Mestrado em Engenharia Informática (MEI)
Mestrado Integrado em Engenharia Informática (MiEI)

Perfil de Especialização **CSI** : Criptografia e Segurança da Informação

Engenharia de Segurança

Tópicos

- **Parte III: Segurança da Informação**
- Parte IV: Cifras simétricas
- Parte V: Funções de sentido único

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Propriedades de segurança

- A criptografia é utilizada para fornecer garantias referentes a um vasto leque de propriedades de segurança:
  - **Confidencialidade**: garantir que o conteúdo da mensagem só é do conhecimento dos intervenientes legítimos.
  - **Integridade**: garantir que o receptor não "aceita" mensagens que tenham sido manipuladas.
  - **Autenticidade**: assegurar a "origem" da mensagem.
  - **Não repúdio**: demonstrar a "origem" da mensagem.
  - **Anonimato**: não fornecer qualquer informação sobre a "origem" da mensagem.
  - **Identificação**: assegurar a "identidade" do interveniente na comunicação

Serviços e protocolos criptográficos

- Normalmente não estamos interessados numa única propriedade per si, mas numa combinação de propriedades de segurança (e.g. num canal seguro entre duas partes pretende-se garantir a confidencialidade, autenticidade e integridade).
- Por outro lado, algumas das propriedades de segurança referidas não resultam diretamente de uma técnica criptográfica específica, mas antes de uma conjugação de técnicas.
- A combinação de técnicas resulta no que se designa por **protocolos criptográficos** – nesses protocolos especificam-se as trocas de mensagens (e as técnicas criptográficas utilizadas) para se atingirem os fins pretendidos.
- A segurança de protocolos criptográficos (i.e. se eles realmente cumprem os requisitos para que foram desenvolvidos) não depende unicamente da segurança das técnicas que os suportam.

Criptografia e Segurança

A segurança das técnicas criptográficas constituem um ingrediente fundamental e necessário na segurança de sistemas informáticos, mas só isso não é suficiente para a segurança da informação/dados.

- Podemos distinguir (pelo menos) os seguintes níveis no estabelecimento da segurança de um sistema informático:
  - Técnica criptográfica;
  - Protocolo criptográfico;
  - Implementação;
  - Utilização;
  - Configuração e Manutenção.
- Uma brecha de segurança em qualquer um destes níveis compromete a segurança de todo o sistema de informação (elo mais fraco).

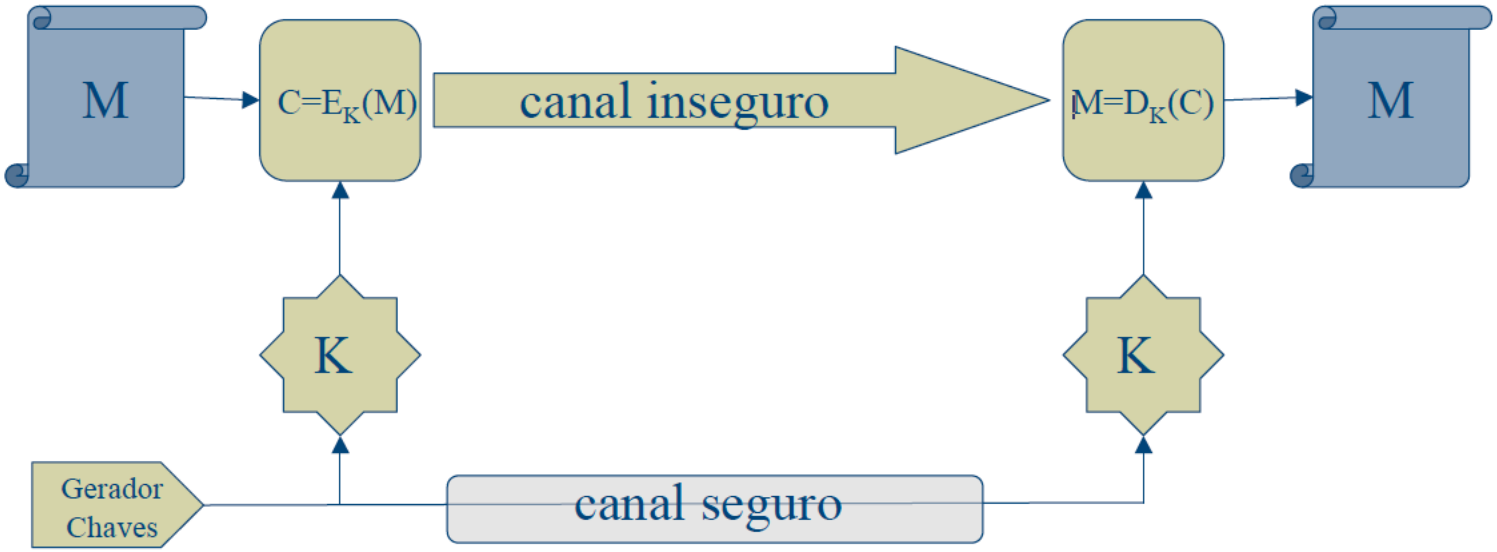
Tópicos

- Parte III: Segurança da Informação
- **Parte IV: Cifras simétricas**
  - Cifras sequenciais (*Stream ciphers*)
  - Cifras por blocos (*Block ciphers*)
  - Utilização
- Parte V: Funções de sentido único

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Cifras simétricas

- As cifras simétricas caracterizam-se pela **mesma chave** ser utilizada na operação de cifra/decifragem.
- Pressupõe por isso que, numa fase prévia à comunicação, se procedeu ao **acordo de chaves**.
  - ... operação que "tipicamente" envolve a utilização de **canais seguros**.

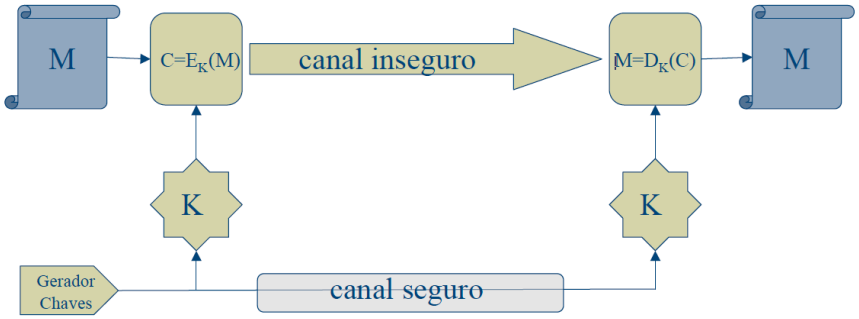


- Exemplos: DES, 3DES, IDEA, AES.

Cifras simétricas

Relembrando o Princípio de Kerckhoff (Um sistema criptográfico deve ser seguro mesmo quando todo o sistema é de conhecimento público, à excepção da chave) como é que formalizamos a cifra simétrica?

- Seja
  - $\mathcal{M}$ o espaço de todas as mensagens, $\mathcal{C}$ o espaço de todos os *ciphertext*, e $\mathcal{K}$ o espaço de todas as chaves;
  - $m \in \mathcal{M}, c \in \mathcal{C}$, e $k \in \mathcal{K}$
  - as funções de encriptação $\mathrm{E}: \mathcal{M} \times \mathcal{K} \rightarrow \mathcal{C}$ e decifra $\mathrm{D}: \mathcal{C} \times \mathcal{K} \rightarrow \mathcal{M}$
- Para garantir a **propriedade de correção** da cifra simétrica temos que garantir que a decifra de uma mensagem cifrada é a mensagem original
  - $\forall\, m, k: \mathrm{D}_k\big(\mathrm{E}_k(m)\big) = m$
- Para garantir a **propriedade de segurança**, idealmente o ciphertext não revela nada sobre a chave ou sobre a mensagem



Como garantir a propriedade da segurança de uma cifra?

Ordene os seguintes argumentos pela efetividade em convencer alguém que uma cifra é **segura** (1: melhor argumento ... 4: pior argumento).

☐	Muitas pessoas, muito inteligentes e altamente motivadas tentaram quebrar a cifra e não o conseguiram
☐	Existem 948 quadriliões de possíveis chaves, pelo que a cifra deve ser segura.
☐	Aqui está uma prova matemática, aceite pelos especialistas, que demonstra que a cifra é segura.
☐	Aqui está um argumento forte em como quebrar a cifra é no mínimo tão difícil como resolver um problema que já sabemos que é difícil de resolver (e se quebrarmos a cifra estamos a resolver esse problema).

Como garantir a propriedade da segurança de uma cifra?

Ordene os seguintes argumentos pela efetividade em convencer alguém que uma cifra é **segura** (1: melhor argumento ... 4: pior argumento).

	Muitas pessoas, muito inteligentes e altamente motivadas tentaram quebrar a cifra e não o conseguiram
	Existem 948 quadriliões de possíveis chaves, pelo que a cifra deve ser segura.
1	Aqui está uma prova matemática, aceite pelos especialistas, que demonstra que a cifra é segura.
	Aqui está um argumento forte em como quebrar a cifra é no mínimo tão difícil como resolver um problema que já sabemos que é muito difícil de resolver (e se quebrarmos a cifra estamos a resolver esse problema).

Como garantir a propriedade da segurança de uma cifra?

Ordene os seguintes argumentos pela efetividade em convencer alguém que uma cifra é **segura** (1: melhor argumento ... 4: pior argumento).

	Muitas pessoas, muito inteligentes e altamente motivadas tentaram quebrar a cifra e não o conseguiram
	Existem 948 quadriliões de possíveis chaves, pelo que a cifra deve ser segura.
1	Aqui está uma prova matemática, aceite pelos especialistas, que demonstra que a cifra é segura.
2	Aqui está um argumento forte em como quebrar a cifra é no mínimo tão difícil como resolver um problema que já sabemos que é muito difícil de resolver (e se quebrarmos a cifra estamos a resolver esse problema).

Como garantir a propriedade da segurança de uma cifra?

Ordene os seguintes argumentos pela efetividade em convencer alguém que uma cifra é **segura** (1: melhor argumento ... 4: pior argumento).

3	Muitas pessoas, muito inteligentes e altamente motivadas tentaram quebrar a cifra e não o conseguiram
	Existem 948 quadriliões de possíveis chaves, pelo que a cifra deve ser segura.
1	Aqui está uma prova matemática, aceite pelos especialistas, que demonstra que a cifra é segura.
2	Aqui está um argumento forte em como quebrar a cifra é no mínimo tão difícil como resolver um problema que já sabemos que é muito difícil de resolver (e se quebrarmos a cifra estamos a resolver esse problema).

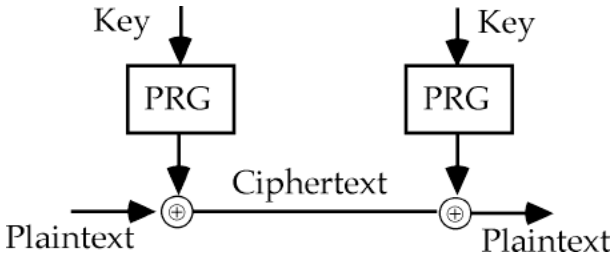
Como garantir a propriedade da segurança de uma cifra?

Ordene os seguintes argumentos pela efetividade em convencer alguém que uma cifra é **segura** (1: melhor argumento ... 4: pior argumento).

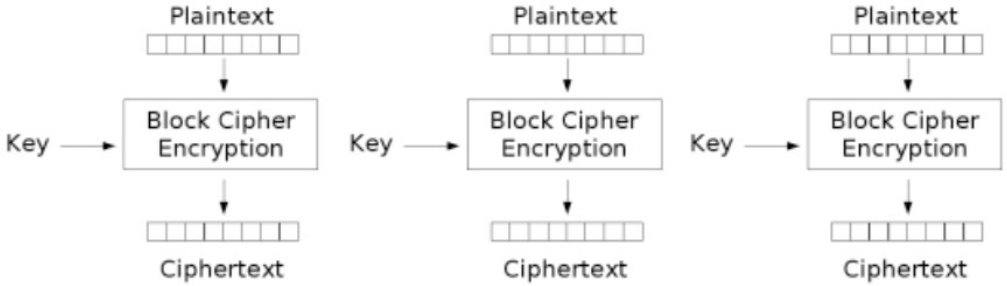
3	Muitas pessoas, muito inteligentes e altamente motivadas tentaram quebrar a cifra e não o conseguiram
4	Existem 948 quadriliões de possíveis chaves, pelo que a cifra deve ser segura.
1	Aqui está uma prova matemática, aceite pelos especialistas, que demonstra que a cifra é segura.
2	Aqui está um argumento forte em como quebrar a cifra é no mínimo tão difícil como resolver um problema que já sabemos que é muito difícil de resolver (e se quebrarmos a cifra estamos a resolver esse problema).

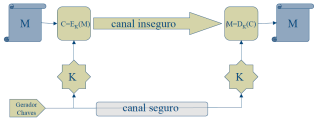
Tipos de cifras simétricas

- As cifras simétricas modernas podem-se agrupar em dois tipos:
  - Cifra sequencial (*stream cipher*)



  - Cifra por blocos (*block cipher*)





Tópicos

- Parte III: Segurança da Informação
- **Parte IV: Cifras simétricas**
  - **Cifras sequenciais (*Stream ciphers*)**
  - Cifras por blocos (*Block ciphers*)
  - Utilização
- Parte V: Funções de sentido único

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

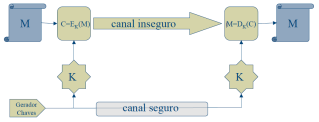
Cifra simétrica sequencial (*stream cipher*)

- A cifra One-Time-Pad é uma cifra simétrica, e tal como vimos na aula anterior
  - O comprimento da chave é o mesmo (ou maior) da mensagem a cifrar;
  - A chave (i) é completamente aleatória, (ii) não pode ser reutilizada em parte ou na sua totalidade, e (iii) tem de ser mantida em completo segredo pelas partes comunicantes;
  - Operações de cifra/decifragem são simplesmente o *XOR* com a chave, bit a bit (ou byte a byte).

$$C_i = T_i \oplus K_i \qquad M_i = C_i \oplus K_i$$

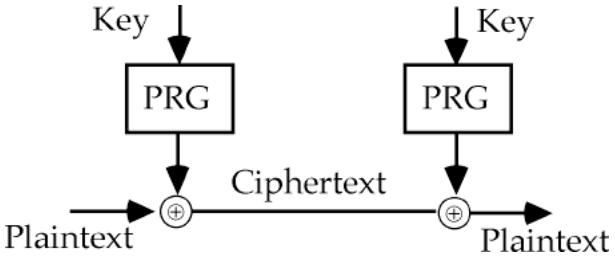
Cifra sequencial

  - Na prática é uma cifra inviável devido aos problemas inerentes à geração e distribuição da chave, assim como à necessidade de utilizar um "verdadeiro" gerador de números aleatórios.

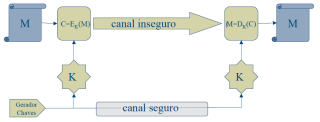


Cifra simétrica sequencial (*stream cipher*)

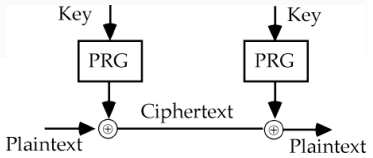
- A ideia base das *stream cipher* consiste em "aproximar" a cifra One-Time-Pad por intermédio de um gerador de chaves (que produz uma sequência de chave a partir de uma chave de comprimento fixo).
  - Utiliza uma chave pequena (e.g., 128 bits);
  - A partir dessa chave, é gerada uma *keystream* pseudoaleatória que pode ser combinada com o *plaintext*, de modo similar à cifra One-Time-Pad;
  - Esta *keystream* é (i) pseudoaleatória (não totalmente aleatória), (ii) o seu processo de geração tem de ser reprodutível (i.e., é determinístico, e pode ser visto como uma máquina de estados finitos) e, (iii) cíclica (o período é o comprimento da *keystream* antes de se começar a repetir).
    Esta simplificação em relação à cifra One-Time-Pad tem uma consequência: a prova matemática de segurança da cifra deixa de ser válida para as *stream cipher*.



- O texto é processado (cifrado/decifrado) "símbolo a símbolo" (i.e, bit a bit, byte a byte, ...).
- Tendem a ser muito eficientes e facilmente implementáveis em *hardware*.

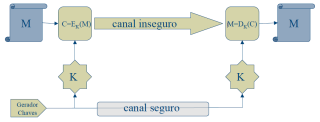


Cifra simétrica sequencial (*stream cipher*)

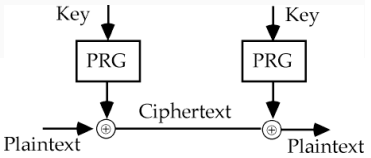


- A segurança das *stream cipher* depende de:
  - Tamanho do período , i.e., tamanho do comprimento da *keystream* antes de se começar a repetir (maior = melhor).
  - Impossibilidade de recuperar a chave da cifra ou o estado interno, a partir da *keystream*. Isto deve ser válido para todas as chaves (não devem existir chaves fracas), mesmo que o atacante possa conhecer ou escolher determinados *plaintext* ou *ciphertext.*
  - Nunca ser reutilizada a *keystream*.
  - O estado inicial nunca ser repetido (consequência de reutilização da *keystream*).

  Chave é muitas vezes combinada com um *initialization vector* (IV).
- Utilização:
  - Em aplicações, onde o tamanho do plaintext não é conhecido, como por exemplo em ligações seguras *wireless*;
  - Militar, onde a *cipher stream* pode ser gerada numa "caixa" separada sujeita a medidas estritas de segurança, ligada a outros dispositivos (e.g., rádio).



Cifra simétrica sequencial (*stream cipher*)



Existem vários tipos de cifras simétricas, entre as quais

- Cifra simétrica síncrona
  - *Keystream* é independente do *plaintext*/*ciphertext*.
  - A perca/inserção de bits no *ciphertext* determinam a "perca de sincronismo" o que significa que ao decifrar, toda a mensagem a partir desse ponto é corrompida/perdida.
  - Erros (alterações de bits) só alteram a posição correspondente da mensagem original.

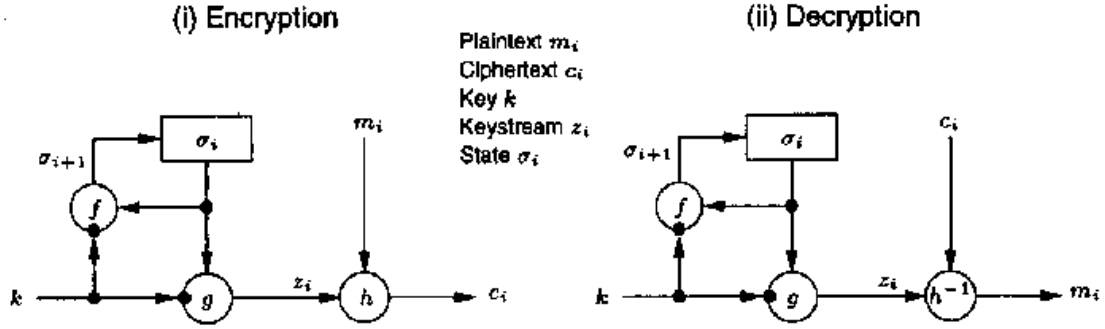
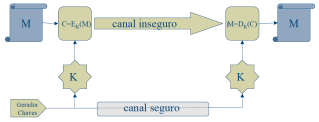
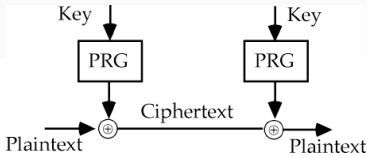


Figure 6.1: *General model of a synchronous stream cipher.*



Cifra simétrica sequencial (*stream cipher*)



Existem vários tipos de cifras simétricas, entre as quais

- Cifra simétrica auto-sincronizável
  - Cada bit do *keystream* é calculado a partir dos últimos *n* bits do *ciphertext* (e da chave, naturalmente)
  - Introduz-se um prefixo de *n* bits aleatórios no texto limpo para permitir sincronização da recepcão.
  - Caso exista erro de transmissão (omissão/inserção de bits no *ciphertext*), ao fim de *n* bits, a decifragem sincroniza. Ou seja, o efeito de um erro está limitado à perca de *n* bits na recuperação do *plaintext*.
  - Problema: vulnerável a ataques por repetição (o intruso pode reenviar uma porção do *ciphertext*)

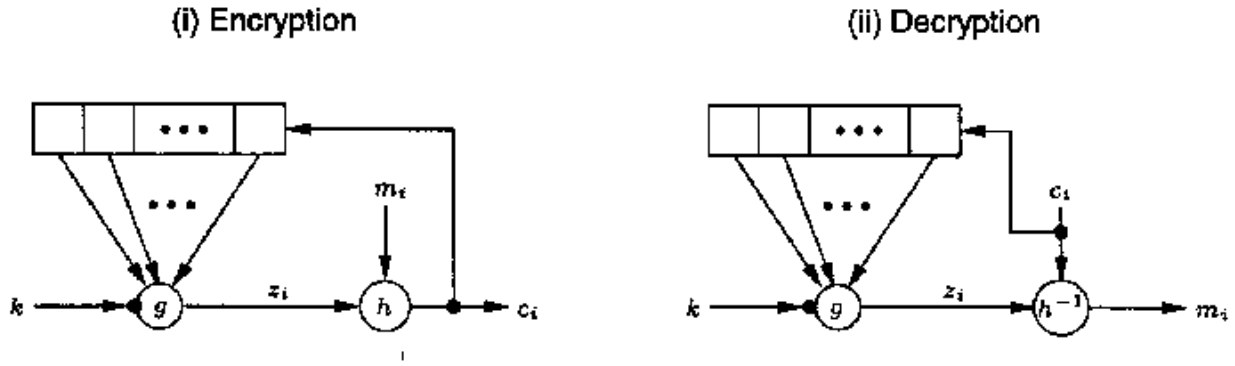
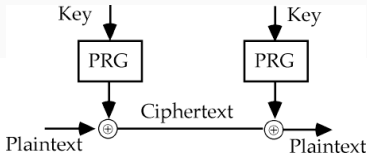


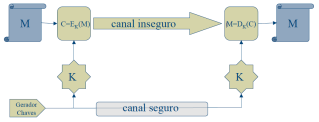
Figure 6.3: *General model of a self-synchronizing stream cipher.*

Cifra simétrica sequencial (*stream cipher*)

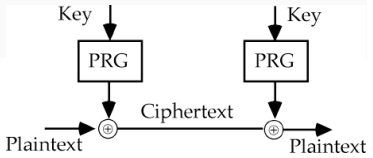


Algumas *stream cipher*:

- A5 (A5/1; A5/2)
  - Utilizada no standard europeu GSM de comunicações móveis (2G);
  - Tamanho da chave: 56 ou 64 bits;
  - Vetor de inicialização: 22 (no 2G) ou 114;
  - Já quebrada!!! (2^{40}, 32 Gb);
  - Vulgarmente reconhecida como um "bom desenho" mas propositadamente fraco em termos de segurança.
- RC4 (ArcFour)
  - Cifra desenvolvida por Ron Rivest (RSA Labs). Originalmente "*trade secret*", mas algoritmo foi divulgado por um *post* anónimo na *newsnet* (descoberto por engenharia reversa).
  - Vocacionada para ser executado em Software com operações ao nível do byte.
  - Tamanho da chave: 8 a 2048 bit (usualmente 40 a 256)
  - Vetor de inicialização: não tem;
  - Já quebrada!!! (2^{13} ou 2^{33});

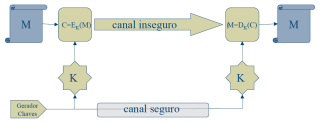


Cifra simétrica sequencial (*stream cipher*)



Algumas *stream cipher*:

- Salsa20 / Chacha20
  - Tamanho da chave: 256 bits;
  - Vetor de inicialização: 64 bit pseudo-aleatório + 64 bit *stream position*;
  - **Chacha20** é evolução (melhorada) da Salsa20, e é uma das duas cifras escolhidas para a encriptação dos novos protocolos de transporte, nomeadamente o TLS 1.3
- Rabbit
  - Tamanho da chave: 128 bits;
  - Vetor de inicialização: 64 bit;
- HC-256
  - Cifra vocacionada para "*bulk encryption*";
  - muito pesada numa fase de pré-processamento, mas rápida na cifra;
  - Tamanho da chave: 256 bits;
  - Vetor de inicialização: 256 bit.



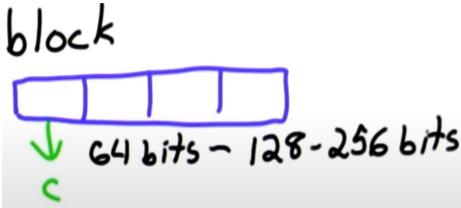
Tópicos

- Parte III: Segurança da Informação
- **Parte IV: Cifras simétricas**
  - Cifras sequenciais (*Stream ciphers*)
  - **Cifras por blocos (*Block ciphers*)**
  - Utilização
- Parte V: Funções de sentido único

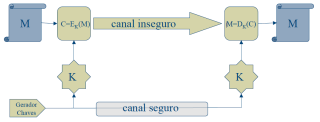
Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Cifra simétrica por blocos (*block cipher*)

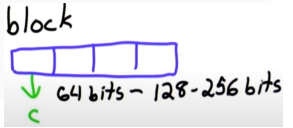
- Na *block cipher*, a unidade de *plaintext* que se vai cifrar é um bloco de dados, com um comprimento fixo (típico) de 64, 128 ou 256 bit.



- O *plaintext* é partido em blocos com o comprimento requerido.
- Muitas vezes, para que o último bloco tenha o comprimento requerido, os bits em falta são preenchidos de acordo com regra pré-estabelecida (*padding*).
- Conceptualmente, a *block cipher* corresponde a uma permutação a operar num alfabeto enorme (e.g. em blocos de 128 bit existirão $(2^{128})!$ possíveis permutações).



Cifra simétrica por blocos (*block cipher*)



Formalmente, uma *block cipher*

- É constituída por dois algoritmos, um para encriptação, *E*, e outro para decifra, *D*. Ambos têm dois parâmetros: um bloco com tamanho de *n* bits, e uma chave com tamanho de *k* bits; e ambos devolve um bloco com tamanho de *n* bits. O algoritmo de decifra, *D*, é definido como a função inversa da encriptação, i.e., $D = E^{-1}$.
- É especificada por uma função de encriptação

$$E_K(P) := E(K,P) : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$$

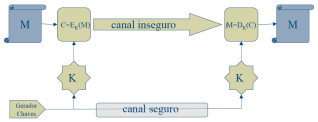
que tem como input uma chave *K* de tamanho *k* (*key size*), e uma string de bits *P* (*plaintext*) com tamanho *n* (*block size*), e devolve uma string *C* (*ciphertext*) com *n* bits.

Para cada *K*, a função $E_K(P)$ tem de ter uma inversa no domínio $\{0,1\}^n$. A inversa de *E* é definida como a função

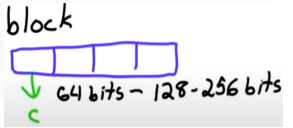
$$E_K^{-1}(C) := D_K(C) = D(K,C) : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$$

e para uma chave *K*, um ciphertext *C* e um plaintext *P*, verifica-se que

$$\forall K : D_K(E_K(P)) = P.$$

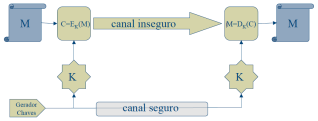


Cifra simétrica por blocos (*block cipher*)

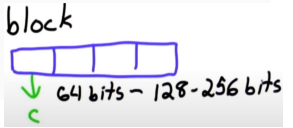


Historicamente, a *block cipher* é baseada no trabalho de Claude Shannon ("*A Mathematical Theory of Cryptography*", 1945, e "*Communication Theory of Secrecy Systems*", 1949).

- Conceito de *product cipher* iterado
  - Um *product cipher* é uma sequência de transformações simples, tais como a substituição (S-box), permutação (P-box) e aritmética modular.
  - *Product cipher* iterados efetuam a encriptação em múltiplos ciclos, em que cada um utiliza uma "sub-chave" diferente, derivada da chave original.
- Propriedades de Difusão e Confusão
  - Difusão significa que cada bit do *plaintext* deve afetar o maior número de bits do *ciphertext*. Desta forma escondemos propriedades estatísticas da mensagem.
  - Confusão significa que cada bit do *ciphertext* deve ser uma função complexa dos bits do *plaintext*. Desta forma torna-se "complicada" a relação entre propriedades estatísticas do *ciphertext* face às propriedades do *plaintext*.

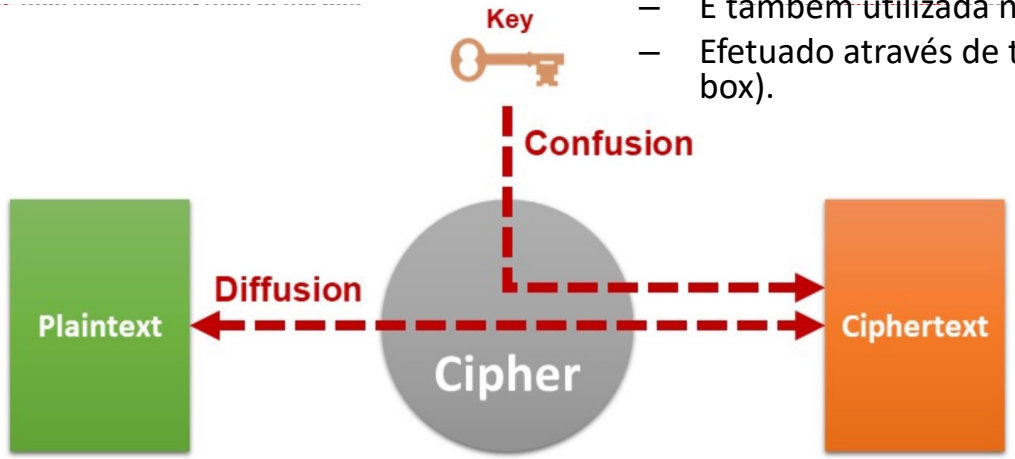


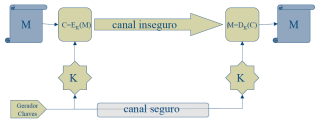
Cifra simétrica por blocos (*block cipher*)



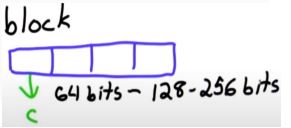
Atualmente, na *block cipher* os conceitos de Difusão e Confusão significam o seguinte:

- Difusão significa que se alterarmos um único bit do *plaintext*, então (estatisticamente) metade dos bits no *ciphertext* são alterados. Do mesmo modo, se alteramos um bit do *ciphertext*, metade do bits do *plaintext* são alterados.
  - O objetivo é esconder as relações estatísticas entre o *ciphertext* e o *plaintext*.
  - Efetuado através de técnica de permutação (P-box).
- Confusão significa que cada bit do *ciphertext* depende de várias partes da chave, obscurecendo a ligação entre os dois.
  - O objetivo é esconder a relação entre o *ciphertext* e a chave.
  - Torna difícil obter a chave a partir do *ciphertext* e, a alteração de um único bit da chave afeta a maior parte dos bits no *ciphertext*.
  - É também utilizada nas *stream ciphers*.
  - Efetuado através de técnica de substituição (S-box).



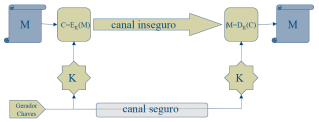


Cifra simétrica por blocos (*block cipher*)

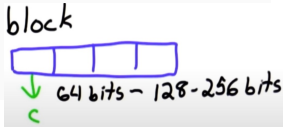


Existem vários modos de operação, i.e., vários modos de particionar uma mensagem em blocos e cifrá-la:

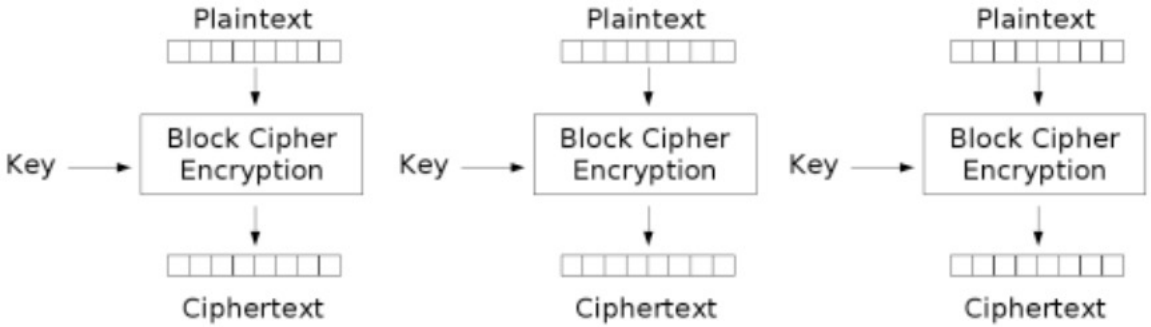
- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Cipher FeedBack (CFB)
- Output FeedBack (OFB)
- Counter Mode (CTR)
- ...



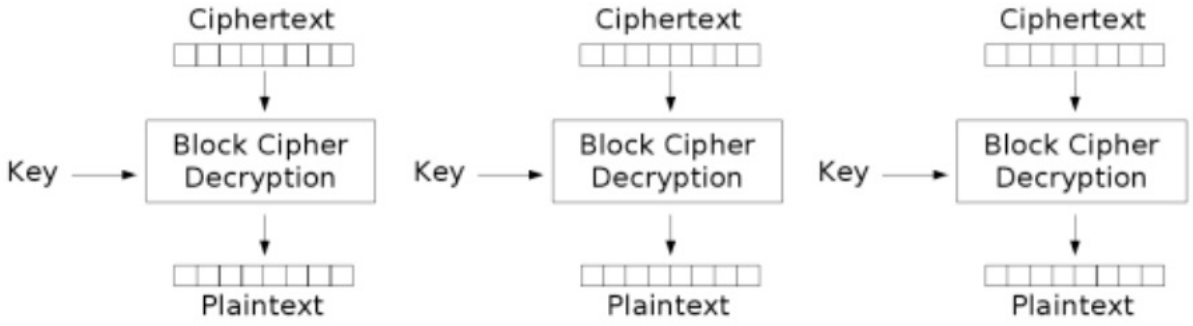
Cifra simétrica por blocos (*block cipher*)



Modos de operação: Electronic Code Book (ECB)

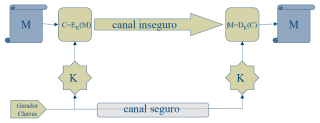


Electronic Codebook (ECB) mode encryption

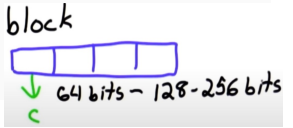


Electronic Codebook (ECB) mode decryption

Imagens de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation



Cifra simétrica por blocos (*block cipher*)



Modos de operação: Electronic Code Book (ECB)

- Método mais simples;
- A mensagem é primeiro dividida em blocos com o tamanho do bloco da cifra (adicionando *padding* ao último bloco, se necessário);
- Cada bloco é cifrado e decifrado de forma independente;
- Usualmente inseguro, porque blocos iguais de *plaintext* geram blocos iguais de *ciphertext* (para a mesma chave), pelo que os padrões na mensagem de *plaintext* são evidentes no *ciphertext*.
- Só deve ser utilizado para cifrar mensagem de um só bloco (ou poucos...).

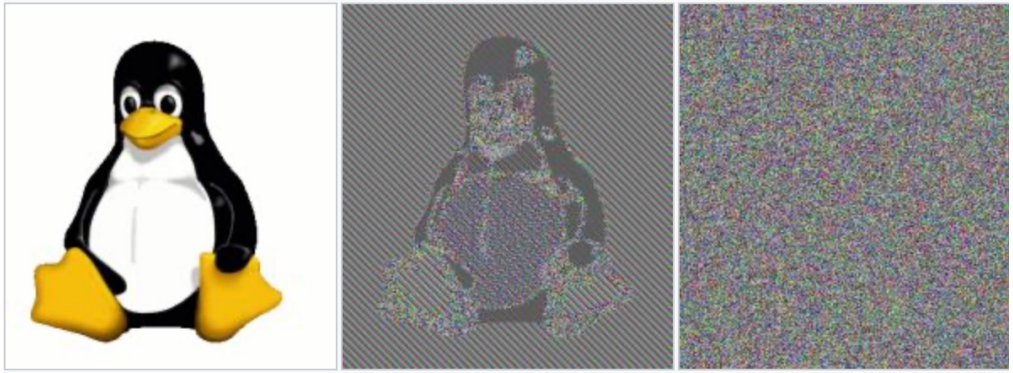
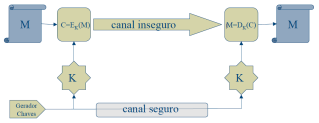
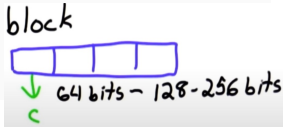


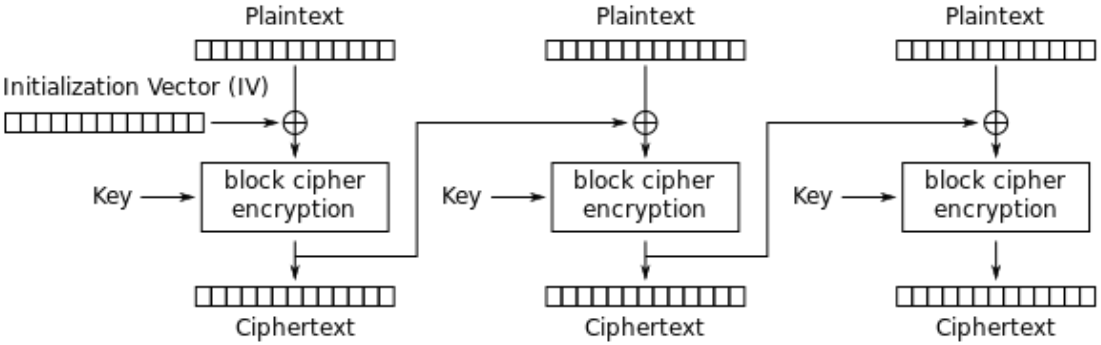
Imagem de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation



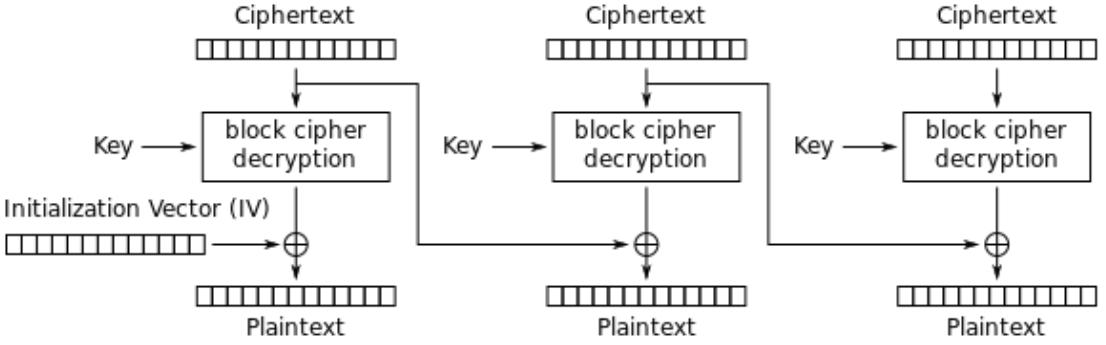
Cifra simétrica por blocos (*block cipher*)



Modos de operação: Cipher Block Chaining (CBC)

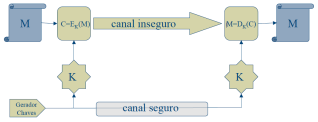


Cipher Block Chaining (CBC) mode encryption

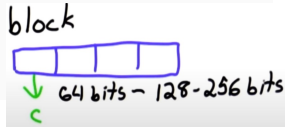


Cipher Block Chaining (CBC) mode decryption

Imagens de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

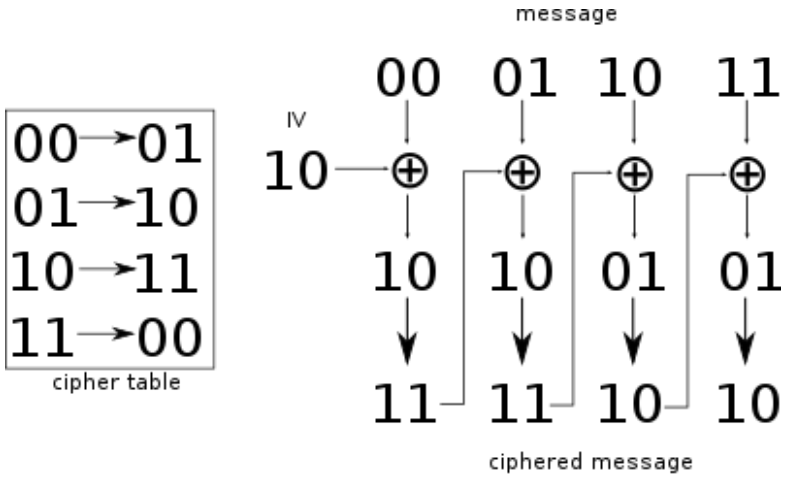


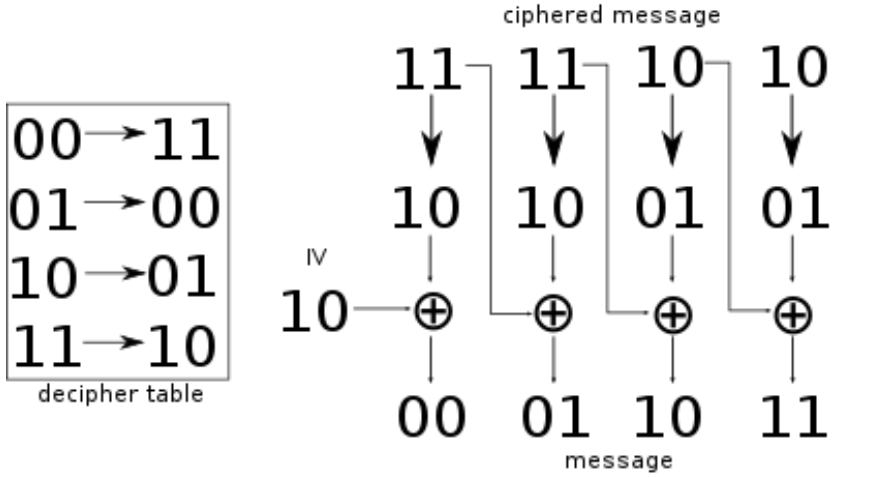
Cifra simétrica por blocos (*block cipher*)



Modos de operação: Cipher Block Chaining (CBC)

- A mensagem é primeiro dividida em blocos com o tamanho do bloco da cifra (adicionando *padding* ao último bloco, se necessário);
- Cada bloco de *plaintext* é XORed com o bloco anterior de *ciphertext*, antes de ser cifrado. Deste modo, cada bloco de *ciphertext* depende de todos os blocos de *plaintext* processados até esse momento;
- É utilizado um "*initialization vector*" (IV) distinto no primeiro bloco, de modo a que cada mensagem seja única.
- Exemplo, com bloco de 2 bits:





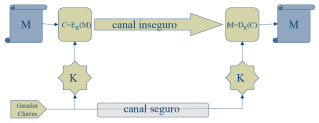
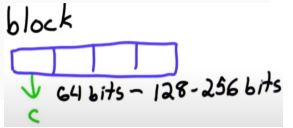
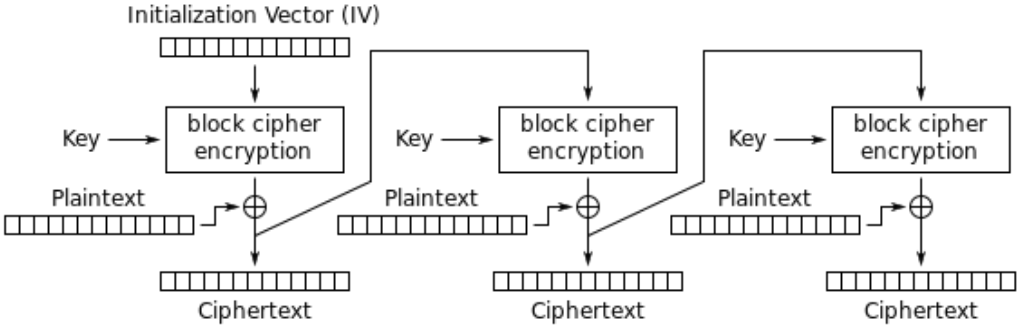


Imagem de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

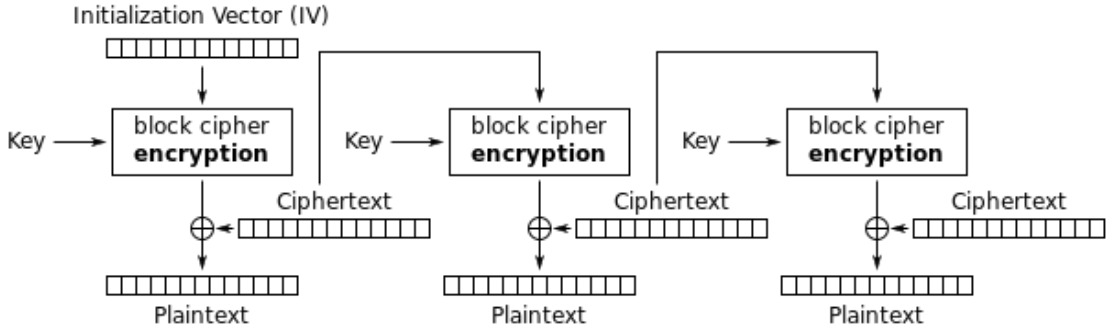
Cifra simétrica por blocos (*block cipher*)



Modos de operação: Cipher FeedBack (CFB)

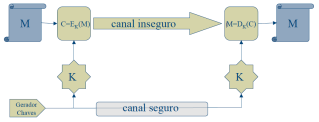


Cipher Feedback (CFB) mode encryption

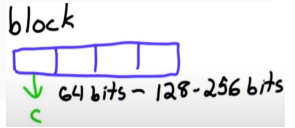


Cipher Feedback (CFB) mode decryption

Imagens de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

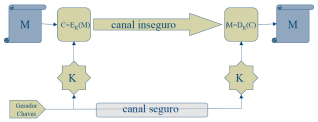


Cifra simétrica por blocos (*block cipher*)

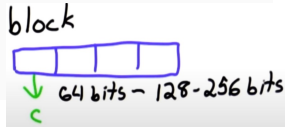


Modos de operação: Cipher FeedBack (CFB)

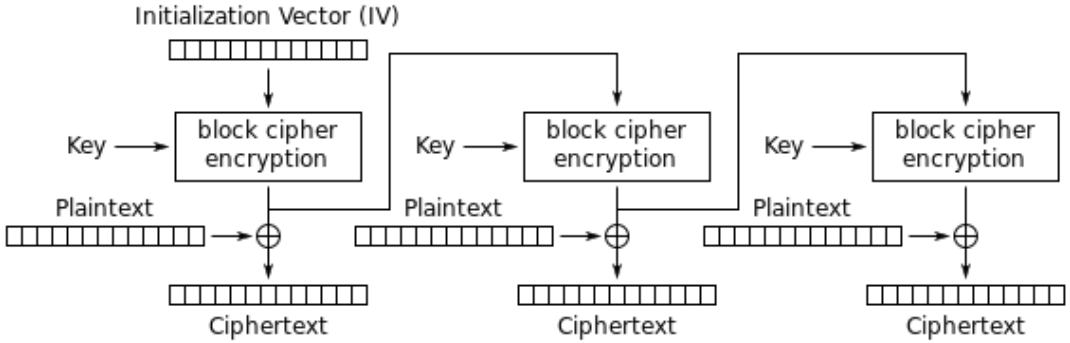
- A mensagem é primeiro dividida em blocos com o tamanho do bloco da cifra (adicionando *padding* ao último bloco, se necessário);
- Cada bloco de *ciphertext* serve de input para a função de cifra do bloco seguinte, sendo o *ciphertext* desse novo bloco o resultado do *plaintext* XORed com o output da função de cifra
  - I.e., este modo implementa uma *stream cipher* auto-sincronizável com uma cifra por blocos.
- É utilizado um "*initialization vector*" (IV) distinto no primeiro bloco;
- *Keystream* depende do IV, chave de cifra e de todo o *plaintext* já cifrado;
- Note-se que se utiliza sempre a operação de "cifrar", quer ao cifrar como ao decifrar.



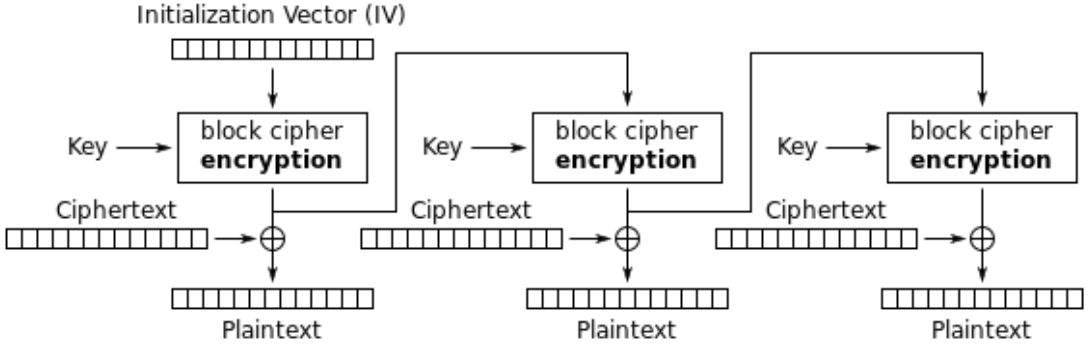
Cifra simétrica por blocos (*block cipher*)



Modos de operação: Output FeedBack (OFB)

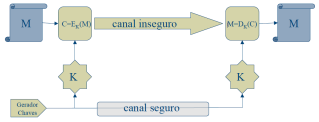


Output Feedback (OFB) mode encryption

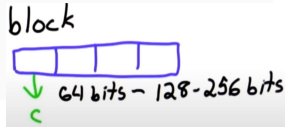


Output Feedback (OFB) mode decryption

Imagens de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

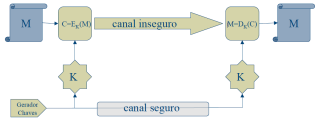


Cifra simétrica por blocos (*block cipher*)

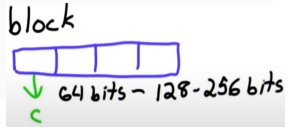


Modos de operação: Output FeedBack (OFB)

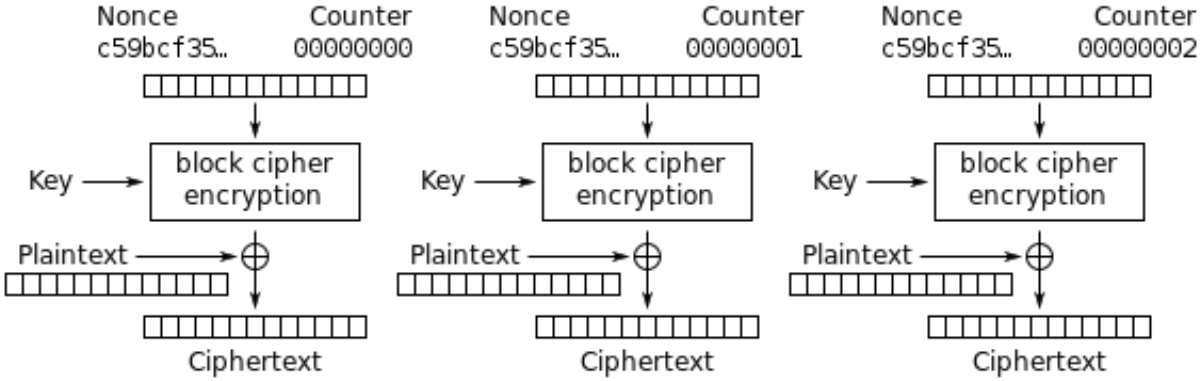
- A mensagem é primeiro dividida em blocos com o tamanho do bloco da cifra (adicionando *padding* ao último bloco, se necessário);
- O *output* da função de cifra de um bloco, serve de input para a função de cifra do bloco seguinte, sendo o *ciphertext* desse novo bloco o resultado do *plaintext* XORed com o output da função de cifra
  - I.e., este modo implementa uma *stream cipher* síncrona com uma cifra por blocos.
- É utilizado um "*initialization vector*" (IV) distinto no primeiro bloco;
- *Keystream* é obtida iterando a cifra sobre um bloco inicial (IV);
- *Keystream* é independente da mensagem (pode assim ser processada independentemente de se ter já disponível a mensagem);
- Note-se que se utiliza sempre a operação de "cifrar", quer ao cifrar como ao decifrar.



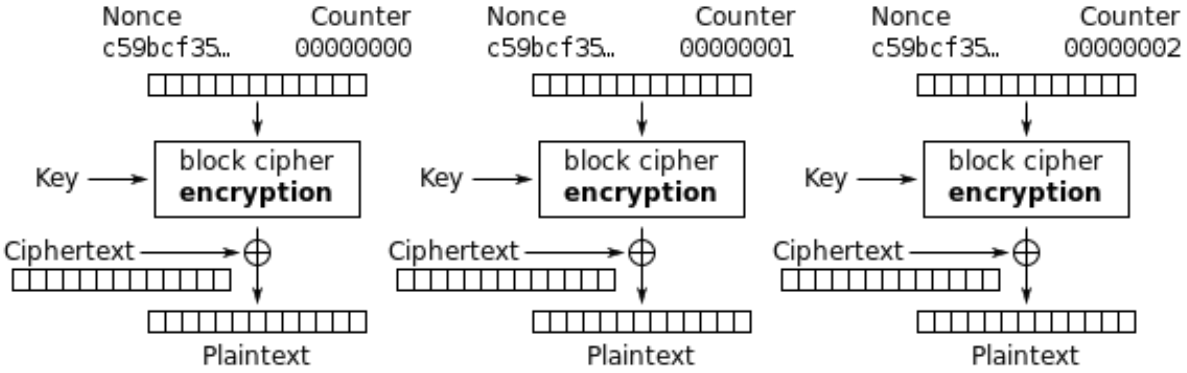
Cifra simétrica por blocos (*block cipher*)



Modos de operação: Counter Mode (CTR)



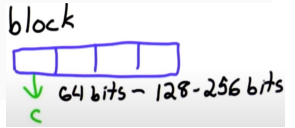
Counter (CTR) mode encryption



Counter (CTR) mode decryption

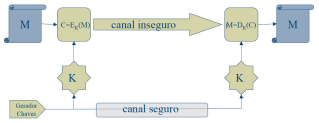
Imagens de https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Cifra simétrica por blocos (*block cipher*)

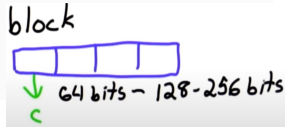


Modos de operação: Counter Mode (CTR)

- A mensagem é primeiro dividida em blocos com o tamanho do bloco da cifra (adicionando *padding* ao último bloco, se necessário);
- A *keystream* é obtida pela cifra sucessiva de um *Nonce* (IV) com um *Counter*.
  - I.e., este modo implementa uma *stream cipher* síncrona com uma cifra por blocos.
- *Nonce* (IV) e *Counter* podem ser conjugados de diferentes formas (concatenados, xored, ...).
- Único requisito para o *Counter* é produzir valores distintos para todos os blocos (o mais simples é ser mesmo implementado como um contador).
- Não impõe dependência entre processamento dos vários blocos (podem ser processados em paralelo; acesso aleatório; ...)
- Note-se que se utiliza sempre a operação de "cifrar", quer ao cifrar como ao decifrar.

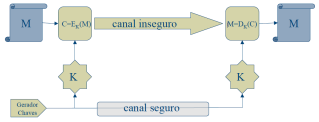


Cifra simétrica por blocos (*block cipher*)

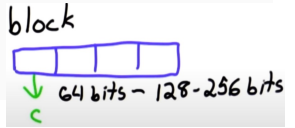


Algumas *block cipher*:

- DES (1976)
  - FIPS PUB 46 Data Encryption Standard (DES);
  - Tamanho da chave: 56 bits;
  - Tamanho do bloco: 64 bits;
  - Já quebrada!!! (em 1998);
- Triple DES (extensão ao DES)
  - Tamanho da chave: 168 bits (3 chaves independentes de 56 bits), mas apenas (equivalente a) 112 bits de segurança;
  - Tamanho do bloco: 64 bits;
- IDEA (1991)
  - Tinha como objetivo substituir o DES/3DES
  - Tamanho da chave: 128 bits;
  - Tamanho do bloco: 64 bits;
- RC5 (1994)
  - Tamanho da chave: 0 a 2040 bits;
  - Tamanho do bloco: 32, 64 ou 128 bits;
  - Muito eficiente;
  - Suscetível a ataques com sucesso, no caso de bloco de 64 bits e 12 ciclos (*product cipher* iterados).

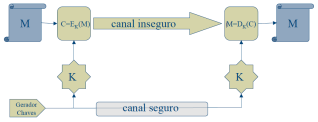


Cifra simétrica por blocos (*block cipher*)

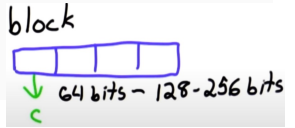


Algumas *block cipher*:

- Rijndael / AES (2001)
  - Uma das cifras que competiram para substituir o DES, e que ganhou a competição pública do NIST (*National Institute of Standards and Technology*), transformando-se no AES (Advanced Encryption Standard)
    - Competição aberta a todos, iniciada em 1997, cujo objetivo era escolher uma cifra para substituir o DES;
    - 15 cifras diferentes submetidas no *Round* 1;
    - Escolhidos 5 finalistas (MARS, RC6, Rijndael, Serpent, Twofish);
    - Vencedor escolhido com base em três critérios: (i) segurança (segurança estimada, já que nenhuma das cifras finalistas possuía uma prova matemática de segurança), (ii) velocidade (implementada em *software* e *hardware*), e (iii) simplicidade.
    - Vencedor foi a cifra Rijndael (desenvolvida por dois criptógrafos belgas), que se passou a chamar de AES.
  - Tamanho da chave: 128, 192 ou 256 bits;
  - Tamanho do bloco: 128 bits;
  - Muito eficiente em software (todas as operação podem ser realizadas por XOR e *lookup* a tabelas);
  - *Block cipher* mais importante, actualmente.

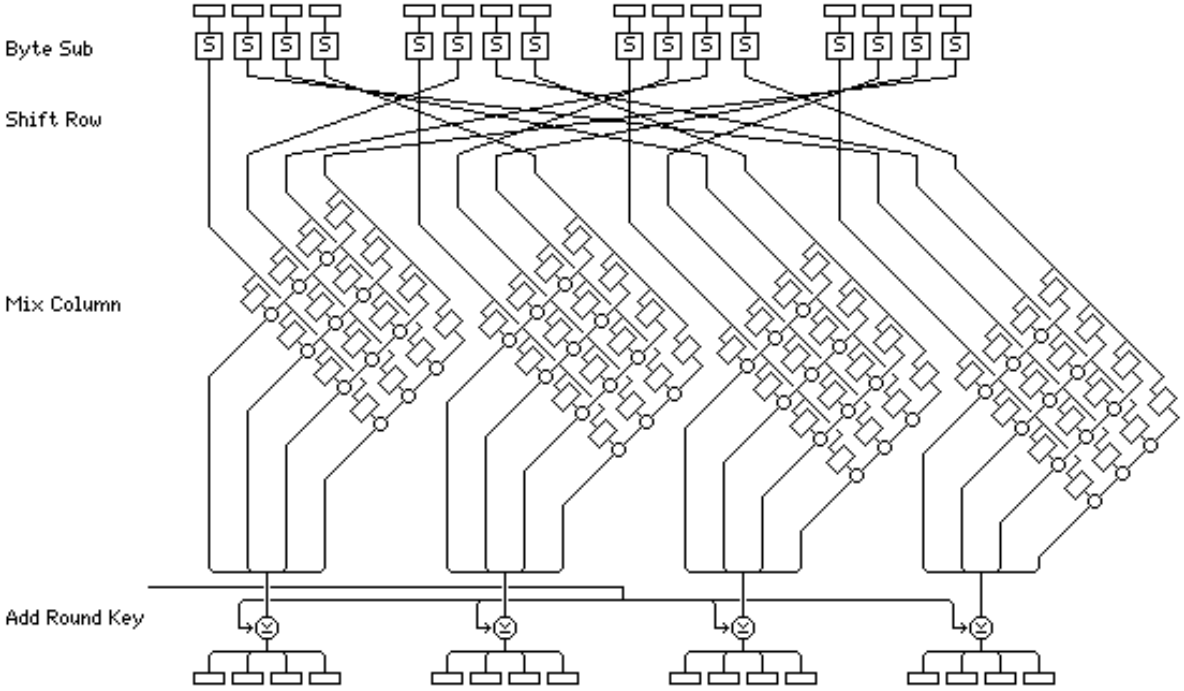


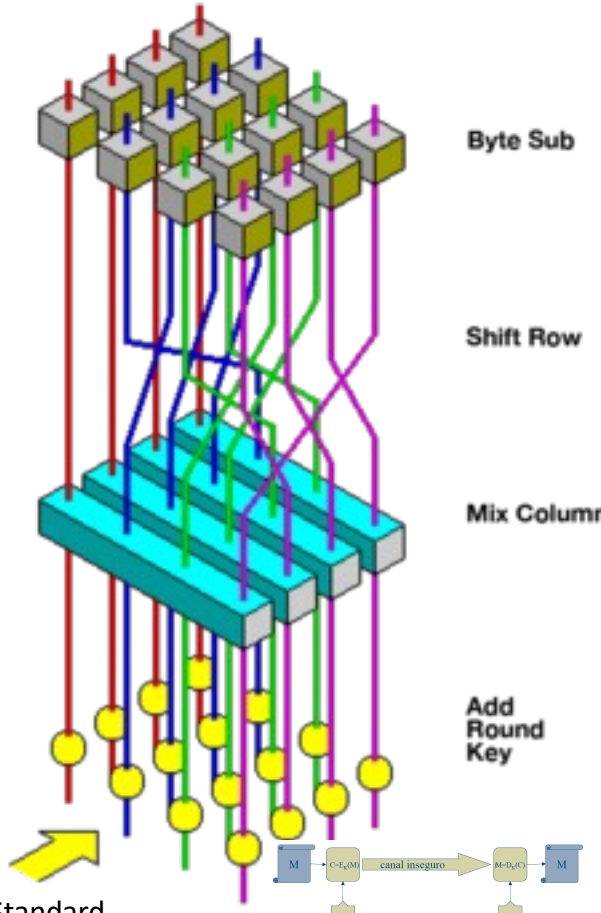
Cifra simétrica por blocos (*block cipher*)



Algumas *block cipher*:

- Rijndael / AES (2001)
  - Visualização de ciclo (*product cipher* iterado) constituído por *Byte Substitution*, *Shift Rows*, *Mix Columns* e *Key Addition.*





Imagens de https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Tópicos

- Parte III: Segurança da Informação
- **Parte IV: Cifras simétricas**
  - Cifras sequenciais (*Stream ciphers*)
  - Cifras por blocos (*Block ciphers*)
  - **Utilização**
- Parte V: Funções de sentido único

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Utilização de cifras simétricas

- As cifras simétricas utilizadas devem ser apropriadas ao seu caso de uso.
- Não necessita (nem deve) desenvolver o código para as cifras simétricas, já que existem bibliotecas/APIs que já disponibilizam o código necessário (i.e., as operações base das cifras simétricas). Por exemplo:
  - Em Python, pode utilizar a cryptography (https://cryptography.io/);
  - Em Javascript ou Node.js pode utilizar o crypto-js (https://www.npmjs.com/package/crypto-js).
  - Em Java, a funcionalidade criptográfica é disponibilizada por duas bibliotecas: *Java Cryptography Architecture* (JCA) e *Java Cryptography Extension* (JCE), disponibilizadas no Java SE.
    - Todas as classes da JCA e JCE são chamadas de ***engines***.
    - Os JCA *engines* estão na java.security package (entre outros, *SecureRandom*, *KeyPairGenerator*, *KeyStore*, *MessageDigest*, *Signature*, *CertificateFactory*, *CertPathBuilder*, *CertStore*), e os da JCE estão na javax.crypto package (entre outros, *Cipher*, *KeyGenerator*, *SecretKeyFactory*, *KeyAgreement*, *Mac*).
    - A JCA e JCE definem todas as operações e objetos criptográficos. Contudo, a implementação dessas funcionalidades está localizada em classes separadas, chamadas de *providers*. Os *providers* implementam a API definida na JCA e JCE, e são responsáveis por fornecer a implementação dos algortitmos criptográficos.
    - Os *providers* podem ser instalados através da configuração do *Java Runtime*: instalar o JAR que contém o *provider*, e ativá-lo através da adição do seu nome ao ficheiro java.security. Em alternativa, os *providers* podem ser instalados durante a execução (através da função Security.addProvider(..)), pela própria aplicação.
    - Existe um conjunto *default* de *providers* da Sun (propriedade da Oracle), nomeadamente SUN, SunJCE, SunPKCS11, …;
    - Existe outro *provider* muito utilizado: Bouncy Castle (https://www.bouncycastle.org/java.html).

Utilização de cifras simétricas

- Exemplo em javascript/node.js, utilizando o crypto-js (pode testar no https://npm.runkit.com/crypto-js)
  - Cifra e decifra AES

```
var cryptoJs = require("crypto-js")

var message = cryptoJs.enc.Hex.parse('00112233445566778899aabbccddeeff');
var key = cryptoJs.enc.Hex.parse('000102030405060708090a0b0c0d0e0f');
var iv = cryptoJs.enc.Hex.parse('101112131415161718191a1b1c1d1e1f');

var encryptedText =  cryptoJs.AES.encrypt(message, key, { iv: iv, mode:
cryptoJs.mode.ECB, padding: cryptoJs.pad.NoPadding });
console.log("Encrpted Text : "+ encryptedText.toString());

var decryptedText = cryptoJs.AES.decrypt(encryptedText, key, { iv: iv, mode:
cryptoJs.mode.ECB, padding: cryptoJs.pad.NoPadding });
console.log("Decrypted Text : "+ decryptedText.toString());
```

Utilização de cifras simétricas

- Exemplo em javascript/node.js, utilizando o crypto-js (pode testar no https://npm.runkit.com/crypto-js)
  - Cifra e decifra Triple DES

```
var cryptoJs = require("crypto-js")

var message = cryptoJs.enc.Hex.parse('101112131415161718191a1b1c1d1e1f');
var key = cryptoJs.enc.Hex.parse('000102030405060708090a0b0c0d0e0f1011121314151617');
var iv = cryptoJs.enc.Hex.parse('08090a0b0c0d0e0f');

var encryptedText =  cryptoJs.TripleDES.encrypt(message, key, { iv: iv, mode: cryptoJs.mode.ECB,
padding: cryptoJs.pad.NoPadding });

var decryptedText = cryptoJs.TripleDES.decrypt(encryptedText, key, { iv: iv, mode: cryptoJs.mode.ECB,
padding: cryptoJs.pad.NoPadding });
console.log("Decrypted Text : "+ decryptedText.toString());
```

Utilização de cifras simétricas – openssl

- O openssl (https://www.openssl.org) é um toolkit (”canivete suíço”) para criptografia e comunicações seguras.
  - Cifra simétricas, utilizando a linha de comando (windows, linux, macos, ...)

```
# cifras disponíveis através da linha de comando, que pode utilizar para cifrar/decifrar
openssl list -cipher-commands

# cifrar com AES 256 bit, em modo CBC
openssl enc -aes-256-cbc -salt -in myfile.txt -out myfile.enc

# decifrar com AES 256 bit, em modo CBC
openssl enc -d -aes-256-cbc -in myfile.enc -out myfile.txt

# cifrar com Triple DES, em modo CBC
openssl enc -des-ede3-cbc -salt -in myfile.txt -out myfile.enc

# decifrar com Triple DES, em modo CBC
openssl enc -d -des-ede3-cbc -in myfile.enc -out myfile.txt
```

Tópicos

- Parte III: Segurança da Informação
- Parte IV: Cifras simétricas
- **Parte V: Funções de sentido único**
  - Funções de Hash criptográficas
  - *Message Authentication Codes* (MAC)
  - *Password-based Key Derivation Functions*
  - *Trapdoor functions*
  - *Utilização*

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Funções de sentido único

- Nas ciências da computação são designadas de **Funções de Sentido Único**, as funções que:
  - possuam um algoritmo eficiente para o seu cálculo, i.e., uma função que é fácil de computar para qualquer input;
  - não disponham de um algoritmo eficiente que calcule uma sua (pseudo)inversa, i.e., difícil de inverter.
  - "fácil" e "difícil" entendidas no contexto da teoria de complexidade computacional, mais especificamente no âmbito da teoria dos problemas de tempo polinomial

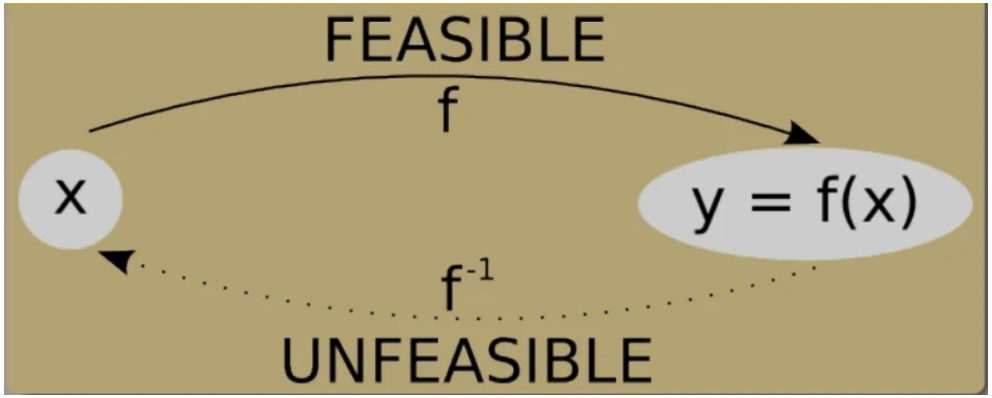


Imagem de https://www.slideshare.net/cpuodzius/seminar-32640647

Funções de sentido único

- Definição teórica
  - Uma função $f : \{0,1\}^* \rightarrow \{0,1\}^*$ é de sentido único,
    - se f pode ser computado por um algoritmo de tempo polinomial,
    - mas qualquer algoritmo aleatório F de tempo polinomial que tente computar a (pseudo-)inversa de f (i.e. f^{-1}) tem uma probabilidade negligenciável de sucesso.

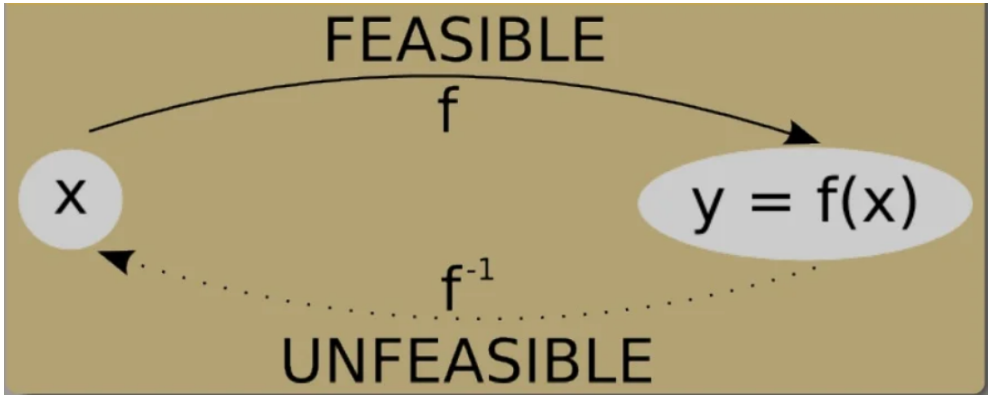
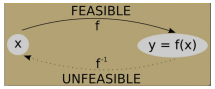


Imagem de https://www.slideshare.net/cpuodzius/seminar-32640647

Funções de sentido único

- A existência de funções de sentido único ainda continua a ser objeto de debate (ou seja, não existe prova matemática que exista).
  - A sua existência provaria que as classes de complexidade P e NP não são iguais, resolvendo uma das mais antigas questões da teoria das ciências de computação
- Existem candidatos a funções de sentido único que resistiram a décadas de intenso escrutínio, e que são essenciais para os sistemas de telecomunicações, comércio eletrónico e banca.
- Por exemplo, conjetura-se (mas não se prova) que as seguintes funções são de sentido único:
  - Problema de fatorização: $f(p, q) = pq$, para número primos p e q, escolhidos aleatoriamente.
  - Problema de logaritmo discreto: $f(p, g, x) = \langle p, g, g^x \pmod p \rangle$, sendo g um gerador de $\mathbb{Z}_p^*$ para determinado número primo p.



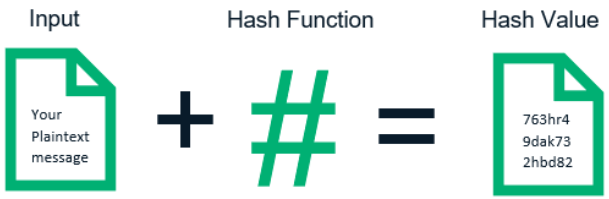
Tópicos

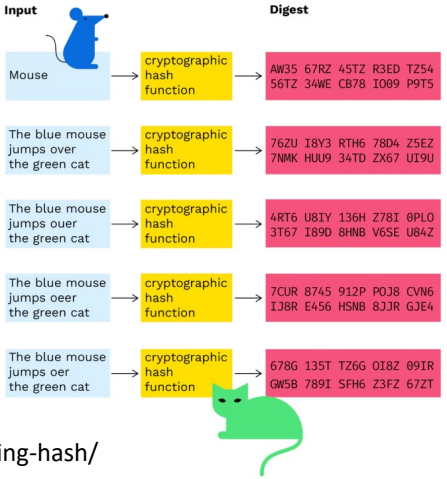
- Parte III: Segurança da Informação
- Parte IV: Cifras simétricas
- **Parte V: Funções de sentido único**
  - **Funções de Hash criptográficas**
  - *Message Authentication Codes* (MAC)
  - *Password-based Key Derivation Functions*
  - *Trapdoor functions*
  - *Utilização*

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Funções de Hash criptográficas

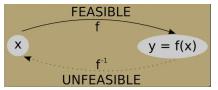
- Um exemplo de aplicação de funções de sentido único são as **funções de hash criptográficas**.
- A sua segurança baseia-se, portanto, em argumentos de natureza de complexidade computacional.
- O objetivo é que mensagens de comprimento arbitrário sejam mapeadas num contradomínio de tamanho fixo.



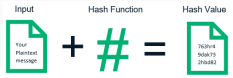


Imagens de https://www.sobyte.net/post/2021-11/string-hash/

- ... mas devem ser "de sentido único" no sentido em que não deve ser possível inverter essa função.
- Exemplos: MD5, SHA-1, SHA-256, RIPEMD-160, SHA-3



Funções de Hash criptográficas – Propriedades



- Os requisitos das funções de hash são normalmente expressos pela seguinte hierarquia de propriedades:
  - **(First) pre-image resistant**: dado um valor de hash *h*, deverá ser inviável conseguir obter uma mensagem *m* tal que hash(*m*) = *h*.

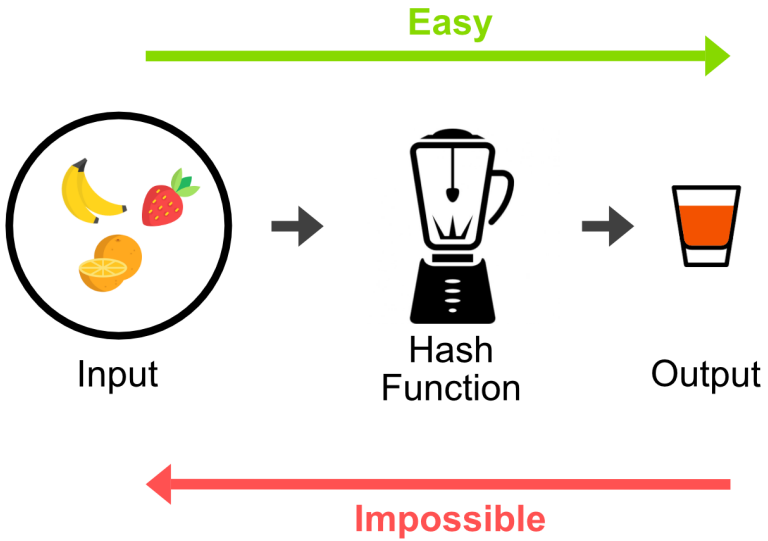
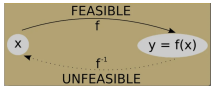


Imagem de https://pjulien.medium.com/blockchain-for-newbies-1-hash-functions-1fb2563bc67c



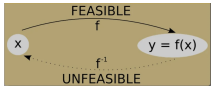
Funções de Hash criptográficas – Propriedades

- Os requisitos das funções de hash são normalmente expressos pela seguinte hierarquia de propriedades:
  - ***(First) pre-image resistant***: dado um valor de hash *h*, deverá ser inviável conseguir obter uma mensagem *m* tal que hash(*m*) = *h*.
  - ***Second pre-image resistant***: dada uma mensagem *m1*, deverá ser inviável obter uma mensagem *m2* distinta de *m1* tal que hash(*m2*) = hash(*m1*).

John : a8cfcd74832004951b4408cdb0a5dbcd8c7e52d43f7fe244bf720582e05241da

john : 96d9632f363564cc3032521409cf22a852f2032eec099ed5967c0d000cec607a

Imagem de https://pjulien.medium.com/blockchain-for-newbies-1-hash-functions-1fb2563bc67c



Funções de Hash criptográficas – Propriedades

- Os requisitos das funções de hash são normalmente expressos pela seguinte hierarquia de propriedades:
  - ***(First) pre-image resistant***: dado um valor de hash *h*, deverá ser inviável conseguir obter uma mensagem *m* tal que hash(*m*) = *h*.
  - ***Second pre-image resistant***: dada uma mensagem *m1*, deverá ser inviável obter uma mensagem *m2* distinta de *m1* tal que hash(*m2*) = hash(*m1*).
  - ***Collision resistant***: não é viável encontrar mensagens distintas *m1* e *m2* tais que hash(*m1*) = hash(*m2*).

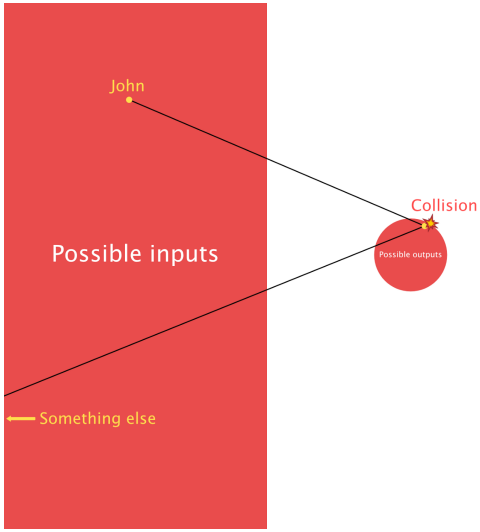
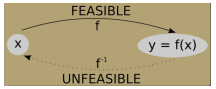
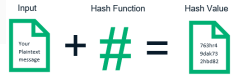


Imagem de https://pjulien.medium.com/blockchain-for-newbies-1-hash-functions-1fb2563bc67c



Funções de Hash criptográficas – *Birthday attack*



- Um resultado famoso da teoria das probabilidades (*birthday attack*) indica-nos que necessitamos de um contradomínio de "tamanho razoável" para se conseguir resistência a colisões.

  Quantas pessoas se tem (em média) que perguntar a idade numa festa de anos para encontrar duas com o mesmo dia de aniversário?

  Testando cerca de sqrt(N) valores aleatórios do domínio dispõe-se de probabilidade superior a ½ de encontrar uma colisão!

- Valores típicos para contradomínios de funções de Hash criptográficas: 128..512 bit.
- ...assim, (de acordo com o *birthday attack*) um ataque por força bruta para encontrar colisões deve testar entre 2^{64} e 2^{256} mensagens.
  - Note que no caso de funções de hash de 256 bit, se todos os computadores do mundo tivessem tentado em conjunto, desde o início do Universo, encontrar uma colisão, a probabilidade de encontrar uma colisão continuaria a ser insignificante.

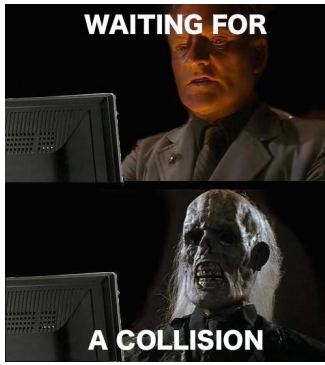
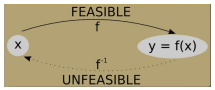
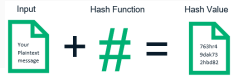


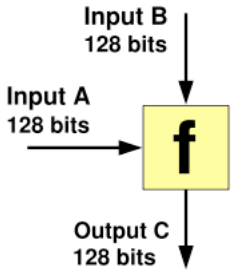
Imagem de https://pjulien.medium.com/blockchain-for-newbies-1-hash-functions-1fb2563bc67c



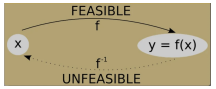
Funções de Hash criptográficas – Desenho



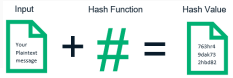
- Um ingrediente fundamental no desenho de funções de hash criptográficas são as funções de compressão.
  - As funções de compressão são funções de sentido único nos seguintes sentidos:
    - conhecendo ambos os inputs, é fácil calcular o output;
    - conhecendo o output, é difícil calcular qualquer um dos input;
    - conhecendo o output e um dos inputs, é difícil calcular o outro input.
  - Devem também ser resistentes a colisões.



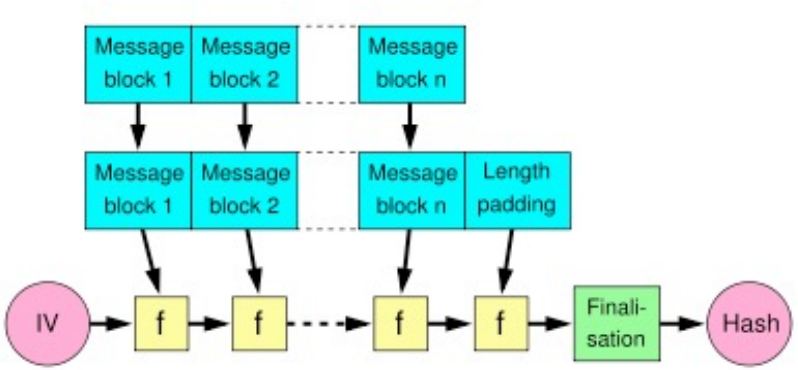
- Podem ser construídas a partir de cifras por blocos... (cumprem metade dos requisitos diretamente - existem construções standard que permitem obter funções de compressão a partir dessas cifras)



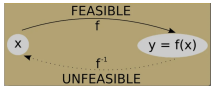
Funções de Hash criptográficas – Desenho



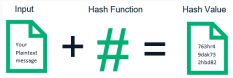
- A generalidade das funções de hash baseia-se na construção de Merkle-Damgard:



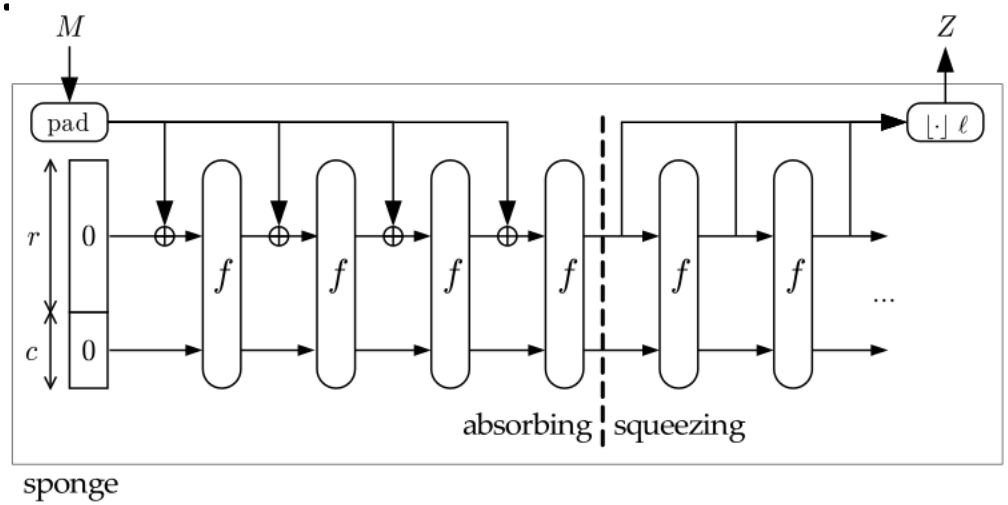
- A função de compressão é responsável por fazer evoluir o estado interno (do estado anterior e de um bloco da mensagem).
- O IV (vetor de inicialização) é normalmente específico do algoritmo (constante).
- É importante o *padding* conter informação relativa ao comprimento da mensagem.
- Demonstra-se que, se *f* é uma função de compressão livre de colisões, a função de hash resultante também o é.



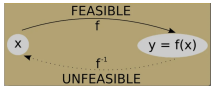
Funções de Hash criptográficas – Desenho



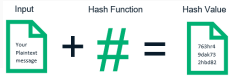
- Funções de hash mais recentes baseiam-se na *sponge construction*:



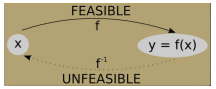
- Permite processar input e gerar output de tamanho arbitrário.
- Apenas uma fração do estado (*bitrate r*) intervém nas fases em que o input é consumido (*absorbed*), e o output é gerado (*squeezed out*).
- Capacidade *c* é determinante na resistência a colisões.



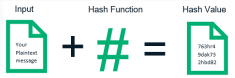
Funções de Hash criptográficas – Aplicações



- Armazenamento de passwords.
- *Commitment schemes* (provas de "posse" de informação).
- Amplificação de entropia (e.g. *Password-based Key Derivation Functions*).
- Como componentes de outras técnicas:
  - MACs
  - Geradores de sequências aleatórias seguras (PRNG)
  - Cifras
  - ...

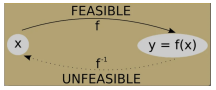


Funções de Hash criptográficas

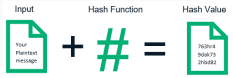


Algumas funções de hash:

- MD5 (RSA Labs – Donald Rivest)
  - Baseada na construção de Merkle-Damgard.
  - Tamanho do contradomínio: 128 bit.
  - Processa a mensagem em blocos de 512 bit.
  - Nos últimos anos tem surgido avanços importantes na sua criptoanálise. Em particular, já foram encontradas colisões.
  - Desaconselhada.
- SHA-1 (1993/1995 – NSA)
  - Baseada na construção de Merkle-Damgard.
  - Tamanho do contradomínio: 160 bit.
  - Otimizada para arquiteturas *big-endian*;
  - Criptoanálise também tem sido alvo de avanços recentes significativos.
  - Desaconselhada para aplicações com requisitos de segurança elevada.

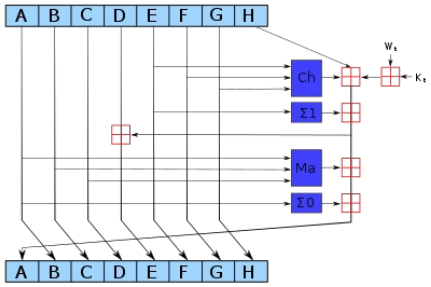


Funções de Hash criptográficas



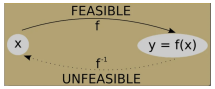
Algumas funções de hash:

- SHA-2 (2001 – NSA)
  - Baseada na construção de Merkle-Damgard.
  - Tamanho do contradomínio: 224 (sha-224), 256 (sha-256), 384 (sha-38) e 512 (sha-512) bits.
  - Implementada/utilizada em aplicações e protocolo de segurança muito utilizados (e.g., TLS, PGP, SSH, S/MIME, e IPsec).



Uma iteração da função de compressão.
In https://en.wikipedia.org/wiki/SHA-2

- SHA-3 (2015 - Keccak)
  - Novo standard para função de hash da NIST (FIPS 202 - 08/2015)
  - Selecionado após concurso o lançado em 2006 (51 candidatos; 14 selecionados para 2ª fase).
  - Algoritmo vencedor: Keccak (instâncias específicas).
  - Utiliza a *sponge construction*, dispondo de variadíssimos parâmetros de configuração que permitem ajustar nível de segurança/eficiência.
  - Tamanho do contradomínio: 224/256/384/512 bit.



Tópicos

- Parte III: Segurança da Informação
- Parte IV: Cifras simétricas
- **Parte V: Funções de sentido único**
  - Funções de Hash criptográficas
  - ***Message Authentication Codes* (MAC)**
  - *Password-based Key Derivation Functions*
  - *Trapdoor functions*
  - *Utilização*

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Message Authentication Codes (MAC)

- As funções de hash, por si só, não garantem a integridade e autenticidade! (... mas quando utilizadas com uma cifra podem permitir estabelecer essas propriedades).
- Um código de autenticação (MAC), pode ser entendido como "uma função de hash com segredo" e visa garantir essas propriedades.

- Exemplo:
  - O emissor utiliza um algoritmo de MAC para produzir um MAC, baseado no segredo (chave) e na mensagem.
  - O MAC e a mensagem são enviados ao destinatário.
  - O destinatário utiliza o mesmo algoritmo de MAC e o mesmo segredo para produzir um MAC.
  - Se o MAC que recebeu for igual ao MAC que produziu, o destinatário pode assumir que a mensagem não foi alterada ou adulterada durante a transmissão (integridade dos dados).
  - Contudo, para evitar ataques de repetição, a mensagem tem que conter informação que garanta que o destinatário perceba se é repetida (e.g., timestamp, número de sequência, ...). Porquê?

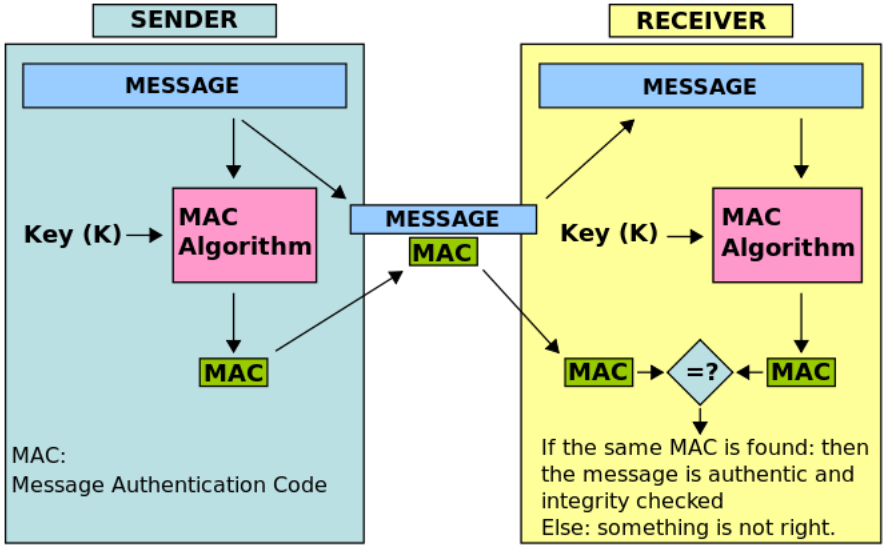


Imagem de https://en.wikipedia.org/wiki/Message_authentication_code

Hash-based MAC (HMAC)

- A forma mais simples de construir um MAC é combinar uma função de hash com um segredo/chave (de forma apropriada) - designada por HMAC.
- Dada uma função de hash *h*, define-se HMAC-*h* como:

  HMAC-h(K,M) = h((K $\oplus$ opad) || h((K $\oplus$ ipad) || M))
  - B = tamanho dos blocos em que opera a função de hash (em bytes)
  - L = tamanho do resultado da função de hash (em bytes)
  - K = chave (tamanho variável entre L e B)
  - ipad = byte 0x36 repetido B vezes
  - opad = byte 0x5C repetido B vezes

- Algumas HMAC:
  - HMAC_MD5, HMAC_SHA1, HMAC_SHA256, ...
  - Note-se que os HMACs são menos afetados pelas colisões do que os algoritmos de hash que utilizam.
  - O uso do HMAC_MD5 é desaconselhado embora ao ataques conhecidos não revelem vulnerabilidades que possam ser aproveitadas (mesmo estando o MD5 comprometido).

MAC derivados de cifras por blocos

- O último bloco de criptograma do modo CBC (da cifra por blocos) pode ser utilizado como um MAC (**CBC-MAC**).

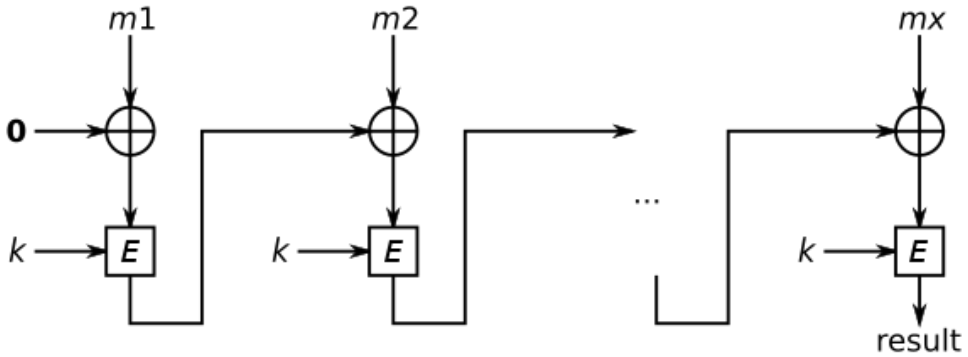


Imagem de https://en.wikipedia.org/wiki/CBC-MAC

  - Note-se que esse método só é seguro para mensagens de comprimento fixo (e este problema não é ultrapassado incluindo informação do comprimento da mensagem no *padding*).
  - Existem modos específicos que ultrapassam as limitações do CBC-MAC (e.g. CMAC – recomendado pelo NIST –, OMAC, PMAC).
- Existem também modos que combinam as garantias de confidencialidade com integridade/autenticação (e.g. OCB, EAX, etc.).

Tópicos

- Parte III: Segurança da Informação
- Parte IV: Cifras simétricas
- **Parte V: Funções de sentido único**
  - Funções de Hash criptográficas
  - *Message Authentication Codes* (MAC)
  - ***Password-based Key Derivation Functions***
  - *Trapdoor functions*
  - *Utilização*

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Password-based Key Derivation Functions (PBKDF)

- Por vezes há necessidade de construir uma chave apropriada para uma dada técnica a partir de chaves fracas (e.g. *passwords* ou *passphrases*).
- Por vezes há a necessidade de "guardar" uma *password* em Base de Dados.
  - Note que **nunca** se guarda uma *password* em claro. Ou se guarda cifrada (no caso especial de necessitar de a obter em claro), ou se guarda a representação (i.e., uma hash utilizando uma PBKDF) da *password* para a validar (no login/autenticação num site web, por exemplo).
- O principal problema é ficar-se vulnerável a ataques de dicionário – o adversário pode "catalogar" todo o espaço de chaves.
- Estratégias para dificultar esses ataques:
  - Considerar fatores aleatórios (designados por *salt*, ou IV). Assim procura-se impedir a pré-computação do dicionário. Na sua forma mais simples, o *salt* é concatenado com o segredo.
  - Aumentar o "peso computacional" da função de derivação da chave. Assim dificulta-se a realização de ataques em tempo real.

PBKDF1 e PBKDF2

- PBKDF1
  - Função de geração de chaves proposta no standard PKCS#5 (*Password-based encryption*).
  - Considera um valor aleatório *S* (salt) e um número de iterações *C* (*iteration count*).
  - Itera uma função de hash *C* vezes aplicada sobre *P*||*S*, em que *P* é a password.
  - Limita o segredo obtido ao tamanho do resultado da função de hash.
- PBKDF2
  - Substitui PBKDF1 no standard PKCS#5.
  - Não limita o segredo ao tamanho da função de Hash.
  - Parametrizada por uma *pseudorandom function* PRF (e.g. HMAC-sha1).
  - Parametrização (para hash de *passwords*):
    - O NIST recomenda um *salt* de 128 bits.
    - Em 2021, o OWASP recomendou a utilização de 720.000 iterações para o PBKDF2-HMAC-SHA1, 310.000 iterações para o PBKDF2-HMAC-SHA256 e 120.000 iterações para o PBKDF2-HMAC-SHA512.

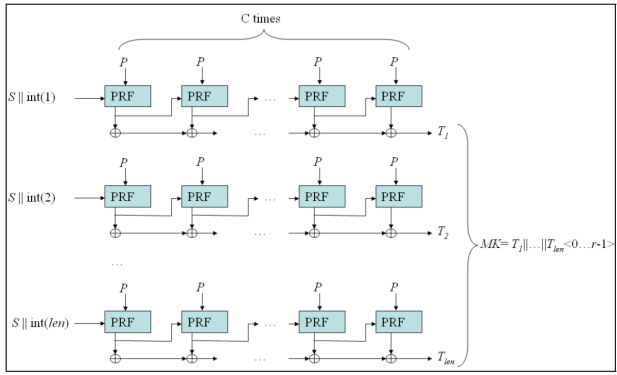


Imagem de https://en.wikipedia.org/wiki/PBKDF2

SCrypt

- SCrypt é uma KDF especificamente desenhada para resistir a "ataques por utilização de hardware dedicado", como os que passam pela utilização de *Application Specific Integrated Circuits* (ASICs) ou *Field Programmable Gate Arrays* (FPGAs).
- Estratégia passa por forçar a utilização de uma quantidade de memória intermédia considerável (que se traduz numa área significativa do respectivo circuito quando implementado em hardware).
  - Internamente, usa repetidamente PBKDF2 para a construção de um estado interno…
  - …juntamente com uma construção/algoritmo (designado de ROMix) que impede a paralelização efectiva do processo.
- Para além da sua utilização como PBKDF, é utilizada em algumas moedas eletrónicas (baseadas em *blockchain*) como algoritmo de *proof-of-work* (e.g., Litecoin, Dogecoin).
- Parametrização (para hash de *passwords*) recomendada pelo OWASP (N é o custo CPU/memória, r é o tamanho do bloco e, p é o grau de paralelismo):
  - $N=2^{16}$ (64 MiB), r=8 (1024 bytes), p=1, ou
  - $N=2^{15}$ (32 MiB), r=8 (1024 bytes), p=2, ou
  - $N=2^{14}$ (16 MiB), r=8 (1024 bytes), p=4, ou
  - $N=2^{13}$ (8 MiB), r=8 (1024 bytes), p=8, ou
  - $N=2^{12}$ (4 MiB), r=8 (1024 bytes), p=15

Argon2

- Argon2 é uma KDF vencedora da *Password Hashing Competition* (2013)
- Disponibiliza 3 modos:
  - Argon2d – maximiza a resistência a ataques por utilização de *hardware* dedicado, como GPU, ASIC ou FPGAs, mas introduz a possibilidade de *side-channel attacks* (ataques baseados na implementação, como por exemplo tempo de execução, consumo de eletricidade, …).
  - Argon2i – está otimizado para resistir aos *side-channel attacks*.
  - Argon2id – é uma versão hibrida das duas anteriores. O RFC 9106 ("*Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*") recomenda o uso do Argon2id se não conhecer as diferenças entre os dois tipos anteriores, ou se considerar que os side-channel attacks são uma ameaça viável.
- Os três modos permitem configurar os parâmetros que controlam:
  - Tempo de execução (t);
  - Memória necessária (m);
  - Grau de paralelismo (p).
- Parametrização (para hash de *passwords*):
  - O OWASP recomenda a utilização do Argon2id com a seguinte parametrização:
    - m=37 MiB, t=1, p=1 ou
    - m=15 MiB, t=2, p=1

Custo de *crackar* uma *password/passphrase*

... mas no limite, tudo depende da

entropia da *password/passphrase* ...

Estimated cost of hardware to crack a password in 1 year.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text
DES CRYPT	< $1	< $1	< $1	< $1	< $1
MD5	< $1	< $1	< $1	$1.1k	$1
MD5 CRYPT	< $1	< $1	$130	$1.1M	$1.4k
PBKDF2 (100 ms)	< $1	< $1	$18k	$160M	$200k
bcrypt (95 ms)	< $1	$4	$130k	$1.2B	$1.5M
scrypt (64 ms)	< $1	$150	$4.8M	$43B	$52M
PBKDF2 (5.0 s)	< $1	$29	$920k	$8.3B	$10M
bcrypt (3.0 s)	< $1	$130	$4.3M	$39B	$47M
scrypt (3.8 s)	$900	$610k	$19B	$175T	$210B

fonte: https://www.tarsnap.com/scrypt/scrypt-slides.pdf

Tópicos

- Parte III: Segurança da Informação
- Parte IV: Cifras simétricas
- **Parte V: Funções de sentido único**
  - Funções de Hash criptográficas
  - *Message Authentication Codes* (MAC)
  - *Password-based Key Derivation Functions*
  - ***Trapdoor functions***
  - *Utilização*

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Trapdoor functions

- Uma ***trapdoor function*** é uma função que é fácil de computar numa direção, mas difícil de computar na direção oposta (i.e., difícil de encontrar a função inversa) sem informação adicional (chamado de "*trapdoor*"/alçapão).
- Do ponto de vista matemático, se *f* é uma *trapdoor function*, então existe alguma informação secreta *t*, de tal modo que fornecendo *f*(*x*) e *t*, é fácil de computar *x*.
- Na criptografia, está ligada ao problema de factorização de números primos (grandes) no RSA
  - Um exemplo com números primos muito pequenos:
    - O número 6895601 (*f*(*x*)) é o produto de dois números primos. Quais?
    - Uma solução de "força bruta" iria tentar dividir 6895601 por vários números primos até encontrar a resposta.
    - Contudo se soubermos que 1931 (*t*) é um dos números, facilmente se encontra a resposta (*x*) que é o resultado de "6895601 ÷ 1931".

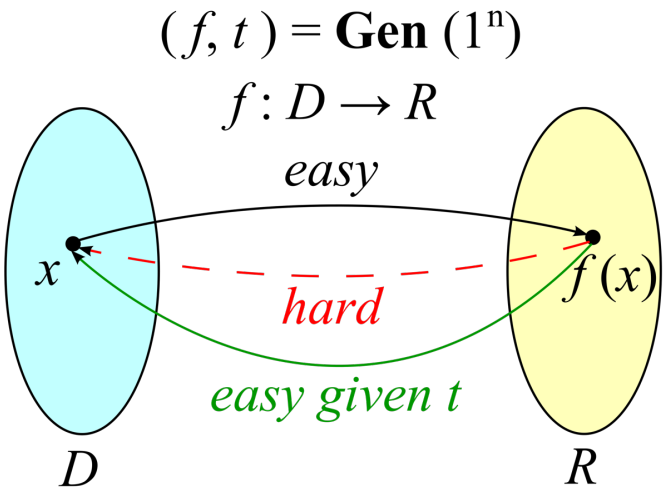


Imagem de https://en.wikipedia.org/wiki/Trapdoor_function

Tópicos

- Parte III: Segurança da Informação
- Parte IV: Cifras simétricas
- **Parte V: Funções de sentido único**
  - Funções de Hash criptográficas
  - *Message Authentication Codes* (MAC)
  - *Password-based Key Derivation Functions*
  - *Trapdoor functions*
  - ***Utilização***

Nota: Apontamentos baseados nos slides de “Tecnologia Criptográfica” do Professor José Bacelar Almeida (com permissão do mesmo)

Utilização de funções de sentido único

- As funções de sentido único utilizadas devem ser apropriadas ao seu caso de uso.
- Não necessita (nem deve) desenvolver o código para as funções de sentido único, já que existem bibliotecas/APIs que já disponibilizam o código necessário (i.e., as operações base das funções de sentido único). Por exemplo:
  - Em Python, pode utilizar a cryptography (https://cryptography.io/) ou a PyCryptodome (https://pycryptodome.readthedocs.io/en/latest/src/introduction.html);
  - Em Javascript ou Node.js pode utilizar o crypto-js (https://www.npmjs.com/package/crypto-js) ou o crypto (https://nodejs.org/api/crypto.html).
  - Em Java, tal como referido para as cifras simétricas, pode utilizar
    - os *default providers* da Sun (propriedade da Oracle), nomeadamente SUN, SunJCE, SunPKCS11, …;
    - O *provider* do Bouncy Castle (https://www.bouncycastle.org/java.html).

Utilização de funções de sentido único

- Exemplo em python, utilizando o cryptography
  - Hash com SHA-2 e SHA-3

```
from cryptography.hazmat.primitives import hashes

digest = hashes.Hash(hashes.SHA256())
digest.update(b"abc")
digest.update(b"123")
digest.finalize()

digest_sha3 = hashes.Hash(hashes.SHA3_256())
digest_sha3.update(b"abc123")
digest_sha3.finalize()
```

Utilização de funções de sentido único

- Exemplo em python, utilizando o cryptography
  - PBKDF2

```
import os
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
# Salts devem ser gerados aleatoriamente, com 128 bits (16 bytes) de acordo com NIST
salt = os.urandom(16)
# Algoritmo de hash
algorithm=hashes.SHA256()
# Iterações – Segundo o OWASP, 310.000 iterações para o PBKDF2-HMAC-SHA256
iterations=310000
# tamanho da hash PBKDF2 da password
length=32
# derivação
kdf = PBKDF2HMAC(algorithm=algorithm, length=length, salt=salt, iterations=iterations)
# password
password = b"a melhor password do mundo"
# hash derivada da password
keyhash = kdf.derive(password)

# verificar
kdf2verify = PBKDF2HMAC(algorithm=algorithm, length=length, salt=salt, iterations=iterations)
kdf2verify.verify(password, keyhash)
```

Utilização de funções de sentido único – openssl

- O openssl (https://www.openssl.org) é um toolkit (”canivete suíço”) para criptografia e comunicações seguras.
  - Funções de sentido único, utilizando a linha de comando (windows, linux, macos, ...)

```
# funções de hash disponíveis através da linha de comando
openssl list -digest-commands

# SHA256 de um conjunto de ficheiros
openssl dgst -sha256 myfile.*

# HMAC-SHA256 de um conjunto de ficheiros
openssl dgst -sha256 -hmac chave_para_o_hmac *.pdf
```