Instituto de Computação
Universidade Estadual de Campinas

Implementação em *software* de algoritmos de resumo criptográfico

Thomaz Eduardo de Figueiredo Oliveira

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Thomaz Eduardo de Figueiredo Oliveira e aprovada pela Banca Examinadora.

Campinas, 29 de junho de 2011.

Prof. Dr. Julio César López Hernández
IC/Universidade Estadual de Campinas
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

OL4i

Oliveira, Thomaz Eduardo de Figueiredo
Implementação em software de algoritmos de resumo criptográfico/Thomaz Eduardo de Figueiredo Oliveira--Campinas, [S.P. : s.n.], 2011.

Orientador : Julio César López Hernández.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Criptografia. 2.Hashing (Computação). 3.Arquitetura de computadores. I. López Hernández, Julio César. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Software implementation of cryptographic hash algorithms

Palavras-chave em inglês (Keywords):
Cryptography
Hashing (Computer science)
Computer architecture

Área de concentração: Teoria da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:
Prof. Dr. Julio César López Hernández (IC – UNICAMP)
Prof. Dr. Ricardo Dahab (IC – UNICAMP)
Dr. José Antônio Carrijo Barbosa (CEPESC-ABIN)

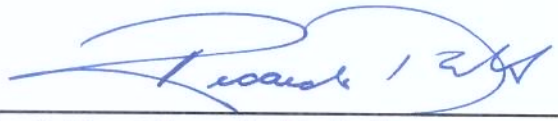
Data da defesa: 16/06/2011

Programa de Pós-Graduação: Mestrado em Ciência da Computação

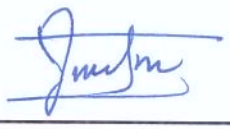
TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 16 de junho de 2011, pela Banca examinadora composta pelos Professores Doutores:

Dr. José Antônio Carrijo Barbosa
CEPESC / ABIN



Prof. Dr. Ricardo Dahab
IC / UNICAMP



Prof. Dr. Julio César López Hernández
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Implementação em *software* de algoritmos de resumo criptográfico

Thomaz Eduardo de Figueiredo Oliveira[1]

16 de junho de 2011

Banca Examinadora:

- Prof. Dr. Julio César López Hernández
  IC/Universidade Estadual de Campinas (Orientador)
- Prof. Dr. Ricardo Dahab
  IC/Universidade Estadual de Campinas
- Dr. José Antônio Carrijo Barbosa
  CEPESC/Agência Brasileira de Inteligência
- Prof. Dr. Paulo Lício de Geus (Suplente)
  IC/Universidade Estadual de Campinas
- Prof. Dr. Routo Terada (Suplente)
  IME/Universidade de São Paulo

[1]Suporte financeiro de: CNPq (processo 133033/2009-0) 2009–2011.

Agradecimentos

Meus agradecimentos vão primeiramente para minha família: Ana Beatriz, Silma e Sebastião, que mesmo distantes fisicamente, me apoiaram de todas as formas possíveis neste caminho.

Agradeço ao orientador Julio López pelo excelente trabalho em me guiar por um ambiente até então desconhecido por mim.

Devo agradecer também a todos os membros do LCA que me auxiliaram durante a pesquisa, além da Unicamp e o CNPq pelo apoio estrutural e financeiro.

Por fim, não posso de deixar de lembrar dos amigos de todas as partes, estejam eles próximos ou distantes, que propiciaram momentos de descontração tão necessários para a realização deste trabalho.

Meu filho pequeno me pergunta: devo aprender matemática?
Para quê, penso em dizer. Que dois pedaços de pão são mais do que um
Você logo notará.

Meu filho pequeno me pergunta: devo aprender francês?
Para quê, penso em dizer. Esse império está no fim. E
Basta você esfregar a mão na barriga e gemer:
Logo lhe compreenderão.

Meu filho pequeno me pergunta: devo aprender história?
Para quê, penso em dizer. Aprenda a enfiar sua cabeça na terra
Talvez então você escape.

Sim, aprenda matemática, digo.
Aprenda francês, aprenda história!

Bertolt Brecht

Resumo

Os algoritmos de resumo criptográfico são uma importante ferramenta usada em muitas aplicações para o processamento seguro e eficiente de informações. Na década de 2000, sérias vulnerabilidades encontradas em funções de resumo tradicionais, como o SHA-1 e o MD5, levou a comunidade a repensar o desenvolvimento da criptanálise destes algoritmos e projetar novas estratégias para a sua construção. Como resultado, o instituto NIST anunciou em novembro de 2007 um concurso público para o desenvolvimento de um novo padrão de funções de resumo, o SHA-3, contando com a participação de autores de todo o mundo. Esta dissertação foca nos aspectos da implementação em *software* de alguns algoritmos submetidos no concurso SHA-3, buscando compreender a forma como os autores desenvolveram a questão do custo computacional de seus projetos em diversas plataformas, além de entender os novos paradigmas de implementação introduzidos pela tecnologia presente nos processadores atuais. Como consequência, propusemos novas técnicas algorítmicas para a implementação em *software* de alguns algoritmos, como o Luffa e o Keccak, levando aos mesmos melhorias significativas de desempenho.

Abstract

Hash algorithms are an important tool of cryptography used in many applications for secure and efficient information processing. During the 2000 decade, serious vulnerabilities found at some traditional hash functions like SHA-1 and MD5 prompted the cryptography community to review the advances in the cryptanalysis of these algorithms and their design strategies. As a result, on November, 2007, NIST announced a public competition to develop a new cryptographic hash function, the SHA-3, which involved competitors throughout the world. This work focuses on the software implementation aspects of some of the SHA-3 submitted algorithms, seeking to comprehend how the authors resolved the computational cost issues at distinct platforms and to understand the new paradigms introduced by the present processors technology. As a consequence, we proposed new algorithmic techniques for the software implementation of Luffa and Keccak hash algorithms, improving their performance significantly.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

As funções de resumo criptográfico são importantes ferramentas para a realização de atividades por meio eletrônico, além de serem essenciais para a concretização de esquemas relacionados à criptografia assimétrica. Seu principal objetivo é identificar um certo dado de maneira única, por meio de uma cadeia de *bits* finita.

O estudo destas funções nos permite compreender um grande escopo de algoritmos simétricos, pois conforme veremos nos capítulos seguintes, muitos projetos são baseados em cifras de bloco e de fluxo, e em algumas vezes, os autores as utilizam diretamente para construírem seus algoritmos de resumo.

No momento atual, esta sub-área da criptografia passa por grandes dificuldades, pois algoritmos tradicionais, amplamente utilizados nas aplicações e protocolos atuais, correm o risco de serem ultrapassados ou quebrados. Desta forma, a comunidade vê a urgente necessidade de revisar os princípios e estratégias básicas para a construção destas funções, assim como considerar novos paradigmas. Esta mudança de direção se reflete nos novos desenhos que fazem parte do concurso internacional para a escolha de um novo padrão mundial, o *SHA-3*.

Neste cenário, através da implementação em *sotfware* pretendemos investigar como os novos algoritmos do concurso SHA-3 se adaptam às tendências tecnológicas dos novos processadores assim como tornam viáveis a implementação em plataformas consolidadas e de ambiente restrito. Assim, realizaremos implementações em ambientes *desktop*, de 64 e 32 *bits*, usando também a tecnologia *SSE* que permite realizar paralelismo de dados em vetores de 128 *bits*. Além disso, trabalharemos com plataformas de 8 *bits*, que possuem recursos computacionais e de armazenamento mais limitados.

Por fim, pretendemos também buscar maneiras de explorar as características e recursos providos por estas arquiteturas para gerar códigos mais eficientes e/ou compactos, contribuindo para a otimização dos novos algoritmos e auxiliando a comunidade criptográfica na escolha do novo integrante da família SHA.

1.1 Definições dos símbolos

Os seguintes símbolos matemáticos foram utilizados neste trabalho.

$\wedge$	A operação binária "e"
$\vee$	A operação binária "ou"
$\neg$	A negação binária
$\oplus$	A operação ou-exclusivo
$\|\|$	A operação de concatenação
$x \ll n$	Deslocamento à esquerda da estrutura x em n *bits*
$x \gg n$	Deslocamento à direita da estrutura x em n *bits*
$x \lll n$	Rotação à esquerda da estrutura x em n *bits*
$x \ggg n$	Rotação à direita da estrutura x em n *bits*
$\boxplus$	Adição modular[1]
$\otimes$	Multiplicação modular[2]
0^n	Sequência de n *bits* 0
$\|x\|_n$	O tamanho da estrutura x codificada em n *bits*

1.2 Organização do documento

Esta dissertação está organizada da seguinte forma. No Capítulo 2 é realizada uma breve introdução das funções de resumo criptográficas, suas aplicações e suas classes. O Capítulo 3 descreve com maiores detalhes os algoritmos que fazem parte do padrão norte-americano *SHA*. No Capítulo 4, faz-se um breve resumo das arquiteturas usadas neste trabalho. O Capítulo 5 apresenta um tipo de construção de funções de resumo chamada função esponja, além de demonstrar sua aplicabilidade com algoritmos concretos. Nos Capítulos 6 e 7 são mostradas otimizações realizadas em algoritmos de resumo recentes, usando tecnologias disponíveis nos processadores atuais. O Capítulo 8 descreve os algoritmos candidatos a serem o novo padrão mundial, além de apresentar comentários sobre suas implementações. Por fim, o trabalho é concluído no Capitulo 9.

[1] O valor do módulo dependerá do tamanho da palavra utilizada no algoritmo.
[2] Os parâmetros do módulo dependerão das especificações do algoritmo.

Capítulo 2

Funções de resumo criptográfico

2.1 Introdução

As funções de resumo criptográfico mapeiam uma cadeia de *bits* de tamanho arbitrário para uma cadeia de *bits* de tamanho fixo. Elas são usadas em aplicações de segurança como esquemas de assinatura digital, identificação de dados, derivação de chaves, entre outros. Seus principais objetivos são prover integridade de dados e autenticação de mensagens.

A cadeia de *bits* gerada por estas funções são chamadas de resumo, que na prática, deve identificar a mensagem de maneira única.

Definição. *Uma função de resumo* $h : M \to R$ *mapeia cadeias de* bits $m \in M$ *de tamanho finito e arbitrário para cadeia de* bits $r \in R$ *de tamanho fixo* n*. Deste modo,* $r = h(m)$*.*

Dado que o domínio M de h é maior do que sua imagem R, é fácil perceber que mais de uma mensagem será mapeada para o mesmo resumo. Algumas aplicações necessitam que seja computacionalmente inviável que um atacante encontre duas mensagens aleatórias que gerem o mesmo resumo, por exemplo, no contexto de assinaturas digitais; outras apenas requerirão que seja inviável computacionalmente encontrar uma mensagem dado o conhecimento do resumo.

2.1.1 Propriedades

Para alcançar os objetivos citados no início desta seção, são estabelecidas propriedades básicas que as funções de resumo devem possuir: resistência à pré-imagem, segunda pré-imagem e colisão.

Resistência à pré-imagem: dada uma função $h : M \mapsto R$ e um resumo $r \in R$, é computacionalmente inviável encontrar $m \in M$ tal que $h(m) = r$.

Resistência à segunda pré-imagem: dada uma função $h : M \mapsto R$ e uma mensagem $m \in M$, é computacionalmente inviável encontrar $m' \in M$ tal que $m' \neq m$ e $h(m') = h(m)$.

Resistência à colisão: dada uma função $h : M \mapsto R$, é computacionalmente inviável encontrar $m, m' \in M$ tal que $m' \neq m$ e $h(m') = h(m)$.

Além das três propriedades, os algoritmos de resumo criptográfico devem processar a mensagem de forma eficiente.

Modelo do oráculo randômico

Introduzido por Mihir Bellare e Phillip Rogaway [14], o modelo do oráculo randômico é uma idealização de uma função de resumo criptográfico ideal.

Neste modelo, não existem fórmulas nem algoritmos para definir a função de resumo, pois esta é representada por um oráculo, e a única forma de obter resumos é através da requisição ao mesmo. Em [36] foi provada a impossibilidade de se instanciar um oráculo randômico, porém, tenta-se quantificar a distância na qual um algoritmo real está do modelo do oráculo randômico. Se o algoritmo for considerado uma caixa-preta, o termo usado é indistinguibilidade de um oráculo. Caso contrário, usa-se o termo indiferenciabilidade.

2.1.2 Aplicações

Armazenamento de senhas

Uma forma de se prover autenticação em um sistema é através de senhas, que são associadas a cada entidade autorizada a acessar determinados recursos. As senhas podem ser armazenadas em um arquivo com proteção para leitura e escrita, porém este método permite com que intrusos ou usuários com acesso total ao sistema possam ter acesso às senhas dos usuários.

Para impedir tal vulnerabilidade, são calculados os resumos das senhas e estes, por sua vez, são armazenados no arquivo de senhas. Para verificar a autenticidade do usuário, o sistema calcula o resumo da senha inserida e confere com a entrada no arquivo de senhas.

Ataques a este método são possíveis com o pré-calculo dos resumos de todas as senhas possíveis por parte do atacante, tornando-se imediata a identificação da senha, uma vez que o mesmo tenha posse do arquivo de senhas. Para dificultar tal ataque, é introduzida uma palavra randômica de n *bits* denominada *salt* que é utilizada juntamente com a senha para calcular o resumo e armazenada juntamente com a mesma. Com isso, o atacante

precisa possuir 2^n variações da tabela pré-calculada das senhas, o que consumiria mais memória e mais tempo para preparar o ataque.

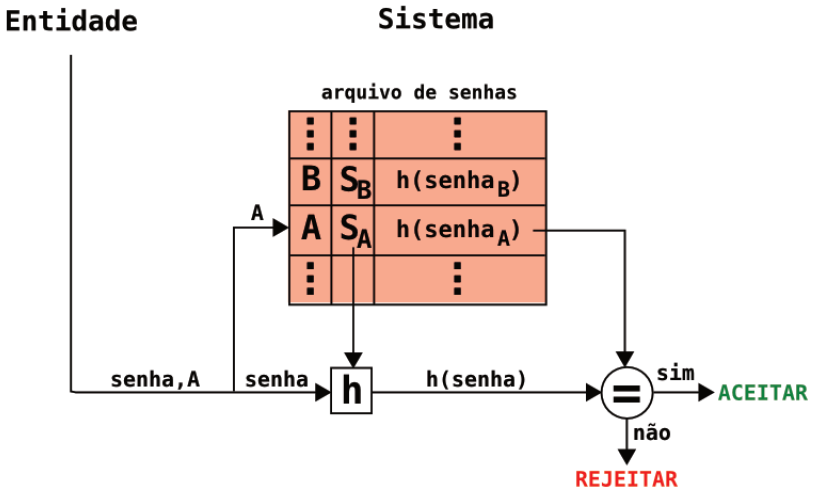


Figura 2.1: Esquema para verificar senhas utilizando-se funções de resumo [108]

Integridade de aplicações

Semelhante ao armazenamento de senhas, as funções de resumo criptográfico também podem ser usadas para garantir a integridade de aplicativos. Esta técnica é realizada através do cálculo do resumo sobre o binário da aplicação. Este resumo deve então ser protegido contra modificações, por meio de um *MAC* por exemplo (ver Seção 2.6). Desta forma, qualquer alteração na aplicação refletiria imediatamente no valor do seu resumo.

Por exemplo, o TSE (Tribunal Superior Eleitoral) do Brasil calcula e publica em sua página eletrônica os resumos dos programas contidos nas urnas eletrônicas que fazem parte do processo eleitoral do país. Estes resumos são assinados antes de que as urnas sejam distribuídas para os estados nacionais [53].

Derivação de chaves

Métodos para estabelecer chaves entre duas ou mais entidades dado o conhecimento de uma senha em comum são chamados de *acordo de chaves autenticado por senha*. Protocolos como o *SPAKE* (*Simple Password Exponential Key Exchange*) [82] permitem que entidades estabeleçam senhas através da troca de mensagens em um canal inseguro. Isto é alcançado através do uso de funções de resumo aplicadas na senha e do método de troca de chaves *Diffie-Hellman* [50].

Suponha que Alice e Beto dividem uma senha S, p um primo grande próprio para

uso no método Diffie-Hellman, $f(S)$ uma função que converte S para uma base Diffie-Hellman, $E_k(m)$ uma função de encriptação simétrica de m usando uma chave k e h uma função de resumo criptográfico.

1. Alice e Beto escolhem números randômicos R_A e R_B, respectivamente,
2. Alice computa $Q_A = f(S)^{R_A} \mod p$ e envia para Beto,
3. Beto computa $Q_B = f(S)^{R_B} \mod p$ e envia para Alice,
4. Alice computa $K = h({Q_B}^{R_A} \mod p)$,
5. Beto computa $K = h({Q_A}^{R_B} \mod p)$,
6. Alice escolhe um número randômico C_A, computa $E_K(C_A)$ e envia para Beto,
7. Beto escolhe um número randômico C_B, computa $E_K(C_B, C_A)$ e envia para Alice,
8. Alice verifica que C_A está correto, computa $E_K(C_B)$ e envia para Beto,
9. Beto verifica que C_B está correto.

Esquemas de assinaturas digitais

Diversos métodos de assinaturas digitais utilizam-se de criptografia de chave pública, porém, a encriptação de dados por meio da criptografia assimétrica é muito custosa computacionalmente. Desta forma, para efetuar a assinatura de documentos, calcula-se o resumo destes e então realiza-se a assinatura. A empresa *RSA Security* publicou uma série de processos para direcionar o uso de seu algoritmo *RSA* em operações de chave pública. Estes processos são denominados *PKCS* (*Public-key Cryptography Standards*) [85], entre os quais, estão descritas técnicas para assinar e verificar assinaturas.

Processo para assinar Dada a mensagem M, o expoente privado d e o módulo n próprios do algoritmo RSA,

1. Calcular o resumo criptográfico da mensagem M, gerando $h(M)$,
2. Codificar $h(M)$,
3. Formatar o resumo $h(M)$ codificado,
4. Calcular a assinatura RSA de $h(M)$: $s = h(M)^d \mod n$,
5. Transformar o inteiro s para octetos S.

Processo para verificar Dada a mensagem M, a assinatura S, o expoente público e e o módulo n próprios do algorítimo RSA,

1. Converter a assinatura S para o número inteiro s,
2. Computar m no algoritmo RSA: $m = s^e \mod n$,
3. Converter m para octetos EB,
4. Decodificar EB, gerando MD,
5. Calcular o resumo criptográfico de M, gerando $h(M)$,
6. Aceitar a assinatura S em M se e somente se $h(M) = MD$.

2.2 Análise e ataques

Para medir a segurança de um algoritmo criptográfico, é possível seguir duas abordagens, a concreta e a assintótica. A primeira abordagem determina a probabilidade de um adversário quebrar o sistema a partir do tempo de computação. Mesmo com as constantes evoluções tecnológicas relacionadas à velocidade de processamento e aumento de paralelismo, esta abordagem é utilizada, porém, deve-se ter precaução na sua interpretação pois tanto as exigências da comunidade quanto as sustentações de segurança dos projetistas podem se tornar obsoletas.

A abordagem assintótica é a mais usada pela comunidade criptográfica, transformando a probabilidade e o tempo de computação do adversário em funções de um parâmetro n. Desta forma, a impossibilidade de se quebrar um sistema criptográfico é determinada a partir de apenas um parâmetro, que no caso das funções de resumo criptográfico, é o tamanho do resumo.

A segurança das propriedades das funções de resumo definidas na Seção 2.1.1 estão resumidas na tabela 2.1.

Devido ao paradoxo do aniversário [160], o ataque à colisão é teoricamente mais plausível de se realizar. Estima-se que computadores atuais conectados à redes distribuídas podem processar 2^{60} instruções em 2 anos [87], portanto, resumos de 128 *bits* presentes em funções tradicionais como *MD4* e *MD5* (ver Seção 2.5.2) já estão comprometidos. Desta forma, o valor mínimo de resumo considerado seguro, no momento da elaboração deste trabalho, é de 160 *bits*.

Tabela 2.1: Segurança das funções de resumo

Propriedade	Resistência ideal
Pré-imagem	2^n
Segunda pré-imagem	2^n
Colisão	$2^{n/2}$

Variantes Para entender melhor as vulnerabilidades de um algoritmo, atacantes promovem variações nos mesmos. Estas variações consistem principalmente em modificar o estado inicial do algoritmo, que recebe o valor IV (do inglês *Initialization Vector*), que é normalmente fixado pelo projetista. O fato do IV ser fixo é premissa para a sustentação de segurança de diversas construções, como a *Merkle-Damgård* por exemplo (ver Seção 2.3.1). Apresentamos a seguir dois ataques que exploram esta característica.

Pseudocolisão ou colisão de início livre [108]: Seja h uma função de resumo, V e V' dois estados iniciais distintos, encontrar duas mensagens m e m' tal que $m \neq m'$ e $h(V, m) = h(V', m')$.

Colisão de início semi-livre [108]: Seja h uma função de resumo, V um estado inicial qualquer, encontrar duas mensagens m e m' tal que $m \neq m'$ e $h(V, m) = h(V, m')$.

É importante ressaltar que as variantes acima definidas podem ser aplicadas também a ataques de pré-imagem.

Variáveis de cadeia Os ataques descritos acima não têm como premissa a natureza iterativa das funções de resumo reais. Nestas, a cada iteração, a variável do estado, também chamado de variável de cadeia, é processada juntamente com um bloco de mensagem em cada iteração, resultando em uma nova variável de estado. Em seguida, alguns ataques relacionados às variáveis de cadeia.

Ataque de correção de bloco [108]: Neste ataque uma mensagem m', tal que $m' \neq m$, é modificada ao longo do processamento para produzir um resumo $h(m') = h(m)$. Em casos mais comuns, o bloco a ser modificado é o primeiro ou o último.

Ataque *meet-in-the-middle* [108]: Este método realiza um ataque de colisão entre variáveis de cadeia intermediárias. O atacante escolhe um ponto intermediário H_i da

iteração do algoritmo e calcula $H_i = h(H_{i-1}, m_i)$ e $H_i = h^{-1}(H_{i+1}, m_{i+1})$, tentando encontrar uma colisão para H_i. Em caso de um sucesso, o adversário produz uma colisão para toda a mensagem.

Ataque de ponto fixo [108]: dada uma função de resumo h, o atacante tenta encontrar um par (H_{i-1}, m_i) tal que $h(H_{i-1}, m_i) = H_{i-1}$. Neste caso, o adversário pode produzir colisões infinitas, apenas concatenando blocos de mensagem m_i em sequência.

Ataques nas funções de compressão Alguns ataques são concentrados nas funções de compressão dos algoritmos. Estas funções funcionam semelhantemente a cifras simétricas de bloco, que são há bastante tempo estudadas. Como consequência, os projetistas precisam se precaver de um grande número de ataques.

Ataques por meio do oráculo randômico Com a publicação do modelo do oráculo randômico, novos ataques foram construidos com o objetivo de distinguir a função de um oráculo. Apesar de não significar uma quebra concreta do sistema, podem gerar futuros ataques práticos.

2.3 Funções de resumo iteradas

Desde as primeiras propostas para construir funções de resumo criptográficas [125], os autores usam a iteração para processar as mensagens e gerar o valor de resumo. Basicamente, a mensagem é dividida em t blocos de tamanhos idênticos $x_1, x_2, \ldots, x_t$ (o último bloco é completado com *bits* se necessário) e o resumo é calculado da seguinte forma,

$$
\begin{aligned}
&H_0 = IV;\\
&H_i = f(x_i, H_{i-1}), \qquad 1 \leq i \leq t;\\
&h(x) = g(H_t).
\end{aligned}
$$

Onde IV é o valor inicial do estado e $g(x)$ uma transformação sobre o valor final, gerando assim o resumo. A função f é chamada de função de compressão, pois a partir de uma entrada de tamanho n gera uma saída de tamanho $r < n$.

2.3.1 Merkle-Damgård

Ralph Merkle e Ivan Damgård [111] estabeleceram condições para que as funções iteradas garantissem resistência à colisão: fixar o valor IV e inserir o tamanho da mensagem processada no último bloco. Estas condições passaram a se chamar reforço Merkle-Damgård (do inglês *Merkle-Damgård strengthening*).

O método Merkle-Damgård assegura que se a função de compressão é resistente à colisão, o algoritmo também será. Desta forma, a meta dos projetistas se reduz a construir uma função de compressão segura.

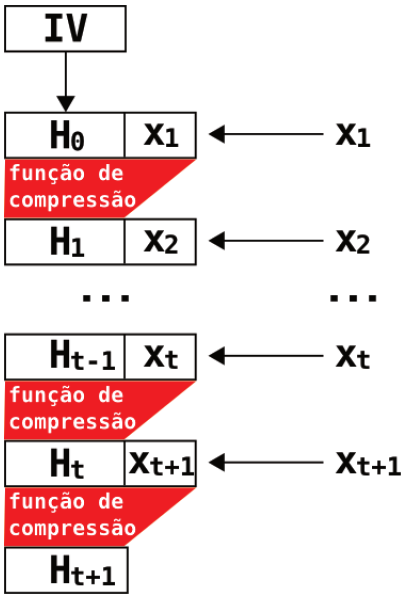


Figura 2.2: Método Merkle-Damgård [139]

Algoritmo do método Merkle-Damgård Dada uma função de compressão $f : \{0,1\}^{n+r} \rightarrow \{0,1\}^n$ que mapeia entradas de $(n + r)$ *bits* para saídas de n *bits* e uma mensagem m, com tamanho b *bits*,

1. Quebrar a mensagem m em t blocos $(x_1, x_2, \ldots, x_t)$ de r *bits* cada, completando o último bloco com *bits* 0 se necessário;

2. Definir um bloco extra x_{t+1} que compreenderá a representação binária de b, ou seja, o tamanho da mensagem;

3. Seja $H_0 = 0^n$, definir $H_i = f(H_{i-1} \| x_i)$, para $1 \leq i \leq t + 1$. O valor do resumo de n *bits* será calculado como $h(x) = H_{t+1} = f(H_t \| x_{t+1})$.

2.3.2 HAIFA

Com a descoberta de vulnerabilidades na construção Merkle-Damgård como ataques de multicolisões e de ponto fixo [86, 90], os pesquisadores Eli Biham e Orr Dunkelman introduziram a construção *HAIFA* (*HAsh Interactive FrAmework*) [28]. A proposta dos

autores consiste em melhoras no método Merkle-Damgård com a inserção de parâmetros nas funções de compressão, diferenciando as várias iterações durante o processo do cálculo do resumo.

Basicamente, HAIFA sugere a inserção de dois parâmetros extras nas funções de compressão: o número de *bits* computados até a chamada da função e um valor *salt* (ver Seção 2.1.2). Assim, enquanto no modelo Merkle-Damgård, a função de compressão era definida como $H_i = f(H_{i-1}, mensagem)$, na construção HAIFA, passa a ser definida como $H_i = f(H_{i-1}, mensagem, \#bits, salt)$.

2.3.3 Esponja

Outro modelo recente é a *construção esponja*, criada por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche [23]. Nesta construção, a função de compressão passa a ser substituída por uma função de permutação. Com este novo modelo, os autores objetivaram facilitar a análise e projeto dos algoritmos, pois ele permite maior aproximação com o modelo do oráculo randômico (ver Seção 2.1.1). Além disso, funções de permutação são mais simples de se projetar se comparadas com funções de compressão.

Neste modelo, uma função de resumo criptográfico é construída a partir de consecutivas iterações de uma função de permutação f, que processa um número fixo n de *bits*. Os *bits* processados são divididos em duas partes: r, a taxa de *bits* (do inglês *bit rate*); esta parte receberá os dados da mensagem, e c, a capacidade (do inglês *capacity*), que determina o nível de segurança da construção. Portanto, $n = r + c$. A construção esponja é dividida em duas etapas: absorção (do inglês *absorbing*) e extração (do inglês *squeezing*).

Absorção

Dada uma função de permutação $f : \{0,1\}^n \to \{0,1\}^n$ e uma mensagem M de m *bits*,

1. Dividir a mensagem M em t blocos de r *bits* $(x_1, x_2, \ldots, x_t)$;
2. Preencher estado inicial H_0 (IV) com n *bits* 0;
3. Fazer $H_i = f(H_{i-1} \oplus x_i)$ para $1 \leq i \leq t$, ou seja, a cada passo da iteração, será feito uma operação ou-exclusivo com um bloco da mensagem M e os primeiros r *bits* do estado atual.

Extração

Dada uma função permutação $f : \{0,1\}^n \to \{0,1\}^n$,

1. Extrair r *bits* iniciais do estado H_t, gerando p_0;

2. Até que se recupere os *bits* necessários para gerar o resumo, fazer: $H_{t+i} = f(H_{t+i-1})$ e extrair r *bits* de H_{t+i}, gerando p_i.

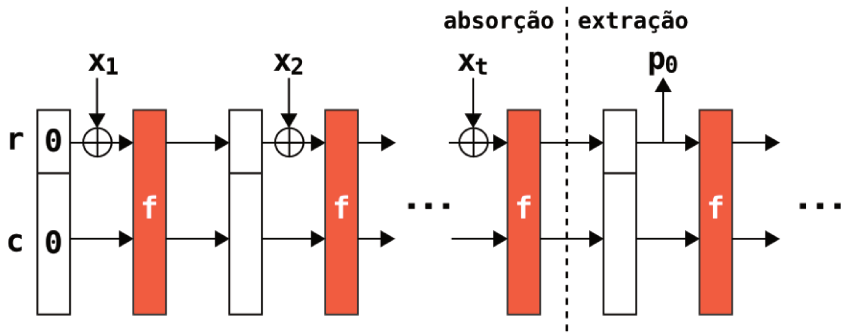


Figura 2.3: Construção Esponja

Uma vez escolhido o tamanho da entrada para a função de permutação, o projetista tem liberdade em escolher os valores da taxa de *bits* r e da capacidade c. Quanto maior for o valor r, mais rápido será o algoritmo, pois processará maior volume de dados por iteração, e quanto maior for c, maior será o nível de segurança. Maiores detalhes desta construção serão discutidos no Capítulo 5.

2.4 Funções de resumo paralelas

Em 1987, o pesquisador Ralph Merkle publicou um método de se gerar assinaturas digitais por meio de funções de encriptação já existentes na época [110, 109], como o *DES* (*Data Encryption Standard*) por exemplo. Com isso, seria possível evitar paradigmas cuja dificuldade computacional em se quebrar ainda não era provada matematicamente, como o método RSA baseado na fatoração numérica.

A estrutura usada para alcançar o método sugerido por Merkle recebeu o nome de Árvore Merkle (do inglês *Merkle-Tree*), e hoje é usada também para calcular resumos criptográficos de forma paralela.

Basicamente, a Árvore Merkle funciona da seguinte forma: primeiramente, a mensagem é repartida em blocos de tamanhos fixos, logo, cada bloco é separadamente processado e gera um resumo. Em seguida, os resumos são unidos em grupos de n, e estes grupos por sua vez, serão separadamente processados. Este processo se repete até que se forme

apenas um valor de resumo.

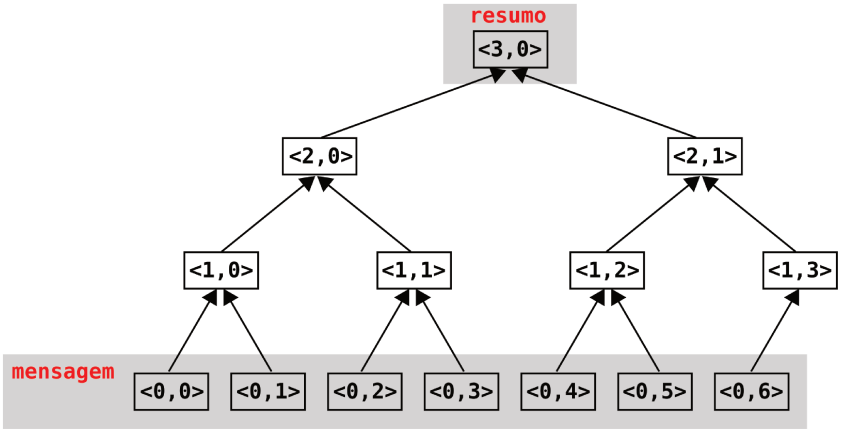


Figura 2.4: A Árvore Merkle com o agrupamento de dois resumos.
O texto sobre os nós representa <altura do nó , número do nó>.

Os projetistas de algoritmos aconselham inserir parâmetros juntamente com as mensagem dos tipos: número do nó, altura da árvore, entre outros, para evitar ataques de colisão.

Com o advento da computação paralela através de processadores de múltiplos núcleos e *GPUs* (*Graphics Processing Unit*), este método pode vir a ser muito usado. A desvantagem da técnica é a necessidade de memória para armazenar os nós intermediários.

2.5 Classificações de funções de resumo

2.5.1 Baseadas em cifras de bloco

As primeiras formas de se gerar funções de resumo criptográfico foram através de cifras de bloco, pois estas haviam sido estudadas por um tempo considerável e algoritmos eficientes já estavam disponíveis. Porém, as cifras de bloco não são funções de sentido único, ou seja, dado o conhecimento da chave, é fácil computar a função inversa. Além disso, os esquemas de geração de subchaves presentes nestas cifras podem gerar fenômenos indesejáveis para uma função de resumo.

Desta forma, após análises e provas de segurança [155, 124], foram criadas construções por meio das quais é possível projetar algoritmos de função de resumos a partir de cifras de bloco.

Matyas-Meyer-Oseas [106]

Dada uma cifra de bloco E_K que processa n *bits*, parametrizada por uma chave K de k *bits*, uma função g que mapeia dados para chaves K, um valor inicial IV e uma mensagem m,

1. A mensagem m é dividida em t blocos de n *bits* $(x_1, x_2, \ldots, x_t)$, completando o último bloco se necessário;
2. Computar, $H_0 = IV$, $H_i = E_{g(H_{i-1})}(x_i) \oplus x_i$, para $1 \leq i \leq t$.

Davies-Meyer [154]

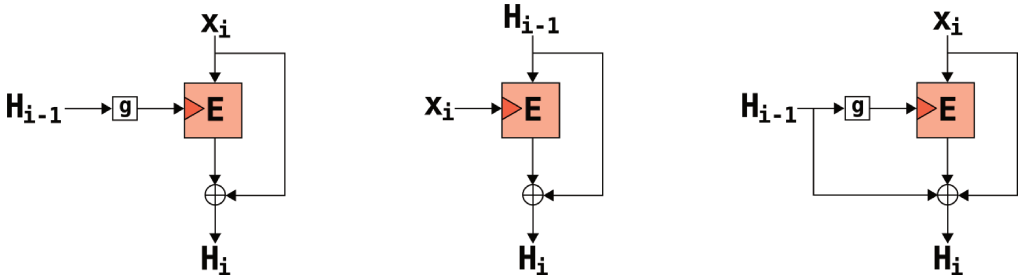
Dada uma cifra de bloco E_K que processa n *bits*, parametrizada por uma chave K de k *bits*, um valor inicial IV e uma mensagem m,

1. A mensagem m é dividida em t blocos de k *bits* $(x_1, x_2, \ldots, x_t)$, completando o último bloco se necessário;
2. Computar, $H_0 = IV$, $H_i = E_{x_i}(H_{i-1}) \oplus H_{i-1}$, para $1 \leq i \leq t$.

Miyaguchi-Preneel [123]

Dada uma cifra de bloco E_k que processa n *bits*, parametrizada por uma chave K de k *bits*, uma função g que mapeia dados para chaves K, um valor inicial IV e uma mensagem m,

1. A mensagem m é dividida em t blocos de n *bits* $(x_1, x_2, \ldots, x_t)$, completando o último bloco se necessário;
2. Computar, $H_0 = IV$, $H_i = E_{g(H_{i-1})} \oplus x_i \oplus H_{i-1}$, para $1 \leq i \leq t$.



(a) M.-M.-Oseas (b) D.-Meyer (c) Miyaguchi-Preneel

Figura 2.5: Construções baseadas em cifras de bloco

2.5.2 Funções de resumo dedicadas

Funções de resumo dedicadas são algoritmos projetados especificamente com o objetivo de gerarem resumos criptográficos. Muitos destes algoritmos foram criados para serem eficientes em *software*, ao contrário das primeiras funções de resumo baseadas em cifras de bloco.

Os algoritmos desta classe se tornaram muito populares, principalmente o MD4, MD5 e a família SHA. Estes projetos também foram bastante analisados pela comunidade criptográfica, e muitos deles foram quebrados, porém ainda são usados em aplicativos e protocolos importantes [101, 102].

MD4 e MD5

Os algoritmos MD4 e MD5 (*Message-Digest algorithm*) [128, 129] foram criados por Ronald Rivest em 1990 e 1991 respectivamente. Estes projetos influenciaram diretamente os algoritmos SHA-1 e *RIPEMD*. Ainda que quebrados, é possível encontrar aplicações que os utilizam em funções que não exijam todas as propriedades básicas das funções de resumo criptográfico.

MD4 O algoritmo MD4 é projetado em uma construção Merkle-Damgård, portanto, seu núcleo consiste em uma função de compressão. Este projeto trabalha com palavras de 32 *bits* e recebe como entrada um bloco de mensagem de 512 *bits* e uma variável de estado de 128 *bits*, gerando como saída outra variável de 128 *bits*.

Antes de definir a função de compressão (ver Algoritmo 1), é preciso estabelecer três importantes funções usadas durante o processo de compressão.

Sejam x, y e z palavras de 32 *bits*,

$F(x, y, z) = xy \vee \neg(x)z;$
$G(x, y, z) = xy \wedge xz \wedge yz;$
$H(x, y, z) = x \oplus y \oplus z.$

Por possuir apenas operações simples, trabalhar com palavras de 32 *bits* e executar as funções repetidas vezes de maneira estruturada, o algoritmo MD4 tem uma performance bastante satisfatória se implementado em *software*. Porém, pouco tempo após sua publicação, vulnerabilidades foram reportadas. Sua resistência à colisão foi quebrada em 1996 [51] e em 2008 publicou-se um ataque em relação à sua resistência à pré-imagem

[98].

Algoritmo 1: O algoritmo MD4

Entrada: Mensagem M dividida em N blocos de 512 *bits*
Saída: Resumo criptográfico de 128 *bits*
Sejam A,B,C e D palavras de 32 *bits* inicializadas da seguinte forma,
$A \leftarrow$ `0x01234567`, $B \leftarrow$ `0x89ABCDEF`, $C \leftarrow$ `0xFEDCBA98`, $D \leftarrow$ `0x76543210`

para *i=0* **até** *N-1* **faça**
`// Copiar palavras dos blocos da mensagem M`
para *j=0* **até** *15* **faça**
$X[j] \leftarrow M[i \times 16 + j]$
fim

`// Guardar valores iniciais do estado`
$AA \leftarrow A$, $BB \leftarrow B$, $CC \leftarrow C$, $DD \leftarrow D$

`// Rodada 1`
Seja $[a, b, c, d, k, s] = (a + F(b, c, d) + X[k]) \lll s$,
[A,B,C,D,0,3], [D,A,B,C,1,7], [C,D,A,B,2,11], [B,C,D,A,3,19]
[A,B,C,D,4,3], [D,A,B,C,5,7], [C,D,A,B,6,11], [B,C,D,A,7,19]
[A,B,C,D,8,3], [D,A,B,C,9,7], [C,D,A,B,10,11], [B,C,D,A,11,19]
[A,B,C,D,12,3], [D,A,B,C,13,7], [C,D,A,B,14,11], [B,C,D,A,15,19]

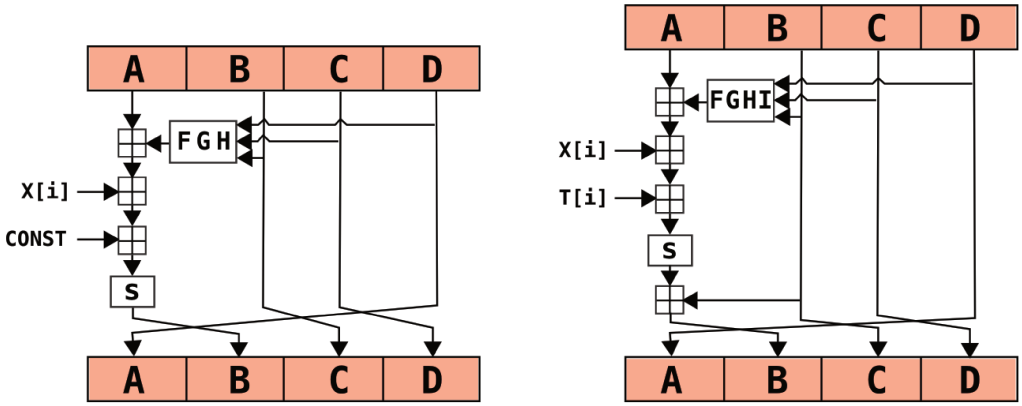
`// Rodada 2`
Seja $[a, b, c, d, k, s] = (a + G(b, c, d) + X[k] +$ `0x5A827999`$) \lll s$,
[A,B,C,D,0,3], [D,A,B,C,4,5], [C,D,A,B,8,9], [B,C,D,A,12,13]
[A,B,C,D,1,3], [D,A,B,C,5,5], [C,D,A,B,9,9], [B,C,D,A,13,13]
[A,B,C,D,2,3], [D,A,B,C,6,5], [C,D,A,B,10,9], [B,C,D,A,14,13]
[A,B,C,D,3,3], [D,A,B,C,7,5], [C,D,A,B,11,9], [B,C,D,A,15,13]

`// Rodada 3`
Seja $[a, b, c, d, k, s] = (a + H(b, c, d) + X[k] +$ `0x6ED9EBA1`$) \lll s$,
[A,B,C,D,0,3], [D,A,B,C,8,9], [C,D,A,B,4,11], [B,C,D,A,12,15]
[A,B,C,D,2,3], [D,A,B,C,10,9], [C,D,A,B,6,11], [B,C,D,A,14,15]
[A,B,C,D,1,3], [D,A,B,C,9,9], [C,D,A,B,5,11], [B,C,D,A,13,15]
[A,B,C,D,3,3], [D,A,B,C,11,9], [C,D,A,B,7,11], [B,C,D,A,15,15]

$A \leftarrow A + AA$, $B \leftarrow B + BB$, $C \leftarrow C + CC$, $D \leftarrow D + DD$
fim
retorna *A,B,C,D*

MD5 Após suspeitas de vulnerabilidades no algoritmo MD4, o algoritmo MD5 foi publicado com algumas modificações básicas em relação ao seu predecessor. Para uma especificação mais detalhada, ver [129].

1. Uma nova rodada foi adicionada. Desta forma o algoritmo MD5 possui quatro rodadas.
2. Cada passo dentro de cada rodada inclui a adição de constantes distintas. As constantes são representadas por $T[1 \ldots 64]$, onde $T[i]$ é a parte inteira de $4294967296 \times |sen(i)|$.
3. A função G agora é representada por $G(x, y, z) = xz \vee y\neg(z)$. Como consequência da adição da quarta rodada, uma nova função foi definida: $I(x, y, z) = y \oplus (x \vee \neg(z))$
4. O resultado da função aplicada nas palavras do passo anterior é somado na palavra A do passo atual.
5. A ordem de acesso dos blocos da mensagem pelas funções são modificadas nas rodadas 2 e 3.
6. O valor dos deslocamentos nas funções foi alterado. Cada rodada possui distintos valores de deslocamento.



(a) MD4 (b) MD5

Figura 2.6: Um passo dos algoritmos MD4 e MD5

O algoritmo também sofreu muitos ataques, sendo quebrado no início dos anos 2000 [148]. Em 2009 um grupo de pesquisadores conseguiu produzir um certificado falso usando o algoritmo MD5 [138], desta forma, o mesmo é considerado impróprio para uso.

SHA

O padrão SHA (*Secure Hash Algorithm*) consiste em uma família de algoritmos de resumo. Seu integrante mais usado é o algoritmo SHA-1, desenvolvido pela agencia norte-americana *NSA* (*National Security Agency*) baseado nos algoritmos MD4 e MD5. Mais detalhes desta família serão vistos no Capítulo 3.

RIPEMD

O algoritmo RIPEMD (*Race Integrity Primitives Evaluation Message Digest*) foi publicado em 1996 por Hans Dobbertin, Antoon Bosselaers e Bart Preneel como parte de um programa europeu para integração das comunicações em banda larga. O projeto é fortemente baseado nos algoritmos MD4 e MD5, porém, estudos na época já estavam cientes da necessidade de produzir valores maiores de resumo para garantir a resistência à colisão. Desta forma, o algoritmo sofreu alterações para gerar resumos de 160 *bits*, sendo denominado RIPEMD-160 [52]. Com isso, os autores ofereceram o projeto como alternativa ao padrão norte-americano SHA-1.

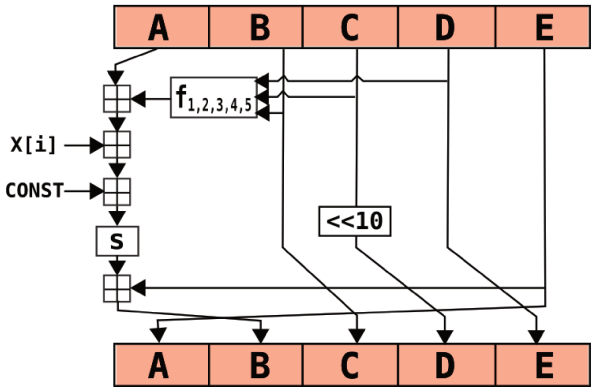


Figura 2.7: Um passo do algoritmo RIPEMD-160

O RIPEMD também é projetado em uma construção Merkle-Damgård. Sua função de compressão trabalha com palavras de 32 *bits* e recebe como entrada um bloco de mensagem de 16 palavras e um estado com 5 palavras, gerando uma variável de 5 palavras. Durante o processo de compressão, as palavras do estado percorrem duas colunas, onde passarão por cinco rodadas distintas.

A cada rodada, diferentes constantes serão adicionadas, além disso, as palavras da mensagem sofrerão permutações particulares (ρ e π) a cada rodada. Por fim, os resultados são somados com os valores iniciais do estado.

Como são cinco rodadas presentes no algoritmo, cinco funções são usadas durante o processo de compressão:

$f_1(x, y, z) = x \oplus y \oplus z;$
$f_2(x, y, z) = (x \wedge y) \vee (\neg(x) \wedge z);$
$f_3(x, y, z) = (x \vee \neg(y)) \oplus z;$
$f_4(x, y, z) = (x \wedge z) \vee (y \wedge \neg(z));$
$f_5(x, y, z) = x \oplus (y \vee \neg(z)).$

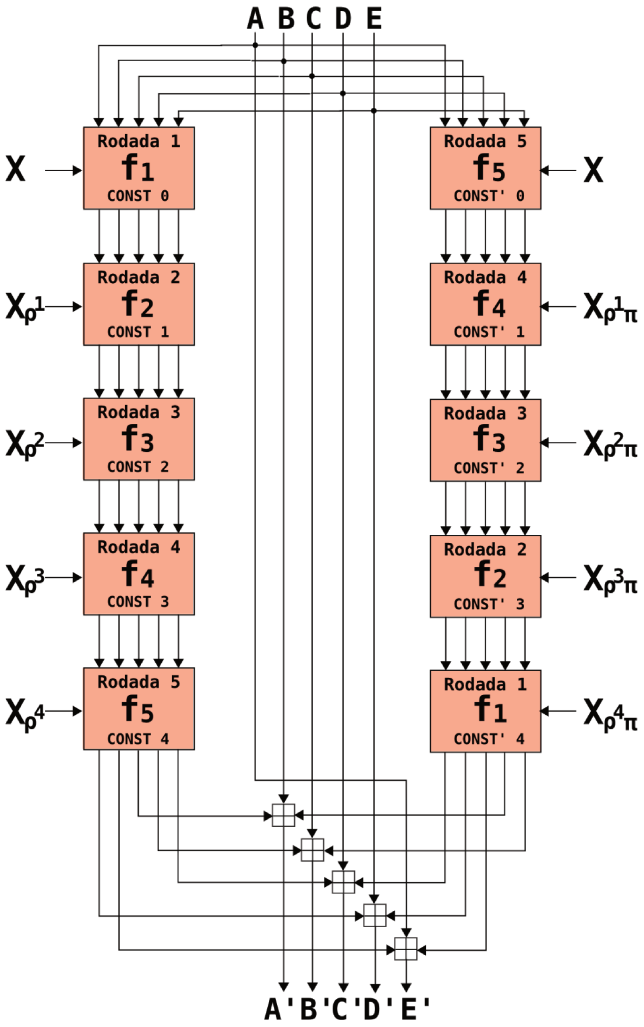


Figura 2.8: O algoritmo RIPEMD-160 [52]

Diferentemente do padrão americano SHA, o algoritmo RIPEMD foi desenvolvido abertamente pela comunidade acadêmica, porém, o primeiro foi mais usado pelo público.

Consequentemente, o projeto RIPEMD não foi devidamente analisado por criptoanalistas, e até o momento, não foi quebrado.

Whirlpool

Whirlpool foi publicado em 2000 por Paulo Barreto e Vincent Rijmen [12], e apesar de sua construção ser baseada em uma cifra de bloco similar ao *AES* (*Advanced Encryption Standard*) denominada *W*, esta foi modificada para ser usada especialmente nesta função de resumo, portanto, podemos classificá-lo nesta seção. Após a correção de algumas vulnerabilidades, o algoritmo foi adotado como padrão *ISO* (*International Organization for Standardization*) [78].

O projeto utiliza-se do método Miyaguchi-Preneel descrito na Seção 2.5.1, trabalhando com blocos de mensagem de 512 *bits*, que são transformados em uma matriz de 8×8, onde cada elemento representa um *byte*, antes de serem processados.

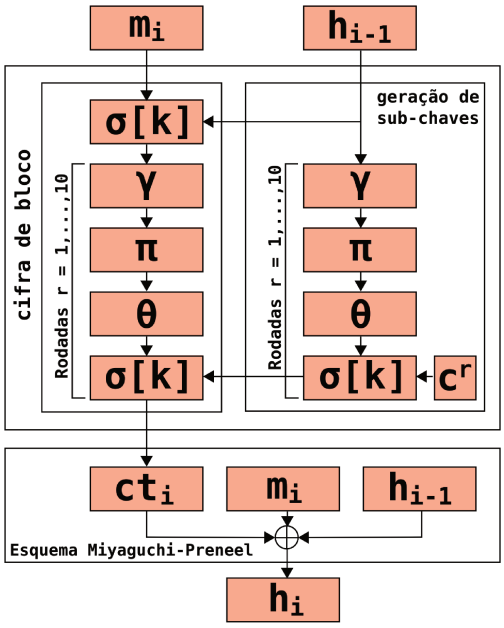


Figura 2.9: O algoritmo Whirlpool [133]

Uma rodada do algoritmo de cifra de bloco é dividido em quatro partes, denominadas $\sigma[k], \theta, \pi$ e γ. $\sigma[k]$ representa a adição das subchaves por meio da operação ou-exclusivo. Estas subchaves são geradas através da variável de estado anterior.

A função π é uma permutação cíclica, onde cada coluna da matriz é rotacionada por um valores distintos:

$$\pi(x) = y \Leftrightarrow y_{ij} = x_{(i-j) \mod 8,j}, \qquad 0 \leq i, j \leq 7.$$

θ realiza uma difusão linear através da multiplicação de sua entrada por uma matriz constante C:

$\theta(x) = y \Leftrightarrow y = x.C.$

A função γ representa a camada não linear, esta é composta por uma *S-box* (*Substituition Box*) contendo componentes menores E, E^{-1} e R, que processam quatro *bits* cada.

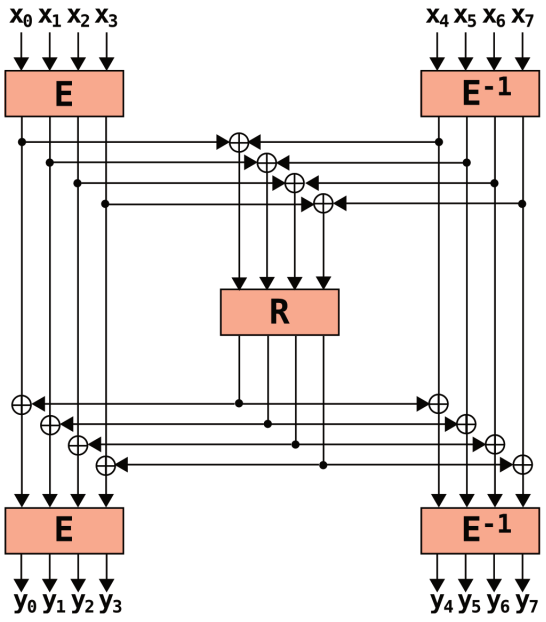


Figura 2.10: A função não linear γ de Whirlpool [12]

O algoritmo Whirlpool é considerado um algoritmo lento se implementado em *software*, porém, com o advento de novas tecnologias, novas implementações surgiram nos últimos anos [73, 133], fazendo com que esta função possivelmente seja incorporada em aplicações futuras.

2.5.3 Baseadas em primitivas aritméticas

Dada a existência de implementações em *software* e *hardware* de operações de aritmética modular, alguns projetistas almejaram construir funções de resumo criptográfico baseadas nestas primitivas. Esta seria uma forma de construir funções cuja segurança fosse apoiada por princípios da teoria dos números. No entanto, muitos algoritmos não tiveram sucesso, tanto por vulnerabilidades de segurança quanto pelo desempenho indesejável.

MASH-1

Após alterações em seu projeto o algoritmo *MASH-1* (*Modular Arithmetic Secure Hash*) [108] foi uma das poucas funções de resumo baseadas em primitivas aritméticas a resistir a ataques até o momento, sendo então adotado como padrão ISO [79]. Basicamente, o projeto se baseia no problema da fatoração, através do uso de módulos RSA, cujo tamanho determina o nível de segurança.

De forma simplificada, o algoritmo MASH-1 é descrito da seguinte forma:

Seja N um módulo RSA cuja fatorização é difícil, M uma mensagem dividida em blocos $x_1, x_2, \ldots, x_t$ e uma constante $A = \texttt{0xF00...0}$, calcular,

$H_0 = 0$;
$H_i = ((x_i \oplus H_{i-1}) \vee A)^2 \mod N) \oplus H_{i-1}$.

VSH

Em 2005, os pesquisadores Contini, Lenstra e Steinfeld publicaram o algoritmo *VSH* (*Very Smooth Hash*) [43], que baseia sua resistência à colisão na dificuldade em encontrar raízes modulares não triviais de determinada classe de números designada pelos autores. O resumo é calculado da seguinte forma.

Seja um inteiro n grande (os autores sugerem que um n de 1024 *bits* equivale à segurança do SHA-1 em relação às colisões), k o tamanho do bloco, onde k é o maior inteiro que satisfaça $\prod_{i=1}^{k} p_i < n$ e m uma mensagem de l *bits* $(m_0, m_1, \ldots, m_l)$.

1. $x_0 = 1$;

2. $\mathcal{L} = \lceil l/k \rceil$. $m_i = 0$ para $l < i < \mathcal{L}k$;

3. Fazer $l = \sum_{i=1}^{k} l_i 2^{i-1}$ com $l_i \in \{0, 1\}$ a representação binária do tamanho da mensagem l e definir $m_{\mathcal{L}k+i} = l_i$ para $1 \leq i \leq k$;

4. Para $j = 0, 1, \ldots \mathcal{L}$, computar $x_{j+1} = x_j^2 \times \prod_{i=1}^{k} p_i^{m_{j.k+i}} \mod n$;

5. Retornar $x_{\mathcal{L}+1}$.

Deve-se ressaltar que o VSH não pode ser um substituto para um oráculo randômico, pois o algoritmo é resistente apenas às colisões [43, 70]. Além disso, ainda é considerado bastante lento, processando aproximadamente 8,8 MB por segundo.

ECOH

O algoritmo *ECOH* (*Elliptic Curve Only Hash*), construido por Brown, Antipa, Campagna e Struik [35], foi submetido em 2008 para o concurso SHA-3 (ver Seção 3.4). O projeto é baseado em outro algoritmo denominado *MuHASH* e possui sua segurança fundamentada no problema do logaritmo discreto em curvas elípticas. O resumo é calculado da seguinte forma.

Sejam *blen*, *ilen* e *clen* os tamanhos dos blocos de mensagem, índice e contador respectivamente, M uma mensagem e G um ponto fixo em uma curva elíptica,

1. Seja $N = M\|1\|0^j$, com j o menor inteiro não negativo que faça N ser divisível por *blen*;
2. Dividir N em k blocos $N_0, \ldots, N_{k-1}$ de *blen bits*;
3. Fazer $O_i = N_i\|I_i$, onde I_i é a representação do inteiro i em *ilen bits*;
4. Fazer $O_k = (\bigoplus_{i=0}^{k-1} N_i)\|I_{mlen}$;
5. Fazer $X_i = (0^{m-(blen+ilen+clen)}\|O_i\|0^{clen}) \oplus C_i$ onde C_i é uma cadeia de m *bits* que representa o menor inteiro não negativo c no qual X_i represente a coordenada x (x_i) de um elemento de $\langle G \rangle$;
6. Fazer $P_i = (x_i, y_i)$ tal que o *bit* mais à direita de y_i/x_i seja igual ao *bit* mais à esquerda de N_i;
7. Fazer $Q = \sum_{i=0}^{k} P_i$;
8. Retornar a representação em n *bits* de $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \mod 2^n$.

Quanto ao desempenho o algoritmo possui os mesmos problemas de seus similares, sendo mil vezes mais lento do que o SHA-1 em longas mensagens [35]. A segurança do projeto foi comprometida por ataques à segunda pré-imagem [68], retirando o algoritmo das fases seguintes do concurso SHA-3.

2.6 MACs: Message Authentication Codes

Os códigos de autenticação de mensagens são funções de resumo criptográfico com chaves, cujo objetivo principal é prover autenticação de dados, mas garante ao mesmo tempo a integridade dos dados processados. Assim, distintas propriedades básicas são definidas para algoritmos MAC.

Facilidade de computação: dada uma função h_k, uma chave K e uma mensagem de tamanho arbitrário e finito m, $h_K(m)$ é fácil de computar. O resultado desta função é chamado de valor MAC.

Compressão: dada uma função h_k, uma chave K e uma mensagem de tamanho arbitrário e finito m, a função $h_K(m)$ gera um resumo de tamanho fixo n.

Resistência à computação: dados zero ou mais pares mensagem-MAC $(m_i, h_K(m_i))$, é computacionalmente inviável computar qualquer par mensagem-MAC $(m, h_K(m))$ para qualquer mensagem $m \neq m_i$ sem o conhecimento da chave K.

Os métodos mais comuns de se construir algoritmos MAC são através de cifras de bloco e por meio de algoritmos de resumo criptográfico sem chave.

2.6.1 HMAC

HMAC (*Hash-based Message Authentication Code*) é um método para gerar MACs a partir de algoritmos de resumo criptográfico sem chave [97]. A técnica foi padronizada pelo *NIST* (*National Institute for Standards and Technology*) em 2002 e hoje é usada em importantes protocolos como *IPSec* [101] e *TLS* [49].

Dada uma função de resumo h, uma chave K, uma mensagem m e duas constantes pré-definidas $ipad =$ `0x363636...36` e $opad =$ `0x5C5C5C...5C`,

1. Computar, $HMAC_K(m) = h((K \oplus opad) \| h((K \oplus ipad) \| m))$.

É importante notar que o tamanho do valor MAC será o mesmo da função de resumo criptográfico h.

2.6.2 CMAC

O esquema *CMAC* (*Cipher-based MAC*), criado pelos pesquisadores John Black, Philip Rogaway, Tatsu Iwata e Kaoru Kurosawa é uma variante da técnica *CBC-MAC* (*Cipher Block Chaining Message Authentication Code*), que possui limitações relacionadas à segurança [34]. O método permite a construção de MACs através de cifras de bloco e é recomendado pelo instituto americano NIST juntamente com o algoritmo padrão de cifra de bloco AES, sendo denominado portanto *AES-CMAC* [137].

O método é dividido em duas fases, a geração de subchaves e a geração do valor MAC.

Geração de subchaves K1 e K2

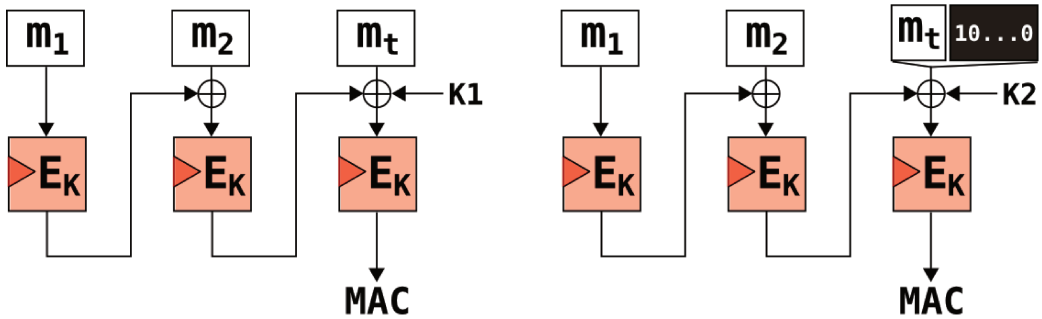
Dada uma cifra de bloco E_k que processe blocos de b *bits* e uma chave K, e uma constante R_b definida como $R_b \leftarrow 0^{120}10000111$ se $b = 128$ ou $R_b \leftarrow 0^{59}11011$ se $b = 64$,

1. Calcular $L = E_K(0^b)$;
2. Se o *bit* mais significante de $L = 0$ então $K1 \leftarrow L \ll 1$, senão, $K1 \leftarrow (L \ll 1) \oplus R_b$;
3. Se o *bit* mais significante de $K1 = 0$ então $K2 \leftarrow K1 \ll 1$, senão, $K2 \leftarrow (K1 \ll 1) \oplus R_b$.

Geração do valor MAC

Dada uma cifra de bloco E_k que processe blocos de b *bits*, duas subchaves $K1$ e $K2$ criadas no processo anterior, T_{mac} o tamanho do valor MAC e uma mensagem m de tamanho T_{msg},

1. Se $T_{msg} = 0$ então $n \leftarrow 1$, senão, $n \leftarrow \lceil T_{msg}/b \rceil$;
2. Dividir m em $(x_1, x_2, \ldots, x_{t-1}, x_t)$, onde $(x_1, x_2, \ldots, x_{t-1})$ são blocos de tamanho b;
3. Se x_t é um bloco de tamanho b então $x_t = x_t \oplus K1$, senão, $x_t = K2 \oplus (x_t \| 10^j)$, onde $j = nb - T_{msg} - 1$;
4. $C_0 = 0^b$;
5. Para $i = 0$ até n, $C_i = E_K(C_{i-1} \oplus x_i)$;
6. Definir o valor MAC como os T_{mac} *bits* mais significativos de C_n.



(a) Último bloco de m completo (b) Último bloco de m incompleto

Figura 2.11: Algoritmo CMAC [137]

O processo de verificação é idêntico ao processo de geração do valor MAC, uma vez de posse da mensagem e da chave, basta executar o processo acima e comparar o MAC gerado com valor MAC recebido junto com a mensagem.

Capítulo 3

O padrão SHA

3.1 Introdução

A autenticação de dados como premissa necessária para a concretização das atividades eletrônicas, sejam elas governamentais ou comerciais, já era conhecida por Diffie e Hellman em [50]. Com o advento da tecnologia as transações eletrônicas passaram a ser viáveis, porém, era necessário o estabelecimento de um padrão de autenticação por meio de assinaturas digitais, e como integrante deste, um algoritmo padrão de resumos criptográficos.

No início da década de 90, o governo dos Estados Unidos através de sua agência governamental NSA, projeta o algoritmo *DSA* (*Digital Signature Algorithm*) [59], para assinaturas digitais, e a função de resumo SHA (*Standard Hash Algorithm*) [58]. Ambos são publicados através do NIST e tomados como padrão a serem utilizados no território norte-americano.

Alguns anos mais tarde, o algoritmo SHA passa a ser usado em diversas aplicações e protocolos por todo o mundo, e acaba se tornando na prática um padrão internacional. Este fato fez com que o projeto se tornasse foco de pesquisadores da comunidade criptográfica, resultando em uma quantidade significativa de análises de segurança. Com a descoberta de vulnerabilidades e o aumento do poder computacional, novos algoritmos foram adicionados ao padrão SHA, chegando ao final da década de 2000 com cinco algoritmos e um concurso para a escolha de mais um projeto.

Nesta seção iremos detalhar os algoritmos que compõem o padrão SHA, assim como descrever o processo da escolha do novo projeto que fará parte desta família em 2012.

3.2 SHA-1

Em 1993, o NIST anuncia um algoritmo padrão para funções de resumo criptográfico denominado *Standard Hash Algorithm*, projetado pela NSA e baseado nos princípios do algoritmo MD4 de Ronald Rivest. Pouco tempo após a publicação, devido a problemas de segurança, a NSA anuncia a anulação do algoritmo e, em 1995, outro algoritmo é publicado, o SHA-1. O primeiro algoritmo passa a ser chamado pela comunidade de *SHA-0*, e sua diferença para o SHA-1 é pequena, apenas uma rotação na etapa inicial do processamento das palavras que compõem um bloco da mensagem. A agencia NSA não explicita a razão desta atualização, porém, análises posteriores do SHA-0 identificaram suas vulnerabilidades [26].

3.2.1 Descrição

O desenho do algoritmo SHA-1 é baseado na construção Merkle-Damgård, aceita mensagens de até 2^{64} *bits* e produz resumos de 160 *bits*. Sua função de compressão trabalha com palavras de 32 *bits*, recebe como entrada um bloco de mensagem de 512 *bits* e uma variável de estado de 160 *bits*, gerando como saída outra variável de 160 *bits*.

O algoritmo é dividido em duas etapas, o preenchimento da mensagem (do inglês, *message padding*) e a computação do resumo.

Preenchimento da mensagem

Antes de processar a mensagem, os seguintes passos são realizados,

1. Adicionar um *bit* "1" no final da mensagem;
2. Adicionar *bits* "0" até que o tamanho da mensagem seja congruente a 448 $\mod 512$;
3. Concatenar 64 *bits* contendo a representação binária do tamanho da mensagem original.

Esta etapa corresponde ao reforço Merkle-Damgård (ver Seção 2.3.1).

Computação do resumo

Inicialmente são definidas três funções lógicas a serem usadas durante o processo de compressão.

Função Ch $Ch(x, y, z) = (x \wedge y) \vee (\neg(x) \wedge z)$. É uma função não linear, onde o resultado da função será y se $x = 1$ e z caso contrário.

Função Parity $Parity(x, y, z) = x \oplus y \oplus z$ É uma função linear, cujo resultado será um ou-exclusivo entre todas os elementos x, y e z.

Função Maj $Maj(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ É uma função não linear, onde o valor de retorno será aquele que for maioria entre os elementos x, y e z.

O bloco de 512 *bits* da mensagem é dividido em 16 palavras de 32 *bits*, estas por sua vez, sofrem uma expansão, resultando em 80 palavras. Em seguida, 80 rodadas são executadas, onde distintas funções e constantes são aplicadas. Ao final, os valores iniciais do estado são adicionados com os novos valores produzidos após as rodadas (ver Algoritmo 2).

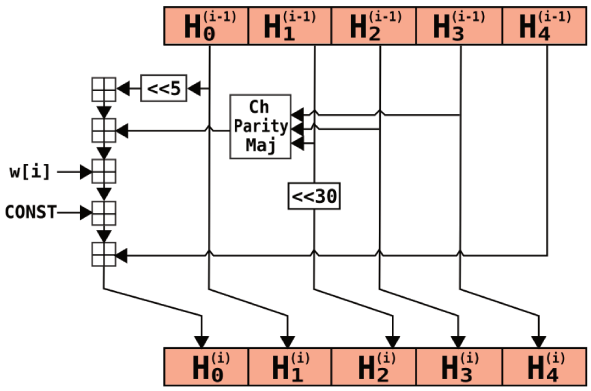


Figura 3.1: Um passo do algoritmo SHA-1

O algoritmo SHA-1 é uma das funções de resumo mais usadas no mundo, estando presente em diversos protocolos como TLS, *SSH* [159] e IPSec.

3.2.2 Ataques

Por ser bastante usado por todo o mundo, o algoritmo SHA-1 foi bastante analisado por pesquisadores da área. Em 2005, [127] e [27] publicaram ataques de colisão em versões reduzidas do SHA-1. Porém, no mesmo ano, Wang et al. [149] anuncia um ataque quebrando teoricamente a resistência à colisão do algoritmo, fato este que incentivou ao NIST a recomendar o uso da versão posterior da família SHA, o SHA-2. Esta quebra é

atualmente o melhor ataque teórico ao SHA-1, com um total de 2^{69} operações.

Algoritmo 2: O algoritmo SHA-1

Entrada: Mensagem M
Saída: Resumo criptográfico de 160 *bits*
$x \leftarrow$ **preenchimento_mensagem**(M)
$x = x_1 \| x_2 \| \ldots \| x_n$, onde cada x_i é um bloco de 512 *bits*

Seja o estado H inicializado da seguinte forma,
$H_0^{(0)} \leftarrow$ `0x67452301`, $H_1^{(0)} \leftarrow$ `0xEFCDAB89`, $H_2^{(0)} \leftarrow$ `0x98BADCFE`,
$H_3^{(0)} \leftarrow$ `0x10325476` $H_4^{(0)} \leftarrow$ `0xC3D2E1F0`

para *i=1* **até** *n* **faça**
$x_i = w_0 \| w_1 \| \ldots \| w_{15}$, onde w_i é uma palavra de 32 *bits*
para *j=16* **até** *79* **faça**
$w_j = (w_{j-3} \oplus w_{j-8} \oplus w_{j-14} \oplus w_{j-16}) \lll 1$
fim

$a \leftarrow H_0^{(i-1)}$, $b \leftarrow H_1^{(i-1)}$, $c \leftarrow H_2^{(i-1)}$, $d \leftarrow H_3^{(i-1)}$, $e \leftarrow H_4^{(i-1)}$

para *j=0* **até** *19* **faça**
$tmp \leftarrow (a \lll 5) + Ch(b,c,d) + e + w_j +$ `0x5A827999`
$e \leftarrow d$, $d \leftarrow c$, $c \leftarrow b \lll 30$, $b \leftarrow a$, $a \leftarrow tmp$
fim
para *j=20* **até** *39* **faça**
$tmp \leftarrow (a \lll 5) + Parity(b,c,d) + e + w_j +$ `0x6ED9EBA1`
$e \leftarrow d$, $d \leftarrow c$, $c \leftarrow b \lll 30$, $b \leftarrow a$, $a \leftarrow tmp$
fim
para *j=40* **até** *59* **faça**
$tmp \leftarrow (a \lll 5) + Maj(b,c,d) + e + w_j +$ `0x8F1BBCDC`
$e \leftarrow d$, $d \leftarrow c$, $c \leftarrow b \lll 30$, $b \leftarrow a$, $a \leftarrow tmp$
fim
para *j=60* **até** *79* **faça**
$tmp \leftarrow (a \lll 5) + Parity(b,c,d) + e + w_j +$ `0xCA62C1D6`
$e \leftarrow d$, $d \leftarrow c$, $c \leftarrow b \lll 30$, $b \leftarrow a$, $a \leftarrow tmp$
fim

$H_0^{(i)} \leftarrow a + H_0^{(i-1)}$, $H_1^{(i)} \leftarrow b + H_0^{(i-1)}$, $H_2^{(i)} \leftarrow c + H_2^{(i-1)}$,
$H_3^{(i)} \leftarrow d + H_3^{(i-1)}$, $H_4^{(i)} \leftarrow e + H_4^{(i-1)}$
fim
retorna $H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)}$

3.2.3 Implementações

No início da década de 90, a arquitetura dominante era de 32 *bits*, desta forma, o algoritmo SHA-1 foi desenhado voltado para esta tecnologia. Com o advento das arquiteturas de 64 *bits* e instruções multimídia de 128 *bits*, alguns trabalhos foram publicados com o objetivo de explorar o paralelismo presente no SHA-1 [30, 100].

Algumas análises também foram feitas em relação às implementações do algoritmo em *hardware* [161], principalmente em ambientes com recursos escassos [42, 121].

3.3 SHA-2

Com suspeitas cada vez maiores de vulnerabilidades no SHA-1 e com o advento de novas tecnologias, a NSA projetou em 2001 a versão seguinte da família SHA. O SHA-2 é composto por quatro algoritmos: SHA-224, SHA-256, SHA-384 e SHA-512, cada um gerando respectivamente resumos de 224, 256, 384 e 512 *bits*. Devido à falta de compatibilidade com aplicações antigas e a ausência da quebra prática do SHA-1, os algoritmos do SHA-2 foram até hoje pouco utilizados. O projeto SHA-2 é incorporada principalmente em sistemas operacionais e em propostas de novos protocolos [83].

3.3.1 Descrição

Inicialmente, é necessário ressaltar que os algoritmos do SHA-2 mantêm uma estrutura muito semelhante ao SHA-1, eles são construídos no modelo Merkle-Damgård e os passos contidos nas funções de compressão são compostos de permutações das palavras do estado depois da aplicação de funções pré-estabelecidas.

Os algoritmos SHA-224 e SHA-384 são idênticos aos algoritmos SHA-256 e SHA-512, respectivamente, apenas sofrendo um truncamento do valor de resumo gerado. Assim como o SHA-1, o SHA-256 processa mensagens de até 2^{64} *bits* e trabalha com palavras de 32 *bits*. Sua função de compressão recebe como entrada um bloco de mensagem de 512 *bits* e uma variável de estado de 256 *bits*, gerando outra variável de 256 *bits*. O SHA-512 processa mensagens de até 2^{128} e trabalha com palavras de 64 *bits*, sua função de compressão recebe um bloco de mensagem de 1024 *bits* e uma variável de estado de 512 *bits* gerando outra variável de 512 *bits*.

Os dois algoritmos são bastante semelhantes, recebendo mudanças nas constantes e funções internas, portanto, iremos descrever nesta seção apenas o algoritmo SHA-256. Para maiores detalhes ver [58].

Preenchimento da mensagem

Esta etapa é idêntica ao algoritmo SHA-1, ver Seção 3.2.1.

Computação do resumo

Inicialmente, a mensagem é dividida em blocos e as variáveis do estado são inicializadas.

Algoritmo 3: Inicialização SHA-256

Entrada: Mensagem M
Saída: Mensagem M dividida em blocos e variável de estado inicializado
$x \leftarrow$ **preenchimento_mensagem**(M)
$x = x_1 \| x_2 \| \ldots \| x_n$, onde cada x_i é um bloco de 512 *bits*

Seja o estado H inicializado da seguinte forma,
$H_0^{(0)} \leftarrow$ `0x6A09E667`
$H_1^{(0)} \leftarrow$ `0xBB67AE85`
$H_2^{(0)} \leftarrow$ `0x3C6Ef372`
$H_3^{(0)} \leftarrow$ `0xA54FF53A`
$H_4^{(0)} \leftarrow$ `0x510E527F`
$H_5^{(0)} \leftarrow$ `0x9B05688C`
$H_6^{(0)} \leftarrow$ `0x1F83D9AB`
$H_7^{(0)} \leftarrow$ `0x5BE0CD19`

retorna *H, e x*

O SHA-256 usa seis funções internas, sendo duas idênticas às funções usadas no SHA-1:

1. $Ch(x, y, z) = (x \wedge y) \oplus (\neg(x) \wedge z)$;
2. $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$;
3. $\sum_0^{\{256\}}(x) = (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22)$;
4. $\sum_1^{\{256\}}(x) = (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25)$;
5. $\sigma_0^{\{256\}}(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3)$;
6. $\sigma_1^{\{256\}}(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10)$.

Assim como no SHA-1, o bloco da mensagem de 512 *bits* é dividido em 16 palavras de 32 *bits*, uma expansão é feita, resultando em 64 palavras de 32 *bits*. Em seguida, 64 rodadas são aplicadas, porém, no SHA-2, cada rodada tem uma constante distinta. Ao

final, as variáveis de estado iniciais são adicionadas ao valor gerado pelas rodadas.

Algoritmo 4: SHA-256

Entrada: Mensagem M é dividida em n blocos de 512 *bits* e as variáveis de estado inicializadas

Saída: Resumo criptográfico de 256 *bits*

para *i=1* **até** n **faça**

$x_i = w_0 \| w_1 \| \ldots \| w_{15}$, onde cada w_i é uma palavra de 32 *bits*

`// Expansão da mensagem`

para *j=16* **até** *63* **faça**

$w_j = \sigma_1^{\{256\}}(w_{t-2}) + w_{t-7} + \sigma_0^{\{256\}}(w_{t-15}) + w_{t-16}$

fim

`// Guardar valores iniciais do estado`

$a \leftarrow H_0^{(i-1)},\ b \leftarrow H_1^{(i-1)},\ c \leftarrow H_2^{(i-1)},\ d \leftarrow H_3^{(i-1)},$
$e \leftarrow H_4^{(i-1)},\ f \leftarrow H_5^{(i-1)},\ g \leftarrow H_6^{(i-1)},\ h \leftarrow H_7^{(i-1)}$

para *j=0* **até** *63* **faça**

$tmp_1 \leftarrow h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + CONST_j + w_j$
$tmp_2 \leftarrow \sum_0^{\{256\}}(a) + Maj(a,b,c)$
$h \leftarrow g$
$g \leftarrow f$
$f \leftarrow e$
$e \leftarrow d + tmp_1$
$d \leftarrow c$
$c \leftarrow b$
$b \leftarrow a$
$a \leftarrow tmp_1 + tmp_2$

fim

$H_0^{(i)} \leftarrow a + H_0^{(i-1)},\ H_1^{(i)} \leftarrow b + H_1^{(i-1)},\ H_2^{(i)} \leftarrow c + H_2^{(i-1)},\ H_3^{(i)} \leftarrow d + H_3^{(i-1)},$
$H_4^{(i)} \leftarrow e + H_4^{(i-1)},\ H_5^{(i)} \leftarrow f + H_5^{(i-1)},\ H_6^{(i)} \leftarrow g + H_6^{(i-1)},\ H_7^{(i)} \leftarrow h + H_7^{(i-1)}$

fim

retorna $H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$

3.3.2 Ataques

Até o momento, apenas foram feitos ataques em versões reduzidas e variações dos algoritmos do SHA-2 com operações modificadas para relaxar a segurança do projeto. O

melhor ataque de colisão foi publicado por Indesteege et al. em 24 passos do algoritmo SHA-224/256 [77]. Porém, devido à semelhança da estrutura dos integrantes do SHA-2 com seu antecessor, o SHA-1, o NIST achou prudente a existência de um algoritmo que oferecesse uma alternativa às funções de resumo modeladas conforme o MD4 e MD5, o que culminou no início do concurso SHA-3.

3.3.3 Implementações

Uma vez que a família SHA-2 ainda não foi intensamente utilizada em aplicações e protocolos, não existem muitos estudos referentes à implementação eficiente dos algoritmos. Gueron e al. concluíram que o SHA-512 é mais propício a ser implementado em arquiteturas de 64 *bits*, principalmente pelo seu número reduzido de rodadas, gerando melhores resultados que o SHA-256 [135].

Outros trabalhos destacam a implementação em hardware do algoritmo, sendo criticado por sua dificuldade a ser aplicado em ambientes com recursos muito restritos, como *chips RFID* (*Radio-frequency Identification*) [56].

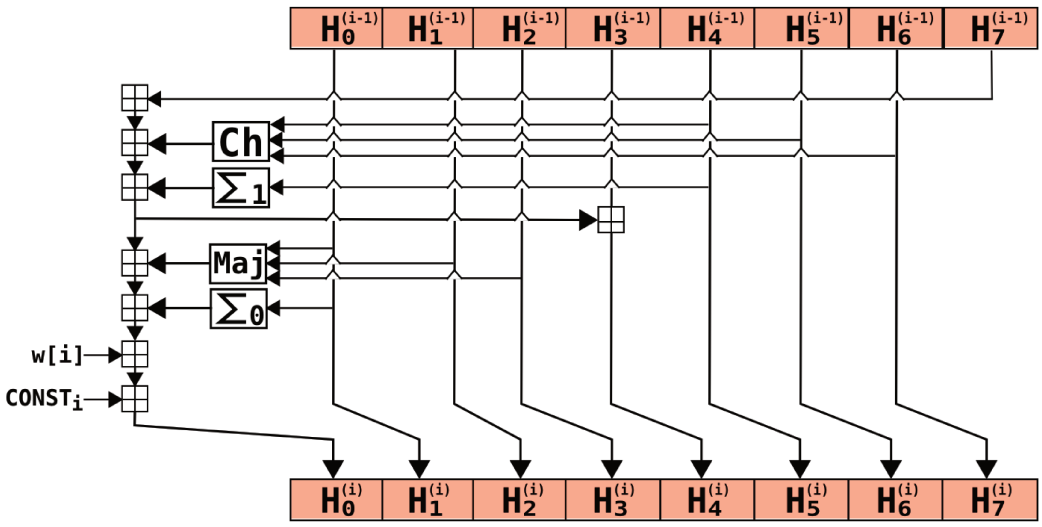


Figura 3.2: Um passo do algoritmo SHA-2

3.4 O concurso SHA-3

Após a quebra dos algoritmos SHA-0, MD4 e MD5, e a publicação da bem sucedida criptoanálise do algoritmo SHA-1, a comunidade começou a se preocupar com o futuro das assinaturas digitais, que são essenciais para o comércio eletrônico. Mesmo que a

criptoanálise dos algoritmos do SHA-2 pareçam longe de originar uma quebra prática, sua estrutura é bastante similar à do SHA-1. Caso este cenário ocorra, os esquemas de assinaturas digitais estarão todos invalidados.

Além disso, críticas foram feitas aos algoritmos SHA-1 e SHA-2 por seus detalhes do projeto serem fechados ao público. Por estes motivos, após um período de consulta pública e dois *workshops*, o NIST decidiu publicar em 2007 uma chamada aberta à toda comunidade para a seleção de um novo algoritmo, o SHA-3. Apesar de não ter invalidado o SHA-2, o NIST almeja a substituição do mesmo pelo SHA-3.

Diferentemente dos outros algoritmos da família SHA, o algoritmo SHA-3 será selecionado por meio de um concurso público, nos mesmos moldes do AES, o algoritmo padrão norte-americano de cifra de bloco, selecionado em 2001 após cinco anos de concurso. Por esta razão, todos os algoritmos submetidos devem ter suas patentes abertas e seu código disponível para toda a comunidade, a fim de serem analisados e testados durante o concurso.

Cronograma O NIST compôs o concurso por meio de três rodadas. Com o anúncio no início de 2007 e a seleção final na metade de 2012.

Tabela 3.1: Cronograma concurso SHA-3

2006	Planejamento do concurso, definição de requisitos básicos, *workshops*
4° trimestre 2007	Anúncio do concurso público com os requisitos básicos para submissão e o critério de avaliação
2° trimestre 2009	Revisão das submissões e seleção dos algoritmos que satisfizerem os requisitos básicos através da primeira conferência das funções de resumo candidatas (do inglês *First Hash Function Candidate Conference*). **Início da primeira rodada**
2° trimestre 2010	Segunda conferência das funções de resumo candidatas. Discussão das análises públicas. Seleção dos algoritmos. **Início da segunda rodada**
4° trimestre 2010	Seleção dos finalistas. **Início da rodada final**
2° trimestre 2012	Discussão das análises públicas. Anúncio do(s) vencedor(es).
3° trimestre 2012	Preparação para a publicação do novo padrão SHA-3

De acordo com o cronograma, o publico tem aproximadamente três anos para avaliar os algoritmos. Dado o número considerável de submissões, críticas foram feitas [9] com relação a impossibilidade de realizar uma análise completa em vários candidatos no tempo disponível. Assim, o NIST adiantou a segunda fase para o primeiro trimestre de 2010.

Requisitos Juntamente com o anúncio do concurso, o NIST publicou uma lista de requisitos básicos que os candidatos devem satisfazer [89].

1. O algoritmo não deve ter restrições de propriedade intelectual, e seu conteúdo deve ser aberto a todas as regiões do mundo;
2. Deve ser possível implementar o algoritmo em várias plataformas de *hardware* e *software*
3. O algoritmo deve gerar resumos de 224, 256, 384 e 512 *bits*, além de processar mensagens com um tamanho de até $2^{64} - 1$.

Critérios de avaliação O NIST também publicou critérios mínimos que devem ser usados para avaliar os algoritmos, incentivando ao público que também faça suas considerações.

1. Segurança. O algoritmo deve oferecer garantias quanto às propriedades básicas (ver Seção 2.1.1), além de resistir a todos os ataques existentes até o momento. Os candidatos também devem ser seguros se inseridos nas aplicações criptográficas padronizadas pelo NIST, como assinaturas digitais, derivação de chaves, MACs, geradores de *bits* pseudorrandômicos entre outros. Este critério é o mais importante, e foi usado como justificativa para a maioria das seleções dos candidatos entre as rodadas [144, 9].
2. Custo. O projeto deve demonstrar bons resultados em duas vertentes: eficiência computacional e memória. O primeiro ponto se refere basicamente à velocidade do algoritmo em *software* e *hardware*. A velocidade será medida em mensagens de distintos tamanhos e em variadas plataformas, como 64/32 *bits*, 8 *bits*, processadores de sinais digitais, entre outros.

   Quanto à memória, serão avaliados principalmente a quantidade de portas para *hardware*. Em *software* serão considerados o tamanho do código e a quantidade de memória RAM necessária.
3. Outras características. De acordo com a documentação, o NIST irá considerar a flexibilidade dos algoritmos em trocar parâmetros de eficiência e segurança, além de valorizar a simplicidade dos projetos.

Apesar do grande número de exigências, um total de 64 submissões foram feitas, mas apenas 51 foram selecionadas pelo NIST para avançarem à primeira fase.

3.4.1 Primeira rodada

Nesta primeira rodada, 51 algoritmos de várias partes do mundo foram selecionados. É importante ressaltar a grande variedade de projetos, muitos baseados no método Merkle-Damgård, com distintas funções de compressão. Além disso, dois algoritmos são baseados em primitivas aritméticas.

Um total de 32 algoritmos foram quebrados pelo público, evidenciando a dificuldade em se projetar funções de resumo criptográficas adequadas. Apenas 14 algoritmos avançaram à fase seguinte. Na Tabela 3.3 estão listados os algoritmos que não passaram para a segunda fase.

3.4.2 Segunda rodada

Diferentemente da primeira rodada, os algoritmos nesta fase são mais concisos. Desta forma, mesmo com publicações relacionadas à criptoanálise dos algoritmos, não houve nenhuma quebra prática.

A performance também foi um fator muito avaliado, principalmente por meio dos projetos *eBASH* [20] e *XBX* [152], que reúnem *benchmarks* realizados em implementações de *software* em várias plataformas. Trabalhos relacionados à implementação em *hardware* também foram publicados [141, 11, 95].

Tabela 3.2: Algoritmos que não avançaram para a fase final

Algoritmo	**Nacionalidade**	**Classificação**	**Quebrado?**
Blue Midnight Wish [64]	Noruega	Dedicada/MD	Não
CubeHash [18]	EUA	Dedicada/Esponja	Não
ECHO [15]	França	Dedicada/HAIFA	Não
Fugue [69]	EUA	Dedicada/Esponja	Não
Hamsi [162]	Bélgica	Dedicada/Concatenar-Permutar-Truncar	Não
Luffa [39]	Bélgica/Japão	Dedicada/Esponja	Não
Shabal [33]	França	Dedicada/Esponja	Não
SHAvite-3 [29]	Israel/França	AES/HAIFA	Não
SIMD [99]	França	Dedicada/MD	Não

3.4.3 Fase final

A fase final do concurso SHA-3 iniciou no último semestre de 2010 e conta com cinco algoritmos: *BLAKE*, *Grøstl*, *JH*, *Keccak* e *Skein*. Maiores detalhes no Capítulo 8.

Tabela 3.3: Algoritmos que não avançaram para a segunda fase

Algoritmo	**Nacionalidade**	**Classificação**	**Quebrado?**
Abacus [136]	EUA	Dedicada/Esponja	Sim
ARIRANG [41]	Coreia do Sul	Dedicada/MD	Não
AURORA [81]	Japão	Dedicada/MD	Sim
Blender [32]	EUA	Dedicada/MD	Sim
Boole [131]	EUA	Dedicada	Sim
Cheetah [91]	Luxemburgo	Dedicada/HAIFA	Não
CHI [72]	Austrália	Dedicada/MD	Não
CRUNCH [66]	França	Dedicada/MD	Não
DCH [153]	EUA	Dedicada/MD	Sim
Dynamic SHA [157]	China	Dedicada/MD	Sim
Dynamic SHA2 [158]	China	Dedicada/MD	Sim
ECOH [35]	Canadá	Baseada em primitivas aritméticas	Sim
Edon-R [65]	Noruega	Dedicada/MD	Sim
EnRUPT [120]	França/EUA/Suíça	Dedicada	Sim
ESSENCE [104]	EUA	Dedicada/MD-Tree	Sim
FSB [6]	França	Dedicada/MD	Não
Khichidi-1 [147]	India	Dedicada/CBC	Sim
LANE [76]	Bélgica	Dedicada/MD	Não
Lesamnta [74]	Japão	Dedicada/MD	Não
LUX [115]	Luxemburgo	Dedicada	Sim
MCSSHA-3 [105]	Coreia do Sul	Dedicada	Sim
MD6 [130]	EUA	Dedicada/HAIFA	Não
MeshHash [55]	Alemanha	Dedicada/MD	Sim
NaSHA [103]	Macedônia	Baseada em primitivas aritméticas	Sim
SANDstorm [142]	EUA	Dedicada/MD	Não
Sarmal [145]	Turquia	Dedicada/HAIFA	Sim
Sgàil [107]	Reino Unido	Dedicada/MD	Sim
SHAMATA [1]	Turquia	Dedicada/MD	Sim
Spectral Hash [132]	EUA	Dedicada/MD	Sim
StreamHash [143]	Polônia	Dedicada	Sim
SWIFFTX [4]	Israel/EUA	Dedicada/HAIFA	Não
Tangle [126]	Espanha/Irlanda	Dedicada/MD	Sim
TIB3 [112]	Argentina	Dedicada/MD	Sim
Twister [54]	Alemanha	Dedicada/MD	Sim
Vortex [96]	Israel	Dedicada/MD	Sim
WaMM [151]	EUA	Dedicada/MD	Sim
Waterfall [71]	Reino Unido	Dedicada	Sim

Capítulo 4

Arquiteturas

Neste capítulo, nos distanciamos brevemente das funções de resumo criptográfico para apresentarmos de maneira sucinta as características gerais das arquiteturas *x86* da *Intel* e *ATmega* da *Atmel*, que foram as plataformas base para a realização deste trabalho. Além disso, explicitaremos as técnicas e instruções mais importantes para as otimizações contidas nos Capítulos 6 e 7.

4.1 Intel x86 / Intel 64

Intel x86 é uma das arquiteturas mais difundidas entre os computadores pessoais e comerciais, sendo integrada em diversos processadores da Intel, *AMD* [2], *VIA* [146], entre outros. Sua origem remonta ao processador *8086* da Intel lançado em 1978, e desde então, foi desenvolvida mantendo a compatibilidade com modelos anteriores [46]. Esta característica contribui para o seu grande seu sucesso comercial, atraindo desenvolvedores e usuários finais de aplicações.

No entanto, sempre houveram críticas quanto à "deselegância" da arquitetura x86, o que reflete em sua ausência no novo mercado de sistemas embarcados, onde os fabricantes também dominam o desenvolvimento de aplicações, e possuem mais liberdade para escolher a arquitetura em seus equipamentos.

4.1.1 Breve história

Os processadores 8086 lançados pela Intel em 1978 possuíam oito registradores de 16 *bits*, com um barramento de endereçamento de memória de 20 *bits*, podendo assim acessar até 1 MB de espaço físico ($2^{20} = 1.048.576$). Este modelo inaugurou a proteção de memória através da segmentação nos projetos da Intel.

Em 1982 foi lançado o processador *80286* com maior capacidade de endereçamento de memória (24 *bits*) podendo acessar até 16 MB de espaço físico. Neste modelo também introduziu-se novos mecanismos de proteção à memória.

O primeiro processador de 32 *bits* da Intel foi lançado em 1985, com o nome de *80386*. Além de incorporar registradores de 32 *bits*, também possuía esta capacidade para endereçar memória, possibilitando acesso a até 4GB de espaço físico. Grandes inovações neste modelo foram o suporte à técnica de memória virtual usada por sistemas operacionais e a introdução de uma nova forma de endereçar a memória.

Com o processador *80486* lançado em 1989, a Intel ampliou a capacidade de execução paralela introduzindo um *pipeline* de cinco estágios. Outra inovação foi uma *cache* de 8 KB para armazenar dados e instruções. Por fim, neste modelo foi integrada uma unidade de execução para dados de tipo ponto flutuante.

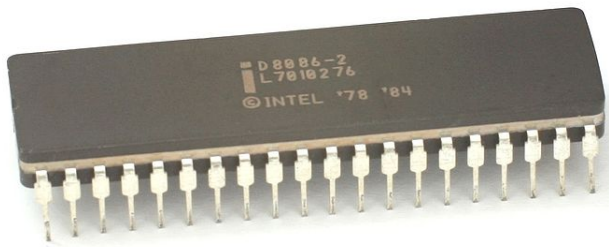


Figura 4.1: O processador 8086

O processador *Pentium* foi lançado em 1993 com a introdução do *pipeline* duplo, permitindo executar instruções de forma superescalar, ou seja, mais de uma instrução por ciclo. Além disso, o tamanho da *cache* dobrou, com 8 KB para dados e 8 KB para instruções. Finalmente, o modelo lançou a tecnologia *MMX*, que proporciona a execução de instruções em múltiplos dados por meio de registradores especiais de 64 *bits*.

A partir de 1995 a Intel lança a família *P6*, que inclui os processadores Pentium Pro, Pentium II e Pentium III. Nestes modelos a capacidade superescalar é ampliada, possibilitando a execução de até três instruções simultaneamente. Outra modificação foi a adição de um segundo nível de *cache* para acelerar o acesso aos dados da memória. Com o Pentium III um conjunto de instruções denominado SSE (*Streaming SIMD Extensions*) foi lançado com o objetivo de processar múltiplos dados de ponto flutuante por meio de registradores especiais de 128 *bits*.

Com a família Pentium 4 lançada em 2000, a Intel lança os conjuntos de instruções SSE

2 e SSE 3, ampliando as *Streaming SIMD Extensions*. Além disso, introduziu tecnologias para aumentar a capacidade de executar instruções de forma paralela.

Em 2006 a Intel lança a família *Core 2*, com um processador de 64 *bits*, que por sua vez foi introduzido comercialmente por sua concorrente, a AMD, em seu processador Opteron em 2003. Esta arquitetura possui 16 registradores de 64 *bits*, podendo endereçar até 256 TB de espaço físico. Outras novidades são a inauguração de múltiplos núcleos de processamento e a expansão do conjunto de instruções SSE.

Por fim, a partir de 2008, a Intel lança os processadores *Core i3*, *Core i5* e *Core i7*. Neste modelos as instruções SSE são expandidas e um terceiro nível de *cache* foi introduzido. A segunda geração destes processadores foi disponibilizada no início de 2011, com instruções dedicadas a algoritmos criptográficos, além de um novo conjunto denominado *AVX* que permite computar múltiplos dados em registradores especiais de 256 *bits*.

4.1.2 Ambiente

Devido à política de compatibilidade com os modelos anteriores, os processadores x86 de 64 *bits*, também chamados pela Intel de *Intel 64* possuem dois modos, o *modo de compatibilidade*, que pode executar aplicativos compilados para os antigos processadores de 16 e 32 *bits*, e o modo 64 *bits*. A maior parte de nosso trabalho ocorreu neste último ambiente, e portanto, nesta seção daremos ênfase ao mesmo. Em temas mais específicos como o acesso à memória e o *pipeline*, a referência será a família Core 2.

Neste ambiente, apesar de haver suporte para o endereçamento de memória de 64 *bits*, é possível apenas acessar 2^{40} *bytes* de espaço físico. Os registradores de acesso geral são 16, todos com 64 *bits*, e suportando também operações em dados de 8, 16 e 32 *bits*. Registradores de controle, *FLAGS* e ponteiros de pilha também são ampliados para 64 *bits*. Por fim, são disponibilizados 16 registradores *XMM*, pelos quais são executados instruções SSE em múltiplos dados dos tipos ponto flutuante e inteiro.

Cache

Cada núcleo do processador possui uma *cache* de primeiro nível, dividindo entre si uma *cache* de segundo nível. Os parâmetros deste sistema estão descritos na Tabela 4.1.

Para acelerar o acesso aos dados, o processador conta com os chamados *prefechers*, que tem como objetivo antecipar o carregamento de dados observando o padrão de acessos. Os *prefetchers* podem ser de dois tipos:

- *Prefecher* na cache de dados, acionado por um acesso a um dado de um nível hierárquico de memória superior. Neste caso, o sistema considera que o dado faz parte de uma cadeia e carrega também dados das linhas seguintes.

- *Prefecher* na cache de instruções, que verifica o padrão de acesso das instruções em carregar dados da memória. Com um padrão definido, o *prefetcher* carrega antecipadamente os dados seguintes.

Tabela 4.1: Parâmetros da memória *cache* da família Core 2

Nível	**Capacidade**	**Associatividade**[1]	**Tamanho da linha**	**Latência de acesso**	***Throughput* de acesso**
Primeiro - Dados	32 KB	8 vias	64 *bytes*	3 ciclos	1 ciclo
Primeiro - Instruções	32 KB	8 vias	N/A	N/A	N/A
Segundo	3,6 MB	12 ou 24 vias	64 *bytes*	15 ciclos	2 ciclos

Registradores

Na arquitetura Intel 64, existem 16 registradores de 64 *bits* de propósito geral, porém é possível usá-los também com operandos de 32, 16 e 8 *bits*.

Ao usar operandos de 64 *bits* ou os registradores R8 - R15, um prefixo adicional denominado *REX* é concatenado à instrução, o que aumenta o tamanho das mesmas, podendo diminuir o desempenho em algumas etapas do *pipeline*.

Se o implementador utilizar operandos de 8 ou 16 *bits*, o conteúdo restante do registrador é mantido, porém, ao usar operandos de 32 *bits*, a metade superior é zerada. Isto ocorre devido à política de compatibilidade com modelos anteriores.

Alem disso, ainda há mais 16 registradores especiais para serem usados em instruções multimídia SSE, mais detalhes na Seção 4.1.4.

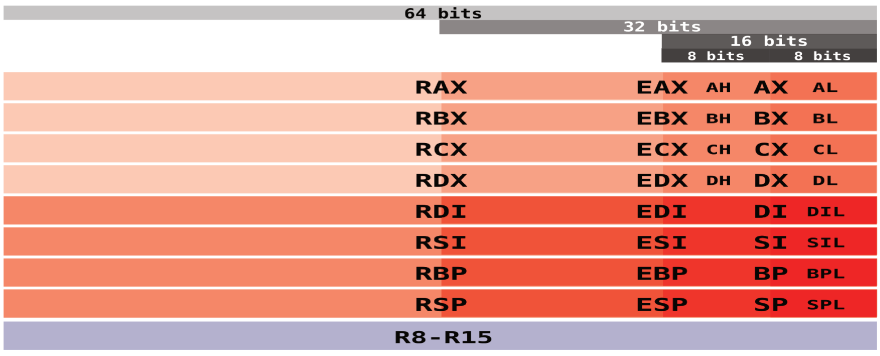


Figura 4.2: Registradores da arquitetura Intel 64

[1]Associatividade da cache é a quantidade de linhas disponíveis para um determinado endereço de memória.

4.1.3 *Pipeline*

O *pipeline* dos processadores Core 2 difere um pouco do modelo anterior, o Pentium 4, pois a evolução dos microprocessadores baseada no aumento do *clock* chegou ao seu limite. Portanto, os projetistas desenharam o novo *pipeline* com o objetivo de alcançar maior capacidade superescalar, ou seja, executar mais instruções por ciclo.

Desta forma, o *pipeline* do Core 2 é composto resumidamente por quatro etapas básicas: busca das instruções e pré-decodificação, decodificação, renomeação de registradores e finalmente, a execução.

Busca das instruções e pré-decodificação Nesta fase, ocorre a retirada de instruções da *cache* e a determinação de seus respectivos tamanhos. A cada ciclo, é possível retirar até 16 *bytes* de instruções, desta forma, a existência de prefixos nas mesmas diminui a eficiência desta etapa.

Decodificação Nesta etapa as instruções são quebradas em micro-operações, estas por sua vez podem ser fundidas em um processo denominado macrofusão. Este processo tem o objetivo de tornar mais eficiente a execução das micro-operações.

Renomeação de registradores Aqui os registradores são traduzidos em registradores virtuais para evitar falsas dependências entre os mesmos. Além disso, esta etapa envia as micro-operações para uma estrutura denominada *buffer* de reordenação (do inglês *Reorder buffer*), que permite a execução fora de ordem. Nesta fase também se aloca as micro-operações em fila na estação de reserva (do inglês *Reservation station*), onde as mesmas esperarão que os seus operandos sejam computados.

Execução Por fim, até seis micro-operações podem ser enviadas às unidades de execução. A arquitetura Core 2 possui três portas para operações lógicas e aritméticas, uma porta para ler e outra para escrever na memória e por fim uma porta para calcular endereços da memória.

O maior desafio dos programadores é conseguir gerar códigos que possam ser executados pelo *pipeline* de maneira mais eficiente possível, em outras palavras, que o mantenha constantemente ocupado. Porém, existem limites, também chamados de "gargalos", impostos pelas estruturas do *pipeline* para a computação das micro-operações. Estes se encontram principalmente na etapa de pré-decodificação e na renomeação de registradores, que pode lidar com no máximo quatro micro-operações por ciclo.

4.1.4 SSE

O conjunto de instruções SSE foi lançado pela Intel no seu processador Pentium III em 1999. Nos anos seguintes, mais instruções foram adicionadas, originando distintas versões: SSE2, SSE3, SSSE3, SSE4 [45] e AVX [47]. A plataforma SSE inclui também registradores de 128 *bits*, sendo 16 na arquitetura Core 2.

Apesar de ser voltado a aplicações multimídia, as instruções se aplicam a todo algoritmo que realiza cálculos semelhantes em uma série de dados distintos, como por exemplo, os algoritmos criptográficos. A seguir detalharemos algumas instruções que foram bastante usadas em nosso trabalho.

Instrução *alignr*

A instrução *alignr* permite deslocar à direita dois registradores de 128 *bits* concatenados. Por exemplo, seja $r_0 = (a_0, a_1, \ldots, a_{15})$ e $r_1 = (b_0, b_1, \ldots, b_{15})$ dois registradores SSE contendo *bytes* a_i e b_i, a instrução _mm_alignr_epi8$(r_0, r_1, 3)$ concatena os registradores r_0 e r_1 $(a_0, a_1, \ldots, a_{15}, b_0, b_1, \ldots b_{15})$, realiza o deslocamento de três *bytes* e retorna r_2 contendo os *bytes* mais à direita $(a_{13}, a_{14}, a_{15}, b_0, b_1, \ldots, b_{12})$.

Com um pequeno truque é possível utilizar *alignr* para fazer rotações à direita, para isto basta usar o mesmo registrador nos dois operandos da instrução.

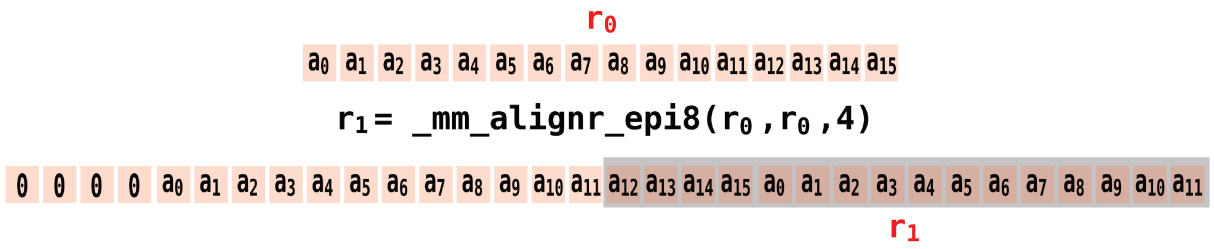


Figura 4.3: Rotação de quatro *bytes* à direita através da instrução *alignr*

Instruções *unpack*

A instrução *unpack* intercala o conteúdo de dois registradores. Dividida em *unpackhi* e *unpacklo*, sendo a primeira intercalando os elementos de posições altas dos registradores e a segunda intercalando itens de posições baixas, esta instrução permite trabalhar com dados de 8, 16 e 32 *bits* contidos dentro dos registradores SSE.

Como exemplo, sejam dois registradores de 128 *bits* $r_0 = (a_0, a_1, a_2, a_3)$ e $r_1 = (b_0, b_1, b_2, b_3)$ contendo quatro elementos de 32 *bits* a_i e b_i. Se quisermos intercalar os dois elementos de posição baixa dos dois registradores fazemos r_2 = _mm_unpacklo_epi32(r_0, r_1). Como resultado obteremos $r_2 = (a_0, b_0, a_1, b_1)$.

Esta operação é muito importante para intercalar *bytes* de distintos registradores de maneira eficiente nas otimizações descritas nos Capítulos 6 e 7.

Instruções *shuffle*

A instrução *shuffle* requer dois ou três operandos, sendo um deles uma máscara. O registrador resultante recebe os elementos dos operandos de acordo com a máscara. Por exemplo, dado um registrador contendo *bytes* $r_0 = (a_0, a_1, \ldots, a_{15})$ e uma máscara de *bytes* $m = (b_0, b_1, \ldots, b_{15})$, a instrução *shuffle* produz o registrador de 16 *bytes* $r_1 = (a_{b_0 \mod 16}, a_{b_1 \mod 16}, \ldots, a_{b_{15} \mod 16})$. Esta instrução pode também ser executada em registradores SSE contendo elementos de 16 e 32 *bits*.

O *shuffle* de *bytes* pode ser usado como uma tabela *lookup*. Se o programador usa a tabela como um operando e o operando como máscara, uma tabela de 128 *bits* é implementada.

```
__m128i mask = {0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,
                0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};

__m128i r0 = {0x00, 0x00, 0x00, 0x00, 0x01, 0x01, 0x01, 0x01,
              0x0E, 0x0E, 0x0E, 0x0E, 0x0F, 0x0F, 0x0F, 0x0F};

// Máscara e operando invertidos.
// A máscara funciona agora como uma tabela de 128 bits.
r0 = _mm_shuffle_epi8(mask, r0);

//r0 = {0x0F, 0x0F, 0x0F, 0x0F, 0x0E, 0x0E, 0x0E, 0x0E,
        0x01, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00}
```

4.2 Atmel AVR ATmega 8

AVR é um microcontrolador *RISC* (*Reduced Instruction Set Computing*) de 8 *bits* desenvolvido pela empresa norte-americana Atmel a partir de 1996. Desde então, a empresa lançou diversos modelos com diferentes recursos de memória, havendo hoje inclusive microcontroladores de 32 *bits*.

Seu conjunto de instruções é bastante simples porém poderoso, pois além de possuir todas as instruções básicas para realizar operações lógicas e aritméticas também possui em algumas versões suporte para *carry* e multiplicações, o que permite executar aplicações bastante complexas.

O modelo usado neste trabalho faz parte da família ATmega 8, que possui 8 KB de memória *flash* programável. Nas seções seguintes detalharemos mais o ambiente desta plataforma.

4.2.1 Ambiente

Memória

A plataforma AVR utiliza do modelo *Harvard* para organizar a memória, ou seja, o código das aplicações e os dados da memória tem barramentos e espaços separados [44]. O código das aplicações reside em uma memória *Flash*, enquanto os dados da memória se encontram em duas estruturas, uma memória SRAM e uma memória EEPROM.

A família ATmega 8 possui 8KB de memória *Flash* reprogramável para armazenar as instruções dos programas. Cada instrução possui um tamanho de 16 ou 32 *bits*, desta forma a memória *Flash* é organizada em 4 KB $\times$ 16 *bits*. O ponteiro para as instruções (PC) possui 12 *bits*, podendo endereçar 4KB de espaço físico ($2^{12} = 4.096$). Como medida de segurança, os dados da aplicação são separados dos dados do *boot*.

Para os dados da memória são disponíveis 1KB (1024 *bytes*) de SRAM. Os acessos podem ser diretos, referenciando o valor da posição da memória e indireto, através de registradores especiais, referenciando uma base e um *offset*. Na memória SRAM também estão localizados o arquivo de registradores e a memória de E/S, ocupando um total de 96 *bytes*.

Finalmente, são disponibilizados 512 *bytes* de memória EEPROM. O acesso a esta memória é um pouco mais lento, e a sua utilização deve ser feita com cautela.

Registradores

Nesta plataforma existem 32 registradores de 8 *bits* para uso geral (R0 - R31). Algumas instruções exigem o uso dos registradores finais (R16 - R31), além disso, os registradores R26 a R31 possuem funções adicionais. São também chamados X (R26 e R27), Y (R28 e R29) e Z (R30 e R31), e funcionam como ponteiros de 16 *bits* para os dados da memória na SRAM. Para endereçar esta memória indiretamente é necessário o uso destes registradores.

Por fim, existem os ponteiros de pilha (SPH e SPL) de 8 *bits* cada, responsáveis por armazenar os endereços de retorno após interrupções e chamadas de sub-rotinas. Os ponteiros são decrementados e incrementados por meio de instruções PUSH e POP, e podem também ser usados pelo implementador para guardar variáveis e valores temporários.

4.2.2 Instruções importantes

ROL e ROR

As instruções *Rotate Left through Carry* e *Rotate Right through Carry* permitem rotacionar múltiplos *bytes* em cadeia com a ajuda da *flag carry*. Estas instruções também permitem dividir e multiplicar variáveis maiores de 8 *bits*.

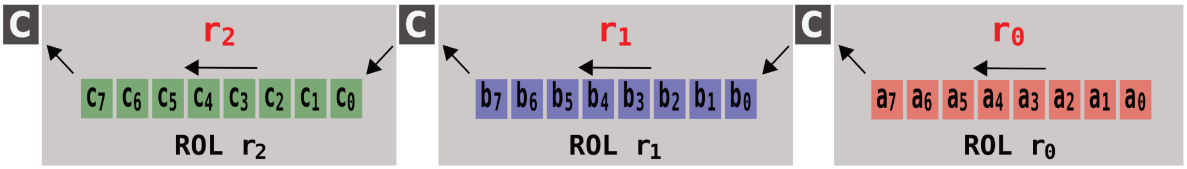


Figura 4.4: Rotação à esquerda de três registradores com o auxílio da *flag carry* *É necessária uma operação extra para levar o valor final do* carry *à posição 0 do registrador* r_0

É importante ressaltar que o valor *carry* se encontra em um registrador denominado SREG *(Status Register)*. Após uma operação lógica ou aritmética, a *flag C* recebe o valor 1 se houver *carry* e 0 caso contrário.

SWAP

Esta instrução permite intercambiar os quatro *bits* altos e baixos de um registrador, por exemplo, seja $r_0 = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$ um registrador AVR com *bits* a_i, a operação SWAP r_0 resulta em $r_0 = (a_4, a_5, a_6, a_7, a_0, a_1, a_2, a_3)$. Desta forma, é possível rotacionar um registrador em quatro *bits* em apenas um ciclo.

ADC e SBC

Com as operações *Add with Carry* e *Subtract with Carry* é possível realizar as operações aritméticas de somar e subtrair em múltiplos *bytes*, o que é essencial para a implementação de muitos algoritmos.

MUL

A instrução *Multiply Unsigned* implementa a multiplicação na plataforma AVR, tomando dois registradores como multiplicando e multiplicador e gerando o resultado de 16 *bits* nos registradores R0 e R1. Esta operação leva dois ciclos e não está disponível em todas os modelos da família ATmega 8.

Capítulo 5

Funções esponja

Neste capítulo descreveremos três algoritmos da segunda rodada do concurso SHA-3 que são baseados na construção esponja.

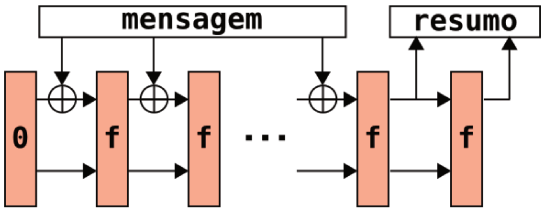
5.1 Introdução

O surgimento de novos ataques às funções de resumo criptográfico levou pesquisadores da área a repensarem a maneira de projetá-las, uma vez que a cada vulnerabilidade descoberta, os algoritmos deveriam apresentar novas provas de segurança. Como consequência, os projetistas pareciam não ter uma meta a seguir, fazendo com que muitos algoritmos fossem quebrados brevemente após sua publicação.

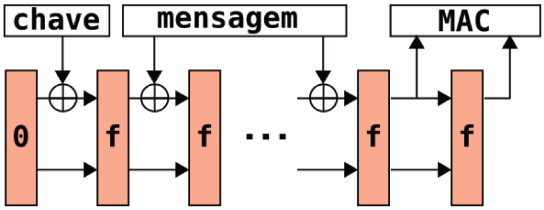
O oráculo randômico trouxe aos pesquisadores um modelo de uma função de resumo ideal [14], porém sua implementação não é possível [36]. Neste contexto, ocorre a formulação da função esponja, na tentativa de isolar os fatores que distinguem uma função real do modelo do oráculo randômico, fatores estes inerentes da maneira iterada de processar a mensagem.

A função esponja permite imitar o comportamento de um oráculo randômico, exceto pelo fato de que na primeira ocorre o fenômeno das colisões internas, ou seja, colisões que ocorrem entre as iterações relativas ao processamento da mensagem. Por meio da função esponja, é possível processar mensagens arbitrárias e finitas e gerar valores infinitos de resumo. Desta forma, é praticável o seu uso em outras aplicações, como MAC ou cifras

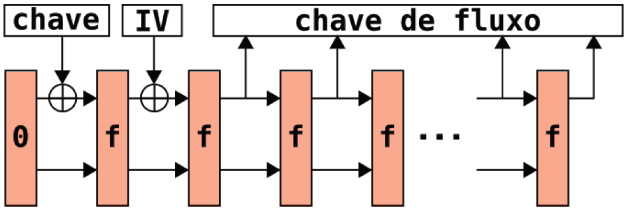
de fluxo.



(a) Função de resumo criptográfico



(b) MAC



(c) Cifra de fluxo

Figura 5.1: Usos da função esponja

Em seguida, apresentaremos uma definição mais formal da função esponja. Para um breve resumo, ver Seção 2.3.3.

Seja $\mathcal{A}$ um grupo que representa os caracteres de entrada e saída, sendo sua operação definida por $+$ e o elemento neutro por 0; $\mathcal{C}$ um conjunto finito de elementos que representam a parte interna do estado da função esponja; $0 \in \mathcal{C}$ um elemento de $\mathcal{C}$, que fará parte do valor inicial do estado da função esponja e $p(m)$ uma função injetora que mapeia mensagens m para caracteres em $\mathcal{A}$, com $|p(m)| \geq 1$ e o último caractere de $p(m)$ diferente de 0.

Definição. *A função esponja é determinada por uma transformação f de $\mathcal{A}\times\mathcal{C}$, possuindo um estado $S = (S_{\mathcal{A}}, S_{\mathcal{C}}) \in \mathcal{A} \times \mathcal{C}$. Ela processa uma cadeia finita e arbitrária p de caracteres $\mathcal{A}$ e gera uma cadeia infinita z de caracteres $\mathcal{A}$.*

O processamento da cadeia p será feito por uma etapa denominada absorção, enquanto a geração da cadeia z se realizará por meio da extração.

Absorção Dado um estado inicial $S_{IV} = (0, 0)$, para cada caractere p_i da entrada, o estado é atualizado da seguinte forma:
$S \leftarrow f(S_{\mathcal{A}} + p_i, S_{\mathcal{C}})$.

Extração Dado o estado $S = (S_{\mathcal{A}}, S_{\mathcal{C}})$ produzido ao final da absorção, os caracteres da cadeia de *bits* z serão gerados gradativamente da seguinte forma:
Até que toda a cadeia seja formada, fazer $z_j = S_{\mathcal{A}}$ e atualizar o estado $S \leftarrow f(S)$.

Conforme citado anteriormente, as funções esponja se diferenciam de um oráculo randômico pela presença das colisões internas. Para defini-las é necessário antes apresentar alguns conceitos.

Definição. *A taxa de uma função esponja é representado por* $r = log_2|\mathcal{A}|$.

Definição. *A capacidade de uma função esponja é representada por* $c = log_2|\mathcal{C}|$.

Definição. *Para uma cadeia de entrada* p, $S_f[p]$ *é definido como o estado obtido após a absorção de* p. *Se* $S = S_f[p]$, *então* p *é um caminho para* S. $S_f = (S_{\mathcal{A},f}, S_{\mathcal{C},f})$ *é definido como* $S_f[\text{cadeia vazia}] = (0, 0)$ *e* $S_f[x\|a] = f(S_f[x] + a)$ *para quaisquer caracteres* x *e* a.

Definição. *Uma colisão nos estados ocorre quando um par de distintos caminhos* $p \neq q$ *geram o mesmo estado* $S_f[p] = S_f[q]$.

Se esta colisão ocorrer durante a absorção, pode gerar valores de resumos iguais, porém, é possível ocorrer colisões durante o processo de extração. Se para algum p e q tivermos $S_f[p] = S_f[p\|0^d]$, valores de saída periódicos serão gerados.

Definição. *Uma colisão interna ocorre quando um par de distintos caminhos* $p \neq q$ *geram o mesmo estado interno* $S_{\mathcal{C},f}[p] = S_{\mathcal{C},f}[q]$.

Uma colisão de estado implica uma colisão interna, porém, o contrário não é verdade. No entanto, é possível gerar facilmente uma colisão de estado a partir de uma colisão interna. Seja $p \neq q$ onde $S_{\mathcal{C},f}[p] = S_{\mathcal{C},f}[q]$, pode-se construir uma colisão de estado em $p\|a \neq q\|b$ para quaisquer a e $b \in \mathcal{A}$ que satisfaça $S_{\mathcal{A},f}[p] + a = S_{\mathcal{A},f}[q] + b$.

Análise de segurança

Ao analisar os ataques existentes nas funções de resumo atuais aplicados na função esponja, os autores mostram que o pior caso, ou seja, o caso que oferece maior probabilidade de sucesso ao atacante é o cenário de se encontrar colisões internas [23], ou seja, justamente o fator que distingue a função esponja de um oráculo randômico. Desta forma, é possível simplificar as declarações de segurança considerando apenas este ataque. Com isso, os parâmetros a se considerar se reduzem ao valor da capacidade c.

Resistência à colisão Se o valor de saída é $n \leq c$, então, o nível de resistência à colisão é $2^{n/2}$. Caso $n \geq c$, o nível de resistência à colisão é $2^{c/2}$.

Resistência à (segunda) pré-imagem Se o valor de saída é $n \leq c/2$, o nível de resistência à (segunda) pré-imagem é 2^n. Caso $n \geq c/2$, o nível é de $2^{c/2}$.

Implementação

Para implementar um algoritmo usando a construção esponja, os autores sugerem a estratégia da esponja hermética (do inglês *hermetic sponge strategy*). Neste método a função esponja é implementada por meio de uma função de permutação cuja distinguibilidade em relação a uma função de permutação randômica é inviável computacionalmente.

Assim como na construção Merkle-Damgård, onde o esforço do projetista se resume à função de compressão, na função esponja, este esforço se dará agora no projeto de uma função de permutação, assim como na escolha do parâmetro c.

Vantagens

Além da simplificação de se realizar provas de segurança, a função esponja traz outras vantagens.

- Flexibilidade em intercambiar níveis de segurança e desempenho, que pode ser feito simplesmente com a escolha dos parâmetros c e r.
- O projeto de uma função de compressão é mais complexo do que uma função de permutação [24].
- Possibilidade de se implementar outras aplicações além das funções de resumo como MAC, cifras de fluxo e geração de máscaras usadas em métodos de encriptação e assinaturas RSA.

No concurso SHA-3 alguns algoritmos podem ser considerados funções esponja ou variantes desta: *Abacus* [136], *CubeHash* [18], *Fugue* [69], Keccak [21], *Luffa* [39] e *Shabal* [33]. Destes, os cinco últimos estiveram presentes na segunda fase e o Keccak está entre os finalistas. Em seguida apresentaremos detalhes de alguns destes projetos.

5.2 Keccak

5.2.1 Descrição

O algoritmo Keccak foi desenhado por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche [22], os mesmos criadores da função esponja. O projeto segue a estratégia da esponja hermética (ver Seção 5.1), portanto, sua segurança se apoia na função de permutação e no parâmetro da capacidade. Uma vez que o Keccak instancia uma função esponja, ele não possui limitações de tamanho de entrada e saída, além de ser flexível quanto ao tamanho dos blocos de mensagem e o tamanho da capacidade. Porém, para a submissão para o concurso SHA-3, os autores estabeleceram alguns parâmetros para orientar a implementação e a análise [21].

O algoritmo trabalha com palavras de 64 *bits* e é dividido em quatro versões que geram diferentes tamanhos de resumo: 224, 256, 384 e 512. O estado é formado por 25 palavras, representadas por meio de uma matriz 5×5, totalizando 1600 *bits*. Os valores da capacidade e da taxa variam de acordo com a versão do algoritmo.

Tabela 5.1: Parâmetros do algoritmo Keccak

Versão algoritmo	**Taxa (r)**	**Capacidade (c)**
Keccak_{224}	1152	448
Keccak_{256}	1088	512
Keccak_{384}	832	768
Keccak_{512}	576	1024

Keccak é dividido em três fases, o preenchimento da mensagem e as duas etapas da função esponja: absorção e extração.

Preenchimento da mensagem

Uma vez que cada versão do Keccak tem diferentes parâmetros r, consequentemente processará distintos blocos de mensagens do mesmo tamanho de r. Desta forma, o preenchimento da mensagem será distinto a cada versão.

Seja M a mensagem inicial, e M' a mensagem após o preenchimento,

1. $M' = M \| 1$;
2. Completar M' com *bits* 0, e finalmente um *bit* 1 para que M' seja múltiplo de r.

Absorção

A absorção se dará do mesmo modo que uma função esponja. O estado será inicializado com *bits* 0, e a cada bloco de mensagem as seguintes etapas são realizadas:

1. Adição (através da operação ou-exclusivo) do bloco de mensagem com a parte correspondente à taxa (r) do estado;
2. Processamento do estado por meio de uma função de permutação f.

A função f

A função de permutação f é o núcleo do algoritmo, ela sustenta as garantias de segurança assim como é responsável pelo desempenho. Ela recebe como entrada uma matriz m 5×5 de 1600 *bits* (cada elemento com 64 *bits*) gerando como saída outra matriz de mesmo tamanho. Na prática, esta matriz pode ser representada por meio de um vetor com 25 posições fazendo $v[5y + x] = m[x, y]$. Porém, para melhor compreensão da função, representaremos a entrada como uma matriz.

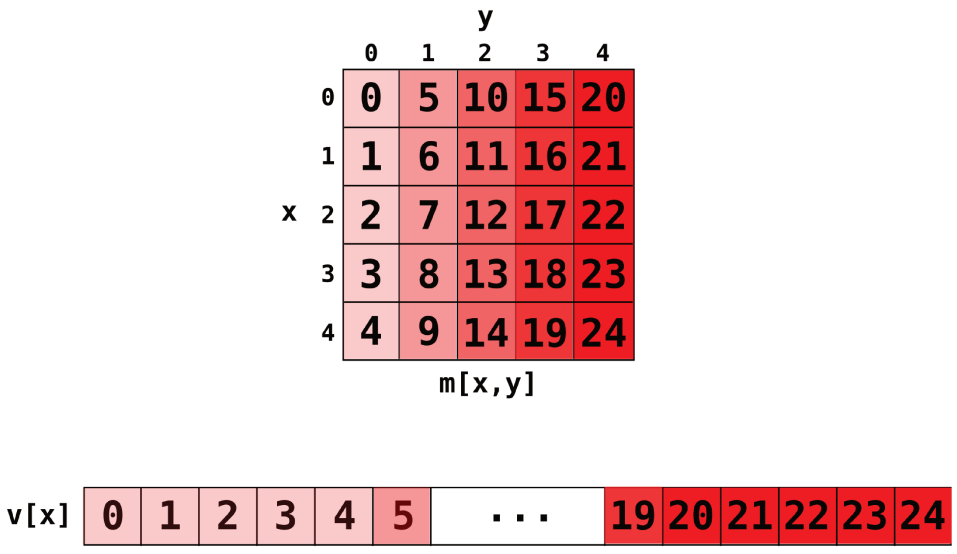


Figura 5.2: Representações da entrada de f

Cada rodada da função f é dividida em cinco etapas: θ, ρ, π, χ e ι. Cada execução de f compreende 24 rodadas.

Etapa θ: Esta etapa é um mapeamento linear responsável pela difusão do algoritmo.

$\theta : m[x, y] \leftarrow m[x, y] \oplus \bigoplus_{y'=0}^{4} m[(x-1), y'] \oplus \bigoplus_{y'=0}^{4} (m[(x+1), y'] \lll 1)$.

Basicamente cada elemento da matriz será somado com a soma dos elementos da linha anterior e a soma dos elementos da linha posterior (esta última precedida de uma rotação de um *bit*).

Algoritmo 5: Etapa θ de f

Entrada: Matriz 5×5 $m[x, y]$
Saída: Matriz 5×5 $m'[x, y]$
para *x=0* **até** *4* **faça**
  $C[x] \leftarrow m[x, 0]$
  para *y=0* **até** *4* **faça**
    $C[x] \leftarrow C[x] \oplus m[x, y]$
  fim
fim

para *x=0* **até** *4* **faça**
  $D[x] \leftarrow C[(x-1) \mod 5] \oplus (C[(x+1) \mod 5] \lll 1)$
  para *y=0* **até** *4* **faça**
    $m'[x, y] \leftarrow m[x, y] \oplus D[x]$
  fim
fim
retorna $m'[x, y]$

Etapa ρ: Nesta etapa são feitas diferentes rotações em cada elemento da matriz, seguindo a fórmula descrita a seguir.

$\rho : m[x, y] \leftarrow (m[x, y] \lll (t+1)(t+2)/2)$;
com $0 \leq t < 24$ e $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ em $GF(5)^{2\times 2}$;
se $x = 0$ e $y = 0$, então $t \leftarrow -1$.

O objetivo desta etapa é acelerar a difusão das outras etapas. Em uma implementação, os valores das rotações podem ser pré-calculados, sem a necessidade de usar a fórmula

acima a cada rodada.

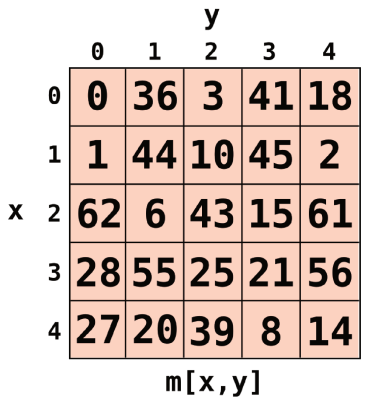


Figura 5.3: Valores das rotações nos elementos da matriz m

Etapa π: Nesta etapa ocorre uma permutação entre os elementos da matriz.

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

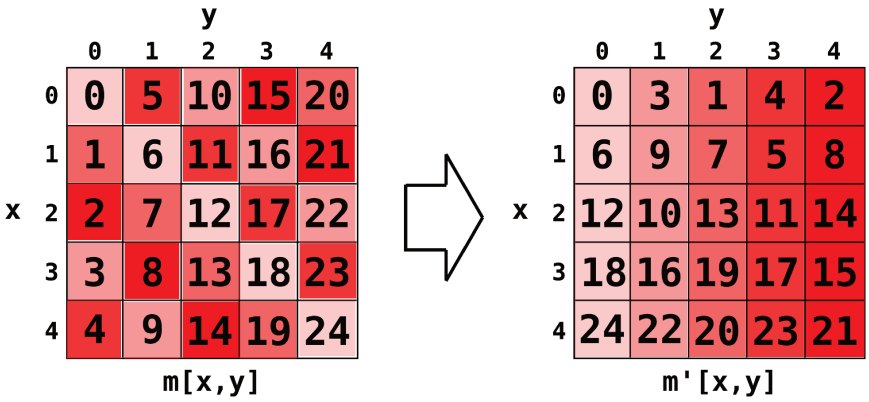


Figura 5.4: Permutação dos elementos da matriz m na etapa π

Esta etapa promove uma difusão a longo prazo dentro das rodadas, a fim de evitar padrões a serem explorados em determinados ataques. Assim como na etapa ρ, os deslocamentos podem ser pré-calculados, fazendo com que a implementação seja mais simples.

Etapa χ: Esta é a única etapa não linear do algoritmo Keccak. Operações lógicas são realizadas entre elementos da mesma coluna.

$\chi : m[x] \leftarrow m[x] \oplus \neg(m[x+1]) \wedge m[x+2]$.

Algoritmo 6: Etapa χ de f

Entrada: Matriz 5×5 $m[x, y]$
Saída: Matriz 5×5 $m'[x, y]$
para *x=0* **até** *4* **faça**
  para *y=0* **até** *4* **faça**
    $m'[x, y] = m[x, y] \oplus (\neg(m[(x+1) \mod 5, y]) \wedge m[(x+2) \mod 5, y])$
  fim
fim
retorna $m'[x, y]$

Etapa ι: A última etapa da função f é uma adição (em $GF(2)$) de uma matriz constante. Esta matriz é variável a cada rodada, e definida da seguinte forma.

Para cada rodada i, e cada *bit* j, $0 \leq j < 64$, $RC[i][0][0][2^j - 1] = rc[j + 7i]$. Todos os valores restantes de $RC[i][x][y]$ são iguais a 0

Os valores de $rc[t] \in GF(2)$ são definidos da seguinte maneira:

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

Extração

Da fase da extração se recupera o valor final do resumo. Conforme descrito na Seção 5.1, os *bits* são extraídos do estado, que é depois processado pela função de permutação f. Este processo se repete até que todos os *bits* do resumo sejam extraídos.

No entanto, em todas as versões do Keccak, a taxa r é maior que o valor do resumo (ver Tabela 5.1). Desta forma, para recuperar o resumo basta extrair os *bits* necessários do estado, sem a necessidade de executar a função de permutação f novamente.

5.2.2 Análises

O algoritmo Keccak foi analisado durante a primeira e segunda fases. Parte dos ataques foram esforços para distingui-lo de um oráculo randômico [8, 31]. Além disso, ataques à resistência à segunda pré-imagem foram publicados por Bernstein [19].

5.3 Luffa

5.3.1 Descrição

Luffa é uma família de quatro algoritmos, Luffa-224, Luffa-256, Luffa-384 e Luffa-512, criados por Christophe De Cannière, Hisayoshi Sato e Dai Watanabe [39]. O projeto chegou até a segunda fase do concurso SHA-3, estando entre os algoritmos mais eficientes tanto em *software* quando em *hardware*.

O algoritmo é considerado uma variante da função esponja pelos seguintes motivos: o estado inicial não é cadeia nula, ou seja, formada por *bits* "0"; a função de permutação é dividida em subfunções e, consequentemente, a inserção do bloco de mensagem é feito de maneira distinta da função esponja; antes de se extrair o primeiro bloco de resumo, o estado é obrigatoriamente processado por uma função de permutação.

Luffa trabalha com palavras de 32 *bits* e opera através da iteração de uma função denominada *round function*. Ao final, o estado passa por uma transformação final antes de se gerar o resumo.

A *round function* recebe como entrada um bloco de mensagem e uma variável de estado, gerando outra variável como saída. Os tamanhos das entradas e saídas são fixos, variando apenas entre os elementos da família Luffa.

Tabela 5.2: Atributos do algoritmo Luffa

Tamanho do resumo (*bits*)	**Tamanho do bloco de mensagem** (*bits*)	**Tamanho da variável de estado** (*bits*)	**Número de permutações**
224	256	3 blocos de 256 (768)	3
256	256	3 blocos de 256 (768)	3
384	256	4 blocos de 256 (1024)	4
512	256	5 blocos de 256 (1280)	5

A *round function* é dividida em duas fases, a inserção do bloco de mensagens às variáveis de estado (*message injection*), e as permutações não lineares. Exceto pela versão Luffa-224 que é apenas um truncamento do Luffa-256, cada versão terá pequenas mudanças na fase de inserção da mensagem e no número de permutações (ver Tabela 5.2). Com a finalidade de simplificar a explicação do algoritmo, iremos focar apenas na

versão Luffa-256. Para maiores detalhes das outras versões ver [39].

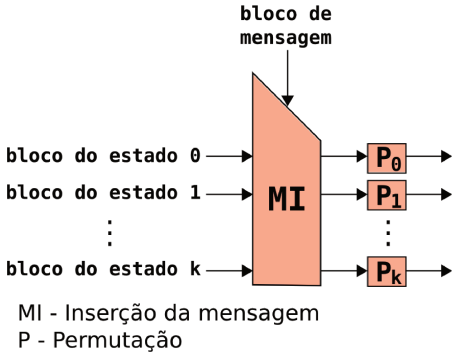


Figura 5.5: *Round function* do algoritmo Luffa

Inserção da mensagem

Nesta fase, um bloco da mensagem é misturado com as variáveis de estado. Esta mistura é realizada através de funções representadas por matrizes sobre um anel $GF((2^{32})^8)$. A função de inserção da mensagem para a versão Luffa-256 é apresentada a seguir:

Sejam H_0, H_1 e H_2 três blocos da variável de estado e M um bloco da mensagem,

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} 3 & 2 & 2 & 1 \\ 2 & 3 & 2 & 2 \\ 2 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} H_0 \\ H_1 \\ H_2 \\ M \end{pmatrix}$$

Em [39] os autores sugerem a implementação das funções de injeção da mensagem usando apenas multiplicações por x (0x02) e instruções de ou-exclusivo. Desta forma, a função apresentada acima pode ser também representada conforme a Figura 5.6.

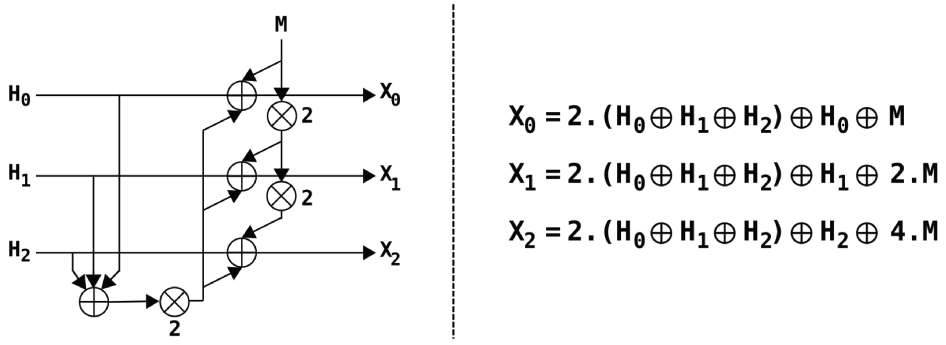


Figura 5.6: Inserção da mensagem para as versões Luffa-224 e Luffa-256

Permutações

As permutações têm entrada e saída de tamanho igual a 256 *bits*, e o seu número varia de acordo com a versão do algoritmo (ver Tabela 5.2). Cada permutação é composta por uma pequena rotação inicial na entrada (*Tweak*), que é em seguida processada por meio de oito rodadas das seguintes funções: *SubCrumb*, *MixWord* e *AddConstant*. Nesta fase, cada entrada será dividida em oito palavras de 32 *bits*.

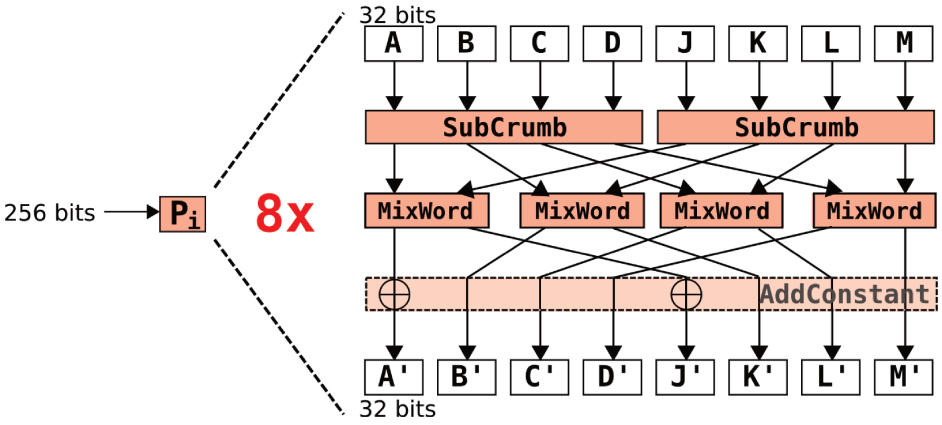


Figura 5.7: Funções da permutação de Luffa

Tweak Este passo é executado apenas uma vez a cada permutação, e é definido como uma rotação de n *bits* nas quatro últimas palavras da entrada de 256 *bits*. O valor n é estabelecido de acordo com o número de blocos do estado.

SubCrumb Esta é uma função não linear de substituição, ela toma quatro *bits* da mesma posição de quatro palavras de 32 *bits* como entrada para uma *S-box* definida a seguir.

$S[16] = \{13, 14, 0, 1, 5, 10, 7, 6, 11, 3, 9, 12, 15, 8, 2, 4\}$.

Em seguida, os *bits* que resultaram de S são retornados às quatro palavras na mesma posição de origem. Esta operação é realizada para cada um dos 32 *bits* das palavras de entrada.

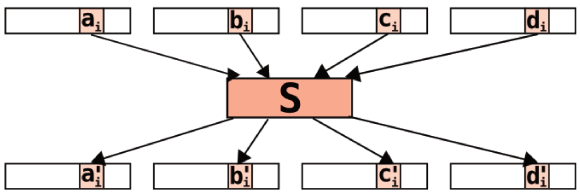


Figura 5.8: *SubCrumb* aplicada aos i-ésimos *bits* das palavras de entrada

Na especificação do algoritmo Luffa, os autores sugerem implementar a função *SubCrumb* através de operações lógicas [39], cuja sequência de instruções é descrita na Tabela 5.3.

Tabela 5.3: Instruções da função *SubCrumb* para o processador *Intel Core 2 Duo*

MOV r4 r0	**OR** r0 r1	**XOR** r2 r3
NOT r1	**XOR** r0 r3	**AND** r3 r4
XOR r1 r3	**XOR** r3 r2	**AND** r2 r0
NOT r0	**XOR** r2 r1	**OR** r1 r3
XOR r4 r1	**XOR** r3 r2	**AND** r2 r1
XOR r1 r0		

MixWord Esta função realiza uma permutação linear em duas palavras de 32 *bits* através de rotações e operações ou-exclusivo. Sua principal função é realizar a difusão dos efeitos provocados pela função *SubCrumb*. O esquema é apresentado na Figura 5.9.

AddConstant As constantes de Luffa são adicionadas (por meio da operação ou-exclusivo) com as palavras 0 e 4. Estas constantes são distintas para cada bloco do estado e para cada uma das oito iterações dentro da permutação, e consiste em dezesseis palavras de 32 *bits* para cada bloco de estado.

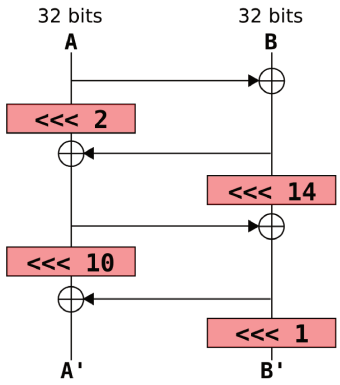


Figura 5.9: A função *MixWord*

O algoritmo executa de três a cinco permutações para cada iteração das *round functions*, dependendo da versão do Luffa. No entanto, conforme mostrado na Figura 5.5, as permutações são independentes, e podem ser executadas de maneira paralela. Este atributo será explorado nas otimizações realizadas pelos autores e descritas no Capítulo 7.

5.3.2 Análises

Durante a segunda fase o algoritmo recebeu algumas análises, entre as quais trabalhos destinados a distingui-lo de um oráculo randômico [8]. Além disso, foram publicados estudos sobre colisões em versões modificadas do algoritmo [84]. Artigos sobre pseudocolisões e pseudo-pré-imagens também foram apresentados [40, 92].

Na segunda fase, os autores publicaram uma atualização no algoritmo, na função *SubCrumb*, em relação a forma como as palavras são ordenadas na entrada da mesma [40]. Dada a relevância desta função para o algoritmo, a mudança provavelmente foi vista com preocupação pelo NIST, e pode ter sido um dos principais motivos para tirá-lo da fase final.

5.4 Shabal

5.4.1 Descrição

O algoritmo Shabal foi projetado por um grupo de pesquisadores integrados em um projeto denominado *Saphir* (*Security and Analysis of Hash Primitives*), da Agência Nacional de Pesquisa da França [33]. Shabal pode ser considerado um variante de função esponja, de fato, os autores utilizam do modelo para fazer provas de segurança. Porém, muitas características são distintas à função esponja original, como a existência de um contador, a forma de adição dos blocos de mensagem e a transformação final.

O algoritmo possui cinco versões que geram resumos de distintos tamanhos: Shabal_{192}, Shabal_{224}, Shabal_{256}, Shabal_{384} e Shabal_{512}. No entanto, as versões são quase idênticas entre si. As mudanças são basicamente no valor do estado inicial (IV) e no truncamento final.

O projeto trabalha com palavras de 32 *bits* e é dividido em três etapas: preenchimento da mensagem, rodadas da mensagem e rodadas finais.

Preenchimento da mensagem

Shabal processa blocos de mensagem de 512 *bits*, e esta fase faz com que o tamanho da mensagem original seja múltiplo de 512 *bits*. O método para completar a mensagem é bastante simples.

1. Concatenar um *bit* “1” ao final da mensagem;
2. Em seguida, concatenar *bits* “0” até que o tamanho da mensagem seja múltiplo de 512 *bits*.

Rodadas da mensagem

Esta é a principal etapa do algoritmo, onde os blocos da mensagem são processados de maneira iterada. O estado do Shabal é dividido em três estruturas (A, B e C) e um contador (W). A estrutura A é formada por 12 palavras (384 *bits*), as estruturas B e C por sua vez são formadas por 16 palavras (512 *bits*) e o contador é definido por duas palavras (64 *bits*), totalizando um estado de 960 *bits*.

A permutação usada pelo algoritmo utiliza duas chaves e é definida da seguinte forma:

Seja l_a o tamanho da estrutura A e l_m o tamanho do bloco de mensagem,
$\mathcal{P} : \{0,1\}^{l_m} \times \{0,1\}^{l_a} \times \{0,1\}^{l_m} \times \{0,1\}^{l_m} \rightarrow \{0,1\}^{l_a} \times \{0,1\}^{l_m}$

O modo de operação do Shabal funciona da seguinte maneira, primeiramente, as estruturas A,B e C são inicializadas com os valores IV de acordo com a versão do algoritmo, o contador é inicializado com o valor "1". A cada iteração, o contador é adicionado (por meio da operação ou-exclusivo) com as primeiras palavras de A. Em seguida, o bloco da mensagem é adicionado intercaladamente com B e C, ou seja, a cada iteração, o bloco é adicionado com uma destas estruturas. Logo, a permutação recebe como entrada a estrutura A e intercaladamente B ou C, as chaves são definidas pelo bloco da mensagem e as estruturas B ou C. Por fim, o bloco da mensagem é subtraído de B ou C, dependendo de qual estrutura foi usada como chave para a permutação, e o contador é incrementado.

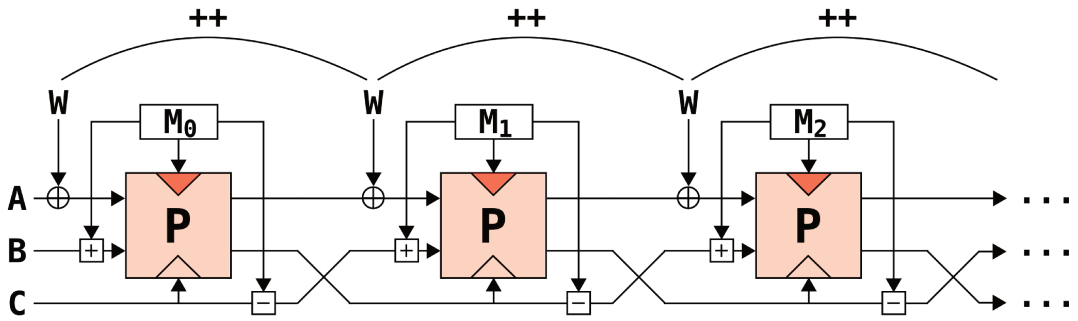


Figura 5.10: Modo de operação do algoritmo Shabal [33]

A permutação $\mathcal{P}$ A permutação do algoritmo Shabal é baseada uma construção denominada *NLFSR* (*Non-Linear Feedback Shift Register*), cujas estruturas de entrada são representações de registradores que são atualizados através de deslocamentos e operações não lineares em seus próprios elementos. A permutação é descrita no Algoritmo 7.

Podemos perceber a ocorrência de 48 rodadas dentro da construção NLFSR. A quantidade de vezes que o *loop* externo é executado é definido pelos autores como três, porém, este parâmetro é flexível, e quanto mais alto, maior será o nível de segurança. Outro valor

que pode ser alterado é o tamanho da estrutura A, com um mínimo de duas palavras, devido ao contador de 64 *bits*.

Algoritmo 7: Permutação $\mathcal{P}$

Entrada: Estruturas A,B e C e um bloco da mensagem M
Saída: Estruturas A e B

para *i=0* **até** *15* **faça**
  $B[i] \leftarrow (B[i] \lll 17)$
fim

// $\mathcal{U} : x \rightarrow 3 \times x \mod 2^{32}$
// $\mathcal{V} : x \rightarrow 5 \times x \mod 2^{32}$
para *j=0* **até** *3* **faça**
  para *i=0* **até** *16* **faça**
    $A[i + 16j \mod r] \leftarrow \mathcal{U}(A[i + 16j \mod 12]$
      $\oplus \mathcal{V}(A[i - 1 + 16j \mod 12] \lll 15)$
      $\oplus C[8 - i \mod 16])$
      $\oplus B[i + 13 \mod 16]$
      $\oplus (B[i + 9 \mod 16] \wedge \neg B[i + 6 \mod 16])$
      $\oplus M[i]$

    $B[i] \leftarrow (B[i] \lll 1) \oplus \neg A[i + 16j \mod r]$
  fim
fim

para *i=0* **até** *35* **faça**
  $A[j \mod 12] \leftarrow A[j \mod 12] + C[j + 3 \mod 16]$
fim
retorna *A e B*

Rodadas finais

Após o processamento dos blocos da mensagem, inicia-se a fase final do algoritmo. Nesta etapa, mais três iterações são realizadas. As estruturas recebem os valores resultantes da última iteração na fase passada, o bloco da mensagem é sempre o último bloco processado nas rodadas da mensagem, ou seja, o mesmo bloco é processado três vezes. O contador também recebe seu valor final da etapa anterior, mas desta vez, o mesmo não é incrementado a cada iteração.

Ao final, a estrutura C é truncada para retornar o valor do resumo.

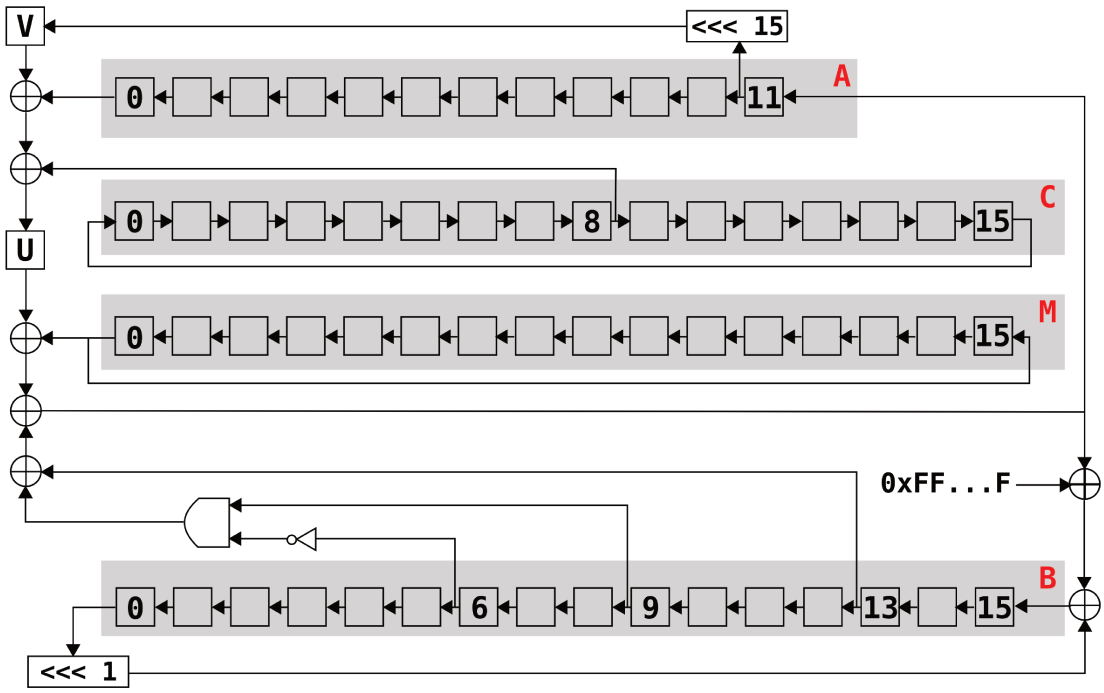


Figura 5.11: Permutação $\mathcal{P}$ do algoritmo Shabal [33]

5.4.2 Implementação

O algoritmo Shabal trabalha com palavras de 32 *bits* e possui instruções variadas, desde simples operações lógicas como ou-exclusivo (XOR), "e" (AND) e rotações, até operações aritméticas de soma e multiplicação. Como veremos, este atributo, juntamente com sua permutação NLFSR traz alguns desafios para a implementação em ambientes com recursos escassos, assim como para explorar seu paralelismo.

A memória necessária para executar o algoritmo compreende as estruturas do estado e o bloco da mensagem, pois este precisa ser subtraído após a aplicação da permutação, resultando em um total de 1984 *bits* (248 *bytes*).

Plataforma SSE 128 *bits*

A permutação $\mathcal{P}$ baseada na construção NLFSR dificulta a implementação das estruturas do estado em registradores de 128 *bits*, principalmente pelo fato de ser necessária a extração das palavras de 32 *bits*, e sua posterior inserção no momento de realizar os cálculos presentes no *loop* principal.

No entanto, é possível explorar o paralelismo nas adições e subtrações do modo de operação do Shabal. Além disso, as rotações feitas na estrutura B no início da permutação $\mathcal{P}$ também podem ser aproveitadas de forma paralela. Porém, o custo de transpor as

palavras para os registradores de 128 *bits* e retirá-los depois compensa os ganhos obtidos com a execução simultânea das operações.

Plataforma 64 *bits*

A utilização de registradores de 64 *bits* acarreta nos mesmos problemas da implementação em 128 *bits*, pois a transposição das palavras de 32 *bits* para estes registradores requer o uso de máscaras, gerando um alto custo, e neste caso, o paralelismo a ser aproveitado é menor que na plataforma SSE.

Plataforma 32 *bits*

Neste ambiente, a implementação é simples e direta, pois o algoritmo trabalha com as palavras de 32 *bits*. Conforme visto no Algoritmo 7, os elementos das estruturas NLFSR são acessados por meio da alteração de seus índices, não necessitando mover as palavras a cada iteração do *loop*.

Em seguida faremos o *profiling* do Shabal, porém, é preciso fazer duas considerações. Primeiramente, todas as versões são praticamente idênticas, portanto, não é necessário obter medições distintas a cada versão. Por fim, como a permutação não possui divisões claras, mas sim uma única função, faremos a seguinte divisão:

Modo de operação São as operações externas à permutação $\mathcal{P}$, como a soma e subtração do bloco de mensagem e das palavras do contador.

***Loop* I** É o primeiro *loop* da permutação, referente às rotações de 17 *bits* da estrutura B.

***Loop* IIa** Representa o cálculo de um elemento da estrutura A dentro do *loop* principal.

***Loop* IIb** Representa o cálculo de um elemento da estrutura B dentro do *loop* principal.

***Loop* III** É o último *loop* da permutação $\mathcal{P}$, que consiste na adição dos elementos das estruturas A e C.

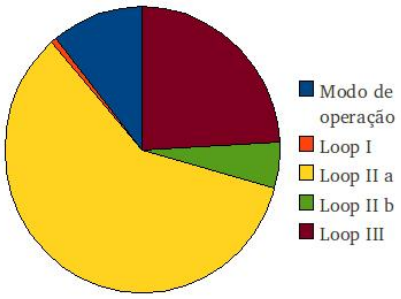


Figura 5.12: *Profiling* do algoritmo Shabal

Como se pode perceber, a operação mais custosa, representando mais de 60% do desempenho do algoritmo é o *Loop* IIa, pois além de trabalhar com vários elementos distintos, possui um grande número de operações lógicas. Ao contrário do esperado, as multiplicações das funções $\mathcal{U}$ e $\mathcal{V}$ não afetam de maneira significativa o custo do *Loop* IIa, uma vez que o compilador as otimiza por meio das instruções *lea* disponíveis na arquitetura x86.

O *Loop* III também gera um custo considerável para a permutação. Isto ocorre devido ao tamanho deste *loop* e aos distintos elementos que esta operação precisa lidar. Uma vez que não há esta quantidade de registradores disponíveis, acessos à memória são necessários.

Por fim, é importante notar que as operações cujo paralelismo podem ser mais explorados (Modo de operação e *Loop* I) não representam um custo significativo (menos de 10% do custo total), o que evidencia a ausência de paralelismo inerente ao algoritmo.

Plataforma 8 *bits*

Nas plataformas com recursos escassos o maior problema são as operações mais complexas como a adição aritmética e a multiplicação, além do tamanho do estado.

A multiplicação pode ser representada por uma pequena série de adições, estas, por sua vez, exigem instruções com *carry*, pois cada palavra representa quatro registradores de 8 *bits*.

O maior problema reside no tamanho da memória (1984 *bits*) que precisa de quase 250 registradores. O acesso de certa maneira aleatório às palavras do estado durante a permutação acentua o gasto de levar e trazer dados repetidamente à memória. O implementador nesta plataforma tem o desafio de agrupar as palavras do estado de acordo com o seu uso, para evitar demasiados acessos à memória.

Uma possível solução é reduzir o tamanho da estrutura A, o que irá diminuir o nível de segurança de acordo com os autores.

Histórico

As implementações em *software* foram feitas em sua maioria pelos autores do algoritmo, em variadas plataformas como *ARM*, AVR, x86 e *PowerPC*. Tentativas também foram feitas para implementar a permutação $\mathcal{P}$ na plataforma SSE sem extrair as palavras de 32 *bits* para processá-las [20].

Em *hardware* alguns trabalhos de implementação em *FPGA* foram publicados [48, 61].

5.4.3 Análises

A permutação $\mathcal{P}$ do algoritmo Shabal foi consideravelmente analisada, com alguns trabalhos publicados para distingui-la de uma permutação randômica [117, 5]. Além disso, ocorreram ataques de pseudocolisão e pré-imagem em versões com menores níveis de segurança (menor estrutura A) [80].

Capítulo 6

Implementação eficiente do algoritmo Keccak

Conforme visto na Seção 5.2, o algoritmo Keccak possui operações dentro de sua função de permutação que dificultam o processamento paralelo de dados por meio de instruções SSE. Nossa implementação visa usar a plataforma SSE para processar distintas mensagens independentes de forma paralela.

6.1 Aspectos Gerais

O algoritmo Keccak é constituído por apenas instruções lógicas simples: ou-exclusivo (XOR), "e"(AND), negação (NOT) e rotações. A memória usada pelo projeto basicamente corresponde ao estado, ou seja, 25 palavras de 64 *bits*, totalizando 1600 *bits* (200 *bytes*). Estas características contribuem em parte para que o projeto possa ser implementado em variadas arquiteturas.

6.1.1 Plataforma SSE 128 *bits*

A implementação do algoritmo utilizando instruções e registradores de 128 *bits* é um pouco complexo, principalmente pelo fato de que na etapa ρ os valores das rotações são distintos a cada palavra de 64 *bits* da matriz. Uma forma de se aproveitar as instruções multimídia seria colocar cada palavra de 64 *bits* em um registrador SSE, deixando a outra metade sem uso, com isso, seria possível ganhar performance na etapa χ, pois o conjunto SSE possui instruções "negação e"(AND NOT). Porém, os processadores atuais possuem apenas 16 registradores SSE, fazendo com que os valores tenham que ser armazenados na memória periodicamente, gerando um custo grande.

A melhor maneira de aproveitar as instruções é processar a mensagem de maneira paralela ao invés da forma iterada (ver Seção 2.4), outra forma seria processar diversas mensagens ao mesmo tempo. Esta técnica será detalhada na Seção 6.2.

6.1.2 Plataforma 64 *bits*

Uma vez que o algoritmo trabalha com palavras de 64 *bits*, a implementação nesta plataforma é simples e direta. A parte mais custosa do algoritmo é a função de permutação f, cujas etapas são executadas 24 vezes a cada bloco de mensagem.

A seguir apresentamos um *profiling* do algoritmo, discriminando as etapas de f, através de uma implementação básica [21] em C, usando o compilador ICC 11.1, em um computador Core 2 Duo T6400 2,00 GHz, rodando sistema *Ubuntu* 10.04 64 *bits*. Os *profiles* são calculados através do processamento de uma mensagem muito longa em cada versão do algoritmo Keccak e apresentados na Figura 6.1.

Pode-se perceber que maior custo do algoritmo se dá em sua parte não linear, ou seja, a etapa χ, mesmo com a aplicação de técnicas sugeridas pelos autores para reduzir operações lógicas, esta etapa consome quase 50% do tempo de processamento. Conforme esperado, a fase π, que envolve permutações de palavras tem um custo desprezível. Por fim, é importante ressaltar que o custo para realizar as operações particulares da função esponja é pequeno se comparado com as etapas da função f.

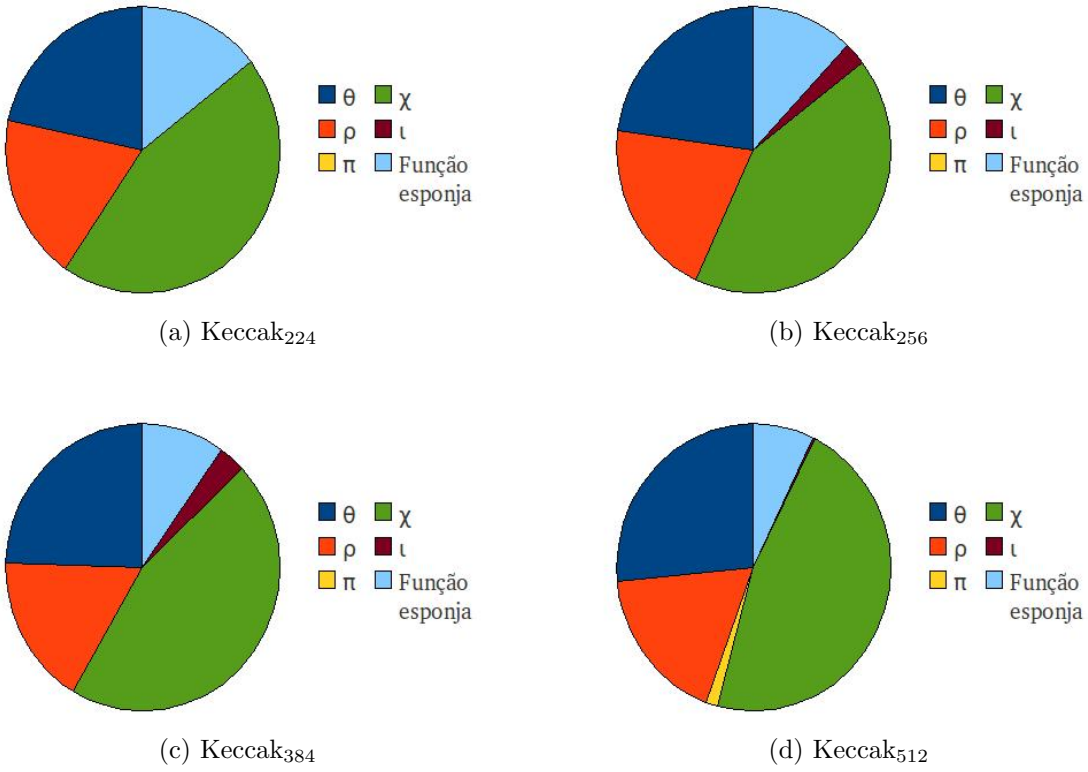
6.1.3 Plataforma 32 *bits*

Para implementar o algoritmo em máquinas de 32 *bits* os autores sugerem intercalar os *bits* das palavras da seguinte maneira, cada palavra de 64 *bits* é dividida em duas palavras de 32 *bits*, uma contendo os *bits* nas posições pares e outra os *bits* nas posições ímpares.

As operações lógicas podem ser facilmente executadas, pois as mesmas são feitas *bit* a *bit*. Este método visa facilitar as rotações presentes nas etapas θ e ρ. Com a intercalação, estas operações podem ser feitas através de rotações nas palavras de 32 *bits*, em ambos os casos possíveis.

Seja uma palavra de 64 *bits* x dividida em duas palavras de 32 *bits* u e v, sendo u contendo os *bits* pares de x e v contendo os *bits* ímpares de x,

- Rotação em um número par $2t$: fazer $u \leftarrow (u \lll t)$ e $v \leftarrow (v \lll t)$.
- Rotação em um número ímpar $2t + 1$: fazer $u \leftarrow (v \lll t + 1)$ e $v \leftarrow (u \lll t)$



(a) Keccak$_{224}$ (b) Keccak$_{256}$

(c) Keccak$_{384}$ (d) Keccak$_{512}$

Figura 6.1: *Profiling* do algoritmo Keccak

Este método também pode ser utilizado com plataformas de 16 ou 8 *bits*, porém, o número de rotações e movimentações de registradores aumenta, além disso, existe um certo custo para dividir as palavras. Semelhante técnica será utilizada para otimizar o algoritmo Luffa em 64 e 128 *bits* (ver Capítulo 7).

6.1.4 Plataforma 8 *bits*

Os principais problemas para as plataformas de 8 *bits* são a memória e as instruções de rotação. O problema da memória se resume no fato de que um estado de 1600 *bits* necessitaria de 200 registradores de 8 *bits*, o que é inadmissível nas arquiteturas de recursos escassos. Além disso, nem todas as plataformas possuem instruções de rotação com *carry*, o que dificultaria a implementação.

Uma possibilidade é pagar o preço pelo armazenamento na memória e utilizar, se disponível, instruções de deslocamento com *carry*, ou até mesmo intercalar os *bits*, assim como no método para implementar em 32 *bits*.

Outra maneira seria implementar uma versão mais reduzida do algoritmo, que utiliza palavras de 8 *bits*, em um estado com um total de 200 *bits*. Neste caso o nível de segurança é menor, e deve-se estar seguro que as aplicações podem funcionar com estes parâmetros.

6.1.5 Histórico

As implementações em *software* mais eficientes foram realizadas pelos próprios autores, que codificaram o algoritmo para várias plataformas: 64 *bits* utilizando otimizações em *Assembly*, 32 *bits* com intercalação e 8 *bits*. Além disso, implementações foram feitas para GPUs através da linguagem *CUDA*, por Hoffman [75] e Sevestre [134].

Em *hardware*, muitas implementações foram feitas em FPGA (*Field-programmable Gate Array*) e *ASIC* (*application-specific integrated circuit*) por Baldwin et. al [10], Gaj et al. [62], Guo et al. [67] e Strömbergson [140]. Kavun e Yalcin [88] realizaram uma implementação da função de permutação f em um chip RFID, mostrando que é possível aplicar o algoritmo em ambientes com recursos restritos.

6.2 Técnica proposta

Primeiramente definimos uma estrutura de dados onde cada vetor SSE de 128 *bits* aloca duas palavras de duas variáveis de estados distintas. As operações são realizadas da mesma maneira que a implementação original na plataforma de 64 *bits*, porém agora com instruções lógicas SSE.

Assim, para n mensagens distintas, precisamos dispor de $\lceil n/2 \rceil \times 25$ variáveis SSE para manter o estado.

Nossa implementação baseada no código dos autores utilizando apenas registradores de 64 *bits* para processar uma mensagem por vez atingiu um desempenho de 19 ciclos/*byte*. Assim, podemos concluir que o processamento paralelo por meio de vetores SSE não foi vantajoso.

Como principais motivos pelo baixo desempenho da implementação paralela podemos citar:

- O tamanho do estado faz com que os dados da memória devam ser transferidos periodicamente para os registradores SSE.

- Existem poucas operações nas quais as instruções especiais SSE possam se beneficiar. Por exemplo, na etapa ρ podemos encontrar apenas duas rotações por *bytes*.

Tabela 6.1: *Profiling* da implementação paralela do algoritmo Keccak_{256}, mensagem longa no Intel Core 2 Duo T6400 2,00 GHz

Função	**Custo (ciclos/*byte*)**
$\theta(Theta)$	6,66
$\rho(Rho)$	4,90
$\pi(Pi)$	2,95
$\chi(Chi)$	4,90
$\iota(Iota)$	0,09
Operações Função Esponja	0,99
Total	20,49

6.2.1 Implementação *bitslice*

Para aproveitar melhor a plataforma SSE, propomos uma implementação que processe oito mensagens de uma vez utilizando uma técnica denominada *bitslicing*, publicada por Eli Biham para melhorar o desempenho do algoritmo DES em *software* [25].

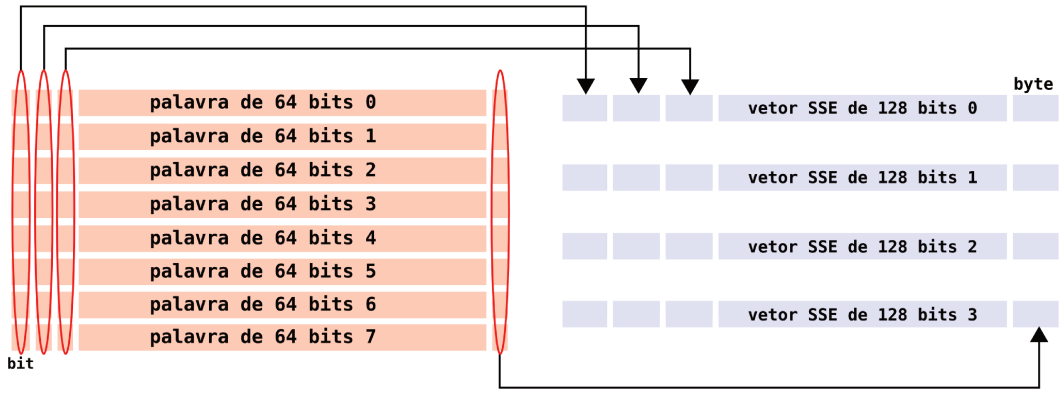


Figura 6.2: Técnica *bitslicing* aplicada no algoritmo Keccak

Uma vez que o algoritmo possui operações lógicas apenas entre palavras (exceto as rotações que são realizadas entre *bits* de uma mesma palavra), é possível implementar as instruções do algoritmo original nesta nova forma de representar os dados. Além disso, as rotações da etapa ρ serão todas em *bytes*, podendo ser processadas por meio das instruções SSE *alignr*, conforme esquema a seguir.

Na nova representação, cada palavra de 64 *bits* estará alocada em quatro vetores SSE, que por sua vez, retém oito palavras. Seja $b_{i,j}$ o *bit* i da palavra j e r_i um vetor SSE,

$r_0 = (b_{0,0}; b_{0,1}; b_{0,2}; b_{0,3}; b_{0,4}; b_{0,5}; b_{0,6}; b_{0,7}; b_{1,0}; b_{1,1}; b_{1,2}; b_{1,3}; \ \ldots \ b_{15,5}; b_{15,6}; b_{15,7});$
$r_1 = (b_{16,0}; b_{16,1}; b_{16,2}; b_{16,3}; b_{16,4}; b_{16,5}; b_{16,6}; b_{16,7}; b_{17,0}; b_{17,1}; \ \ldots \ b_{31,5}; b_{31,6}; b_{31,7});$
$r_2 = (b_{32,0}; b_{32,1}; b_{32,2}; b_{32,3}; b_{32,4}; b_{32,5}; b_{32,6}; b_{32,7}; b_{33,0}; b_{33,1}; \ \ldots \ b_{47,5}; b_{47,6}; b_{47,7});$
$r_3 = (b_{48,0}; b_{48,1}; b_{48,2}; b_{48,3}; b_{48,4}; b_{48,5}; b_{48,6}; b_{48,7}; b_{49,0}; b_{49,1}; \ \ldots \ b_{63,5}; b_{63,6}; b_{63,7}).$

Desta forma, para rotacionar à direita as oito palavras em um *bit*, deve-se rotacionar um *byte* à direita entre os vetores SSE.

aux = **_mm_alignr_epi8**(r1, r0, 1);
r1 = **_mm_alignr_epi8**(r2, r1, 1);
r2 = **_mm_alignr_epi8**(r3, r2, 1);
r3 = **_mm_alignr_epi8**(r0, r3, 1);
r0 = **_mm_load_si128**(aux);

Tabela 6.2: *Profiling* da implementação *bitslice* do algoritmo Keccak_{256}, mensagem longa no Intel Core 2 Duo T6400 2,00 GHz

Função	**Custo (ciclos/*byte*)**
$\theta(Theta)$	5,86
$\rho(Rho)$	4,29
$\pi(Pi)$	2,85
$\chi(Chi)$	4,59
$\iota(Iota)$	0,19
Transformação *Bitslice*	1,09
Operações Função Esponja	1,20
Total	20,07

Podemos perceber que a nova implementação teve uma melhora de desempenho muito pequena, pois os problemas da implementação paralela ainda persistem. Além disso, rotações de valores maiores que 15 *bits* (metade das rotações da etapa ρ) obrigam ao implementador efetuar um deslocamento nos vetores antes de processar as rotações, diminuindo os ganhos obtidos com a instrução *alignr*. Por fim, conforme visto na Seção 6.1.2, as rotações da etapa ρ correspondem a uma fração pequena do custo total da função f.

Uma possibilidade de se melhorar a técnica seria efetuar uma implementação em *Assembly*, com o objetivo de minimizar o deslocamento memória-registradores. Com o surgimento de novas instruções AVX (ver Seção 4.2), os problemas das rotações maiores de 15 *bits* também serão eliminados.

Capítulo 7

Implementação eficiente do algoritmo Luffa

O principal foco para otimizar o algoritmo Luffa são as subpermutações independentes, que propiciam o paralelismo de dados por meio de registradores de 64 ou 128 *bits*. Para alcançar este objetivo, funções do algoritmo precisam ser modificadas, além disso, novas estruturas devem ser adicionadas.

Também podemos observar nos *profilings* do algoritmo Luffa na Seção 7.1.3 que a função mais custosa é a *MixWord*, responsável por difundir as transformações ocorridas na fase anterior, a função *SubCrumb*. A partir da tentativa em implementar as operações de *MixWord* de forma eficiente em registradores maiores de 32 *bits* obtemos melhoras em mais de 32% usando registradores de 64 *bits* e 16% na plataforma SSE. Com o uso da linguagem *Assembly*, conseguimos resultados ainda melhores, obtendo assim a implementação mais rápida do algoritmo [20].

No ambiente de 8 *bits* da plataforma AVR também conseguimos em uma implementação eficiente, transferindo o estado reduzido das subpermutações para os 32 registradores de 8 *bits*.

7.1 Aspectos gerais

Assim como o algoritmo Keccak, Luffa possui apenas instruções lógicas simples, "e"(AND), "ou"(OR), ou-exclusivo (XOR), negação (NOT) e rotações, permitindo a implementação em variadas arquiteturas. A memória gasta pelas variáveis de estado varia entre as versões do algoritmo, desde 768 *bits* (96 *bytes*) no Luffa-224/256 até 1280 *bits* (160 *bytes*) no Luffa-512. Deve-se considerar também o gasto para realizar a inserção da mensagem, que inclui o armazenamento de um bloco de mensagens de 256 *bits* (32 *bytes*). Desta forma, em algumas plataformas mais restritas, é possível que apenas a versão Luffa-224/256 seja

viável de se implementar.

7.1.1 Plataforma SSE 128 *bits*

A implementação do algoritmo em uma plataforma SSE é bastante direta, uma vez que o algoritmo trabalha com palavras de 32 *bits* e possui permutações paralelas. Na especificação dos autores, esta implementação é considerada a mais rápida, e provavelmente, o algoritmo foi projetado pensando-se nesta arquitetura.

No entanto, existem dois revezes nesta plataforma, o primeiro é o fato de que um registrador de 128 *bits* suporta quatro palavras de 32 *bits*, com isso, o aproveitamento total dos registradores se dá apenas na versão Luffa-384, que possui quatro permutações simultâneas. Outro problema são as rotações com valores não múltiplos de *byte*. A plataforma SSE não possui uma instrução para rotacionar *bits*, por isso, é necessário executar múltiplas instruções para produzir a operação. Mais detalhes da implementação neste ambiente serão apresentados na Seção 7.2.

7.1.2 Plataforma 64 *bits*

Neste ambiente, a implementação não é muito direta, pois apesar de ser possível processar duas permutações de uma só vez, as dificuldades para realizar rotações reduz substancialmente o desempenho do algoritmo. A resolução deste problema nos levou a desenvolver a otimização descrita na Seção 7.2.

7.1.3 Plataforma 32 *bits*

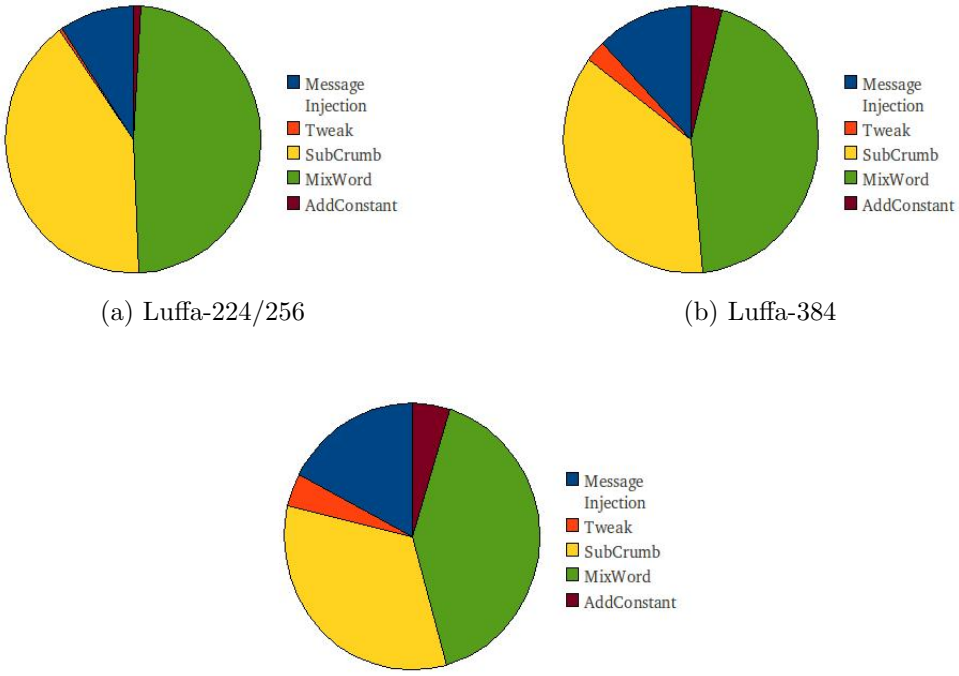
O algoritmo Luffa trabalha com palavras de 32 *bits*, e portanto, sua implementação neste ambiente é direta e eficiente, principalmente pelo fato das rotações poderem ser executadas com uma instrução apenas. No entanto, nesta arquitetura, o paralelismo inerente às permutações independentes não pode ser aproveitado.

A seguir apresentamos o *profiling* do algoritmo nesta plataforma de uma implementação básica [39] em C, usando o compilador ICC 11.1, em um computador Core 2 Duo T6400 2,00 GHz, rodando sistema Ubuntu 10.04 64 *bits*. Os *profiles* são calculados através do processamento de uma mensagem muito longa em cada versão do algoritmo Luffa e apresentados na Figura 7.1.

Inicialmente, pode-se notar o impacto das funções *SubCrumb* e *MixWord*, correspondendo mais de 60% do custo do algoritmo. Estas funções são bastante simples, portanto difíceis de serem otimizadas. É importante ressaltar também o pequeno acréscimo de custo que a fase *message injection* vai tomando a cada versão. Isto é explicado pelo maior

grau de complexidade para misturar um bloco de mensagem em variáveis de estado mais numerosas.

A complexidade da inserção da mensagem deriva da presença de permutações independentes, ou seja, esta etapa foi uma forma que os autores pensaram para tornar os dados resultantes das permutações dependentes entre si e dos blocos da mensagem. Visto de outra forma, pode-se comparar o *message injection* com as operações do algoritmo Keccak externas à permutação f (ver Seção 6.1).



(a) Luffa-224/256

(b) Luffa-384

(c) Luffa-512

Figura 7.1: *Profiling* do algoritmo Luffa

7.1.4 Plataforma 8 *bits*

A restrição de memória e de registradores comprometem a viabilidade de implementar as versões Luffa-384 e Luffa-512. As premissas básicas para uma implementação eficiente do Luffa é que a arquitetura tenha suporte para rotações (ou deslocamentos com carry), e as instruções lógicas básicas.

Uma vantagem presente no Luffa é a independência das permutações, dividindo o estado em blocos de 256 *bits*, o que equivale a 32 registradores de 8 *bits*. Com isso, é possível processar as etapas mais custosas do projeto sem a necessidade de acessar a

memória constantemente. Detalhes de otimizações para esta plataforma serão discutidos na Seção 7.4.

7.1.5 Histórico

Desde a submissão do algoritmo ao concurso SHA-3, houveram diversas implementações do Luffa, tanto em *software* quanto em *hardware*. Os autores realizaram implementações para plataformas de 8, 32 e 128 *bits* [40], explorando quando possível o paralelismo inerente às permutações. Pornin também publicou uma implementação em C e Java, incluindo em sua biblioteca *sphlib* [122]. A implementação mais rápida do algoritmo foi realizada por Oliveira e López [119] (ver Tabela 7.1), cujos detalhes serão vistos na Seção 7.2. Os autores em [118] disponibilizaram implementações para GPUs, processando várias mensagens diferentes.

A seguir, a posição do Luffa com relação ao desempenho em *software* em comparação com os algoritmos da segunda fase do concurso SHA-3. Os resultados foram obtidos da página eBASH [20], acessada em dezembro de 2010.

Tabela 7.1: Desempenho Luffa, mensagem longa, Intel Core 2 Duo E8400, 3,137 GHz, 64 *bits*

Algoritmo	**Ciclos/*byte***
Blue Midnight Wish	5.16
Shabal	5.88
Skein	6.46
Blake	6.93
Luffa	**9.38**
SIMD	9.68
CubeHash	10.23
SHA-2	10.27
Keccak	10.95
JH	16.91
Fugue	17.48
Grøstl	21.33
SHAvite-3	22.43
ECHO	24.32
Hamsi	28.81

Em *hardware* o algoritmo teve um bom desempenho, demonstrado pelos trabalhos de [94, 141, 93, 113], com implementações em FPGA e ASIC.

7.2 x86

7.2.1 *Perfect Shuffle*

A técnica *perfect shuffle* é uma permutação de *bits* que entrepõe perfeitamente os *bits* de duas ou mais palavras [150]. Esta operação aplicada em duas palavras de 32 *bits* é ilustrada na Figura 7.2.

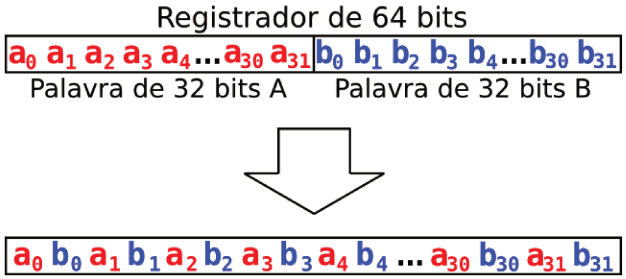


Figura 7.2: Técnica *perfect shuffle* aplicada em duas palavras de 32 *bits* alocadas dentro de um registrador de 64 *bits*

Algumas operações, como as rotações, se tornam mais simples com a técnica *perfect shuffle*. Por exemplo, para realizar uma rotação à esquerda de duas palavras A e B de 32 *bits* representadas na Figura 7.2 em n *bits* sem aplicar a técnica *perfect shuffle*, precisaríamos das seguintes instruções em na arquitetura x86-64:

//rax mantém o registrador C e rbx é um registrador auxiliar

mov rbx, rax
shl rax, \$$n$
shr rbx, \$(32-n)
and rax, MASK0 *//É necessário aplicar máscaras para limpar*
and rbx, MASK1 *//os bits que ultrapassaram os limites das palavras.*
xor rax, rbx

O que requer seis instruções. Contudo, com os *bits* das duas palavras de 32 *bits* entrepostos, uma rotação à esquerda em n *bits* pode ser feita com apenas uma instrução:

//rax mantém o registrador C

rotl rax, \$(2*n)

Uma operação similar, chamada de *bit permutation*, é proposta pelos autores do algoritmo Keccak [22] para acelerar implementações em plataformas menores que 64 *bits*. Além disso, na cifra de bloco *Serpent* [3], os autores utilizam a técnica apresentada nesta seção para a implementação voltada à *hardware*.

7.2.2 Técnica proposta

Nesta seção explicaremos como o método *perfect shuffle* foi usado juntamente com os registradores e instruções das plataformas de 64 *bits* e SSE para promover otimizações no algoritmo Luffa.

Plataforma 64 *bits*

Neste cenário, o paralelismo de dados é possível por meio da alocação de duas palavras de 32 *bits* de diferentes blocos de cadeia em um registrador de 64 *bits*, conforme descrito na Figura 7.3. Desta forma, podemos processar duas subpermutações simultaneamente.

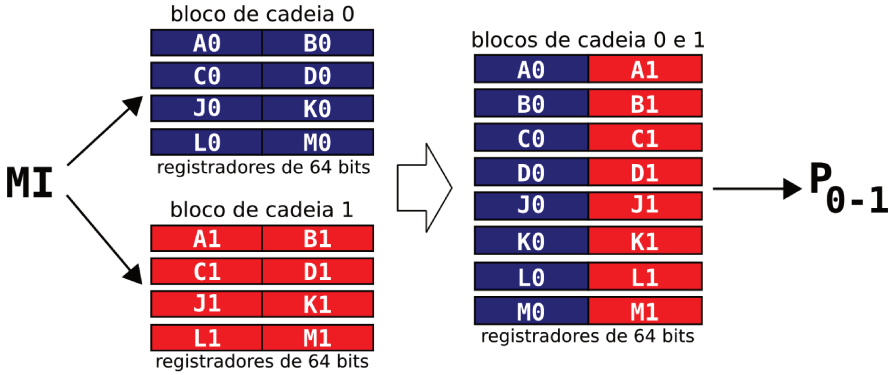


Figura 7.3: Paralelismo em 64 *bits*

Porém, esta maneira de dispor as duas palavras de 32 *bits* causa um grande custo no processamento das rotações, que são frequentemente executadas no Luffa, principalmente na função *MixWord*. O custo desta aumentou em 139% em comparação com a implementação de 32 *bits* dos autores. A principal razão deste acréscimo foi o número de instruções necessárias para rotacionar as duas palavras, conforme mostrado na Seção 7.2.1.

Através da aplicação da técnica *perfect shuffle*, podemos reduzir o custo das rotações para apenas uma instrução. Para diminuir o custo de *perfect shuffle*, nós aplicamos a técnica apenas uma vez por rodada, no bloco de mensagem, mantendo a transformação através do algoritmo. A transformação reversa é feita apenas uma vez, antes de computar o valor do resumo. Este esquema é mostrado na Figura 7.4.

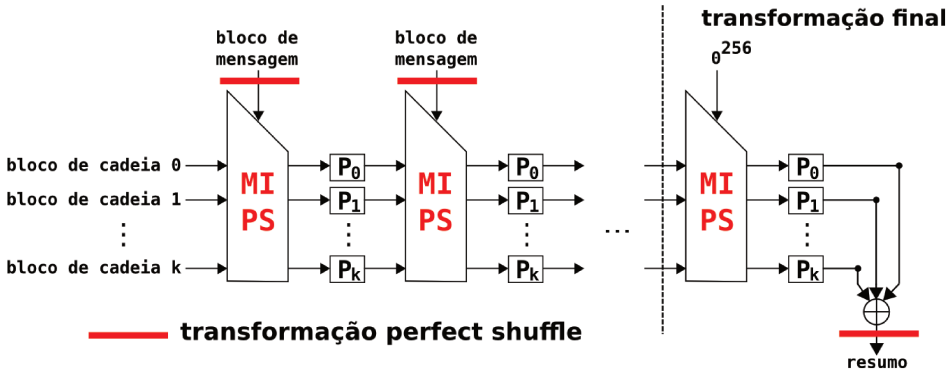


Figura 7.4: A técnica *perfect shuffle* aplicada no algoritmo Luffa

O custo extra desta implementação consiste em duas operações: dispor os registradores da conforme a Figura 7.3, o que custa aproximadamente 2 ciclos/*byte*; e a técnica *perfect shuffle* aplicada no bloco de mensagem, custando 3 ciclos/*byte* usando a implementação sugerida em [150]. Portanto, o custo extra total resulta em aproximadamente 5 ciclos/*byte*.

A versão Luffa-384 é a mais beneficiada por esta técnica, pois necessita processar quatro subpermutações em quatro blocos de cadeia, ocupando completamente dois registradores de 64 *bits*. Para esta versão o desempenho foi melhorado em 32%.

Plataforma SSE

Neste ambiente dispomos de registradores de 128 *bits* que podem alocar quatro palavras de 32 *bits*. Consequentemente, quatro subpermutações podem ser processadas ao mesmo tempo. O método dos autores em alocar as palavras nos registradores SSE é representado na Figura 7.5.

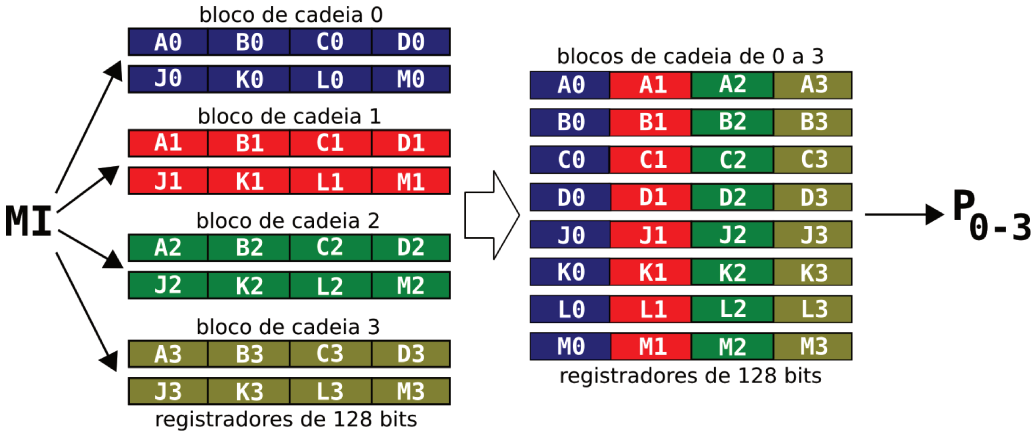


Figura 7.5: O paralelismo SSE

Na função *MixWord*, observamos quatro tipos de rotações: 1, 2, 10 e 14 *bits*. Com as instruções SSE, podemos rotacionar em n *bits* quatro palavras de 32 *bits* independentemente e sem máscaras:

```
//r0 mantém o dado a ser rotacionado
r0 = _mm_xor_si128(_mm_slli_epi32(r0, n), _mm_srli_epi32(r0, 32−n));
```

No entanto, se aplicarmos a técnica *perfect shuffle* no registrador de 128 *bits*, entrepondo as quatro palavras de 32 *bits*, conforme mostrado na Figura 7.6, as rotações de *MixWord* se tornam 4, 8, 40 e 56 *bits*. Três tipos de rotações podem ser implementadas por meio de instruções *shuffle* em *bytes*, com uma máscara adequada, com um custo de apenas uma instrução. Como exemplo, uma rotação à esquerda de 8 *bits* (1 *byte*):

```
//r0 mantém o dado a ser rotacionado
mascara = _mm_set_epi8(0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08, 0x07,
                       0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00, 0x0F);

r0 = _mm_shuffle_epi8(r0, mascara);
```

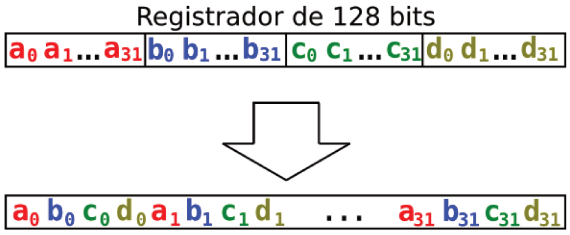


Figura 7.6: Técnica *perfect shuffle* aplicada em quatro palavras de 32 *bits* alocadas dentro de um registrador de 128 *bits*

Neste cenário realizamos dois tipos de implementações: aplicando *perfect shuffle* juntamente com o método de paralelismo de dados dos autores; e aplicando apenas *perfect shuffle*. No primeiro caso a implementação é direta, procedendo assim como o cenário de 64 *bits*. No entanto, no segundo caso precisamos mudar a forma de implementar a função *SubCrumb*. Devido à nova disposição dos *bits* dentro dos registradores, usamos uma tabela através de instruções *shuffle* (ver Seção 4.1.4) ao invés de usar operações lógicas sugeridas pelos autores.

Além disso, temos que usar uma distinta abordagem para implementar a multiplicação por x processada na fase da inserção da mensagem.

SubCrumb A função *SubCrumb* recebe como entrada oito palavras de 32 *bits*, porém as quatro últimas são dispostas em diferente ordem.

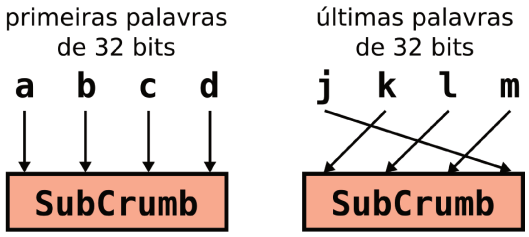


Figura 7.7: Entrada da função *SubCrumb*

Mudar a ordem dentro dos registradores $[JKLM]$ é inviável devido ao custo, assim, usamos duas tabelas para lidar com distintos ordenamentos da entrada.

Inserção da mensagem A multiplicação por x (0x02) no anel $GF((2^{32})^8)$ com o polinômio irredutível $\phi(x) = x^8 + x^4 + x^3 + x + 1$ é frequentemente usado na fase de inserção da mensagem (ver Seção 5.3.1).

Um elemento $a = A + Bx + Cx^2 + Dx^3 + Jx^4 + Kx^5 + Lx^6 + Mx^7 \in GF((2^{32})^8)$ pode ser representado usando dois registradores SSE de 128 *bits*. A operação $a.x \mod \phi(x)$ pode ser computada da seguinte forma:

$$a.x = Ax + Bx^2 + Cx^3 + Dx^4 + Jx^5 + Kx^6 + Lx^7 + Mx^8$$
$$\text{uma vez que } x^8 \equiv x^4 + x^3 + x + 1 \mod \phi(x) \text{ temos}$$

$$\begin{aligned} a.x &\equiv Ax + Bx^2 + Cx^3 + Dx^4 + Jx^5 + Kx^6 + Lx^7 \\ &\quad + M(x^4 + x^3 + x + 1) \mod \phi(x) \\ &\equiv M + (A + M)x + Bx^2 + (C + M)x^3 + (D + M)x^4 \\ &\quad + Jx^5 + Kx^6 + Lx^7 \mod \phi(x). \end{aligned}$$

Na Figura 7.8 representamos a multiplicação por x. Em termos de registradores e instruções SSE, a implementação envolve alguns deslocamentos e operações de ou-exclusivo nas palavras de 32 *bits*.

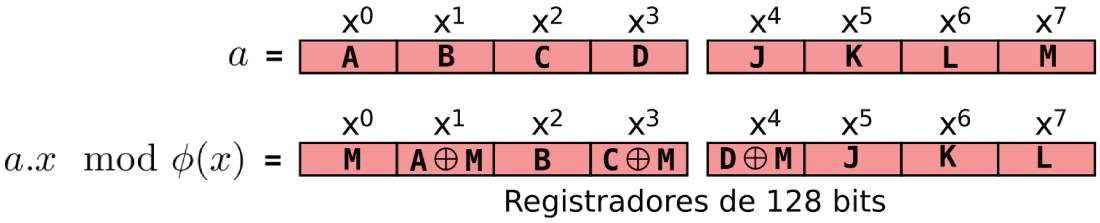


Figura 7.8: A multiplicação $a.x \mod \phi(x)$ representada por registradores de 128 *bits*

Quando um elemento a é representado usando a técnica *perfect shuffle*, a implementação de $a.x$ é um pouco mais complexa, o que requer operações extras a nível de *bits*. Porém, o aumento no custo não é substancial.

Obtemos bom desempenho na versão Luffa-256, com 15% de melhora na velocidade com o compilador Intel (ICC), A versão Luffa-512 pode ser implementada misturando os dois tipos de implementação; os primeiros quatro blocos foram implementados usando *perfect shuffle* e paralelismo de dados, enquanto o último bloco é melhor implementado usando apenas *perfect shuffle*.

7.2.3 Otimizações na linguagem *Assembly*

Nossa implementação também foi otimizada usando a linguagem *Assembly*, alcançando até 20% de aumento no desempenho em relação a nossa melhor implementação em SSE. As técnicas usadas foram baseadas em informações coletadas em [46, 114, 60] e em experimentação prática. Todas as otimizações foram direcionadas para a arquitetura Intel Core 2.

Registradores SSE sem prefixo adicional

Registradores SSE *xmm8* até *xmm15* são acessados usando prefixos adicionais na instrução. Estes registradores devem ser evitados se possível para alcançar um melhor desempenho na fase de pré-decodificação do *pipeline* do processador Core 2.

Instruções inteiras e de ponto flutuante

A Intel recomenda em [114] não misturar instruções SSE inteiras e de ponto flutuante no mesmo registrador, devido ao atraso para mover dados de diferentes unidades de execução. Esta penalidade é chamada de atraso de desvio (do inglês *bypass delay*), e na arquitetura Core 2 é esperado um atraso de 1 ciclo [60].

No entanto, as instruções para vetores de ponto flutuante de precisão simples estão presentes desde versões mais antigas do SSE. Além disso, usar instruções lógicas de ponto flutuante em valores inteiros resulta o mesmo valor originado por instruções inteiras.

Por experimentação, concluímos que é preferível usar instruções de ponto flutuante quando possível em processadores Core 2. No Luffa-512, a implementação de ponto flutuante foi 9,68% mais rápida do que o código usando apenas instruções inteiras.

Porém, testes em um processador Core i7 apontaram para outra conclusão. Seu atraso de desvio é mais alto [60], portanto, a implementação usando instruções SSE inteiras foi mais rápida, com uma diferença de até 33,90% (Luffa-512).

Reordenando instruções

Processadores Core 2 possuem três unidades de execução para operações lógicas e aritméticas, uma para ler e outra para escrever dados da memória e um para calcular endereços da mesma. Manter as unidades de execução sempre ocupadas é uma tarefa difícil e requer experimentação. Misturar operações de lógica e acesso a memória pode ajudar.

movaps xmm2, table0
pshufb xmm2, xmm0
movaps xmm0, table1
pshufb xmm0, xmm1
xorps xmm0, xmm2

Máscaras para economizar instruções

Usar máscaras alocadas na *cache* pode ser mais rápido do que computar repetidas instruções lógicas. Por exemplo, para realocar os *bits* dos registradores SSE conforme a técnica *perfect shuffle* é necessário trocar partes dos registradores, removendo-as antes de trocar.

Esta técnica também contribui para uma maior ocupação das unidades de execução.

//xmm0 mantém os dados principais. xmm1 e xmm2 são registradores auxiliares

//Implementação básica	//Implementação eficiente
movaps xmm1, xmm0	**movaps** xmm1, xmm0
movaps xmm2, xmm1	**movaps** xmm2, xmm0
andps xmm1, mask1 `;0x0000F0F0`	**andps** xmm1, mask1 `;0x0000F0F0`
...	...
andps xmm2, mask2 `;0x0F0F0000`	**andps** xmm2, mask2 `;0x0F0F0000`
...	...
xorps xmm0, xmm1	**andps** xmm0, mask1+2 `;0xF0F00F0F`
xorps xmm0, xmm2	...

Shuffle para mover registradores

A instrução *shuffle* pode ser usada para evitar mais instruções. Por exemplo, para rotacionar à esquerda quatro *bits* no último passo da função *MixWord*.

```
//xmm0 mantém os dados a serem rotacionados. xmm1 é um registrador auxiliar

//Implementação básica
movaps xmm1, xmm0 ; mov. p/ reg. aux.
psrlq xmm1, 60 ; deslocar 60 bits à dir.
pshufd xmm1, 0xC6 ; trocar palavras
psllq xmm0, 4 ; deslocar 4 bits à esq.
xorps xmm0, xmm1 ; ou-excl. entre regs.

//Implementação eficiente
pshufd xmm1, xmm0, 0x6C ; trocar regs.
psrlq xmm1, 60 ; deslocar 60 bits à dir.
psllq xmm0, 4 ; deslocar 4 bits à esq.
xorps xmm0, xmm1 ; ou-excl. entre regs.
```

Parada de registradores

Durante a fase de renomeação de registradores do *pipeline* do processador Core 2, um evento denominado parada de registradores (do inglês *register stall*) pode ocorrer. Ele causa um atraso por causa de registradores que são lidos frequentemente mas raramente escritos. Em nossa implementação este efeito foi evitado executando diferentes instruções das funções *SubCrumb* e *MixWord* sequencialmente, ao invés de as intercalar.

```
/*Possível parada de registradores em
xmm2, xmm5 e xmm8*/
pshufb xmm2, xmm0
pshufb xmm5, xmm3
pshufb xmm8, xmm6
movaps xmm0, table1
movaps xmm3, table1
movaps xmm6, table1
pshufb xmm0, xmm1
pshufb xmm3, xmm4
pshufb xmm6, xmm7
xorps xmm0, xmm2
xorps xmm3, xmm5
xorps xmm6, xmm8

/*Por experimentação, esta
implementação resultou em melhores
resultados*/
pshufb xmm2, xmm0
movaps xmm0, table1
pshufb xmm0, xmm1
xorps xmm0, xmm2
pshufb xmm5, xmm3
movaps xmm3, table1
pshufb xmm3, xmm4
xorps xmm3, xmm5
pshufb xmm8, xmm6
movaps xmm6, table1
pshufb xmm6, xmm7
xorps xmm6, xmm8
```

Carregar dados provenientes dos primeiros acessos à memória

Os primeiros acessos a localizações da memória (por exemplo, máscaras e constantes únicas) deve ser feitos antes do uso de seus dados. Esta otimização pode ser feita por meio da alocação de dados em registradores livres, se possível.

7.2.4 Resultados experimentais

Nesta parte apresentamos os resultados de nossas implementações do algoritmo Luffa para plataformas de 64 *bits* e SSE. Para a plataforma de 64 *bits* o código foi escrito em C, enquanto para a plataforma SSE implementações em C e *Assembly* foram realizadas. Os programas foram compilados com ICC v.11.1, GCC v.4.3 (Ubuntu 10.04 64 *bits*, kernel 2.6.32-24) e Visual Studio 2010 C++ Compiler (Windows Vista Business Edition 64-bit). Para o código em *Assembly*, foi usado GNU Assembler. Os códigos foram executados em uma máquina Intel Core 2 Duo T6400 2.00 GHz. As opções de compilador usadas foram `-O2` para ICC, `-O3` e `-march=core2` para GCC. Para a plataforma cd 64 *bits*, a opção `-no-vec` (`-mno-sse` para GCC) foi usada para impedir os compiladores a usarem instruções SSE. O compilador Visual Studio foi executado com as opções `\O2`, `\GL` e `\favor:INTEL64`.

Poucas diferenças foram percebidas entre os compiladores *GCC*, *ICC* e *Visual Studio*. GCC requer mais ajustes de código e produz códigos menos eficientes na plataforma SSE. Na plataforma de 64 *bits*, os três compiladores apresentaram comportamentos similares. O compilador GCC foi escolhido por sua popularidade e também por ser frequentemente usado em medições. ICC foi selecionado por sua boa performance em instruções SSE. O compilador Visual Studio foi definido pelo NIST como padrão para avaliação dos algoritmos em *software*.

O desempenho foi medido nas seguintes versões do algoritmo Luffa para 64 *bits*: Referência, o código proposto pelos autores em seu pacote de submissão ao NIST (*"ANSI C" in 64-bit mode*) [40]; Básica, nossa implementação baseada nas otimizações usadas pelo código dos autores; Paralela, nossa implementação usando duas palavras de 32 *bits* em um registrador de 64 *bits* executando duas subpermutações em paralelo; e Paralela+PS, nossa implementação baseada na técnica *perfect shuffle* combinada com a execução de duas subpermutações em paralelo.

Para manter a consistência, o desempenho de nossa implementação é comparado com a nossa implementação Básica. Além disso, apresentamos o desempenho do código dos autores, devido ao fato de que este é o código referência usado pelo NIST para sua avaliação. Nas Tabelas 7.2 e 7.3, são dados os resultados de desempenho para três versões do Luffa: Luffa-256, Luffa-384 e Luffa-512. O algoritmo de Luffa-224 é idêntico ao Luffa-256, com um truncamento antes de apresentar o valor de resumo. O texto em negrito representa a implementação mais rápida.

Tabela 7.2: Plataforma de 64 *bits* (ciclos/*byte*), mensagem longa no Intel Core 2 Duo

Implementação	**ICC**			**GCC**			**Visual Studio 2010**		
	256	384	512	256	384	512	256	384	512
Referência [40]	26,81	41,46	57,31	27,65	41,80	58,49	27,46	42,03	57,90
Este trabalho:									
Básica	24,31	33,09	41,50	24,43	34,35	46,35	25,34	36,37	46,03
Paralela	33,25	35,28	52,25	35,58	36,83	56,60	51,53	53,81	79,75
Paralela+PS	**20,00**	**22,47**	**35,09**	**20,39**	**23,89**	**35,85**	**22,62**	**25,37**	**37,34**

Um custo extra pode ser percebido na implementação Paralela. Este custo é consideravelmente reduzido ao aplicar a técnica *perfect shuffle*, obtendo melhoras de desempenho em até 32% (Luffa-384 ICC) em relação à implementação Básica.

As implementações para a plataforma SSE são apresentadas da seguinte forma: Referência, o código proposto pelos autores em seu pacote de submissão ao NIST (*"ANSI C" in 64-bit mode*); Básica[1], nossa implementação baseada na otimização usada pelo código dos autores; PS, nossa implementação baseada na técnica *perfect shuffle*; e Paralela+PS, nossa implementação com a técnica *perfect shuffle* combinada com a execução de quatro subpermutações em paralelo.

Tabela 7.3: Plataforma SSE (ciclos/*byte*), mensagem longa no Intel Core 2 Duo

Implementação	**ICC**			**GCC**			**Visual Studio 2010**		
	256	384	512	256	384	512	256	384	512
Referência [40]	17,96	20,93	31,34	16,59	17,96	32,03	17,15	19,09	28,62
Este trabalho:									
Básica	13,84	**14,78**	19,87	**15,28**	**16,18**	**24,81**	20,06	21,28	27,18
PS	**11,75**	16,22	22,56	15,31	20,90	28,15	**15,03**	**18,40**	**23,84**
Paralela+PS	12,34	14,96	**19,81***	15,81	18,15	25,31*	20,81	24,18	30,09*

(*) Esta implementação usa uma mistura de PS e Paralela+PS. Ver Seção 7.2.2.

Até 15% de melhora no desempenho pode ser alcançado na versão Luffa-256 (ICC), além de uma nova maneira de representar a função *SubCrumb*, explorando instruções SSE. Consideramos que a implementação PS permitiu os compiladores ICC e Visual Studio a otimizarem melhor o código. GCC não conseguiu aproveitar do novo método e gerou códigos com desempenho similar. Também pode ser observado que, apenas para o Visual Studio, o código Referência foi melhor que nossa implementação Básica.

[1]A implementação da função *MixWord* foi modificada para melhorar as otimizações realizadas pelo compilador Visual Studio. Consequentemente, os códigos gerados pelo ICC e GCC tornaram-se um pouco mais lentos.

Os códigos em *Assembly* são apresentados na seguinte forma; PS, nossa implementação baseada na técnica *perfect shuffle* e Paralela+PS, nossa implementação com a técnica *perfect shuffle* combinada com a execução de quatro subpermutações em paralelo. Para a implementação SSE, nós apresentamos dois tipos de código, conforme discutido na Seção 7.2.3; usando instruções de ponto flutuante ou inteiras para operações lógicas.

Tabela 7.4: Código *Assembly*, plataforma SSE (ciclos/*byte*), mensagem longa no Intel Core 2 Duo

Implementação	**Instruções ponto flutuante**			**Instruções inteiras**		
	256	384	512	256	384	512
PS	**9,40**	13,90	18,56	**9,71**	15,18	20,03
Paralela+PS	12,06	**13,75**	**17,78**	12,00	**13,96**	**17,78**

Até 20% de melhora no desempenho foi alcançada na versão Luffa-256 em comparação com nossa implementação PS. Além disso, podem ser observadas diferenças entre os códigos implementados usando instruções de ponto flutuante e inteiras.

Em 64 *bits*, a melhora de desempenho não foi muito significante, obtendo 8% na versão Luffa-512. Nesta plataforma, menos espaço em registradores está disponível, o que reduz o campo para otimizações. Além disso, melhores desempenhos são esperados dos compiladores na plataforma de 64 *bits*, devido à simplicidade de suas instruções, se comparado com a plataforma SSE.

Tabela 7.5: Código *Assembly*, 64 *bits* (ciclos/*byte*), mensagem longa no Intel Core 2 Duo

Implementação	256	384	512
Paralela+PS	19,06	21,37	32,25

Uma comparação do tamanho do código das implementações Referência e PS é apresentada na Tabela 7.6. Pode ser observada uma redução no tamanho na plataforma de 64 *bits*, devido às instruções de rotação, além de um pequeno aumento na plataforma SSE.

Tabela 7.6: Tamanho (KB) implementações de 64 *bits* e SSE em C, compiladas em ICC

	64-bit		**SSE**	
	Paralela	Paralela+PS	Básica	PS
Luffa-256	39,175	36,167	35,136	37,057
Luffa-384	35,428	36,071	35,133	37,043
Luffa-512	39,239	39,282	35,598	40,599

7.3 OpenMP

Nesta implementação, tentamos explorar os múltiplos núcleos presentes nos processadores *desktop* atuais. Por meio da linguagem *OpenMP*, processaremos os dados da variável de estado nos núcleos dos processadores da família Core 2 e Core i7, usando o método da memória compartilhada.

O fato do algoritmo Luffa possuir subpermutações independentes nos abre a oportunidade de processá-las nos distintos núcleos. Os processadores disponíveis para a experimentação incluem um Core 2 Duo T6400 2,00 GHz de dois núcleos e um Core i7-860 2,80 GHz de quatro núcleos. Com o objetivo de poder comparar melhor o desempenho entre os dois processadores, escolhemos a versão Luffa-256 que processa o menor número de subpermutações (3) entre os outros membros da família.

A linguagem OpenMP permite ao implementador delimitar blocos de código a serem calculados independentemente nos núcleos do processador. Estes blocos são chamados de *threads*.

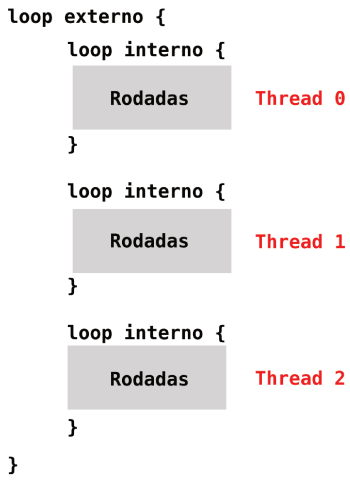


Figura 7.9: Estrutura do algoritmo Luffa
O tamanho do loop *externo depende do tamanho da mensagem, o* loop *interno, essencial para o paralelismo a nível de* threads *é fixo, e equivale a oito.*

Com o fraco desempenho apresentado na Tabela 7.7, podemos concluir que o algoritmo Luffa não oferece oportunidades em sua estrutura interna para explorar o paralelismo a nível de *threads*. Entre as razões podemos citar:

- A cada iteração, a fase da injeção de mensagens impede a execução das *threads* em núcleos distintos para criar a interdependência entre os blocos do estado, operação esta necessária para garantir a segurança do algoritmo.

- O número de rodadas é insuficiente para que o processamento paralelo nos núcleos cubra os custos gerados pela transferência de dados na etapa serial do projeto.

Tabela 7.7: Desempenho do algoritmo Luffa-256, mensagem de 32 MB, implementação OpenMP, compilada em ICC v.11.1

Versão	**Core 2 Duo**	**Core i7**	
	2 Threads	2 Threads	3 Threads
Luffa-256	52 ciclos/*byte*	52 ciclos/*byte*	71 ciclos/*byte*

7.4 AVR ATmega 8

Na plataforma ATmega de 8 *bits* realizamos duas implementações, uma Básica, baseada na lógica proposta pelos autores; e outra usando a técnica *perfect shuffle*. Ambos os códigos foram escritos na linguagem *Assembly* e executados através do simulador AVR Studio v.4.18, em uma máquina Intel Core 2 Duo T6400 2.00GHz.

7.4.1 Implementação Básica

O principal objetivo nesta implementação é manter a variável de cadeia inteiramente no estado durante as subpermutações. Uma vez que estas processam 256 *bits* do estado independentemente, é possível mantê-lo nos 32 registradores de 8 *bits* do processador.

Tabela 7.8: *Profiling* da Implementação Básica, mensagem de um bloco

Função	**Custo (ciclos)**
Inicialização	1011
Preenchimento mensagem	243
Rodada	15858
Message Injection	2309
SubCrumb	3792
MixWord	7488
AddConstant	1608
Transformação final	16382
Total	33500

RAM: 160 *bytes*. ROM: 2330 *bytes* (código), 296 *bytes* (dados).

A partir dos resultados da Tabela 7.8 podemos concluir os seguintes itens,

- A operação para mover os dados da memória para os registradores foi bastante reduzida, ocupando pouco mais de 1% do custo total, alcançando o objetivo primário da implementação.

- Assim como na arquitetura x86, devido às rotações em palavras de 32 *bits*, a função *MixWord* é a mais custosa. Com registradores de 8 *bits*, as rotações precisam ser feitas por meio de *carry*.

  lsl r3 ; desl. à esq.
  rol r2 ; rotação à esq. com *carry*
  rol r1 ; rotação à esq. com *carry*
  rol r0 ; rotação à esq. com *carry*
  adc r3, r0 ; adicionar *carry*

- A transformação final aplicada na mensagem traz um grande custo para plataformas pequenas que processam mensagens pequenas, duplicando o custo total.

7.4.2 Implementação *Perfect Shuffle*

Assim como na plataforma de x86, o objetivo do uso da técnica *perfect shuffle* no algoritmo Luffa é reduzir o custo das rotações na função *MixWord*.

Tabela 7.9: *Profiling* da Implementação *Perfect Shuffle*, mensagem de um bloco

Função	**Custo (ciclos)**
Inicialização	1110
Preenchimento mensagem	222
Perfect Shuffle	1946
Rodada	19091
Message Injection	3329
SubCrumb	4488
MixWord	4824
AddConstant	5736
Transformação final	19615
Total	41990

RAM: 160 *bytes*. ROM: 4454 *bytes* (código), 1384 *bytes* (dados).

- As operações relacionadas ao *perfect shuffle* são pouco custosas, responsáveis por aproximadamente 4,5% do total.

- Todas as rotações em *bytes* da função *MixWord* agora podem ser feitas através da renomeação de registradores, sem gasto de ciclos. Como consequência, podemos observar a redução do custo desta função em 35% em relação à implementação básica.

- A função *SubCrumb* foi implementada por meio de duas tabelas de 256 *bytes*, o que resultou em um aumento na memória ROM, além do aumento de ciclos em relação às instruções lógicas da implementação básica.

- A principal função afetada foi a adição de constantes, com um aumento de custo em mais de 250% em relação à implementação básica.

O grande motivo pela queda no desempenho da implementação *perfect shuffle* na plataforma AVR está na etapa da adição de constantes. O maior número de acessos à memória aumenta consideravelmente o custo total do algoritmo, mesmo com os ganhos obtidos na função *MixWord*. Este problema também ocorre na plataforma x86, porém, com o transporte maior de dados da memória por ciclo, a presença da *cache* e o *pipeline* superescalar o custo extra se torna irrelevante.

Não há publicação de implementações realizadas em *Assembly* do algoritmo Luffa na plataforma AVR. Em [152], uma implementação em C compilada com AVR-GCC processa um bloco de mensagem em 46141 ciclos.

Capítulo 8

Última rodada do concurso SHA-3

A fase final do concurso SHA-3 é composta por apenas cinco algoritmos, que serão analisados publicamente até o fim de 2011. De acordo com [144] o principal critério para a escolha dos finalistas foi a segurança. Alguns dos itens considerados positivos pelo NIST foi a quantidade significante de criptanálise e parâmetros ajustáveis de segurança bem definidos, no entanto, o instituto afirma que devido à complexidade em classificar ataques quanto a sua importância, análises quantitativas tiveram que ser mapeadas para dados qualitativos, que por sua vez, variam a cada algoritmo.

O NIST também afirma ter considerado critérios de custo e desempenho, eliminando algoritmos que exigiam uma área muito grande ao serem implementados em *hardware*. Além disso, o instituto espera que os algoritmos tenham uma eficiência equiparada à do SHA-2.

Por fim, a diversidade de construções e funções internas também foram consideradas, com o objetivo de que uma quebra em um dos finalistas não implique na quebra dos demais algoritmos.

Tabela 8.1: Algoritmos finalistas

Algoritmo	**Nacionalidade**	**Classe**	**Quebrado?**
BLAKE [7]	Suíça/Reino Unido	Dedicadas/HAIFA	Não
Grøstl [63]	Dinamarca/Áustria	Dedicada/MD	Não
JH [156]	Cingapura	Dedicada/Constr. Própria	Não
Keccak [21]	Holanda/Suíça	Dedicada/Esponja	Não
Skein [57]	EUA/Alemanha	Baseados em bloco de cifra/Threefish	Não

8.1 BLAKE

8.1.1 Descrição

O algoritmo BLAKE foi projetado pelos pesquisadores Jean-Philippe Aumasson, Luca Henzen, Willi Meier e Raphael Phan [7], e consiste em uma família de quatro algoritmos, BLAKE-224, BLAKE-256, BLAKE-384 e BLAKE-512. Seu modo de operação é projetado por meio da construção HAIFA, onde parâmetros extras são adicionados ao bloco de mensagem e à variável do estado. Sua função de compressão é baseada na cifra de fluxo *ChaCha* [17] de Daniel Bernstein.

Os algoritmos da família BLAKE são parecidos entre si, diferenciando-se apenas nos valores iniciais do estado, constantes, tamanho do bloco da mensagem e das palavras. Com o objetivo de simplificar a descrição de BLAKE, iremos focar apenas no algoritmo BLAKE-256, para mais detalhes ver [7].

BLAKE-256 trabalha com palavras de 32 *bits*, processando mensagens de até 2^{64} *bits* divididas em blocos de 512 *bits* para gerar resumos de 256 *bits*. O algoritmo é composto por duas etapas, o preenchimento da mensagem e o seu processamento.

Preenchimento da mensagem

Inicialmente é concatenada à mensagem m um *bit* "1", em seguida *bits* "0" são concatenados até que o tamanho da mesma seja congruente a $447 \mod 512$. Logo, outro *bit* "1" é concatenado e finalmente a representação binária de $|m|$ em um bloco de 64 *bits* é anexada. Este processo pode ser representado pela seguinte fórmula,

$$m' \leftarrow m \| 100 \ldots 001 |m|_{64}.$$

Processamento da mensagem

A mensagem é dividida em blocos de 512 *bits* e processada de forma iterativa por uma função de compressão dividida em três etapas: inicialização, rodadas e finalização.

Inicialização Nesta fase, a variável do estado representada por 8 palavras de 32 *bits* é expandida por meio do valor *salt*, do contador e 8 das 16 constantes de 32 *bits* pré-definidas.

Seja h_i uma palavra do estado resultante da iteração anterior, s_i uma palavra do valor *salt*, t_i uma palavra do contador e c_i uma palavra da constante pré-definida,

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} = \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix}$$

Rodadas Nesta etapa as palavras do estado expandido serão processadas juntamente com as palavras do bloco da mensagem em dez rodadas. A cada rodada são aplicadas oito vezes a função G.

Seja, v_i uma palavra do estado expandido,

$G_0(v_0, v_4, v_8, v_{12})$ $\quad$ $G_4(v_0, v_5, v_{10}, v_{15})$

$G_1(v_1, v_5, v_9, v_{13})$ $\quad$ $G_5(v_1, v_6, v_{11}, v_{12})$

$G_2(v_2, v_6, v_{10}, v_{14})$ $\quad$ $G_6(v_2, v_7, v_8, v_{13})$

$G_3(v_3, v_7, v_{11}, v_{15})$ $\quad$ $G_7(v_3, v_4, v_9, v_{14})$

A função $G_i(a, b, c, d)$ é definida por uma série de operações lógicas e aritméticas em suas palavras de entrada.

Seja m_i uma palavra do bloco de mensagem, c_i uma palavra das constantes pré-definidas e σ_r permutações pré-definidas e distintas a cada rodada r,

$a \leftarrow a + b + (m_{\sigma_r(2i)} \oplus c_{\sigma_r(2i+1)})$

$d \leftarrow (d \oplus a) \ggg 16$

$c \leftarrow c + d$

$b \leftarrow (b \oplus c) \ggg 12$

$a \leftarrow a + b + (m_{\sigma_r(2i+1)} \oplus c_{\sigma_r(2i)})$

$d \leftarrow (d \oplus a) \ggg 8$

$c \leftarrow c + d$

$b \leftarrow (b \oplus c) \ggg 7$

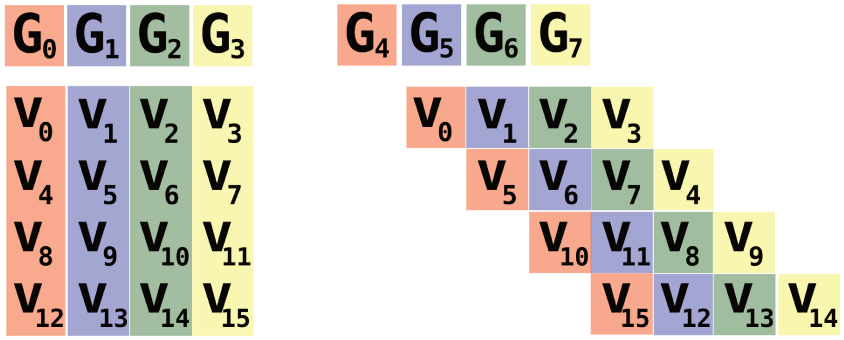


Figura 8.1: Passos verticais e diagonais da função G

Finalização O estado é comprimido por meio de operações com as variáveis de estado antigo, os valores *salt* e o estado atual.

Seja h_i uma palavra do estado resultante da iteração anterior, s_i uma palavra do valor *salt* e v_i uma palavra do estado atual expandido,

$h'_0 \leftarrow h_0 \oplus s_0 \oplus v_0 \oplus v_8$;
$h'_1 \leftarrow h_1 \oplus s_1 \oplus v_1 \oplus v_9$;
$h'_2 \leftarrow h_2 \oplus s_2 \oplus v_2 \oplus v_{10}$;
$h'_3 \leftarrow h_3 \oplus s_3 \oplus v_3 \oplus v_{11}$;
$h'_4 \leftarrow h_4 \oplus s_0 \oplus v_4 \oplus v_{12}$;
$h'_5 \leftarrow h_5 \oplus s_1 \oplus v_5 \oplus v_{13}$;
$h'_6 \leftarrow h_6 \oplus s_2 \oplus v_6 \oplus v_{14}$;
$h'_7 \leftarrow h_7 \oplus s_3 \oplus v_7 \oplus v_{15}$.

8.1.2 Comentários

O núcleo do algoritmo BLAKE é a função G, que através de sua aplicação nas colunas e nas diagonais da matriz do estado, realiza etapas não lineares e de difusão entre os elementos. Esta função é bastante simples, com quatro rotações, seis instruções de ou-exclusivo e seis adições. Como consequência, sua implementação não é complexa, mesmo em plataformas com recursos reduzidos. Além disso, os autores conseguiram adaptar o algoritmo para plataformas de 32 e 64 *bits* através de suas duas versões que utilizam palavras de diferentes tamanhos.

O tamanho do estado nas versões BLAKE-384 e BLAKE-512 também facilita a implementação na plataforma x86-64, pois é possível alocá-lo totalmente nos registradores disponíveis ($16 \times 64 = 1024$ *bits*). A aplicação de BLAKE na plataforma SSE é mais

vantajosa nas versões de palavras de 32 *bits* (BLAKE-224 e BLAKE-256), pois nestas é possível alocar quatro elementos da matriz em somente um registrador.

No entanto, duas características do projeto prejudicam sua implementação nas plataformas SSE: o fato das entradas de G serem feitas ora em colunas e ora em diagonais, pois obriga ao implementador mudar a posição dos elementos na matriz a cada aplicação de G; e as escolhas das palavras do bloco da mensagem e das constantes serem definidas pelas permutações aleatórias pré-definidas, o que faz com que a cada chamada de G um vetor SSE tenha que ser montado por meio da instrução _mm_set_epi32, que é um pouco custosa. Mesmo assim, a implementação com vetores de 128 *bits* supera todas as outras [20].

```
//Operações necessárias para processar os dados em diagonais.
//Note que a primeira linha da matriz náo precisa ser alterada.

H1 = _mm_shuffle_epi32(H1, 0x39);
H2 = _mm_shuffle_epi32(H2, 0x4E);
H3 = _mm_shuffle_epi32(H3, 0x93);

//Após o processamento, o shuffle precisa ser revertido para outra etapa em colunas.
```

Por fim, a finalização do algoritmo tem a vantagem de não exigir processamento extra para mensagens pequenas (≤ 400 *bytes*).

8.2 Grøstl

8.2.1 Descrição

O algoritmo Grøstl foi projetado por um grupo de pesquisadores [63] da Universidade Técnica da Dinamarca e do instituto austríaco *IAIK* (*Institute for Applied Information Processing and Communications*). O seu modo de operação é baseado na construção Merkle-Damgård e sua função de compressão pode ser vista como uma construção Davies-Meyer [13], usando a cifra de bloco AES como referência.

Quatro algoritmos fazem parte da família Grøstl: Grøstl_{224}, Grøstl_{256}, Grøstl_{384} e Grøstl_{512}. Os integrantes diferenciam-se no tamanho do bloco da mensagem e do estado, em algumas subfunções internas à função de compressão e nos valores das constantes. Como forma de simplificar a descrição do algoritmo, iremos detalhar apenas o algoritmo Grøstl_{256}. Para mais detalhes, ver [63].

O Grøstl_{256} trabalha com palavras de 64 *bits*, processando blocos de mensagem de 512 *bits* de forma iterativa. O algoritmo é dividido em três etapas básicas, preenchimento da mensagem, processamento e transformação final.

Preenchimento da mensagem

O método de completar a mensagem do algoritmo Grøstl é semelhante ao usado na técnica de Mekle-Damgård (ver Seção 2.3.1).

Seja a mensagem original m,

1. Concatenar um *bit* "1" ao final de m;
2. Completar com *bits* "0" até que o tamanho da mensagem seja congruente a 448 mod 512;
3. Concatenar um bloco de 64 *bits* representando a quantidade de blocos de m modificada após os passos 1 e 2.

Processamento da mensagem

A função de compressão f de Grøstl recebe como entrada uma variável de estado h com oito palavras de 64 *bits* e um bloco de mensagem m do mesmo tamanho. Em seguida, ambos são processados por permutações P e Q:

$$f(h, m) = P(h \oplus m) \oplus Q(m) \oplus h.$$

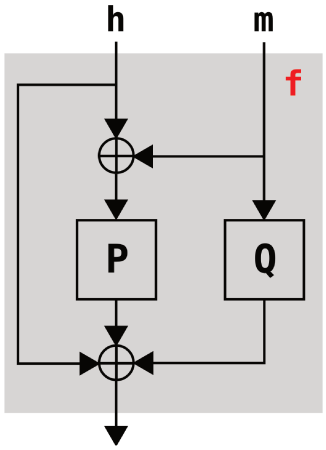


Figura 8.2: Função f do algoritmo Grøstl

As permutações P e Q Dez rodadas são realizadas por meio de subfunções semelhantes ao algoritmo AES [116]: *AddRoundConstant*, *SubBytes*, *ShiftBytes* e *MixBytes*. Exceto pelas subfunções *AddRoundConstant* e *ShiftBytes*, as permutações P e Q são idênticas entre si.

As entradas destas permutações são representadas por matrizes C 8×8 com elementos de 8 *bits*. A transformação de vetores de *bytes* para matriz é feita de maneira semelhante ao algoritmo Keccak (ver Seção 5.2), coluna a coluna.

AddRoundConstant Nesta etapa, uma constante é adicionada em uma posição da matriz de entrada C.

Seja o i o número da rodada em que a subfunção é executada, e C a matriz de entrada,

$AddRoundConstant_P$:
$C[x, y] \leftarrow$ `0x00`, Para todo $0 \leq x \leq 7$ e $0 \leq y \leq 7$;
$C[0, y] \leftarrow C[0, y] \oplus i \oplus (y \ll 4)$, Para todo $0 \leq y \leq 7$.

$AddRoundConstant_Q$:
$C[x, y] \leftarrow$ `0xFF`, Para todo $0 \leq x \leq 7$ e $0 \leq y \leq 7$;
$C[7, y] \leftarrow C[7, y] \oplus i \oplus (y \ll 4)$, Para todo $0 \leq y \leq 7$.

SubBytes Esta é a fase não linear da função de compressão. Cada *byte* da matriz C será substituído por outro *byte* por meio de uma tabela de substituição S, que é idêntica à tabela do algoritmo AES.

$C[i, j] \leftarrow S(C[i, j])$, Para todo $0 \leq i \leq 7$ e $0 \leq j \leq 7$.

ShiftBytes Nesta fase, para cada linha da matriz C, os elementos serão rotacionados à esquerda por um valor pré-definido. Os valores são definidos por dois vetores $\sigma_P = [0, 1, 2, 3, 4, 5, 6, 7]$ e $\sigma_Q = [1, 3, 5, 7, 0, 2, 4, 6]$.

Seja $V[i]$ um vetor contendo os elementos da linha i da matriz $C[i, j]$,

$AddShiftBytes_P$: $V[i] \leftarrow (V[i] \lll \sigma_P[i])$;

$AddShiftBytes_Q$: $V[i] \leftarrow (V[i] \lll \sigma_Q[i])$.

MixBytes Por fim, cada coluna da matriz C passa por uma transformação. Para isso, cada *byte* da matriz será considerado um elemento do corpo finito $\mathbb{F}_{256}$, com o polinômio irredutível representado por $x^8 + x^4 + x^3 + x + 1$, que é idêntico ao usado no algoritmo AES.

Em seguida, a matriz C é multiplicada por uma matriz constante B.

$$B = \begin{pmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{pmatrix}$$

Transformação final

A transformação final é simples, o estado resultante da etapa de processamento da mensagem passa pela permutação P:

$g(x) = (P(x) \oplus x)$.

Finalmente, o resultado é truncado para o valor de resumo.

8.2.2 Comentários

A função de compressão do algoritmo Grøstl é semelhante ao algoritmo AES, porém operando em um estado de quatro a oito vezes maior. As implementações mais plausíveis devem basear-se em tabelas, que ocupa no mínimo 2KB.

A implementação em forma de tabelas consiste em colocar cada coluna da matriz do estado em uma palavra, deste modo, o algoritmo acaba por forçar o implementador a utilizar palavras de 64 *bits*. Implementações em 32 *bits* também são possíveis, porém tem seu desempenho um pouco reduzido. Em ambientes mais restritos, as etapas não podem ser implementadas em uma só tabela, devido ao seu tamanho. Desta forma, as subfunções da função de compressão devem ser implementadas separadamente, com a possibilidade de substituir operações de multiplicações por tabelas.

```
//Exceto a adição de constantes, todas as operações do AES podem ser feitas
//por substituição em tabelas.
//A substituição é feita a cada coluna, m corresponde à matriz de bytes.

//Coluna 0
ch[0] = T0[m[0]] ^ T1[m[9]] ^ T2[m[18]] ^ T3[m[27]] ^ T4[m[36]]
      ^ T5[m[45]] ^ T6[m[54]] ^ T7[m[63]];
```

A transformação final exige uma aplicação extra das funções do AES, prejudicando o desempenho em mensagens pequenas.

Como conclusão, este algoritmo não possui grande flexibilidade com relação à implementação em distintas plataformas, devido às complexas operações do AES aplicadas a um estado muito grande. Consequentemente, o Grøstl está nas últimas posições em desempenho em *software* com relação aos algoritmos da fase final [20]. Sua velocidade somente aumenta com a utilização de instruções especiais AES-NI [16] que estão presentes apenas em processadores Intel mais recentes.

8.3 JH

8.3.1 Descrição

O algoritmo JH foi projetado pelo pesquisador Hongjun Wu de Cingapura [156], seu modo de operação é distinto dos outros algoritmos, e sua função de compressão é baseada no AES, porém aplicada em uma estrutura de diferente dimensão. O algoritmo é composto por quatro versões idênticas, diferenciando-se apenas no estado inicial.

O autor propõe duas maneiras de se implementar o algoritmo, uma forma básica, com as rodadas idênticas, próprio para ser implementado em *hardware* e uma forma *bitslice*, para ser implementada em *software*. Nesta seção descreveremos a implementação básica, comentando a implementação *bitslice* na Seção 8.3.2.

O algoritmo trabalha com palavras de quatro *bits* e processa blocos de mensagem de 512 *bits*, mantendo um estado de 1024 *bits*. A geração do resumo se divide em duas partes, o preenchimento da mensagem e o processamento da mensagem por meio de iterações de uma função de compressão.

Preenchimento da mensagem

O preenchimento da mensagem consiste em três operações,

1. Concatenar um *bit* "1" ao final da mensagem;
2. Completar com *bits* "0" até que o tamanho da mensagem seja congruente a 384 mod 512;
3. Concatenar um bloco de 128 *bits* correspondendo ao tamanho da mensagem em *bits*.

Processamento da mensagem

O estado do JH corresponderá a uma estrutura semelhante à do AES, porém através de uma matriz de oito dimensões com elementos de quatro *bits*, totalizando 1024 *bits* ($2^8 \times 4$).

A função de compressão F_8 Esta função é formada por duas adições dos blocos de mensagem à variável do estado, intercaladas por uma função bijetora E_8.

Seja $H^{(i),j}$ o j-ésimo *bit* do estado da iteração i, $M^{(i),j}$ o j-ésimo *bit* do bloco de mensagem i, B^j e A^j os j-ésimos *bits* das palavras auxiliares A e B,

$$
\begin{aligned}
&A^j = H^{(i-1),j} \oplus M^{(i),j}, && \text{para } 0 \leq j \leq 511;\\
&A^j = H^{(i-1),j}, && \text{para } 512 \leq j \leq 1023;\\
&B = E_8(A);\\
&H^{(i),j} = B^j, && \text{para } 0 \leq j \leq 511;\\
&H^{(i),j} = B^j \oplus M^{(i),j-512}, && \text{para } 512 \leq j \leq 1023.
\end{aligned}
$$

A função bijetora E_8 É a principal função do algoritmo e é dividida em três partes: substituição, transformação linear e permutação. Estas três etapas são executadas por 42 rodadas.

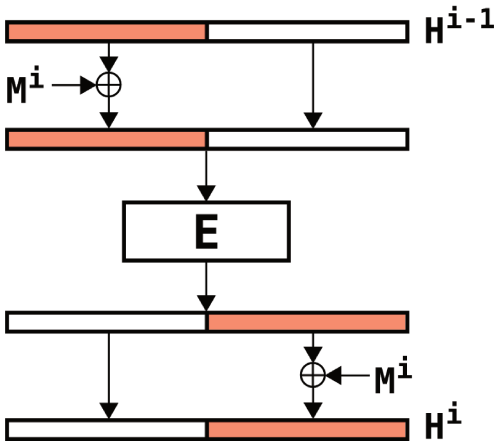


Figura 8.3: Função F do algoritmo JH [156]

Substituição Cada elemento de quatro *bits* da matriz do estado passa por uma substituição por meio de duas tabelas 4×4. A tabela a ser utilizada é determinada pelas constantes pré-definidas, que são distintas a cada rodada.

Tabela 8.2: Tabelas de substituição da função bijetora B_8

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$S_0(x)$	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
$S_1(x)$	3	12	6	13	5	7	1	9	15	2	0	4	11	10	14	8

Cada *bit* das constantes definirá qual tabela será usada em cada elemento da matriz, desta forma, a cada rodada, cada constante terá 256 *bits* (1024/4).

Transformação Linear Nesta etapa, algumas operações lógicas são realizadas nos *bits* de duas palavras consecutivas.

Sejam $A = A^0\|A^1\|A^2\|A^3$, $B = B^0\|B^1\|B^2\|B^3$, $C = C^0\|C^1\|C^2\|C^3$ e $D = D^0\|D^1\|D^2\|D^3$ palavras com 4 *bits*, a transformação $(C, D) = L(A, B)$ é feita da seguinte forma,

$D^0 = B^0 \oplus A^1$;
$D^1 = B^1 \oplus A^2$;
$D^2 = B^2 \oplus A^3 \oplus A^0$;
$D^3 = B^3 \oplus A^0$;
$C^0 = A^0 \oplus D^1$;
$C^1 = A^1 \oplus D^2$;
$C^2 = A^2 \oplus D^3 \oplus D^0$;
$C^3 = A^3 \oplus D^0$.

Permutação Por fim, uma permutação P_8 é aplicada nos elementos das matrizes. Ela é constituída por três subpermutações π_8, P'_8 e ϕ_8. Para todas as descrições das permutações, a_i e b_i serão considerados elementos de quatro *bits* da matriz do estado.

A permutação π_8 é definida da seguinte forma,
$b_{4i+0} = a_{4i+0}$, para $0 \leq i \leq 2^6 - 1$; $b_{4i+1} = a_{4i+1}$, para $0 \leq i \leq 2^6 - 1$;
$b_{4i+2} = a_{4i+3}$, para $0 \leq i \leq 2^6 - 1$; $b_{4i+3} = a_{4i+2}$, para $0 \leq i \leq 2^6 - 1$.

A permutação P'_8 é definida da seguinte forma,
$b_i = a_{2i}$, para $0 \leq i \leq 2^7 - 1$;
$b_{i+2^{d-1}} = a_{2i+1}$, para $0 \leq i \leq 2^6 - 1$.

A permutação ϕ_8 é definida da seguinte forma,
$b_i = a_i$, para $0 \leq i \leq 2^7 - 1$;
$b_{2i+0} = a_{2i+1}$, para $2^6 \leq i \leq 2^7 - 1$;
$b_{2i+1} = a_{2i+0}$, para $2^6 \leq i \leq 2^7 - 1$.

Finalmente, os *bits* do estado final são truncados para gerar o valor do resumo.

8.3.2 Comentários

A descrição apresentada na Seção 8.3.1 é própria para *hardware*, pois possui instruções manipulando *bits* e preza por manter as rodadas idênticas, o que é essencial para um implementação compacta. Os autores também sugerem uma implementação *bitslice* para *software*, cujas principais características são a utilização de palavras de 128 *bits* e a presença de distintas permutações.

Na implementação *bitslice*, a fase da substituição e da transformação linear são realizadas por meio de instruções lógicas simples. A permutação por sua vez se divide em sete subpermutações, que se intercalarão entre as rodadas. A cada subpermutação, elementos de tamanhos fixos serão trocados dentro de uma palavra de 128 *bits*, conforme a Figura 8.4.

```
//Permutações acima de 8 bits podem ser feitas com apenas uma instrução shuffle.

//Permutação 32 bits
x = _mm_shuffle_epi32(x, 0xB1);

//Permutação 64 bits
x = _mm_shuffle_epi32(x, 0x4E);
```

Um problema do algoritmo JH com relação à arquiteturas x86 é sua dependência do conjunto SSE para uma implementação eficiente, isso ocorre devido ao tamanho de seu estado, que exige movimentações para a memória, além de quase quatro vezes mais operações lógicas, se implementado em registradores de 32 *bits*. Além disso, as subpermutações são responsáveis por um custo relevante de processamento, e com instruções SSE, mais da metade delas podem ser feitas com apenas um ciclo.

Usar a tecnologia SSE para tentar implementar a versão para *hardware* do algoritmo poderia ser uma alternativa, porém, um grande obstáculo é a implementação de maneira eficiente da dupla tabela de substituição de acordo com os *bits* das constantes.

Para ambientes restritos, o principal entrave é o tamanho do estado (1024 *bits*) e o bloco da mensagem que deve ser armazenada para posterior adição (512 *bits*). É preciso considerar também o tamanho das constantes, que ocupam pouco mais de 1KB. Consequentemente, o principal custo nestas plataformas será a movimentação de dados entre

registradores e memória.

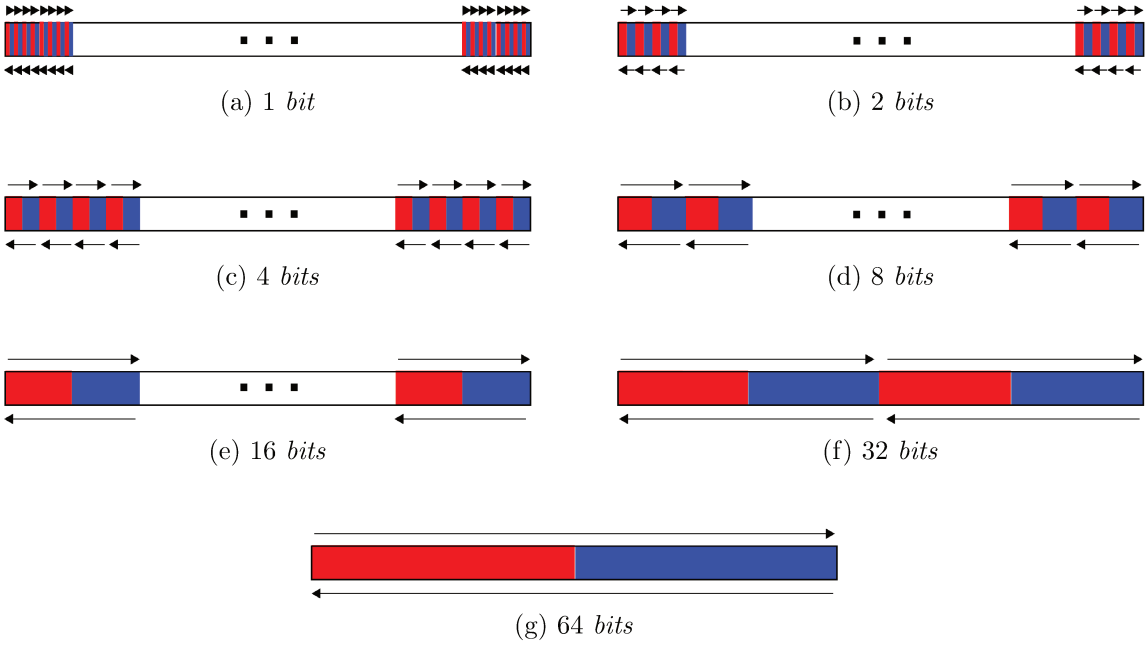


Figura 8.4: Subpermutações de JH movimentando elementos de distintos tamanhos em um vetor de 128 *bits*

8.4 Keccak

8.4.1 Descrição

Uma descrição detalhada do algoritmo Keccak pode ser encontrada na Seção 5.2.

8.4.2 Comentários

Apesar de trabalhar com palavras de 64 *bits*, através de operações de *bit shuffle* é possível implementar o algoritmo Keccak em distintas plataformas, pagando o preço de realizar operações extras para entrepor os *bits* das palavras. A presença de instruções lógicas simples e a ausência de tabelas de substituição contribuem para sua simplicidade. Além disso, as constantes do algoritmo podem ser armazenadas em aproximadamente 960 *bits*, um valor adequado para plataformas mais restritas.

O uso de instruções SSE ficam comprometidas com as distintas rotações realizadas na etapa ρ. Uma forma de utilizar estas instruções é através do processamento paralelo de

mensagens, conforme mostrado no Capítulo 6.

As etapas do algoritmo podem também ser unidas para obter alguma otimização, porém isto depende de cada processador.

```
//Primeiras etapas das funções Theta, Rho e Pi implementadas de forma conjunta
//para maior desempenho.

m[0] = m[0] ^ tmp[1];
m[5] = m[5] ^ tmp[1];
aux = (m[5] << 36) | (m[5] >> 28);
m[3] = m[3] ^ tmp[4];
m[5] = (m[3] << 28) | (m[3] >> 36);
m[18] = m[18] ^ tmp[4];
m[3] = (m[18] << 21) | (m[18] >> 43);
m[17] = m[17] ^ tmp[3];
m[18] = (m[17] << 15) | (m[17] >> 49);
m[11] = m[11] ^ tmp[2];
m[17] = (m[11] << 10) | (m[11] >> 54);
```

8.5 Skein

8.5.1 Descrição

O algoritmo Skein foi desenhado por um grupo de pesquisadores da academia e da indústria [57], sua função de compressão foi projetada a partir do algoritmo *Threefish*, desenhado especialmente para este projeto de resumo criptográfico.

Os autores propõem três algoritmos para compor a família Skein: Skein-256, Skein-512 e Skein-1024, que possuem distintos tamanhos de estado, respectivamente, 256, 512 e 1024 *bits*. O primeiro membro é destinado a ambientes com recursos restritos, enquanto o último possui parâmetros que garantem necessidades acima das exigidas pelo NIST. O Skein-512 é a proposta primaria dos autores, e a partir desta iremos descrever o algoritmo nesta seção. Para mais detalhes ver [57].

Resumidamente, o Skein opera de forma iterativa, trabalhando com palavras de 64 *bits* e processando blocos de mensagem de 512 *bits* através da cifra Threefish implantada em um modo de operação denominado UBI (*Unique Block Iteration*), uma variante do método Matyas-Meyer-Oseas (ver Seção 2.5.1). O UBI funciona com uma variável denominada *tweak*, que juntamente com o estado e o bloco da mensagem será entrada para a cifra Threefish. Desta forma, podemos organizar o algoritmo através de três etapas: inicialização e configuração inicial da variável *tweak*, processamento da mensagem e geração

do valor de resumo.

Inicialização e configuração da variável *tweak*

A variável *tweak* é composta por duas palavras de 64 *bits* e armazena informações relativas à quantidade de *bytes* processados em determinada iteração ou ao nível da árvore de resumo se o modo de operação for paralelo. Além disso, esta variável também carrega dois campos sinalizando o último e o primeiro bloco. Por fim, o *tweak* também possui um campo denominado "tipo", que permite diferenciar fases do algoritmo além de identificar a aplicação de maior nível na qual o Skein está sendo utilizado. Desta forma, nesta etapa do algoritmo, os valores iniciais como o "tipo" e o sinal de bloco inicial devem ser inseridos.

As variáveis iniciais de estado, que são pré-determinadas de acordo com o tamanho do resumo final, também devem ser configuradas.

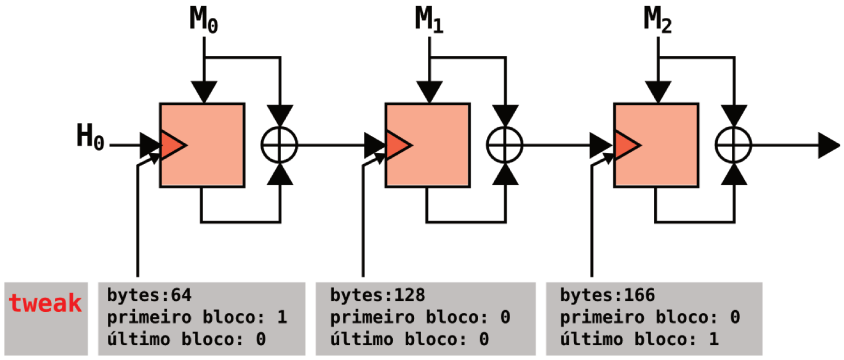


Figura 8.5: Modo de operação UBI

Processamento da mensagem

O processamento dos blocos de mensagem se dá por meio da cifra Threefish, que recebe como entrada a variável de estado resultante da iteração anterior de 512 *bits* como chave, o bloco de mensagem de 512 *bits* e o tweak de 128 *bits*.

Primeiramente, as entradas são divididas em palavras de 64 *bits* e é iniciado o processo de geração de subchaves.

Geração de subchaves O processo começa adicionando palavras adicionais k_8 e t_2 à chave k e ao *tweak* t.

$$k_8 \leftarrow \lfloor 2^{64}/3 \rfloor \oplus \bigoplus_{i=0}^{7} k_i;$$
$$t_2 \leftarrow t_0 \oplus t_1.$$

Assim, cada subchave é definida de acordo com as 8 palavras do estado e a rodada.

Seja s o número da rodada, i uma palavra do estado, k_i as palavras da chave e t_i as palavras do *tweak*,

$$\begin{aligned}
&k_{s,i} \leftarrow k_{(s+i) \mod (N_w+1)}, && \text{para } 0 \leq i \leq 4;\\
&k_{s,i} \leftarrow k_{(s+i) \mod (N_w+1)} + t_{s \mod 3}, && \text{para } i = 5;\\
&k_{s,i} \leftarrow k_{(s+i) \mod (N_w+1)} + t_{(s+1) \mod 3}, && \text{para } i = 6;\\
&k_{s,i} \leftarrow k_{(s+i) \mod (N_w+1)} + s, && \text{para } i = 7.
\end{aligned}$$

Rodadas Um total de 72 rodadas são efetuadas no bloco de mensagem. Cada rodada corresponde a três etapas, adição de subchaves, função MIX e permutações. Para melhor compreensão, $v_{d,i}$ será considerada a palavra i do estado após d rodadas.

As adições das subchaves são feitas a cada rodada que seja múltipla de quatro.

$$e_{d,i} = \begin{cases} (v_{d,i} + k_{d/4,i}) \mod 2^{64}, & \text{se } d \mod 4 = 0,\\ v_{d,i}, & \text{caso contrário.} \end{cases}$$

Em seguida, a função MIX é aplicada em toda rodada a cada duas palavras do estado.

$$(f_{d,2j}, f_{d,2j+1}) = MIX_{d,j}(e_{d,2j}, e_{d,2j+1}), \qquad \text{para } 0 \leq j \leq 3.$$

A função $MIX_{d,j}$ recebe como entrada duas palavras (x_0, x_1) e retorna outras duas palavras (y_0, y_1).

$$\begin{aligned}
&y_0 \leftarrow (x_0 + x_1) \mod 2^{64};\\
&y_1 \leftarrow (x_1 \lll R_{(d \mod 8),j}) \oplus y_0.
\end{aligned}$$

As constantes de rotação $R_{d,j}$ são pré-definidas, variando a cada par de palavras e a cada rodada.

Finalmente, permutações são realizadas entre as palavras por meio de uma tabela π definida a cada versão do algoritmo.

$$v_{d+1,i} \leftarrow f_{d,\pi(i)}, \qquad \text{para } 0 \leq j \leq 7.$$

Tabela 8.3: Tabela π do algoritmo Skein-512

x	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$\pi(x)$	2	1	4	7	6	5	0	3

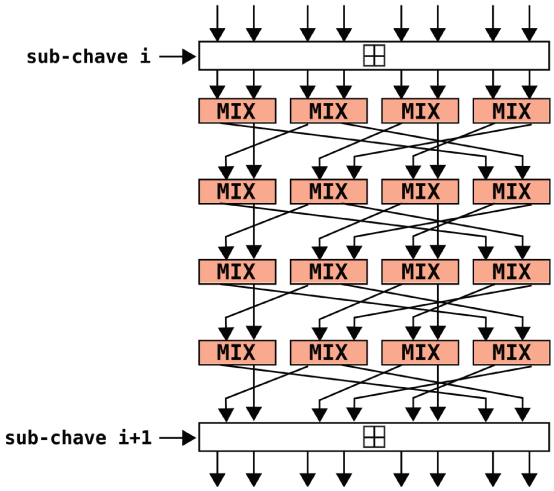


Figura 8.6: Quatro rodadas da cifra Threefish$_{512}$ [57]

Geração do resumo

A geração do resumo se dá inicialmente com uma reconfiguração do *tweak*, para sinalizar o início de uma nova etapa. Logo, a variável de estado gerada da última iteração é utilizada como entrada, juntamente com o *tweak* e um bloco de mensagem com valores 0.

Por fim, o resultado é truncado de acordo com o tamanho do resumo.

8.5.2 Comentários

O processamento do algoritmo Skein se resume basicamente na função MIX e na geração de subchaves, que pode ser feita ao longo das rodadas. O algoritmo se baseia em palavras de 64 *bits*, e seu desempenho se reduz consideravelmente se implementado em plataformas de 32 *bits*. Isto ocorre principalmente pelas rotações que variam entre as palavras do estado e as rodadas.

Estas rotações também impedem uma implementação eficiente na plataforma SSE, pois rotacionar duas palavras por valores distintos dentro de um vetor é bastante custoso.

No entanto, os autores se preocuparam em realizar variantes menores do algoritmo para serem implementadas em ambientes com recursos restritos. Por exemplo, o Skein-256 é bastante eficiente se implementado em um microcontrolador AVR de 8 *bits*, pois o estado pode ser alocado em todos os registradores ($32 \times 8 = 256$ *bits*). Além disso, com a chave sendo gerada ao longo das rodadas evita um armazenamento grande de dados.

//A permutação pode ser aplicada juntamente com a função MIX,
//apenas invertendo a ordem das entradas.

//Rodada 0

```
MIX_permutacao(text[0], text[1], 14);
MIX_permutacao(text[2], text[3], 16);

//Rodada 1
MIX_permutacao(text[0], text[3], 52);
MIX_permutacao(text[2], text[1], 57);
```

Concluindo, para uma implementação eficiente em arquiteturas menores que 64 *bits*, é necessária uma boa manipulação de instruções com *carry*, que compõem boa parte da função MIX. Esta característica parece ser a responsável pelo bom desempenho do algoritmo, evitando a implantação de funções não lineares por meio de tabelas.

8.6 Desempenho

Abaixo detalhamos o desempenho em *software* dos algoritmos nas plataformas de 64, 32 e 8 *bits* em relação aos outros quatro finalistas. Os resultados foram obtidos da página eBASH [20], acessada em janeiro de 2011.

Tabela 8.4: Plataforma de 64 *bits*, mensagem longa, Intel Core 2 Duo E6400, 2,137 GHz

Algoritmo	**Ciclos/*byte***
Skein	6,42
BLAKE	7,04
Keccak	10,90
JH	17,42
Grøstl	21,37

Tabela 8.5: Plataforma de 32 *bits*, mensagem longa, Intel Core 2 Duo E6400, 2,137 GHz

Algoritmo	**Ciclos/*byte***
BLAKE	10,51
Skein	17,56
JH	19,32
Keccak	19,51
Grøstl	23,67

Tabela 8.6: Plataforma de 8 *bits*, mensagem longa, Atmel ATmega1281, 16 GMHz

Algoritmo	**Ciclos/*byte***
Keccak	357,10
BLAKE	892,15
Skein	1203,90
Grøstl	1309,43
JH	2652,46

Capítulo 9

Conclusões

O foco deste trabalho é o estudo referente ao aspecto implementacional em *software* de funções de resumo em diferentes arquiteturas, assim como a análise de suas construções e tendências. Para isso, foram realizadas pesquisas em algoritmos padrões e tradicionais de função de resumo com o objetivo de entender melhor como os projetos estão relacionados com as circunstâncias de segurança (ataques existentes e poder computacional) e os recursos computacionais (arquiteturas com aplicações práticas) da época. Além disso, uma investigação de algumas arquiteturas foi necessário, assim como de seus futuros desenvolvimentos.

Para compreender o estado da arte dos algoritmos de funções de resumo, estudamos os projetos do concurso SHA-3, buscando entender como os autores desenvolvem os três principais requisitos propostos pelo NIST: segurança, custo e flexibilidade, com ênfase no segundo item, devido à natureza do trabalho. Com o propósito de reduzir os objetos de estudo e assim realizar uma investigação mais completa, focamos nas funções esponja, uma construção recente que recebeu muitos adeptos no concurso SHA-3.

Nossas principais contribuições foram uma organização das técnicas utilizadas pelos autores dos projetos inscritos no concurso SHA-3 para alcançar maior eficiência em *software*, assim como para economizar recursos, garantindo maior flexibilidade a seus algoritmos. Além disso, desenvolvemos o paralelismo de dados no algoritmo Luffa, por meio das instruções SSE, da técnica *perfect shuffle* e de otimizações em *Assembly*, o que gerou a implementação mais eficiente deste projeto em vários processadores [20], resultando em um artigo [119] apresentado no 10° Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais em 2010.

Realizamos também uma implementação eficiente para o algoritmo Luffa na plataforma ATmega de 8 *bits*, buscando otimizar os custos relativos ao acesso à memória RAM. Por fim, desenvolvemos um método de utilizar as instruções SSE em outro algoritmo baseado na construção esponja, o Keccak. O método permite a computação simultânea de

distintas mensagens.

9.1 Trabalhos futuros

Como trabalhos futuros, propomos a realização de implementações eficientes dos algoritmos presentes na última rodada do concurso SHA-3 voltadas para arquiteturas de 8 e 16 *bits*, uma vez que alguns projetos possuem características que implicam em desafios relacionados aos recursos restritos destas plataformas. Para as arquiteturas de 32 e 64 *bits* é necessário verificar o impacto do novo conjunto de instruções AVX [47] que aumenta a capacidade de realizar paralelismo de dados por meio dos vetores de 256 *bits*.

Além disso, é importante identificar e analisar técnicas utilizadas pelos autores de funções de resumo que procuram garantir propriedades de segurança, relacionando-as com o impacto do custo computacional e de memória em diversas arquiteturas.

Outra área a ser explorada são as GPUs, que podem vir a ser uma tendência na área criptográfica. Desta forma, é preciso pesquisar formas de implementar os algoritmos de funções de resumo de maneira a aproveitar o paralelismo fornecido por estas arquiteturas, seja por meio do processamento de dados independentes, pela técnica da Árvore Merkle ou até mesmo com propostas de novos projetos voltados especificamente para as GPUs.

Por fim, uma maior investigação nas formas de implementar os algoritmos atuais ou a proposição novos projetos para ambientes de recursos extremamente escassos, como sensores RFID ou redes de sensores, que garantam níveis de segurança exigidos pelas futuras aplicações destes dispositivos.

Referências Bibliográficas

[1] Ferhat Karakoc Adem Atalay, Orhun Kara and Cevat Manap. SHAMATA hash function algorithm specifications. Submission to NIST, 2008. http://www.uekae.tubitak.gov.tr/uekae_content_files/crypto/SHAMATASpecification.pdf.

[2] Inc. Advanced Micro Devices. AMD Athlon XP Processor Model 8. Data Sheet, 2003. http://www.amd.com.

[3] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard, June 1998. www.cs.technion.ac.il/~biham/Reports/Serpent/serpent-1.ps.gz.

[4] Yuriy Arbitman, Gil Dogon, Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFTX: A Proposal for the SHA-3 Standard. Submission to NIST, 2008. http://www.eecs.harvard.edu/~alon/PAPERS/lattices/swifftx.pdf.

[5] Gilles Van Assche. A rotational distinguisher on Shabal's keyed permutation and its impact on the security proofs. Available online, 2010. http://gva.noekeon.org/papers/ShabalRotation.pdf.

[6] Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, and Nicolas Sendrier. SHA-3 proposal: FSB. Submission to NIST, 2008. http://www-rocq.inria.fr/secret/CBCrypto/fsbdoc.pdf.

[7] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. SHA-3 proposal BLAKE. Submission to NIST, 2008. http://131002.net/blake/blake.pdf.

[8] Jean-Philippe Aumasson and Willi Meier. Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi. NIST mailing list, 2009. http://www.131002.net/data/papers/AM09.pdf.

[9] Multiple Authors. Comments submitted to the NIST hash function mailing list. hash-forum@nist.gov.

[10] Brian Baldwin, Andrew Byrne, Liang Lu, Mark Hamilton, Neil Hanley, Maire O'Neill, and William P. Marnane. FPGA Implementations of the Round Two SHA-3 Candidates. Second SHA-3 Candidate Conference, August 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/BALDWIN_FPGA_SHA3.pdf.

[11] Brian Baldwin and William P. Marnane. An FPGA Technologies Area Examination of the SHA-3 Hash Candidate Implementations. Cryptology ePrint Archive, Report 2009/603, 2009. http://eprint.iacr.org/.

[12] P.S.L.M. Baretto and V. Rijmen. The Whirlpool Hashing Function, 2000, revised in May 2003. http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html.

[13] Paulo S. L. M. Barreto. An observation on Grøstl. Available online, 2008. http://www.larc.usp.br/~pbarreto/Grizzly.pdf.

[14] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, CCS '93, pages 62–73. ACM, 1993.

[15] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 Proposal: ECHO. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/9/91/Echo.pdf.

[16] R. Benadjila, S. Gueron, and M. Robshaw. The Intel AES Instructions Set and the SHA-3 Candidates. In *ASIACRYPT 2009*, 2009.

[17] D. J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008: The State of the Art of Stream Ciphers, January 2008. http://cr.yp.to/papers.html#chacha.

[18] Daniel J. Bernstein. CubeHash specification (2.B.1). Submission to NIST, 2009. http://cubehash.cr.yp.to/submission2/spec.pdf.

[19] Daniel J. Bernstein. Second preimages for 6 (7? (8??)) rounds of Keccak? NIST mailing list, 2010. http://ehash.iaik.tugraz.at/uploads/6/65/NIST-mailing-list_Bernstein-Daemen.txt.

[20] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to. Accessed 27 January 2011.

[21] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications. Submission to NIST, 2010. `http://keccak.noekeon.org/Keccak-specifications.pdf`.

[22] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document version 2.1, 2010. `http://keccak.noekeon.org/Keccak-main-2.1.pdf`.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. ECRYPT Hash Workshop 2007, 2007. Also available as public comment to NIST from `http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html`.

[24] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology*, volume 4965 of *LNCS*, pages 181–197. Springer-Verlag, 2008.

[25] Eli Biham. A Fast New DES Implementation in Software. In *Proceedings of the 4th International Workshop on Fast Software Encryption*, volume 1267 of *LNCS*, pages 260–272. Springer-Verlag, 1997.

[26] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In *EUROCRYPT '05*, volume 3494 of *LNCS*, pages 36–57. Springer-Verlag, 2005.

[27] Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In *EUROCRYPT*, volume 3494 of *LNCS*, pages 36–57. Springer, 2005.

[28] Eli Biham and Orr Dunkelman. A Framework for Iterative Hash Functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. `http://eprint.iacr.org/`.

[29] Eli Biham and Orr Dunkelman. The SHAvite-3 Hash Function. Submission to NIST, 2009. `http://www.cs.technion.ac.il/~orrd/SHAvite-3/Spec.15.09.09.pdf`.

[30] Antoon Bosselaers, Renè Govaerts, and Joos Vandewalle. SHA: a design for parallel architectures? In *Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, volume 1233 of *LNCS*, pages 348–362. Springer-Verlag, 1997.

[31] Christina Boura and Anne Canteaut. Zero-Sum Distinguishers for Iterated Permutations and Application to Keccak-f and Hamsi-256. In *Selected Areas in Cryptography*, volume 6544 of *LNCS*, pages 1–17. Springer Berlin / Heidelberg, 2011.

[32] Colin Bradbury. BLENDER: A Proposed New Family of Cryptographic Hash Algorithms. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/5/5e/Blender.pdf`.

[33] Emmanuel Bresson, Anne Canteaut, Benoît Chevallier-Mames, Christophe Clavier, Thomas Fuhr, Aline Gouget, Thomas Icart, Jean-François Misarsky, Marìa Naya-Plasencia, Pascal Paillier, Thomas Pornin, Jean-René Reinhard, Céline Thuillet, and Marion Videau. Shabal, a Submission to NIST's Cryptographic Hash Algorithm Competition. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/6/6c/Shabal.pdf`.

[34] Karl Brincat and Chris J. Mitchell. New CBC-MAC Forgery Attacks. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy*, volume 2119 of *LNCS*, pages 3–14. Springer-Verlag, 2001.

[35] Daniel R. L. Brown, Adrian Antipa, Matt Campagna, and Rene Struik. ECOH: the Elliptic Curve Only Hash. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/a/a5/Ecoh.pdf`.

[36] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51:557–594, July 2004.

[37] Christophe De Cannière, Florian Mendel, and Christian Rechberger. Collisions for 70-Step SHA-1: On the Full Cost of Collision Search. In *Selected Areas in Cryptography*, volume 4876 of *LNCS*, pages 56–73. Springer, 2007.

[38] Christophe De Cannière and Christian Rechberger. Preimages for Reduced SHA-0 and SHA-1. In *CRYPTO*, volume 5157 of *LNCS*, pages 179–202. Springer, 2008.

[39] Christophe De Cannière, Hisayoshi Sato, and Dai Watanabe. Hash Function Luffa: Specification 2.0.1. Submission to NIST, 2009. `http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_Specification_20091002.pdf`.

[40] Christophe De Cannière, Hisayoshi Sato, and Dai Watanabe. Hash Function Luffa: Supporting Document. Submission to NIST, 2009. `http://www.sdl.hitachi.co.jp/crypto/luffa/Luffa_v2_SupportingDocument_20090915.pdf`.

[41] Donghoon Chang, Seokhie Hong, Changheon Kang, Jinkeon Kang, Jongsung Kim, Changhoon Lee, Jesang Lee, Jongtae Lee, Sangjin Lee, Yuseop Lee, Jongin Lim, and Jaechul Sung. Arirang. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/2/2c/Arirang.pdf`.

[42] Yongje Choi, Mooseop Kim, Taesung Kim, and Howon Kim. Low power implementation of SHA-1 algorithm for RFID system. In *Consumer Electronics, 2006. ISCE '06. 2006 IEEE Tenth International Symposium on*, 2006.

[43] Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an Efficient and Provable Collision Resistant Hash Function. Cryptology ePrint Archive, Report 2005/193, 2005. `http://eprint.iacr.org/`.

[44] Atmel Corporation. ATmega8/ATmega8L. Data Sheet, 2009. `http://www.atmel.com`.

[45] Intel Corporation. Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference, 2002. `http://www.intel.com`.

[46] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, 2009. `http://www.intel.com`.

[47] Intel Corporation. Intel Advanced Vector Extensions Programming Reference, 2010. `http://www.intel.com`.

[48] Jérémie Detrey, Pierrick Gaudry, and Karim Khalfallah. A Low-Area yet Performant FPGA Implementation of Shabal. Cryptology ePrint Archive, Report 2010/292, 2010. `http://eprint.iacr.org/`.

[49] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. `http://www.ietf.org/rfc/rfc5246.txt`.

[50] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22:664–654, November 1976.

[51] Hans Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11:253–271, 1998.

[52] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Proceedings of the Third International Workshop on Fast Software Encryption*, volume 1039 of *LNCS*, pages 71–82. Springer-Verlag, 1996.

[53] Tribunal Superior Eleitoral. Urna Eletrônica – Segurança, 2010. http://www.tse.gov.br.

[54] Christian Forler Ewan Fleischmann and Michael Gorski. The Twister Hash Function Family. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/3/39/Twister.pdf.

[55] Björn Fay. MeshHash. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/5/5a/Specification_DIN-A4.pdf.

[56] Martin Feldhofer and Christian Rechberger. A Case Against Currently Used Hash Functions in RFID Protocols. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4277 of *LNCS*, pages 372–381. Springer Berlin / Heidelberg, 2006.

[57] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. Submission to NIST, 2009. http://www.skein-hash.info/sites/default/files/skein1.2.pdf.

[58] FIPS 180-3. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, October 2008.

[59] FIPS 186-3. *Digital Signature Standard (DSS)*. U.S. Department of Commerce/-NIST, National Technical Information Service, Springfield, VA, June 2009.

[60] Agner Fog. Software optimization resources. http://www.agner.org/optimize/.

[61] Julien Francq and Céline Thuillet. Unfolding Method for Shabal on Virtex-5 FPGAs: Concrete Results. Cryptology ePrint Archive, Report 2010/406, 2010. http://eprint.iacr.org/.

[62] Kris Gaj, Ekawat Homsirikamol, and Marcin Rogawski. Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *LNCS*, pages 264–278. Springer Berlin / Heidelberg, 2010.

[63] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen. Grøstl – a SHA-3 candidate. Submission to NIST, 2008. http://www.groestl.info/Groestl.pdf.

[64] Danilo Gligoroski, Vlastimil Klima, Svein Johan Knapskog, Mohamed El-Hadedy, Jorn Amundsen, and Stig Frode Mjolsnes. Cryptographic Hash Function BLUE MIDNIGHT WISH. Submission to NIST, 2009. http://people.item.ntnu.no/~danilog/Hash/BMW-SecondRound/Supporting_Documentation/BlueMidnightWishDocumentation.pdf.

[65] Danilo Gligoroski, Rune Steinsmo Ødegård, Marija Mihova, Svein Johan Knapskog, Ljupco Kocarev, Aleš Drápal, and Vlastimil Klima. Cryptographic Hash Function EDON-R. Submission to NIST, 2008. http://people.item.ntnu.no/~danilog/Hash/Edon-R/Supporting_Documentation/EdonRDocumentation.pdf.

[66] Louis Goubin, Mickael Ivascot, William Jalby, Olivier Ly, Valerie Nachef, Jacques Patarin, Joana Treger, and Emmanuel Volte. CRUNCH. Submission to NIST, 2008. http://www.voltee.com/crunch/cdrom/Supporting_Documentation/crunch_specifications.pdf.

[67] X. Guo, S. Huang, L. Nazhandali, and P. Schaumont. Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations. Second SHA-3 Candidate Conference, 2010. http://csrc.nist.gov/groups/ST/hash/sha-3/.

[68] Michael A. Halcrow and Niels Ferguson. A Second Pre-image Attack Against Elliptic Curve Only Hash (ECOH). Cryptology ePrint Archive, Report 2009/168, 2009. http://eprint.iacr.org/.

[69] S. Halevi, W. E. Hall, and C. S. Jutla. The Hash Function Fugue. Submission to NIST, 2008. http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue.index.html/$FILE/NIST-submission-Oct08-fugue.pdf.

[70] Kimmo Halunen and Juha Röning. Preimage attacks against variants of very smooth hash. In *Proceedings of the 5th international conference on Advances in information and computer security*, IWSEC'10. Springer-Verlag, 2010.

[71] Bob Hattersley. Waterfall Hash - Algorithm Specification and Analysis. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/1/19/Waterfall_Specification_1.0.pdf.

[72] Phil Hawkes and Cameron McDonald. Submission to the SHA-3 Competition: The CHI Family of Cryptographic Hash Algorithms. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/2/2c/Chi_submission.pdf.

[73] Yedidya Hilewitz, Yiqun Yin, and Ruby Lee. Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation. In *Fast Software Encryption*, volume 5086 of *LNCS*, pages 173–188. Springer Berlin / Heidelberg, 2008.

[74] Shoichi Hirose, Hidenori Kuwakado, and Hirotaka Yoshida. SHA-3 Proposal: Lesamnta. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/5/5c/Lesamnta.pdf`.

[75] Gerhard Hoffman. Keccak implementation on GPU. `http://www.cayrel.net/spip.php?article189`.

[76] Sebastiaan Indesteege. The LANE hash function. Submission to NIST, 2008. `http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf`.

[77] Sebastiaan Indesteege, Florian Mendel, Bart Preneel, and Christian Rechberger. Collisions and other Non-Random Properties for Step-Reduced SHA-256. In *Selected Areas in Cryptography – SAC 2008*, volume 5381 of *LNCS*, pages 276–293. Springer, 2008.

[78] ISO 10118-3:2004. *Part 3: Dedicated hash-functions – Information technology – Security techniques – Hash-functions.* ISO, Geneva, Switzerland, 2004.

[79] ISO 10118-4:1998. *Part 4: Hash-functions using modular arithmetic: nformation technology – Security techniques – Hash-functions.* ISO, Geneva, Switzerland, 1998.

[80] Takanori Isobe and Taizo Shirai. Low-weight Pseudo Collision Attack on Shabal and Preimage Attack on Reduced Shabal-512. Cryptology ePrint Archive, Report 2010/434, 2010. `http://eprint.iacr.org/`.

[81] Tetsu Iwata, Kyoji Shibutani, Taizo Shirai, Shiho Moriai, and Toru Akishita. AURORA: A Cryptographic Hash Algorithm Family. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/b/ba/AURORA.pdf`.

[82] David P. Jablon. Strong password-only authenticated key exchange. *SIGCOMM Comput. Commun. Rev.*, 26:5–26, October 1996.

[83] J. Jansen. Use of SHA-2 Algorithms with RSA in DNSKEY and RRSIG Resource Records for DNSSEC. RFC 5702 (Proposed Standard), October 2009.

[84] Keting Jia. Practical Pseudo-Cryptanalysis of Luffa. Cryptology ePrint Archive, Report 2009/224, 2009. `http://eprint.iacr.org/`.

[85] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003. `http://www.ietf.org/rfc/rfc3447.txt`.

[86] Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In *CRYPTO*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.

[87] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/-CRC, 2007.

[88] Elif Kavun and Tolga Yalcin. A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications. In *Radio Frequency Identification: Security and Privacy Issues*, volume 6370 of *LNCS*, pages 258–269. Springer Berlin / Heidelberg, 2010.

[89] R. F. Kayser. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. Federal Register. Volume 72., November 2007. `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`.

[90] John Kelsey and Bruce Schneier. Second Preimages on n-bit Hash Functions for Much Less than 2^n Work. Cryptology ePrint Archive, Report 2004/304, 2004. `http://eprint.iacr.org/`.

[91] Dmitry Khovratovich, Alex Biryukov, and Ivica Nikolić. The Hash Function Cheetah: Specification and Supporting Documentation. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/c/ca/Cheetah.pdf`.

[92] Dmitry Khovratovich, María Naya-Plasencia, Andrea Röck, and Martin Schläffer. Cryptanalysis of Luffa v2 Components. In *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 388–409. Springer Berlin / Heidelberg, 2011.

[93] Miroslav Knežević and Ingrid Verbauwhede. Hardware evaluation of the Luffa hash family. In *WESS '09: Proceedings of the 4th Workshop on Embedded Systems Security*, pages 1–6. ACM, 2009.

[94] Kazuyuki Kobayashi, Jun Ikegami, Shin'ichiro Matsuo, Kazuo Sakiyama, and Kazuo Ohta. Evaluation of Hardware Performance for the SHA-3 Candidates Using SASEBO-GII. Cryptology ePrint Archive, Report 2010/010, 2010. `http://eprint.iacr.org/`.

[95] Kazuyuki Kobayashi, Jun Ikegami, Shin'ichiro Matsuo, Kazuo Sakiyama, and Kazuo Ohta. Evaluation of Hardware Performance for the SHA-3 Candidates Using SASEBO-GII. Cryptology ePrint Archive, Report 2010/010, 2010. http://eprint.iacr.org/.

[96] Michael Kounavis and Shay Gueron. Vortex: A New Family of One Way Hash Functions based on Rijndael Rounds and Carry-less Multiplication. Submission to NIST, 2008. http://eprint.iacr.org/2008/464.pdf.

[97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. http://www.ietf.org/rfc/rfc2104.txt.

[98] Gaëtan Leurent. MD4 is Not One-Way. In *Fast Software Encryption*, volume 5086 of *LNCS*, pages 412–428. Springer Berlin / Heidelberg, 2008.

[99] Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. SIMD Is a Message Digest. Submission to NIST, 2009. http://www.di.ens.fr/~leurent/files/SIMD.pdf.

[100] Max Locktyukhin. Improving the Performance of the Secure Hash Algorithm (SHA-1), March 2010. http://www.intel.com.

[101] C. Madson and R. Glenn. The Use of HMAC-MD5-96 within ESP and AH. RFC 2403 (Proposed Standard), November 1998. http://www.ietf.org/rfc/rfc2403.txt.

[102] C. Madson and R. Glenn. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404 (Proposed Standard), November 1998. http://www.ietf.org/rfc/rfc2404.txt.

[103] Smile Markovski and Aleksandra Mileva. 2.B.1 Algorithm Specification. Submission to NIST, 2008. http://inf.ugd.edu.mk/images/stories/file/Mileva/part2b1.pdf.

[104] Jason Worth Martin. ESSENCE: A Candidate Hashing Algorithm for the NIST Competition. Submission to NIST, 2008. http://www.math.jmu.edu/~martin/essence/Supporting_Documentation/essence_NIST.pdf.

[105] Mikhail Maslennikov. Secure hash algorithm MCSSHA-3. Submission to NIST, 2008. http://registercsp.nets.co.kr/MCSSHA/MCSSHA-3.pdf.

[106] S.M. Matyas, C.H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithm. IBM Techn. Disclosure Bull., 1985.

[107] Peter Maxwell. The Sgàil Cryptographic Hash Function. Submission to NIST, 2008. `http://www.allicient.co.uk/files/sgail/Supporting_Documentation/specification.pdf`.

[108] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[109] Ralph Merkle. A Certified Digital Signature. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, volume 435 of *LNCS*, pages 218–238. Springer Berlin / Heidelberg, 1990.

[110] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, volume 293 of *LNCS*, pages 369–378. Springer-Verlag, 1988.

[111] Ralph Charles Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford University, 1979.

[112] Miguel Montes and Daniel Penazzi. The TIB3 Hash. Submission to NIST, 2008. `http://www.famaf.unc.edu.ar/~penazzi/tib3/submitted/Supporting_Documentation/TIB3_Algorithm_Specification.pdf`.

[113] A. H. Namin and M. A. Hasan. Hardware Implementation of the Compression Function for Selected SHA-3 Candidates. Technical Report from CACR 2009-28, 2009. `http://www.cacr.math.uwaterloo.ca/techreports/2009/cacr2009-28.pdf`.

[114] Khang Nguyen, Bob Valentine, Erik Niemeyer, and Paul Lindberg. Preparing Applications for Intel Core Microarchitecture. `http://www.intel.com`.

[115] Ivica Nikolić, Alex Biryukov, and Dmitry Khovratovich. Hash family LUX - Algorithm Specifications and Supporting Documentation. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/f/f3/LUX.pdf`.

[116] NIST. The Advanced Encryption Standard, Federal Information Processing Standards Publication, FIPS Pub 197. Technical report, Department of Commerce, November 2001. `http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf`.

[117] Peter Novotney. Distinguisher for Shabal's Permutation Function. Cryptology ePrint Archive, Report 2010/398, 2010. http://eprint.iacr.org/.

[118] Kazuki Oikawa, Jiahong Wang, Eiichiro Kodama, and Toyoo Takata. Implementation and Evaluation of Cryptographic Algorithm on OpenCL. SCIS 2010, 3C4-2 (in Japanese), 2010.

[119] Thomaz Oliveira and Julio López. Improving the performance of Luffa Hash Algorithm. In *X Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 405–418, 2010.

[120] Sean O'Neil, Karsten Nohl, and Luca Henzen. EnRUPT Hash Function Specification. Submission to NIST, 2008. http://enrupt.com/SHA3/Supporting_Documentation/EnRUPT_Specification.pdf.

[121] M. O'Neill. Low-Cost SHA-1 Hash Function Architecture for RFID Tags. Workshop on RFID Security – RFIDSec'08, 2008.

[122] Thomas Pornin. Software library Sphlib 2.1. http://www.saphir2.com/sphlib/. Accessed 26 July 2010.

[123] B. Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven, 1983.

[124] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: a synthetic approach. In *Proceedings of the 13th annual international cryptology conference on Advances in cryptology*, pages 368–378. Springer-Verlag New York, Inc., 1994.

[125] M. O. Rabin. Digitalized signatures. In *Foundations of Secure Computation*, pages 155–166. Academic Press, 1978.

[126] Gary McGuire Rafael Alvarez and Antonio Zamora. The Tangle Hash Function. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/4/40/Tangle.pdf.

[127] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In *CT-RSA*, volume 3376 of *LNCS*, pages 58–71. Springer, 2005.

[128] R. Rivest. The MD4 Message-Digest Algorithm. RFC 1320 (Informational), April 1992. http://www.ietf.org/rfc/rfc1320.txt.

[129] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992. `http://www.ietf.org/rfc/rfc1321.txt`.

[130] R. Rivest. The MD6 hash function – A proposal to NIST for SHA-3. Submission to NIST, 2008. `http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf`.

[131] Gregory G. Rose. Design and Primitive Specification for Boole. Submission to NIST, 2008. `http://seer-grog.net/BoolePaper.pdf`.

[132] Gokay Saldaml, Cevahir Demirkiran, Megan Maguire, Carl Minden, Jacob Topper, Alex Troesch, Cody Walker, and Çetin Kaya Koç. Spectral Hash. Submission to NIST, 2008. `http://www.cs.ucsb.edu/~koc/shash/sHash.pdf`.

[133] Karl Scheibelhofer. A Bit-Slice Implementation of the Whirlpool Hash Function. In *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *LNCS*, pages 385–401. Springer Berlin / Heidelberg, 2006.

[134] Guillaume Sevestre. Implementation of Keccak hash function in Tree hashing mode on Nvidia GPU. `http://sites.google.com/site/keccaktreegpu/`.

[135] Jesse Walker Shay Gueron, Simon Johnson. SHA-512/256. Cryptology ePrint Archive, Report 2010/548, 2010. `http://eprint.iacr.org/`.

[136] Neil Sholer. Abacus: A Candidate for SHA-3. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/b/be/Abacus.pdf`.

[137] JH. Song, R. Poovendran, J. Lee, and T. Iwata. The AES-CMAC Algorithm. RFC 4493 (Informational), June 2006. `http://www.ietf.org/rfc/rfc4493.txt`.

[138] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Osvik, and Benne de Weger. Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate. In *Advances in Cryptology - CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer Berlin / Heidelberg, 2009.

[139] Douglas Stinson. *Cryptography: Theory and Practice,Second Edition*. CRC/C&H, 2nd edition, 2002.

[140] Joachim Strömbergson. Implementation of the Keccak Hash Function in FPGA Devices, 2010. `http://www.strombergson.com/files/Keccak_in_FPGAs.pdf`.

[141] Stefan Tillich, Martin Feldhofer, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Alexander Szekely. High-Speed Hardware Implementations of BLAKE, Blue Midnight Wish, CubeHash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. Cryptology ePrint Archive, Report 2009/510, 2009. http://eprint.iacr.org/.

[142] Mark Torgerson, Richard Schroeppel, Tim Draelos, Nathan Dautenhahn, Sean Malone, Andrea Walker, Michael Collins, and Hilarie Orman. The SANDstorm Hash. Submission to NIST, 2008. http://www.sandia.gov/scada/documents/SANDstorm_Submission_2008_10_30.pdf.

[143] Michal Trojnara. StreamHash Algorithm Specifications and Supporting Documentation. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/0/09/Streamhash.pdf.

[144] Meltem Sönmez Turan, Ray Perlner, Lawrence E. Bassham, William Burr, Donghoon Chang, Shu jen Chang, Morris J. Dworkin, John M. Kelsey, Souradyuti Paul, and Rene Peralta. Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition, February 2011. http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Round2_Report_NISTIR_7764.pdf.

[145] Kerem Varici, Onur Özen, and Çelebi Kocair. Sarmal: SHA-3 Proposal. Submission to NIST, 2008. http://www.metu.edu.tr/~e127761/Supporting_Documentation/SarmaL.pdf.

[146] Inc. VIA Technologies. VIA Nano Processor. White Paper, 2008. http://www.via.com.tw.

[147] Natarajan Vijayarangan. A new hash algorithm: Khichidi-1. Submission to NIST, 2008. http://ehash.iaik.tugraz.at/uploads/d/d4/Khichidi-1.pdf.

[148] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. http://eprint.iacr.org/.

[149] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *CRYPTO '05*, volume 3621 of *LNCS*, pages 17–36. Springer-Verlag, 2005.

[150] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[151] John Washburn. WaMM: A candidate algorithm for the SHA-3 competition. Submission to NIST, 2008. `http://www.washburnresearch.org/cryptography/archive/WaMM-SHA3.pdf`.

[152] Christian Wenzel-Benner and Jens Gräf. XBX: eXternal Benchmarking eXtension. `http://xbx.das-labor.org`. Accessed 27 January 2011.

[153] David A. Wilson. The DCH Hash Function. Submission to NIST, 2008. `http://web.mit.edu/dwilson/www/hash/dch/Supporting_Documentation/dch.pdf`.

[154] Robert S. Winternitz. Producing a One-Way Hash Function from DES. In *Advances in Cryptology — CRYPTO' 83 Proceedings*, pages 203–207. Plenum Press, 1983.

[155] Robert S. Winternitz. A Secure One-Way Hash Function Built from DES. *IEEE Symposium on Security and Privacy*, 0:88, 1984.

[156] Hongjun Wu. The Hash Function JH. Submission to NIST (updated), 2009. `http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh_round2.pdf`.

[157] Zijie Xu. Dynamic SHA. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/e/e2/DyamicSHA.pdf`.

[158] Zijie Xu. Dynamic SHA2. Submission to NIST, 2008. `http://ehash.iaik.tugraz.at/uploads/5/5b/DyamicSHA2.pdf`.

[159] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), January 2006. `http://www.ietf.org/rfc/rfc4252.txt`.

[160] G. Yuval. How to Swindle Rabin. *Cryptologia 3*, pages 187–189, 1979.

[161] Dai Zibin and Zhou Ning. FPGA implementation of SHA-1 algorithm. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 2, pages 1321–1324, 2003.

[162] Özgül Küçük. The Hash Function Hamsi. Submission to NIST (updated), 2009. `http://www.cosic.esat.kuleuven.be/publications/article-1203.pdf`.