

Função de Hash Criptográfica SHA-3

Ranieri Althoff

¹Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Segurança em Computação

1. Introdução

Uma função *hash* é uma função que aceita um bloco de dados de tamanho variável como entrada e produz um valor de tamanho fixo como saída, chamado de valor de *hash*. Esta função tem a forma:

$$h = H(M)$$

Onde:

- h é o valor hash de tamanho fixo gerado pela função hash.
- H é função hash que gerou o valor h .
- M é o valor de entrada de tamanho variável.

Espera-se que uma função *hash* produza valores h que são uniformemente distribuídos no contra-domínio e que são aparentemente aleatórios, ou seja, a mudança de apenas um *bit* em M causará uma mudança do valor h . Por esta característica, as funções *hash* são muito utilizadas para verificar se um determinado bloco de dados foi indevidamente alterado.

As funções *hash* apropriadas para o uso em segurança de computadores são chamadas de “função *hash* criptográfica”. Este tipo de função *hash* é implementada por um algoritmo que torna inviável computacionalmente encontrar:

- um valor M dado um determinado valor h : $M | H(M) = h$
- dois valores M_1 e M_2 que resultem no mesmo valor: $(M_1, M_2) | H(M_1) = H(M_2)$

Os principais casos de uso de funções *hash* criptográficas são:

- Autenticação de Mensagens: é um serviço de segurança onde é possível verificar que uma mensagem não foi alterada durante sua transmissão e que é proveniente do devido remetente.
- Assinatura Digital: é um serviço de segurança que permite a uma entidade assinar digitalmente um documento ou mensagem.
- Arquivo de Senhas de Uma Via: é uma forma de armazenar senhas usando o valor *hash* da senha, permitindo sua posterior verificação sem a necessidade de armazenar a senha em claro, cifrá-la ou decifrá-la.
- Detecção de Perpetração ou Infecção de Sistemas: é um serviço de segurança em que é possível determinar se arquivos de um sistema foram alterados por terceiros sem a autorização dos usuários do sistema.

1.1. Propriedades

Como observado na seção anterior, uma função *hash* criptográfica precisa ter certas propriedades para permitir seu uso em segurança de computadores. Nas seções a seguir estão destacadas algumas dessas propriedades.

Antes, define-se dois termos usados a seguir:

- Pré-Imagem: um valor M do domínio de uma função *hash* dada pela fórmula $h = H(M)$ é denominado de “pré-imagem” do valor h .
- Colisão: para cada valor h de tamanho n *bits* existe necessariamente mais de uma pré-imagem correspondente de tamanho m *bits* se $m > n$, ou seja, existe uma “colisão”.

O número de pré-imagens de m *bits* para cada valor h de n *bits* é calculado pela fórmula $2^{m/n}$. Se permitirmos um tamanho em *bits* arbitrariamente longo para as pré-imagens, isto aumentará ainda mais a probabilidade de colisão durante o uso de uma função *hash*. Entretanto, os riscos de segurança são minimizados se a função de *hash* criptográfica oferecer as propriedades descritas nas próximas seções.

1.1.1. Resistente a Pré-Imagem

Uma função *hash* criptográfica é resistente a pré-imagem quando esta é uma função de uma via. Ou seja, embora seja computacionalmente fácil gerar um valor h a partir de uma pré-imagem M usando a função de *hash*, é computacionalmente inviável gerar uma pré-imagem a partir do valor h .

Se uma função *hash* não for resistente à pré-imagem, é possível atacar uma mensagem autenticada M_1 para descobrir o valor secreto S usado na mensagem, permitindo assim ao perpetrante enviar uma outra mensagem M_2 ao destinatário no lugar do remetente sem que o destinatário perceba a violação da comunicação. O ataque ocorre da seguinte forma:

- O perpetrante tem conhecimento do algoritmo de *hash* usado na comunicação entre as partes.
- Ao escutar a comunicação, o perpetrante descobre qual é a mensagem M e o valor de *hash* h .
- Visto que a inversão da função de *hash* é computacionalmente fácil, o perpetrante calcula $H^{-1}(h)$.
- Como $H^{-1}(h) = S||M$, o perpetrante descobre S .

Desta forma, o perpetrante pode utilizar a chave secreta S no envio de uma mensagem M_2 para o destinatário sem que este perceba a violação.

1.1.2. Resistente a Segunda Pré-Imagem

Uma função *hash* criptográfica é resistente a segunda pré-imagem quando esta função torna inviável computacionalmente encontrar uma pré-imagem alternativa que gera o mesmo valor h da primeira pré-imagem.

Se uma função de *hash* não for resistente a segunda pré-imagem, um perpetrante conseguirá substituir uma mensagem que utiliza um determinado valor de *hash*, mesmo que a função de *hash* seja de uma via, ou seja, resistente a pré-imagem.

1.1.3. Resistente a Colisão

Uma função *hash* criptográfica é resistente a colisão quando esta tornar inviável computacionalmente encontrar duas pré-imagens quaisquer que possuam o mesmo valor de *hash*. Neste caso, diferentemente da resistência a segunda pré-imagem, não é dado uma pré-imagem inicial para a qual precisa se achar uma segunda pré-imagem, mas é suficiente encontrar duas pré-imagens quaisquer tal que $H(M_1) = H(M_2)$.

Quando uma função *hash* é resistente a colisão, está é consequente resistente a segunda pré-imagem. Porém, nem sempre uma função resistente a segunda pré-imagem será resistente a colisão. Por isto, diz-se que uma função *hash* resistente a colisão é uma função de *hash* forte.

Se uma função *hash* não for resistente a colisão, então é possível para uma parte forjar a assinatura de outra parte. Por exemplo, se Alice deseja que Bob assine um documento dizendo que deve 100 reais a ela, caso Alice saiba que um documento contendo o valor de 1000 reais contém o mesmo valor de *hash* que o documento original, Alice pode fazer com que Bob seja responsável por uma dívida maior que a original, pois a assinatura valerá para ambos os documentos.

1.1.4. Uso das Propriedades de Funções *Hash*

Abaixo, temos uma tabela que mostra quais propriedades das funções *hash* são necessárias para alguma das aplicações de segurança de computadores:

Aplicação	Resistente a Pré-Imagem	Resistente a Segunda Pré-Imagem	Resistente a Colisão
Autenticação de Mensagens	X	X	X
Assinatura Digital	X	X	X
Infecção de Sistemas		X	
Arquivo de Senhas de Uma Via	X		

No caso da infecção de sistemas, não há problema em usar uma função de *hash* com fácil inversão, pois não é necessário embutir um valor secreto na geração do valor de *hash* de um arquivo. Já, num arquivo de *hash* de senhas, a inversão permitiria descobrir a senha a partir do valor de *hash*.

Se a função de *hash*, porém, permitir o descobrimento de uma segunda pré-imagem, seria possível infectar um arquivo de um sistema sem detecção, pois seu valor de *hash* não mudaria. Isto não seria um problema para um arquivo de *hash* de senhas, pois o perpetrante não possui a senha, que é a primeira pré-imagem e, portanto, não teria condições de descobrir a segunda pré-imagem.

2. O algoritmo SHA-3

O **SHA-3** é um algoritmo de *hash* que foi escolhido em uma competição do **NIST**, o instituto de padrões e normas dos Estados Unidos, para criar uma função mais segura

que as anteriores. Como os padrões MD5 e SHA-0 haviam sido quebrados e o SHA-1 já possuía ataques teóricos, e com o fato de que o SHA-2 era bastante semelhante ao SHA-1 e possíveis ataques poderiam enfraquecer ambos os algoritmos, o NIST buscou um algoritmo com uma estrutura diferente que pudesse resistir e substituí-los nesse caso.

O algoritmo escolhido como SHA-3 é baseado na família de funções esponja **Keccak**, mais especificamente na variante com 1600 bits de largura da função de permutação [Dworkin 2015].

O Keccak segue a estrutura dos algoritmos de *hash* comuns, onde os blocos P da mensagem a ser cifrada vão sendo concatenados alternadamente com aplicações de uma função de transformação f , de forma que a passagem i tenha o seguinte formato:

$$S_i = f(S_{i-1} \oplus P_i)$$

Adicionalmente, explorando essa forma genérica de algoritmos de hash e de uma função f específica, o Keccak permite que tanto sua entrada quanto sua saída tenham um tamanho variável, e possa ser usado não somente para verificar a integridade de uma mensagem, mas como um gerador de números pseudoaleatórios, além de outras aplicações.

2.1. Funções esponja

Funções esponja são uma classe de funções que recebem uma entrada de tamanho finito qualquer e produzem uma saída com outro tamanho qualquer desejado, sendo definidas por três parâmetros: um estado S , que contém b bits; uma função f que permuta ou transforma o estado S ; e uma função de *padding* P [Bertoni et al. 2011].

Na inicialização, a função P é aplicada na entrada M e dividida em blocos de r bits. Os b bits do estado S são zerados. A construção da esponja se dá em duas fases, chamadas de absorção e compressão.

O tamanho r também é chamado de *bitrate*, porque representa a quantidade de bits da entrada que são consumidos em cada iteração da função esponja, e o tamanho $c = |S| - r$ é chamado de capacidade, e representa o nível de segurança atingido pela variante da função - SHA-3, $r + c = 1600$.

Aumentar o tamanho de r para tornar o algoritmo mais rápido (gerando o *hash* com menos iterações) diminui, portanto, o nível de segurança do *hash* gerado, porém um nível muito alto de segurança implica numa quantidade muito alta de iterações para gerar o *hash*.

2.2. Absorção e compressão

Na fase de absorção, cada bloco P de r bits é combinado com os primeiros r bits, com os restantes c bits preenchidos com zero, do estado S com xor e é aplicada a função f no resultado, ou seja, $S_i = f(S_{i-1} \oplus P_i || 0^c)$.

Ao final de todas as iterações, ou seja, após a mensagem M ser completamente consumida pela absorção da esponja, a saída da função f será o *hash* gerado.

Na fase de compressão, os primeiros r bits do estado S são retornados, e caso mais bits sejam desejados, se aplica novamente a função f em S para transformar o estado. A representação gráfica dessas fases pode ser vista na figura 1.

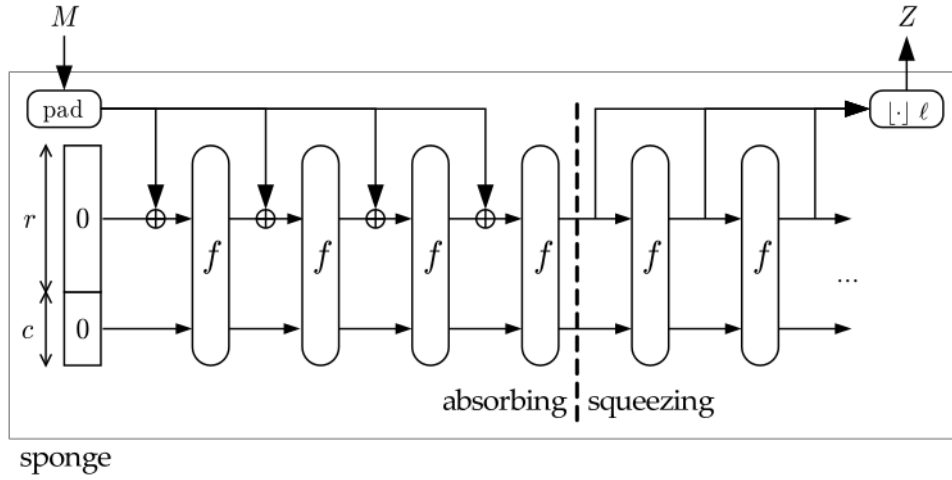


Figura 1. Diagrama de uma função esponja

Os últimos c bits não são diretamente afetados pela entrada M e não são produzidos como saída da função. A função esponja produz uma quantidade indefinida de bits aleatórios.

2.3. Função Keccak

A função de compressão Keccak (f , não confundir com a compressão da esponja) é uma função que tem como entrada um valor S de b bits, de $r + c = 1600$ bits no SHA-3. Esse valor S é então dividido em uma matriz A 5×5 , onde cada célula contém 64 bits. As posições dessa matriz possuem índices x (linha), y (coluna) e z (célula).

Uma vez criado esse estado interno, a função aplica 24 rodadas de processamento R_i que levam em consideração o resultado da rodada anterior e uma constante RC_i

Cada uma das rodadas R consiste na composição de cinco funções e pode ser expressa pela seguinte função:

$$R_i = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

As funções possuem a seguinte fórmula:

- $\theta(M[x, y, z]) = M[x, y, z] \oplus \sum_{y'=0}^4 M[x-1, y', z] \oplus \sum_{y'=0}^4 M[x+1, y', z-1]$

Este passo é uma função de substituição que utiliza bits de células adjacentes, cada bit dependente de outros 11, o que provê boa difusão, o que é importante para o efeito avalanche.

- $\rho(M[x, y, z]) = \begin{cases} M[x, y, z], & \text{se } x = y = 0 \\ M[x, y, z - \frac{(t+1)(t+2)}{2}] & | 0 \leq t < 24 \wedge \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ em } GF(5) \end{cases}$

Função de espalhamento dos bits de uma célula para auxiliar na difusão de forma mais rápida.

- $\pi(M[x, y]) = M[x', y'], | \begin{pmatrix} x \\ y \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix}$

Outra função de espalhamento dos bits, mas entre células diferentes.

- $\chi(M[x, y, z]) = M[x, y, z] \oplus (\neg M[x + 1, y, z] \wedge M[x + 2, y, z])$
Esta função é uma substituição baseada no valor atual do bit e dos próximos 2 bits, tornando o Keccak uma função não-linear, característica que os outros passos não provêm.
- $\iota(M[x, y, z]) = M[x, y, z] \oplus RC_i$, onde RC_i é a constante da rodada citada acima e diferente para cada rodada i .
Função de substituição baseada em uma tabela e que envolve uma constante que é diferente a cada passo da função aplicado.

2.4. Parâmetros do SHA-3

O SHA-3 define um algoritmo padrão para uso com parâmetros diferentes, dependendo do nível de segurança e tamanho de *bits* desejados na saída. A tabela abaixo enumera os parâmetros normalmente utilizados com o SHA-3:

Tamanho do valor de <i>hash</i>	Tamanho do bloco <i>r</i>	Capacidade <i>c</i>	Resistência a colisão	Resistência a segunda pré-imagem
224	1152	448	2^{112}	2^{224}
256	1088	512	2^{128}	2^{256}
384	832	768	2^{192}	2^{384}
512	576	1024	2^{256}	2^{512}

Como visto nas seções anteriores, quanto maior o tamanho do bloco (ou *bitrate*), maior a vazão de *bits*, porém menor a segurança do SHA-3. Isto é evidente nos valores de resistência a colisão e à segunda pré-imagem, que mostram que quanto menor é o *bitrate*, maior é a resistência. Observe também que, para os tamanhos de valor de *hash* da tabela acima, não há necessidade de se usar a fase de espremer a esponja do algoritmo do SHA-3, pois é sempre menor que nestes casos.

3. Questionário

3.a. O que é e para que serve o *state array*?

O *state array* do Keccak é uma representação do estado do algoritmo e uma matriz de três dimensões, de tamanho $5 \times 5 \times w$, onde w é o comprimento do bloco a ser cifrado. Na maioria das linguagens de programação, é uma estrutura muito simples de ser representada, mas tem um uso bastante complexo no Keccak.

As partes bidimensionais dessa matriz são chamadas de *plane* (plano), *slice* (fatia) e *sheet* (folha), e as unidimensionais são chamadas de *row* (linha), *column* (coluna) e *lane* (pista), conforme mostrado na figura 2. Um bit é indexado por sua linha, coluna e pista.

O uso do *state array* permite que as funções internas sejam definidas em termos de posições nessa matriz de forma prática. O estado guarda valores parciais do *hash* e todas as funções internas recebem como parâmetro o estado atual e retornam como saída o estado atualizado.

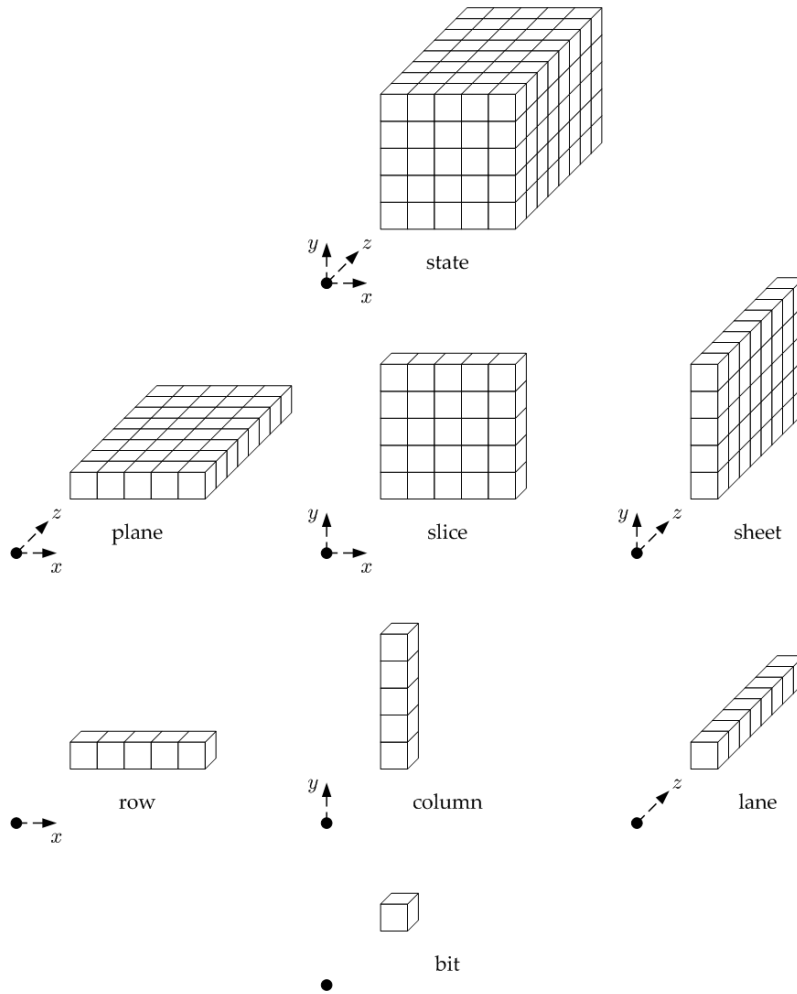


Figura 2. Diagrama do *state array*

3.b. Como é feita a conversão de *strings* para *state array*?

Seja S uma *string* de b *bits* que representa o estado da permutação Keccak- $p[b, n_r]$. O *state array* A é definido da seguinte forma no SHA-3, usando $b = 1600$ e $w = 64$:

Para toda tripla (x, y, z) tal que $0 \leq x < 5$, $0 \leq y < 5$ e $0 \leq z < w$,

$$A[x, y, z] = S[w \cdot 5y + w \cdot x + z] = S[w \cdot (5y + x) + z]$$

Por exemplo, a posição $A[2, 1, 4]$ é obtida do *bit* $S[64 \cdot (5 \cdot 1 + 2) + 4] = S[772]$ da *string* de entrada.

Usando essa fórmula, os *bits* são mapeados sequencialmente por pista, coluna e linha, ou seja, os primeiros 64 *bits* serão mapeados na pista da coluna 0 e linha 0, os próximos serão mapeados na pista da coluna 1 e linha 0, e assim sucessivamente.

3.c. Como é feita a conversão de *state array* para *strings*?

A conversão inversa, ou seja, de *state array* para *string*, é feita de forma análoga, concatenando todas as pistas por coluna e então por linha. Em passos, primeiro se concatenam os *bits* de uma pista:

$$Lane(i, j) = A[i, j, 0] \cdot A[i, j, 1] \cdot \dots \cdot A[i, j, w - 1]$$

Por exemplo, para a linha 0 e coluna 0, e usando $w = 64$:

$$Lane(0, 0) = A[0, 0, 0] \cdot A[0, 0, 1] \cdot \dots \cdot A[0, 0, 63]$$

$$Lane(1, 0) = A[1, 0, 0] \cdot A[1, 0, 1] \cdot \dots \cdot A[1, 0, 63]$$

E assim sucessivamente para todas as pistas. Cada plano é a concatenação de todas as pistas na sua coluna, indexadas por $0 \leq j < 5$:

$$Plane(j) = Lane(0, j) \cdot Lane(1, j) \cdot \dots \cdot Lane(4, j)$$

3.d. Explicar os cinco passos de mapeamento (*step mappings*)

3.e. Explicar a permutação Keccak-p[b,nr].

3.f. Descrever o *framework sponge construction*

3.g. Explique a família de funções esponja Keccak

3.h. Explique a especificação da função SHA-3

- i. Funções de *hash* SHA-3
- ii. Funções de saída estendida

3.i. Explique a construção da esponja

3.j. Apresente a análise de segurança

3.k. Exemplos

Referências

- Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. (2011). Cryptographic sponge functions.
- Dworkin, M. J. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report.