorbs theses blocks in internal states. These states are originated from the iteration of a zero state with a round function composed of five different mappings, responsible for performing the diffusion and dispersion of the blocks in the states. After the absorbing phase, the squeenzing phase beings, extracting information with round functions until the desired output bits are produced at a chosen security level.

This work studies the *Keccak* hash function and presents a didactic implementation. By focusing on understanding the conceptual aspects, one can relate them to the definition elements and pratical functioning of the model proposed in the literature.

Keywords: cryptographic hash functions, Keccak, SHA-3, software implementation

Sumário

Lista de Figuras

Capítulo 1

Introdução

A Segurança da Informação é um dos ramos da computação que mais tem ganhado destaque nas discussões no cenário acadêmico, industrial e nos demais setores que compõem a Informática. Cada vez mais se discute a importância de restrições e sigilos, elementos fundamentais que trazem para quem faz uso do meio tecnológico um cenário cheio de nuances. Esse ambiente tecnológico, hoje, não visa apenas facilidades, mas também condições para assegurar confiança nas atividades desempenhadas.

Diversas são as ferramentas desenvolvidas para tentar alcançar tal finalidade. A Criptografia é um dos elementos essenciais dentro desses mecanismos, por causa de sua ampla difusão e gama de funcionalidades. Dentre um conjunto de outros recursos, se mostra uma das mais importantes formas de se prover a segurança que se almeja.

A Criptografia é definida classicamente como sendo a forma de habilitar duas pessoas a transmitir uma mensagem em um canal inseguro, sem que um oponente consiga acessar essa mensagem [20]. Sintetizando, é a maneira de assegurar que somente as pessoas destinadas visualizem dados trafegados, mesmo que em meios vulneráveis. Uma definição mais moderna estende os serviços de segurança fornecidos pela Criptografia para além da confidencialidade, incluindo autenticação, verificação de integridade, irretratabilidade (não-repúdio) e até anonimização.

1.1 Motivação

Dentro de todos os elementos estudados na Segurança da Informação, especialmente na área de Criptografia, as funções de resumo criptográfico, também chamadas de funções de *hash* criptográfico, possuem grandes aplicações e em diversos segmentos. Seu vasto uso pode ser notado como por exemplo no auxílio para construção de protocolos criptográficos ou também como verificador de autenticidade de mensagens, entre outras finalidades. Tavares [10] cita possíveis aplicações de uma função de *hash* criptográfica, dentre as quais:

- garantia de integridade dos dados e da origem de uma mensagem;
- cálculo de respostas que são função de uma chave secreta e de uma mensagem de desafio (em protocolos de identificação através de desafio);
- confirmação de chave;

- confirmação de conhecimento (habilidade de comprovar conhecimento prévio de algo sem a necessidade de expor os dados previamente);
- derivação de chave;
- geração de números pseudoaleatórios.

Uma função de resumo criptográfico serve para mapear uma cadeia de *bits* de tamanho arbitrário a uma cadeia de *bits* de tamanho fixo. A saída dessa função chama-se resumo. Essa função deve ser capaz de prover integridade, isto é, cada mensagem produz um único resumo de saída e, sempre que essa mesma mensagem é submetida à função de resumo criptográfico, deve sempre produzir a mesma saída. Podem haver entradas diferentes que, quando submetidas a essas função, produzem a mesma saída, as chamadas colisões, algumas vezes causadas por eventos indesejáveis. A esses eventos damos o nome de vulnerabilidades das funções de resumo criptográfico. São feitos esforços no sentido que essas vulnerabilidades não sejam usadas para atacar essas funções, no qual atacar corresponde a usar uma vulnerabilidade ou causar uma, intencionalmente, a fim de obter vantagem ilícita ou denegrir algum dos serviços de segurança.

Outro elemento e um dos focos da aplicação dessas funções é a integridade. Isso se deve ao fato que, como define Stinson [20], essas funções fazem uma *"fingerprint"*, o qual seria uma identidade digital do conteúdo. Assim, o texto que é colocado como entrada desses algoritmos é identificado, o que corresponde a dizer que produzem uma cadeia de *bits* que uma vez mapeado é unicamente relacionado. A consequência é que somente o número gerado é ligado ao texto que o gerou, pois existe uma relação matemática entre ambos.

Alterações no conteúdo do texto de entrada invalidam a identificação digital. Inclusive, o fato de mapear para um único resumo nos permite detectar se ocorreram alterações no seu conteúdo, exceto nos casos de probabilidade de colisão fortuita. Possibilitam também saber se a identidade do remente da mensagem é de fato a esperada pelo receptor. Isto é possível aliando à assinatura digital, que também é meio de prover autenticidade. Explicando sucintamente [20], pode promover uma assinatura, só que digital, análoga à assinatura manual e também as propriedades de integridade e principalmente autenticidade.

Toda sua grande gama de aplicações deve-se a essas características. Além disso elas são baseadas na propriedade das funções matemáticas bijetoras possuírem a propriedade de inversão. Isto diz respeito às funções que, como diz Bizelli [9], desfazem a operação de outras. Para algumas funções, é fácil de se obter a sua função inversa; outras requerem um grande esforço computacional ou sequer já foram obtidas. É esse último grupo de funções com essas propriedades matemáticas que nos interessa para compor a classe das funções de resumo. Isso nos permite que, uma vez de posse do resumo tido por resultado, não podemos dele ter acesso à qualquer das cadeias de *bits* que o podem fornecer como resposta.

Atualmente esse campo de estudo, focado na construção de funções de resumo criptográfico está passando por diversas mudanças. As mais difundidas e antigas funções estão apresentando necessidade de reformulação e adesão a novos paradigmas. Os risco do uso contínuo dos algoritmos tradicionais é alto, pois possuem grandes chances de apresentarem ataques, quebras e falhas. Com os riscos desses impactos, surge assim, na comunidade científica e no ambiente de pesquisadores de modo geral, como relata Nascimento [5], a

indispensabilidade de se criar novas soluções e novas contribuições. Para tanto, NIST [13] pensou num concurso que elegesse outro algoritmo, para que substituisse os antigos e que superasse as deficiências nestes apontadas.

1.2 Objetivo

Esse trabalho tem por objetivo estudar as principais características do algoritmo *Keccak*. Esse algoritmo foi o vencedor do concurso promovido pelo *National Institute of Standards and Technology(NIST)* [13], que objetivava eleger um algoritmo que sucedesse o SHA-2 e suprisse as falhas em versões reduzidas do algoritmo que ameaçam o uso contínuo do padrão para algoritmos de resumo criptográfico, como retratam Nascimento [5] e Marques [24]. Esse estudo será possível por meio de um paralelo comparativo com seus antecessores. É necessário também discutir o paradigma das chamadas funções esponja, usado para inovar e aprimorar o mais antigo, que era o amplamente usado método de Merkle-Damgard. Outro objetivo é levantar características próprias e específicas dessa função e, por fim, fornecer uma implementação didática desse algoritmo, a fim de facilitar outros estudos acerca do assunto.

1.3 Estrutura do documento

Esse trabalho foi estruturado para abordar os tópicos em capítulos. Os primeiros tratam do referêncial teórico usado para construir um entendimento conceitual acerca dos assuntos que se relacionam com o tema. Os finais são direcionados para aplicação dos conceitos entendidos e para relatar a experiência de implementação realizada.

No segundo capítulo, iniciaremos as abordagens teóricas para produzir insumos para compreensão do algoritmo estudado. Nele, tópicos são apresentados com as definições, as propriedades de um resumo criptográfico e as características fundamentais do que vem a ser uma função de resumo criptográfico.

No terceiro capítulo, explicamos o paradigma usado para as antigas versões de resumo e as razões que levaram a motivar o uso de novos paradigmas na construção de resumos criptográficos, abordando exemplos dos algoritmos que antes faziam uso desse paradigma e uma breve descrição de cada um.

No quarto capítulo, abordaremos as notações e convenções do paradigma de funções esponja, quais são suas propriedades, como procede seu funcionamento, quais suas relações com o paradigma anteriormente usado, seus requisitos de segurança e como são pensadas as esponjas aleatórias.

No quinto capítulo, serão apresentadas as definições e notações do algoritmo SHA-3. Relataremos como funciona a estrutura de esponja dentro do algoritmo, como são pensados os elementos de mapeamento e quais as características inerentes de cada uma.

No sexto, a parte anterior será apresentada em nível de implementação. As propriedades conceituais serão relatadas na linguagem de programação escolhida. Estabeleceremos um elo entre os elementos do programa e os que compõem sua definição.

No capítulo final do trabalho, se concluirá apontando os resultados obtidos e fazendo uma análise do estudo realizado acerda de sua contribuição no que tange aspectos concei-

tuais e seu potencial didático. Serão sugeridas propostas de tópicos que podem acrescentar ou fomentar novos trabalhos.

Capítulo 2

Funções de resumo criptográfico

Como dito no capítulo anterior, funções de resumos criptográficos são construídas como funções que visam mapear uma cadeia de *bits* de tamanho arbitrário em cadeias de *bits* de tamanho fixo. Esse mapeamento sempre retorna o mesmo tamanho de saída, independentemente do tamanho que obteve de entrada. Essa cadeia de saída possui tamanho fixo não muito extenso, estabelecido pelo nível de segurança adotado. Busca-se que a saída seja única para cada valor diferente que é submetido, no domínio de suas aplicações, como cadeia de entrada. Uma vez que uma entrada é modificada, essa deve produzir um outro valor de saída. São construídas de maneira que sua inversa não seja computacionalmente viável de ser encontrada, a partir de provas e evidências estatísticas e matemáticas de que a inversão dessas funções tenham de fato esse alto custo computacional. Seu formato é de uma cadeia de caracteres que se assemelha às saídas de funções geradoras de números pseudo-aleátorios, conforme a Figura 2.1.

Em seguida, se aprofundarão seus conceitos e propriedades, entenderão sua estrutura e suas deficiências e por fim se explorarão os modelos de paradigmas existentes no campo.

2.1 Definição de funções de resumo criptográfico

Para compreender seu funcionamento, suponha h uma função de resumo criptográfico. Seja m uma mensagem no formato de cadeia de bits de tamanho arbitrário originada por qualquer mensagem que se deseja produzir o resumo. Se r é a saída da função h quando a entrada é a cadeia m, ou em outras palavras, $r = h(m)$, esse resultado é o valor que permite atestar a integridade da mensagem. Normalmente, r é uma curta cadeia de *bits*, com tamanho aferido pelo nível de segurança do algoritmo empregado na implementação da função de resumo, como se pode visualizar na Figura 2.1.

Quando validamos a integridade de m, precisamor supor que r tenha sido depositado e armazenado em local seguro. No caso do m ter sofrido mudanças e ter se tornado um m', quando submetido à função h produzirá um r' com alta probabilidade, ou o mesmo que dizer que $h(m') = r'$. Como é provável que $r \neq r'$, concluímos que a integridade foi violada.

Uma definição mais objetiva, oferecida por Oliveira [4], é a de que uma função de resumo $h : M \rightarrow R$ mapeia cadeias de *bits* $m \in M$ de tamanho finito e arbitrário para cadeia de *bits* $r \in R$ de tamanho fixo n. Deste modo, $r = h(m)$. O domínio M, dada essa

função h, precisa ser menos extenso que a imagem R [4], para evitar colisões inerentes dessa função e tentar aumentar a distribuição de saídas de h.

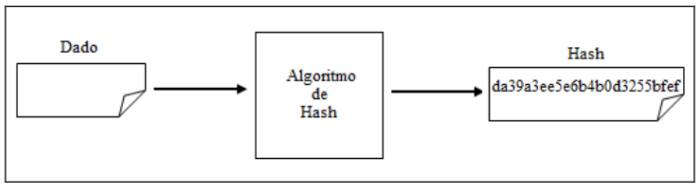

Figura 2.1: Função de Resumo Criptográfico [24]

2.2 Propriedades das funções de resumo criptográfico

As funções de resumos criptográficos possuem características que são específicas de sua construção. Para que se enquadre nessa categoria de algoritmos, uma função deve pelo menos apresentar as seguintes características:

- **Compressão**: as funções de resumo possuem como entrada cadeia de *bits* de qualquer comprimento. Sempre tem-se por saída uma cadeia de *bits* de comprimento fixo. Não importa o tamanho que é obtido de entrada, essa função sempre retorna valores de mesmo comprimento. Por esse motivo dizemos que ela comprime a entrada. Essa propriedade é semelhante à propriedade característica das demais funções de compressão, com a diferença que a compressão não deve ser facilmente reversível.
- **Eficiência**: segundo Tavares [10] as funções de resumo devem possuir rapidez na obtenção de resultados. Eficiência computacional obtida através de otimizações de seus algoritmos e implementações muitas vezes em termos de aritmética.
- **Resistência à primeira pré-imagem**: as funções que fazem parte do grupo de funções de resumo são aquelas em que a entrada não é facilmente obtida a partir da saída. Ou seja, de posse da função $h : M \mapsto R$ e de um resumo $r \in R$, é inviável o cálculo computacional para encontrar um $m \in M$, tal que $h(m) = r$. Essas funções são chamadas de funções de mão única *(one-way)*.
- **Resistência à segunda pré-imagem**: para uma mensagem $m \in M$, é inviável o cálculo computacional para encontrar uma $m' \in M$, tal que $m \neq m'$ e $h(r) = h(r')$. Essas funções são chamadas de funções fracamente resistentes à colisão *(weak collision resistance)*.
- **Resistência à colisões**: em uma função de resumo é computacionalmente difícil encontrar para duas entradas $m, m' \in M$, tais que $m \neq m'$, com $h(m) = h(m')$, para a função $h : M \mapsto R$. A essas funções atribui-se o nome de funções fortemente resistentes à colisão *(strong collision resistance)*.

Essas propriedades estão relacionadas e são facilmente confundidas. No entanto, uma não implica necessáriamente em outra, mas pode vir a implicar. Por exemplo, resistência à colisão garante a resistência à segunda pré-imagem, mas o contrário não é válido [4].

Para entender melhor as diferenças e consequências que acarretam cada uma dessas propriedades pode-de descrever cenários para entendimento. O primeiro deles é o cenário para compreender a propriedade de resistência à primeira pré-imagem. Neste tem-se inicialmente, o conhecimento do algoritmo da função de resumo criptográfico e um resumo que seja saída dessa mesma função. A resistência à primeira pré-imagem é a inviabilidade computacional de encontrar uma mensagem, que uma vez submetida à essa função, retorne o mesmo resumo que se obteve previamente. A mensagem pode ser qualquer uma do escopo de entrada, desde que, quando seja processada, retorne exatamente um resumo igual ao que se tinha antes de processá-la.

A resistência à primeira pré-imagem só é possível pela dificuldade de encontrar as inversas das funções escolhidas, mesmo em ambientes computacionais com grande capacidade de processamento. Para isso, se escolhem funções compostas de operações que aumentem a complexidade no cálculo da inversa. A exemplo dessa complexidade, a operação subtração que é a inversa da adição e no caso das operações de multiplicação, essas são inversas das operações de fatoração. Calcular a inversa de um número na operação de adição é consideravelmente mais fácil de obter que a inversa nas operações de fatoração. Essa complexidade que garante essa propriedade de resistência à primeira pré-imagem.

A resistência à segunda pré-imagem, em outro cenário, é possível quando se conhece o algoritmo da função de resumo criptográfico e uma mensagem que pertença ao escopo de mensagens de entrada dessa mesma função. De posse desses elementos prévios, tem que demandar alto custo computacional para encontrar uma outra mensagem, diferente da anterior, que também faça parte do escopo de mensagem de entrada dessa função, que produza o mesmo resumo que a mensagem previamente conhecida. Quando duas mensagens diferentes produzem o mesmo resumo, sendo que já se conhecia um delas, o algoritmo não respondeu à propriedade de resistência à segunda pré-imagem.

Quanto ao caso da resistência a colisão, conhecendo-se o algoritmo da função de resumo criptográfico, deve ser inviável encontrar duas mensagens diferentes que fazem parte do escopo de mensagens de entrada dessa mesma função e que essas mensagens produzam o mesmo resumo. A diferença da resistência à colisão para as demais resistências é que não se conhece previamente as mensagens ou mesmo algum resumo. Consistem, a partir do conhecimento da função, em se obter duas mensagens que produzam resumos iguais.

Esses cenários nos permitem entender que as funções de resumo apresentam características específicas, muito embora correlatas. Todas de algum modo evitam alguns ataques e preservam as propriedades intuivamente esperadas de funções de resumo, evitando assim a colisão por parte das saídas das funções, mas também condições de vulnerabilidades para encontrar indícios de vulnerabilidades comparando saída e entrada. Outro aspecto que limita essas propriedades é que inverter as funções é sempre possível, pois todas as funções do universo matemático possuem inversa, mesmo que seu cálculo seja difícil. Sendo assim, busca-se as de alta complexidade computacional, mas essa complexidade pode sofrer interferências toda vez que se encontre um mecanismo na estrutura da função que acelere esses cálculos.

2.2.1 Ataque de aniversário

Um conhecido ataque às funções de resumo criptográfico chama-se ataque de aniversário. Esse ataque possui ênfase na propriedade de resistência à colisão. Ou seja, consiste

em achar duas mensagens diferentes quaisquer que gerem resumos idênticos, como define Rezende [2]. Formalmente, trata-se de encontrar $m, m' \in M'$, sendo M o conjunto de mensagens, no qual $m \neq m'$, tais que $h(m) = h(m')$. Cheswick [3] explica o impacto de ataques de aniversário na medida de nível de segurança do algoritmo. No caso de função de resumo criptográfico, reduz pela metade o nível de segurança anterior. A esse problema se chama ataque de aniversário, por se assemelhar ao problema conhecido como Paradoxo do Aniversário, onde se afirma que em um grupo de 23 pessoas(ou mais), agrupadas de forma aleatória, a chance de que duas ou mais pessoas façam aniversário na mesma data é maior que 50%. Se no grupo houver 57 pessoas, a probabilidade para sobe 99%; que pode chegar a 100% se no grupo houver 367 pessoas.

A consequência do ataque de aniversário é uma diminuição no esforço computacional para encontrar uma colisão de 2^n para $2^{\frac{n}{2}}$ execuções da função de resumo criptográfico. Isso provoca uma redução no nível de segurança para $\frac{n}{2}$ *bits* e acarreta na necessidade de dobrar a quantidade de *bits* de saída. Sendo assim, quando se quer um nível de segurança com 80 *bits* (determinado pelo nível de segurança estabelecido) deve-se pensar no tamanho de 160 *bits*. Em razão dessa dimuição da segurança, para Cheswick [3], funções de resumos criptográficas seguras precisam ter comprimento de saída de pelo menos 160 *bits*. Valores menores que esses diminuem os valores possíveis para saída e aumentam as possibilidades de colisões.

Segundo Oliveira [4], os atacantes podem ocasionar mudanças no algoritmo. Essas modificações envolvem principalmente o estado inicial, quando nele se recebe o vetor de inicialização, normalmente estabelecido pelo projetista do algoritmo. Devido o valor desse vetor ser fixo e importante para manter a segurança das diversas construções. Esse vetor de inicialização atrai atacantes que buscam características para procurar por vulnerabilidades. Exemplos de ataques que surgem desse estado inicial são:

- **Pseudocolisão (ou colisão de início livre)**: sendo h uma função de resumo, consistem em encontrar uma colisão que envolva estados iniciais diferentes e mensagens diferentes, mas que tenham por resultado resumos iguais, ou seja, encontrar $m, m' \in M$, no qual $m \neq m'$, sendo V, V' estados iniciais diferentes e a colisão ocorre para $h(V, m) = h(V', m')$.
- **Colisão de início semi-livre**: sendo h uma função de resumo e V um estado inicial qualquer, consistem em encontrar $m, m' \in M$, no qual $m \neq m'$ e a colisão ocorre como $h(V, m) = h(V, m')$.

Ainda segundo Oliveira [4], variantes desses ataques podem ser aplicados às pré-imagens.

Os ataques de pseudocolisão e colisão de início semi-livre não possuem variável de estado que é iterada, como é mais comum nos algoritmos de resumo criptográficos reais. Nestes, o comum é encontrar as variáveis de estado sendo iteradas e processadas junto aos blocos de mensagem. Essas variáveis são chamadas variáveis de estado ou variáveis encadeadas. Para essas podemos listas ataques como:

- **Ataque de correção de bloco**: sejam m, m' mensagens de M, no qual $m \neq m'$ essas são modificadas para produzir $h(m) = h(m')$. Em geral, apenas o primeiro e último blocos são alterados.

- **Ataque de meet-in-the-middle**: ocasionam-se colisões com variáveis encadeadas intermediárias. Nele, o atacante escolhe um estado intermediário da iteração do algoritmo H_i e faz o cálculo de $H_i = h(H_{i-1}, m_i)$ e $H_i = h^{-1}(H_{i+1}, m_{i+1})$. Na tentativa de encontrar alguma colisões para H_i. Se for encontrada o atacante encontra colisão para toda a mensagem.

- **Ataque de ponto fixo**: seja h uma função de resumo, neste o atacante procura um par (H_{i-1}, m_i), tal que $h(H_{i-1}, m_i) = H_{i-1}$. Com isso, se produzem colisões em série apenas concantenando os blocos da mensagem sequencialmente.

- **Ataque nas funções de compressão**: essas funções de compressão, que serão mais detalhadas no capítulo posterior, possuem funcionamento semelhantes às cifras de blocos, por processarem blocos da mensagem, ao invés de *bit-a-bit* como ocorre na cifra de fluxo. Elas são estudadas há bastante tempo, com o objetivo de alertar projetistas a se previnirem de ataques e causas análogas.

- **Ataques por meio de oráculo aleatório**: esse modelo embora não seja concreto, pode gerar ataques de efeitos futuros, visto que o modelo de oráculo aleatório é o modelo ideal de função de resumo criptográfico e, que por isso, os modelos reais não atinjam a mesma resistência à ataques que o modelo idealizado.

O próximo capítulo, trata do paradigma de Merkle Damgard e suas propriedades. Detalhando o funcionamento do paradigma e como esse foi empregado em várias funções de resumo criptográfico.

Capítulo 3

Paradigma Merkle-Damgard

O paradigma Merkle-Damgard foi construído pelo pesquisador e professor Ralph Merkle e pelo também professor e pesquisador Ivan Damgard no ano de 1979 [17]. Hoje é um dos mais conhecidos paradigmas de construção de funções de resumo criptográfico na criptografia moderna. Empregado na construção de algoritmos como MD5, SHA1 e SHA2, o método consiste em transformar funções de compressão resistentes à colisão em funções de resumo criptográfico.

3.1 Definição do Paradigma de Merkle-Damgard

Primeiramente, é importante o entendimento das funções de resumo iteradas. Essas funções são criadas fazendo uso de funções de compressão, que transformam entradas de quaisquer tamanhos em saídas de tamanho fixo, com a propridade de comprimir qualquer extensão, larga ou não, em uma cadeia de tamanho fixo. Estas são construídas para funcionarem como uma função de resumo, ou seja, que recebe mensagens longas e reduz a mensagens mais curtas. Restringimos também o domínio dessas funções, no qual entradas e saídas são cadeia de *bits*. Adotando a notação de $|x|$ para o tamanho da cadeia e de $||$ correspondente à notação da operação de concatenação, sendo assim, $x||y$ equivale dizer a concatenação das cadeias x e y.

Suponha que $f : \{0,1\}^{\{m+t\}} \rightarrow \{0,1\}^m$ é uma função de compressão, na qual $t \geq 0$. Denotamos funções de resumo iteradas da seguinte forma:

$$h : \bigcup_{i=m+t+1}^{\infty} \{0,1\}^i \rightarrow \{0,1\}^m$$

Essa definição é baseada na função de compressão f. Para entendermos o comportamento da função h, podemos seguir três passos:

- **Pré-processamento**: A partir de uma cadeia x como entrada, no qual tem-se que $|x| \geq m + t + 1$, construir uma cadeia y, de tal forma que $|y| \equiv 0$ $(mod\ t)$. Denote assim, $|y| = y_1||y_2||...||y_p$, tal que $|y_i| = t$ para $1 < i < p$.
- **Processamento**: Considere IV um valor inicial conhecido, sendo ele uma cadeia de *bits* de tamanho m. Com isso calculamos:

$$Z_0 \leftarrow IV$$

$$Z_1 \leftarrow f : (Z_0||y_1)$$

$$Z_2 \leftarrow f : (Z_1||y_2)$$

$$Z_3 \leftarrow f : (Z_2||y_3)$$

$$\dots$$

$$Z_p \leftarrow f : (Z_{p-1}||y_p)$$

- **Transformação da saída**: Considere $g : \{0,1\}^m \rightarrow \{0,1\}^l$, uma função. Define-se a função iterada como sendo $h(x) = g(Z_p)$. Esse passo não é uma exigência da função de resumo iterada, uma vez que podemos considerar $h(x) = Z_p$.

É comum encontrarmos na construção da função h, uma função de preenchimento (função que completa com *bits* uma cadeia para que atinja o tamanho definido para o tamanho dos blocos, denotado por t). Com isso, a construção da função de resumo iterada é da forma $y = m||pad(x)$. Assim, a função de preenchimento *pad* incorpora *bits* na cadeia x (que por exemplo podem ser todos *bits* 0), até que a *string* y tenha um tamanho que seja múltiplo de t.

O passo de pré-processamento precisa garantir que o mapeamento $X \rightarrow Y$ é uma injeção [20]. Esse fato que garante a inexistência de colisão. Essa garantia ocorre devido à possibilidade do mapeamento $x \mapsto y$ ser um para um (*one-to-one*). Para garantir a resistência ao mapeamento no pré-processamento, deve-se considerar valor de $|y| = pt > |x|$, exigida pela injeção.

3.2 Construção do Paradigma de Merkle-Damgard

O método de Merkle Damgard é uma forma de construção de funções de resumo iteradas. Essa construção é capaz de garantir as propriedades de resistência à colisão, quando provida por sua função de compressão.

Suponha que $f : \{0,1\}^{\{m+t\}} \rightarrow \{0,1\}^m$, no qual f é uma função de compressão resistente à colisão, tal que $t > 1$. Usando f para construir uma função de resumo criptográfico resistente à colisão denotada por $h : x \rightarrow \{0,1\}^m$, temos que:

$$x = \bigcup_{i=m+t+1}^{\infty} \{0,1\}^i$$

O algoritmo que descreve o funcionamento dessa função pode ser escrito como no Algoritmo 1:

Algoritmo 1 Merkle-Damgard

Comentário: $compressao : \{0,1\}^{m+t} \rightarrow \{0,1\}^m$, no qual $t \geq 0$
$n \leftarrow |x|$
$k \leftarrow [n \backslash (t-1)]$
$d \leftarrow k(t-1) - n$
for $i \leftarrow 1$ $k-1$ **do**
  $y_i \leftarrow x_i$
end for
$y_k \leftarrow x_k || 0^d$
$y_{k+1} \leftarrow$ a representação binária de d
$Z_1 \leftarrow 0^{m+1} || y_1$
$g_1 \leftarrow compressao(Z_1)$
for $i \leftarrow 1$ **to** k **do**
  $Z_{i+1} \leftarrow g_i || 1 || y_{i+1}$
  $g_{i+1} \leftarrow compressao(Z_{i+1})$
end for
$h(x) \leftarrow g_{k+1}$
return $h(x)$

O algoritmo inicialmente considera valores de $t \geq 2$ para satisfazer a condição de que $|y| = pt > |x|$. Essa exigência é atribuída às demais funções de compressão que são resistentes à colisão, a fim de garantir que saída será de comprimento menor que a entrada. Supondo valores de x, tal que $x \in X$ e sendo x uma cadeia de *bits*. Consequentemente, a cadeia x é definida como sendo uma concatenação $|x| = x_1 \mid\mid x_2 \mid\mid ... \mid\mid x_k$, consequentemente $|x_1| = |x_2| = ... = |x_k| = t - 1$, no qual $|x_k|$ é o tamanho dos blocos em que x é subdividida. Para tanto, é necessário que $|x| = n \geq m + t + 1$, que garante como antes dito, a propriedade da injeção.

Também observamos do algoritmo que $|x_k| = (t - 1 - d)$, no qual $0 \leq d \leq t - 2$. Com isso, podemos encontrar o valor de k, que representa o número de parte da cadeia n, ou no caso das funções de preenchimento da forma $k = [n/(t-1)]$. Denote $y(x) = y_1||y_2||...||y_{k+1}$ como a função de preenchimento. Pode-se observar que seguindo $X \mapsto Y$, que y_k é subcadeia de x_k , assim o bloco y_i, no qual $(0 = i = k)$ é de tamanho $t - 1$. E também, quando preciso, o tamanho de y_{k+1} deve ser completado com valores de preenchimento, até que $|y_{k+1}| = t - 1$.

A referência [20] faz uma demonstração do teorema que afirma que se a função de compressão f é resistente à colisão, então a função de resumo é resistente também à colisão. Isso é primeiramente demonstrado válido para $t > 2$, depois válido para $t > 1$. A consequência desse teorema é que basta procurar colisões nas funções de compressão, pois sendo essa segura, logo a função de resumo também é segura. Com base no tamanho do bloco poder ser $t > 1$ construiu-se um algoritmo que representa o método de Merkle Damgard da maneira apresentada no Algoritmo 2.

Outro teorema que Stinson demonstra em seu livro [20] é que sendo uma função de compressão f resistente à colisão, então a função de resumo também será resistente à colisão. No entanto, diferentemente do teorema anterior, esse teorema expande ainda mais o tamanho de bloco de mensagem, que pode ser qualquer valor de t.

Algoritmo 2 Merkle-Damgard

Interface: $compressao : \{0,1\}^{m+t} \rightarrow \{0,1\}^m$
$n \leftarrow |x|$
$y \leftarrow 11||f(x_1)||f(x_2)||\cdots||f(x_n)$
denote $y = y_1||y_2||\cdots||y_k$, no qual $y_i \in \{0,1\}, 1 \leq i \leq k$
$g_1 \leftarrow compressao(0^m||y_1)$
for $i \leftarrow 1$ **to** $k-1$ **do**
  $g_{i+1} \leftarrow compressao(g_1||y_{i+1})$
end for
return g_k

Vários algoritmos fortemente usados na critografia fizeram uso desse paradigma em sua construção. Antes do SHA-3 trazer a difusão do novo paradigma esponja, o paradigma adotado pela NIST e para compor o padrão de função de resumo criptográfico era o de Merkle-Damgard. Para entender o funcionamento do novo padrão é preciso compreender como esses algoritmos, anteriormente mais usados, como MD5, SHA-1 e SHA-2, construíram sua estrutura com o paradigma de Merkle Damgard e entender também quais mudanças em relação ao novo paradigma.

3.2.1 Função de resumo criptográfico MD5

O algoritmo MD5(*Message Digest 5*) [18] [17] foi construído por Ronald Rivest em 1991. Foi um algoritmo fortemente difundido e que implementou aplicações comuns para qualquer função de resumo criptográfico, como estruturas que necessitavam de geradores de números aleatórios, como código de integridade de mensagens, como parte de esquemas de autenticação e como integrante de protocolos de assinatura digital.

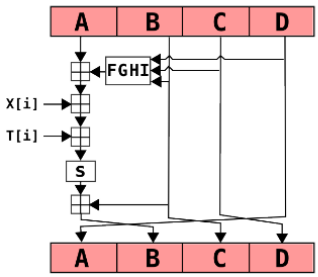

Figura 3.1: Função de rodada do algoritmo criptográfico MD5 [4]

Esse algoritmo faz uso de função de compressão que tem por entrada blocos de 512 *bits*, que são concatenados com um valor de 128 *bits*, gerando uma saída de também *128 bits*. O primeiro bloco é inicializado com um valor arbitrário inicial. Cada bloco é dividido em 16 palavras. Posteriormente, essas palavras são expandidas para 64 palavras. Para cada palavra é realizada uma rodada de processamento pelas funções de compressão. Nesse algoritmo são realizadas rotações, ou deslocamentos, somente na direção da direita para esquerda. Essas funções de compressão são compostas de operações lógicas, como mostra a Figura 3.1, tais como: OR exclusivos, AND e NOT *bit-a-bit*. O procedimento

de preenchimento funciona adicionando um *bit* 1 no final da mensagem e acrescentando *bits* 0 até que a mensagem tenha 32 *bits*.

Esse padrão de algoritmo foi amplamente utilizado. Muito embora não se deva mais usá-lo, encontra-se vestígios de implementações e tentativas de incrementos, para que se continuem em bom funcionamentos as aplicações que fizeram uso desse algoritmo. Vários pesquisadores tentam criar mecanismos nesses algoritmos, como aplicação desse algoritmo iteradas vezes e também a inclusão de um conjunto de bits aleatórios[1] à entrada para prevenir ataques de dicionários calculados *off-line*, mas ainda assim não se exclui a ameaça já demonstrada. Esse algoritmo foi amplamente atacado e seus ataques reconhecidamente tomam proveito de vulnerabilidades da construção do algoritmo, que mesmo com melhorias, não suavizam as colisões e os ataques de pré-imagem, deixando o algoritmo e seu uso suscetíveis, facilmente, a quebras [25] [21] [19].

3.2.2 Função de resumo criptográfico SHA-1

A função de resumo criptográfico SHA-1 (*Secure Hash Algorithm*) foi uma versão construída pelo NIST publicada como *Federal Information Processing Standard* em 1993 (FIPS PUB 180), e posteriormente revisado em outra versão (FIPS PUB 180-1) [14] em 1995 pelo NIST. Desde que foi publicado, tem sido usado em muitas aplicações. Foi desenvolvido como componente importante em aplicações comuns de função de resumo criptográfico em vários esquemas e protocolos criptográficos, autenticador de usuários, acordos de chaves e como gerador de números pseudoaleatórios. Seu uso foi tão difundido que se tornou o padrão como de fato adotado como função de resumo por algum tempo. Em consequência de sua grande utilização, tornou-se foco de várias análises.

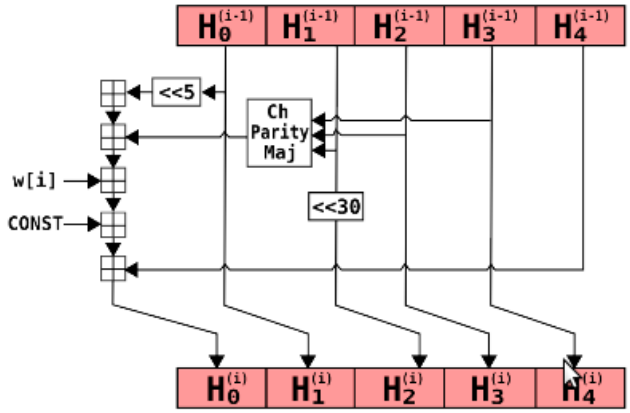

Figura 3.2: Função de rodada do algoritmo criptográfico SHA-1 [4]

Sua criação foi baseada no algoritmo de resumo criptográfico MD4 do pesquisador Ronald Rivest. A primeira versão lançada ficou conhecida como SHA-0, em razão de falhas terem sido logo descobertas. Depois de sua revisão, realizada pelo NIST, passou a se chamar SHA-1 (Figura 3.2). A mudança consiste de uma rotação na etapa inicial

[1] Do inglês "*salts*" e também utilizado em outras funções de resumo criptográfico, além do MD5.

do processamento da palavra que forma o bloco da mensagem. O SHA-1 é um algoritmo construído com a estrutura das funções de resumo iteradas. A função iterada que faz uso, é uma função implementada de acordo com o paradigma de Merkle-Damgard.

Essa função de resumo criptográfico aceita cadeias com o tamanho máximo de 2^{64} *bits*. A saída da função é cadeia de *bits* com comprimento fixo de 160 *bits*. A mensagem é fracionada em blocos de 512 *bits*. Esses blocos são processadas em uma estrutura constituída de 4 etapas com 20 passos (ou funções de rodada conforme demonstrado na Figura 3.2) cada uma delas. Essas etapas contém a função de compressão, que realiza seu processamento fazendo uso de palavras com 32 *bits*. As etapas também fazem uso de uma variável de estado de 160 *bits*, que por sua vez gera uma palavra de saída com também 160 *bits* e de uma técnica de preenchimento.

Os passos do algoritmo de SHA-1 correspondem [4] a:

1. Acrescentar com um *bit* "1" ao final da mensagem.
2. Acrescentar *bits* zero, a fim de que a mensagem seja proporcional ao tamanho dos blocos, o que corresponde em outras palavras, que seu tamanho seja congruente a 448 mod 512.
3. Concatenar com 64 *bits* que representam o tamanho da mensagem original.

Os passos seguintes consistem na aplicação o uso de funções de compressão, construídas como funções lógicas que correspondem ao método de Merkle Damgard:

- $Ch(x, y, z) = (x \wedge y) \vee (\sim (x) \wedge z)$, que é uma função não linear que retorna y se $x = 1$ e caso contrário retorna o valor de z.
- $P(x, y, z) = x \oplus y \oplus z$, uma função linear que realiza uma operação de XOR (ou-exclusivo) entre os elementos x, y e z.
- $M(x.y.z) = (x \wedge y) \vee (x \wedge z) \vee (z \wedge y)$, uma função não linear que tem como saída a sequência binária que mais ocorrer entre eles.

Cada bloco de 512 *bits* da mensagem é divido em 16 partes com 32 *bits* cada. Estas por sua vez, são expandidas até ocuparem espaço para 80 palavras. Em seguida, para cada palavra é realizada uma rodada. Por fim, o valor das rodadas é acrescentado no valor inicial de cada estado.

Como dito, o algoritmo de SHA-1 foi usado de forma intensa, bem como analisado fortemente. Porém em 2005, Wang et al. [23] [22] anunciaram que um ataque teórico sobre a resistência à colisão do algoritmo foi encontrado. Depois desse, que foi posteriormente aprimorado, o NIST passou a recomendar outro padrão para funções de resumo criptográfico, as funções SHA-2.

3.2.3 Função de resumo criptográfico SHA-2

Assim como a função SHA-1, as funções da família SHA-2 (*Secure Hash Algorithm*) também foram criadas pelo NIST como sendo uma publicação de *Federal Information Processing Standard* (FIPS 180-2 e FIPS 180-4) [16]. Mais que um algoritmo, o padrão SHA-2 é uma família de algoritmos. Essa família é composta das seguintes versões: SHA-224, SHA-256, SHA-384 e SHA-512. Esses números representam o tamanho das saídas

para cada um desses algoritmos, em *bits*. Os algoritmos de SHA-2, embora tenham-se recomendações para fazer uso desse padrão de funções, ainda não teve seu uso tão disseminado quanto o SHA-1.

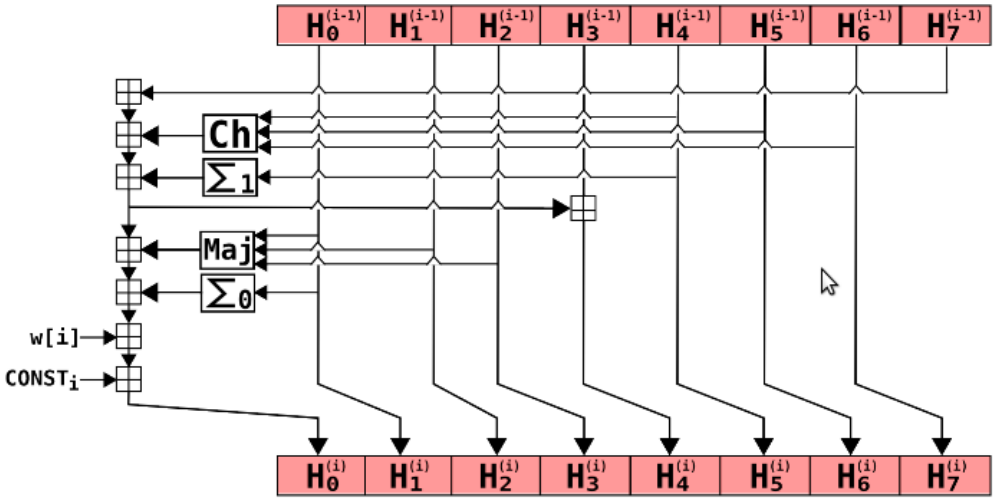

Figura 3.3: Função de rodada do algoritmo criptográfico SHA-2 [4]

O modelo dos algoritmos de SHA-2 segue a mesma estrutura adotada no algoritmo SHA-1. Da mesma forma, implementam funções de resumo iteradas, novamente seguindo o paradigma Merkle Damgard. Os valores de 224 e 384 correspondem aos valores de 256 e 512, sendo apenas um truncamento dos valores de saída. Por exemplo o SHA-256 processa, assim como no SHA-1, mensagens de valor máximo equivalente a 2^{64} *bits* e também faz uso de palavras com 32 *bits*.

A construção dos algoritmos de SHA-2 é muito semelhante à construção do algoritmo de SHA-1, como mostra a Figura 3.3. As diferenças consistem no número de blocos, no número de rodadas das funções de compressão, o uso de deslocamentos de *bits* para esquerda e para a direita e no tamanho da constante que define as mensagens de entrada e saída do algoritmo. Como dito, o SHA-256 processa mensagens com 2^{64} *bits* e trabalha com palavras de 32 *bits*. A entrada da função de compressão possui 512 *bits*. Sua variável de estado contém 256 *bits*, que por sua vez, gera outra variável com 256 *bits*. Essas características são idênticas no algoritmo SHA-512, com as peculiaridades de tamanho, tais quais: mensagem de 2^{128} *bits*, as palavras de 64 *bits* e a entrada da função consiste de um bloco com 1024 *bits*.

O preenchimento da mensagem no algoritmo de SHA-2 ocorre de forma análoga ao que acontece no algoritmo de SHA-1. Ou seja, sucintamente, acrescenta-se um *bit* 1, no final da mensagem, acrescenta-se 0 até que a palavra tenha o tamanho proporcional à quantidade de blocos, e por fim concatenar-se com *bits* para sinalizar o tamanho da mensagem ao último bloco. A computação do resumo inicia-se com a mensagem sendo dividida em blocos de 512 *bits* e a inicialização das variáveis de estado. Em seguida é realizada uma série de operações lógicas descritas por funções. Essas operações são deslocamentos, XOR(ou exclusivo), AND e NOT. Alguns desses algoritmos das família SHA-2 fazem uso das mesmas funções antes presentes na estrutura do algoritmo de SHA-

1. A mensagem de 512 *bits* é dividida em 16 palavras, cada uma com 32 *bits* é expandida para 64 palavras de 32 *bits*. Posteriormente, para cada palavra é realizada uma rodada do algoritmo, mas uma particularidade do SHA-2 é que cada rodada é composta por uma constante diferente. Por fim, as variáveis de estados geradas inicialmente são somadas módulo 2 (XOR) com os valores resultante das rodadas.

Em todos os algoritmos da família SHA-2, esses procedimentos são os mesmos, variando tão somente os tamanhos de 32 *bits* para 64 *bits* para o blocos de mensagem. Na família de algoritmo SHA-2, as operações são simples e possuem bom desempenho. Entretanto, o algoritmo é fortemente sequencial, não permitindo fácil paralelização. Cada rodada do algoritmo somente pode ser computada antes de cada palavra com seus então 32 *bits* tendo sido calculados anteriormente. Isso requer uma sequência para realizar a computação. O último bloco que contém o preenchimento para garantir que todos sejam múltiplos de 512 *bits* usa, como dito, as funções de preenchimento que hoje são implementadas em *software* que requerem custo mínimo de processamento.

Recentemente, o NIST mostrou preocupação com o uso prolongado do algoritmo SHA-2. Como sua estrutura é semelhante à construída no algoritmo de SHA-1, solicitou a busca por uma nova abordagem. Essas recomendações devem-se ao fato do SHA-1 ter sofrido ataques de colisão também observados em versões simplificadas do SHA-2. Esses receios se aplicam no tocante às colisões, sem excluir a possibilidade de ataques à resistência à primeira pré-imagem, como abordam Aoki et al [12] e Isobe et al. [11]. Não se conseguiu mostrar ataques que de fato ameaçassem o uso do SHA- 2. No entanto, como hoje seu funcionamento é fundamental para os protocolos de assinaturas, dentre outras formas de aplicação, se uma ameaça é efetivada, consequentemente coloca-se em risco a segurança de muitos sistemas. Em razão disso, o NIST, prevendo possíveis ataques, realizou um concurso que elegeu o SHA-3 para sobrepor o uso do SHA-1.

3.2.4 Concurso da NIST para escolha do SHA-3

No dia 2 de novembro de 2007, o NIST anunciou uma competição [1] para desenvolver um novo algoritmo de resumo criptográfico, chamado SHA-3, para obter um novo padrão. A competição contou com avançados meios de criptoánalise de algoritmos de resumo criptográfico. A chamada para o concurso foi realizada em uma *Federal Notice* [15] e solicitou a inscrição dos candidatos, com a submissão dos algoritmos para outubro do anos seguinte.

O NIST recebeu 64 propostas de criptólogos do mundo todo em 31 de outubro de 2008 e selecionou 51 candidatos em dezembro do mesmo ano para a quinta rodada. Já na segunda rodada a competição tinha 14 candidatos. Os cinco finalistas BLAKE, Gr⊘stl, JH, Keccak e Skein foram selecionados em 2010 para a última rodada.

Durante a competição, a comunidade de criptografia fornecido uma enorme quantidade de comentários. A maior parte desses comentários eram enviados para a NIST e para um fórum de resumo criptográfico público. Muitas dessas criptoanálises e estudos de desempenho foram publicados como artigos em conferências de criptografia e em revistas de criptografia. O NIST também sediou uma conferência com cada candidato finalista ao SHA-3 para comentários e conhecimento do público.

O NIST afirma que o concurso foi motivado pelos estudos de criptoanálise nos algoritmos MD5 e SHA-1, que comprovaram a viabilidade de busca de colisão nesses algoritmos.

Outros estudos recentes estariam agora encontrando ataques de pré-imagem e colisão [12] em versões reduzidas do SHA-2. Com as colisões dos outros dois algoritmos com falhas comprovadas e estudos que apontam indícios de possibilidade de falhas também no SHA-2, o NIST achou prudente, embora afirme nessa nota que não se tem motivos reais para afirmar o desuso do SHA-2, providenciar um concurso para obter um algoritmo que pudesse resguardar o uso das funções de resumo criptográfico e não deixar para que ocorra quebra do SHA-2 e a comunidade criptográfica fique sem solução em aplicações que requerem integridade e dependam da função de resumo para provê-la.

O NIST, em 2 de outubro de 2012, baseando-se nos comentários e revisões feitas sobre as submissões dos candidatos, escolheu o algoritmo Keccak para vir a ser o SHA-3, construído por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche.

Capítulo 4

Paradigma Esponja

As funções esponja são um método desenvolvido por Andrey Bogdanov, Miroslav Knezevic, Deniz Toz, Kerem Varıcı e Ingrid Verbauwhede [8]. As funções esponja são funções baseadas em um tamanho fixo de permutações e uma regra para preenchimento. Com isso, se constrói uma função que mapeia um tamanho variável de entrada em um tamanho variável de saída. Esta tem como entrada elementos do conjunto denotado por $f : \{0,1\}^n \mapsto \{0,1\}^n$, uma cadeia de *bits* de um tamanho qualquer e retorna uma cadeia de *bits* de tamanho determinado, estipulado pelo projetista do algoritmo.

As funções desse paradigma são generalizações das funções de resumo criptográficas. Elas possuem um tamanho de saída fixo e utilizam cifras de fluxo com um tamanho fixo de entrada. As cifras de fluxo cifram um texto *bit–a–bit*, que pode ser de qualquer tamanho. A cifração é construída por meio de uma chave de tamanho fixo e gerada de *bits* pseudoaleatórios. Em razão da criptoanálise, deseja-se que o período de fluxo dos *bits* pseudoaleatórios seja grande, no mínimo maior que a mensagem. Por isso, na utilização do paradigma esponja, limita-se o tamanho da mensagem, a fim de se evitar que essa seja maior que o período dos *bits* pseudoaleatórios usados para cifração [20].

Bertoni et al. [8] introduzem a função esponja ressaltando sua importância no papel de atender as premissas de segurança. Esse modo de função de resumo pode ser entendido como uma aproximação do modelo de oráculo aleatório. O modelo de oráculo aleatório é um modelo ideal para funções de resumo criptográfico, que retorna a cada consulta uma cadeia de *bits* verdadeiramente aleatória [20].

No modelo real, as funções de resumo criptográfico são usadas no lugar dos oráculos e, portanto, os algoritmos devem buscar se aproximar delas. Funções de resumo criptográfico operam usando memória finita. Isso permite que se crie colisões no estado interno, ou seja, diferentes entradas são mapeadas para um mesmo estado interno, consequentemente, para a mesma saída. Como no oráculo a memória é um recurso infinito, esse fato não ocorre. Com isso, nenhuma função real, com uso de memória finita, consegue ser de fato um oráculo aleatório [8].

Uma alternativa à construção do oráculo aleatório no paradigma de esponja são as permutaçôes escolhidas ocorrendo aleatoriamente, que chamamos de esponja aleatória. Isso aproxima as funções esponjas de oráculos aleatórios, exceto que as funções esponjas aleatórias possuem também recursos de memória finitos. Pode-se então manter uma analogia entre esponjas aleatórias com oráculos aleatórios, visto que os contextos são os mesmos.

Além de um quase oráculo, as funções esponjas usam o modelo conhecido por construção dupla, ou do inglês *duplex constuction* que pode ser utilizado para implementar um grande conjunto de funções de criptografia simétrica. Incluindo além de funções de resumo, outras possibilidades de uso para esse tipo de construção são os geradores de números aleatórios, no auxílio da derivação de chaves, em cifração autenticada e também em códigos de autenticação de mensagens. Nessas aplicações utilizam-se permutações de tamanho fixo em razão de sua necessidade e do uso memória finita, mas o método em si não estipula a necessidade de fixação para sua construção. As permutações são construídas, por sua vez, de maneira a otimizar o funcionamento da esponja e com a vantagem de não se ter escalonamento de chave [8].

O desenvolvimento dessas funções surgiu em janeiro de 2007 no seminário Dagstuhl com o paradigma utilizado na construção de função de resumo RadioGatún [7]. Esse método foi contruído para tamanho variável de entrada e um tamanho variável de saída. Então se depararam com a abordagem que expressa a fixação de valores sendo uma necessidade da área de segurança. Funções de resumo com tamanhos fixos são bons modelos, mas suas saídas são truncadas para conter o tamanho estipulado. Esse truncamento já acarreta na necessidade de serem resistentes aos ataques tradicionais como o ataque de aniversário e a resistência à segunda pré-imagem.

Primitivas como no RadioGatún [7] que possuem tamanhos varáveis de saída expressam a resistência necessária com seu respectivo tamanho de saída, agregando segurança indefinida, por poder retornar saídas tão longas quanto solicitado. Ao invés de exigir níveis de resistência contra ataques tradicionais às funções de resumo, se decidiu por expressar a exigência de segurança que uma função ideal pudesse alcançar. No RadioGatún [7] propuseram que se definisse algo que chamaram de um função de mistura[1] ideal, muito embora, depois da publicação, detectaram que a construção não era ideal e decidiram aprofundar mais o assunto. Pretendiam desenvolver uma função que se comportasse como um modelo de oráculo aleatório, com a existência de colisões internas. O resultados desses estudos foram as funções de esponjas aleatórias. Esses resultados foram apresentados no *Workshop* ECRYPT Hash em Barcelona [7].

4.1 Construção da esponja

A função esponja F é uma simples construção iterada com um tamanho variável de entrada e um tamanho arbitrário de saída baseado em permutações de tamanho fixo operando em um número fixo de b *bits*. Esse número b é denominado de largura. A função opera no estado de largura $b = r + c$ *bits*. O valor de r é chamado de taxa de *bits* e o valor c de capacidade.

Inicialmente, todos os *bits* do estado são inicializados com zero. A mensagem de entrada é preenchida e colocada em blocos de r *bits*. O processamento da esponja segue então duas fases: a de absorção[2], seguida pela fase de esmagamento[3] (Figura 4.1).

- **Fase de absorção**:a mensagem é quebrada em blocos e os r *bits* do bloco de entrada são submetidos a operações de XOR com os primeiros r *bits* de estado, intercalando

[1] Do inglês *"mangling"*
[2] Do inglês *"absorbing"*
[3] Do inglês *"squeezing"*

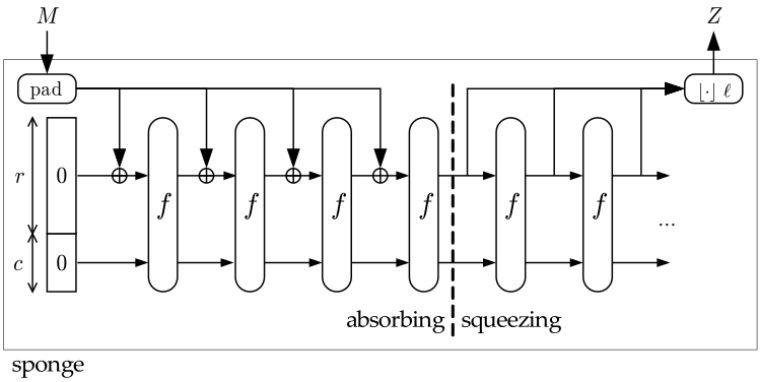

Figura 4.1: Construção Sponja [8]

com a aplicação da função f. Quando todos os blocos da mensagem forem processados, a construção esponja passa para a fase de esmagamento.

- **Fase de esmagamento**: os primeiros r *bits* de estado são retornados como blocos de saída, intercalando com a aplicação da função f até que se tenha a quantidade de *bits* estabelecida pela capacidade c definida pelo projetista e escolhida pelo usuário.

Os últimos c *bits* do estado nunca serão diretamente afetados pelos blocos de entrada e nunca são saída durante a fase de esmagamento. Quando o tamanho da entrada do bloco é escolhido para a função de permutação, o projetista tem liberdade em escolher os valores da taxa de bits r e da capacidade c. Quanto maior for o valor r, mais rápido será o processamento do algoritmo, pois processará maior quantidade de dados a cada iteração, e quanto maior for c, maior será o nível de segurança [8].

4.2 A segurança nas funções esponja

Pode-se enumerar uma lista de propriedades que uma função de resumo criptográfico deve resistir e assinalá-las a níveis de segurança, alegando que a segurança de um algoritmo consiste realmente na probabilidade de sucesso de resistir a um ataque. Um modelo que assinale todas as características pode servir para parâmetro de comparação, com a finalidade de testar se outros modelos compreendem essas propriedades ou parte delas. Em funções de resumo criptográfico, a resistência contra ataques é expressa em relação ao tamanho do resumo.

Até pouco tempo, se esperava que as funções de resumo fossem tão resistentes quanto os oráculos aleatórios e consideravam os resultados dos ataques nas funções e nos oráculos para avaliar a segurança dos modelos reais. Mas as publicações de ataques recentes mudaram essas comparações. Uma função iterada tem memória finita para armazenar seus estados e processar a entrada bloco por bloco. Em qualquer instante de tempo, o estado da função iterada comprime os blocos recebidos. Por ele ter um número finito de *bits*, a colisão pode ocorrer nesse estado [8]. Os modelos de oráculos aleatórios, por sua vez, não possuem colisão nesse estado. Essa é a principal razão pela qual os oráculos não podem ser usados para expressar segurança quando comparados às funções reais.

Por outro lado, funções esponja aleatória (denotadas por F) são uma alternativa para expressar as premissas de segurança. A esponja aleatória é uma instância da construção

de esponja com a função f, que corresponde às funções de permutação escolhidas aleatoriamente dentro do conjunto de permutações, mas com b *bits* para o tamanho dos blocos. Uma função esponja aleatória não é tão forte quanto um modelo de oráculo aleatório, exceto para os efeitos produzidos pela memória finita, que são as colisões nos estados internos. Uma esponja aleatória pode servir para expressar as premissas de segurança em cifras de fluxo e para funções de resumo criptográfico iteradas [10].

Quando usamos uma esponja aleatória como um modelo de segurança, considera-se o sucesso para um ataque particular. Essa probabilidade de sofrer esse ataque não depende apenas da natureza do ataque, mas também dos parâmetros escolhidos que formam a esponja aleatória, ou seja, sua capacidade, taxa de *bits* e da aleatoridade da escolha da transformação ou permutação. O modelo de esponja plana[1] é quando usamos uma simplificação da esponja que considera apenas o sucesso da probabilidade do pior caso, determinada pela impossibilidade de distinguir seus limites, que são determinados pela capacidade da esponja aleatória. Por isso, quando falamos em sucesso usamos um único parâmetro: o requisito que a capacidade estabelece. Ainda assim, é possível processar mensagens arbitrárias e finitas e gerar resumos de comprimento infinito, com a possibilidade de escolha para a capacidade.

4.3 A construção esponja como ferramenta

O objetivo inicial dos criadores da esponja [8] era definir uma referência para desenvolver propriedades de segurança para funções de resumo criptográfico. O desejo era criar uma construção de esponja que também pudesse compreender as funções de resumo criptográfico usadas na prática. Um importante aspecto desse desenvolvimento é que pode ser baseado em permutações, ao contrário das cifras de bloco ou funções de compressão. Construir uma transformação adequada é tão fácil quanto construir uma cifra de bloco ou uma função de compressão. Isso é por si uma boa notícia, pois todas as primitivas de criptografia simétrica são baseadas no tamanho fixado para as permutações. Uma permutação tem uma simples entrada, e portanto, trata todos os *bits* da mesma forma. Isto é desejável comparado aos modos utilizados em cifra de bloco ou em cifra de bloco ajustável[2].

Ataques genéricos exploram apenas propriedades de construção e não os ataques reais do paradigma. A propriedade de indistinguibilidade nos proporciona uma forma de limitante superior de probabilidade de sucesso para ataques genéricos e usamos desse fato para mostrar que as funções esponjas são efetivamente resistentes aos ataques genéricos que tenham uma complexidade maior que $2^{\frac{c}{2}}$. Na verdade, esses resultados demonstram que qualquer ataque contra uma função esponja implica que a utilização das permutações pode ser distinguida a partir de uma típica escolha aleatória das funções de permutação. Isso naturalmente leva à seguinte estratégia de construção, que chama-se de estratégia hermética[3]: adotar a construção esponja e construção de uma função f de permutação subjacente que não deve ter qualquer distinção estrutural.

[1] Do inglês *"flat sponge"*.
[2] Do inglês *"tweakable"*.
[3] Do inglês "*the hermetic sponge strategy*".

Nessa abordagem, sendo F uma função esponja, composta por funções de permutação f, no qual $b = r + c$ *bits* e corresponde ao comprimento do bloco a ser processado por essas funções f. Além disso, se faz uma afirmação na esponja plana sobre F que teria a capacidade igual à capacidade usada na construção da esponja chamada de requisito c. Isso equivale a dizer que o melhor ataque à função esponja seria um ataque genérico e esse ataque, quando feito sobre a F, tem a complexidade esperada abaixo de $2^{\frac{c}{2}}$. Com isso, a distinção natural entre as funções f e o projeto da permutação deve evitar que essas sejam diferenciadas. Na estratégia hermética das funções esponja, a capacidade determina o nível alegado para segurança e pode-se trocar a segurança requerida pela velocidade, aumentando a capacidade e diminuindo a conformidade da taxa de *bits*, e vice-versa.

Em modelos reais, o projetista pode escolher o comprimento do bloco b, fornecido como entrada do algoritmo. O valor da capacidade também deve ser fornecido como parâmentro de entrada pelo projetista.

4.4 Vantagens do uso do Paradigma Esponja

Com sua grande flexibilidade nos tamanhos de saída e entrada, a construção da esponja permite o desenvolvimento de várias primitivas, tais como função de resumo criptográfico, cifras de fluxo e MAC(Código de Autenticação de Mensagens[4]). Em algumas aplicações, a entrada é curta (por exemplo uma chave), enquanto em outras a entrada é longa (por exemplo, um fluxo de chaves). Em algumas aplicações a entrada é longa (por exemplo, uma mensagem para se calcular resumo criptográfico), mas outras aplicações exigem que a saída seja curta (por exemplo, uma chave ou um MAC).

Outro conjunto de vantagens da forma de uso é a vantagem da construção *duplex*, uma construção que está inteiramente relacionada com a construção da esponja, cuja a segurança pode ser demonstrada equivalente. A construção *duplex* permite que haja alternância de entrada e de saída de blocos na mesma taxa que a construção da esponja, como ocorre em uma comunicação *full-duplex*. Isso viabiliza implementar uma geração eficiente de uma sequência de *bits* pseudoaleatórios e um esquema de cifração autenticada, necessitando apenas de uma chamada para f por bloco de entrada.

4.5 Convenções e notações

Denota-se como o valor absoluto de um número real x por $|x|$. Também denota-se pela aproximação $log(1 + \varepsilon) \approx \varepsilon$ quando $\varepsilon \ll 1$. A cardinalidade de um conjunto S é representada por $|S|$.

Para o tamanho em *bits* da palavra M, denota-se por $|M|$. Essa palavra M pode ser considerada como uma sequência de blocos de um tamanho determinado x, no qual o último bloco pode ser mais curto. O número de blocos de M foi definido por $|M|_x$. Os blocos de M são identificados por $|M|_i$, no qual $0 \leq i \leq |M|_{x-1}$. A palavra nula ou cadeia vazia possui tamanho 0 e nenhum *bit*. A menos que explicitado, considera-se que no caso de cadeia vazia tem-se 0 blocos. Considera-se no truncamento da palavra M para l primeiros *bits* por $|M|_l$ [8]. Uma palavra que consista de n zeros é denotada por 0^n e

[4]Do inglês *"Message Authentication Code"*.

a concatenação de duas palavras M e N é denotada por $M||N$. Define-se também para se referir a todas as palavras, incluindo a palavra nula, por $\mathbb{Z}_2^*$. Quando excluir a cadeia vazia, por $\mathbb{Z}_2^+$, e a cadeia de tamanho infinito, por $\mathbb{Z}_2^\infty$.

O termo ataque genérico é frequentemente usado. Um ataque é considerado ataque genérico em funções esponja se não explora as propriedades específicas das funções f.

4.6 Regra de preenchimento

Para a regra de preenchimento usam-se as seguinte notações: o preenchimento da mensagem M para uma sequência de x *bits* por bloco é denotado por $M||pad[x](|M|)$. Essa notação ressalta que somente é considerado preenchimento o que é adicionado à palavra que é determinada pelo quantidade de *bits* de M e pelo tamanho x dos blocos. Pode-se omitir $[x]$, $(M||)$ ou ambas se eles foram valores claros dentro do contexto. Note que para algumas regras de preenchimento, algumas palavras não são possíveis [8].

A regra de preenchimento é compatível com esponja se nunca resulta em uma cadeia vazia e se satisfaz os seguintes critérios:

$$\forall n \geq 0, \forall M, M' \in \mathbb{Z}_2^* : M \neq M' \Rightarrow M||pad[r](|M|) \neq M'||pad[r](|M'|)||0^{nr}$$

.

Agora define-se a regra de preenchimento que é mais simples e compatível com o paradigma esponja . Preenchimento simples, denotado por pad10*, adiciona um simples *bit* 1 seguido de um número mínimo de *bits* zero. Os zeros são adicionados são até que o tamanho resultante seja múltiplo do tamanho do bloco. Preenchimento simples adiciona pelo menos um *bit* 1 e no máximo o número de *bits* de um bloco [8].

A regra de preenchimento mais simples e segura que permite usar a mesma função f com taxas diferentes é a seguinte: preenchimento multi-taxa, denotado por 10^*1, que adiciona um único *bit* 1 seguido por um número mínimo de *bits* 0, também seguido por um único *bit* 1 até que possuam um tamanho múltiplo do tamanho do bloco.

Claramente, esta regra de preenchimento é compatível com esponja, assim como ela tem a propriedade de ser injetora e não poder resultar em uma cadeia vazia ou uma cadeia com o último bloco todo zero. O preenchimento multi-taxa adiciona pelo menos 2 *bits* e, no máximo, o número de bits de bloco mais um.

4.7 As permutações na função esponja

A construção esponja é definda como uma função $esponja[f, pad, r]$ com domínio em $\mathbb{Z}_2^*$ e contradomínio em $\mathbb{Z}_2^\infty$, que usa um tamanho fixo de permutação ou transformação f, uma regra de preenchimento compatível com a esponja, que se chama *"pad"* e uma taxa de *bits* r.

O tamanho finito da saída pode ser obtido truncando para os l primeiros *bits*. Chama-se uma instância da construção esponja de função esponja.

A transformação ou permutação f opera sobre uma taxa fixa de *bits*, a largura b, que constitui o estado da construção com b *bits*. No primeiro estado, todos os *bits* são

inicializados com zero. A mensagem de entrada é preenchida e partida em r *bits* por bloco e então é processada em duas fases: a fase de absorção e a fase de esmagamento.

Na fase de absorção, os primeiros r *bits* do estado R e o restante dos $r - b$ *bits* são processados de maneira diferente. O tamanho restante $b - r$ do estado interior é chamado de capacidade.

Na fase de esmagamento, a parte exterior do estado é iterativamente retornada com blocos de saída, intercalados com aplicações da função f. O número de iterações é determinado pelo número estipulado de *bits* l.

A saída é truncada para os l primeiros *bits*. Os c *bits* do estado interno, não são diretamente afetados nas fases de absorção e esmagamento. A capacidade c de fato determina o alcance do nível de segurança da construção.

Nos primeiros artigos de Bertoni et al. [8] de funções esponja, os autores trataram de forma mais geral a parte externa, os blocos de mensagens como sendo elementos arbitrários dentro de um grupo e elementos da parte interna como conjunto arbitrário. Devido à sua relevância prática, abandonamos essa representação genérica para o caso mais específico, no qual o estado é uma seqüência binária de um determinado comprimento b e os blocos de mensagem são cadeias r *bits*.

Assim como a construção esponja, a construção $duplex[f, pad, r]$ usa tamanho fixo nas transformações ou permutações f, regra de preenchimento e as taxas de *bit* r. Ao contrário das funções esponjas que são definidas a cada chamada, a construção *duplex* resulta em um objeto que recebe uma cadeia de entrada e resulta numa cadeia de saída que depende de todas as entradas recebidas até o momento. Chama-se uma instância da construção *duplex* de objeto *duplex*, denotado por D, no qual uma chamada feita a um objeto específico *duplex* D é feita pelo seu nome D e um ponto.

Algoritmo 3 Construção Esponja

Requisitos:$r < b$
Interface:$Z = esponja(M, l)$, com $M \in \mathbb{Z}_2^*$, no qual $l > 0$ e $Z \in \mathbb{Z}_2^l$
$P = M||pad[r](|M|)$
$s = 0^b$
for $i \leftarrow 1$ **to** $|P|_{r-1}$ **do**
  $s = s \oplus (P_i||0^{b-r})$
  $s = f(s)$
end for
$Z = [s]_r$
while $|Z|_r r < l$ **do**
  $s = f(s)$
  $Z = Z||[s]_r$
end while
return $[Z]^l$

Após recebimento da $D.duplexing(\sigma, l)$ chama-se o objeto *duplex* preenchido, a cadeia de entrada σ e a parte exterior do estado submetida a operações de XOR. Então se aplica a função f no estado e retorna os l primeiros *bits* da parte externa do estado de saída. Denota-se a chamada quando σ é cadeia vazia, pelo termo chamada em "branco" e uma chamada com $l = 0$, ou seja, sem saída, de chamada muda.

Algoritmo 4 Construção Duplex

Requisitos:$r < b$
Requisitos:$\rho_{max}(pad, r) > 0$
Interface:$D.inicio()$
$s = 0^b$
Interface:$D.duplexing(\sigma, l)$, com $l \leq r, \sigma \in \cup_{n=0}^{\rho_{max}(pad,r)} \mathbb{Z}_2^n$ e $Z \in \mathbb{Z}_2^l$
$P = \sigma||pad[r](\sigma)$
$s = s \oplus (P_i||0^{b-r})$
$s = f(s)$
return $[s]_l$

A primeira função auxilar é a função de absorção $absorcao[f, r]$. Ela recebe uma cadeia P de tamanho $|P|$ e multiplo de r e retorna um valor obtido depois de absorver P.

Algoritmo 5 Função de Absorção

requisitos:$r < b$
Interface:$s = absorcao(P)$, com $P \in \mathbb{Z}_2^*$ e $s \in \mathbb{Z}_2^b$
$s = 0^b$
for $i \leftarrow 0$ **to** $|P|_{r-1}$ **do**
  $s = s \oplus (P_i||0^{b-r})$
end for
return s

Chama-se P de caminho para o estado s se $s = absorcao(P)$. Claramente $absorcao$ (cadeira vazia) $= 0^b$. Em geral o bloco na posição j que foi processado por uma função esponja de entrada M é igual:

$$Z_j = absorcao(P||0^{rj}), j \geq 0,$$

com

$$P = M||pad[r](|M|).$$

Uma função de esmagamento é como uma dupla forma de absorver. A função de esmagamento é denotada por $esmagamento[f, r]$. Para um determinado estado s $esmagamento(s, l)$ aponta o resultado truncado em l *bits*.

Algoritmo 6 Função de Esmagamento

Requisitos:$r < b$
Interface:$Z = esmagamento(P)$, com $s \in \mathbb{Z}_2^b$ e $Z \in \mathbb{Z}_2^l$
$Z = [s]_r$
while $Z_r, r > l$ **do**
  $s = f(s)$
  $Z = Z||[s]_r$
end while
return $[Z]_l$

A construção da esponja pode ser definida como a aplicação posterior de uma regra de preenchimento, uma função de absorção e uma função de esmagamento. Para $Z = esponja[f, pad, r](M, l)$, temos:

$$P = M||pad[r](|M|).$$

$$s = absorb[f, r](P).$$

$$Z = esmagamento[f, r](s, l).$$

Depois de analisar, nesse capítulo, as propriedades do paradigma esponja, serão analisados as propriedades das funções de permutação usadas no $Keccak$.

Capítulo 5

Função Keccak

A função Keccak, pronunciada como "Ketchac" é uma instância do paradigma de funções esponjas. Essa função usa como construção permutações de blocos de tamanho fixo definidas por 7 permutações. Nesse capítulo se faz um estudo de como foi realizada a definição dessa função, como se comporta na qualidade de função esponja, notações e conceitos importantes.

5.1 As permutações $Keccak - f$

Existem 7 permutações $Keccak - f$, indicadas por $Keccak - f[b]$, no qual $b = 25 \times 2^l$. O l da fórmula que define b, representa os *bits* da *string* s e pode assumir valores entre 0 e 6. $Keccak - f[b]$ é uma permutação em Z_2^b, no qual seus *bits* são enumerados de 0 até $b - 1$. Com isso o valor de b representa a extensão das permutações.

As funções de permutações $Keccak - f[b]$ são uma série de operações em um estado representado por um array tridimensional de elementos de GF(2), nomeado de $a[5][5][w]$, com $w = 2^l$. A expressão $a[x][y][z]$, com $x, y \in Z_5$ e $z \in Z_w$, denota o *bit* na posição (x, y, z). O primeiro índice é o zero. O mapeamento entre os *bits* de s e o estado de a é $s[w(5y + x) + z] = a[x][y][z]$. A expressão envolvendo as coordenadas x e y deve ser módulo 5 e a coordenada z deve ser módulo w. Algumas vezes omite-se o índice $[z]$, também o índice $[y][z]$ ou todos os três índices, significando que é válido para todos os valores de índices omitidos (Figura 5.5).

$Keccak-f[b]$ é uma permutação iterada, que consiste de uma sequência de n_r rodadas R, indicadas por i_r de 0 até $n_r - 1$. Uma rodada consiste em:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta.$$

- $\theta : a[x][y][z] \leftarrow a[x][y][z] + \sum_{y'=0}^{4} a[x-1][y'][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1]$,
- $\rho : a[x][y][z] \leftarrow a[x][y][z - (t+1)(t+2)/2]$,
  com $0 \leq t \leq 24$ e $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ em que $GF(5)^{2x2}$,ou $t = -1$ se $x = y = 0$,
- $\pi : a[x][y] \leftarrow a[x'][y']$, com $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$,

- $\chi : a[x] \leftarrow a[x] + (a[x+1] + 1)a[x+2]$,
- $\iota : a \leftarrow a + RC[i_r]$.

As adições e multiplicações são sobre os termos de GF(2). Os valores de $RC[i_r]$ são constantes e dados por $RC[i_r][0][0][2^j - 1] = rc[j + 7i_r]$ para todo $0 \leq j \leq l$. Os demais valores de $RC[i_r][x][y][z]$ são zero. Os valores de $rc[t] \in$ GF(2) são definidos como saída binária de um registrador de deslocamento com retroalimentação linear(LFSR)[1]. Assim,

$$rc[t] = (x^t \; mod \; x^8 \; + \; x^6 \; + \; x^5 \; + \; x^4 \; + \; 1) \; mod \; x \in GF(2)[x].$$

O número de rodadas n_r é determinado pelo alcance das permutações, como sendo $n_r = 12 + 2l$.

5.2 A construção esponja

O paradigma esponja constrói uma função $esponja[f, pad, r]$ para tamanho variável de entrada e uma saída de tamanho arbitrário, usando uma permutação f de tamanho fixo. Também faz uso de uma regra de preenchimento "*pad*" e um parâmetro para taxa de *bits* r. A função de permutação f opera sobre um número fixo de *bits*, com largura b. Chamamos de capacidade o valor $c = b - r$. Para a regra de preenchimento, usa-se a seguinte notação: o preenchimento da mensagem M para uma sequência de x *bits* por bloco sendo $M||pad[x](|M|)$, no qual $|M|$ é o tamanho da mensagem em *bits*.

O estado inicial tem um valor 0^b, chamado de estado raiz. O estado raiz é fixo e não deve ser considerado entrada. Esse fator é importante na segurança da construção esponja.

A função $Keccak[r, c]$ foi definida [6] pela aplicação da construção esponja exatamente como explicitado no Algoritmo 3, no capítulo 4, com $Keccak - f[r + c]$, preenchimento em multi-taxas e a taxa de *bits* r.

$$Keccak[r, c] \equiv esponja[Keccak - f[r + c], \; pad10^*1, \; r],$$

em que $r = 1600 - c$ e $c = 576$

Esta especificação $Keccak[r, c]$ para alguma combinação de $r > 0$ e c de modo que $r+c$ é a largura suportada pela função de permutação $Keccak - f$. O guia de referência [6] do algoritmo afirma que o valor padrão de r é $1600 - c$ e também cita que o valor padrão de c é 576 para o nível de segurança.

5.3 Nível de segurança para a função esponja $Keccak$

Para cada valor de parâmetro suportado, Bertoni et al. [6] definiram um plano de requisitos para a esponja. A probabilidade de sucesso esperado de algum ataque contra $Keccak[r, c]$ com um esforço equivalente a N chamadas de $Keccak - f[r + c]$ ou seu inverso deve ser inferior ou igual ao de uma construção de um oráculo aleatório. Essa probabilidade de sucesso é dada por:

[1] Do inglês "*Linear Feedback Shift Register*".

$$1 - exp(-N(N+1)2^{-(c+1)}).$$

Excluíram-se deficiências devido a mero fato que $Keccak - f[r + c]$ pode ser descrita compactamente e pode ser eficientemente executada. Ele cita a implementação impossível do oráculo aleatório como exemplo, o que tornou complexo o entendimento.

Note que a capacidade requisitada é a mesma da capacidade usada pela construção esponja.

5.4 Partes do estado na $Keccak$

Para auxiliar na análise ou descrição das propriedades da $Keccak - f$ os autores da referência [6] defiram nomes das partes do estado da $Keccak - f$.

Na primeira dimensão as partes são:

- Uma **linha** é definida com 5 *bits* com coordenadas y e z constantes.
- Uma **coluna** é definida com 5 *bits* com coordenadas x e z constantes.
- Uma **fileira** é definida com w *bits* com coordenadas x e y constantes.

Na segunda dimensão as partes são:

- Uma **camada** é definida com $5w$ *bits* com cordenada x constante.
- Um **plano** é definida com $5w$ *bits* com cordenada y constante.
- Uma **fatia** é definida com $25w$ *bits* com cordenada z constante.

5.5 As estapas de mapeamento da $Keccak - f$

Uma rodada é composta por uma sequência de mapeamentos, cada um deles executando uma tarefa em particular. As estapas são compostas de uma descrição simples para uma especificação que não pode esconder falhas [6].

Mapeando as fileiras do estado, por exemplo, os *subarrays* da primeira dimensão na direção do eixo z, já resultam em uma simples e eficiente implementação de *software* para os passos de mapeamento. Inicia-se a descrição de cada um dos passos de mapeamentos por um pseudo código, no qual as variáveis $a[x, y]$ representam os valores anteriores da fileira e $A[x, y]$ os novo valores. As operações nas fileiras são limitadas em operações lógicas *booleanas bit-a-bit* e rotações na matriz de estados. No pseudo código da referência denotou-se por $ROT(a, d)$ uma versão de a sobre d *bits*, cujo *bit* na posição z é mapeado para posição $z + d \ mod \ w$ com as instruções de rotação. Caso contrário, o número de deslocamentos e de operações *bit-a-bit* deve ser combinada ou intercalação de *bit* pode ser aplicada.

Algoritmo 7 O mapeamento χ

for $y = 0$ **to** 4 **do**
  for $x = 0$ **to** 4 **do**
    $A[x, y] = a[x, y] \oplus ((NOT\ a[x + 1, y])\ AND\ a[x + 2, y])$
  end for
end for

5.5.1 Propriedades do mapeamento χ

O mapeamento χ, conforme mostra a Figura 5.1, é o único mapeamento não-linear na $Keccak - f$. Sem ela a $Keccak - f$ seria linear.O mapeamento χ é invariante à translação em todas as direções e tem grau algébrico 2 [6]. Isto tem consequências para propagação diferencial e propriedade de correlação.

O mapeamento χ é inversível, mas forma sua forma inversa é diferente da χ natural. Por exemplo, não possui grau 2. O mapeamento χ é complemento simplificado de uma função chamada γ, que é usado no RadioGatún [7]. Esta foi escolhida por ser uma simples propagação não-linear, ser uma simples expressão algébrica e por conter poucas operações: um XOR, um NOT e um AND para operação por *bit* no estado.

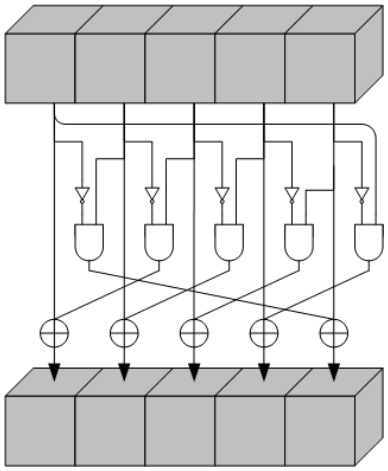

Figura 5.1: Mapeamento χ [6]

5.5.2 Propriedades do mapeamento θ

O mapeamento θ é linear, tem objetivo de difusão e é invariante à translação em todas as direções [6], conforme mostra a Figura 5.2. Este efeito pode ser descrito como: a adição de cada *bit* $a[x][y][z]$ em soma *bit-a-bit* das partes de duas colunas, da coluna $a[x - 1][\cdot][z]$ e da coluna $a[x + 1][\cdot][z - 1]$. Sem θ o mapeamento e a função de rodada $Keccak - f$ não poderia ter difusão significante. O mapeamento θ tem um número de ramificações 4, mas em média provê alto nível de difusão. De fato, θ foi escolhida por ter alta média de difusão e possuir poucas operações: dois XORs por *bit*. Graças à iteração com χ, cada *bit* da entrada afeta 31 *bits* de saída e cada *bit* de saída de uma rodada depende de 31 *bits* da entrada. Note que, sem a versão de duas camadas, esta parte seria apenas 25 *bits*.

Algoritmo 8 O mapeamento θ

comentário: C e D são matrizes temporárias.
for $x = 0$ **to** 4 **do**
  $C[x] = a[x, 0]$
  for $y = 1$ **to** 4 **do**
    $C[x] = C[x] \oplus a[x, y]$
  end for
end for
for $x = 0$ **to** 4 **do**
  $D[x] = C[x - 1] \oplus ROT(C[x + 1], 1)$
  for $y = 0$ **to** 4 **do**
    $A[x, y] = a[x, y] \oplus D[x]$
  end for
end for

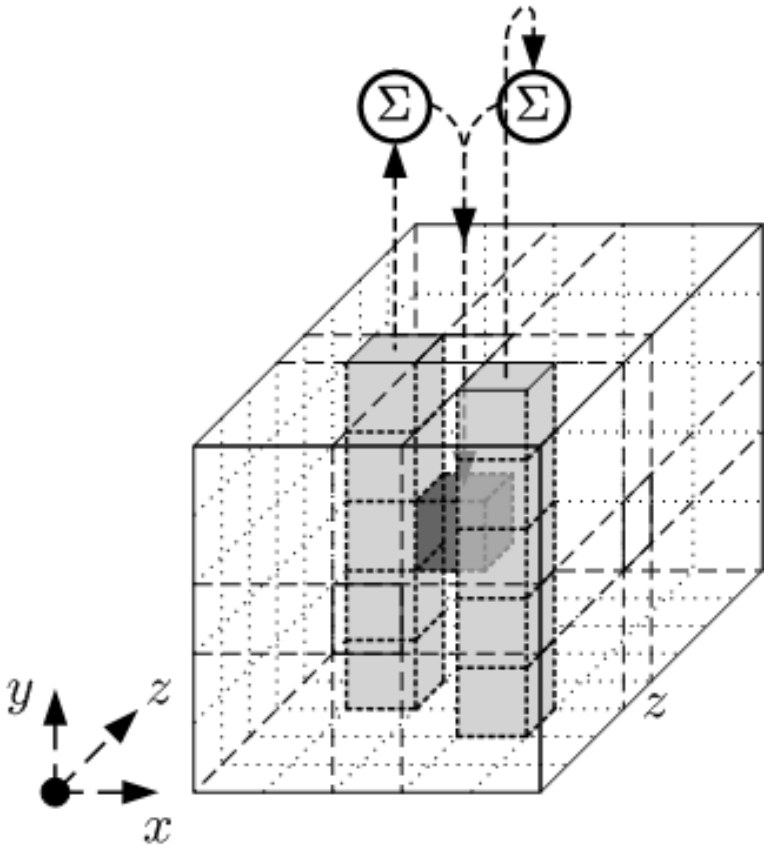

Figura 5.2: Mapeamento θ [6]

5.5.3 Propriedade do mapeamento π

O mapeamento π é uma transposição de fileiras que proporciona dispersão ao longo da difusão. Sem isto, $Keccak - f$ poderia exibir períodos de curta extensão. Operações de π são uma forma de transformação linear(x,y): a fileira na posição (x, y) vai para a

Algoritmo 9 O mapeamento π

for $x = 0$ **to** 4 **do**
  for $y = 0$ **to** 4 **do**
    $\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
    $A[X, Y] = a[x, y]$
  end for
end for

posição $(x, y)M^t$ com $M = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$, uma matriz de 2 por 2 com elementos em GF(5). O mapeamento π é uma função linear e é uma transposição de matrizes baseada no fato que $M^T = M^{-1}$. Logo, é ortogonal e tal que $M \cdots M^T = I$ [6], conforme mostra a Figura 5.3.

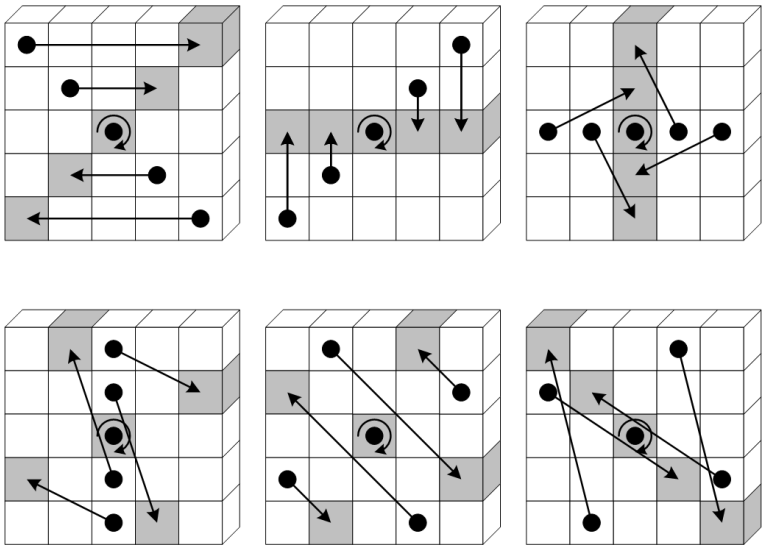

Figura 5.3: Mapeamento π [6]

5.5.4 Propriedades do mapeamento ρ

Algoritmo 10 O mapeamento ρ

$A[0, 0] = a[0, 0]$
$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
for $t = 0$ **to** 23 **do**
  $A[x, y] = ROT(a[x, y], (t + 1)(t + 2)/2)$
  $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
end for

O mapeamento ρ consiste na translação dentro das fileiras com objetivo de prover dispersão entre as camadas. Sem isto, difusão entre fileiras seria muito lenta. Esta é

translação que é invariante no eixo z. O inverso da ρ é definido por translações de fileiras, cujas constantes são as mesmas, mas nas direções inversas, conforme mostra a Figura 5.4.

Como as funções ρ são lineares e se baseiam no fato que ρ é a transposição de um *bit* e usa uma matriz ortogonal no qual $M^t M = I$.

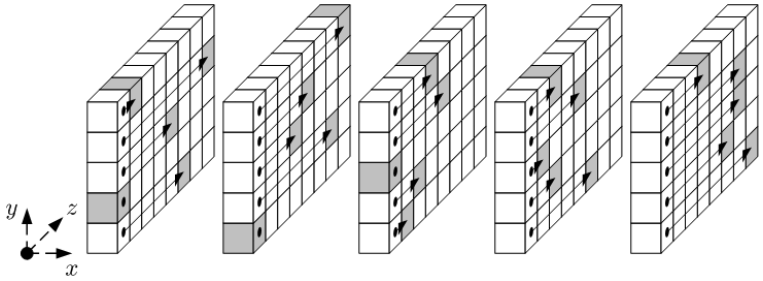

Figura 5.4: Mapeamento ρ [6]

5.5.5 Propriedades do mapeamento ι

O mapeamento ι consiste da adição de constantes de rodada e o objetivo é distribuir simetricamente as camadas. Sem isso, a função de rodada poderia ser invariável a translação na direção z e todas as rodadas seriam iguais na $Keccak-f$, permitindo um ataque focado nessa simetria. O número da posição do *bit* ativo das constantes de rodada, por exemplo, as posições dos *bits* que na constantes de rodada for diferente de zero, é $l+1$. Como l aumenta, as constantes de rodada são acrescidas e de maneira assimétrica.

Os *bits* das constantes de rodada são diferentes de rodada para rodada e retomam como saída o tamanho máximo LFSR [6]. As constantes somente são acrescidas em uma fileira simples do estado. Por conta disso, a distribuição difusa entre θ e χ para todas as fileiras do estado é simples em todas as rodadas. Em *software*, isso representa uma instrução de XOR *bit-a-bit* simples.
[2]

[2]Do inglês "*Lane, row, column, slice, sheet, plane*" para camada, fatia, linha, coluna, fileira e plano

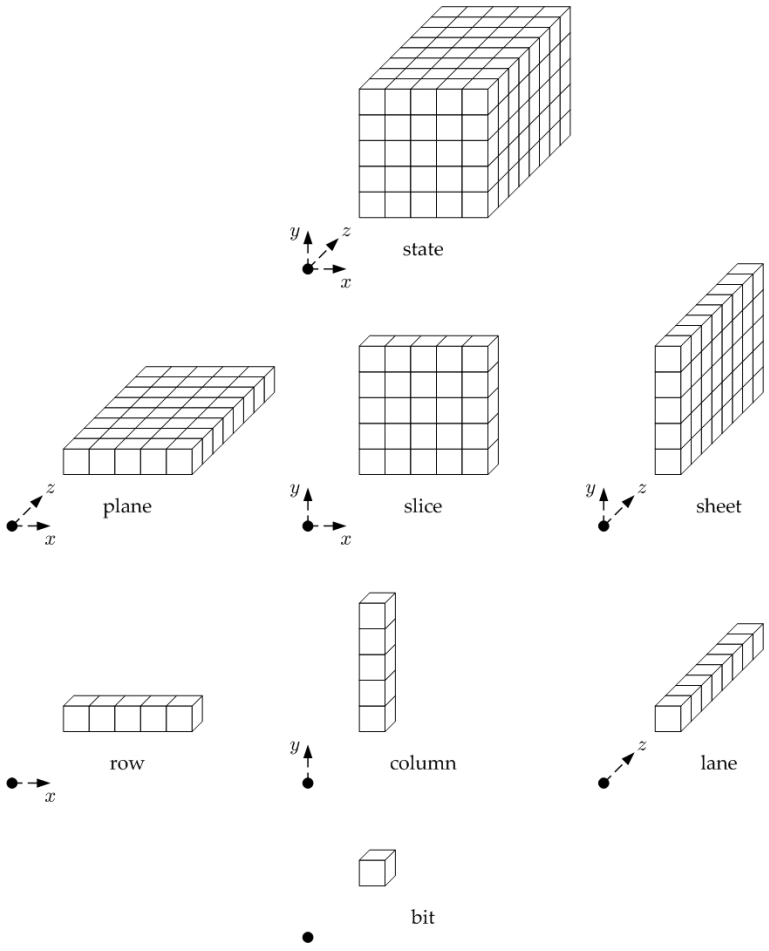

Figura 5.5: Partes do Estado [6]

Capítulo 6

A implementação da função $Keccak$

Neste capítulo, mostra-se a implementação do algoritmo da função esponja $Keccak-f$. Como seus elementos e sua descrição se relacionam e estruturam a construção esponja. E ainda, como foi projetada implementação dos mapeamentos para suas funções de permutação, segundo proposto pelo guia de referência [6].

6.1 Inicialização e preparação das estruturas do algoritmo

Como descrito nos Capítulos 4 e 5 as funções esponja contam com variáveis de entrada de tamanhos arbitrários que no $Keccak$ são fixos para que se tornem possíveis e compor uma implementação. O primeiro elemento necessário é o paramêtro l. Esse parâmetro define o tamanho do bloco. No nosso algoritmo fixamos o $l = 6$ e por consequência o tamanho do bloco é de 1600. Isso se deve pelo tamanho do bloco ter a relação com o parâmetro l estabelecida por $b = 25$ x 2^l.

Obtido o tamanho do bloco, passamos aos tratamentos da mensagem para se adequar a esse valor. A mensagem, aqui denotada de M, possui um tamanho $|M|$ estabelecido pelo usuário do algortimo. No caso, essa mensagem é passada ou por arquivo texto ou digitada pelo usuário no arquivo "`stdin`" (entrada padrão). Cada caractere da mensagem recebeu o tratamento para convertê-lo para binário, e em seguida, cada um desses *bits* foi armazenado em um tipo *byte* que na linguagem especificada pode ser obtida por um vetor de oito posições de *unsigned char*.

Cada *byte* contendo um *bit* dos caracteres da mensagem M foi armazenado em uma posição de um outro vetor, também de *bytes*, no entanto com tamanho do bloco. Esse vetor preenche seus índices com os *bytes* da mensagem alocados até que completasse o tamanho do bloco ou até que os caracteres da mensagem sejam esgotados. Caso sobrassem caracteres esses seriam armazenados em outro vetor semelhante e da mesma forma que esse havia sido tratado. Na insuficiência de *bytes* fornecidos pelos caracteres da mensagem, passa-se para aplicação da regra de preenchimento.

Em comparação ao referenciado pelo guia, até este ponto tem-se as definições da mensagem M, de seu tamanho $|M|$, o tamanho do bloco fixado e por contas inferidas desses valores quantos são os blocos necessários e a exigência ou não da aplicação da regra de preenchimento.

6.1.1 Regra de preenchimento

A regra de preenchimento foi avaliada pelo tamanho da mensagem M fornecida pelo usuário. Uma vez que o tamanho dessa mensagem em *bytes* é menor que o tamanho do bloco, o vetor que armazena o bloco tem suas posições preenchidas com os *byte* de M até que não hajam mais caracteres. Depois envia-se esse vetor para o trecho que pega as posições restantes do vetor e completa-se com preenchimento. Esse bloco agora deve conter a mensagem cujo tamanho é inferior ao tamanho do bloco seguida do preenchimento de 10^*1 e ainda o tamanho da mensagem. Esse bloco contém então o que na referência é denotado por $M||pad[x](|M|)$, que se chama preenchimento em multi taxa e o seu tamanho resultante é necessáriamente multiplo do tamanho do bloco.

6.2 As permutações $Keccak - f$

Como descrito anteriormente, são 5 as permutações de $Keccak - f$ (Figura 6.1), indicados pelo $Keccak - f[b]$, para $l = \{0, 1, 2, 3, 4, 5, 6\}$, no qual $b = 25$ x 2^l o que retorna como possibilidades de tamanhos de $b = \{25, 50, 100, 200, 400, 800, 1600\}$. As permutações são todas dentro do espaco de Z_2^b, no qual todo os *bits* da mensagem, chamados no guia de s, são enumerados, correspondendo no código à posição que ocupam no vetor que armazena os *bytes* da mensagem.

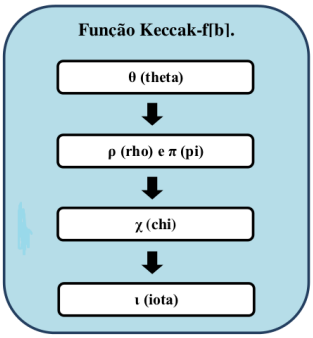

Figura 6.1: Função de Resumo Criptográfico SHA-3 [24]

Nessa implementação fixou-se o $l = 6$ e o tamanho do bloco em 1600.

As funções de permutações $Keccak - f[b]$ são resultado de várias operações em um estado a que é representado por uma matriz tridimensional segundo a expressão $a[x][y][z]$, com $x, y \in Z_5$ e $z \in Z_w$. Para saber quais posições dessa matriz os *bytes* da mensagem ocupariam, obteve-se o bloco armazenado no vetor seguindo $s[w(5y + x) + z]$, sempre conservando que os índices x, y são módulos 5 e o índice z é tomado módulo w.

De posse da matriz tridimensional que armazena os *bits* de M distribuídos pela expressão anterior, essa matriz passa a ser manipulada pelas funções de permutação. Essas operações são realizadas repetidas vezes, cujo número de repetições é definido pela fórmula $n_r = 12$ x $2l$, o que para nossos valores de l *e* b são 24 rodadas.

A primeira função de permutação implementada foi a θ. Nela a matriz resultante do bloco tem suas posições alteradas em torno de todos os eixos x,y e z. Para isso se fez dois XOR para usar as posições atuais e acrescentar duas vezes mudanças nos índices.

Na primeira mudança dos índices as linhas x são giradas uma posição a menos para cada valor da coluna y. Na segunda mudança de índice agora é deslocado uma posição da linha x para frente para cada coluna y da matriz, e ainda, rotacionadas para trás uma posição as camadas, ou eixo z. Na implementação, são operações algébricas de somas e subtrações nos índices e em seguida reduzidas módulo 5 para respeitar o escopo. Os *bits* resultantes das mudanças de índices são submetidos a operação de XOR com o estado anterior na matriz. O *bit* resultante do XOR passa a ocupar aquela posição de $a[x][y][z]$.

```
/*teta */
for (x = 0; x < 5; x++) {
  for (y = 0; y < 5; y++) {
     for (z = 0; z < w; z++) {
       for (i = 0; i < 5; i++) {
        aux2 = 0;
        aux3 = 0;
        if (x - 1 < 0) {
          aux2 = 5 + ((x - 1) % 5);
        } else {
          aux2 = (x - 1) % 5;
        }
        if (z - 1 < 0) {
          aux3 = w + (z - 1) % w;
        } else {
          aux3 = (z - 1) % w;
        }
        a[x][y][z] = a[x][y][z] ^ a[aux2][i][z]
        ^ a[(x +1) % 5][i][aux3];

      }
    }
  }
}
```

A segunda operação de $Keccak-f$ foi a ρ. Esta função translada todos os eixos x, y e z. Para alterar as camadas(eixo z) criam-se 24 constantes, resultantes da multiplicação de matrizes. Para obter essas constantes multiplicou-se a matriz $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$ por si mesma t vezes, no qual $0 \leq t \leq 23$ e o resultado ainda é multiplicado por $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Para ganho de desempenho da implementação, essa operação é pré-calculada e realizada uma única vez. As constantes são guardadas após cálculo em uma matriz de declaração global. Essas constantes são valores duplos para cada índice z. Por isso essa matriz global tem o formato matrizCTEs[24][2], assim para cada indíce a posicão matrizCTEs[t][0] armazena valores de x e a matrizCTEs[t][1] os valores de y. Esses valores são operados módulo 5 e passam a ser os novos índices para linha e coluna. No eixo z, as operações são realizadas deslocando o índíce z da forma $z - (t+1)(t+2)/2$. O *bit* resultante passa a ocupar a matriz $a[x][y][z]$.

```
/*Rho*/
for (x = 0; x < 5; x++) {
  for (y = 0; y < 5; y++) {
    for (z = 0; z < w; z++) {
```

```
      for (t = 0; t < 24; t++) {
        aux2 = 0;
        x_linha = matrizCTEs[t][0];
        y_linha = matrizCTEs[t][1];
        int interm = t;
        if (((z - ((interm + 1) *
        (interm + 2) / 2)) % w) < 0) {
          aux2 = w + ((z - ((interm + 1)
          * (interm + 2) / 2)) % w);
        } else {
        aux2 = (z - ((interm + 1)
        * (interm + 2) / 2) % w);
      }
      a[x][y][z] = a[x_linha][y_linha][aux2];
      interm = 0;
     }
    }
  }
}
```

O mapeamento que ocorre em seguida é π, no qual somente linha e coluna são alteradas. Para obter os novos índices de linha e coluna se multiplica o inverso da matriz $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$ por si mesma t, que também foi pré-calulada e tratada para ser módulo 5, resultando na matriz $\begin{pmatrix} 1 & 3 \\ 1 & 0 \end{pmatrix}$ que foi multiplicada pela matriz de índices anteriores $\begin{pmatrix} x \\ y \end{pmatrix}$. O *bit* resultante passa a ocupar o índice da matriz $a[x][y]$.

```
/*pi */
  for (x = 0; x < 5; x++) {
    for (y = 0; y < 5; y++) {
      for (z = 0; z < w; z++) {
        //multiplicando pela inversa de ((0 1)(2 3))
        //que    ((-3/2 1/2)(1 0))
        matrizResult[0][0] = ((1 * x) + (3 * y)) % 5;
        matrizResult[1][0] = ((1 * x) + (0 * y)) % 5;
        if (matrizResult[0][0] < 0) {
          x_1 = 5 + matrizResult[0][0];
        } else {
          x_1 = matrizResult[0][0];
        }
        if (matrizResult[1][0] < 0) {
          y_1 = 5 + matrizResult[1][0];
        } else {
          y_1 = matrizResult[1][0];
        }
        a[x][y][z] = a[x_1][y_1][z];
      }
    }
  }
```

O mapeamento realizado em seguida foi χ, no qual somente são alterados os valores das linhas ou eixo x. Para obter no novo indíce se faz um XOR da linha atual com a linha atual operada algebricamente da forma $(a[x+1]+1)a[x+2]$. Lembrando que essa

operação é a única não-linear do $Keccak - f$. No código a implementação consistiu de um XOR que foi realizada entre o *bit* no índice anterior. Antes ainda do XOR, o indíce anterior realiza uma operação de AND com o índice da linha anterior duas posições acima e nega esse resultado com a operação de NOT. O *bit* resultando passa a ocupar a posição $a[x][y][z]$ da matriz .

```
/*chi */
for (x = 0; x < 5; x++) {
  for (y = 0; y < 5; y++) {
    for (z = 0; z < w; z++) {
       a[x][y][z] = a[x][y][z] ^ (~(a[(x + 1) % 5][y][z] ^ 1))
          & (a[(x + 2) % 5][y][z]);
    }
  }
}
```

O último mapeamento realizado foi ι, no qual se desloca a matriz inteira de acordo com uma constantes derivadas do número da rodada. Para encontrar essas constantes precisamos de outras matriz. Ela em teoria teria 4 dimensões, mas tem 0 em todas as posições, exceto nas posições $constantesRodadaIota[i_r][0][0][2^j - 1c]$. Essas posições, que não são nulas e encontradas por $rc[j + 7i_r]$ para $0 \leq j \leq l$. O valor de $j + 7i_r$ é passado para um LFSR definido por $(x^t \ mod x^8 + x^6 + x^5 + x^4 + x^0) \ mod \ x$ que no código corresponde a um deslocamento desse valor de x por esses expoentes. Essa constante é número binário que é operado com XOR.

```
/*iota */
for (z = 0; z < w; z++) {
  a[0][0][z] ^= constantesRodadaIota[aux][z];
}
```

6.3 A construção esponja

A construção esponja, como dito anteriormente, possui duas fases: a de arbsoção e a de esmagamento. Os bits são zerados no primeiro estado. Nos demais estados, coleta-se os dados da mensagem, obtidos na forma binária, em blocos de r *bits*, que são sumetidos à operação lógica XOR com os *bits* do estado inicial.

Posteriomente, se inicia a fase de absorção, que consiste de realizar operações lógicas de XOR nos r bits de um estado com os r *bits* do estado anterior, intercalando a aplicação das funções de permutação. A fase de absorção se dá até que todos os blocos de mensagens sejam processados.

```
//absorb
for (i = 0; i < r; i++) {
  state[i] ^= block[i];
}
keccak(state, l);
```

Em seguida, inicia-se a fase de esmagamento que consiste de realizar chamadas das funções de permutação $Keccak - f$ até que seja possível realizar o truncamento para c

bits que correspondem ao nível de segunraça escolhido, que no caso dessa implementação correspondeu a 256.

```
//squeezing
if(c > r){
  keccak(state, l);
}else{
  keccak(state, l);
  for(i = 0; i < c; i++){
    out[i] = state[i];
  }
}
```

O resultado do processamento desse algoritmo é a saída da fase de esmagamento que é uma cadeia de caracteres binários que são também mostrados em hexadecimal. Esse valor corresponde ao resumo obtido pela mensagem submetida à função

Capítulo 7

Conclusão

Com esse estudo se concluiu do ponto de vista teórico que o NIST trouxe o algoritmo de SHA-3 como alternativa ao uso de função de resumo criptográfico e que esse algoritmo usa o paradigma esponja, que possui propriedades que embora garantam as características de função de resumo, não demandam grandes cálculos matemáticos para que seja viável. Esse algoritmo é bem versátil visto que as operações que compõe um mapeamento são operações lógicas, sem grandes complexidades, mas que são suficientes para embaralhar os estados que guardam os blocos de mensagens e garantir as difusões e distribuições necessárias.

Para consolidar um entendimento conceitual foi necessário os conceitos e definições que formam uma função uma função de resumo criptográfico. Entender suas propriedades e como essas atuam no provimento da segurança no algoritmo. Buscou-se entender o paradigma e o funcionamento superficial das funções que são ou foram padrões de funções de resumo. Depois de entender o funcionamento, de modo geral do paradigma Merkle-Damgard e de funções que o implementaram, entendeu-se as razões que levaram a buscar outro algoritmo.

Uma vez que entendemos o funcionamento e as definições passamos para a compreensão da construção do algoritmo que emprega o paradigma de função esponja. Depois de entender a construção do *Keccak* e como esse emprega o paradigma esponja, iniciou-se a implementação algorítmica. Na implementação realizada buscou-se uma abordagem didática, sem objetivar elementos de desempenho e priorizar o entendimento das estruturas e como essas empregavam os conceitos inferidos. Além de entender o relacionamento da estrutura conceitual e a abordagem prática, conseguiu-se compreender o comportamento ao se processar e vizualizar como promovia o que era sugerido pelas referências.

Essa abordagem didática, além de ferramenta de ensino, pode ser uma abordagem mais simples para entendimento inicial de futuros estudiosos, visto que a implementação sugerida pela referência possui uma leitura complexa e uma maior preocupação com desempenho. Facilitando o entendimento e o estudo pode-se expandir o objetivo desse trabalho para melhorias no desempenho. Fazer uma criptoanálise mais profunda das propriedades dos mapeamentos e confrontar com os resultados publicados.

Esse estudo e os futuros podem assim difundir o uso desse algoritmo, acelerar a publicação de referência da NIST com RFC, e também trazer mais possibilidades para aplicações que dependem de funções de resumo criptográficos.

Referências

[1] NIST, SHA-3 Competition(2007 - 2012). 17

[2] Pedro Antônio Dourado de Rezende. 5: Algoritmos criptográficos. 8

[3] W.R. Cheswick, S.M. Bellovin, and A.D. Rubin. *Firewalls e Seguranca na Internet*. Bookman Companhia ED. 8

[4] Eduardo de Figueredo Oliveira Thomaz. *Implementação em software de algoritmo de resumo criptográfico*. PhD thesis, Instituto de Computação, Unicamp, Campinas, SP, Brazil, 2011. vii, 5, 6, 8, 13, 14, 15, 16

[5] Desirré Bueno do Nascimento. Proposta de uma Arquitetura de Referência para o Algoritmo Keccak, novembro 2011. 2, 3

[6] Michaël Peeters e Gilles Van Assche Guido Bertoni, Joan Daemen. The Keccak reference. vii, 29, 30, 31, 32, 33, 34, 35, 36

[7] Michaël Peeters e Guilles Van Assche Guido Bertoni, Joan Daemen. RadioGat¸, a belt-and-mill hash functions. neokeon, 2006. 20, 31

[8] Michaël Peeters e Guilles Van Assche Guido Bertoni, Joan Daemen. Cryptographic sponges functions. neokeon, 2011. vii, 19, 20, 21, 22, 23, 24, 25

[9] Maria Helena S. S. Bizelli. Funções inversas. 2

[10] Paulo Henrique Nobrega Tavares. *Estudo e Implementação de algoritmos de resumos(hash) criptográfico na plataforma Intel Scale*. PhD thesis, Instituto de Computação, Unicamp, Campinas, SP, Brazil, 2006. 1, 6, 22

[11] Kyoji Isobe, Takanori Shibutani. Preimage Attacks on Reduced Tiger and sha-2. Lecture Notes in Computer Science, pages 139–155. Springer, 2009. 17

[12] Yu Sasaki Krystian Matusiewicz Jian Guo e Kazumaro Aoki Mitsuru Matsui, Lei Wang. *Preimages for Step-Reduced SHA-2*, volume 5912. Advances in Cryptology – ASIACRYPT 2009, 2009. 17, 18

[13] NIST. Cryptographic hash Algorithm Competition,2005. 3

[14] NIST. Security Hash Standard - FIPS PUB 180-1, April 1995. 14

[15] NIST. Federal Register Notice, November 2007. 17

[16] NIST. Security Hash Standard - FIPS PUB 180-4, March 2012. 15

[17] R.L. Rivest. RFC1321 - The MD5 Message Digest algorithm, April 1992. 10, 13

[18] L. Chen S. Tuner. RFC6151 - Update Security Considerations for the MD5 Message Digest, march 2011. 13

[19] Marc Stevens. Fast Collision Attack on MD5. In *Dempartament of Mathematics and Computer Science, Endhoven University of Technology*. 14

[20] Douglas R. Stinson. *Cryptography Theory and Practice*. Chapman & Hall/CRC, Waterloo, 2005. 1, 2, 11, 12, 19

[21] Xuejia Lai e Hongbo Yo Xiaoyun Wang, Dengguo Feng. Collisions for hash functions MD4, MD5, HAVAL-128 e RIPEMD. In *Cryptology ePrint Archive*. ePrint, 2004. 14

[22] Hongbo Yu Xiauyun Wang, Yiaqun Lisa Yin. Collision Search Attacks on SHA-1. 15

[23] Hongbo Yu Xiauyun Wang, Yiaqun Lisa Yin. Finding Collision in The Full SHA-1. 15

[24] Fenando Yokota Marques. Desenvolvimentom e análise de um dos algoritmos SHA-3 finalistas em smart cards. vii, 3, 6, 37

[25] Noboru Kinuhiro e Kazuo Ohata Yu Sasaki, Yusure Naito. Improved Collision Attack on MD5. In *Cryptology ePrint Archive*. ePrint, 2005. 14