Função de Hash Criptográfica SHA-3

Quênio César Machado dos Santos (14100868)

INE5429 - Segurança em Computadores

Florianópolis, 28/06/2016

Sumário

1 Função Hash Criptográfica

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 *hash*. Esta função tem a forma:

$$h = H(M)$$

Onde:

- H é a função *hash* que gerou o valor h.
- h é o valor *hash* de tamanho fixo gerado pela função *hash*.
- 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:

$$M \mid H(M) = h$$

- dois valores M_1 e M_2 que resultem no mesmo valor h:

$$(M_1, M_2) \mid H(M_1) = H(M_2)$$

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 Infeção 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, defini-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 formula: $2^{\frac{m}{n}}$. Se permitimos 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 para descobrir o valor secreta S usada 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* $h = H(M)$ 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 $H^{-1}(h) = S \parallel M$, 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 sem 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 Pre-Imagem	Resistente a Segunda Pre-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 SHA-3

SHA-3 é uma função *hash* criptográfica publicada pelo NIST em agosto de 2015 para substituir o SHA-2 como o padrão para os sistemas de informação dos departamentos e das agências do governo americano. SHA-3 provavelmente será adotado por sistemas operacionais e também por organizações privadas e públicas de todo o mundo, assim como foi o caso do SHA-2 e SHA-1.

2.1 A Estrutura do SHA-3

A estrutura de entrada do SHA-3 segue a estrutura genérica de outras funções *hash* iterativas, onde o resultado de uma função de compressão f é iterativamente aplicado sobre a mesma função, juntamente com o próximo bloco P_i da mensagem de entrada, como ilustrado no diagrama abaixo:

$$\begin{tikzcd}
 & {} \arrow[d, "|P_1| = r"] & {} \arrow[d, "|P_2| = r"] & & {} \arrow[d, "|P_k| = r"] & \\
{} \arrow[r, "|S_0|=b"] & f_1 \arrow[r, "|S_1|=b"] & f_2 \arrow[r, "|S_2|=b"] & \cdots \arrow[r, "|S_{k-1}|=b"] & f_k \arrow[r, "|S_k|=b"] & {}
\end{tikzcd}$$

No esquema ilustrado acima, uma mensagem de entrada de n *bits* é dividida em k blocos de tamanho r *bits*: P_1, P_2, ..., P_n. O último bloco é sempre preenchido para que tenha r *bits*. Cada bloco é processado com a saída S_{i-1} da execução anterior da função f. O símbolo f_i, com $1 \leq i \leq k$, representa a execução da função f na iteração i gerando o resultado S_i de b *bits*. Após todos os blocos P_i serem processados, produz-se o valor S_k.

Apesar de seguir o esquema genérico, descrito acima, na sua estrutura de entrada, o SHA-3 possui uma característica peculiar, descrita abaixo, na sua estrutura de saída. Combinando ambas as estruturas, o SHA-3 permite um número variável de *bits* tanto na entrada como na saída. Este fato o torna mais flexível e aplicável não somente como função *hash*, mas também como um gerador de números pseudo-aleatórios; além de permitir outras aplicações. Devido a esta característica, os criadores do SHA-3 chamam sua estrutura de função *esponja*.

Observe a estrutura de saída do SHA-3 no diagrama abaixo:

$$\begin{tikzcd}
{} \arrow[r, "|S_k|=b"] & f_{k+1} \arrow[r, "|S_{k+1}|=b"] \arrow[d, "|Z_1| = r"] & f_{k+2} \arrow[r, "|S_{k+2}|=b"] \arrow[d, "|Z_2| = r"] & \cdots \arrow[r, "|S_{k+j-1}|=b"] & f_{k+j} \arrow[r, "|S_{k+j}|=b"] \arrow[d, "|Z_j| = r"] & {} \\
 & {} & {} & & {} &
\end{tikzcd}$$

No esquema ilustrado acima, o valor S_k proveniente da estrutura de entrada serve como valor inicial da estrutura de saída. Após processar os k blocos da mensagem de entrada, e usando a mesma função de compressão f, a função esponja gera uma sequência de j blocos: Z_1, Z_2, ..., Z_j. O número de blocos de saída j é determinado pelo número de *bits* de saída desejado. Se l *bits* são necessários na saída, então:

$$(j-1) \times r < l \leq j \times r$$

Esta flexibilidade no número de l *bits* de saída é o que dá o nome *esponja* à função do SHA-3. De acordo com esta analogia, quando a estrutura de entrada está consumindo os n *bits* da mensagem de entrada, diz-se que a função *esponja* está "absorvendo" os *bits* de entrada. E quando a estrutura de saída está gerando os l *bits* de saída, diz-se que a função *esponja* está sendo "espremida" para liberar os *bits* de saída.

2.2 A Fase de Absorção

A primeira fase da função esponja se chama de *absorção* e refere-se ao processamento dos blocos da mensagem de entrada. Veja a fase de absorção na ilustração abaixo:

$$\begin{tikzcd}
 & {} \arrow[d, "P_1^r \parallel 0^c"] & & & {} \arrow[d, "P_2^r \parallel 0^c"] & & & & {} \arrow[d, "P_k^r \parallel 0^c"] & & \\
{} \arrow[r, "S_0^b = 0^{r+c}"] & \oplus \arrow[r] & f \arrow[r, "S_1^b"] & {} & \oplus \arrow[r] & f \arrow[r, "S_2^b"] & \cdots \arrow[r, "S_{k-1}^b"] & {} & \oplus \arrow[r] & f \arrow[r, "S_k^b"] & {}
\end{tikzcd}$$

Como ilustrado na figura acima, existe uma variável de estado s que é utilizada nesta fase. Estava variável serve como entrada e saída de cada iteração que aplica a função de compressão f. Inicialmente, s contém 0 em todos os seus *bits*. Seu valor vai se modificando em cada iteração.

O tamanho de s é de b *bits*, onde:

$$b = r + c$$

Como visto na seção anterior, r é o tamanho de cada bloco P_i da mensagem de entrada. Também é chamado de *bitrate*, ou vazão de *bits*, pois representa o número de *bits* consumidos em cada iteração da função esponja.

O número de *bits* c é chamado de *capacidade* e representa o nível de segurança atingido pela função esponja. Dado o valor padrão de $b = r + c = 1600$ no SHA-3, quanto maior o numero c, maior a segurança da função, porém menor o *bitrate*.

Em cada iteração, a fase de absorção ocorre da seguinte forma:

- O próximo bloco P_i da mensagem de entrada é preenchido com zeros ($P_i^r \parallel 0^c$) para aumentar seu tamanho de r para b *bits*.
- Ao resultado do passo anterior, uma operação *XOR* é aplicada, tendo como segundo operando o valor de s_{i-1} proveniente da iteração anterior, ou zeros se for a primeira iteração ($S_0^b = 0^{r+c}$).
- O resultado da operação *XOR* serve então de entrada para a função de compressão f. O resultado desta função é o novo valor S_i da variável s, que é usada como entrada da próxima iteração, juntamente com o próximo bloco P_{i+1} da mensagem de entrada.

Se o tamanho desejado da saída da função esponja é menor que o tamanho de s - ou seja, se $l \leq b$ - então os primeiros l *bits* de s_k - retornado pela última iteração - é o resultado da função esponja. Caso deseja-se $l > b$, então a fase de "espremer a esponja" inicia-se, como descrito na próxima seção.

2.3 "Espremendo a Esponja"

Na faze de absorção da função esponja, descrita na seção anterior, foram consumidos todos os blocos da mensagem de entrada, resultando num valor final de s de b *bits*. Se o tamanho desejado de saída (l) da função esponja for $l > b$, diz-se que é preciso *espremer a esponja* para obter uma saída com o número de *bits* desejado.

A fase de *espremer a esponja* é ilustrada abaixo:

$$\begin{tikzcd}
s_k \arrow[r, "f"] \arrow[d, "Z_0^r"'] & s_{k+1} \arrow[r] \arrow[d, "Z_1^r"'] & \dots \arrow[r, "f"] & s_{k+j} \arrow[d, "Z_{j-1}^r"'] \\
{} & {} & & {}
\end{tikzcd}$$

Observando a ilustração acima, pode-se descrever esta fase da seguinte forma:

- Primeiramente, os primeiros r *bits* de s_k são colocados num bloco Z_0.
- Então o valor de s_k é aplicado na função f para se obter novo valor de s_{k+1}.
- Os primeiro r *bits* do novo valor de s_{k+1} são colocados num bloco Z_1.
- Este processo se repete até que se tenha j blocos (Z_0, Z_1, ..., Z_{j-1}) tal que $(j-1) \times r < l \leq j \times r$.

Ao final deste processo, a saída da função esponja serão os primeiros l *bits* dos blocos concatenados $Z_0 \parallel Z_1 \parallel \dots \parallel Z_{j-1}$.

2.4 Função de Compressão Keccak

Nas seções anteriores, foi dado uma visão geral da estrutura da função esponja utilizada por SHA-3. Nesta seção, o foco será a função de compressão utilizada em cada iteração do SHA-3, que é chamada de *Keccak* pelos seus autores.

Como visto anteriormente, a função Keccak (f) tem como entrada um valor s de b *bits*, onde $b = r + c = 1600$. No processamento interno da função f, o valor s é organizado numa matriz 5×5 com valores de 64 *bits* em cada uma de suas células. Esta matriz pode ser linearizada num vector de *bits*, correspondendo ao valor s, usando a seguinte fórmula:

$$s[64(5y + x) + z] = M[x, y, z]$$

Onde:

- M é a matriz 5×5 com valores de 64 *bits*.
- x é o índice de coluna na matriz, que vai de 0 a 4.
- y é o índice de linha na matriz, também de 0 a 4.
- z é o índice de *bit* de uma célula na matriz, que vai de 0 a 63.

Uma vez criada a matriz, a função f vai executar 24 rodadas de processamento:

$$
\begin{array}{ccccccccc}
 & RC_1 & & RC_2 & & & & RC_{24} & \\
 & \downarrow & & \downarrow & & & & \downarrow & \\
\xrightarrow{S_i} & R_1 & \longrightarrow & R_2 & \longrightarrow & \dots & \longrightarrow & R_{24} & \xrightarrow{S_{i+1}}
\end{array}
$$

Observe acima que todas as rodadas são idênticas, exceto pela constante RC_i diferente em cada rodada. Cada uma das rodadas, consiste de 5 passos. Cada passo executa uma operação de permutação ou substituição sobre a matriz.

A aplicação dos cinco passos de cada rodada é expressa pela composição das seguintes funções:

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

Onde cada passo tem sua fórmula na tabela seguinte:

Função	Fórmula
Theta	$\theta : M[x, y, z] \leftarrow M[x, y, z] \oplus \sum_{y'=0}^{4} M[x-1, y', z] \oplus \sum_{y'=0}^{4} M[x+1, y', z-1]$
Rho	$\rho : M[x, y, z] \leftarrow \begin{cases} M[x, y, z], \text{ se } x = y = 0 \\ M[x, y, z - \frac{(t+1)(t+2)}{2}], \text{ onde } 0 \leq t < 24 \text{ e } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \\ \text{em } GF(5)^{2\times 2} \end{cases}$
Pi	$\pi : M[x, y] \leftarrow M[x', y']$, onde $\begin{pmatrix} x \\ y \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix}$
Chi	$\chi : M[x, y, z] \leftarrow M[x, y, z] \oplus (\neg M[x+1, y, z] \wedge M[x+2, y, z])$
Iota	$\iota : M[x, y, z] \leftarrow M[x, y, z] \oplus RC(i)$, onde RC é uma tabela com um valor para cada rodada i.

Baseado na tabela acima, aqui estão algumas características de cada passo:

- *Theta* (θ) é uma função de substituição que utiliza *bits* das colunas anteriores e posteriores, além dos *bits* da célula sendo substituída. Cada *bit* substituído depende de outros 11 *bits*, o que provê uma difusão de alto grau.
- *Rho* (ρ) é uma função de permutação dos *bits* dentro de cada célula. Sem esta função, a difusão entre as células, ocorreria de forma muito lenta.
- *Pi* (π) é também uma função de permutação, mas entre células. A rotação não se dá nos *bits* de uma célula, mas entre as células da matriz.
- *Chi* (χ) é uma função de substituição baseada no valor do *bit* corrente e dos *bits* em

posições correspondentes das duas próximas células. Sem esta função, SHA-3 seria completamente linear.

- *Iota* (ι) é uma função de substituição baseada numa tabela - chamada RC, ou seja, constantes de rodada - que usará um valor constante e diferente para cada rodada do SHA-3.

2.5 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 *Hash* (l)	Tamanho do Bloco (r)	Capacidade (c)	Resistência a Colisão	Resistência à 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 (r 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 l 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 l é sempre menor que r nestes casos.

3 Perguntas e Respostas

Nas sub-seções abaixo, encontram-se as perguntas e respostas sobre a referência [FIPS202] que apresenta o SHA-3 padronizado pelo NIST.

3.1 O que é e para que serve o *State Array*?

State Array é simplesmente a *string* S - que serve de entrada à função Keccak - representada na forma de uma matriz M de $5x5$, tendo como suas células as palavras de *bits* de comprimento w, que varia de 1 a 64 *bits*.

Representando a string S numa matriz facilita a definição e a implementação das sub-funções de substituição e permutação ($\theta, \rho, \pi, \chi, \iota$) que fazem parte da função Keccak.

Após realizar as várias transformações no *State Array* (ou seja, na matriz M), a função Keccak terá como sua saída o estado final encontrado nesta matriz, que então será convertida novamente no formato da *string* S, servindo por sua vez como a entrada da próxima rodada do SHA-3.

3.2 Como é feita a conversação de strings para State Arrays?

Uma *bit string* S pode ser convertida numa matriz M de 5×5 de palavras de comprimento w *bits*. Para tanto, é preciso que o comprimento b da *bit string* S tenha como múltiplos 25 ($5x5 = 25$) e w, de tal forma que $b = 25w$.

Respeitada a condição definida acima, é possível estabelecer um mapeamento entre cada *bit* da *string* S e cada *bit* de uma palavra da matriz M, usando a seguinte equação:

$$M[x, y, z] = S[w(5y + x) + z]$$

Onde:

- M é a matriz 5×5 com palavras de w *bits*.
- x é o índice de coluna na matriz, que vai de 0 a 4.
- y é o índice de linha na matriz, também de 0 a 4.
- z é o índice de *bit* de uma palavra na matriz M, que vai de 0 a $w - 1$.

Fazendo $w = 2$ e $b = 50$, o que faz as palavras da matriz M ter apenas 2 *bits* e a *bit string* S ter 50 *bits*, podemos visualizar o mapeamento na seguinte tabela:

$M[x, y]$	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 0$	$S[0] \| S[1]$	$S[2] \| S[3]$	$S[4] \| S[5]$	$S[6] \| S[7]$	$S[8] \| [9]$
$y = 1$	$S[10] \| S[11]$	$S[12] \| S[13]$	$S[14] \| S[15]$	$S[16] \| S[17]$	$S[18] \| S[19]$
$y = 2$	$S[20] \| S[21]$	$S[22] \| S[23]$	$S[24] \| S[25]$	$S[26] \| S[27]$	$S[28] \| S[29]$
$y = 3$	$S[30] \| S[31]$	$S[32] \| S[33]$	$S[34] \| S[35]$	$S[36] \| S[37]$	$S[38] \| S[39]$
$y = 4$	$S[40] \| S[41]$	$S[42] \| S[43]$	$S[44] \| S[45]$	$S[46] \| S[47]$	$S[48] \| S[49]$

Ou seja:

- $M[0, 0, 0] = S[0]$
- $M[0, 0, 1] = S[1]$
- $M[1, 0, 0] = S[2]$
- $M[1, 0, 1] = S[3]$
- ...

E assim por diante, até que todas as palavras da matriz M sejam preenchidas com os *bits* da *string* S.

3.3 Como é feita a conversão de State Array para Strings?

Pode-se converter a matriz M (o *State Array*) de volta a uma *bit string* S usando a mesma equação definida anteriormente, apenas invertendo as expressões, como mostrado abaixo:

$$S[w(5y + x) + z] = M[x, y, z]$$

Uma vez definidos cada um dos *bits*, faz-se a concatenação destes para obter *string* S:

$$S = M[x, y, 0] \| M[x, y, 1] \| \ldots \| M[x, y, w - 1], 0 \leq x \leq 4, 0 \leq y \leq 4$$

Usando ainda o exemplo com $w = 2$ e $b = 50$, temos:

$$S = \\ M[0,0,0] \| M[0,0,1] \| M[1,0,0] \| M[1,0,1] \| M[2,0,0] \| M[2,0,1] \| M[3,0,0] \| M[3,0,1] \| M[4,0,0] \| M[4,0,1] \| \\ M[0,1,0] \| M[0,1,1] \| M[1,1,0] \| M[1,1,1] \| M[2,1,0] \| M[2,1,1] \| M[3,1,0] \| M[3,1,1] \| M[4,1,0] \| M[4,1,1] \| \\ \ldots \| M[4,4,1]$$

3.4 Passos de Mapeamento (*Step Mappings*)

Para cada rodada do SHA-3, os cinco passos de mapeamento de *Keccak-p* - representados pelas funções $\theta, \rho, \pi, \chi, \iota$ - fazem substituições e permutações sobre as palavras da matriz M (o *State Array*) até que se encontre o estado final da *string* S naquela rodada.

Cada um dos passos de mapeamento são descritos a seguir na ordem em que são aplicados à matriz M.

3.4.1 Passo Theta (θ)

$$\theta : M[x,y,z] \leftarrow M[x,y,z] \oplus \sum_{y'=0}^{4} M[(x-1) \bmod 5, y', z] \oplus \sum_{y'=0}^{4} M[(x+1) \bmod 5, y', z-1]$$

Theta (θ) é uma função de substituição que utiliza *bits* das colunas anteriores e posteriores, além dos *bits* da palavra $M[x,y]$ sendo substituída.

Para cada palavra $M[x,y]$:

- O primeiro somatório faz um XOR entre todas as palavras da coluna anterior. O segundo somatório, faz o mesmo com a coluna posterior. Se a coluna da palavra $M[x,y]$ for a primeira, usa-se a última coluna para o primeiro somatório. Se a coluna de $M[x,y]$ for a última, usa-se a primeira coluna para o segundo somatório.
- Um XOR então é aplicado entre os valores resultantes dos somatórios e o valor de $M[x,y]$.

Observa-se que cada *bit* substituído pela função θ depende de outros 11 *bits* da matriz M (das colunas anteriores e posteriores, assim como o *bit* anterior na mesma posição). Isto provê uma difusão de alto grau.

3.4.2 Passo Rho (ρ)

$$\rho : M[x,y,z] \leftarrow \begin{cases} M[x,y,z], \text{ se } x = y = 0 \\ M[x,y,z - \frac{(t+1)(t+2)}{2}], \text{ onde } 0 \le t < 24 \text{ e } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \\ \text{ em } GF(5)^{2\times 2} \end{cases}$$

Rho (ρ) é uma função de permutação dos *bits* dentro de cada palavra $M[x,y]$.

Este passo funciona da seguinte forma:

- Se $(x,y) = (0,0)$, então $M[x,y]$ não é afetado.
- O valor de t é usado para determinar quantas posições de *shift* circular serão usadas ($\frac{(t+1)(t+2)}{2}$) e também em que palavra ocorrerá o *shift*:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Observa-se que, sem esta transformação, a difusão entre as palavras da matriz M ocorreria de forma muito lenta.

3.4.3 Passo Pi (π)

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

Pi (π) é também uma função de permutação, mas entre as palavras. Ou seja, a rotação *não* se dá nos *bits* de uma palavra, mas entre as palavras da matriz M.

3.4.4 Passo Chi (χ)

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

Chi (χ) é uma função de substituição baseada no valor do *bit* corrente e dos *bits* em posições correspondentes das duas próximas células.

Este passo é o único mapeamento não-linear. Sem este passo, SHA-3 seria mais suscetível a ataques.

3.4.5 Passo Iota (ι)

$$\iota : M[x, y, z] \leftarrow M[x, y, z] \oplus RC(i)$$

Iota (ι) é uma função de substituição baseada numa tabela - chamada RC, ou seja, constantes de rodada - que usará um valor constante e diferente para cada rodada i do SHA-3.

A tabela RC pode ser calculada com o seguinte algoritmo:

```
função RC(i):
    se i mod 255 = 0, retorne 1
    faça R = 10000000
    para i de 1 to t mod 255, faça:
        R = 0 || R
        R[0] = R[0] + R[8]
        R[4] = R[4] + R[8]
        R[5] = R[5] + R[8]
        R[6] = R[6] + R[8]
        R = Truncar8[R]
retorne R[0]
```

3.5 Permutação $Keccak\text{-}p[b, n_r]$

Keccak-p é a generalização das funções de substituição e permutação *Keccak* que podem ser usadas pelo algoritmo do SHA-3. Tem como parâmetros:

- b: o comprimento em *bits* da *bit string* S, usada como a variável de estado do SHA-3.
- n_r: o número de rodadas da fase de absorção do SHA-3.

Cada rodada de substituições e permutações de *Keccak-p* consiste numa sequência de cinco transformações (*step mappings*), como visto anteriormente, modificando o estado da *string* S:

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

Onde:

- $i \geq 2l + 12 - n_r$ e $i < 2l + 12, l \in \mathbb{Z}$.
- S_i é o estado da *string* S na rodada i.

Observe que, em *Keccak-p*, o índice da rodada i pode ser um inteiro negativo.

3.5.1 *Keccak-f*

Keccak-f é uma especialização de *Keccak-p*, onde fixamos os parâmetros da seguinte forma:

- $b \in \{25, 50, 100, 200, 400, 800, 1600\}$
- $n_r = 2l + 12, l \in \mathbb{Z}$.

Devido a fixação dos parâmetros, pode-se definir que:

$$Keccak\text{-}f[b] = Keccak\text{-}p[b, 12 + 2l]$$

Observe que, em *Keccak-f*, o índice da rodada será sempre um inteiro não-negativo no intervalo:

$$0 \leq i < n_r$$

3.6 Estrutura Esponja

É uma estrutura usada para encadear a execução de uma função f, operando sobre dados binários de comprimento variável (a mensagem) e gerando um valor *hash* de tamanho arbitrário.

Para que a mensagem possa ser consumida pela estrutura esponja, esta precisa ter um determinado tamanho em *bits*. Para isto, é necessário aplicar uma regra de *padding* que alongue a mensagem para o tamanho desejado.

A estrutura, como ilustrada na figura 7 de [FIPS2002], é dividida em duas partes (absorção e liberação), descritas nas próximas seções.

3.6.1 Absorção

A primeira fase da função esponja se chama de *absorção* e refere-se ao processamento dos blocos da mensagem de entrada.

Como ilustrada na figura 7 de [FIPS202], o resultado da execução anterior da função f serve como entrada em cada iteração para a próxima execução da função f. Inicialmente, uma *bit string* que contém 0 em todos os seus *bits* é fornecida à primeira iteração. Esta *bit string* (que chamaremos de S a partir de agora) vai se modificando em cada iteração.

O tamanho de S é de b *bits*, onde:

$$b = r + c$$

Onde:

- r: o tamanho de cada bloco da mensagem binária de entrada. Também é chamado de *bitrate*, ou vazão de *bits*, pois representa o número de *bits* consumidos em cada iteração da função esponja.
- c: chamado de *capacidade*, representa o nível de segurança atingido pela função esponja. Quanto maior o numero c, maior a segurança, porém menor o *bitrate*.

Analisando o algoritmo 8 em [FIPS202], pode-se dizer que, em cada iteração, a fase de absorção ocorre da seguinte forma:

- O próximo bloco P_i da mensagem de entrada é preenchido com zeros ($P_i^r \parallel 0^c$) para aumentar seu tamanho de r para b *bits*.
- Ao resultado do passo anterior, uma operação *XOR* é aplicada, tendo como segundo operando o valor de S_{i-1} proveniente da iteração anterior, ou zeros se for a primeira iteração ($S_0^b = 0^{r+c}$).
- O resultado da operação *XOR* serve então de entrada para a função de compressão f. O resultado desta função é o novo valor S_i da variável s, que é usada como entrada da próxima iteração, juntamente com o próximo bloco P_{i+1} da mensagem de entrada.

Se o tamanho desejado l da saída da função esponja é menor que o tamanho de s - ou seja, se $l \leq b$ - então os primeiros l *bits* de s_k - retornado pela última iteração - é o resultado da função esponja. Caso deseja-se $l > b$, então a fase de liberação inicia-se, como descrito na próxima seção.

3.6.2 Liberação

Na fase de absorção da estrutura esponja, descrita na seção anterior, foram consumidos todos os blocos da mensagem de entrada, resultando num valor final S_k de *b bits*. Se o tamanho desejado de saída (l) da função esponja for $l > b$, é preciso “espremer a esponja” para obter uma saída com o número de *bits* desejado.

Observando a figura 7 e analisando o algoritmo 8 de [FIPS202], pode-se descrever esta fase da seguinte forma:

- Primeiramente, os primeiros *r bits* de S_k são colocados num bloco Z_0, o primeiro bloco de Z.
- Então o *bit string* S_k é aplicado à função f para se obter novo *bit string* S_{k+1}.
- Os primeiro *r bits* do novo valor de S_{k+1} são colocados num bloco Z_1.
- Este processo se repete até que se tenha j blocos (Z_0, Z_1, ..., Z_{j-1}) tal que $(j-1) \times r < l \leq j \times r$.

Ao final deste processo, a saída da estrutura esponja serão os primeiros *l bits* dos blocos concatenados $Z_0 \parallel Z_1 \parallel \ldots \parallel Z_{j-1}$.

3.7 Funções Esponja Keccak

As funções esponja Keccak são um grupo de funções definidas para serem usadas na estrutura esponja:

- *pad10*1* é a função de *padding* aplicação à mensagem que servirá de entrada para a estrutura esponja.
- $Keccak[c]$ é a aplicação de $Keccak$-$p[b, n_r]$ como a função f sobre a estrutura esponja com parâmetros b, n_r e r pré-definidos, sendo r baseado em c.

As funções listadas acima são definidas na próximas seções.

3.7.1 *pad10*1*

De acordo com o algoritmo 9 de [FIPS202], *pad10*1* retorna uma *bit string PAD* de tamanho:

$$b - (m \bmod b)$$

Onde:

- b é o tamanho do bloco em *bits* da estrutura esponja.
- m é o tamanho da mensagem de entrada em *bits*.
- *PAD* tem o seguinte formato binário: $1 \parallel 0^{m-2} \parallel 1$

3.7.2 $Keccak[c]$

$Keccak[c]$ é definida da seguinte forma:

$$Keccak[c] = Sponge[Keccak\text{-p}[1600, 24], pad10 * 1, 1600\text{–}\, c]$$

Ou seja, é a aplicação da estrutura esponja, tendo:

- como função f, a função $Keccak$-p com $b = 1600$ e $n_r = 24$.
- *pad10*1* como o algoritmo de *padding*.
- como bloco da mensagem de entrada, uma *bit string* de tamanho $r = b - c$.

Portanto, $Keccak[c]$ vai aceitar uma mensagem de tamanho variável, que será *padded* com *pad10*1*. Cada rodada irá processar um bloco da mensagem de entrada de tamanho $r = b - c$, com uma capacidade (ou segurança) de tamanho $c = b - r$.

3.8 Especificação das Funções SHA-3

[FIPS202] especifica quatro versões da função de *hash* SHA-3 (SHA3-224, SHA3-256, SHA3-384, SHA3-512) e duas versões da função de saída extensível (XOF SHAKE128 e SHAKE256). Estas funções são descritas nas próximas seções.

3.8.1 Funções de *Hash* SHA-3

De forma genérica, pode-se definir uma única função de *hash* SHA-3 da seguinte forma:

$$SHA3[d](M) = Keccak[2d](M \parallel 01, d)$$

Onde:

- M é a mensagem a ser processada. Observa-se a concatenação dos bits “01" à mensagem M antes de aplicar a função $Keccak[c]$.
- d é o tamanho em *bits* do valor de *hash* a ser gerado pela função. Observa-se aqui que a capacidade se Keccak é o dobro do tamanho de d.

De acordo com a forma genérica, podemos definir as quatro versões específicas desta forma:

- $SHA3\text{-}224(M) = SHA3[224](M)$
- $SHA3\text{-}256(M) = SHA3[256](M)$
- $SHA3\text{-}384(M) = SHA3[384](M)$
- $SHA3\text{-}512(M) = SHA3[512](M)$

Portanto, SHA-3 possui funções que podem gerar valor de *hash* nos tamanhos de 224, 256, 384 e 512 *bits*, respectivamente.

3.8.2 Funções de Saída Extensível SHA-3 (XOF)

Além de permitir a geração de valores de *hash* de tamanho fixo, o SHA-3 também permite a geração de valores de tamanho arbitrário através da função genérica:

$$SHAKE[c](M, d) = Keccak[c](M \parallel 1111, d)$$

Onde:

- M é a mensagem a ser processada. Observa-se a concatenação dos bits “1111" à mensagem M antes de aplicar a função $Keccak[c]$.
- d é o tamanho em *bits* do valor de *hash* a ser gerado pela função. Se $d > 1600$, será preciso “espremer a esponja”, como foi descrito nas seções anteriores.
- c é o tamanho da capacidade (ou segurança) da função $Keccak[c]$, que pode ser diferente de d.

Dado a forma genérica acima, podemos definir as duas funções XOF específicas do SHA-3:

- $SHAKE128(M, d) = SHAKE[128](M, d)$
- $SHAKE256(M, d) = SHAKE[256](M, d)$

Portanto, o SHA-3 define apenas duas funções XOF com capacidades de 128 e 256 *bits*, respectivamente.

3.8.3 Separação de Domínios

Terminando as mensagens das funções de *hash* com “01" e das funções XOF com “1111" garante que os domínios destas funções não tenham interseção.

Os últimos *bits* “11" das funções XOF também garantem compatibilidade com o esquema de codificação Sakura, que permite processamento paralelo na geração de valores de *hash* para mensagens longas.

3.9 Análise de Segurança do SHA-3

De acordo com o tipo de ataque, a tabela abaixo mostra a “forca” de segurança em *bits* das diferentes versões das funções SHA-3:

Função	Tamanho Saída	Colisão	Pré-Imagem	Segunda Pré-Imagem
SHA-1	160	< 80	160	160–L(M)
SHA-224	224	112	224	min(224,256–L(M))
SHA-512/224	224	112	224	224
SHA-256	256	128	256	256–L(M)
SHA-512/256	256	128	256	256
SHA-384	384	192	384	384
SHA-512	512	256	512	512–L(M)
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-384	384	192	384	384
SHA3-512	512	256	512	512
SHAKE128	d	min(d/2,128)	≥ min(d,128)	min(d,128)
SHAKE256	d	min(d/2,256)	≥ min(d,256)	min(d,256)

Onde:

$$L(M) = \lceil log2(len(M)/B) \rceil$$

É interessante observar que, quanto maior a capacidade (c), maior a segurança do SHA-3. Isto é evidente nos valores de resistência a colisão e à segunda pré-imagem mostrados na tabela acima.

3.10 Exemplo

Aqui temos um exemplo de uma mensagem de 5 *bits* para a qual gerou-se um *hash* code SHA-3 de 256 *bits*:

3.10.1 Mensagem (Entrada)

```
11001
```

3.10.2 Valor de *Hash* (Saída)

```
7B 00 47 CF 5A 45 68 82 36 3C BF 0F B0 53 22 CF
65 F4 B7 05 9A 46 36 5E 83 01 32 E3 B5 D9 57 AF
```

4 Implementação

4.1 Funções SHA-3

Na código abaixo, estão implementadas todas as funções do SHA3, utilizando-se da função keccak:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "keccak.h"

// Nota: A classe cstream foi copiada do Stack Overflow.
// cstream concatena duas input streams,
// fazendo o padding do delimitador da mensagem.
// Observe que na especificação o padding está em bits,
// enquanto na implementação de referência está em bytes.

template< size_t D >
std::string sha3(std::istream & message)
{
    return keccak< D*2 >(cstream(message, "0x06"), D);
}

std::string sha3_224(std::istream & message)
{
    return sha3<224>(message);
}

std::string sha3_256(std::istream & message)
{
    return sha3<256>(message);
}

std::string sha3_384(std::istream & message)
{
    return sha3<384>(message);
}

std::string sha3_512(std::istream & message)
{
    return sha3<256>(message);
}

template< size_t N >
std::string shake(std::istream & message, size_t digest_bit_size)
{
```

```
    return keccak< N*2 >(cstream(message, "0x1F"), digest_bit_size);
}

std::string shake_128(std::istream & message, size_t digest_bit_size)
{
    return shake<128>(message, digest_bit_size);
}

std::string shake_256(std::istream & message, size_t digest_bit_size)
{
    return shake<256>(message, digest_bit_size);
}
```

4.2 Funções Keccak

Na código abaixo, estão implementadas todas as funções rnd, keccak_f e keccak:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "theta.h"
#include "rho.h"
#include "pi.h"
#include "chi.h"
#include "iota.h"
#include "sponge.h"

template< size_t W >
StateArray< W > rnd(StateArray< W > & a, int i)
{
    return iota(chi(pi(rho(theta(a)))), i);
}

template< size_t B >
BitString< B > keccak_f(BitString< B > bs)
{
    StateArray< B/25 > a { bs };

    for (int i = 0; i < 24; i++)
    {
        a = rnd(a, i);
    }

    return a.bs();
}

template< size_t C >
std::string keccak(std::istream & message, size_t digest_bit_size)
{
    return sponge< 1600, 1600-C, keccak_f >(message, digest_bit_size);
}
```

4.3 Função Esponja

Na código abaixo, está implementada a função esponja:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "bit_string.h"
#include "padding.h"

template< size_t B >
using SpongeF = BitString< B >(*)(BitString< B >);

template< size_t B, size_t R, SpongeF< B > F>
std::string sponge(std::istream & message, size_t digest_bit_size)
{
    static_assert(R >= 8, "Rate must have at least one byte.");
    static_assert(B > R, "State size must be greater than rate.");

    BitString< B > state;
    BitString< R > block;
    BitString< B-R > capacity;

    while (!message.eof())
    {
        message >> block;
        if (message.eof()) {
             pad101(block, block.bits_read());
        }
        state = F(state ^ (block + capacity));
    }

    constexpr size_t hex_bit_size = 4;

    std::string digest = truncate< R >(state).to_hex();
    while (digest_bit_size > (digest.size() * hex_bit_size))
    {
        state = F(state);
        digest += truncate< R >(state).to_hex();
    }

    return digest.substr(0, digest_bit_size / hex_bit_size);
}
```

4.4 Padding

Na código abaixo, está implementada a função pad101:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once
```

```
#include "bit_string.h"

template< size_t R >
void pad101(BitString< R > & block, size_t actual_size)
{
    if (block.size() > actual_size)
    {
        block.set(actual_size, 1);
        block.set(R - 1, 1);
    }
}
```

4.5 Theta

Na código abaixo, está implementada a função Theta:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "state_array.h"
#include <algorithm>
#include <bitset>
#include <iostream>

using namespace std;

template< size_t W >
StateArray< W > theta(const StateArray< W > & a)
{
    using SA = StateArray< W >;
    using Coord2D = typename SA::Coord2D;

    SA b;

    for (Coord2D coord = SA::begin2D(); coord != SA::end2D(); coord.next())
    {
        Coord2D previous_column = coord;
        previous_column.p_cycle_x();

        Coord2D next_column = coord;
        next_column.cycle_x();

        b[coord] = a.column_xor(previous_column.x)
                 ^ a[coord]
                 ^ (rotate(a.column_xor(next_column.x), 1));
    }

    return b;
}
```

4.6 Rho

Na código abaixo, está implementada a função Rho:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "state_array.h"

template< size_t W >
StateArray< W > rho(const StateArray< W > & a)
{
    using SA = StateArray< W >;
    using Coord2D = typename SA::Coord2D;

    SA b;

    Coord2D coord { 0, 0 };
    b[coord] = a[coord];

    coord = Coord2D { 1, 0 };
    for (int t = 0; t < 24; t++)
    {
        const int offset = (t+1)*(t+2)/2;

        b[coord] = rotate(a[coord], offset);

        coord = Coord2D { coord.y, (2 * coord.x + 3 * coord.y) % 5 };
    }

    return b;
}
```

4.7 Pi

Na código abaixo, está implementada a função Pi:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "state_array.h"

template< size_t W >
StateArray< W > pi(const StateArray< W > & a)
{
    using SA = StateArray< W >;
    using Coord2D = typename SA::Coord2D;

    SA b;

    for (Coord2D target { 0, 0 }; target != SA::end2D(); target.next())
    {
        const Coord2D source { (target.x + 3 * target.y) % 5, target.x };
        b[target] = a[source];
```

```
    }

    return b;
}
```

4.8 Chi

Na código abaixo, está implementada a função Chi:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include "state_array.h"

template< size_t W >
StateArray< W > chi(const StateArray< W > & a)
{
    using SA = StateArray< W >;
    using Coord2D = typename SA::Coord2D;

    SA b;

    for (Coord2D target { 0, 0 }; target != SA::end2D(); target.next())
    {
        Coord2D plus_one = target;
        plus_one.cycle_x();

        Coord2D plus_two = plus_one;
        plus_two.cycle_x();

        b[target] = a[target] ^ (~a[plus_one] & a[plus_two]);
    }

    return b;
}
```

4.9 Iota

Na código abaixo, está implementada a função Iota:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include <cassert>
#include "state_array.h"

bool rc_bit(int t)
{
    int n = t % 255;
    if (n == 0) return 1;
```

```
    BitString<9> r { "100000000" };
    for (int i = 0; i < n; i++)
    {
        r >>= 1;
        r.set(0, r[0] ^ r[8]);
        r.set(4, r[4] ^ r[8]);
        r.set(5, r[5] ^ r[8]);
        r.set(6, r[6] ^ r[8]);
    }

    return r[0];
}

template< size_t W >
BitString< W > rc(int round_index)
{
    BitString< W > word;

    const int offset = 7 * round_index;
    size_t i = 1;
    for(int j = 0; i <= W; j++)
    {
        word.set(i - 1, rc_bit(j + offset));
        i *= 2;
    }

    // Observe que na especificação não se fala desta reversão,
    // porém é necessária de acordo com a implementação de referência,
    // que utiliza left-shift para construir a round constant.
    return word.reversed();
};

template< size_t W >
StateArray< W > iota(const StateArray< W > & a, int round_index)
{
    assert(round_index >= 0 && round_index < 24);

    using SA = StateArray< W >;
    using Coord2D = typename SA::Coord2D;

    const Coord2D target = { 0, 0 };

    SA b = a;
    b[target] = a[target] ^ rc< W >(round_index);

    return b;
}
```

4.10 State Array

Na código abaixo, está implementado o `state array`:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.
```

```
#pragma once

#include "bit_string.h"

using namespace std;

#include <climits>

template< size_t W >
struct StateArray
{
    static constexpr size_t row_count = 5;
    static constexpr size_t column_count = row_count;
    static constexpr size_t lane_count = row_count * column_count;
    static constexpr size_t lane_size = W;
    static constexpr size_t string_size = lane_count * lane_size;

    using Lane = BitString< lane_size >;

    // A abstração das coordenadas do state array implementadas abaixo,
    // facilitam a iteração sobre este e simplificam o código
    // das funções de permutação.

    struct Coord2D
    {
        int x, y;

        Coord2D(int x, int y): x(x), y(y) {}

        bool operator !=(const Coord2D & other) const
        {
            return x != other.x || y != other.y;
        }

        void p_cycle_x() { x = (x == 0 ? column_count : x) - 1; }
        void p_cycle_y() { y = (y == 0 ? row_count : y) - 1; }

        void previous()
        {
            p_cycle_x();
            if (x == (column_count - 1)) y--;
        }

        void p_cycle()
        {
            p_cycle_x();
            if (x == (column_count - 1)) p_cycle_y();
        }

        void cycle_x() { x = (x + 1) % column_count; }
        void cycle_y() { y = (y + 1) % row_count; }

        void next()
        {
```

```
        cycle_x();
        if (x == 0) y++;
    }

    void cycle()
    {
        cycle_x();
        if (x == 0) cycle_y();
    }

    int linear_index() const
    {
        return (y * column_count) + x;
    }

    friend inline ostream & operator << (ostream & os, const Coord2D & coord)
    {
        return os << "(" << coord.x << "," << coord.y << ")";
    }

};

struct Coord3D
{
    int x, y, z;

    Coord3D(int x, int y, int z): x(x), y(y), z(z) {}

    void p_cycle_x() { x = (x == 0 ? column_count : x) - 1; }
    void p_cycle_y() { y = (y == 0 ? row_count : y) - 1; }
    void p_cycle_z() { z = (z == 0 ? lane_size : z) - 1; }

    void previous()
    {
        p_cycle_z();
        if (z == (lane_size - 1)) p_cycle_x();
        if (x == (column_count - 1) && z == (lane_size - 1)) y--;
    }

    void p_cycle()
    {
        p_cycle_z();
        if (z == (lane_size - 1)) p_cycle_x();
        if (x == (column_count - 1) && z == (lane_size - 1)) p_cycle_y();
    }

    void cycle_x() { x = (x + 1) % column_count; }
    void cycle_y() { y = (y + 1) % row_count; }
    void cycle_z() { z = (z + 1) % lane_size; }

    void next()
    {
        cycle_z();
        if (z == 0) cycle_x();
        if (x == 0 && z == 0) y++;
```

```
    }

    void cycle()
    {
        cycle_z();
        if (z == 0) cycle_x();
        if (x == 0 && z == 0) cycle_y();
    }

    bool operator !=(const Coord3D & other) const
    {
        return x != other.x || y != other.y || z != other.z;
    }

    int linear_index() const
    {
        return lane_size * ((y * column_count) + x) + z;
    }

    friend inline ostream & operator << (ostream & os, const Coord3D & coord)
    {
        return os << "(" << coord.x << "," << coord.y << "," << coord.z << ")";
    }

};

static Coord3D begin()
{
    return { 0, 0, 0 };
}

static Coord3D end()
{
    return { 0, row_count, 0 };
}

static Coord2D begin2D()
{
    return { 0, 0 };
}

static Coord2D end2D()
{
    return { 0, row_count };
}

StateArray() {}

StateArray(BitString< string_size > s)
{
    for (Coord3D coord = begin(); coord != end(); coord.next())
    {
        this->set(coord, s[coord.linear_index()]);
    }
```

```
    swap_endian();
}

BitString< string_size > bs() const
{
    StateArray st = *this;
    st.swap_endian();

    BitString< string_size > s;
    for (Coord3D coord = begin(); coord != end(); coord.next())
    {
        s.set(coord.linear_index(), st[coord]);
    }

    return s;
}

bool operator [](const Coord3D & coord)
{
    return matrix[coord.x][coord.y][coord.z];
}

const bool operator [](const Coord3D & coord) const
{
    return matrix[coord.x][coord.y][coord.z];
}

Lane & operator [](const Coord2D & coord)
{
    return matrix[coord.x][coord.y];
}

const Lane & operator [](const Coord2D & coord) const
{
    return matrix[coord.x][coord.y];
}

void set(const Coord3D & coord, const bool & bit)
{
    matrix[coord.x][coord.y].set(coord.z, bit);
}

void XOR(const Coord3D & left_coord, const Coord3D & right_coord)
{
    set(left_coord, (*this)[left_coord] xor (*this)[right_coord]);
}

Lane column_xor(int column_index) const
{
    Coord2D coord = { column_index, 0 };
    Lane result = (*this)[coord];

    while (++coord.y < row_count)
    {
        result ^= (*this)[coord];
```

```
        }

        return result;
    }

    std::string to_string() const
    {
        return bs().to_string();
    }

    std::string to_hex() const
    {
        StateArray st = *this;
        st.swap_endian();

        std::string str;
        for (Coord2D coord = begin2D(); coord != end2D(); coord.next())
        {
            str += st[coord].to_hex();
        }

        return str;
    }

private:

    // Na implementação de referência, se faz a troca de bytes mais significativos
    // com os menos significativos.
    // Porém, não há qualquer informação sobre isto na especificação da FIPS.
    void swap_endian()
    {
        if (lane_size % BitString< W >::byte_size == 0)
        {
            for (Coord2D coord = begin2D(); coord != end2D(); coord.next())
            {
                (*this)[coord].swap_endian();
            }
        }
    }

    Lane matrix[column_count][row_count];
};
```

4.11 Bit String

Na código abaixo, está implementado a `bit string`:

```
// Copyright (c) 2016 Quenio Cesar Machado dos Santos. All rights reserved.

#pragma once

#include <bitset>
#include <string>
#include <istream>
```

```
#include <sstream>
#include <iomanip>
#include <cassert>

// Neste arquivo, somente a implementação de swap_endian foi copiada do Stack Overflow:
template < typename T >
T swap_endian(T u)
{
    static_assert (CHAR_BIT == 8, "CHAR_BIT != 8");

    union
    {
        T u;
        unsigned char u8[sizeof(T)];
    } source, dest;

    source.u = u;

    for (size_t k = 0; k < sizeof(T); k++)
    dest.u8[k] = source.u8[sizeof(T) - k - 1];

    return dest.u;
}

// Nota: classe BitString abstrai as operações
// sobre sequencias de bits,
// assim permitindo uma implementação de Keccak
// para tamanhos arbitrários
// de state array e capacidade.
// Apesar disto, a performance é adequada,
// pois utiliza a classe bitset da standard library,
// que é implementada em memória e registradores
// para state array de até 64 bits.

template< size_t N >
struct BitString
{
    static constexpr size_t byte_size = 8;

    BitString() {}
    BitString(uint64_t val): _bs(val) {}
    BitString(const std::string & s): _bs(s.c_str()) {}
    BitString(const char * s): _bs(s) {}
    BitString(std::bitset< N > bs): _bs(bs) {}

    std::string to_string() const { return _bs.to_string(); }

    std::string to_hex() const
    {
        constexpr size_t block_size = 64;
        const std::string bin_str = to_string();

        std::stringstream ss;
        for (int i = 0; i < size(); i += block_size)
        {
```

```
        std::bitset< block_size > bs64
        {
            bin_str.substr(i, block_size).c_str()
        };
        const size_t bit_width = (i + block_size) > size() ? size() - i : block_size;
        ss << std::setfill('0')
           << std::setw(bit_width/4)
           << std::uppercase
           << std::hex << bs64.to_ulong();
    }

    return ss.str();
}

size_t count() const { return _bs.count(); }
size_t size() const { return _bs.size(); }

BitString& operator &= (const BitString& rhs)
{
    _bs &= rhs._bs; return *this;
}

BitString& operator |= (const BitString& rhs)
{
    _bs |= rhs._bs; return *this;
}

BitString& operator ^= (const BitString& rhs)
{
    _bs ^= rhs._bs; return *this;
}

BitString& operator <<= (size_t pos)
{
    _bs <<= pos; return *this;
}

BitString& operator >>= (size_t pos)
{
    _bs >>= pos; return *this;
}

BitString& set() { _bs.set(); return *this; }

BitString& set(size_t pos, bool val = true)
{
    _bs.set(reversed(pos), val); return *this;
}

BitString& reset() { _bs.reset(); return *this; }
BitString& reset(size_t pos) { _bs.reset(reversed(pos)); return *this; }

BitString operator~() const { return BitString(~_bs); }

BitString& flip() { _bs.flip(); return *this; }
```

```
BitString& flip(size_t pos){ _bs.flip(reversed(pos)); return *this; }

bool operator [] (size_t pos) { return _bs[reversed(pos)]; }
const bool operator [] (size_t pos) const { return _bs[reversed(pos)]; }

bool operator == (const BitString& rhs) const { return _bs == rhs._bs; }
bool operator != (const BitString& rhs) const { return _bs != rhs._bs; }

bool test(size_t pos) const { return _bs.test(reversed(pos)); }
bool all() const { return _bs.all(); }
bool any() const { return _bs.any(); }
bool none() const { return _bs.none(); }

BitString operator << (size_t pos) const { return BitString(_bs << pos); }
BitString operator >> (size_t pos) const { return BitString(_bs >> pos); }

BitString reversed() const
{
    std::string str = to_string();
    std::reverse(str.begin(), str.end());
    return BitString(str.c_str());
}

void swap_endian()
{
    if (_bs.size() % byte_size == 0)
    {
        _bs = ::swap_endian(_bs.to_ullong());
    }
}

friend inline BitString operator & (const BitString & lhs, const BitString & rhs)
{
    return BitString(lhs._bs & rhs._bs);
}

friend inline BitString operator | (const BitString & lhs, const BitString & rhs)
{
    return BitString(lhs._bs | rhs._bs);
}

friend inline BitString operator ^ (const BitString & lhs, const BitString & rhs)
{
    return BitString(lhs._bs ^ rhs._bs);
}

template < size_t L, size_t R >
friend inline BitString< L+R > operator + (const BitString< L > & lhs, const BitString< R >
{
    return BitString< L+R >(lhs._bs.to_string() + rhs._bs.to_string());
}

template< size_t O, size_t I >
friend BitString< O > truncate(const BitString< I > & input);
```

```
    friend inline std::istream & operator >> (std::istream & is, BitString & bstr)
    {
        constexpr size_t byte_size = 8;
        constexpr size_t block_size = N / byte_size;

        std::stringstream ss;

        bstr._bits_read = 0;
        for (size_t i = 0; i < block_size; i++)
        {
            unsigned char value;
            is >> value;
            if (is.eof())
            {
                ss << "00000000"; // one byte
            }
            else
            {
                std::bitset< byte_size > bs = value;
                ss << bs.to_string();
                bstr._bits_read += byte_size;
            }
        }

        bstr._bs = std::bitset< N > { ss.str() };

        return is;
    }

    size_t bits_read() { return _bits_read; }

private:
    // bitset tem indice zero no digit mais a direita,
    // enquanto BitString precisa que este encontre-se na esquerda.
    size_t reversed(size_t pos) const { return N - pos - 1; }

    std::bitset< N > _bs;
    size_t _bits_read;
};

template< size_t O, size_t I >
inline BitString< O > truncate(const BitString< I > & input)
{
    return BitString< O >(input.to_string());
}

inline std::string hex_to_bin(std::string hex_str)
{
    constexpr int byte_size = 2;

    assert(hex_str.size() % byte_size == 0);

    std::string bin_str;
    for (int i = 0; i < hex_str.size(); i += byte_size)
    {
```

```
        std::stringstream ss(hex_str.substr(i, byte_size));

        unsigned int value;
        ss >> std::hex >> value;
        std::bitset<8> bs = value;

        bin_str += bs.to_string();
    }

    return bin_str;
}

template< size_t N >
inline BitString< N > hex_to_bs(std::string hex_str)
{
    return BitString< N >(hex_to_bin(hex_str).c_str());
}

template < size_t N >
inline BitString< N > rotate(BitString< N > b, int n)
{
    return b << (n % N) | b >> ((N-n) % N);
}
```

5 Referências

[**Stalling2014**] Cryptography and Network Security, Sixth International Edition, 2014.
[**FIPS202**] FIPS PUB 202, 2015.
[**Wikipedia2016**] Wikipedia, 2016.