

# Aula 10 - Funções Hash

Prof. Gabriel Rodrigues Caldas de Aquino

Instituto de Computação  
Universidade Federal do Rio de Janeiro  
gabrielaquino@ic.ufrj.br

Compilado em:  
September 19, 2025

# Funções de Hash

- Uma função de hash aceita uma mensagem de tamanho variável  $M$  como entrada.
- Produz um valor de tamanho fixo  $h = H(M)$ .
- O valor  $h$  é chamado de **hash** ou **digest**.

## Propriedades Desejáveis de Hash

- A saída deve parecer **aleatória** e estar **uniformemente distribuída**.
- Uma pequena mudança em  $M$  altera, com alta probabilidade, muitos bits de  $h$ .
- Principal objetivo: **integridade de dados**.
- Exemplo: se qualquer bit de  $M$  for alterado, o hash  $H(M)$  também mudará.

# Função de Hash Criptográfica

- Tipo especial de função de hash usada em aplicações de segurança.
- Deve ser **computacionalmente inviável** de quebrar com eficiência maior que força bruta.
- Usada para verificar se os dados foram alterados.

## Propriedades de uma Função de Hash Criptográfica

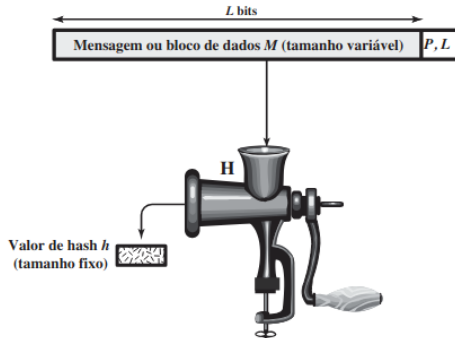
- **Mão única (one-way):** Dado um hash  $h$ , é inviável encontrar uma mensagem  $M$  tal que  $H(M) = h$ .
- **Livre de colisão (collision-free):** É inviável encontrar duas mensagens  $M_1$  e  $M_2$  tais que  $H(M_1) = H(M_2)$ .

# Preenchimento em Funções de Hash

- Funções de hash processam mensagens em blocos de tamanho fixo.
- Quando a mensagem não é múltiplo do tamanho do bloco, adiciona-se **preenchimento** (padding).
  - Mensagem preenchida até se tornar um múltiplo de um tamanho fixo (ex: 1024 bits).
- O preenchimento inclui o **tamanho original da mensagem** em bits.
  - **Objetivo:** dificultar que um atacante crie uma mensagem alternativa com o mesmo hash.
- Garante que cada mensagem de tamanho diferente resulte em um hash único e seguro.

# Preenchimento em Funções de Hash

**Figura 11.1** Função de hash criptográfica;  $h = H(M)$ .



$P, L$  = preenchimento mais campo de tamanho

# Aplicações de Funções de Hash Criptográficas

## Uso do Hash:

- Ela é usada em diversas aplicações de segurança e protocolos da Internet.
- Talvez o hash seja o algoritmo criptográfico mais versátil.

Uso onde a Hash é empregada:

- Autenticação de mensagem
- Assinaturas digitais
- Arquivo de senha de mão única
- Detecção de intrusão e detecção de vírus
- Função pseudoaleatória (PRF)
- Gerador de número pseudoaleatório (PRNG)

# Autenticação de Mensagem

- Autenticação de mensagem é um mecanismo usado para verificar a integridade de uma mensagem
- Garante que os dados recebidos estão exatamente como foram enviados, sem modificação, inserção, exclusão ou repetição.
- Em muitos casos, também é exigido que a identidade declarada do emissor seja validada.
- Em muitas aplicações, o valor gerado pelo Hash é chamado de **resumo de mensagem** (ou *digest*, em inglês).

# Autenticação de mensagem

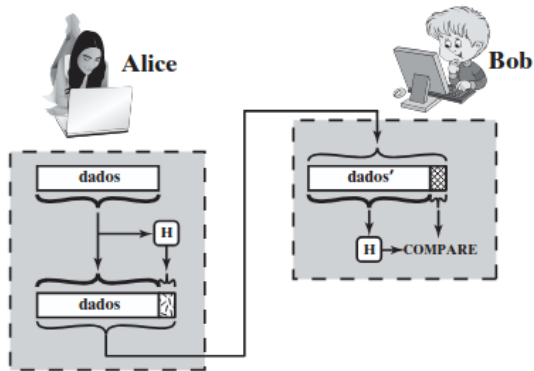
A essência do uso de uma função de hash para autenticação de mensagem é a seguinte:

1. O emissor calcula um valor de hash como função dos bits da mensagem.
2. O emissor transmite a mensagem junto com o valor de hash.
3. O receptor recalcula o valor de hash sobre a mensagem recebida.
4. O receptor compara o valor calculado com o valor recebido.

**Se houver divergência**, o receptor sabe que a mensagem (ou o valor de hash) foi alterada.



# Autenticação básica



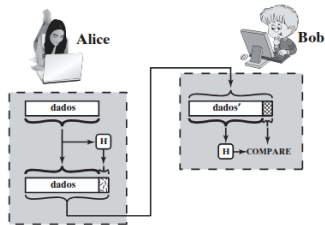
(a) Uso da função de hash para verificar integridade de dados

## Pergunta

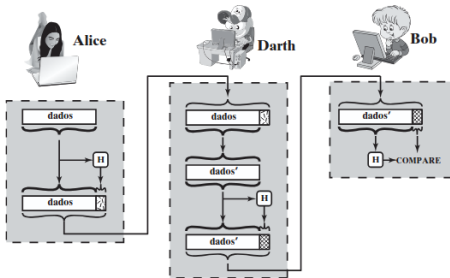
Qual o problema neste cenário?

# Autenticação básica - Problema

Figura 11.2 Ataque contra função de hash.



(a) Uso da função de hash para verificar integridade de dados



(b) Ataque man-in-the-middle

# Proteção do valor de hash

A função de hash precisa ser **transmitida de forma segura**.

- Se um adversário **alterar ou substituir** a mensagem, não deve ser viável alterar também o valor de hash para enganar o receptor.
- Exemplo de ataque:
  1. Alice transmite dados com o hash
  2. Darth intercepta, altera a mensagem e calcula um novo hash
  3. Bob recebe sem perceber a modificação.
- Para impedir esse ataque, **o valor de hash gerado por Alice precisa ser protegido**.

## Pergunta

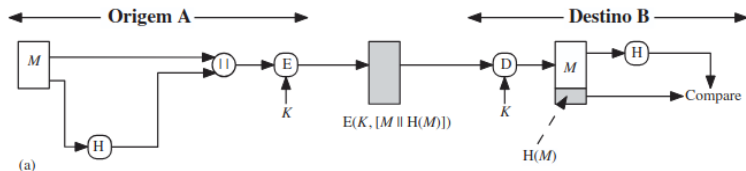
Como podemos proteger o Hash nesse cenário?

# Métodos de proteção da Hash

- Método A: Autenticação com hash + cifragem simétrica
- Método B: Cifrando o hash com cifragem simétrica
- Método C: Mensagem com Hash e Valor Secreto
- Método D: Mensagem com Hash mais Valor Secreto com confidencialidade

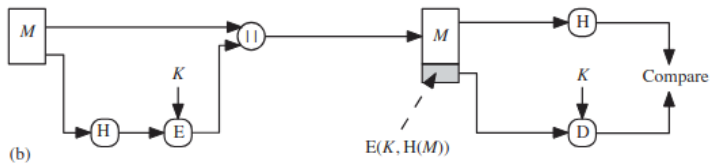
## Método A: Autenticação com hash + cifragem simétrica

- Mensagem mais o código de hash concatenado **são encriptados** usando a encriptação simétrica.
- Como somente A e B compartilham a chave secreta, a mensagem deverá ter vindo de A e sem alteração.
- O código de hash oferece a estrutura ou redundância exigida para conseguir a autenticação.
- Como a encriptação é aplicada à mensagem inteira mais o código de hash, a confidencialidade também é fornecida.



## Método B: Cifrando o hash com cifragem simétrica

- Somente o código de hash é encriptado, usando a encriptação simétrica.
- Isso reduz o peso do processamento para as **aplicações que não exigem confidencialidade**.



### Pergunta

Qual o motivo de passar a mensagem em texto plano?

# Vantagens do Uso de Hash mas sem a cifragem da mensagem

- Quando a **confidencialidade não é exigida**:
  - usar apenas hash (método com valor secreto) requer menos cálculos que cifrar a mensagem inteira.
  - O software de encriptação é relativamente lento, especialmente com fluxo constante de mensagens.
  - Custos de hardware de encriptação podem ser altos; chips de baixo custo existem, mas cada nó precisa ter capacidade.

## Exemplo desse cenário

Baixando a .iso do Linux Mint e verificando com GPG

# O que o GPG faz com esses dois arquivos?

## Comando:

```
gpg --verify sha256sum.txt.gpg sha256sum.txt
```

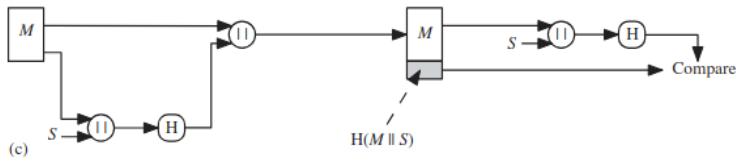
## Etapas realizadas pelo GPG:

1. Lê a assinatura digital contida em sha256sum.txt.gpg.
2. Calcula o hash real do conteúdo atual de sha256sum.txt.
3. Compara o hash calculado com o hash assinado.
  - Se forem **iguais**: a assinatura é válida (Good signature).
  - Se forem **diferentes**: a assinatura falha (BAD signature).



## Método C: Mensagem com Hash e Valor Secreto

- Usar apenas uma função de hash, sem cifragem, para autenticação de mensagem.
- Ambas as partes compartilham um valor secreto comum:  $S$ .
- Funcionamento:
  1. Emissor (A) calcula o hash sobre a concatenação da mensagem e do segredo:  $H(M \parallel S)$ .
  2. O valor de hash resultante é anexado à mensagem e enviado a B.
  3. Receptor (B), possuindo  $S$ , recalcula o hash para verificar a integridade e autenticidade.
- Como  $S$  não é enviado, um adversário não consegue modificar a mensagem nem gerar mensagens falsas.



## Método C é a base do HMAC - Hash Based Message Authentication Code

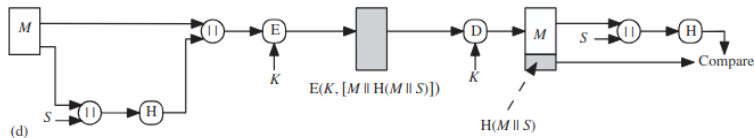
- A autenticação de mensagem normalmente é alcançada usando um **MAC** (Message Authentication Code), também chamado de função de hash chaveada.
- MACs são usados entre duas partes que compartilham uma **chave secreta** para autenticar informações trocadas.
- A função MAC recebe como entrada a chave secreta e um bloco de dados, produzindo um valor de hash (**MAC**) associado à mensagem.

# Código de Autenticação de Mensagem (MAC)

- Para verificar integridade, aplica-se novamente a função MAC sobre a mensagem e compara-se com o MAC recebido.
- Um invasor que altere a mensagem não poderá gerar o MAC correto sem conhecer a chave secreta.
- A verificação também garante autenticidade: apenas a parte que conhece a chave secreta pode ter gerado o MAC.

# Método D: Mensagem com Hash mais Valor Secreto com confidencialidade

- A **confidencialidade** pode ser acrescentada à abordagem do método anterior encriptando a mensagem inteira mais o código de hash

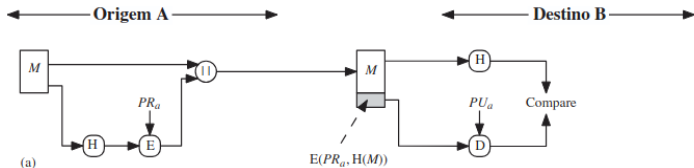


## Cenário

Esse cenário mostra exatamente um canal criptografado com a autenticação da mensagem, igual na VPN!

# Assinaturas Digitais

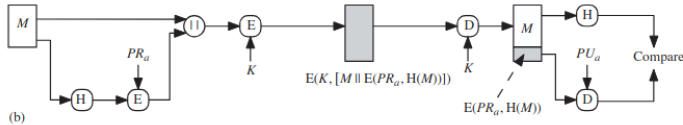
- Usa **Chave pública e privada**
- Assinatura digital é uma aplicação importante, similar à autenticação de mensagem.
- O valor de hash da mensagem é encriptado com a **chave privada** do usuário.
- Qualquer pessoa que conheça a **chave pública** do usuário pode verificar a integridade da mensagem associada à assinatura.
- Um invasor que tente alterar a mensagem precisaria conhecer a chave privada do usuário.



Esse caso é exatamente o caso do hash da iso do Linux Mint!

# Assinaturas Digitais com Confidencialidade

- Se, além da assinatura digital, o que se procura é confidencialidade, então a mensagem mais o código hash encriptado com a chave privada pode ser encriptado usando uma chave secreta simétrica. Essa é uma técnica comum.



# Arquivos de Senha de Mão Única

- Funções de hash são usadas para criar arquivos de senha de **mão única**.
  - No linux usa-se o `/etc/shadow`
- Em vez de armazenar a senha real, o sistema operacional armazena o **hash da senha**.
- Assim, mesmo que um hacker acesse o arquivo, a senha real não pode ser recuperada.
- Processo de autenticação:
  1. O usuário fornece a senha
  2. O sistema compara o **hash informado** com o hash armazenado.
- Este método é amplamente usado na maioria dos sistemas operacionais.

# Detecção de Intrusão e Vírus com Hash

- Funções de hash podem ser usadas para **detecção de intrusão** e **detecção de vírus**.
- Armazene  $H(F)$  para cada arquivo em um sistema e guarde os valores de hash de forma segura.
- Posteriormente, verifique se um arquivo foi modificado recalculando  $H(F)$ .
- Um intruso precisaria alterar  $F$  sem alterar  $H(F)$  para evitar detecção, o que é computacionalmente inviável.

## Trabalho interessante:

- [Documentação Labrador](#)
- [Código Labrador](#)

Além disso, Hashes são usadas em função pseudoaleatória (PRF) ou gerador de número pseudoaleatório (PRNG)



# Funções de Hash Simples

- Para entender considerações de segurança, apresentamos funções de hash simples e **não seguras**.
- Todas as funções de hash operam sobre blocos de  $n$  bits da entrada (mensagem, arquivo etc.).
- A entrada é processada bloco a bloco em um padrão iterativo para produzir um hash de  $n$  bits.
- Um exemplo simples: o **XOR bit a bit** de cada bloco.

$$C_i = b_{i1} \oplus b_{i2} \oplus \dots \oplus b_{im}$$

onde

$C_i$  =  $i$ -ésimo bit do código de hash,  $1 \leq i \leq n$

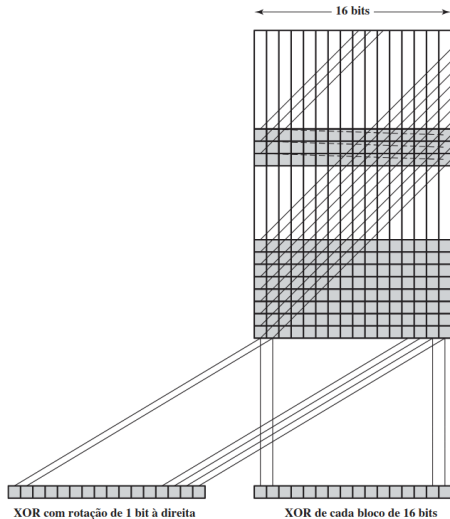
$m$  = número de blocos de  $n$  bits na entrada

$b_{ij}$  =  $i$ -ésimo bit no  $j$ -ésimo bloco

$\oplus$  = operação XOR

# Esquema de Hash simples com XOR

**Figura 11.5** Duas funções de hash simples.



# Limitações do XOR em Funções de Hash

- XOR simples não é suficiente quando apenas o código de hash é encriptado.
- Problema: blocos de texto cifrado podem ser **reordenados** sem alterar o valor do hash.
- Isso permite que um atacante modifique a mensagem sem que a integridade seja detectada.
- Conclusão: para garantir a integridade, precisamos de funções de hash criptográficas **não lineares e resistentes a colisões**.

# Pré-imagem e Colisões em Funções de Hash

- Para um valor de hash  $h = H(x)$ ,  $x$  é chamado de **pré-imagem** de  $h$ .
- Isso significa que  $x$  é um bloco de dados cuja função hash produz  $h$ .
- Funções hash são **mapas muitos-para-um**: para qualquer valor  $h$ , podem existir várias pré-imagens.
- Uma **colisão** ocorre se existirem  $x \neq y$  tal que  $H(x) = H(y)$ .
- Colisões são indesejáveis quando funções de hash são usadas para **integridade de dados**.

# Pré-imagens e Potenciais Colisões

- Suponha uma função de hash  $H$  com saída de  $n$  bits e entrada de  $b$  bits ( $b > n$ ).
- Total de mensagens possíveis:  $2^b$ .
- Total de valores de hash possíveis:  $2^n$ .
- Em média, cada valor de hash corresponde a  $2^{b-n}$  pré-imagens.
- Se  $H$  distribui uniformemente os valores de hash, cada hash terá aproximadamente  $2^{b-n}$  pré-imagens.
- Para entradas de tamanho variável, a variação de pré-imagens por valor de hash aumenta.
- Apesar disso, os riscos de segurança não são tão graves; é necessário definir requisitos precisos de segurança para funções de hash criptográficas.

# Requisitos de Funções de Hash

**Tabela 11.1** Requisitos para função de hash criptográfica H.

Requisito	Descrição
Tamanho de entrada variável	H pode ser aplicado em um bloco de dados de qualquer tamanho.
Tamanho da saída fixo	H produz uma saída de tamanho fixo.
Eficiência	$H(x)$ é relativamente fácil de calcular para qualquer valor de $x$ informado, através de implementações tanto em hardware quanto em software.
Resistência à pré-imagem (propriedade de mão única)	Para qualquer valor de hash $h$ informado, é computacionalmente impossível encontrar $y$ , de modo que $H(y) = h$ .
Resistência à segunda pré-imagem (resistência à colisão fraca)	Para qualquer bloco $x$ informado, é computacionalmente impossível encontrar $y \neq x$ com $H(y) = H(x)$ .
Resistência à colisão forte	É computacionalmente impossível encontrar qualquer par $(x, y)$ , de modo que $H(x) = H(y)$ .
Pseudoaleatoriedade	A saída de H atende os testes padrão de pseudoaleatoriedade.

As primeiras três propriedades são requisitos para a aplicação prática de uma função de hash.

# Resistência à Pré-imagem (Propriedade de Mão Única)

- A resistência à pré-imagem significa que é fácil gerar o código de hash a partir da mensagem.
- Porém, é praticamente impossível gerar a mensagem a partir do código de hash.
- Essencial quando a autenticação envolve um valor secreto que não é transmitido.
- Se a função de hash não tiver esta propriedade, um invasor pode:
  - Observar a mensagem  $M$  e o hash  $h = H(S||M)$ .
  - Inverter a função de hash para obter  $S||M = H^{-1}(h)$ .
  - Recuperar o valor secreto  $S$  facilmente.
- Portanto, a resistência à pré-imagem é crucial para proteger valores secretos.

# Resistência à Segunda Pré-imagem

- Garante que é impossível encontrar uma mensagem alternativa com o mesmo valor de hash de uma mensagem específica.
- Importante para prevenção contra falsificação quando se usa um hash encriptado.
- Se a função de hash não possuir essa propriedade, um invasor poderia:
  - Observar ou interceptar uma mensagem com seu hash encriptado.
  - Decriptar o hash da mensagem original.
  - Criar uma nova mensagem diferente com o mesmo hash.
- Portanto, esta propriedade protege contra ataques de falsificação.



# Funções Hash Fraca e Forte

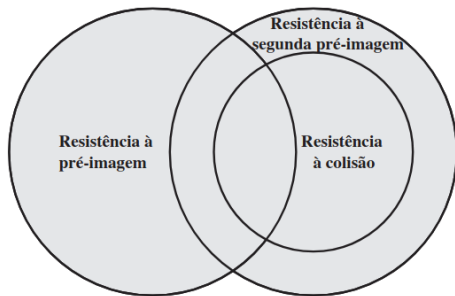
- Uma função de hash que satisfaz as primeiras cinco propriedades é chamada de **função hash fraca**.
- Se a função também satisfaz a sexta propriedade, **resistência à colisão**, é chamada de **função hash forte**.
- Função hash forte protege contra ataques onde terceiros tentam gerar mensagens diferentes com o mesmo hash.
- Exemplo de ataque sem resistência à colisão:
  - Bob cria duas mensagens diferentes com o mesmo hash ( $m_1$  e  $m_2$ ).
  - Alice assina a mensagem  $m_1$ .
  - Bob usa o hash da mensagem  $m_1$  para reivindicar que a mensagem  $m_2$  foi assinada.
- Portanto, a resistência à colisão é crucial para evitar falsificação de assinaturas.

# Pseudoaleatoriedade em Funções de Hash Criptográficas

- A pseudoaleatoriedade não é tradicionalmente citada como requisito formal, mas é implicitamente importante.
- Funções de hash criptográficas são frequentemente usadas para:
  - Derivação de chaves.
  - Geração de números pseudoaleatórios (PRNG/PRF).
- Nas aplicações de integridade de mensagens, as propriedades de resistência dependem de a saída parecer aleatória.
- Portanto, é apropriado considerar que uma função de hash produz uma saída pseudoaleatória.

# Relação entre propriedades de Funções de Hash

**Figura 11.6** Relação entre as propriedades das funções de hash.



- Uma função que é resistente à colisão também é resistente à segunda pré-imagem, mas o inverso não é necessariamente verdadeiro.
- Uma função pode ser resistente à colisão, mas não ser resistente à pré-imagem, e vice-versa. Uma função pode ser resistente à pré-imagem, mas não ser resistente à segunda pré-imagem e vice-versa.

# Propriedades de resistência Funções de Hash e seu uso

**Tabela 11.2** Propriedades de resistência necessárias para várias aplicações de integridade de dados.

	Resistência à pré-imagem	Resistência à segunda pré-imagem	Resistência à colisão
Hash + assinatura digital	sim	sim	sim*
Deteção de intrusão e detecção de vírus		sim	
Hash + encriptação simétrica			
Arquivo de senha de mão única	sim		
MAC	sim	sim	sim*

\*Resistência necessária se o invasor é capaz de elaborar um determinado ataque de mensagem

# Ataques de Força Bruta em Funções de Hash

- Ataques de força bruta não dependem do algoritmo de hash, apenas do tamanho do valor de hash em bits.
- Consiste em tentar todas as combinações possíveis até encontrar uma pré-imagem ou colisão.
- O esforço necessário cresce exponencialmente com o tamanho do hash: para um hash de  $n$  bits, há  $2^n$  possíveis valores.
- Diferente da criptoanálise, que explora vulnerabilidades específicas do algoritmo.
- Exemplo: para um hash de 128 bits, seriam necessárias  $2^{128}$  tentativas em média para encontrar uma colisão por força bruta.

# Ataques de Pré-imagem e Segunda Pré-imagem

**Cenário:** O atacante conhece uma saída da Hash

- O adversário busca um valor  $y$  tal que  $H(y) = h$ , onde  $h$  é um hash conhecido.
- Método de força bruta: testar valores aleatórios de  $y$  até encontrar uma correspondência.
- Para um hash de  $m$  bits, o esforço médio necessário é  $2^{m-1}$  tentativas.
- Esse ataque explora a dificuldade de inverter a função de hash (resistência à pré-imagem).

**Cenário:** O atacante busca descobrir pares de saída da Hash

- O adversário busca duas mensagens  $x$  e  $y$  tal que  $H(x) = H(y)$ .
- Esse ataque exige **menos esforço** do que um ataque de pré-imagem ou segunda pré-imagem.
- Baseado no **paradoxo do aniversário**: a probabilidade de colisão aumenta rapidamente com o número de tentativas.
- Para um hash de  $m$  bits, o esforço médio para encontrar uma colisão é aproximadamente  $2^{m/2}$  tentativas.

# Ataque de Colisão Explorado via Paradoxo do Aniversário

Estratégia para explorar o paradoxo do dia do aniversário:

- Preparação da origem A: mensagem legítima  $x$  é criada
- Oponente gera  $2^{m/2}$  variações  $x'$  de  $x$  com mesmo significado e armazena os hashes.
- Oponente prepara uma mensagem fraudulenta  $y$  para a qual deseja a assinatura.
- Pequenas variações  $y'$  de  $y$  são geradas; o oponente calcula  $H(y')$  e verifica correspondência com algum  $H(x')$ .
- Quando há correspondência, a variação válida de A é usada para assinatura, que é então aplicada à variação fraudulenta  $y'$ .

Ambas produzem a mesma assinatura.



# Exemplo de Ataque de Colisão com Hash de 64 bits

- Com um hash de 64 bits, o esforço necessário é da ordem de  $2^{32}$ .
- Criação de variações que mantêm o mesmo significado não é difícil.
- Exemplos de variações:
  - Inserção de pares de caracteres “espaço-espaço-retrocesso” entre palavras.
  - Substituição de “espaço-retrocesso-espaço” em posições selecionadas.
  - Reescrita da mensagem mantendo o significado original.

# Resumo do esforço exigido

**Resumo:** Para um código de hash de tamanho  $m$ , o nível de esforço exigido, conforme vimos, é proporcional ao seguinte:

Resistência à pré-imagem	$2^m$
Resistência à segunda pré-imagem	$2^m$
Resistência à colisão	$2^{m/2}$

# Resistência a Colisões em Hashes

- A resistência à colisão é desejável em códigos de hash seguros.
- Para um hash de  $m$  bits, a força contra ataques por força bruta é aproximadamente  $2^{m/2}$ .
- Van Oorschot e Wiener [VANO94] projetaram uma máquina de US\$10 milhões para MD5 (128 bits):
  - Capaz de encontrar uma colisão em 24 dias.
  - Mostra que 128 bits é inadequado para segurança moderna.
- Hashes de 160 bits (como SHA-1) aumentam a resistência:
  - Mesma máquina levaria mais de 4.000 anos para encontrar uma colisão.
  - Contudo, com tecnologia atual, 160 bits começa a ficar suspeito.

# Criptanálise de Funções de Hash e MACs

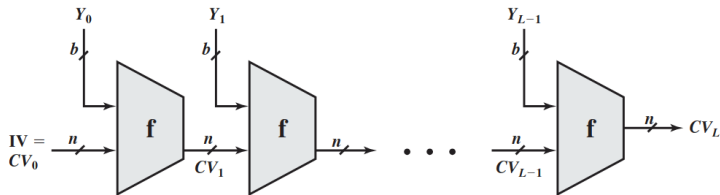
- Assim como em algoritmos de criptografia, ataques criptoanalíticos em **hashes** e **MACs** buscam explorar propriedades do algoritmo para superar a simples busca exaustiva.
- A resistência de um hash ou MAC à criptoanálise é medida comparando-se seu esforço com o esforço de um ataque por força bruta.
- Um hash ou MAC ideal exigirá **esforço criptoanalítico maior ou igual ao esforço por força bruta**.

# Estrutura Iterativa de Funções de Hash

- A estrutura iterativa de hash foi proposta por Merkle [MERK79, MERK89] e é usada na maioria das funções de hash atuais, incluindo SHA.
- Funcionamento geral:
  - A mensagem de entrada é dividida em  $L$  blocos de tamanho fixo  $b$  bits.
  - Se necessário, o bloco final é preenchido para completar  $b$  bits e inclui o tamanho total da mensagem.
  - Incluir o tamanho dificulta ataques, pois o oponente deve encontrar colisões com mensagens do mesmo ou de tamanhos diferentes que levem ao mesmo hash.
- O algoritmo usa repetidamente uma **função de compactação  $f$** :
  - Recebe duas entradas: a **variável de encadeamento** ( $n$  bits da etapa anterior) e o bloco atual ( $b$  bits). Normalmente,  $b > n$ ;
  - Produz uma saída de  $n$  bits.
  - A variável de encadeamento inicial é definida pelo algoritmo e o valor final dela é o **hash resultante**.

# Estrutura geral de hash seguro

**Figura 11.8** Estrutura geral do código de hash seguro.



$IV$  = valor inicial  
 $CV_i$  = variável de encadeamento  
 $Y_i$  =  $i$ -ésimo bloco de entrada  
 $f$  = algoritmo de compactação

$L$  = Número de blocos de entrada  
 $n$  = Tamanho do código de hash  
 $b$  = Tamanho do bloco de entrada

A função de hash pode ser resumida da seguinte forma:

$$CV_0 = IV = \text{valor inicial de } n \text{ bits}$$

$$CV_i = f(CV_{i-1}, Y_{i-1}) \quad 1 \leq i \leq L$$

$$H(M) = CV_L$$

onde a entrada da função de hash é uma mensagem  $M$  consistindo nos blocos  $Y_0, Y_1, \dots, Y_{L-1}$

# Motivação e Criptoanálise de Funções de Hash

- Motivação da estrutura iterativa (Merkle [MERK89], Damgård [DAMG89]):
  - Se a função de compactação  $f$  for à prova de colisão, a função de hash iterativa resultante também será.
  - Permite criar hashes seguros para mensagens de qualquer tamanho.
  - O design seguro de uma função de hash se reduz ao design de uma função de compactação segura para blocos de tamanho fixo.
- Criptoanálise de funções de hash:
  - Foca na estrutura interna de  $f$ .
  - Ataques procuram produzir colisões eficientes para uma única execução de  $f$ , considerando o valor fixo do IV.
  - Normalmente,  $f$  consiste em várias rodadas, e o ataque analisa padrões de mudança de bits entre rodadas.



# Colisões em Funções de Hash

- **Importante:** Para qualquer função de hash, **colisões sempre existem**:
  - Mensagens têm tamanho  $\geq 2b$  (devido ao campo de tamanho)
  - Hashes têm tamanho fixo  $n$ , com  $b > n$
- O objetivo de uma função de hash segura não é eliminar colisões (isso é impossível), mas torná-las **computacionalmente inviáveis de encontrar**.
- Assim, a segurança é definida pelo **esforço necessário para descobrir uma colisão**, não pela sua inexistência.

# Secure Hash Algorithm (SHA)

- O **SHA** é a função de hash mais utilizada nos últimos anos.
- Em 2005, era praticamente o último algoritmo de hash padronizado restante após vulnerabilidades em outros algoritmos.
- Desenvolvido pelo **NIST** e publicado como padrão federal (**FIPS 180**) em 1993.
- Primeira versão (**SHA-0**) apresentou vulnerabilidades criptoanalíticas.
- Revisão lançada em 1995 (**FIPS 180-1**), conhecida como **SHA-1**.
- Baseado na função de hash **MD4**, seguindo de perto seu projeto.

# SHA-1 e SHA-2

- **SHA-1** produz um hash de 160 bits.
- Em 2002, o **NIST** revisou o padrão (**FIPS 180-2**), definindo três novas versões:
  - SHA-256 (256 bits)
  - SHA-384 (384 bits)
  - SHA-512 (512 bits)
- Coletivamente, estas versões são conhecidas como **SHA-2**.
- Mantêm a mesma estrutura básica do SHA-1, usando aritmética modular e operações binárias lógicas.
- Em 2008, o **FIPS PUB 180-3** adicionou uma versão de 224 bits (SHA-224).
- SHA-1 e SHA-2 também são especificados no **RFC 6234**, que inclui implementação em C.

# Descontinuação do SHA-1

- Em 2005, o **NIST** anunciou a intenção de retirar a aprovação do SHA-1 e adotar o SHA-2 por volta de 2010.
- Uma equipe de pesquisa demonstrou um ataque que poderia gerar duas mensagens diferentes com o mesmo hash SHA-1 usando  $2^{69}$  operações.
- Este número é significativamente menor que as  $2^{80}$  operações anteriormente estimadas para encontrar uma colisão.
- O resultado acelerou a necessidade de transição para SHA-2.
- Referência: Wang et al. [WANG05].

# Tabela SHA

**Tabela 11.3** Comparação de parâmetros do SHA.

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Tamanho do resumo da mensagem	160	224	256	384	512
Tamanho da mensagem	$< 2^{64}$	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Tamanho do bloco	512	512	512	1024	1024
Tamanho da word	32	32	32	64	64
Número de etapas	80	64	64	80	80

*Nota:* todos os tamanhos são medidos em bits.

# SHA-512: Etapa 1 - Preenchimento

- Entrada: mensagem com tamanho menor que  $2^{128}$  bits.
- Saída: resumo (hash) de 512 bits.
- Processamento em blocos de 1024 bits.
- **Etapa 1 - Preenchimento:**
  - A mensagem é preenchida para que o tamanho seja congruente a 896 módulo 1024.
  - O preenchimento é sempre aplicado, mesmo se a mensagem já tiver o tamanho desejado.
  - Número de bits de preenchimento: entre 1 e 1024.
  - Estrutura do preenchimento: um bit 1 seguido pelos bits 0 necessários.

- **Etapa 2 - Anexar tamanho:**
  - Um bloco de 128 bits é anexado à mensagem.
  - O bloco contém o tamanho da mensagem original (antes do preenchimento) como um inteiro de 128 bits sem sinal.
  - Ordem dos bytes: byte mais significativo primeiro.
- Após as duas primeiras etapas, a mensagem resultante tem comprimento múltiplo de 1024 bits.
- Mensagem expandida representada como blocos de 1024 bits:  $M_1, M_2, \dots, M_N$ .
- Tamanho total da mensagem expandida:  $N \times 1024$  bits.

## SHA-512: Inicialização do Buffer de Hash

- Um buffer de 512 bits mantém resultados intermediários e finais.
- Representado por 8 registradores de 64 bits:  $a, b, c, d, e, f, g, h$ .
- Inicialização com valores hexadecimais:

$a =$	6A09E667F3BCC908	$e =$	510E527FADE682D1
$b =$	BB67AE8584CAA73B	$f =$	9B05688C2B3E6C1F
$c =$	3C6EF372FE94F82B	$g =$	1F83D9ABFB41BD6B
$d =$	A54FF53A5F1D36F1	$h =$	5BE0CD19137E2179

- Valores armazenados em **big-endian**.
- Derivados dos primeiros 64 bits das partes fracionárias das raízes quadradas dos oito primeiros números primos.