

# Função de Hash Criptográfica SHA-3

Ranieri Althoff

<sup>1</sup>Universidade 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 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,  $r + c = 1600$ .

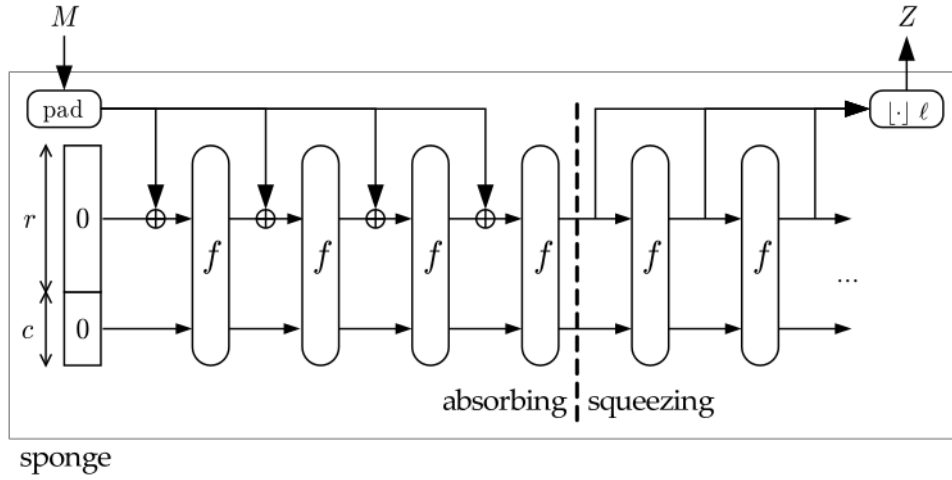
Aumentar o tamanho de  $r$  para tornar o algoritmo mais rápido (gerando o *hash* com menos iterações) diminui, portanto, o nível de segurança do *hash* gerado, porém um nível muito alto de segurança implica numa quantidade muito alta de iterações para gerar o *hash*.

## 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  $\text{xor}$  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  $A$   $5 \times 5$ , 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, 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

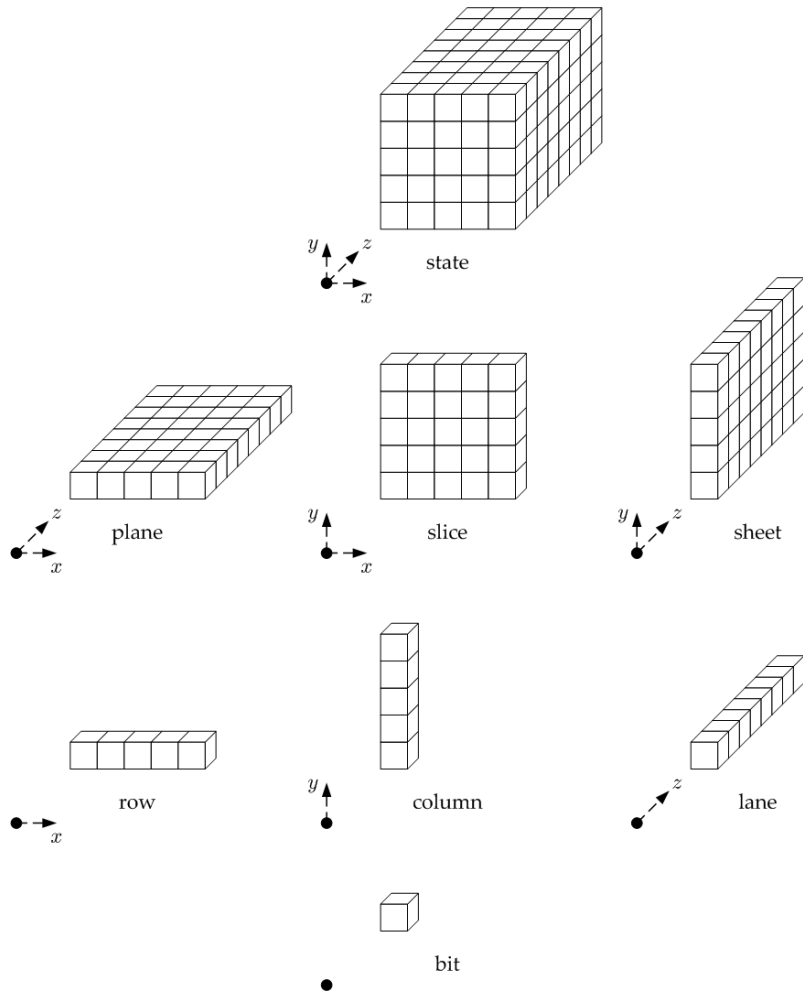
### 3.a. O que é e para que serve o *state array*?

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.

O primeiro valor do estado é gerado à partir da mensagem  $S$  a ser cifrada, e o resultado do hash é o estado após o último passo convertido novamente em *string*, passos estes que serão explicados nas próximas questões.



**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  $\text{KECCAK-}p[b, n_r]$ . 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.

É esta característica que gera o tamanho  $b$  da *string*  $S$ , porque para cada pista de  $w$  *bits*, existem  $5 \cdot 5 = 25$  células pelo formato da matriz, portanto  $b = 25 \cdot w$ .

### 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 (gerando os planos) e então por linha. Utilizando-se da

mesma fórmula posicional,

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

ou

$$S[i] = A[\lfloor i \div (5w) \rfloor, \lfloor i \div w \pmod{w} \rfloor, i \pmod{w}]$$

Primeiro se concatenam os  $w$  *bits* de uma pista:

$$\text{Lane}(i, j) = A[i, j, 0] \parallel A[i, j, 1] \parallel \dots \parallel A[i, j, w - 1]$$

Onde  $\parallel$  denota concatenação de *strings*, de forma que as pistas da coluna  $i = 0$ , usando  $w = 64$ , sejam:

$$\begin{aligned} \text{Lane}(0, 0) &= A[0, 0, 0] \parallel A[0, 0, 1] \parallel \dots \parallel A[0, 0, 63] \\ \text{Lane}(1, 0) &= A[1, 0, 0] \parallel A[1, 0, 1] \parallel \dots \parallel A[1, 0, 63] \\ &\vdots \\ \text{Lane}(5, 0) &= A[5, 0, 0] \parallel A[5, 0, 1] \parallel \dots \parallel A[5, 0, 63] \end{aligned}$$

E assim para todas as pistas. A *string* que representa cada plano é a concatenação de todas as pistas na sua coluna  $j$ :

$$\text{Plane}(j) = \text{Lane}(0, j) \parallel \text{Lane}(1, j) \parallel \dots \parallel \text{Lane}(4, j)$$

De forma que os planos de cada uma das colunas  $0 \leq j < 5$  seja:

$$\begin{aligned} \text{Plane}(0) &= \text{Lane}(0, 0) \parallel \text{Lane}(1, 0) \parallel \dots \parallel \text{Lane}(4, 0) \\ \text{Plane}(1) &= \text{Lane}(0, 1) \parallel \text{Lane}(1, 1) \parallel \dots \parallel \text{Lane}(4, 1) \\ &\vdots \\ \text{Plane}(4) &= \text{Lane}(0, 4) \parallel \text{Lane}(1, 4) \parallel \dots \parallel \text{Lane}(4, 4) \end{aligned}$$

A concatenação destes planos, então, gera a *string* a partir do estado:

$$S = \text{Plane}(0) \parallel \text{Plane}(1) \parallel \dots \parallel \text{Plane}(4)$$

No resultado final, a concatenação de todos os *bits* gera a seguinte *string*:

$$S = A[0, 0, 0] \parallel \dots \parallel A[0, 0, 63] \parallel A[0, 1, 0] \parallel \dots \parallel A[0, 4, 63] \parallel A[1, 0, 0] \parallel \dots \parallel A[4, 4, 63]$$

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

Cada rodada do SHA-3 é composta por cinco passos de mapeamento do KECCAK- $p$ , representados pelas funções  $\iota$ ,  $\chi$ ,  $\pi$ ,  $\rho$  e  $\theta$ , que recebem como parâmetros posições de *bits*, colunas e linhas e manipulam o *state array* da operação.

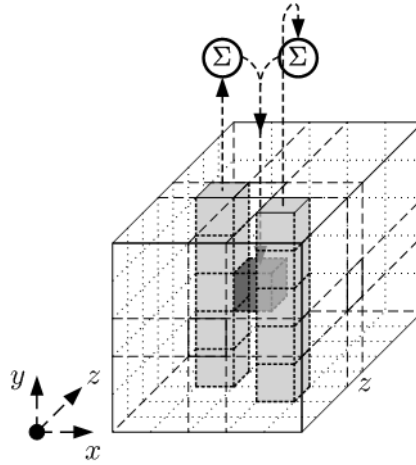
#### 3.d.1. Função *theta* $\theta$

A função *theta* é definida pela seguinte fórmula:

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

É uma função de substituição que utiliza os *bits* de colunas adjacentes e da mesma pista na qual está sendo aplicada. Para cada pista  $A[x, y]$ , cada um dos  $w$  *bits*  $z$  é logicamente somado com um somatório de uma coluna da linha anterior e com um somatório de uma coluna da linha posterior, como mostrado na figura 3.





**Figura 3. Visualização da função theta.**

Dessa forma, cada um dos *bits* do *state array* é afetado por 11 *bits* do estado anterior, sendo 5 de cada coluna utilizada e o próprio *bit* em que a função foi aplicada. Isso cria um alto grau de difusão dos efeitos de todas as funções no resultado.

A função *theta* é a primeira porque mistura a parte interna do estado (invisível para um atacante) com a parte externa, muito por causa do funcionamento das funções esponja, portanto um atacante não pode acessar a entrada das funções subsequentes apenas analisando a parte visível do estado.

### 3.d.2. Função *rho* $\rho$

A função *rho* é definida pela seguinte fórmula:

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

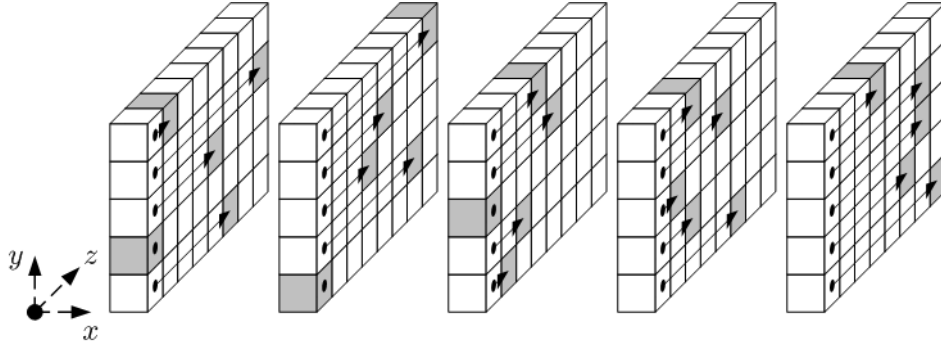
É uma função de permutação que utiliza os *bits* de uma pista  $A[x, y]$ . No caso de  $(x, y) = (0, 0)$ , a função não altera os *bits*, mas para os outros casos, ela aplica um deslocamento circular dos *bits* como um embaralhamento dos mesmos, usando o valor  $t$  para definir quantos bits será deslocado, como mostrado na figura 4.

Essa transformação gera a difusão entre os *bits* de uma mesma pista, acelerando os efeitos das funções em *bits* próximos da *string* de entrada e do *state array*, já que as outras funções criam difusão entre linhas, colunas e pistas, mas não entre *bits*.

### 3.d.3. Função *pi* $\pi$

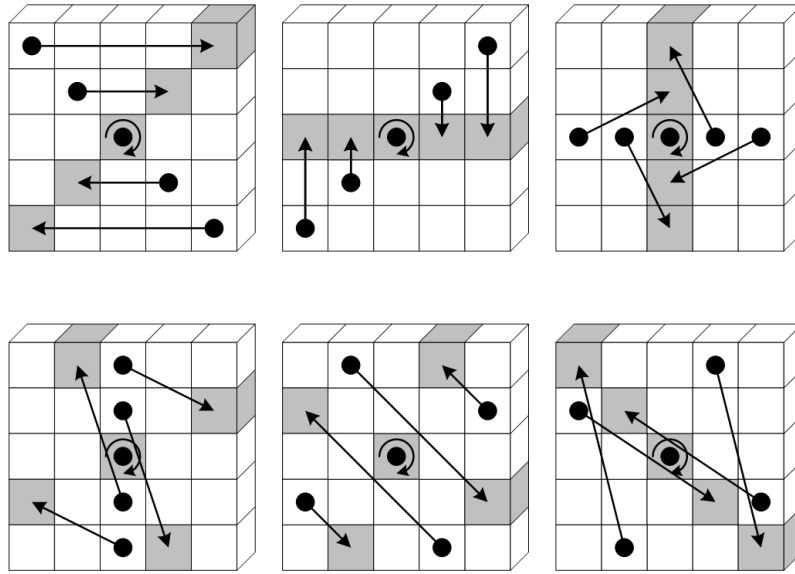
A função *pi* é definida pela seguinte fórmula:

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



**Figura 4. Visualização da função rho.**

É uma função de permutação que utiliza as pistas. Assim como *theta*, *pi* faz um deslocamento circular usando uma fórmula semelhante, como mostrado na figura 5.



**Figura 5. Visualização da função pi.**

Essa transformação gera a difusão entre diferentes pistas.

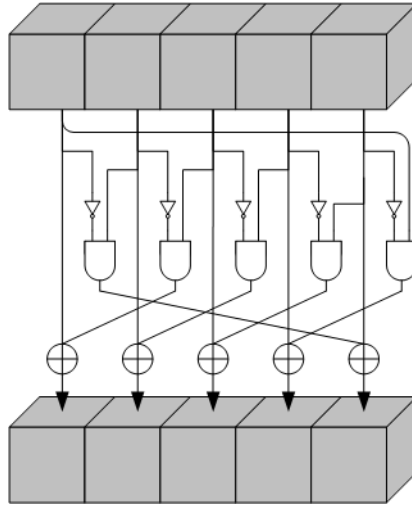
#### 3.d.4. Função *chi* $\chi$

A função *chi* é definida pela seguinte fórmula:

$$\chi(M[x, y, z]) = M[x, y, z] \oplus (\neg M[x + 1, y, z] \wedge M[x + 2, y, z])$$

É uma função de substituição que utiliza os *bits* posteriores ao *bit* sendo aplicado, sendo o passo não-linear que torna a função KECCAK uma função irreversível, ou seja, impede a obtenção da pré-imagem a partir do *hash*. O passo *chi* é mais facilmente visualizado se usando um circuito digital, como mostrado na figura 6.

Sua definição possui propriedades algébricas que garantem que a saída não possui correlação direta com a entrada em termos de paridade, ou seja, não é possível concluir nada sobre a entrada analisando a quantidade de *bits* em 0 ou 1 da saída.



**Figura 6. Visualização da função chi.**

### 3.d.5. Função *iota* $\iota$

A função *iota* é definida pela seguinte fórmula:

$$\iota(M[x, y]) = M[x, y] \oplus RC_i$$

É uma função de substituição baseada em um valor tabelado e diferente para cada rodada do SHA-3, e é gerado a partir de uma fórmula que utiliza um gerador de deslocamento linear realimentado (LFSR, na sigla em inglês) em  $GF(2)$ :

$$RC_i[2^j - 1] = (x^{j+7i} \pmod{x^8 + x^6 + x^5 + x^4 + 1}) \pmod{x} \text{ para } 0 \leq j \leq \log w$$

Dessa forma, não existe relação de simetria entre as diferentes rodadas, porque sem este passo, todas as rodadas teriam a mesma fórmula e poderiam ser simplificadas para permitir ataques.

A função *iota* é somente aplicada na primeira linha e coluna, ou seja, os parâmetros  $x$  e  $y$  são sempre 0, e seu efeito é difundido para as outras linhas e colunas através dos passos *theta* e *chi*.

### 3.e. Explicar a permutação KECCAK- $p[b, n_r]$

Uma função KECCAK- $p[b, n_r]$  é uma generalização das funções de substituição e permutação KECCAK que podem ser usadas pelo SHA-3 e recebe como parâmetros o comprimento da *string*  $S$ , que define o tamanho do *state array*, e o número de rodadas na fase de absorção.

Cada uma das rodadas  $R_i$  de KECCAK- $p$  consiste na aplicação dos cinco passos de transformação, vistos na seção anterior:

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

O *state array* na rodada  $i$ , denominado  $S_i$ , é encontrado pela seguinte composição de funções:

$$S_i = \iota(\chi(\pi(\rho(\theta(S_{i-1}))), i)$$

Onde  $j \geq 2l + 12 - n_r$ , sendo  $l = \log w$ , e  $i < 2l + 12$ . Essa definição genérica nos permite gerar versões do KECCAK para diferentes tamanhos de palavras  $w$ , desde que  $w = 2^k$ . É conveniente utilizar 32 ou 64 por ser o tamanho de uma palavra em processadores modernos, e o FIPS 202 define os seguintes parâmetros por padrão:

$$b \in \{25, 50, 100, 200, 400, 800, 1600\}$$

$$n_r = 2l + 12, \text{ onde } l \in \mathbb{Z}$$

### 3.f. Descrever o *framework sponge construction*

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,  $r + c = b$ .

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  $\times \oplus r$  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.

### 3.g. Explique a família de funções esponja KECCAK

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 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:

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

Para que a mensagem seja transformada pela função esponja, ela precisa que uma função de *padding*  $pad$  seja aplicada. O algoritmo utilizado pelo KECCAK para *padding* da entrada se chama  $pad10 * 1$  e é definido por  $pad(M) = b - (|M| \bmod b)$ , onde  $b$  é o tamanho do bloco em *bits* da esponja e  $|M|$  é o comprimento da mensagem em *bits*.

O resultado  $P$  da função  $pad10 * 1$  tem o formato binário  $1 || 0^{m-2} || 1$ , ou seja,  $m - 2$  zeros cercados por uns.

A função KECCAK[ $c$ ], onde  $c$  define a capacidade da função, é definida por:

$$\text{KECCAK}[c] = \text{Sponge}[\text{KECCAK-}p[1600, 24], \text{pad}10 * 1, 1600 - c]$$

Ou seja, é uma aplicação de função esponja cujo parâmetro  $f$ , a função de transformação, é KECCAK- $p$  com 1600 *bits* de estado e 24 rodadas, utiliza  $\text{pad}10 * 1$  como função de *padding* da mensagem e possui bloco de tamanho  $r = 1600 - c$ .

A utilização destes parâmetros define KECCAK[ $c$ ] como uma função que aceita uma mensagem de tamanho arbitrário, onde cada uma das 24 rodadas processa um bloco da mensagem de tamanho  $r = 1600 - c$  e com grau de segurança  $c$ .

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

A definição do SHA-3 publicada pelo NIST prevê dois tipos de função baseadas no KECCAK, funções de *hash* criptográficas e funções de saída estendida (XOF, ou *extendable output function*), também chamadas de SHAKE.

#### i. Funções de *hash* SHA-3

As funções de *hash* podem ser definidas genericamente para um tamanho do *hash* gerado  $d$  e mensagem  $M$  com a seguinte fórmula:

$$\text{SHA-3}[d](M) = \text{KECCAK}[2d](M || 01, d)$$

Observando que os bits 01 são concatenados à mensagem  $M$  antes da execução a função. O NIST definiu  $d \in \{224, 256, 384, 512\}$  para o SHA-3, existindo portanto SHA3-224, SHA3-256, SHA3-384 e SHA3-512:

- $\text{SHA3-224}(M) = \text{KECCAK}[448](M || 01, 224)$
- $\text{SHA3-256}(M) = \text{KECCAK}[512](M || 01, 256)$
- $\text{SHA3-384}(M) = \text{KECCAK}[768](M || 01, 384)$
- $\text{SHA3-512}(M) = \text{KECCAK}[1024](M || 01, 512)$

Estes tamanhos são escolhidos de forma que o SHA-3 seja uma substituição do SHA-2, que utiliza os mesmos tamanhos.

Seria possível, caso desejado, definir tamanhos maiores, como 1024 ou 2048, provendo um espaço muito maior e diminuindo a quantidade de colisões, mas tornando o algoritmo mais lento e diminuindo o grau de segurança do *hash*.

#### ii. Funções de saída estendida (SHAKE)

As funções XOF podem ser definidas genericamente para uma capacidade de segurança  $c$  e mensagem  $M$  com a seguinte fórmula:

$$\text{SHAKE}[c](d, M) = \text{KECCAK}[c](M || 1111, d)$$

Observando que os bits 1111 são concatenados à mensagem  $M$  antes da execução da função. O NIST definiu  $c \in \{128, 256\}$  para o SHAKE, existindo portanto SHAKE-128 e SHAKE-256.

- $\text{SHAKE-128}(M) = \text{KECCAK}[128](M || 1111, 128)$

- $\text{SHAKE-256}(M) = \text{KECCAK}[256](M \parallel 1111, 256)$

Nota-se que o tamanho da saída, diferente das funções SHA-3, é apenas  $d = c$ , e não  $2d$ , porque o KECCAK pode gerar um comprimento infinito de *bits*, e mais podem ser gerados se "espremendo" a esponja.

### 3.i. Apresente a análise de segurança

### 3.j. Exemplos

- SHA3-224("segurança em computação")

```
87 0B D5 50 EB 20 44 45 22 AF 19 1A A0 DA 03 CC 11 FB
76 92 70 B2 4F DA BB 64 6F EC
```

- SHA3-256("segurança em computação")

```
B1 CB C2 AF 9D 81 58 B2 32 DE 7B C0 88 C0 48 E1 F9 0F
7F E2 65 6C 76 A1 AB 44 0A EF 46 EF DF 2E
```

- SHA3-384("segurança em computação")

```
B7 9C D7 EE 92 2B 78 A3 D5 CB D2 2E 4C 75 69 42 AE 10
86 07 78 D4 BD 90 2E A3 EE 74 38 D1 CD 26 C0 B2 58 06
14 20 89 FB 85 71 07 29 BE 94 26 AB
```

- SHA3-512("segurança em computação")

```
C9 AC B8 EE F9 8F 4A 4D 11 75 DC 5B 3B CF C9 83 D8 AB
C7 88 6E 90 27 C7 8A 6F E6 91 57 0F CE F9 26 10 79 1E
1E DF 00 5C D6 12 98 0E 2D 71 5E 43 CD E9 CE 4F 68 59
BE B5 30 14 D3 4D 64 67 EB 28
```

## 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.