Funções de Hashing Criptográficas

Ivan da Silva Sendin

Dissertação de Mestrado

Funções de Hashing Criptográficas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Ivan da Silva Sendin e aprovada pela Banca Examinadora.

Campinas, 21 de julho de 1999.

Ricardo Dahab (Doutor)

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

Instituto de Computação
Universidade Estadual de Campinas

Funções de Hashing Criptográficas

Ivan da Silva Sendin

Julho de 1999

Banca Examinadora:

- Ricardo Dahab (Doutor)
- Dr. Routo Terada
  IME - USP
- Dr. Paulo Lício de Geus
  IC - Unicamp
- Dr. Cid Carvalho de Souza (Suplente)
  IC - Unicamp

Prefácio

A cada dia que passa o computador tem uma participação maior na vida das pessoas. Formas tradicionais de interação estão sendo substituídas por suas equivalentes digitais, virtuais ou eletrônicas. Correio eletrônico, lojas virtuais, dinheiro digital, entre outros, já fazem parte do cotidiano das pessoas. Estas novas formas de interação não podem conviver com as formas tradicionais de garantir segurança. Uma assinatura em uma carta, uma impressão digital em um documento, um lacre de cera em um envelope, um cofre de aço cheio de cédulas também terão que ser substituídos por seus equivalentes eletrônicos. A criptografia moderna tem resposta para a maioria destes desafios.
As funções criptográficas tradicionais de ciframento foram projetadas com o objetivo de garantir privacidade dos dados, mas nem sempre são suficientes para garantir outros requisitos de segurança.
Devido ao fato de serem simples, rápidas e facilmente implementadas, tanto em *hardware* como em *software*, as funções criptográficas de hashing são utilizadas para gerar representações compactas de cadeias de bits (chamadas de *impressão digital*, *message digest*, *valor hash* ou simplesmente *hash*) que serão tratadas como seus identificadores únicos pelos protocolos criptográficos. Os principais usos das funções de hashing estão nos protocolos que visam garantir integridade, autenticidade e não repúdio.
Este texto tem como objetivo estudar as funções de hashing criptográficas apresentando conceitos teóricos, implementações, usos e questões relevantes quanto à sua segurança.

Agradecimentos

Ao Dahab, pela amizade e por ter me mostrado o maravilhoso mundo da criptografia,

aos meus pais, por terem me dado amor e educação desde os primeiros dias,

a Yeda, pela sua paciência, compreensão e amor nestes dois últimos anos,

e a Deus por ter colocado todas estas pessoas no meu caminho.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Algoritmos e Métodos

Capítulo 1

Introdução

1.1 Motivação

A cada dia que passa o computador tem uma participação maior na vida das pessoas. Formas tradicionais de interação estão sendo substituídas por suas equivalentes digitais, virtuais ou eletrônicas. Correio eletrônico, lojas virtuais, dinheiro digital, entre outros, já fazem parte do cotidiano das pessoas. Estas novas formas de interação não podem conviver com as formas tradicionais de garantir segurança. Uma assinatura em uma carta, uma impressão digital em um documento, um lacre de cera em um envelope, um cofre de aço cheio de cédulas também terão que ser substituídos por seus equivalentes eletrônicos. A criptografia moderna tem resposta para a maioria destes desafios.
Entretanto, a cada dia surgem novos desafios que também precisam ser respondidos.
As funções criptográficas tradicionais de ciframento foram projetadas com o objetivo de garantir privacidade dos dados, e apesar de serem utilizadas em vários protocolos de segurança, essas funções nem sempre são suficientes para garantir outros requisitos de segurança.
Devido ao fato de serem simples, rápidas e facilmente implementadas, tanto em *hardware* como em *software*, as funções criptográficas de hashing são utilizadas para gerar representações compactas de cadeias de bits (chamadas de *impressão digital*, *message digest*, *valor hash* ou simplesmente *hash*) que serão tratadas como seus identificadores únicos pelos protocolos criptográficos. Os principais usos das funções de hashing estão nos protocolos que visam garantir:

- Integridade: detecção e/ou prevenção contra alterações, maliciosas ou não, de informações;
- Autenticidade: associação da origem de um documento a uma entidade;

- Não repúdio: uma entidade que tenha criado um documento não poderá negar a sua criação posteriormente.

A seguir serão apresentadas as funções de hashing tradicionais, funções de hashing criptográficas, suas propriedades principais, suas aplicações e construções típicas

1.2 Funções de hashing tradicionais

As funções de hashing, também chamadas de função de espalhamento ou dispersão, mapeiam elementos de um universo grande X em outros elementos de um universo bem menor, geralmente posições em uma tabela de tamanho n:

$$h : X \to \{0, 1 \ldots n-1\}$$

O número $y = h(x)$ é chamado de valor hash de x.
As funções de hashing tradicionais são utilizadas para construir tabelas em aplicações que requeiram a implementação eficiente das operações de inserção, busca e remoção. O principal atrativo para o uso de tais tabelas é a velocidade: as operações tem tempo médio $O(1)$. Um exemplo clássico de uso de hashing é no armazenamento da tabela de símbolos de um compilador.

1.2.1 Colisões

Como a função de hashing mapeia um universo grande em um pequeno, certamente deverão existir situações onde valores diferentes m e m' possuirão o mesmo valor hash, isto é:

$$h(m) = h(m')$$

, o par m, m' é chamado de **colisão** de h.
O tratamento de colisões é um tópico que requer cuidado do projetista de software, e tem consequências diretas na performance dos algoritmos que se utilizam de funções de hashing. Assim, a minimização do número de colisões é sempre desejável. Uma condição necessária para tal é que a função de hashing comporte-se da maneira mais aleatória possível; isto é, para todo valor $x \in X$ a probabilidade de que $h(x)$ seja igual a um dado valor $y \in \{0, \ldots, n-1\}$ é próxima de $1/n$.
Existem dois métodos clássicos para resolução das colisões:

1. Encadeamento: a tabela não armazena diretamente os elementos, mas armazena referências para estruturas dinâmicas de armazenamento, geralmente listas ligadas. Assim, se dois elementos possuem o mesmo valor hash, estarão armazenados na mesma lista.

2. Hash múltiplo: caso dois elementos possuam o mesmo valor hash, novas posições da tabela são calculadas sistematicamente a partir da primeira, até que uma posição livre da tabela seja encontrada.

Informações mais detalhadas sobre funções de hashing tradicionais podem ser encontradas em [9, 23]

1.3 Funções de hashing criptográficas

Em contextos criptográficos, o comportamento aleatório de uma função de hashing é fundamental. Assim, para um domínio X e uma imagem Y, onde $|X| = 2^m$ e $|Y| = 2^n$, com $m > n$ é desejável que aproximadamente 2^{m-n} elementos x de X sejam tais que $y = h(x)$. Consequentemente, a probabilidade de que dois elementos quaisquer, não necessariamente distintos, de X tenham o mesmo hash é $2^{m-n}/2^m = 2^{-n}$, independente do tamanho de X.

1.3.1 Propriedades básicas das funções de hashing criptográficas

Para serem úteis em contextos criptográficos, as funções de hashing devem possuir as seguintes propriedades [4, 35]:

1. **eficiência** - dados h e x, $h(x)$ é eficientemente computável.
2. **compressão** - h mapeia uma entrada x de tamanho arbitrário em uma saída $h(x)$ de tamanho fixo, bem menor que $|x|$.
3. **unidirecionalidade** - dado um elemento $y \in Y$, é computacionalmente inviável achar um valor $x \in X$ tal que $h(x) = y$.
4. **resistência a segunda pré-imagem** - dado x é computacionalmente inviável encontrar $x' \neq x$ tal que $h(x') = h(x)$.
5. **resistência a colisões** - é computacionalmente inviável achar x, x' tal que $h(x) = h(x')$.

Note que:

- Resistência a colisões implica em resistência a segunda pré-imagem, já que a geração sistemática de uma segunda pré-imagem x' de $y = h(x)$ implica na capacidade de encontrar a colisão (x, x') sistematicamente.

- Resistência a colisão não implica em unidirecionalidade. Vejamos por que:

  Seja g uma função de hashing resistente a colisões que produz um valor hash de n bits. Vamos definir uma função h da seguinte forma:

$$h(x) = \begin{cases} 1||x \text{ se } x \text{ tem comprimento n,} \\ 0||g(x) \text{ nos demais casos,} \end{cases}$$

  onde $||$ denota concatenação.

  É fácil perceber que h será resistente a colisões, mas quando a entrada tiver tamanho n, h não será unidirecional.

Funções que possuem algumas das propriedades acima são as seguintes:

- **adição módulo** 2^n: a operação de adição módulo 2^n de cada uma das palavras de n bits que compoem uma cadeia de bits é facilmente computável e oferece compressão, mas não é resistente a colisões ou unidirecional.

- x^2 mod p: a função $f(x) = x^2 \bmod p$ tem comportamento próximo de aleatória; entretanto, $f(x)$ não é unidirecional, pois a raiz quadrada módulo um número primo é facilmente computável. Já a função $g(x) = x^2 \bmod n$, onde n é o produto de 2 números primos é unidirecional pois inverter a função é computacionalmente equivalente a fatorar n [4]. Apesar de $g(x)$ ser unidirecional ela não é um boa função de hashing criptográfica, pois não é resistente a colisões, já que, dado x,$-x$ é uma colisão.

- **funções unidirecionais baseadas em cifradores de blocos**: seja $E_k(x)$ um cifrador de blocos com a chave k, a função $f(x) = E_k(x) \oplus x$ é unidirecional mas não possui a propriedade de compressão.

1.3.2 Propriedades adicionais

Além das propriedades básicas, é desejável que as funções de hashing possuam propriedades adicionais que proporcionem maior segurança e flexibilidade em seu uso. As principais propriedades adicionais desejáveis em uma função de hashing são:

- **não-correlação entre a entrada e a saída:** deve ser muito difícil deduzir qualquer relação entre qualquer subconjunto de bits de saída e de entrada. Além disso, a função de hashing deve possuir o efeito avalanche, onde uma pequena alteração na entrada deve produzir uma grande alteração na saída;

- **resistência a colisões aproximadas:** deve ser difícil gerar duas entradas cujos valores hash diferem de um número pequeno de bits;
- **unidirecionalidade local:** a recuperação de uma parte da entrada a partir da saída deve ser tão difícil como recuperar a entrada inteira. Caso parte da entrada seja conhecida deve ser difícil encontrar o resto dela;
- **independência de relações matemáticas simples:** em algumas situações [5] as funções de hashing também devem ser:
  - livres de adição: deve ser difícil encontrar x, y e z tais que $h(x) = h(y) + h(z)$;
  - livres de multiplicação: deve ser difícil encontrar x, y e z tais que $h(x) = h(y) \times h(z)$;
  - livres de complementação: deve ser difícil encontrar x e y tal que $h(x) = \overline{h(y)}$

1.3.3 Funções de hashing iterativas

As funções de hashing iterativas utilizam o meta-método de Merkle-Damgård para transformar funções de compressão resistentes a colisões em funções de hashing. Este meta-método foi desenvolvido, de forma independente, por Ivan Damgård [10] e Ralph Merkle [25].

Meta-Método de Merkle-Damgård

Dada uma função de compressão resistente a colisões, $f : \{0,1\}^{\{m+n\}} \rightarrow \{0,1\}^n$ e uma cadeia x com um número arbitrário de bits, a função de hashing é construída da seguinte forma:

1. Primeiramente x é dividida em, digamos, t blocos x_i de comprimento m cada. Caso $|x|$ não seja múltiplo de m, x deve ser preenchida até que resulta numa cadeia de comprimento múltiplo de m.
2. $IV := \{0\}^m$.
3. O valor hash de x, $h(x)$, é calculado iterativamente da seguinte forma:

$$\begin{cases} H_0 := IV; \\ H_i := f(H_{i-1} || x_i) \text{ para } 1 \leq i \leq t; \\ h(x) := H_t. \end{cases}$$

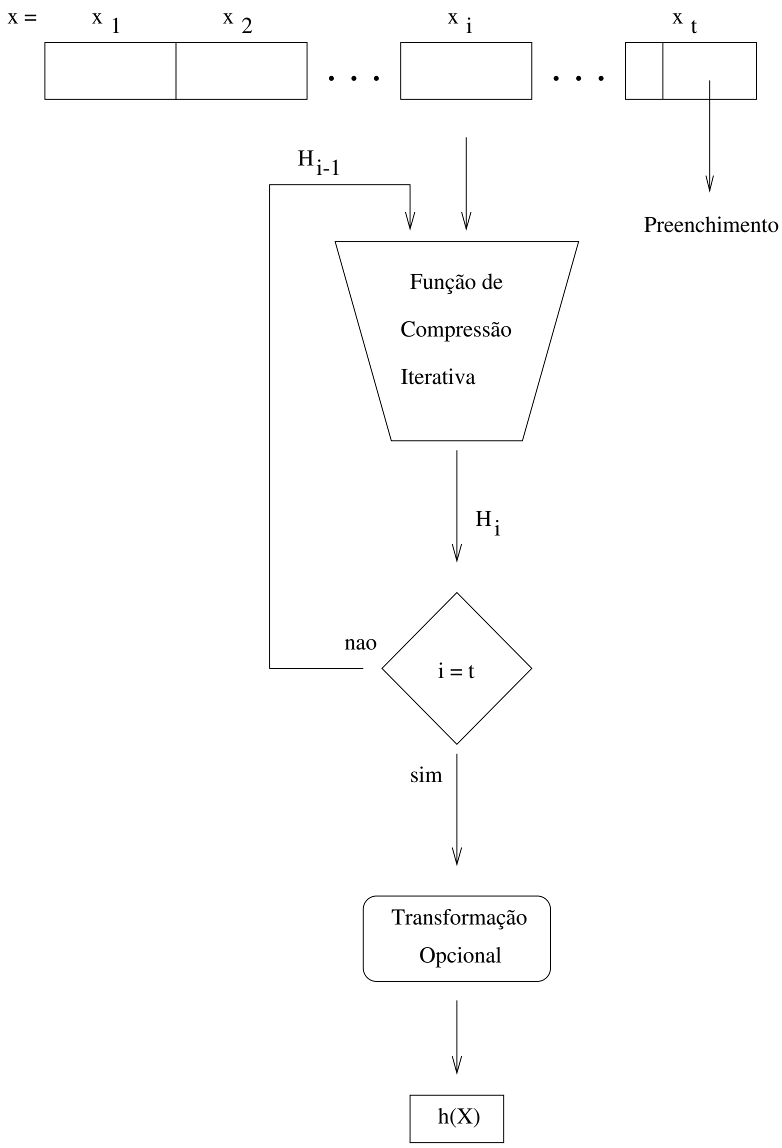

Figura 1.1: Função de hashing iterativa

O processo está representado na Figura 1.1.
As funções de hashing iterativas criadas com o meta-método possuem a **propriedade de encadeamento**.
Dadas i cadeias de bits $x_1, x_2 \cdots x_i$, cada uma com o tamanho igual ao da entrada da função de compressão de h, a propriedade de encadeamento é:

$$h(x_1 \cdots x_i) = h^{h(x_1 \cdots x_j)}(x_{j+1} \cdots x_i)$$

onde $h^y(x)$ é uma função de hashing h usando o valor inicial IV igual a y.

Um exemplo de função de compressão usando quadrado modular

Vamos apresentar agora um exemplo de função de compressão que pode ser usada para construir funções de hashing iterativas. A segurança deste exemplo está na dificuldade de se extrair raiz quadrada modulo um produto de dois números primos grandes.
Escolhido um número $n = pq$, onde p e q são números primos, a função f de compressão é definida da seguinte forma:

$$f(x) = (c||x)^2 \pmod n$$

onde c é uma constante grande o suficiente para que a redução modular sempre ocorra.

1.3.4 Aplicações

As funções de hashing são tipicamente utilizadas para gerar MDC's (*Manipulation Detection Code*) ou MAC's (*Message Authentication Code*) de mensagens e para serem usadas em sistemas de assinaturas digitais. Descreveremos essas aplicações com detalhes no próximo capítulo. Fazemos a seguir um esboço dessas aplicacões.
O MDC de uma mensagem m é simplesmente $h(m)$, que é transmitido juntamente com a mensagem m. Na recepção, o hash de m é re-calculado e a mensagem é aceita se o valor recebido e o re-calculado forem iguais. O MAC de uma mensagem m é, a grosso modo, um valor hash, função de m e de uma chave secreta k, conhecida apenas pelas duas entidades em comunicação. Dessa forma, uma mensagem recebida com MAC correto é também uma garantia de origem dela.
Os mecanismos atuais de assinaturas digitais não necessitam de funções de hashing para a sua funcionalidade; entretanto, os algoritmos de assinaturas e verificação são muito lentos. As funções de hashing são utilizadas para gerar representações compactas e únicas das mensagens que serão assinadas, agora, mais rapidamente.

1.3.5 Outras aplicações criptográficas interessantes

Aproveitando a alta velocidade das funções de hashing, foram desenvolvidos usos diferentes dos apresentados anteriormente. Entre estes destacamos:

Micropagamentos. Sistemas de micropagamentos devem fazer uso de mecanismos rápidos para prover níveis de segurança compatíveis com os valores envolvidos nas transações: valores pequenos não precisam de segurança extrema. Utilizando este conceito foram desenvolvidos os sistemas PayWord e MicroMint, que serão descritos em 3.1.

Controle de acesso. A maioria dos sistemas de controle de acesso utilizam funções de hashing. O uso de funções de hashing unidirecionais permite que as senhas sejam armazenadas de maneira segura, protegidas até dos administradores do sistema (veja 3.3.1), e possibilitam uma verificação rápida. Sistemas mais seguros, como o de senhas descartáveis (veja 3.3.2), ou com funcionalidades diferentes, como o sistema HashCash, descrito em 3.3.3, também usam hashing.

Geração de cadeias pseudo-aleatórias. Números pseudo-aleatórios são de extrema importância para os protocolos criptográficos. Devido ao seu comportamento aleatório, as funções de hashing são úteis na construção de funções geradoras de números pseudo-aleatórios (veja 3.2).

Outros objetivos criptográficos. Objetivos criptográficos além da autenticidade e integridade podem ser providos utilizando funções de hashing. Por exemplo, confidencialidade (veja 3.4) ou assinaturas rápidas de fluxos (veja 3.5) podem ser obtidas usando funções de hashing.

1.3.6 Construções típicas

Construções práticas de funções de hashing são iterativas: uma função de compressão é aplicada iterativamente na mensagem até que o valor hash seja obtido. Em geral, o que difere uma função de hashing de outra é a função de compressão. Estas funções podem ser construídas utilizando cifradores de blocos ou serem desenvolvidas especificamente para funções de hashing, como ocorre no MD4, MD5, SHA-1 e RIPEMD.
As funções de MAC também são iterativas e podem ser construídas usando cifradores, funções de hashing ou serem completamente desenvolvidas para este fim.

1.4 Resumo das contribuições da tese

As duas principais contribuições foram:

1. A consolidação do tema, dado que não existem bons textos em português ou inglês que unifiquem a discussão sobre funções de hashing, com exemplos de aplicações, construções e protocolos importantes.

2. As implementações: as implementações estão sendo utilizadas em outros projetos e estão disponíveis no site do Instituto de Computação da Unicamp.

1.5 Organização da Dissertação

No Capítulo 2 é mostrado como as funções de hashing são utilizadas para prover integridade, autenticação e não-repúdio. No capítulo seguinte são mostrados outros usos interessantes para as funções de hashing.
No Capítulo 4 são descritas as principais funções de hashing existentes e alguns ataques desenvolvidos para serem usados contra elas. No capítulo seguinte o mesmo é feito com as funções de MAC.
No sexto capítulo são abordadas questões relativas à segurança: objetivos dos adversários, ataques e métodos para tornar as funções de hashing mais seguras.
No último capítulo são apresentados aspectos práticos do uso de hashing, sugestões para trabalhos futuros e a conclusão.

Capítulo 2

Aplicações padrão

Nesta parte do trabalho serão descritas formas de se usar funções de hashing para prover os seguintes objetivos criptográficos: integridade de dados, autenticação de origem e não repúdio.

2.1 Integridade de dados

Integridade é definida como a garantia de que dados não tenham sido alterados de forma não autorizada, acidentalmente ou não, desde sua criação, transmissão ou armazenamento. Alterações possíveis de ocorrerem em uma informação digital são:

- Inserção de bits, ou até de mensagens inteiras válidas;
- remoção de bits;
- reordenação de bits;
- inversão de bits;
- qualquer combinação dos ítens anteriores.

Existem dois processos para verificar a integridade de uma informação:

1. Verificação do formato esperado: em alguns tipos de dados, é possível verificar sua integridade observando um formato esperado. Por exemplo, em um arquivo texto, uma alteração acidental provavelmente será detectada pois alguma palavra ficará sem sentido. Outro exemplo cotidiano vem do formato rígido dos pacotes IP.
2. Verificação de redundâncias: pode-se adicionar redundâncias às informações, que serão checadas posteriormente. Por exemplo, em uma comunicação via *modem* o

emissor adiciona um bit (chamado de bit de paridade) para cada grupo de bits a ser enviado. O valor do bit adicionado é determinado por uma função conhecida pelo emissor e pelo receptor. Após o recebimento da mensagem o receptor usa a mesma função para calcular o bit de paridade; se este tiver o mesmo valor do bit recebido, a mensagem é considerada íntegra; caso contrário o receptor pede ao emissor para que transmita a mensagem novamente.

Técnicas criptográficas de integridade usam a criação e verificação de redundâncias, chamadas de MDC (*Manipulation Detection Code*) aliadas a informações secretas ou canais seguros.
As funções de hashing podem ser usadas para gerar MDC's de mensagens e prover integridade de dados em canais inseguros. O processo de verificação de integridade de uma mensagem x é descrito da seguinte forma (Figura 2.1):

1. Geração do MDC: $MDC(x) := h(x)$

2. Transmissão ou armazenamento do MDC: o MDC deve ser transmitido e armazenado de uma maneira segura. Para tal, pode-se usar ciframento ou um canal autêntico[1] . Um adversário que puder trocar $h(x)$ por $h(x')$ poderá fazer com que a mensagem fraudulenta x' seja aceita no lugar da mensagem original x. A mensagem não precisa ser transmitida de forma segura.

3. Verificação: tendo recebido x' e $MDC(x)$, onde x' é a versão transmitida de x, calculado-se
$$y = h(x').$$
Se $y = MDC(x)$, conclui-se que $x' = x$ e a mensagem é íntegra. Caso contrário, conclui-se que a mensagem foi modificada.

O MDC pode ser utilizado em:

- Distribuição de software: um distribuidor de software pode calcular os MDC's dos seus softwares, publicá-los em algum lugar seguro (no próprio site, jornais, etc) e fazer a sua distribuição usando *mirror sites*. Assim um usuário que obtenha uma cópia de um *mirror site* necessita apenas recalcular o hash de sua cópia e compará-lo com o hash fornecido diretamente pelo distribuidor.

[1]Canal autêntico é um canal que garante a origem da informação e a proteje contra modificações não autorizadas. Exemplos de canais autênticos são: uma página na Internet, uma publicação em um jornal, uma transmissão de voz via rede telefônica ou um canal seguro tradicional.

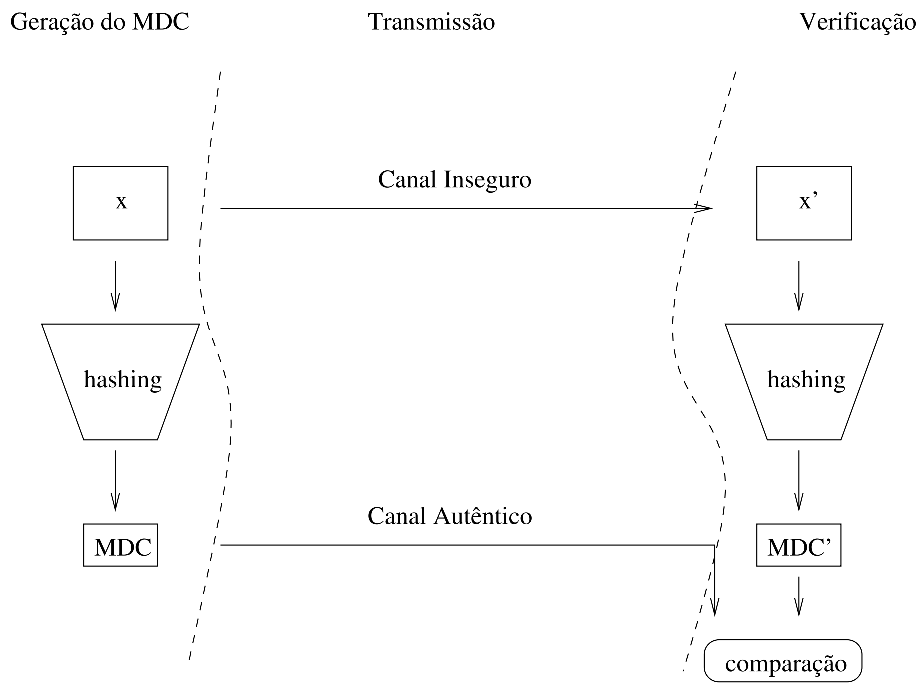

Figura 2.1: Verificação de Integridade usando MDC

- Detecção de vírus ou alterações maliciosas de arquivos: calcula-se os MDC's dos arquivos de um sistema de arquivos, armazenando-os em um disquete. Periodicamente faz-se comparações para detectar alterações.
- Comunicação em geral: os MDC's podem ser utilizados para garantir integridade em qualquer comunicação; para tal basta providenciar uma maneira segura de transmiti-los.

2.1.1 Ciframento e Integridade

Por algum tempo, acreditou-se que o ciframento, sozinho, fosse capaz de prover integridade de dados. Esta crença deve-se ao fato de que a alteração de apenas um bit do texto cifrado, utilizando um cifrador de blocos, deve produzir uma alteração muito maior no texto claro, tornando possível a detecção da alteração. Entretanto, estas considerações não são sempre corretas:

- Os cifradores de fluxo são seguros para garantir privacidade; sistemas *one-time pad* oferecem até mesmo segurança incondicional, mas um adversário pode inverter facilmente alguns bits do texto claro invertendo os bits correspondentes do texto cifrado.

- Cifradores de blocos em modo CFB funcionam da mesma maneira que os cifradores de fluxo, possuindo o mesmo problema.
- Uma mensagem maior que um bloco, cifrada em modo ECB está sujeita a reordenação, remoção ou inclusão de blocos.
- Alguns tipos de dados não possuem significado detectável, principalmente por computadores.
- Mesmo em dados com um formato específico, a alteração de alguns blocos pode passar desapercebida.

2.2 Autenticação de Origem

Autenticação de origem, o simplesmente autenticação, é a propriedade que associa a criação de uma mensagem a uma entidade. Entre os métodos de autenticação estão:

- Códigos de Autenticação de Mensagens (MAC);
- assinaturas digitais.

Os métodos de autenticação sempre usam alguma informação secreta. Quando o MAC é utilizado, uma chave secreta é compartilhada entre as entidades envolvidas; quando as assinaturas digitais são utilizadas, a entidade que assina possui uma chave privada e as demais partes possuem sua chave pública.

Uma mensagem autêntica também é, implicitamente, íntegra. Não faz sentido associar a criação de uma mensagem a uma entidade se a mensagem foi alterada.

Para prover autenticidade são usadas funções de hashing com dois parâmetros: os dados e uma chave secreta; estas funções são representadas por $h(k,x)$ ou $h_k(x)$. O valor hash gerado é chamado de MAC (*Message Authentication Code*).

O uso de MAC's é parecido com o de hashing: primeiro a geração do MAC, seguida pela transmissão da mensagem e do MAC e, por último, a verificação (veja Figura 2.2). A principal diferença é que o MAC não precisa ser protegido por ciframento ou canal seguro; ele já é seguro devido ao uso da chave compartilhada.

A autenticação feita com o MAC é chamada de **autenticação simétrica**; as partes envolvidas no processo possuem o mesmo conhecimento (a chave secreta) e as mesmas possibilidades (gerar e conferir o MAC). Devido à igualdade entre as partes envolvidas, o MAC não possui a garantia de não-repúdio: se uma entidade é capaz de verificar o MAC, ela também é capaz de gerá-lo.

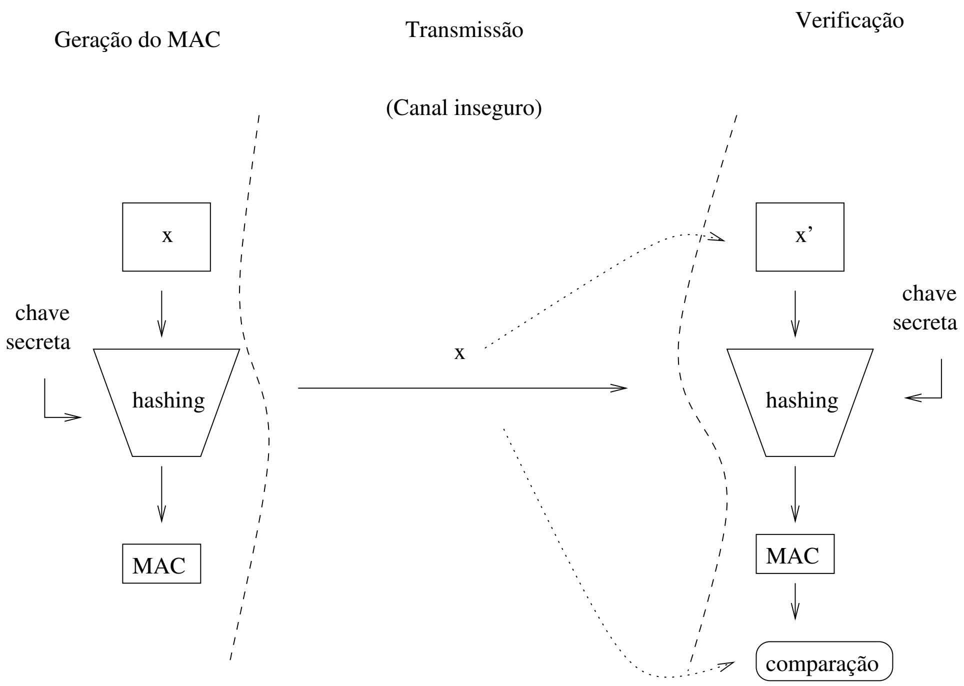

Figura 2.2: Autenticação usando MAC

2.3 Não-repúdio

Não-repúdio, é a prevenção da negação de acordos ou atos [4]. Este objetivo criptográfico pode ser conseguido usando assinaturas digitais.
Documentos convencionais e assinados visam garantir integridade, autenticidade e não-repúdio. A integridade é obtida com a observação das características físicas do documento: letras borradas devem invalidar o documento; as assinaturas presentes no final de um documento dão a ele autenticidade e não-repúdio. Pessoas olham a assinatura e reconhecem-na como a do autor do documento, e este não pode negar que criou (ou concordou com) o conteúdo do documento. Outra característica da assinatura é que ela está fisicamente relacionada ao documento, não podendo ser transferida para outro documento.
Assinaturas digitais são mecanismos criptográficos que proporcionam às mensagens digitais os mesmos objetivos de segurança que uma assinatura proporciona aos documentos escritos: integridade, autenticidade e não-repúdio.
Existem vários tipos de assinaturas digitais, com suas vantagens e desvantagens. Atualmente, o sistema RSA, baseado em fatoração de números inteiros, é o mais usado para gerar assinaturas digitais, mas apresenta alguns problemas:

- Segurança: o RSA trabalha com blocos pequenos (512 a 2048 bits, dependendo do grau de segurança). Caso o RSA seja utilizado sozinho para gerar assinaturas, cada bloco deve ser assinado individualmente, permitindo a reordenação, remoção ou inserção não-autorizada de blocos.

- Velocidade: os algoritmos usados pelo RSA são lentos; a assinatura de mensagens longas é impraticável.

As funções de hashing são usadas junto com o RSA para suprir suas falhas. Ao invés de se assinar uma mensagem, deve-se assinar o hash da mensagem. Desta forma obtém-se as seguintes vantagens:

- Aumento de velocidade: o hash é normalmente muito menor que as mensagens utilizadas, tornando os processos de assinatura e verificação mais rápidos,

- Aumento de segurança: devido à dificuldade de encontrar duas mensagens que possuam o mesmo valor hash, este pode ser usado como a representação única de toda a mensagem; assim, assinar o hash proporciona a mesma segurança que uma assinatura na mensagem completa proporcionaria, e não de cada bloco individualmente.

2.4 Exemplo

O serviço **Digital Notary**[37], da Surety, usa funções de hashing para produzir certificados. Do modo como o serviço é oferecido, também são obtidos:

- Privacidade: a entidade autenticadora, a Surety, não fica conhecendo o conteúdo do documento,

- confiança distribuída: não é preciso confiar na entidade autenticadora para confiar na autenticação do documento. Mesmo a Surety não consegue modificar a data e hora em que um certificado foi feito.

Funcionamento
Para obter um certificado de uma mensagem M, o cliente deve gerar dois valores hash desta mensagem, um usando a função MD5 (veja 4.3) e o outro usando o SHA-1 (veja 4.4), concatená-los e enviá-los para a Surety, que não tem acesso à mensagem, mas apenas aos seus valores hash, garantindo o sigilo das mesmas. A Surety retorna ao cliente um certificado com um registro de validade temporal (*timestamp*).
O certificado é a raiz de uma árvore cujas folhas são os valores hash enviados à Surety em um período pré-definido de tempo (figura 2.3). A raiz da árvore, que é o valor hash de todos os valores hash, é então publicada em jornais, sites na Internet e em

CD-ROM's. Os valores intermediários da árvore são armazenados na Surety para possibilitar a verificação dos certificados.
A verificação de uma mensagem M é feita reconstruindo o certificado utilizando $h(M)$ e os valores intermediários armazenados anteriormente.
A distribuição dos valores hash em lugares diferentes e independentes permitem que os certificados sejam confiáveis mesmo que a Surety, possivelmente, não o seja.

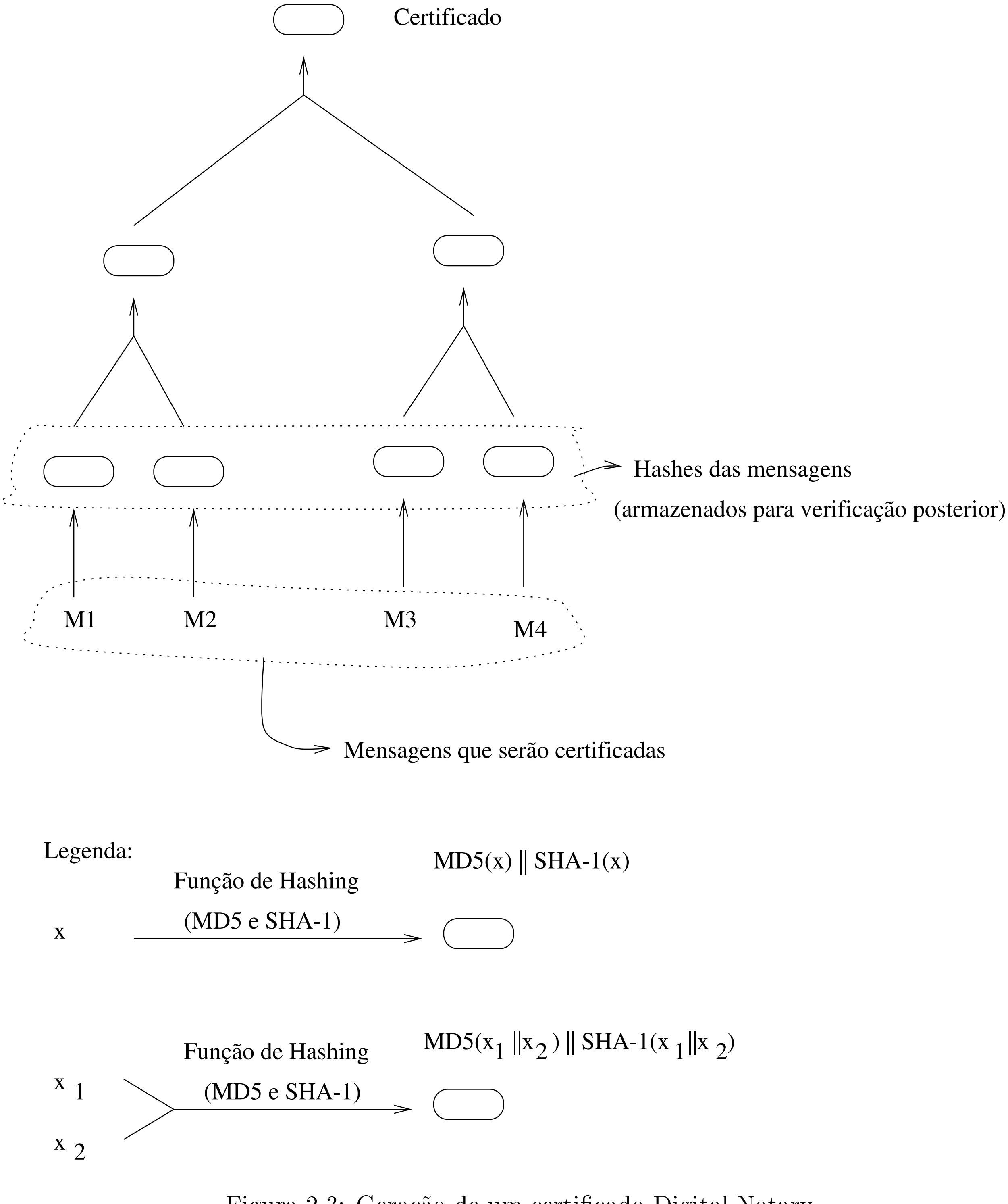

Figura 2.3: Geração de um certificado Digital Notary

Capítulo 3

Aplicações específicas

As aplicações que serão descritas nesta parte do trabalho usam as funções de hashing de uma maneira inovadora, não se atendo aos objetivos primários descritos anteriormente no Capítulo 2.

3.1 Micropagamentos Eletrônicos

Os sistemas de micropagamentos disponibilizam mecanismos para transações envolvendo pequenas quantias, de frações de centavo a alguns reais. As seguintes entidades estão presentes em um sistema de micropagamentos:

- Cliente
- Vendedor
- Corretor ou entidade responsável pelo controle do sistema

Devido aos baixos valores envolvidos nas transações, as seguintes características geralmente estão presentes nesses sistemas:

- Economia de processamento: as transações envolvendo micropagamentos não podem consumir muito processamento, caso contrário o custo do suporte à transação pode ser maior que o valor dela. Portanto, deve-se minimizar o processamento efetuado nas transações, principalmente assinaturas digitais.

- Comunicação off-line: a comunicação on-line com o corretor deve ser mínima. Processamento em batch, aproveitando horários de menor tráfego, são preferíveis ao processamento on-line.

- Venda de informações: o produto vendido é geralmente algum tipo de informação ou serviço on-line.

- Detecção de fraudes: a segurança dos sistemas de micropagamentos é pequena. Alguns ataques são previstos, mas não deverão ocorrer devido ao seu custo e à possibilidade de detecção da fraude.

Descrevemos a seguir dois sistemas de micropagamentos, o PayWord e MicroMint.

3.1.1 PayWord

O PayWord[33] utiliza a propriedade de unidirecionalidade das funções de hashing. É um sistema baseado em crédito: o cliente primeiramente estabelece crédito com o corretor para depois gastá-lo com os vendedores utilizando fichas (chamadas de paywords).

1. **Inicialização do PayWord**

   Antes que as vendas se iniciem, algumas relações são estabelecidas:

   - Vendedor/Corretor: o vendedor se cadastra no corretor, obtendo de maneira segura a chave pública do corretor e estabelece uma forma de pagamento distinta do PayWord (o pagamento feito pelo corretor ao vendedor envolve grandes quantias, impossibilitanto o uso do PayWord). Também é estabelecido o valor das paywords utilizadas por este corretor.
   - Cliente/Corretor: o cliente se cadastra no corretor e obtém deste um certificado (Figura 3.1). Também é definida uma forma de pagamento, geralmente cartão de crédito, e é estabelecido um crédito, tipicamente entre R$20 e R$50, que pode ser renovado automaticamente.

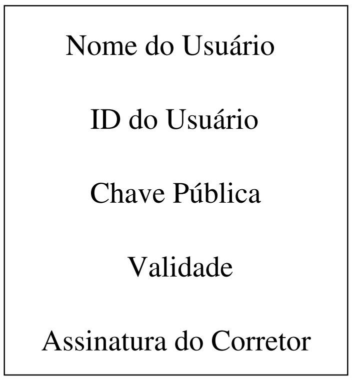

Figura 3.1: Certificado Payword

2. **Venda e pagamento com o PayWord**

   Quando um cliente decide fazer uma compra em um determinado vendedor, ele gera uma cadeia de paywords da seguinte forma (Figura 3.2):

   - Define p como o número de produtos que o cliente espera adquirir nesta compra[1];
   - gera um número aleatório w_p;
   - calcula $w_i = h(w_{i+1})$, para $i = p - 1, p - 2 \ldots, 0$;
   - produz um compromisso (*commitment*) composto de w_0 e do identificador do vendedor;
   - w_0 é concatenado com um identificador do vendedor (endereço IP, CGC ou número de série) e o resultado desta concatenação é assinado pelo cliente, produzindo um compromisso.

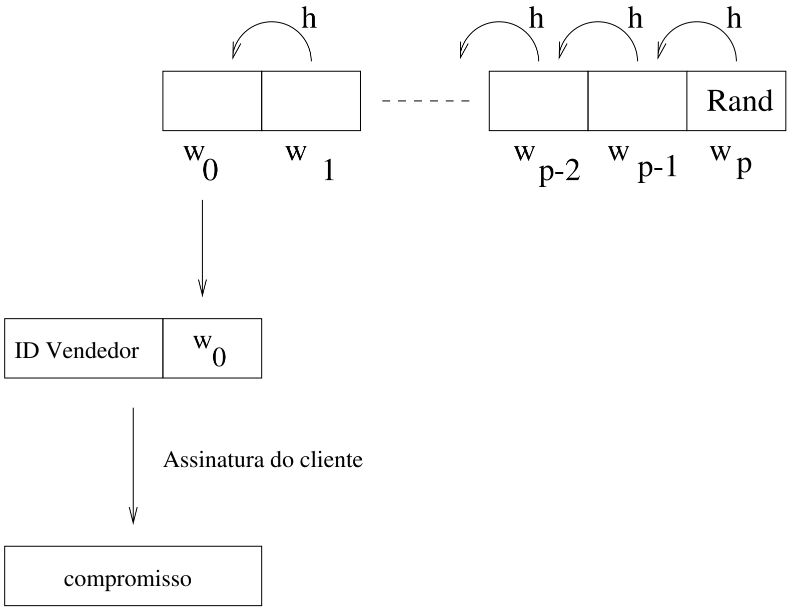

Figura 3.2: Geração da cadeia de payword

Os valores $w_1, w_2 ... w_p$ são as paywords que serão utilizadas nos pagamentos do cliente a este vendedor específico. Devido à propriedade de unidirecionalidade da função de hashing h utilizada, dada uma payword de índice i, é fácil calcular a payword de índice $i - 1$, mas é inviável calcular a de índice $i + 1$.

[1]No payword uma compra é formada pelo conjunto de produtos adquiridos em um certo prazo de tempo, utilizando o mesmo compromisso.

Antes de efetuar qualquer compra, o cliente envia seu certificado e o compromisso ao vendedor, que deve verificar as validades do certificado e do compromisso.

Agora o cliente está apto a efetuar as compras neste vendedor. Conforme os produtos são comprados o cliente envia, em ordem crescente, as paywords e seus respectivos índices (w_i, i). O vendedor verifica as paywords usando a função h e compara seu resultado com a payword recebida anteriormente (a primeira payword é comparada com o compromisso); somente a última payword recebida precisa ser armazenada.

Ao final do dia, após vários clientes terem feito várias compras, o vendedor envia ao corretor os compromissos e seus respectivos pares payword/índice (w_i, i) recebidos. O corretor aplica i vezes a função h nas paywords recebidas e compara o resultado com o compromisso: se foram iguais, o crédito do cliente é diminuído e o valor correspondente é pago ao vendedor. O processo de venda e pagamento está ilustrado na Figura 3.3.

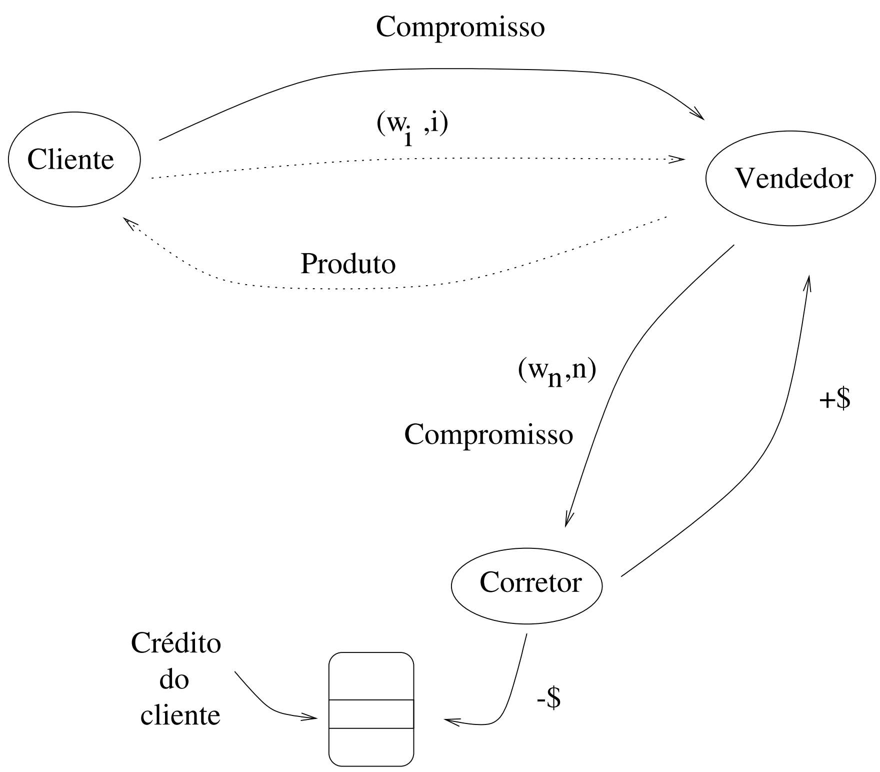

Figura 3.3: Venda e pagamento usando o PayWord

3. **Custos do PayWord**

Os custos para os envolvidos na utilização do sistema são:

- Corretor(off-line)
  - Para cada cliente: geração do certificado (assinatura).
  - Para cada compra: verificação do compromisso (verificação de assinatura).
  - Para cada produto: verificação de uma payword(hashing).
- Vendedor (on-line)
  - Para cada compra: verificação do certificado e do compromisso (verificação de assinaturas).
  - Para cada produto: verificação da payword (hashing).
- Cliente (on-line)
  - Para cada compra: geração do compromisso (assinatura).
  - Para cada produto: a geração e armazenamento da cadeia de payword (hashing).

4. **Segurança do PayWord**

   O PayWord é um sistema de micropagamento seguro, suficiente para ser usado com valores de até R$20. Não são conhecidos ataques eficientes ao sistema: os ataques se limitam a ataques às primitivas criptográficas utilizadas (assinaturas e hashing).

   Um problema do PayWord, que não é propriamente um ataque, é que é permitido ao cliente gastar mais dinheiro do que possui, da mesma forma que o sistema de cheques utilizados no Brasil. Isto porque a verificação de crédito do cliente é feita posteriormente à transação.

3.1.2 MicroMint

O MicroMint [33] foi projetado para oferecer segurança a um custo muito baixo, sem usar operações de chave pública.
As moedas do MicroMint são pares de cadeias de bits que possuem o mesmo valor hash (Figura 3.4). Elas são produzidas pelo corretor, que as vende para os clientes, que gastam-nas com os vendedores, que por sua vez retornam as moedas ao corretor, fechando o ciclo.

1. **Geração das moedas**

   O corretor produz as moedas MicroMint usando um processo de busca exaustiva: são gerados e armazenados valores aleatórios e seus respectivos valores hash até que um número suficiente de moedas (colisões) tenham sido obtidas. A função de

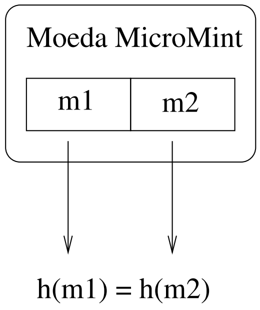

Figura 3.4: Moeda MicroMint válida

hashing utilizada pelo sistema é produzida pelo truncamento, ou dobramento com ou-exclusivo, do valor de uma função de hashing criptográfica tradicional; esta modificação tem o objetivo de diminuir o comprimento n do valor hash para aproximadamente 30 ou 40 bits.

Utilizando este processo de busca de colisões, a primeira moeda deve surgir após $2^{n/2}$ operações. Após este custo inicial, cada iteração da função de hashing tem chance maior ou igual a 1/2 de produzir uma moeda.

Esta particularidade das buscas por colisões, proveniente do paradoxo do aniversário (veja 6.4.2), faz com que o custo de produção de poucas moedas seja maior, por moeda, do que o custo de produzir muitas moedas, desestimulando a fraude.

2. **Distribuição e gasto**

   Os clientes compram lotes de moedas utilizando sistemas tradicionais de pagamentos. Por questões de segurança, o corretor deve manter uma tabela informando para quem as moedas foram vendidas. As moedas são gastas com os vendedores que as retornam ao corretor. Cada vez que a moeda é transferida, sua validade é conferida rapidamente utilizando a função de hashing.

3. **Custos**

   Os custos do MicroMint se concentram no corretor, que gasta muito tempo para gerar as moedas. Mas este custo alto não é proibitivo, pois o processo de geração de moedas é feito *off-line*

4. **Segurança**

   Alguns aspectos de segurança do MicroMint devem ser considerados:

   - As moedas MicroMint são simples colisões de hashing, não possuindo qualquer identificação, permitindo que:

  - o cliente gaste várias vezes a mesma moeda;
  - o vendedor gaste a moeda recebida com outro vendedor e cobre esta moeda do corretor;
  - moedas roubadas, ou copiadas, sejam utilizadas.

  O MicroMint não pode prevenir estes tipos de ataques; entretanto, a fraude pode ser detectada se o corretor mantiver um cadastro relacionando as moedas com os clientes.

- k-way collision: o modelo de MicroMint descrito anteriormente utiliza duas cadeias com mesmo valor hash como moeda, e é chamado de 2-*way collision*. Uma variação do modelo que utilize k colisões (k-*way collision*) aumenta a segurança do sistema, pois aumenta o custo de produção de poucas moedas. Para $k \geq 2$ a primeira colisão deve ocorrer depois de $2^{n(k-1)/k}$ operações de hashing.
- Validade e formatação da moeda: o corretor pode especificar um formato e uma validade paras moedas. Por exemplo, ele pode informar aos seus clientes que para um determinado mês as moedas deverão iniciar com uma cadeia de bits r. Esta cadeia r é escolhida pelo corretor e mantida em segredo até o momento em que se inicia a distribuição das moedas. Um fraudador terá pouco tempo para gerar e gastar suas próprias moedas.

3.2 Geração de cadeias pseudo aleatórias

Os geradores de números pseudo aleatórios (GNPA) são extremamente importantes para criptografia. É difícil encontrar um protocolo criptográfico que não faça uso de algum gerador de números pseudo aleatório.

3.2.1 Funcionamento de GNPA's

Os GNPA's são funções que utilizam um estado secreto para gerar sequências imprevisíveis (para um adversário que não conheça o estado secreto) de números. A cada número gerado o estado secreto do GNPA deve ser alterado pelo próprio GNPA ou por algum evento do sistema(Figura 3.5).
Devido à velocidade, unidirecionalidade e imprevisibilidade, as funções de hashing são boas escolhas para serem usadas como função interna de um GNPA.
Um fator importante na segurança dos GNPA's é a escolha da semente(*seed*) utilizada para inicializar o estado secreto do GNPA. O estado secreto inicial deve ser uma função de uma ou mais fontes de entropia do sistema, entre elas:

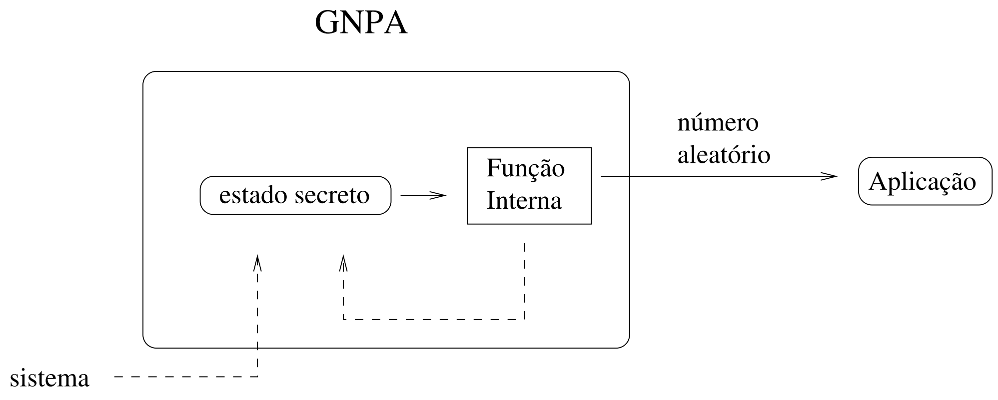

Figura 3.5: Funcionamento de um GNPA

- Interações do usuário com o sistema
  - Velocidade de digitação
  - Microfone
  - Câmeras
- Fontes do próprio sistema
  - relógio
  - memória (livre ou utilizada)
  - processos
- Hardware específico

Deve-se escolher cuidadosamente as fontes de entropia: caso o adversário conheça-as, mesmo que parcialmente, ele poderá concretizar um ataque [22]. Uma vez escolhidas as fontes, a semente é o valor hash da união das mesmas.

3.2.2 Exemplo de implementação: GNPA do DSA

O DSA[2] do NIST especifica um GNPA, baseado no SHA1, que deve ser utilizado para gerar as chaves secretas dos usuários e um número secreto para cada mensagem assinada. Uma vez escolhida a semente s (o DSA não especifica como) e uma entrada opcional w, a função geradora funciona da seguinte forma:

$temp$:= SHA1(w+s);
s:=s+$temp$+1;
retorna $temp$.

As operações de soma podem ser módulo 2^{160} a 2^{512}, dependendo da implementação.

3.3 Controle de Acesso

3.3.1 Login

As funções de hashing são também usadas para prover mecanismos de identificação de um usuário em um sistema de computadores; este mecanismo é conhecido como *login*.
Nos primeiros sistemas de computação existentes, as senhas dos usuários eram armazenadas de forma clara, não cifrada, em um arquivo de senhas. A identificação era simples: o usuário digitava seu *user id* e senha, o sistema então verificava a identidade do usuário comparando a senha digitada com a senha armazenada no arquivo [20].
A proteção das senhas ficava por conta do sistema operacional; somente alguns usuários e programas podiam acessar o arquivo. Este tipo de proteção não era eficiente: alguns *bugs* nos programas que acessavam o arquivo de senhas eram explorados, comprometendo as senhas.
Para solucionar este problema, o sistema de controle de acesso foi modificado. As senhas não são mais armazenadas de forma clara, mas sim o valor hash das senhas. A identificação do usuário continua simples: é comparado o valor hash da senha digitada com o valor hash armazenado.
Por exemplo, os sistemas Unix utilizam o algoritmo crypt() para armazenar as senhas no arquivo */etc/passwd*. O crypt() é uma variação do DES [2]: os 7 bits menos significativos de cada um dos caracteres da senha do usuário são utilizados para cifrar, repetidamente, uma constante (geralmente uma cadeia de bits zero). O resultado do último ciframento é transformado em 11 caracteres imprimíveis e armazenado no arquivo */etc/passwd* [24].
Em princípio, a busca exaustiva pode ser utilizada para burlar o sistema de controle de acesso. As 2^{56} possíveis senhas disponíveis nos sistemas Unix já não oferecem tanta segurança quanto ofereciam na época do desenvolvimento do DES. Organizações com grande poder de processamento já conseguiram quebrar o DES tradicional [14, 18]: o DES modificado pode ser o próximo.
Mesmo que um adversário não tenha muito poder de processamento é possível quebrar a segurança do controle de acesso utilizando ataques de dicionário. O ataque de dicionário consiste em armazenar em um arquivo (chamado de dicionário) as palavras mais prováveis de serem utilizadas como senhas: a própria palavra **senha**, nomes próprios, atletas, artistas, nomes de cidades, etc.

[2] O DES utilizado no crypt() sofreu modificações que visam dificultar a construção de *hardware* específico para uma busca exaustiva.

Com o dicionário pronto, basta usar o algoritmo crypt() no dicionário e compará-lo com o arquivo de senhas. Existem vários programas que automatizam este processo, como o Crack, CrackerJak e Qcrack[6].
Os perigos oferecidos pelos ataques de dicionários podem ser minimizados ensinando-se os usuários a escolherem boas senhas. Como nem sempre isso é possível, pode-se adicionar elementos aleatórios na senha do usuário para aumentar a segurança. Quando o usuário escolher uma senha (seja ele um usuário novo ou um que esteja mudando de senha) o sistema gera uma cadeia aleatória de 12 bits, chamada de **salt**. O salt é concatenado no início da senha antes do valor hash ser calculado e no início do valor hash gerado(Figura 3.6).

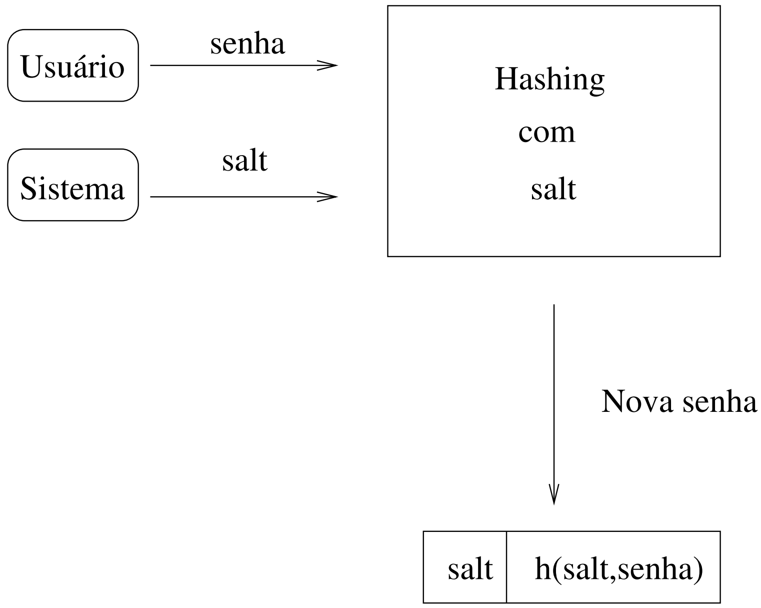

Figura 3.6: Armazenamento de senha com salt

A utilização do salt dificulta o ataque de dicionário, mas não resolve todos os problemas. Um salt de 12 bits, como o usado em alguns sistemas Unix, possui 4096 combinações, mas o aumento do tempo do ataque dependerá do objetivo do adversário:

- Para quebrar uma senha específica: o salt praticamente não altera o tempo do ataque,
- Para quebrar várias senhas: o aumento do tempo será proporcional ao número de usuários ou ao comprimento do salt, o que for menor.

3.3.2 Senhas Descartáveis

Quando o *login* é utilizado para controlar o acesso de serviços remotos, existe a possibilidade de uma falha de segurança: as senhas dos usuários são transmitidas pela rede, sem proteção alguma. Adversários utilizando *sniffers* podem facilmente capturar a senha e utilizá-la posteriormente, personificando um usuário válido do sistema. A utilização de senhas descartáveis (*one-time passwords*) inibem estes tipos de ataques.

Funcionamento

O primeiro passo do funcionamento de um sistema de senhas descartáveis é o cadastramento de uma senha por parte do usuário.
Esta senha, chamada de semente, passa por vários hashes sucessivos (usualmente 100 hashes). Um contador c indicando o número de hashes calculados e o último hash code são armazenados no sistema. O usuário precisa apenas guardar a semente (Figura 3.7).

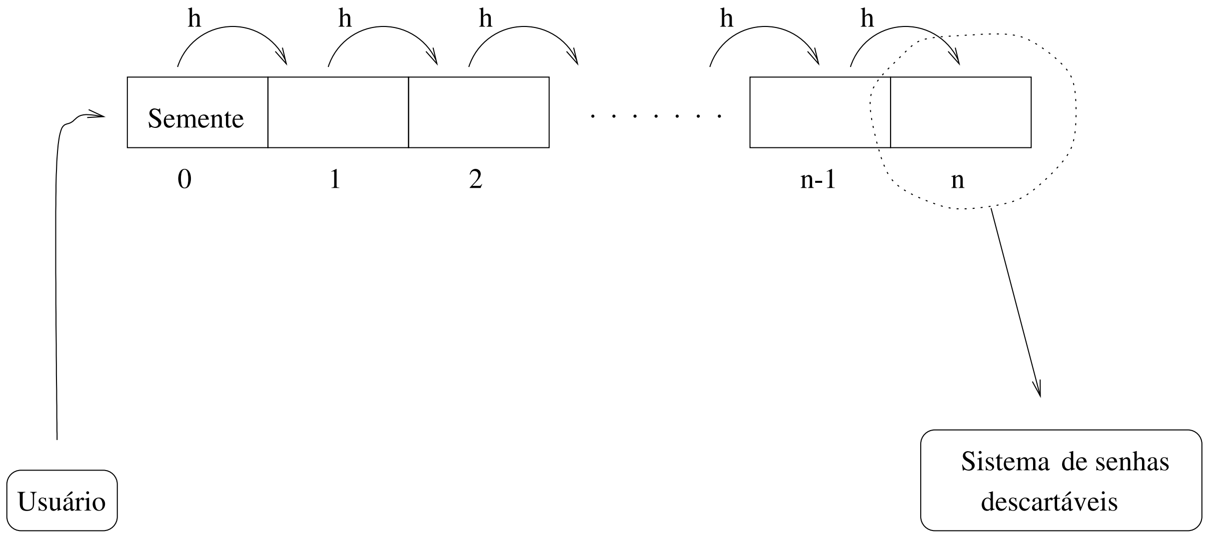

Figura 3.7: Iniciação do one-time password

Ao tentar entrar no sistema, o usuário recebe do servidor o valor c e calcula $c - 1$ hashes sucessivos sobre a semente e envia o valor hash ao servidor, que calcula o valor hash do valor recebido. Se o valor hash calculado pelo servidor for igual ao valor hash armazenado, é dado ao usuário acesso ao sistema, o contador c é diminuido em uma unidade e o valor hash armazenado é trocado pelo valor hash recebido (Figura 3.8).
Caso os valores hash sejam diferentes, o usuário não ganha acesso ao sistema.
Este processo é repetido cada vez que o usuário tenta entrar no sistema. Quando o valor de c chegar a 2, o usuário deve cadastrar uma nova senha no servidor.

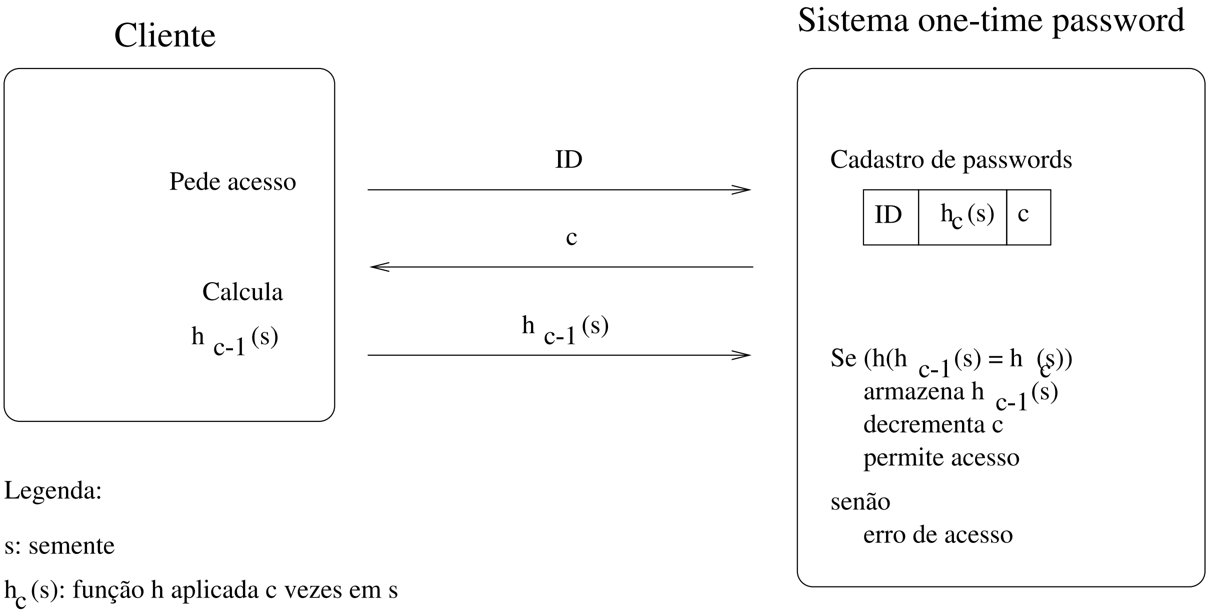

Figura 3.8: Verificação

3.3.3 Hash Cash

A Internet vem demonstrando ser um ótimo meio de propaganda. As empresas gastam muito tempo e dinheiro para disponibilizar gratuitamente informações aos seus consumidores, através de aplicativos que executam consultas complexas em grandes bases de dados. Estes tipos de serviços abrem uma brecha para ataques **DoS**(*Denial of Service*). Caso um adversário possa iniciar consultas com uma frequência maior do que a necessária ao sistema para resolvê-las, rapidamente o sistema não poderá responder a consultas dos verdadeiros clientes.
A cobrança de valores monetários poderia resolver o problema de DoS. Entretanto, nestes tipos de serviços a cobrança não pode ser feita, pois o objetivo da empresa é atrair os consumidores oferecendo informações gratuitas.
Uma outra solução é reduzir o número de consultas que um adversário pode executar em um período de tempo, obrigando-o a "gastar" um pouco do seu tempo antes de cada consulta.

1. **Funcionamento do HashCash**

   O HashCash é um sistema de controle de acesso ou de cobrança sem transferência de valores. As "moedas" do HashCash são colisões parciais de valores hash[7]. Para cada "compra" feita pelo consumidor, ele é obrigado a gastar um tempo de processamento na busca de uma colisão parcial de uma função de hashing.

O protocolo HashCash está descrito na Figura 3.9. O valor c, gerado aleatoriamente, é o valor hash que deve ser invertido; o valor t é o preço do serviço; o cliente deverá gerar uma colisão de t bits (conforme descrito na Figura 3.10). O valor de t pode variar de acordo com o tipo de serviço oferecido.

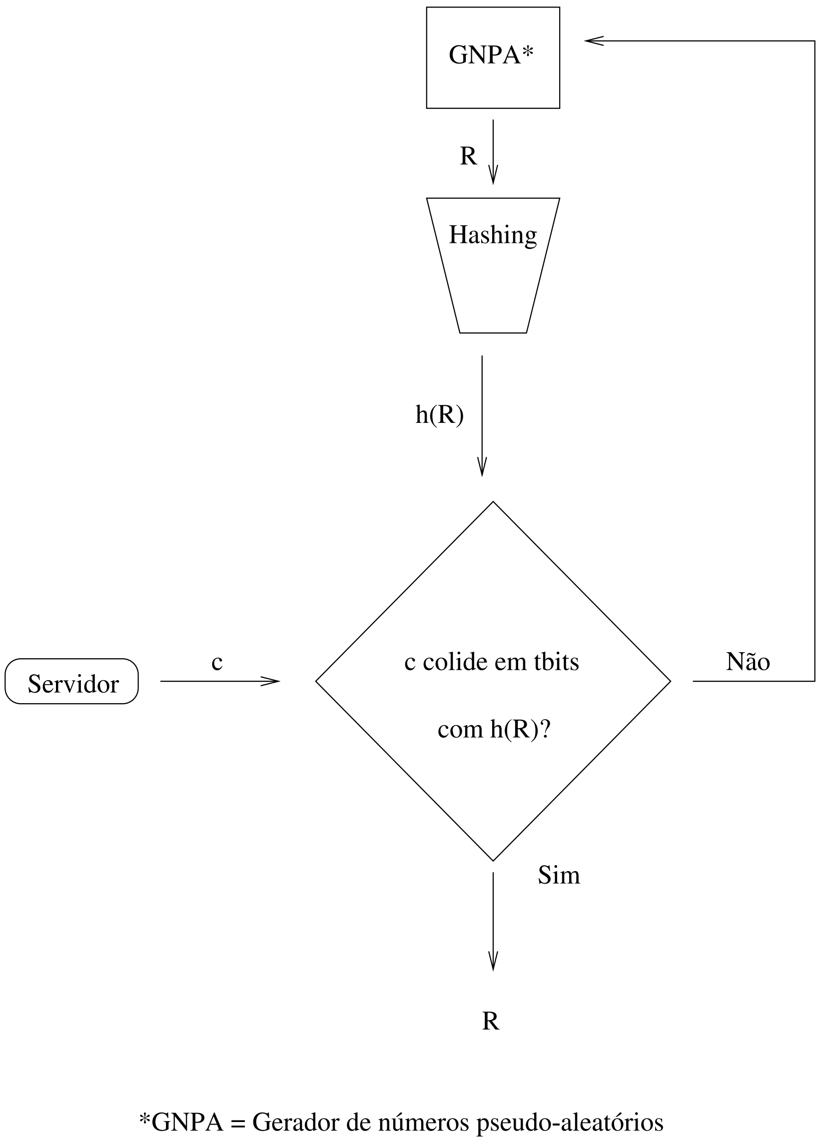

Figura 3.9: Geração da Colisão no Hashcash

O protocolo pode ser modificado para diminuir a carga do servidor e o número de mensagens trocadas. Ao invés de inverter um valor hash enviado pelo servidor, o cliente deverá buscar uma colisão parcial para o valor hash do seu pedido. Por exemplo, para pesquisar uma string s em um servidor, o cliente deverá enviar s e r tal que $h(r)$ colida com $h(s)$ em t bits.

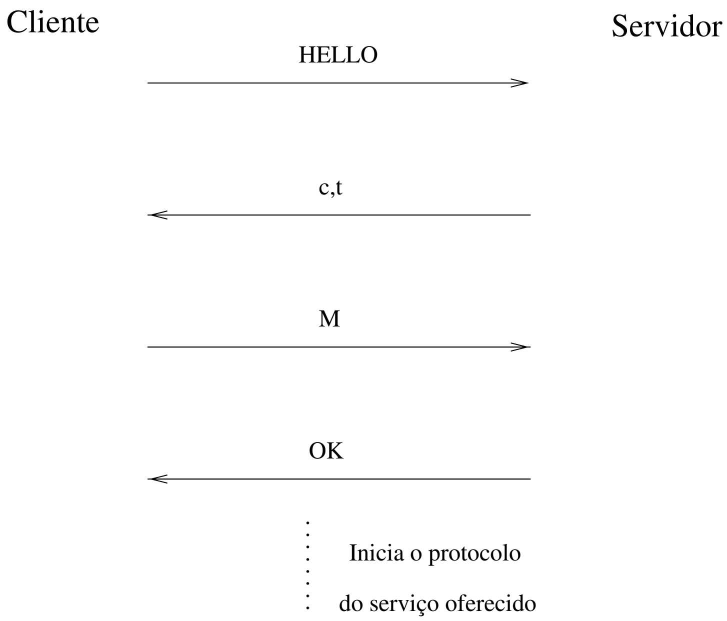

Figura 3.10: O protocolo Hashcash

2. **Implementação**

   O HashCash foi implementado em java e possui 3 classes:

   (a) public abstract class HashCash extends Object.
   Define a função de hashing utilizada e implementa a verificação da colisão.

   (b) public class HashCashServer extends HashCash.
   Implementa a geração do hash e faz a verificação da colisão.

   (c) public class HashCashClient extends HashCash.
   Implementa a busca da colisão parcial.

3. **Usos**

   O HashCash foi desenvolvido para policiar o uso de recursos gratuitos; portanto, pode ser usado na maioria das aplicações que atualmente são gratuitas. Um exemplo de uso foi implementado na *applet* HashCashApplet, uma aplicação cliente/servidor para fatorar números.

   Pode-se também usar o HashCash na prevenção de *mail spam*(envio de mail publicitário para várias pessoas), cobrando uma colisão de t bits para cada mail recebido. O problema do recebimento de mensagens de *mailing lists* [7] pode ser resolvido com a cobrança de valores diferentes para pessoas ou *hosts* diferentes, t igual a zero para colegas, parentes e servidores de *mailing lists* e t com valor grande para os demais *hosts*.

4. **Vantagens**

   O HashCash apresenta algumas vantagens quando comparado com outros sistemas:

   - Custo para o cliente: o único custo para o cliente é o tempo de busca da colisão.
   - Custo para o servidor: apenas a geração de um hash, utilizando um GNPA, e a verificação da colisão - uma compressão da função de hashing e comparação de t bits.
   - Transações Anônimas: o uso do HashCash não exige identificação do cliente.

5. **Ataques ao HashCash**

   Devido ao pequeno tamanho do valor hash utilizado, o HashCash está sujeito aos seguintes ataques:

   - Pré-computação: usando pré-computação, um adversário poderá solicitar serviços rapidamente, podendo causar DoS.
   - Ataque do aniversário (veja 6.4.2).

   Estes problemas podem ser solucionados limitando a liberdade na busca por colisão. Ao enviar os parâmetros do protocolo, o servidor envia também uma cadeia com p bits de comprimento; estes p bits devem estar no início da mensagem calculada pelo cliente, elevando a complexidade da pré-computação de 2^t para 2^{t+p}.

3.4 Confidencialidade

O método *Chaffing and Winnowing*, apresentado por Ron Rivest em [32], proporciona confidencialidade sem o uso direto de ciframento: a confidencialidade é feita com uma combinação de esteganografia e autenticação.

1. **Cenário**

   O cenário do *Chaffing and Winnowing* é semelhante ao do uso de um cifrador: duas entidades desejam trocar informações com privacidade utilizando um meio de comunicação inseguro. As entidades devem compartilhar uma chave secreta k.

2. **Funcionamento**

   A primeira etapa do método, chamada de *chaffing*, consiste em preparar a mensagem. Esta parte é equivalente ao ciframento e segue os seguintes passos:

   (a) Divisão da mensagem em pacotes.

(b) Geração de um código de autenticação (MAC) para cada pacote, utilizando uma chave secreta k, combinada anteriormente.

(c) Geração de pacote falsos (*chaffs*), pelo menos um para cada pacote verdadeiro, com códigos de autenticação inválidos.

Após a preparação da mensagem, ela é transmitida. Ao receber a mensagem, a outra entidade calculará o MAC de cada pacote e comporará o MAC calculado com o MAC recebido para selecionar os pacotes válidos. Esta última etapa, chamada de *winnowing*, é equivalente ao processo de deciframento.

Por exemplo, considere uma mensagem composta por 4 blocos:

(1,Olá, 456123)

(2,O jogo está marcado para,7895)

(3,18:00 horas,3442)

(4,Até mais,312312)

onde o primeiro campo representa o número de sequência do pacote, o segundo campo a mensagem do pacote e o terceiro o MAC. Após o *chaffing*, ficaria da seguinte forma:

(1,Olá, 456123)

(1,Boa tarde,34423)

(2,O negócio foi fechado, 7987900)

(2,O jogo está marcado para,7895)

(3,18:00 horas,3442)

(3,em 5 milhões,423543)

(4,Até mais,312312)

(4,Parabéns,55364)

O receptor da mensagem conhece a chave secreta, podendo então conferir os MACs, descartar os pacotes falsos (*winnowing*), guardar os pacotes verdadeiros e obter a mensagem original.

Note que não houve uso de ciframento; toda a mensagem foi enviada às claras. Sua confidencialidade foi protegida pelos pacotes falsos que não podem ser detectados sem o conhecimento da chave secreta.

3. **Segurança e Eficiência**

   Uma maneira de um adversário separar os pacotes verdadeiros dos pacotes falsos é analisar o conteúdo da mensagem, visto que não há criptografia. No exemplo dado anteriormente, analisando o conteúdo dos pacotes seria dificil fazer a separação, pois tanto os pacotes verdadeiros como os falsos possuem "significado". Entretanto, nas aplicações verdadeiras os pacotes falsos seriam gerados aleatoriamente, facilitando a separação pela análise do significado dos pacotes. Uma maneira de evitar este tipo de ataque é gerar pacotes pequenos (possivelmente de apenas um bit) que, isoladamente, não teriam significado. Considere, por exemplo, a mensagem:

   (1,0,4123423)

   (1,1,4324256)

   (2,0,6758654)

   (2,1,7456757)

   (3,0,5423526)

   (3,1,6456343)

   (4,0,6546333)

   (4,1,5803989)

   (5,0,5233452)

   (5,1,5698699)

   A análise do conteúdo dos pacotes não revela informação alguma ao adversário, e a busca exaustiva com a combinação dos pacotes geraria todas as 2^5 mensagens possíveis.

   A diminuição do tamanho do pacote aumenta a segurança, inviabilizando a análise dos conteúdos dos pacotes; entretanto, diminui a eficiência do sistema, pois para cada bit transmitido existe a transmissão dos bits de controle e do MAC. É possível aumentar a eficiência utilizando técnicas de tudo-ou-nada (*all-or-nothing*) ou transformações de pacotes (*package transform*) (transformações independentes de chave, que transformam uma mensagem em pacotes; o conteúdo de um pacote isolado não revela nenhuma informação sobre a mensagem original, que só poderá ser reconstruída utilizando todos os pacotes [31]).

   A utilização inadequada do método tudo-ou-nada pode diminuir a segurança. Caso o número de pacotes da mensagem seja pequeno (significativamente menor que o tamanho da chave), pode-se fazer busca exaustiva combinando os blocos e utilizando o processo inverso do tudo-ou-nada, na expectativa de uma mensagem com significado.

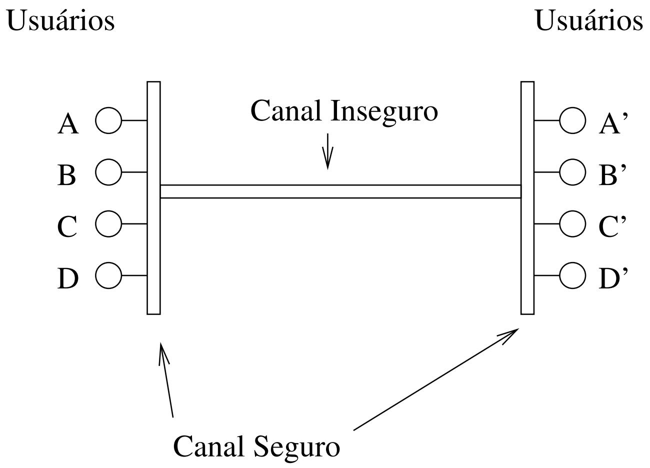

Figura 3.11: Aumento de eficiência para vários usuários

A eficiência do método *Chaffing and Winnowing* pode ser aumentada ainda mais. Imagine um canal inseguro (Internet ou linha telefônica) multiplexando as conexões entre dois canais seguros (rede local de uma empresa), onde vários pares de usuários utilizam o *Chaffing and Winnowing* (figura 3.11). Nesta situação, a criação dos pacotes falsos não é necessária, pois os pacotes dos outros usuários farão o papel dos pacotes falsos. O custo do *Chaffing and Winnowing* ficaria apenas na detecção dos pacotes verdadeiros.

4. **Questões Legais**

   Uma das principais vantagens do uso do *Chaffing and Winnowing* é que ele não usa criptografia e, portanto, não está sujeito a controles governamentais (restrições quanto à exportação, tamanho da chave ou *key escrow*). Entretanto, alguns especialistas em criptografia consideram este método um ciframento, pois a mensagem só pode ser recuperada com a utilização da chave secreta.

3.5 Assinaturas em fluxos

Aplicações que usam fluxos (*streams*) estão ficando mais populares. O uso de fluxos permite a transmissão de sons e imagens ao vivo por redes, como acontece na Internet. O fluxo possui duas características importantes:

- Tamanho: o fluxo pode ser indefinidamente grande, podendo ser considerado de tamanho *infinito* em alguns aplicações,
- Fracionável: o fluxo pode ser fracinoado e ainda ter significado. Um usuário pode ver apenas parte de um jornal que está sendo transmitido pela rede.

As soluções criptográficas tradicionais não conseguem prover segurança nas transmissões de fluxo:

- MAC: o uso do MAC poderia ser uma solução, já que fluxos podem ser quebrados em pedaços e o MAC aplicado nestes pedaços. Entretanto, o MAC não oferece não-repúdio, característica desejável em muitas aplicações,
- Assinaturas Digitais: assinando o valor hash de um fluxo obtém-se autenticidade e não-repúdio e um grande problema: o fluxo pode ser muito grande, algumas horas ou dias de sons ou imagens, e é desejável que a verificação da autenticidade e não-repúdio seja feita ao longo da transmissão e não somente ao seu fim. Uma outra saída é quebrar o fluxo e assinar os pedaços, como sugerido no MAC; entretanto, os algoritmos de assinaturas são muito lentos, e se as assinaturas forem feitas em pedaços pequenos, provavelmente o servidor que gera o fluxo não seria capaz de assinar tais pedaços. Se os pedaços forem grandes a autenticidade e o não-repúdio seriam obtidos com retardo.

A seguir serão descritos dois métodos, primeiramente publicados em [21], para assinar fluxos

3.5.1 Assinaturas em fluxos gravados

Supondo que o fluxo é inteiramente conhecido pelo transmissor antes do inicio da transmissão, um fluxo F, formado por k blocos $b_1 b_2 ... b_k$ é transformado em um fluxo $F' = b'_0 b'_1 b'_2 ... b'_k$ da seguinte forma:

1. $b'_k = (b_k, 000...0)$;
2. $b'_i = (b_i, h(b'_{i+1}))$, para $i = 1, ..., k-1$;
3. $b'_0 = (h(b'_1), S(pk, h(b'_1)))$.

onde $S(pk, h(b'_1))$ é a assinatura em b'_1 usando a chave pública pk.
O usuário que recebe o fluxo poderá verificar a integridade e autenticidade já no primeiro bloco do fluxo recebido, e a assinatura será "transmitida" aos demais blocos pelo encadeamento proporcionado pela função de hashing. O último bloco será detectado pela presença de uma cadeia formada somente por zeros.

3.5.2 Assinaturas em fluxo ao vivo

O processo anterior não pode ser usado para assinar fluxos ao vivo (*on-line*), já que o primeiro passo é o calculo do valor hash do último bloco, que é desconhecido quando se está fazendo uma transmissão ao vivo.
A solução utilizada é assinar cada bloco com assinaturas descartáveis, representadas por $s(pk_i, M)$, que são mais rápidas que as tradicionais, e em cada bloco adicionar a chave pública que será usada para verificar o próximo bloco. A construção do fluxo assinado segue os seguintes passos:

1. $b'_0 = pk_0, S(pk, pk_0)$
2. $b'_i = b_i, pk_i, s(pk_{i-1}, h(b_i, pk_i))$ para $i \geq 1$

O bloco b'_0 apenas contém a chave utilizada para verificar o bloco b'_1, e é assinado com a chave (não descartável) do gerador do fluxo. Os demais blocos contém o fluxo em si e uma chave que será utilizada no próximo bloco. Estes blocos são assinado com chaves descartáveis, recebidas no bloco imediatamente anterior.

3.6 Resumo

A maioria das aplicações apresentadas poderia ser construída sem o uso de hashing. Entretanto, devido à sua alta velocidade, as funções de hashing são utilizadas para substituir outras funções criptográficas.
Abaixo é apresentada uma tabela com um resumo das aplicações descritas:

Aplicação	Propriedade
PayWord	unidirecionalidade
MicroMint	resistência a colisões
GNPA	unidirecionalidade
Login	unidirecionalidade
One-time Password	unidirecionalidade
HashCash	resistência a colisões ou unidirecionalidade
Chaffing and Winnowing	MAC
Assinaturas em Fluxo	resistência a colisões

Tabela 3.1: Aplicações que usam hash

Capítulo 4

Construções típicas de funções de hashing

Na seção 1.3.3 foi apresentado um método para construir funções de hashing usando o método do quadrado modular. Esse método é muito lento. Existem funções mais rápidas e que podem ser utilizadas em situações práticas.
A seguir são apresentadas algumas construções, todas obedecendo ao meta-método de Merkle-Damgård, na seção 1.3.3. Os símbolos usados nas descrições abaixo são os mesmos usados na descrição do meta-método.

4.1 Baseadas em Cifradores de Blocos

Existem várias propostas de construção de funções de hashing utilizando cifradores de blocos. Este tipo de construção visa aproveitar *software* ou *hardware* já disponíveis, diminuindo os custos do desenvolvimento e uso das funções de hashing; por exemplo, em *smartcards*, a economia de alguns kbytes podem ser vitais.
Assim como nas funções de hashing iterativas, a entrada x é dividida em blocos $x_1 x_2 \ldots x_t$ de tamanho pré-especificado, e um valor inicial (IV), não secreto, é definido. O valor hash é calculado com chamadas sucessivas à função de ciframento, que é utilizada como função de compressão.
Entre as propostas existentes de funções de hashing utilizando cifradores de blocos estão:

1. **Matyas-Meyer-Osea**. O valor hash,H_t, é calculado da seguinte forma:

$$H_0 := IV; H_i := E_{g(H_{i-1})}(x_i) \oplus x_i, 1 \leq i \leq t,$$

   onde g é uma função que produz uma chave adequada ao cifrador E, sem outras propriedades criptográficas.

2. **Miyaguchi-Preenel**. O valor hash é calculado da seguinte forma:

$$H_0 := IV; H_i := E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}, 1 \leq i \leq t,$$

para g uma função definida como no método anterior.

3. **Davies-Meyer**. O valor hash H_t é calculado da seguinte forma:

$$H_0 := IV; H_i := E_{x_i}(H_{i-1}) \oplus H_{i-1}, 1 \leq i \leq t$$

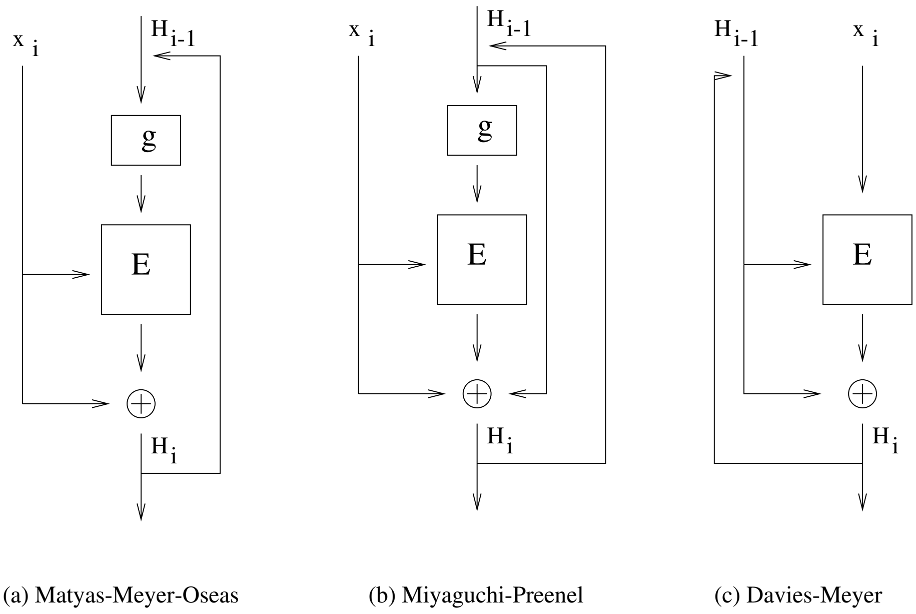

Figura 4.1: Construções de hashing usando cifradores

Embora sejam consideradas seguras contra criptoanálise, as construções apresentadas produzem um valor hash do mesmo tamanho do bloco utilizado pelo cifrador. Como a maioria dos cifradores de blocos atualmente em uso trabalha com blocos de 64 bits, estas funções de hashing produzirão valores hash de 64 bits, que são muito pequenos, principalmente quanto à resistência a colisões (veja o ataque do aniversário em 6.4.2). Para contornar o problema do tamanho do valor hash, foram propostas construções que produzem um tal valor hash com o dobro do tamanho do bloco:

- MDC-2: são definidas as constantes IV e $\overline{IV}$; o valor hash $H_t||\overline{H_t}$ é calculado da seguinte forma:

$$H_0 := IV; \;\; k_i := g(H_{i-1}); \;\; C_i := E_{k_i}(x_i) \oplus x_i; \;\; H_i := C_i^L || \overline{C_i}^R;$$

$$\overline{H_0} := \overline{IV}; \;\; \overline{k_i} := \overline{g}(\overline{H_{i-1}}); \;\; \overline{C_i} := E_{\overline{k_i}}(x_i) \oplus x_i; \;\; \overline{H_i} := \overline{C_i}^L || C_i^R;$$

- MDC-4: o hash code $G_t || \bar{G}_t$ é calculado da seguinte forma:

$$G_0 := IV; \;\; \overline{G}_0 := IV';$$

$$k_i := g(G_{i-1}); \;\; C_i := E_{k_i}(x_i) \oplus x_i; H_i := C_i^L || \overline{C_i}^R;$$

$$\overline{k_i} := \overline{g}(\overline{G}_{i-1}); \;\; \overline{C_i} := E_{\overline{k_i}}(x_i) \oplus x_i; \overline{H_i} := \overline{C_i}^L || \overline{C_i}^R;$$

$$j_i := g(H_i); \;\; D_i := E_{j_i}(\overline{G_{i-1}}) \oplus \overline{G_{i-1}}; \;\; G_i := D_i^L || \overline{D}_i^R;$$

$$\overline{j}_i := \overline{g}(\overline{H}_i); \;\; \overline{D}_i := E_{\overline{j_i}}(G_{i-1}) \oplus G_{i-1}; \overline{G}_i := \overline{D}_i^L || D_i^R;$$

Onde C_i^L são os bits mais a esquerda de C_i e C_i^R são os bits mais a direita.
Estas duas construções são consideradas bastantes seguras. Quando o DES é utilizado, o valor hash produzido tem comprimento de 128 bits; entretanto, elas são muitos lentas: o MDC-2 utiliza dois ciframentos para processar 64 bits e o MDC-4 utiliza 4 ciframentos.
Além dos problemas do tamanho do valor hash e da velocidade, alguns cifradores, mesmo seguros, possuem algumas falhas que podem ser exploradas quando utilizados para construir funções de hashing (veja 6.6).

4.2 MD4

O algoritmo MD4 [29, 36] foi desenvolvido por Ron Rivest. Durante o desenvolvimento do MD4, seu autor objetivou as seguintes características:

- Resistência a colisões: é computacionalmente inviável produzir 2 mensagens com o mesmo valor hash. O melhor ataque contra o MD4 é a força bruta;
- Segurança direta: a segurança do MD4 é baseada na complexidade do algoritmo;
- Velocidade: o MD4 utiliza manipulação de bits em operandos de 32 bits;
- Simplicidade: não são utilizadas estruturas de dados complexas no MD4;

- Favorecimento de arquiteturas *little-endian*: o MD4 é otimizado para arquiteturas Intel.

O valor hash produzido pelo MD4 tem o comprimento de 128 bits, bem maior que o valor hash produzido pelas funções de hashing atuais que utilizam cifradores de bloco.

4.2.1 Algoritmo MD4

O MD4 divide a entrada x em blocos x_i de 512 bits. Cada chamada à função de compressão do MD4 processa 512 bits da entrada e 128 bits da variável de encadeamento, produzindo uma saída de 128 bits.
A função de compressão é dividida dem 3 iterações, cada uma possuindo uma função interna distinta.

1. **Definições**

   $f_0(a, b, c) := a \wedge b \vee \bar{a} \wedge c;$

   $f_1(a, b, c) := a \wedge b \vee a \wedge c \vee b \wedge c;$

   $f_2(a, b, c) := a \oplus b \oplus c;$

   $x_i[j], 0 \leq j \leq 15$, é uma subdivisão de x_i com 32 bits.

2. **Inicialização**[1]

   (a) Inicialização das variáveis de encadeamento: h_1, h_2, h_3 e h_4;
   (b) Inicialização do vetor $y[0..2]$;
   (c) Inicialização da matriz $z[0..2, 0..15]$ que define a ordem de acesso a palavras do bloco;
   (d) Inicialização da matriz $s[0..2, 0..16]$ que define o número de bits das rotações;
   (e) Preenchimento da entrada (veja 6.7).

3. **Processamento**

   Para cada bloco x_i faça
   $\quad (a, b, c, d) := (h_1, h_2, h_3, h_4)$
   $\quad$ Para k de 0 a 2 faça
   $\quad\quad$ Para j de 0 a 15 faça
   $\quad\quad\quad t := (a + f_k(b, c, d) + x_i[z[k, j]] + y[k])$
   $\quad\quad\quad (a, b, c, d) := (d, rol(t, s[k, j]), b, c)$
   $\quad (h_1, h_2, h_3, h_4) := (h_1 + a, h_2 + b, h_3 + c, h_4 + d)$

[1]Os valores utilizados para inicialização estão definidos em [29]

4. **Finalização**
   $h(x)$ é igual à concatenação de h_1, h_2, h_3, h_4.

4.2.2 Ataques

Ataque ao MD4 reduzido

O ataque ao MD4 [12] reproduzido a seguir, tem como alvo uma versão modificada do MD4, a saber, o MD4 com apenas as duas últimas iterações. Um ataque a uma função criptográfica modificada não tem efeito prático ou imediato: um adversário que tenha condições de modificar uma função criptográfica não precisará procurar fraquezas nas funções, ele poderá criá-las. Entretanto, um tal ataque mostra falhas no projeto da função, alertando para possíveis ataques futuros ou até mesmo para existência de atalhos secretos (*trapdoors*).
A restrição às duas últimas iterações do MD4 mostra um padrão de acesso às palavras: os primeiros quatro passos de ambas iterações acessam o mesmo subconjunto de palavras, os próximos oito passos acessam um outro subconjunto e os quatro últimos passos acessam um terceiro subconjunto (veja Tabela 4.1), fato que possibilitará a geração de segundas pré-imagens e colisões. Considerando este padrão de acesso e duas mensagens com as palavras 0,3,4,7,8,11,12 e 15 iguais e com as demais palavras diferentes, os seguintes fatos são obtidos:

- As variáveis de encadeamento utilizadas no cálculo do valor hash de ambas mensagens serão iguais nos primeiros quatro passos;
- Se o valor das variáveis forem iguais após o passo 12, que usa a palavra 14 do bloco de dados, elas permanecerão iguais até o final da primeira iteração e nos quatro passos iniciais da segunda;
- Se no passo 29 as variáveis forem iguais, elas continuarão iguais até o fim, gerando uma colisão.

O comportamento da diferença entre as variáveis de encadeamento está descrito na Figura 4.2. As linhas congruentes indicam que as variáveis de encadeamento possuem o mesmo valor e as linhas em paralelo indicam valores diferentes.
O processamento de duas mensagens com estas características permite a criação de um sistema de equações composto pelas transformações que acessam as palavras diferentes das mensagens (subconjunto 2).
Resolvido este sistema de equações, descobre-se que a diferença entre as mensagens que produzem colisão é igual a:

0	$-2N$	$2N$	0	0	$-2N$	$2N$	0	0	$-N$	N	0	0	$-N$	N	0

onde cada bloco acima é uma palavra de 32 bits e $N = (55555555)_{16}$.

	iteração 2	iteração 3
	0	0
Subconjunto	4	8
1	8	4
	12	12
	1	2
	5	10
	9	6
Subconjunto	13	14
2	2	1
	6	9
	10	5
	14	13
	3	3
Subconjunto	7	11
3	11	7
	15	15

Tabela 4.1: Ordem de acesso nas iterações 2 e 3 do MD4

Criptoanálise completa do MD4

A criptoanálise do MD4 completo foi feita por Hans Dobbertin em [16]. Este ataque combina técnicas aplicadas no ataque ao RIPEMD-128 [15] com técnicas de criptoanálise diferencial, para gerar colisões.
O ponto forte deste ataque é que ele produz uma forja seletiva: apesar das mensagens diferirem em apenas um bit, o ataque permite compor uma mensagem com bastante liberdade e criar mensagens com significado.

Ataque à unidirecionalidade do MD4

Hans Dobbertin também mostrou em [17] que o MD4 sem a última iteração não é unidirecional.

4.3 MD5

Também desenvolvido por Ron Rivest, o MD5 [30] é uma versão melhorada do MD4. A maioria das alterações feitas na criação do MD5 são respostas ao ataque nas duas últimas iterações do MD4 [12]:

- Adição de uma iteração;

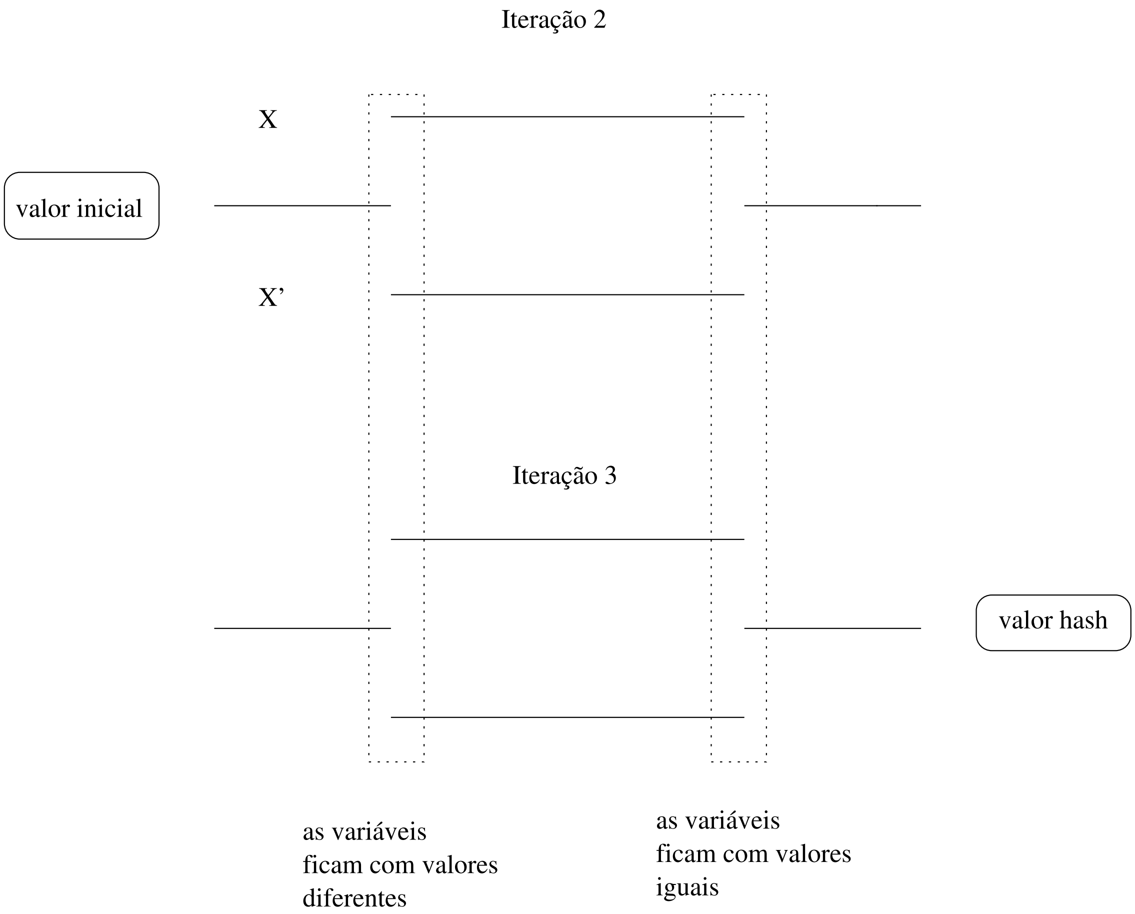

Figura 4.2: Comparação dos IV's na busca da colisão

- mudança na ordem de acesso das palavras;
- mudança no número de bits rotacionados;
- mudança da função utilizada na segunda iteração;
- cada passo possui uma constante de adição (no MD4 é utilizada uma constante para cada iteração);
- resultado de cada passo é adicionado ao próximo.

4.3.1 Algoritmo MD5

O algoritmo MD5 é bem parecido com o algoritmo do MD4, as diferenças ficam por conta das constantes e por uma iteração a mais no MD5.

1. **Definições**

   $f_0(a,b,c) := a \wedge b \vee \bar{a} \wedge c$

   $f_1(a,b,c) := a \wedge c \vee b \wedge \bar{c}$

   $f_2(a,b,c) := a \oplus b \oplus c$

   $f_3(a,b,c) := b \oplus (a \vee \bar{c})$

   $x_i[j], 0 \leq j \leq 15$, é uma subdivisão de x_i com 32 bits

2. **Inicialização**

   (a) Inicialização das variáveis de encadeamento: h_1, h_2, h_3 e h_4;

   (b) Inicialização da matriz $y[0..3, 0..15]$;

   (c) Inicialização da matriz $z[0..3, 0..15]$ que define a ordem de acesso a palavras do bloco;

   (d) Inicialização da matriz $s[0..3, 0..16]$ que define a quantidade de bits das rotações;

   (e) Preenchimento da entrada (veja em 6.7).

3. **Processamento**

```
Para cada bloco x_i faça
    (a, b, c, d) := (h_1, h_2, h_3, h_4)
    Para k de 0 a 3 faça
        Para j de 0 a 15 faça
            t := (a + f_k(b, c, d) + x_i[z[k, j]] + y[k])
            (a, b, c, d) := (d, b + rol(t, s[k, j]), b, c)
    (h_1, h_2, h_3, h_4) := (h_1 + a, h_2 + b, h_3 + c, h_4 + d)
```

4. **Finalização**

   $h(X)$ é igual à concatenação de h_1, h_2, h_3, h_4.

4.3.2 Ataque de pseudo-colisões

Dada uma função de compressão $g(IV, x)$ onde IV representa a variável de encadeamento da função de hashing e x um bloco de dados, uma pseudo-colisão é o conjunto formado por IV, IV' (com $IV \neq IV'$), x e x' (os valores x e x' podem ser iguais) que satisfazem a equação:

$$g(IV, x) = g(IV', x').$$

O primeiro ataque ao MD5 que foi publicado [13] descreve um algoritmo capaz de encontrar pseudo-colisões para função de compressão do MD5 em poucos minutos.
A idéia principal do ataque é produzir um IV de forma que o complemento do bit mais significativo (MSB) de cada uma das suas palavras, representadas por (A, B, C, D), não altere a saída da função de compressão g:

$$g((A, B, C, D) \oplus (2^{31}, 2^{31}, 2^{31}, 2^{31}), x) = g((A, B, C, D), x). \tag{1}$$

Se substituirmos as 64 iterações internas de g (veja o algoritmo na página 44) por uma função i, teremos:

$$g(A, B, C, D) = i(A, B, C, D) + (A, B, C, D). \tag{2}$$

Agora, considerando que $(A + 2^{31}) \pmod{2^{32}} = A \oplus 2^{31}$ e substituindo g em (1) por (2) temos:

$$i((A, B, C, D) \oplus (2^{31}, 2^{31}, 2^{31}, 2^{31}), x) =$$
$$\text{i((A,B,C,D),x)} \oplus (2^{31}, 2^{31}, 2^{31}, 2^{31}). \tag{3}$$

Observando um passo do MD5:

$$A = B + (rol((\underbrace{A + f_i(B, C, D)}_{\textit{(a)}} + x[j] + t), s)),$$

podemos concluir que se a inversão do MSB de B, C e D provocar a inversão do MSB de f_i, esta será anulada pela inversão do MSB de A (detalhe *(a)*), propagando a inversão do MSB de B, fazendo com que (3) seja válido.
Para que o ataque seja concretizado, resta escolher valores adequados para serem usados como entrada nas f_i's:

- Para f_0: (0,0,0),(1,0,0) e seus complementos (1,1,1), (0,1,1) propagam a inversão do MSB, mas a tripla (1,0,0) provocará uma entrada no próximo passo com o valor (0 ou 1,1,0) que não pode ser aceito. O mesmo ocorre com o seu complemento (0,1,1), restando então as triplas (1,1,1) e (0,0,0) para serem usadas. Por razões de simplicidade, o algoritmo de colisões irá trabalhar somente com a tripla (1,1,1).

- Para f_1: (0,0,0),(0,0,1) e seus complementos (1,1,1), (1,1,0). Para f_1 são feitas as mesmas considerações que foram feitas para f_0, restando então a tripla (1,1,1).

- Para f_2: qualquer valor é aceito.

- Para f_3: (0,0,0),(0,1,0) e seus complementos (1,1,1), (1,0,1) são aceitos. Estas quatro combinações representam metade das 2^3 possíveis; assim após os 16 passos desta iteração, a chance de todas as triplas terem sido adequadas ao ataque é de 2^{-16}.

O algoritmo utilizado para gerar pseudo-colisões irá gerar um valor inicial e uma entrada para que as restrições feitas para f_0 e f_1 sejam cumpridas; f_2 não tem restrições e o valor correto para f_3 é achado por tentativa e erro.
Uma vez que os valores tenham sido encontrados, a pseudo-colisão é feita complementando o bit mais significativo de cada uma das palavras do valor inicial gerado. Segue uma descrição simplificada do algoritmo:

1. (A, B, C, D) são inicializados com valores aleatórios;
2. o algoritmo começa na segunda iteração, inicializando o bloco de dados x para que ele gere os valores adequados para o ataque (detalhe a na Figura 4.3);
3. a primeira iteração é percorrida pelo caminho inverso testando os valores do bloco de dados (detalhe b). Caso ocorra algum erro, a palavra do bloco de dados é alterada e testada nas duas iterações (detalhe c);
4. (A, B, C, D) são alterados pela iteração 3 (detalhe d).
5. os dados x e a variável de encadeamento (A, B, C, D) são testados pela iteração 4 (detalhe e), com chance de 2^{-16} de serem adequados; caso sejam, a colisão é: $((A, B, C, D), x)$ e $((A \oplus 2^{31}, B \oplus 2^{31}, C \oplus 2^{31}, D \oplus 2^{31}), x)$.

Como o algoritmo não é determinístico (depende de valores aleatórios), o tempo esperado é bastante variável: de alguns segundos a 30 minutos.

4.3.3 Colisões

A criptoanálise do MD5 também foi desenvolvida por Hans Dobertin. Este ataque utiliza as mesmas técnicas do ataque ao MD4 e produz 2 mensagens com pequenas diferenças; porém, este ataque produz uma forja existencial (veja 6.2).

4.3.4 Condições atuais de segurança do MD5

Devidos a estes ataques, a RSA publicou em [34] as seguintes considerações sobre o MD5:

- Assinaturas existentes que utilizam o MD5 não estão sob risco; os ataques existentes não ajudam a calcular uma segunda pré-imagem.

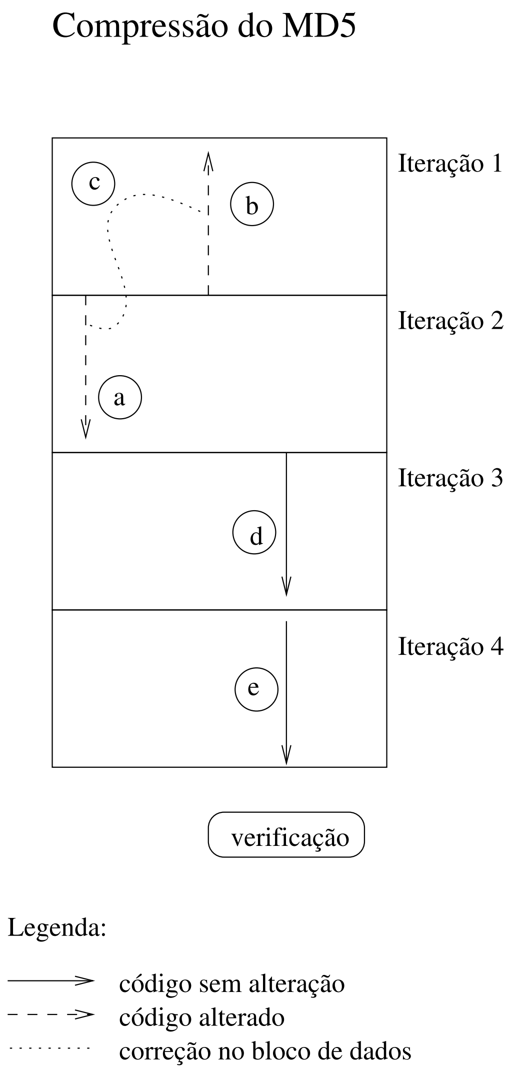

Figura 4.3: Algoritmo de pseudo-colisões

- O MD5 ainda pode ser usado para produzir saídas de aparência aleatória e como função unidirecional.
- O MD5 ainda pode ser utilizado para produzir MAC's, como por exemplo HMAC e MD5-MAC.

4.4 SHA

O NIST(*National Institute of Standards and Technology*)/NSA(*National Security Agency*) desenvolveram o SHA [1] para ser usado no DSS (*Digital Signature Standard*).

Devido a problemas de segurança não revelados, o algoritmo foi modificado dando origem ao SHA-1 [3]. Somente recentemente um ataque, descrito em 4.4.2, ao SHA foi desenvolvido, expondo o motivo da mudança.

4.4.1 Algoritmo SHA-1

A função de compressão do SHA-1 expande o bloco de entrada de 512 bits em um *buffer* de 2560 bits (no algoritmo representado por um vetor w). Após a expansão, as palavras de 32 bits do *buffer* são acessadas em ordem crescente. Diferentemente do MD4 e MD5 o SHA-1 utiliza notação *big-endian*, que modifica a função de preenchimento.

1. **Definições**

   $f_0(a,b,c) := a \wedge b \vee \bar{a} \wedge c$

   $f_1(a,b,c) := a \oplus b \oplus c$

   $f_2(a,b,c) := a \wedge b \vee a \wedge a \vee b \wedge c$

   $f_3(a,b,c) := a \oplus b \oplus c$

   $x_i[j], 0 \leq j \leq 15$, é uma subdivisão de x_i com 32 bits

2. **Inicialização**

   (a) Inicialização das constantes h_1, h_2, h_3, h_4 e h_5;

   (b) Inicialização do vetor $y[0..3]$ que define a ordem de acesso à palavras do bloco;

   (c) Criação do vetor $w[0..79]$;

   (d) Inicialização do vetor $c[0..3]$;

   (e) Preenchimento da entrada (veja em 6.7).

3. **Processamento**

```
Para cada bloco x_i faça
  (a, b, c, d, e) := (h_1, h_2, h_3, h_4, h_5)
  Para j de 0 a 15
    w[j] := x_i[j]
  Para j de 16 a 79
    w[j] := rol((w[j − 3] ⊕ w[j − 8] ⊕ w[j − 14] ⊕ w[j − 16]), 1)
  Para k de 0 a 3 faça
    Para j de 0 a 19 faça
      t := rol(a, 5) + f_k(b, c, d) + e + w[k ∗ 20 + j] + c[k]
      (a, b, c, d, e) := (t, a, rol(b, 30), c, d)
  (h_1, h_2, h_3, h_4, h_5) := (h_1 + a, h_2 + b, h_3 + c, h_4 + d, h_5 + e)
```

4. **Finalização**

   $h(x)$ é igual à concatenação de h_1, h_2, h_3, h_4, h_5.

4.4.2 Ataque ao SHA

Este ataque ao SHA é considerado a explicação para o surgimento do SHA-1, visto que ele é ineficiente quando usado contra o SHA-1. A diferença entre o SHA e o SHA-1 é somente a expansão dos blocos de dados: a linha

$$w[j] := rol((w[j-3] \oplus w[j-8] \oplus w[j-14] \oplus w[j-16]), 1)$$

presente no SHA-1 deve ser trocada por

$$w[j] := w[j-3] \oplus w[j-8] \oplus w[j-14] \oplus w[j-16]$$

para obtermos o algoritmo do SHA.
Primeiramente, uma versão mais fraca do SHA é criada e atacada; posteriormente, são feitas pequenas modificações no SHA enfraquecido para torná-lo mais "forte" e esta versão também é atacada. Este processo de "fortalecimento" e ataque é repetido até que o SHA seja completamente criptoanalisado.
A primeira versão enfraquecida, que será chamada de SHAv1 possui as seguintes diferenças em relação ao SHA:

- As funções internas são sempre ou-exclusivos:

  $$f_j(a, b, c) = a \oplus b \oplus c,$$

  para qualquer j;

- As adições presentes nas iterações são substituídas por ou-exclusivos:

  $$t := rol(a, 5) \oplus f_k(b, c, d) \oplus e \oplus w[k * 20 + j] \oplus c[k].$$

Observando o algoritmo podemos concluir que a alteração de 1 bit no vetor w pode ser corrigida alterando-se 5 outros bits, que serão usados nas 5 iterações seguintes. Assim, a colisão pode ser gerada escolhendo-se uma mensagem qualquer e fazendo-se 5 correções para cada modificação inserida. Este processo pode ser executado rapidamente criando máscaras com as alterações e correções.
Uma segunda versão do SHA, que será chamada de SHAv2, é criada mantendo as funções internas originais e alterando as adições nas iterações:

$$t := rol(a, 5) \oplus f_k(b, c, d) \oplus e \oplus w[k * 20 + j] \oplus c[k].$$

É fácil perceber que em alguns casos as funções internas do SHAv2 se comportam como as do SHAv1.
A última versão enfraquecida do SHA, SHAv3, é criada mudando-se as funções internas para ou-exclusivos. Novamente, em alguns casos, a versão enfraquecida se comportará como o SHAv1.
O SHAv1 resulta de duas modificações do SHA; o SHAv2 é criado a partir do SHA usando uma destas modificações; finalmente, o SHAv3 é criado usando a outra. Uma colisão gerada para o SHAv1 será uma colisão no SHA se ela for colisão no SHAv2 e SHAv3, então, a chance de uma colisão no SHAv1 ser também no SHA pode ser calculada combinando-se as chances das colisões no SHAv2 e SHAv3: fazendo os cálculos chegamos que uma colisão no SHAv1 será também no SHA com chances de 1 em 2^{71}, melhor que o ataque do aniversário (veja 6.4.2) para uma função de 160 bits.

4.5 RIPEMD

Desenvolvidas pelo projeto RACE (*Research and Development in Advanced Communication Technologies in Europe*), as funções de hashing da família RIPEMD têm dois tamanhos de hash: 128 e 160 bits. Como o RIPEMD-128 já foi criptoanalisado em [15] e seu sucessor foi o RIPEMD-160, nos só apresentaremos o RIPEMD-160.

4.5.1 Algoritmo RIPEMD-160

A função RIPEMD-160 é bem parecida com o MD4 e com o MD5, com a diferença que o RIPEMD-160 possui 2 funções de compressão, que se diferenciam pelas constantes utilizadas. A compressão é feita em paralelo pelas funções de compressão. No final os valores das variáveis de encadeamento das duas funções são unificados.

1. **Definições**

   $f_0(a, b, c) := a \oplus b \oplus c$

   $f_1(a, b, c) := a \wedge b \vee \bar{a} \wedge c$

   $f_2(a, b, c) := (a \vee \bar{b}) \oplus c$

   $f_3(a, b, c) := a \wedge c \vee b \wedge \bar{c}$

   $f_4(a, b, c) := a \oplus (b \vee \bar{c})$

   $x_i[j], 0 \leq j \leq 15$, é uma subdivisão de x_i com 32 bits

2. **Inicialização**

   (a) Inicialização das constantes h_1, h_2, h_3, h_4 e h_5;

   (b) Inicialização das matrizes $zr[0..4, 0..15]$ e $zl[0..3, 0..15]$ que definem a ordem de acesso a palavras do bloco;

   (c) Inicialização das matrizes $sr[0..4, 0..15]$ e $sl[0..3, 0..15]$ que definem a quantidade de *bits* na rotação;

   (d) Inicialização dos vetores $yr[0..4]$ e $yl[0..4]$;

   (e) Preenchimento da entrada da entrada (veja em 6.7).

3. **Processamento**

Para cada bloco x_i de 512 bits da entrada faça
  $(a_r, b_r, c_r, d_r, e_r) := (h_1, h_2, h_3, h_4, h_5)$
  $(a_r, b_r, c_r, d_r, e_r) := (h_1, h_2, h_3, h_4, h_5)$
  Para k de 0 a 4 faça
    Para j de 0 a 15 faça
      $t := (a_l + f_k(b_l, c_l, d_l) + x_i[zl[k, j]] + yl[k]$
      $(a_l, b_l, c_l, d_l, e_l) := (e_l, a_l + rol(t, sl[k]), b_l, rol(c_l, 10), d_l)$
      $t := (a_r + f_k(b_r, c_r, d_r) + x_i[zr[k, j]] + yr[k]$
      $(a_r, b_r, c_r, d_r, e_r) := (e_r, a_r + rol(t, sr[k]), b_r, rol(c_r, 10), d_r)$
  $t := h_1$
  $h_1 := h_2 + c_l + d_r$
  $h_2 := h_3 + d_l + e_r$
  $h_3 := h_4 + e_l + a_r$
  $h_4 := h_5 + a_l + b_r$
  $h_5 := t + b_l + c_r$

4. **Finalização**

   $h(X)$ é concatenação de h_1, h_2, h_3, h_4, h_5.

Capítulo 5

Construções típicas de funções de MACs

Quanto ao modo de construção, as funções MAC são divididas nos seguintes grupos:

- Customizadas: são funções construídas especificamente para serem usadas como funções MAC.
- Baseadas em cifradores de blocos: utilizam cifradores já desenvolvidos.
- Baseadas em funções de hashing: utilizam funções de hashing já desenvolvidas.
- Adaptação do código das funções de hashing: através de pequenas alterações no código das funções de hashing consegue-se gerar um MAC. As alterações são simples e aplicáveis a qualquer função de hashing.

5.1 Baseadas em cifradores de blocos

Assim como as funções de hashing, as funções de MAC também podem ser construídas utilizando cifradores de blocos. Uma construção comum é o uso do cifrador em modo CBC com a chave MAC.
As funções de MAC baseadas em cifradores de blocos são relativamente mais seguras que as de hashing baseadas em cifradores de blocos, pois a chave secreta inviabiliza alguns tipos de ataques, e o tamanho do bloco (geralmente 64 bits) é grande o suficiente para ser usado como MAC.

5.2 Baseadas em funções de hashing

As funções de MAC podem ser geradas a partir de funções de hashing, aproveitando o hardware ou software já desenvolvidos. Na extensão de uma função de hashing para um MAC é necessário possibilitar a entrada de um segundo argumento, a chave secreta. Métodos para esse fim são descritos a seguir.

5.2.1 Método do IV Secreto

Este método consiste em substituir o IV da função de hashing pela chave MAC (ou por uma função da chave). Como na função de hashing cada compressão é dependente do IV, cada compressão da função de MAC também será dependente da chave.
Devido à propriedade de encadeamento (veja 1.3.3), este tipo de construção permite que MACs válidos sejam gerados concatenando cadeias de bits ao final de mensagem original, pois:

$$h^k(x||y) = h^{h^k(x)}(y),$$

onde $h^k(x)$ é o valor hash de x usando a função h com o IV igual a k. Mesmo o uso do fortalecimento-MD (veja 6.7) não impede este tipo de ataque [4].

5.2.2 Método do Prefixo Secreto

O método do prefixo secreto consiste em prefixar a entrada de uma função de hashing com a chave MAC. Isto é:

$$h_k(x) = h(k||x).$$

Assim como as funções de IV secreto, as funções com prefixo secreto também podem ser fraudadas com a concatenação de bits.

5.2.3 Método do Sufixo Secreto

O método do sufixo secreto consiste em concatenar ao final da entrada da função de hashing a chave MAC. Isto é:

$$h_k(x) = h(x||k).$$

Um adversário pode atacar este método gerando uma mensagem x' tal que $h(x) = h(x')$ (usando o ataque do aniversário). Como existe uma colisão entre os valores hash das mensagens, também ocorrerá uma colisão entre os seus MAC's, pois eles são dependentes do valor hash e da chave MAC.

Método do Envelope

O método do envelope consiste em concatenar a chave MAC ao início e ao fim da mensagem:

$$h_k(x) = h(k||x||k).$$

Existem variações deste método em que a chave concatenada ao início da mensagem é diferente da chave concatenada ao fim. Uma delas é o método proposto em [26]:

$$h_k(x) = h(k||p||x||k),$$

onde p é uma cadeia de bits, tal que o tamanho de $k||p$ tem o mesmo tamanho da entrada da função de compressão de h. Este método já foi quebrado em [28]; para uma chave secreta de 128 bits o ataque necessita de aproximadamente 2^{66} operações de hashing.

5.3 MAA

O MAA (*Message Authenticator Algorithm*) é uma função de MAC customizada desenvolvida em 1983 [11]. Ela executa operações simples (adição, rotação, ou-exclusivo, ou-lógico e e-lógico) em palavras de 32 bits. O comprimento da chave MAC e da variável de encadeamento é de 64 bits e o comprimento do MAC é de 32 bits.

5.3.1 Algoritmo MAA

1. **Inicialização**

   Inicialização das constantes A,B,C,D e W.

   Inicialização das variáveis e e v como uma função da chave, independente da mensagem.

2. **Processamento**

   Para cada bloco x_i da mensagem x

   $v := rol(v, 1);$

   $e := v \oplus W;$

   $x := ((((e + y) \bmod 2^{32}) \vee A \wedge C) * (x \oplus x_i)) \bmod 2^{32} - 1;$

   $y := ((((e + x) \bmod 2^{32}) \vee B \wedge D) * (y \oplus x_i)) \bmod 2^{32} - 2;$

   $h_k(x) := x \oplus y.$

5.4 HMAC

Desenvolvido por Mihir Bellare, Ran Canetti e Hugo Krawczyk , o HMAC [8] transforma uma função de hashing em uma função de MAC, usando a função de hashing como uma caixa-preta, sem a necessidade de alterar seu código. O HMAC é adotado em vários protocolos (como o IPSEC, s-http e SSL) usando as funções MD5 e SHA.

5.4.1 Algoritmo HMAC

Seja h uma função de hashing.
É escolhida uma chave secreta k com o comprimento igual ao comprimento do bloco da função de compressão de h, caso o comprimento de k seja menor, bits com valor zero deverão ser acrescentados ao final de k até o seu tamanho ser igual.
É definido *ipad* como uma concatenação de *bytes* de valor $(36)_{16}$ cujo comprimento total é igual ao comprimento de k.
É definido *opad* como uma concatenação de *bytes* de valor $(5C)_{16}$ cujo comprimento total é igual ao comprimento de k.
Finalmente, $HMAC_k(x)$ é definida como $h(k \oplus opad || h(k \oplus ipad || x))$

5.5 MDx-MAC

Desenvolvido por Bart Preneel e Paul C. van Oorschot. O MDx-MAC transforma uma função de hashing iterativa (baseada no MD4) em uma função de MAC com chave de 128 bits. O MDx-MAC altera o código da função de hashing.
Visando a facilidade de implementação, velocidade e segurança (contra vários ataques propostos pelos próprios autores em [27]), os seguintes critérios foram adotados durante o desenvolvimento do MDx-MAC:

1. A chave secreta deve estar envolvida no início, no meio, no fim e em cada iteração da função de hashing;
2. a alteração na função de hashing deve ser mínima;
3. o desempenho deve ser próximo ao desempenho da função de hashing;
4. o requerimento de memória adicional deve ser mínimo;
5. as alterações devem ser genéricas, podendo ser aplicadas em qualquer função de hashing baseada no MD4.

5.5.1 MD5-MAC

Como exemplo de construção de uma função de MAC usando o MDx-MAC será descrito o algoritmo MD5-MAC, que é o MD5 modificado.

1. **Inicialização**

   Inicialização do vetor $U[0..2]$;

   Inicialização do vetor $T[0..2]$;

   $MD5Compress$ é definido como o MD5 sem o preenchimento (veja 6.7).

2. **Expansão da chave**

   A chave k de 128 bits escolhida pelo usuário é expandida da seguinte forma:

   $$K_i = MD5compress(k||U_i||k)$$

   para $0 \leq i \leq 2$

   O valor K_1 é dividido em quatro palavras de 32 bits:

   $$K_1 = K_1[0]||K_1[1]||K_1[2]||K_1[3]$$

3. **Alterações**

   O IV é substituído por K_0;

   $K_1[i]$ é adicionado modulo 2^{32} às constantes de adição ($y[i, 0..15]$) usadas nos 4 passos do MD5;

   K_2 é utilizado para gerar um bloco de 512 bits da seguinte forma:

   $$K_2||K_2 \oplus T[0]||K_2 \oplus T[1]||K_2 \oplus T[2]$$

   este bloco é concatenado no final da entrada, já preenchida, do MD5.

4. **Geração do MAC**

   Após estas modificações, o MD5 é usado normalmente e o MAC gerado é composto pelos 64 bits mais à esquerda da saída do MD5.

Capítulo 6

Análise geral da segurança de funções de hashing

Além dos ataques específicos já descritos no capítulo 4, outros, de caráter mais geral, são descritos neste capítulo. Apresentamos também métodos para fortalecimento de funções de hashing.

6.1 Objetivos dos adversários

Um adversário que queira atacar uma função de hashing deve tentar anular alguma de suas propriedades criptográficas:

Unidirecionalidade. Conhecendo um valor hash $y = h(x)$, com o x desconhecido, o adversário deve calcular um valor x' tal que $h(x') = y$. Note que provavelmente x' será diferente de x.

Exemplo: em sistemas de controle de acesso (veja 3.3.1) o adversário geralmente tem acesso aos valores hash das senhas utilizadas. Para atacar o sistema o adversário deve inverter algum dos valores hash a que ele teve acesso.

Resistência a segunda pré-imagem. Dados x e $h(x)$ o adversário deve gerar um valor $x' \neq x$ tal que $h(x) = h(x')$. Eventualmente, o cálculo da segunda pré-imagem pode ser mais fácil que a inversão, pois x pode revelar informações úteis ao adversário.

Exemplo: fraude de uma assinatura já existente.

Resistência a colisões. O adversário deve obter qualquer par x, x' tal que $h(x) = h(x')$.

Exemplo: preparação de mensagens para fraudar assinaturas. Encontrada a colisão x, x', onde o conteúdo de x é "interessante" para uma entidade A e o conteúdo de x' não, o adversário induz A a assinar x da seguinte forma:

$$S(pk, h(x))$$

Como $h(x) = h(x')$, a cadeia de bits que representa a assinatura em x será aceita também como a assinatura de x'.

Ataques à funções de MAC visam gerar um valor MAC válido para uma mensagem x criada por ele. Para tal existem duas abordagens:

- Forjar uma mensagem ou MAC sem o conhecimento da chave (veja 5.2.1, 5.2.3 e 5.2.2);
- recuperar o valor da chave.

Para efetuar o ataque o adversário deve conhecer um ou mais pares $x, MAC(k, x)$.

6.2 Quanto ao resultado do ataque

Dependendo do nível de controle que um adversário tenha sobre o processo de forja, classificamos os ataques em dois tipos:

1. Forja Existencial: o adversário não possui muito controle sobre o resultado obtido, interessando-lhe apenas alguma cadeia que satisfaça seu objetivo.
2. Forja Seletiva: o adversário tem condições de escolher e definir, mesmo que parcialmente, como será o resultado da forja.

A forja seletiva tem uma importância prática maior que a forja existencial: se uma mensagem forjada não tiver significado, para a pessoa que a lê ou para a aplicação que a usa, a fraude não obterá êxito e possivelmente será detectada.

6.3 Tipos de ataques

Quanto à forma em que são desenvolvidos, os ataques são divididos em:

1. Independentes de algoritmo: ataques independentes de algoritmo podem ser aplicados em qualquer função de hashing.

2. Dependente do encadeamento: podem ser aplicados em qualquer função de hashing iterativa.

3. Dependente do cifrador utilizado: estes ataques utilizam alguma falha existente no cifrador para quebrar a função de hashing.

4. Dependentes de algoritmo: este tipo de ataque é desenvolvido pela análise das características específicas de cada função de hashing: constantes, número de iterações e funções internas. Um ataque desenvolvido para uma determinada função pode não ser aplicável a outra.

Os ataques do tipo 1, 2 e 3 serão descritos a seguir; os ataques dependentes de algoritmo já foram descritos logo após os seus respectivos algoritmos.

6.4 Ataques independentes de algoritmos

6.4.1 Força bruta

O ataque de força bruta, ou busca exaustiva, consiste na tentativa de se quebrar alguma propriedade criptográfica de uma função utilizando grande poder computacional para testar todas as possibilidades existentes. O sucesso de um ataque de força bruta depende fundamentalmente dos seguintes fatores:

- **Tamanho do problema**. O tamanho do problema é determinado, principalmente, pelo tamanho dos parâmetros das funções utilizadas, tais como o tamanho da chave e do valor hash.

- **Poder de processamento**. Depende de quantas operações, no caso quantas operações de hashing, o adversário é capaz de executar em um período determinado de tempo. Para se determinar, pelo menos aproximadamente, o poder de processamento de um adversário, deve-se considerar o tipo de adversário e sua motivação. Quanto ao tipo, os adversários são divididos em: pessoas sozinhas, grupos de usuários, empresas ou governos; e quanto à motivação: diversão, fama, lucro ou vinganãa. O conhecimento do tipo de adversário e sua motivação, determinará quanto esforço será gasto no ataque, se serão usados poucos computadores, muitos computadores ou ainda se será desenvolvido hardware específico para ser usado na busca exaustiva.

- **Tempo relevante para conclusão do ataque**. O tempo que um adversário tem para efetuar seu ataque depende do tipo de uso que está sendo feito da função de hashing. Uma assinatura digital em um valor hash de um contrato pode ter validade de anos, ou até mesmo décadas; por outro lado, a chave MAC para transações em rede pode ter validade de alguns segundos.

- **Evolução da Tecnologia**. Se o tempo relevante para a conclusão do ataque for grande, deve-se também considerar a evolução que a tecnologia deve sofrer neste período de tempo.

Então, no desenvolvimento ou na escolha de uma função criptográfica, deve-se considerar a possibilidade de um ataque de força bruta e escolher uma função com parâmetros de tamanho seguros, levando em consideração os possíveis adversários, suas capacidades, o tempo útil para o ataque e como a função de hashing será usada.
A seguir mostraremos como a força bruta pode ser usada para quebrar as propriedades criptográficas da funções de hashing:

1. **Inversão e segunda pré-imagem**

   O ataque de força bruta para inverter ou calcular uma segunda pré-imagem é bastante simples: dado $y = h(x)$, o adversário deve gerar valores de x' até que $h(x') = y$. O valores assumidos por x' durante o ataque vão depender da estratégia usada pelo adversário:

   - x' pode ter valor inicial igual a 0 e ser incrementado em 1 a cada tentativa, ou pode ser gerado aleatoriamente;
   - os valores para x' podem ser provenientes de uma base de dados específica para o ataque: um conjunto de mensagens "interessantes" para o adversário ou de um dicionário (veja o ataque de dicionário na página 26).

   A cada tentativa o adversário tem chance de 2^{-n} de ter sucesso no ataque [1]; então, em média, o adversário deve obter sucesso após 2^{n-1} operações de hashing.

   Uma particularidade no ataque de força bruta para inverter ou calcular uma segunda pré-imagem, é que o seu limite superior é determinado pelo tamanho da *entrada* da função de compressão, e não pela saída. Por exemplo, usando a função de compressão f, que mapeia uma entrada de 512 bits para uma saída de 128, definida como:

$$f(x) = \begin{cases} \{1\}^{128} \text{ se } x = \{1\}^{512}, \\ \{0\}^{128} \text{ se } MD5(x) = \{1\}^{128}, \\ MD5(x) \text{ nos demais casos,} \end{cases}$$

[1]Usando ataque de dicionário a chance é maior

podemos concluir que o número médio de operações de hashing para inverter um valor aleatório diferente de 1_{128} deve ser de 2^{127}; mas, para inverter o valor 1_{128}, se o adversário inicializar x com 0 e incrementar x de 1 a cada tentativa, o ataque terá sucesso após 2^{512} operações, que está muito distante do valor pensado no início.

A função mostrada, obviamente, não é uma boa função de compressão. Ela não apresenta uma distribuição boa e um adversário que conheça seu algoritmo poderá facilmente inverter o valor hash 1_{128}. Serve, porém, para demonstrar que a busca exaustiva por uma inversão pode ser maior que 2^n, pois também depende do tamanho da entrada da função de compressão.

2. **Forja de MAC sem o conhecimento da chave**

   Algumas funções de MAC possibilitam a forja de um MAC sem o conhecimento prévio da chave. Exemplos destes tipos de função são as construções que usam sufixo ou prefixo secreto (veja 5.2.3 e 5.2.2).

3. **Cálculo da chave**

   Um adversário que tenha acesso a pares $(x, MAC(k, x))$ pode tentar recuperar k utilizando busca exaustiva. O processo de busca exaustiva consiste em gerar todas as chaves possíveis e calcular o MAC da mensagem usando as chaves geradas e comparando o resultado com o MAC recebido. Para uma chave de t bits, a busca tem tempo esperado de $O(2^t)$.

   Se o tamanho da chave for maior que o tamanho do MAC devem ocorrer colisões de chaves, isto é existem $k' \neq k$ tal que $MAC(k', M) = MAC(k, M)$ para alguns valores de M. Estas colisões fazem com que o processo de busca de chaves encontre aproximadamente 2^{t-n} chaves falsas. As chaves falsas podem ser detectadas e eliminadas se o adversário tiver acesso a outros pares $(x, MAC(k, x))$. Considerando que a função MAC se comporta como um mapeamento aleatório, serão necessários t/n pares (texto,MAC) para que o ataque encontre, com probabilidade perto de 1, a chave verdadeira [4].

   Tomados os devidos cuidados, um adversário que queira recuperar uma chave MAC usando força bruta terá muito pouco tempo para fazê-lo. Suponha que duas entidades A e B troquem mensagens por uma linha insegura, usando MAC para garantir autenticidade nas suas mensagens. Um adversário que queira usar a força bruta para descobrir a chave utilizada e inserir mensagens falsas na conversa entre A e B, terá que obter um número grande de pares $(x, MAC(k, x))$ e descobrir a chave k *enquanto* A e B ainda a utilizam. Supondo que A e B troquem de chave a cada hora, se o adversário gastar mais de uma hora para descobrir a chave ela não trará mais utilidade, pois as mensagens criadas pelo adversário não serão mais aceitas na conversa entre A e B.

O algoritmo MAA (veja 5.3) possui um MAC de 32 bits de comprimento e uma chave de 64 bits. Utilizando a busca exaustiva em um par $(x, MAC(k, x))$, o adversário deverá selecionar $1 + 2^{32}$ possíveis chaves para este par. Este subconjunto de chaves deverá ser testado em outros 2^{32} pares texto-MAC para que a chave verdadeira seja encontrada.

As exigências de processamento e memória para este ataque não são muito grandes, mas se o MAA for bem utilizado, o ataque não poderá ser concluído, pois o adversário precisa ter acesso a 2^{32} pares texto-MAC gerado pela mesma chave. O usuário do MAA pode evitar o ataque de força bruta usando chaves diferentes para cada cálculo de MAC.

4. **Força bruta no MAC**

   MAC's de comprimento pequeno, tais como o MAA de 32 bits, podem possibilitar ataques sem o conhecimento da chave. Para tal, um adversário gera uma mensagem que lhe convém e gera o MAC aleatoriamente. As ameaças deste ataque dependem do comprimento do MAC e das medidas tomadas quando o erro é detectado: se o adversário puder fazer este ataque com uma frequência alta sem levantar suspeitas, provavelmente o ataque poderá ser concluído.

5. **Combinando a busca da chave com a força no MAC**

   Supondo que um adversário possua poder de processamento suficiente para executar o ataque de força bruta para recuperar a chave mas não possua pares $(x, MAC(k, x))$ suficientes para determinar a chave correta, ele poderá usar a força bruta nos pares que possui, reduzindo o número de chaves falsas ao mínimo possível, e tentar o ataque com estas chaves.

6.4.2 Ataque do aniversário

O ataque do aniversário é um método que pode ser usado em qualquer função de hashing para diminuir o tempo de busca de uma colisão. Enquanto o ataque de força bruta exige 2^n operações de hashing no pior caso, o ataque do aniversário exige apenas $2^{n/2}$ operações de hashing.

1. **Paradoxo do Aniversário**

   O paradoxo do aniversário tem origem no seguinte questionamento:

Que tamanho um grupo de pessoas deve ter para que a probabilidade de que dois de seus integrantes tenham a mesma data de nascimento seja pelo menos 1/2?

Considerando que a chance de uma pessoa fazer aniversário em um determinado dia é de 1 em 365, a intuição inicial nos faz pensar que este número deva ser grande. O número correto, porém, é apenas 23. Vejamos porque:

Dado um grupo de q pessoas, a chance p de pelo menos duas delas compartilharem a data de aniversário é igual ao complemento da probabilidade de que tal par não exista. Isto é:

$$p = 1 - (364/365) \times (363/365) \ldots ((365 - q)/365).$$

Pode-se demonstrar que $p \approx 1/2$, ocorrerá quando q for aproximadamente igual a $1.17 \times \sqrt{365} \approx 23$.

O raciocínio acima pode ser facilmente estendido para calcular-se o trabalho exigido para gerar uma colisão: uma função cujo comprimento do valor hash é de n bits possui conjunto imagem com 2^n elementos; então, uma colisão deve ser encontrada, com chance de 1/2, após $2^{n/2}$ cálculos de valores hash.

2. **Funcionamento**

   Um adversário que deseje que uma mensagem M tenha o mesmo valor hash que uma mensagem M' (provavelmente para fraudar um sistema de assinaturas digitais) utiliza o Ataque do Aniversário da seguinte forma:

   - Gera $2^{n/2}$ variações de M. As variações devem ser simples e não detectáveis. Em um texto, podem ser escolhidas $n/2$ palavras para serem substituídas, de forma sistemática, por sinônimos; valores monetários podem ter os centavos alterados.
   - Calcula e armazena os valores hash de todas variações de M.
   - Produz $2^{n/2}$ variações de M'. Estas variações são obtidas da mesma forma que no ítem anterior.
   - Calcula o valor hash de cada uma das variações de M'. Cada valor hash gerado deve ser comparado com os valores hash das variações de M armazenados anteriormente. Quando dois valores iguais forem encontrados, o ataque estará concluído.

3. **Uso prático do Ataque do Aniversário**

   Uma colisão gerada por este ataque é seletiva: o autor do ataque escolheu cadeias de bits livremente, fato que torna o ataque muito poderoso. Mas, devido ao tamanho do valor hash usado pelas funções atualmente em uso, entre 128 e 160 bits, a busca pela colisão exige 2^{64} operações de hashing, o que é factível paralelizando o processamento [14, 18]. As exigências de memória para armazenar as mensagens é muito grande, o que torna o ataque inviável.

4. **Proteção contra o ataque do aniversário**

   A proteção mais óbvia é utilizar um valor hash de tamanho grande, mas existem outras alternativas:

   - Usar funções de hashing universais: o elemento aleatório utilizado por estas funções (veja em 6.8) invalida o trabalho de pré-computação do ataque do aniversário;
   - fazer alterações na mensagens antes de assiná-las: uma mensagem criada por outra pessoa pode ter sido produzida com o uso do ataque do aniversário. Então qualquer alteração na mensagem antes de assiná-la será uma proteção ao ataque.

6.5 Dependente do encadeamento

Os ataques dependentes do encadeamento podem ser aplicados em qualquer função de hashing iterativa. Estes ataques se concentram nas variáveis de encadeamento e nas funções de compressão, procurando gerar ou encontrar colisões internas e estendê-las para a função de hashing.
Os principais ataques dependentes do encadeamento são:

- **Correção de bloco:** este ataque consiste na alteração de um, ou poucos, blocos de uma mensagem para gerar uma colisão ou inverter um valor hash. O tempo esperado para este ataque é o mesmo que o do ataque por força bruta: 2^n para inversão e segunda pré-imagem e $2^{n/2}$ para colisão (usando o ataque do aniversário).

- **Meet-in-the-middle:** semelhante ao ataque do aniversário, este ataque procura colisões nos valores das variáveis de encadeamento, produzidas no calculo do hashing de duas mensagens. Se duas variáveis tiverem o mesmo valor, a colisão é formada pelo início das duas mensagens até o ponto em que as variáveis coincidem, concatenadas com a mesma cadeia: supondo que $h(x_0 \dots x_i) = h(x'_0 \dots x'_j)$ a colisão será o par $(x_0 ... x_i | y, x'_0 ... x'_j | y)$ onde y é uma cadeia qualquer de bits.

- **Ponto fixo:** um ponto fixo em uma função de compressão f (veja 1.3.3) é um par (H_{i-1}, x_i) tal que $f(H_{i-1}, x_i) = H_{i-1}$. Se um adversário puder gerar, ou encontrar, um ponto fixo, ele poderá repetir o bloco x_i, quantas vezes quiser, sem alterar as variáveis de encadeamento, gerando colisões. Este ataque pode ser evitado usando o fortalecimento-MD 6.7.

- **Criptoanálise diferencial:** a criptoanálise diferencial provou ser uma poderosa ferramenta na criptoanálise de cifradores. Acredita-se que ela possa ser usada para comparar várias entradas e saídas das funções de compressão e encontrar alguma característica que possa ser explorada [4].

6.6 Dependente do cifrador utilizado

Alguns cifradores de blocos possuem fraquezas que não são relevantes na proteção da privacidade mas que podem ser exploradas quando este cifradores são utilizados em funções de hashing. As principais fraquezas que podem ser exploradas são:

- **Chaves fracas**: a existência de chaves fracas, isto é chaves k para as quais $E_k(E_k(x)) = x$ para qualquer x, em um cifrador permite que o adversário replique quantas vezes achar necessário o bloco em que a chave fraca ocorreu.

- **Propriedade de complementação**: caso o cifrador utilizado possua a propriedade de complementação, isto é, $y = E_k(x) \Leftrightarrow \bar{y} = E_{\bar{k}}(\bar{x})$, a geração de mensagens com diferenças pré-determinadas é trivial em algumas implementações. Por exemplo, usando a função de compressão:

$$f(H_{i-1}, x_i) = E_{H_{i-1} \oplus x_i}(x_i) \oplus x_i,$$

  uma mensagem qualquer e seu complemento sempre terão o mesmo valor hash.

- **Pontos fixos**: os pontos fixos de um cifrador, isto é valores de x para os quais $E_k(x) = x$, podem ser utilizados para gerar ataques de ponto fixo na função de compressão.

- **Colisão de chaves**: a existência de k e k' para as quais $E_k(x) = E_{k'}(x)$ pode facilitar a busca por colisões na função de compressão.

6.7 Fortalecimento-MD

O método Merkle-Damgård de fortalecimento (fortalecimento-MD) consiste em concatenar ao final da entrada de dados x um bloco com a representação binária do comprimento de x. Isto é:

$$\text{Fortalecimento-MD}(x)=x||\ \ |x|$$

O uso do fortalecimento-MD aumenta a segurança da função de hashing, pois acrescenta redundância; os ataques que geram mensagens de tamanhos diferentes, como os de ponto fixo e Meet-in-the-middle, tornam-se assim inefetivos.
O fortalecimento-MD na função MD4 prepara a entrada para ser utilizada na função, além de concatenar o tamanho da entrada ao final da mesma. Faz também com que o tamanho da entrada seja múltiplo de 512. O algoritmo é descrito abaixo:

$x' := x||1$
$x' := x'||0^l$, onde $|x'| + l + 64$ é múltiplo de 512
$x' := x'|||x|$
, onde o bit mais significativo de $|x|$ segue se imediatamente ao último bit de x' (alinhamento à esquerda)

Este mesmo algoritmo também é usado nas funções MD5 e RIPEMD. Na função SHA-1 o algoritmo é o mesmo, mas o alinhamento de $|x|$ é feito à direita.

6.8 Funções de hashing universais

Funções de hashing universais são funções de hashing que dependem de fatores aleatórios para gerarem o valor hash. Uma construção simples de função de hashing universal (que chamaremos de hu) usando uma função de hashing convencional é feita concatenando um valor aleatório ao início da entrada x; isto é:

$$hu(x) = r||h(r||x).$$

A adição de elementos aleatórios à função de hashing aumenta a segurança: ataques de pré-computação (como o ataque do aniversário) são dificultados ou inibidos pois o adversário não tem conhecimento suficiente para executá-lo.
Suponha que um adversário A saiba qual função de hashing uma pessoa B normalmente usa para assinar contratos, e A queira que B assine x. Usando o ataque do aniversário, A consegue gerar x e x', tais que $h(x) = h(x')$, executando aproximadamente $2^{n/2}$ operações de hashing. Ao assinar x', B concatena ao seu início um valor aleatório r; assim B assina $h(r||x')$. Neste ponto, A descobre que o ataque do aniversário foi inútil: vai precisar usar novamente a força bruta para procurar uma **segunda pré-imagem**, que exige 2^n operações de hashing.

6.9 Tamanhos seguros

Considerando os ataques descritos e que a execução de 2^{80} operações de hashing está além do limite de possíveis adversários, as funções de hashing devem ter os seguintes parâmetros para serem consideradas seguras [4]:

- **Para garantia de unidirecionalidade**: o ataque de força bruta para inverter $h(x)$ exige 2^n operações; assim n deve ser maior ou igual a 80.
- **Para funções resistentes a segunda pré-imagem**: assim como para inverter, o ataque de força bruta usa 2^n operações; então, n deve ser maior ou igual a 80.
- **Para garantia de resistência a colisões**: o ataque do aniversário exige $2^{n/2}$ operações de hashing, o que implica que o tamanho do valor hash seguro contra este ataque deve ser o dobro do tamanho do valor recomendado contra inversão, isto é, 160 bits.
- **Para serem usadas em MAC**: na maioria das aplicações, em que a chave do MAC é usada poucas vezes e por pouco tempo, uma chave de 64 bits e um MAC de 32 bits de saída são grandes o suficiente para garantir segurança. Entretanto, o MAC de 32 bits pode encorajar ataques aleatórios ou de força bruta (veja na página 63). Assim, medidas de detecção devem ser implementadas ou deve-se aumentar o tamanho do MAC para 64 bits.

Capítulo 7

Aspectos práticos e conclusões

Neste capítulo descrevemos as implementações das funções em Java. Também serão feitas considerações para o uso das funções em situações práticas com comparação de velocidade e a apresentação de um protocolo criptográfico. No final será apresentada a conclusão e sugestões para trabalhos futuros.

7.1 Implementação

As funções de hashing apresentadas no Capítulo 5 foram implementadas para que os testes de colisões e velocidade fossem feitos, e também para serem utilizadas em aplicações que estão sendo desenvolvidas em outros projetos. A implementação, feita em Java, e informações sobre o uso das funções estão disponíveis em `http://www.dcc.unicamp.br/~`cripto.
Cada função de hashing possui duas implementações:

1. Implementação de Referência: é uma implementação que visa a compreensão e a facilidade de implementação. No geral ela é uma simples tradução do código em C para Java.

2. Implementação Otimizada: a linguagem Java possui diferenças em relação à linguagem C que devem ser observadas para obtermos eficiência. No desenvolvimento da versão otimizada os seguintes cuidados foram tomados:

   - Expansão das funções: na versão em C as iterações das funções de hashing são implementadas com #define, para evitar desvios e perda de velocidade. Abaixo, um exemplo de como uma iteração do MD4 é implementada em C:
     #define FF(a,b,c,d,x,s) (a)+=F((b),(c),(d))+(x); (a)=ROL((a),(s)); (1)

Como Java não possui #define, o equivalente mais parecido é a expansão das funções (funções *inline*). Entretanto, uma função equivalente à expressão (1) é considerada muito complexa pelo compilador, que não faz a expansão. Para evitar desvios e ganhar velocidade as iterações foram expandidas manualmente.

- Variáveis locais: as variáveis locais aos métodos são acessadas mais rapidamente e possuem um código mais compacto. O código para empilhar uma variável local ocupa apenas 1 byte, enquanto que o código para empilhar uma variável global ocupa de 3 a 4 bytes, dependendo do deslocamento Y (Tabela 7.1). Para aumentar a velocidade, as variáveis utilizadas na função de compressão são variáveis locais.

A classe principal que definimos é a interface CryptoHash, que define a interface disponibilizada pelas funções de hashing:

- public void reset() : inicializa a funções de hashing. Nas implementações feitas este método apenas inicializa as variáveis de encadeamento.
- public void setBlock(int newBlock[ ]) : define o bloco que será utilizado pela função de compressão.
- int[ ] hash(InputStream in): calcula o valor hash de **in**, que pode ser um arquivo, uma conexão de rede ou outros dados convertidos em InputStream.
- void compress(): invoca a função de compressão.
- int[ ] getHash(): retorna o valor hash.
- String getName(): retorna o nome da função de hashing.
- int getSize(): retorna o tamanho, em palavras de 32 bits, do valor hash utilizado pela função.

Local	Global
iload_X	aload_0 getfield #Y

Tabela 7.1: Empilhamento de valores

Além das funções de hashing, também foram implementadas duas classes para executar o preenchimento: a classe Padding é utilizada pelo MD4, MD5 e RIPEMD-160 e a classe SHAPadding é usada pelo SHA-1.
O calculo do valor hash utilizando estas implementações pode ser feito de duas maneiras:

- Direta: usando a função int[ ] hash(InputStream in), o valor hash é gerado diretamente e retornado em um vetor de inteiros. O preenchimento e as chamadas iterativas à função de compressão são feitas automaticamente;

- Indireta: utilizando a função void reset() para inicializar as variáveis de encadeamento e as funções void setBlock(int newBlock[ ]) e void compress() para executar as iterações e a função int[ ] getHash(), o valor hash pode ser calculado indiretamente. O calculo indireto é util caso se queira usar um preenchimento fora do padrão ou mesmo não usar o preenchimento, como ocorre no sistema HashCash.

Compilação JIT
Uma das vantagens do uso da linguagem Java é portabilidade: o bytecode gerado pode ser executado na maioria dos sistemas existentes e pode ser facilmente disponibilizado em paginas na Internet em forma de applets. A portabilidade é possível pois o bytecode é independente de sistema, sendo interpretado por uma JVM (Java Virtual Machine) dependente do sistema.
O custo de se atingir esta portabilidade é alto pois a interpretação do bytecode consome tempo de processamento, diminuindo a velocidade da aplicação. Uma solução para este problema é o uso de compilação JIT (just in time): as aplicações são distribuídas em códigos independentes do sistema e são compiladas antes da execução. A compilação JIT é feita por alguns browsers e por algumas JVM. Para testes de desempenho usamos o JIT TYA, disponível em http://www.blackdawn.org.

7.2 Comparação de desempenho

Abaixo segue a tabela comparativa de desempenho das funções de hashing implementadas (usando a implementação otimizada). Os testes foram feitos em um Pentium 133 rodando Linux.

Função	Tamanho da classe em bytes	Velocidade (kbytes/s)	Velocidade usando Tya (kbytes/s)
MD4	2415	0.44	0.75
MD5	3553	0.32	0.60
SHA-1	4203	0.12	0.23
RIPEMD-160	8322	0.09	0.15

Tabela 7.2: Velocidade das funções de hashing usando Java

7.3 Velocidade das funções de hashing para aplicações práticas

As velocidades das funções de hashing apresentadas anteriormente foram feitas com implementações em Java. A implementação em Java tem a vantagem da portabilidade mas apresenta o grande problema da velocidade: a interpretação tem um custo alto e mesmo a compilação JIT não é eficiente.
Implementações otimizadas apresentam velocidades muito superiores. A tabela 7.3 mostra as velocidades das funções de hashing implementadas em Assembly pelo grupo COSIC, o computador utilizado nos testes foi um Pentium 90.

Função	Velocidade (Mbits/s)
MD4	191.2
MD5	136.7
SHA-1	55.1
RIPEMD-160	45.5

Tabela 7.3: Velocidade das funções de hashing usando Assembly

A grande diferença nas velocidades mostradas nas Tabelas 7.2 e 7.3 deve-se às diferenças das linguagens utilizadas e na maneira de como os testes foram feitos (os resultados de 7.2 foram obtidos em situações reais e os de 7.3 foram obtidos em situação de teste, os dados e o código estavam na memória *cache* do processador).
Estas implementações permitem que as funções de hashing sejam usadas, mesmo em aplicações *on line*, sem causar impacto perceptível na maioria das situações. Por exemplo, um computador pessoal poderá calcular o hashing de uma transmissão de video *on line*, pois a função mais lenta, a RIPEMD-160, é capaz de calcular o hashing a 45 Mbits por segundo, suficiente para uma tranmissão de video MPEG [38].

7.4 SSL

Mostraremos a seguir como as funções de hashing são usadas no protocolo SSL 3.0 [19]. O SSL (*Secure Socket Layer*) foi escolhido pois é um dos protocolos mais populares e apresenta várias funcionalidades, como ciframento, integridade, autenticação e verificação de certificados. Como veremos, o SSL utiliza as funções de hashing de uma maneira bem conservadora:

- Sempre que possível o MD5 e o SHA-1 são utilizados juntos;
- o MAC é sempre cifrado antes de ser transmitido;

- as chaves e os valores secretos utilizados com hashing são maiores que as utilizadas no ciframento.

O SSL disponibiliza as seguintes funções de hashing: MD5, SHA-1 e HMAC (usando o MD5 ou o SHA-1).
As funções de hashing são utilizadas da seguinte forma:

1. Assinaturas digitais.

   O SSL pode fazer assinaturas da mesma forma que no DSS [2], onde o hash é calculado com o SHA-1. Uma alternativa disponibilizada, e aconselhada pois é mais segura, é a assinatura usando o SHA-1 e o MD5:

   $$S(pk, MD5(M)||SHA-1(M))$$

   .

2. Autenticidade e Integridade.

   São obtidos usando HMAC. O MAC gerado é cifrado junto com a mensagem antes de ser transmitido.

3. Geração de Chaves.

   O MD5 e o SHA-1 são utilizados para gerar as chaves de sessão utilizadas no SSL da seguinte forma:

   $$MD5(ms||SHA-1('A'||ms||serverhello||clienthello))||$$

   $$MD5(ms||SHA-1('BB'||ms||serverhello||clienthello))||$$

   $$MD5(ms||SHA-1('CCC'||ms||serverhello||clienthello))||\ldots$$

   onde ms é um segredo, de 48 bytes de comprimento, compartilhado pelo cliente e pelo servidor, e $serverhello$ e $clienthello$ são valores aleatórios, não secretos, gerados respectivamente pelo servidor e pelo cliente.

   Este processo se repete até que tenham sido gerados bytes suficientes para serem usados como chaves nas funções de MAC e de ciframento.

7.5 Conclusões e trabalhos futuros

O estudo das funções de hashing, sua implementação e o desenvolvimento deste texto nos permite chegar as seguintes conclusões:

- As funções de hashing, embora pouco conhecidas, tem uma presença importante em vários protocolos criptográficos, quer como parte essencial, quer como uma maneira de torná-los mais eficientes,
- em geral, as aplicações descritas no texto são recentes e ainda não estão sendo utilizadas. Com o passar do tempo estas aplicações devem sofrer melhorias e deverão ficar seguras para serem usadas em situações reais,
- as pessoas que trabalham com criptografia geralmente conhecem cifradores: funcionalidades, modos de operação, parâmetros de segurança, etc. As funções de hashing apresentam características específicas que devem ser observadas com cuidado: aplicações diferentes usam propriedades diferentes e propriedades diferentes usam parâmetros de segurança diferentes.

Alguns aspectos relativos às funções de hashing não puderam ser devidamente estudados, então, como sugestões para trabalhos futuros temos:

- **Implementações:** um estudo mais profundo de Java (linguagem, implementações das máquinas virtuais, *bytecodes*, etc.) deve possibilitar o desenvolvimento de funções mais rápidas. As aplicações apresentadas ainda estão para serem implementadas ou ainda podem sofrer melhorias,
- **estudo teórico das funções de hashing:** um estudo teórico mais profundo, com embasamento matemático e estatístico, deve revelar informações sobre a segurança das funções e os cuidados tomados pelos seus autores durante o seu desenvolvimento.

Apêndice A

Resumo dos conceitos criptográficos

Este apêndice tem como objetivo definir e explicar conceitos criptográficos básicos que serão úteis para a compreensão do texto principal.

A.1 Ciframento

A.1.1 Ciframento Simétrico

Ciframento é o processo que utiliza um pequeno segredo para tornar mensagens ininteligíveis para as pessoas que não conhecem este segredo. O processo de ciframento é representado por:

$$C = E_k(M)$$

onde E é a função de ciframento utilizada no processo; k é o segredo utilizado no ciframento; M é a mensagem original, também chamada de **texto-claro**, que se deseja proteger; e C é o produto do ciframento, também chamado de **texto-cifrado**.
Qualquer pessoa que conheça a função inversa de E, representada por D; que tenha tido acesso ao texto-cifrado C e que conheça a chave k poderá recuperar M da seguinte forma:

$$M = D_k(C).$$

Este tipo de cifrador, que utiliza a mesma chave na função de ciframento e na de deciframento é chamado de **ciframento simétrico**.

A.1.2 Ciframento Assimétrico

Imagine que duas pessoas, que nunca trocaram informação entre si, queiram trocar informações secretas e suspeitem que **todos** os seus meios de comunicação estão sendo vigiados, o ciframento simétrico não pode ser usado pois as pessoas envolvidas não compartilham uma chave k e tampouco podem escolhar uma chave e enviar ao outro, pois esta certamente seria capturada e usada para vigiar o resto da comunicação. Este tipo de problema pode ser resolvido usando **cifradores assimétricos**. Estes tipos de cifradores usam chaves diferentes para ciframento e deciframento. Vejamos como uma entidade A pode receber mensagens com privacidade:

- A gera um par de chaves (pk, sk). pk significa *public key*, isto é, uma chave que qualquer pessoa pode conhecer e sk siginifica *secret key*, uma chave que somente A conhece. O processo de escolha das chaves depende do tipo de cifrador utilzado.
- A informa a uma outra entidade B o valor de sua chave pk. Esta comunicação pode ser feita utilizando meios inseguros (e-mail, carta convencional) ou até mesmo utilizando meios públicos (páginas na internet, jornais),
- B envia $C = E_{pk}(M)$, onde M é a mensagem que se deseja proteger,
- A recebe C e recupera M usando a sua chave secreta: $M = D_{sk}(C)$.

Note que um adversário que tenha observado a comunicão entre A e B conhece pk e C, não podendo recuperar M.
Os cifradores assimétricos são bem mais lentos que os simétricos e, portanto, são utilizados para trocarem uma chave que será usada em um cifrador simétrico, e este sim será usado para cifrar a comunicação propriamente dita.

Cifradores de blocos e de fluxo

Quanto ao tipo da entrada e da saída, os cifradores são divididos em dois grupos:

- **Cifradores de blocos:** operam em blocos de dados, produzindo um bloco cifrado,
- **cifradores de fluxo:** operam em fluxo de bits, produzindo um outro fluxo de bits cifrados.

Modos de Operação

Usualmente os cifradores de blocos cifram blocos de 64 bits e na maioria das situações a mensagem que se deseja cifrar é maior que 64 bits, então foram desenvolvidos **modos de operação** para cifrar mensagens maiores que o bloco, dentre eles estão:

- Eletronic Codebook Mode (ECB): é o modo mais simples de operação: a mensagem é dividida em blocos do tamanho da entrada da função de ciframento e cada bloco é cifrado de forma independente. Este modo de operação é considerado muito inseguro pois permite que blocos cifrados sejam reordenados, removidos ou copiados,

- Cipher Block Chaining (CBC): neste modo de operação cada ciframento depende do bloco cifrado anteriormente:

$$C_i = E_k(M_i \oplus C_{i-1}),$$

onde C_i é o iésimo bloco cifrado e M_i é o iésimo bloco do texto-claro; M_0 é definido com valores aleatórios. O deciframento no modo CBC também é bastante simples:

$$M_i = C_{i-1} \oplus D_k(C_i).$$

O modo CBC é mais seguro que o ECB e o último bloco cifrado é dependente de todos os blocos do texto-claro e da chave, podendo então ser usado como MAC ou como hash. Veja a Figura A.1 para maiores detalhes.

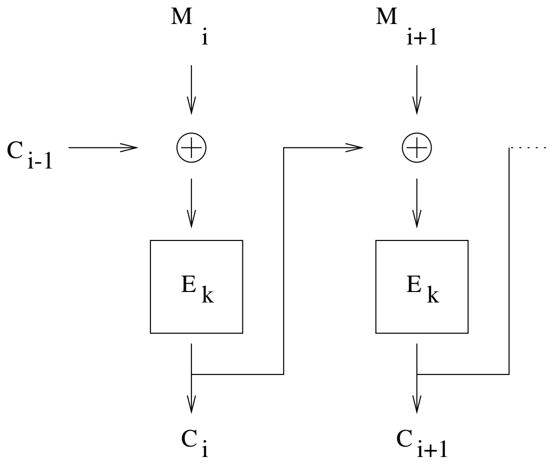

Figura A.1: Cifrador em modo CBC

- Cipher-feedback (CFB): neste modo de operação o cifrador produz um fluxo de bits pseudo-aleatórios que é utilizado para cifrar um fluxo de dados:

$$C_i = M_i \oplus E_k(C_{i-1})$$

o deciframento também é simples:

$$M_i = C_i \oplus E_k(C_{i-1}),$$

note que o deciframento usa a mesma função do ciframento. Um valor para M_0 deve ser escolhido antes do processamento iniciar. Veja a Figura A.2 para maiores detalhes.

Mais detalhes sobre segurança e funcionalidade, ou outros modo de operação estão disponíveis em [36, 4, 35].

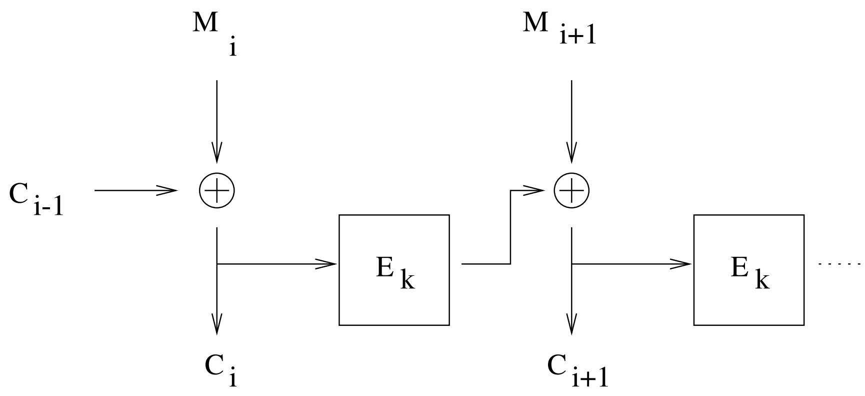

Figura A.2: Cifrador em modo CFB

A.2 Assinaturas Digitais

As assinaturas tradicionais (escritas em um papel) visam garantir as seguintes propriedades:

- **Autenticidade e integridade:** um documento sem rasuras e com uma assinatura é considerado autêntico, isto é, não sofreu modificações desde o momento em que foi assinado,
- **Identificação do autor ou concordância com o conteúdo:** a pessoa que assinou o documento ou é o seu autor ou concordou com o seu conteúdo,
- **não-repúdio:** a observação das duas propriedades citadas anteriormente impede a negação da autoria ou do conhecimento do conteúdo de um documento.

As assinaturas digitais visam proporcionar as propriedades existentes em documentos em papel para os documentos digitais.

A.2.1 Funcionamento

O funcioanmento das assinaturas digitais é parecido com o funcionamento dos cifradores assimétricos (geralmente são usados os mesmos algoritmos nos dois sistemas). O processo de assinatura e verificação é descrito a seguir:

- Uma entidade A gera um par de chaves (pk, sk). A informa a todos os seus conhecidos que sua chave pública é pk e guarda a sua chave secreta sk,
- para assinar uma mensagem M, A utiliza sk e um algoritmo de assinatura S:

$$y = S(sk, M),$$

y é a assinatura de A em M. O documento assinado é composto pela concatenação de M e y,

- uma outra entidade que conheça a chave pública de A, pk, verifica se a assinatura y é válida usando um algoritmo V:

$$M' = V(pk, y).$$

Se M' for igual a M a assinatura y é considerada válida, pois a função V usando pk é a inversa de S usando sk.

Bibliografia

[1] Secure hash standard. Technical Report FIPS Publication 180, National Bureau of Standards, 1993.

[2] Digital signature standard. Technical Report FIPS Publication 186, National Bureau of Standards, 1994.

[3] Secure hash standard. Technical Report FIPS Publication 180-1, National Bureau of Standards, 1995.

[4] Paul C. van Oorschot Alfred J. Menezes and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[5] R. Anderson. The classification of hash functions. In P. G. Farrell, editor, *Codes and cyphers: cryptography and coding IV: 4th Conference — December 1993, Cirencester*, pages 83–94. International Academic Publishers, 1993.

[6] Anônimo. *Maximum Security*. Sams.net Publising, 1997.

[7] Adam Back. Hash cash - a partial hash collision based postage scheme. http://www.dcs.ex.ac.uk/ aba/hashcash.

[8] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions: the HMAC construction. *CryptoBytes*, 2(1):12–15, Spring 1996.

[9] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[10] Ivan Bjerre Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 416–427. Springer-Verlag, 1990, 20–24 August 1989.

[11] Donald Watts Davies. A message authenticator algorithm. In G. R. Blakley and D. C. Chaum, editors, *Proc. CRYPTO 84*, pages 393–400. Springer, 1985. Lecture Notes in Computer Science No. 196.

[12] B. den Boer and A. Bosselaers. An attack on the last two rounds of md4. In *Advances in Cryptology—CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991, August 1991.

[13] B. den Boer and A. Bosselaers. Collisions for the compression function of md5. In *Advances in Cryptology - Eurocrypt 93*, volume 765 of *2*. Springer-Verlag, August 1993.

[14] Distributed.Net. Project des. http://www.distributed.net/des.

[15] H. Dobbertin. RIPEMD with two-round compress function is not collision-free. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 10(1):51–69, Winter 1997.

[16] Hans Dobbertin. Cryptanalysis of md4. In *Fast Software Encryption - Cambridge Workshop*, volume 1039, pages 53–69. Springer-Verlag, August 1996.

[17] Hans Dobbertin. The first two rounds of md4 are not one-way. In *Advances in Cryptology - Eurocrypt 93*, volume 765 of *2*. Springer-Verlag, August 1997.

[18] Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, July 1998.

[19] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol Version 3.0, 1996.

[20] Simson Garfinkel and Gene Spafford. *Pratical Unix & Internet Security*. O'Reilly & Associates, 1996.

[21] Rosario Gennaro and Pankaj Rohatgi. How to sign digital streams. In Burton S. Kaliski Jr., editor, *Advances in Cryptology—CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 180–197, 17–21 , publisher = 1997.

[22] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal of Software Tools*, 21(1):66, 68–70, January 1996.

[23] Donald E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, second edition, 1973.

[24] Unix Manual. *Crypt man page*.

[25] Ralph C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology—CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 428–446. Springer-Verlag, 1990, 20–24 August 1989.

[26] P. Metzger and W. Simpson. Ip authentication using keyed md5. *RFC 1828*, 1995.

[27] Bart Preneel and Paul C. van Oorschot. MDx-MAC and building fast MACs from hash functions. In *Crypto 95*, LNCS, pages 1–14. SV, 1995.

[28] Bart Preneel and Paul C. van Oorschot. On the security of two mac algorithms. In *Eurocrypt 96*, LNCS. SV, 1996.

[29] Ron Rivest. The md4 message-digest algorithm. *RFC 1320*, 1992.

[30] Ron Rivest. The md5 compress algorithm. *RFC 1321*, 1992.

[31] Ronald L. Rivest. All-or-nothing encryption and the package transform. In *Fast Software Encryption Conforence*, 1997.

[32] Ronald L. Rivest. Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes*, 4(1):12–17, Summer 1998.

[33] Ronald L. Rivest and Adi Shamir. Payword and MicroMint: Two simple micropayment schemes. *CryptoBytes*, 2(1):7–11, Spring 1996.

[34] M.J.B. Robshaw. On recent results for md2,md4 and md5. Technical report, RSA Laboratories, 1996.

[35] Douglas R.Stinson. *Cryptography: Theory and Pratice*. CRC Press, 1995.

[36] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley, 1996.

[37] Surety. How does digital notary service work? www.surety.com, 1998.

[38] Andrew S Tanenbaum. *Computer Networks*. International Edition. Prentice-Hall International, 1996.