**Criptografia Simétrica**

Neste capítulo, você será introduzido aos conceitos, à teoria e aos aspectos práticos da criptografia simétrica. Será dado mais foco aos elementos que são especificamente relevantes no contexto da tecnologia blockchain. Este capítulo fornecerá os conceitos necessários para entender o material abordado nos capítulos posteriores.

Você também será introduzido a aplicações de algoritmos criptográficos para que possa obter experiência prática na implementação de funções criptográficas. Para isso, a ferramenta de linha de comando OpenSSL é utilizada.

Este capítulo abordará os seguintes tópicos:

* Introdução à criptografia
* Primitivas criptográficas
* Padrão de Criptografia Avançada (AES)

Para os exercícios neste capítulo, a instalação do OpenSSL é necessária. Na distribuição Linux Ubuntu, o OpenSSL geralmente já está disponível. No macOS, o LibreSSL já está disponível. No entanto, ele pode ser instalado usando o seguinte comando:

$ sudo apt-get install openssl

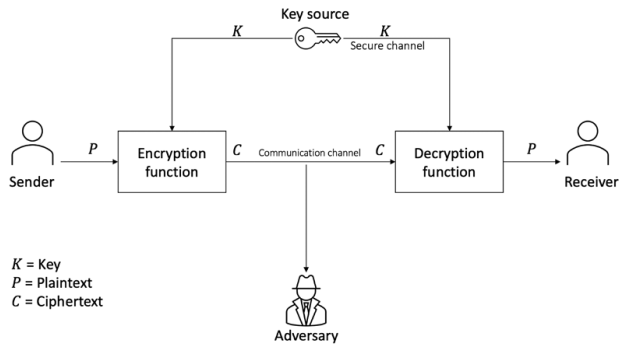
Os exemplos neste capítulo foram desenvolvidos usando o OpenSSL 3.0.1. Recomenda-se o uso dessa versão específica, já que todos os exemplos do capítulo foram desenvolvidos e testados com ela. Se você estiver usando uma versão diferente da versão 3, os exemplos podem ainda funcionar, mas isso não é garantido.

**Introdução à criptografia**  
Criptografia é a ciência de tornar informações seguras na presença de adversários. Cifras são algoritmos usados para criptografar ou descriptografar dados, de modo que, se forem interceptados por um adversário, os dados sejam ininteligíveis para ele sem a descriptografia, a qual requer uma chave secreta.

A criptografia é usada principalmente para fornecer o serviço de confidencialidade. Por si só, ela não pode ser considerada uma solução completa; em vez disso, serve como um bloco de construção crucial dentro de um sistema de segurança mais amplo para resolver um problema de segurança. Por exemplo, proteger um ecossistema blockchain requer diversas primitivas criptográficas diferentes, como funções de hash, criptografia de chave simétrica, assinaturas digitais e criptografia de chave pública, que discutiremos no próximo capítulo.

Além do serviço de confidencialidade, a criptografia também fornece outros serviços de segurança, como integridade, autenticação (autenticação de entidade e autenticação da origem dos dados) e não repúdio. Adicionalmente, a responsabilização (accountability) é fornecida, o que é um requisito em muitos sistemas de segurança.

Um modelo criptográfico genérico é mostrado no diagrama a seguir:



**Figura 3.1: Um modelo genérico de criptografia e descriptografia**  
No diagrama anterior, P, C e K representam, respectivamente, o texto plano (plaintext), o texto cifrado (ciphertext) e a chave (dados usados para criptografar o texto plano e descriptografar o texto cifrado).

**Serviços fornecidos pela criptografia**  
Mencionamos o fato de que a criptografia fornece vários serviços. Na seção seguinte, apresentaremos esses serviços.

**Confidencialidade** é a garantia de que a informação está disponível apenas para entidades autorizadas.

**Integridade** é a garantia de que a informação pode ser modificada apenas por entidades autorizadas.

**Autenticação** fornece garantias sobre a identidade de uma entidade ou a validade de uma mensagem. Existem dois tipos de mecanismos de autenticação, a saber, autenticação de entidade e autenticação da origem dos dados, que são discutidos nas seções a seguir.

**Autenticação de entidade** é a garantia de que uma entidade está atualmente envolvida e ativa em uma sessão de comunicação. Tradicionalmente, os usuários recebem um nome de usuário e uma senha que são usados para obter acesso às várias plataformas com as quais estão trabalhando. Essa prática é conhecida como autenticação de fator único (*single-factor authentication*), já que há apenas um fator envolvido, ou seja, algo que você sabe — isto é, a senha e o nome de usuário.

Esse tipo de autenticação não é muito seguro por uma variedade de razões, por exemplo, vazamento de senhas; portanto, fatores adicionais são comumente usados agora para fornecer melhor segurança. O uso de técnicas adicionais para identificação do usuário é conhecido como autenticação de múltiplos fatores:

* O primeiro método usa **algo que você tem**, como um token de hardware ou um cartão inteligente. Nesse caso, um usuário pode usar um token de hardware além das credenciais de login para obter acesso a um sistema. Um usuário que tenha acesso ao token de hardware e saiba as credenciais de login poderá acessar o sistema. Se o token de hardware estiver inacessível (por exemplo, perdido ou roubado) ou se a senha de login for esquecida pelo usuário, o acesso ao sistema será negado. O token de hardware não terá qualquer utilidade por si só, a menos que a senha de login (algo que você sabe) também seja conhecida e usada em conjunto com o token de hardware.
* O segundo método usa **algo que você é**, o que utiliza características biométricas para identificar o usuário. Com este método, a impressão digital, retina, íris ou geometria da mão de um usuário é usada para garantir que o usuário estava de fato presente durante o processo de autenticação. No entanto, é necessária uma implementação cuidadosa para garantir um alto nível de segurança, pois algumas pesquisas sugeriram que sistemas biométricos podem ser contornados sob condições específicas.

**Autenticação da origem dos dados** é outro tipo de autenticação, que também é conhecido como autenticação de mensagem. É a garantia de que a origem da informação foi de fato verificada. A autenticação da origem dos dados garante a integridade dos dados porque, se uma fonte for corroborada, então os dados não devem ter sido alterados. Vários métodos são usados para este tipo de autenticação, como códigos de autenticação de mensagem (MACs) e assinaturas digitais, que abordaremos mais adiante neste capítulo e no próximo.

Outra garantia importante fornecida pela criptografia é o **não repúdio**. Trata-se da garantia de que uma entidade não pode negar um compromisso ou ação anterior ao se apresentar evidência criptográfica incontestável. Essa propriedade é essencial em situações controversas nas quais uma entidade nega as ações realizadas, por exemplo, a realização de um pedido em um sistema de comércio eletrônico. O não repúdio tem sido uma área ativa de pesquisa por muitos anos. Disputas em transações eletrônicas são uma questão comum, e há uma necessidade de resolvê-las para aumentar os níveis de confiança dos consumidores nesses serviços.

O protocolo de não repúdio geralmente é executado em uma rede de comunicações, e é usado para fornecer evidência de que uma ação foi realizada por uma entidade (originadora ou receptora) na rede. Nesse contexto, dois modelos de comunicação podem ser usados para transferir mensagens do originador A para o destinatário B:

* Uma mensagem é enviada diretamente do originador A para o destinatário B
* Uma mensagem é enviada para um agente de entrega pelo originador A, que então entrega a mensagem ao destinatário B

Os requisitos principais de um protocolo de não repúdio são **justiça**, **efetividade** e **pontualidade**. Em muitos cenários, há múltiplos participantes envolvidos em uma transação. Por exemplo, em sistemas de negociação eletrônica, podem haver agentes de compensação, corretores, negociadores e outras entidades que podem estar envolvidas em uma única transação. Para lidar com esse problema, foram desenvolvidos protocolos de não repúdio de múltiplas partes (*multi-party non-repudiation – MPNR*).

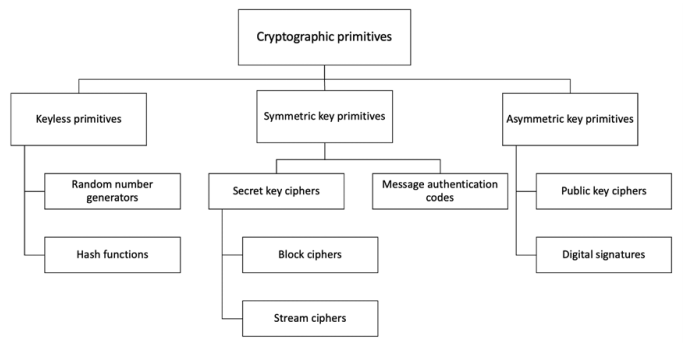
**Responsabilização** (*Accountability*) é a garantia de que ações que afetam a segurança possam ser rastreadas até a parte responsável. Isso normalmente é fornecido por mecanismos de registro (*logging*) e auditoria em sistemas onde uma auditoria detalhada é exigida devido à natureza do negócio, por exemplo, em sistemas de negociação eletrônica. Registros detalhados são vitais para rastrear as ações de uma entidade, como quando uma negociação é realizada em um registro de auditoria com data e carimbo de hora e a identidade da entidade é gerada e salva no arquivo de log. Esse arquivo de log pode, opcionalmente, ser criptografado e fazer parte do banco de dados ou ser um arquivo de log de texto ASCII autônomo em um sistema.

Para fornecer todos os serviços discutidos nesta seção, diferentes primitivas criptográficas são usadas, que são apresentadas na próxima seção.

**Primitivas criptográficas**

Primitivas criptográficas são os blocos de construção básicos de um protocolo ou sistema de segurança. Um protocolo de segurança é um conjunto de etapas tomadas para atingir os objetivos de segurança exigidos, utilizando mecanismos de segurança apropriados. Vários tipos de protocolos de segurança estão em uso, como protocolos de autenticação, protocolos de não repúdio e protocolos de gerenciamento de chaves.

A taxonomia das primitivas criptográficas pode ser visualizada como mostrado aqui:



**Figura 3.2: Primitivas criptográficas**

Como mostrado no diagrama de taxonomia de primitivas criptográficas anterior, a criptografia é dividida principalmente em três categorias: primitivas sem chave (*keyless*), primitivas de chave simétrica e primitivas de chave assimétrica.

**Primitivas sem chave**

Nesta seção, apresentaremos duas primitivas sem chave, a saber, **números aleatórios** e **funções de hash**.

**Números aleatórios**

A aleatoriedade fornece um elemento indispensável para a segurança de protocolos criptográficos. Ela é usada para a geração de chaves e em algoritmos de criptografia. A aleatoriedade garante que as operações de um algoritmo criptográfico não se tornem previsíveis a ponto de permitir que criptoanalistas infiram as saídas e operações do algoritmo, o que tornaria o algoritmo inseguro. Gerar uma aleatoriedade adequada com alto grau de incerteza é uma verdadeira façanha, mas existem métodos que garantem um nível adequado de aleatoriedade para uso em algoritmos criptográficos.

Existem duas categorias de fontes de aleatoriedade: **geradores de números aleatórios** e **geradores de números pseudoaleatórios**.

**Geradores de números aleatórios**

**Geradores de números aleatórios (RNGs)** são sistemas de software ou hardware que utilizam a aleatoriedade disponível no mundo real, chamada de aleatoriedade real. Isso pode incluir variações de temperatura, ruídos térmicos de vários componentes eletrônicos ou ruído acústico. Outras fontes são baseadas em fenômenos físicos como pressionamentos de teclas, movimentos do cursor do mouse ou movimentos do disco de um sistema de computador em execução. Esses tipos de fontes de aleatoriedade não são muito práticas devido à dificuldade de obter esses dados ou pela falta de entropia suficiente. Além disso, essas fontes nem sempre estão disponíveis ou podem estar disponíveis apenas por um tempo limitado.

**Geradores de números pseudoaleatórios**

**Geradores de números pseudoaleatórios (PRNGs)** são funções determinísticas que operam com base no princípio de usar um valor inicial aleatório chamado de *semente* (*seed*) para produzir um conjunto de elementos com aparência aleatória. PRNGs são comumente usados para gerar chaves para algoritmos de criptografia. Um exemplo comum de PRNG é o **Blum-Blum-Shub (BBS)**. PRNGs são uma alternativa melhor aos RNGs devido à sua confiabilidade e natureza determinística.

**Gerando cadeias aleatórias**

O seguinte comando pode ser usado para gerar uma cadeia pseudoaleatória usando o OpenSSL:

$ openssl rand -hex 16

Mais informações sobre o BBS estão disponíveis no seguinte artigo de pesquisa original:  
Blum, L., Blum, M., e Shub, M., 1986. *A simple unpredictable pseudo-random number generator*. SIAM Journal on Computing, 15(2), pp.364–383:

<https://shub.ccny.cuny.edu/articles/1986-A_simple_unpredictable_pseudo-random_number_generator.pdf>

O comando acima produzirá uma cadeia aleatória de 16 bytes codificada em hexadecimal, por exemplo:

06532852b5da8a5616dfade354a9f270

Existem outras variações que você pode explorar com a ferramenta de linha de comando do OpenSSL. Informações adicionais e ajuda podem ser obtidas com o seguinte comando:

$ openssl help

Na próxima seção, veremos as **funções de hash**, que desempenham um papel crucial no desenvolvimento de blockchains.

**Funções de hash**

Funções de hash são usadas para criar resumos (*digests*) de comprimento fixo a partir de cadeias de entrada de comprimento arbitrário. Funções de hash não usam chave e fornecem um serviço de integridade de dados. Elas são geralmente construídas utilizando técnicas de construção de funções de hash iteradas e dedicadas.

Várias famílias de funções de hash estão disponíveis, como **MD**, **SHA-1**, **SHA-2**, **SHA-3**, **RIPEMD** e **Whirlpool**. Funções de hash são comumente usadas para assinaturas digitais e códigos de autenticação de mensagem (*MACs*), como **HMACs**.

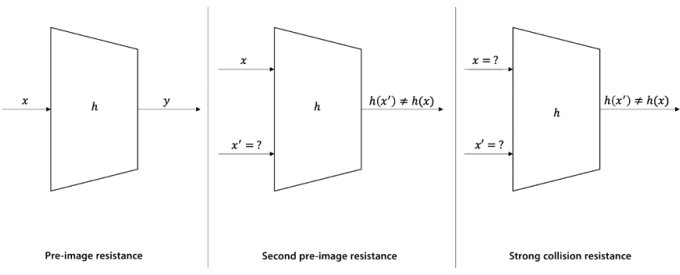
As funções de hash também são normalmente usadas para fornecer serviços de integridade de dados. Elas podem ser utilizadas tanto como funções unidirecionais quanto para construir outras primitivas criptográficas, como MACs e assinaturas digitais. Algumas aplicações usam funções de hash como um meio para gerar PRNGs. Existem duas propriedades práticas das funções de hash que devem ser atendidas, dependendo do nível de integridade exigido:

* **Compressão de mensagens arbitrárias em resumos de comprimento fixo**: esta propriedade se refere ao fato de que uma função de hash deve ser capaz de receber um texto de entrada de qualquer tamanho e gerar uma mensagem comprimida de tamanho fixo. Funções de hash produzem uma saída comprimida em vários tamanhos de bits, geralmente entre 128 bits e 512 bits.
* **Fácil de computar**: funções de hash são funções unidirecionais eficientes e rápidas. Requer-se que as funções de hash sejam muito rápidas de calcular, independentemente do tamanho da mensagem. A eficiência pode diminuir se a mensagem for muito grande, mas a função ainda deve ser suficientemente rápida para uso prático.

Existem também três propriedades de segurança que devem ser satisfeitas, dependendo do nível de integridade:

* **Resistência à pré-imagem**: esta propriedade afirma que, dado um valor *y*, é computacionalmente inviável (quase impossível) encontrar um valor *x* tal que h(x) = y. Aqui, *h* é a função de hash, *x* é a entrada e *y* é o hash. A primeira propriedade de segurança exige que *y* não possa ser revertida para *x*. *x* é considerado uma pré-imagem de *y*, daí o nome *resistência à pré-imagem*. Isso também é chamado de *propriedade unidirecional*.
* **Resistência à segunda pré-imagem**: essa propriedade afirma que, dado *x*, é computacionalmente inviável encontrar outro valor *x’* tal que *x’ ≠ x* e *h(x’) = h(x)*. Essa propriedade também é conhecida como *resistência fraca a colisões*.
* **Resistência a colisões**: essa propriedade afirma que é computacionalmente inviável encontrar dois valores distintos *x’* e *x* tais que *h(x’) = h(x)*. Em outras palavras, duas mensagens de entrada diferentes não devem gerar a mesma saída de hash. Essa propriedade também é conhecida como *resistência forte a colisões*.

Essas propriedades de segurança são representadas no seguinte diagrama:



**Figura 3.3: Três propriedades de segurança das funções de hash**

Devido à sua própria natureza, funções de hash sempre terão algumas colisões. Esta é uma situação em que duas mensagens diferentes geram o mesmo valor de hash. No entanto, essas colisões devem ser computacionalmente impraticáveis de encontrar. Um conceito conhecido como **efeito avalanche** é desejável em todas as funções de hash criptográficas. O efeito avalanche especifica que uma pequena mudança, mesmo uma única letra no texto de entrada, resultará em uma saída de hash completamente diferente.

As funções de hash são geralmente projetadas seguindo uma abordagem de **funções de hash iteradas**. Com este método, a mensagem de entrada é comprimida em múltiplas rodadas em uma base de bloco por bloco para produzir a saída comprimida. Um tipo popular de função de hash iterada é a **construção Merkle-Damgård**. Esta construção baseia-se na ideia de dividir os dados de entrada em blocos de tamanho igual e então alimentá-los por funções de compressão de forma iterativa. A resistência a colisões da propriedade das funções de compressão garante que a saída do hash também seja resistente a colisões. Além da Merkle-Damgård, há várias outras construções de funções de compressão propostas por pesquisadores, por exemplo, **Miyaguchi-Preneel** e **Davies-Meyer**.

Múltiplas categorias de funções de hash são apresentadas na seção a seguir.

**Funções de resumo de mensagem (*Message digest functions*)**

Funções de resumo de mensagem (MD) eram prevalentes no início dos anos 1990. **MD4** e **MD5** pertencem a esta categoria. Ambas as funções MD foram consideradas inseguras devido a colisões de mensagens encontradas e não são mais recomendadas para uso. **MD5** é uma função de hash de 128 bits que era comumente usada para verificações de integridade de arquivos.

**Algoritmos de Hash Seguro (*Secure Hash Algorithms*)**

A lista a seguir descreve os algoritmos de hash seguro mais comuns (SHAs):

* **SHA-0**: Esta é uma função de 160 bits introduzida pelo Instituto Nacional de Padrões e Tecnologia dos EUA (NIST) em 1993.
* **SHA-1**: O SHA-1 foi introduzido em 1995 pelo NIST como substituto do SHA-0. Também é uma função de hash de 160 bits. O SHA-1 é comumente usado em implementações de SSL e TLS. Deve-se observar que o SHA-1 agora é considerado inseguro e está sendo descontinuado pelas autoridades certificadoras. Seu uso é desencorajado em quaisquer novas implementações.
* **SHA-2**: Esta categoria inclui quatro funções definidas pelo número de bits do hash: **SHA-224**, **SHA-256**, **SHA-384** e **SHA-512**.
* **SHA-3**: Esta é a família mais recente de funções SHA. **SHA3-224**, **SHA3-256**, **SHA3-384** e **SHA3-512** são membros desta família. O SHA-3 é uma versão padronizada pelo NIST do **Keccak**.
* **RIPEMD**: RIPEMD é o acrônimo para *RACE Integrity Primitives Evaluation Message Digest*. Ele é baseado nas ideias de design usadas para construir o MD4. Existem várias versões do RIPEMD, incluindo de 128 bits, 160 bits, 256 bits e 320 bits.
* **Whirlpool**: Baseia-se em uma versão modificada da cifra Rijndael conhecida como **W**. Utiliza a função de compressão **Miyaguchi-Preneel**, que é um tipo de função unidirecional usada para a compressão de duas entradas de comprimento fixo em uma única saída de comprimento fixo. É uma função de compressão de comprimento de bloco único.

As funções de hash têm muitas aplicações práticas, que vão desde simples verificações de integridade de arquivos e armazenamento de senhas até o uso em protocolos e algoritmos criptográficos. Elas são usadas em redes ponto a ponto (*Peer to Peer – P2P*), compartilhamento de arquivos P2P, identificação de vírus, filtros de Bloom, árvores de Merkle, árvores de Patricia e tabelas hash distribuídas.

Na seção seguinte, você será introduzido ao design do **SHA-256** e do **SHA-3**. Ambos são usados no Bitcoin e no Ethereum, respectivamente. O Ethereum usa o **Keccak**, que é o algoritmo original apresentado ao NIST, em vez da versão padronizada SHA-3. O NIST, após algumas modificações, como o aumento do número de rodadas e o uso de preenchimento de mensagens mais simples, padronizou o Keccak como SHA-3.

**SHA-256**

O SHA-256 possui um limite de tamanho da mensagem de entrada de 2⁶⁴ - 1 bits. O tamanho do bloco é de 512 bits, e o tamanho da palavra é de 32 bits. A saída é um *digest* de 256 bits. A função de compressão processa um bloco de mensagem de 512 bits e um valor de hash intermediário de 256 bits. Existem dois componentes principais dessa função: a **função de compressão** e o **agendamento de mensagem**. O algoritmo funciona como segue, em nove etapas:

**Pré-processamento:**

a. O **preenchimento da mensagem** é usado para ajustar o comprimento de um bloco para 512 bits, se for menor que o tamanho de bloco exigido.

b. A **análise da mensagem** em blocos de mensagens garante que a mensagem e seu preenchimento sejam divididos em blocos iguais de 512 bits.

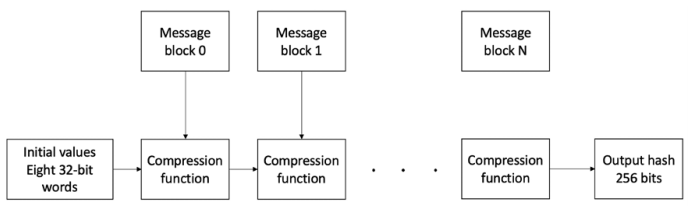
c. A **configuração do valor de hash inicial**, que consiste em oito palavras de 32 bits obtidas ao se tomar os primeiros 32 bits das partes fracionárias das raízes quadradas dos primeiros oito números primos. Esses valores iniciais são fixos e escolhidos para inicializar o processo. Eles fornecem um nível de confiança de que não há *backdoor* no algoritmo.

**Cálculo do hash:**

a. Cada bloco de mensagem é processado em sequência, e são necessárias 64 rodadas para calcular a saída completa do hash. Cada rodada usa constantes ligeiramente diferentes para garantir que nenhuma rodada seja igual à outra.

b. O **agendamento da mensagem** é preparado.  
c. Oito variáveis de trabalho são inicializadas.  
d. A **função de compressão** é executada 64 vezes.  
e. O valor de hash intermediário é calculado.  
f. Finalmente, após repetir as etapas 5 a 8 até que todos os blocos (partes dos dados) da mensagem de entrada sejam processados, o hash de saída é produzido pela concatenação dos valores de hash intermediários.

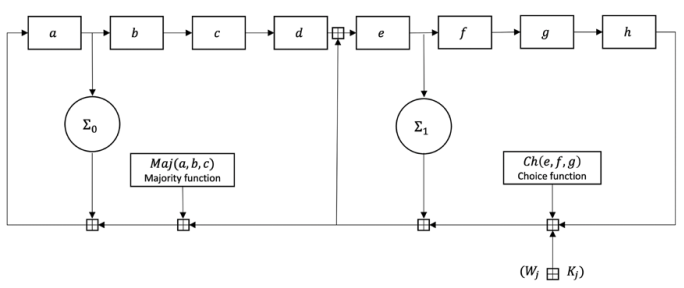
Em alto nível, o SHA-256 pode ser visualizado no seguinte diagrama:



**Figura 3.4: Visão geral em alto nível do SHA-256**

Como mostrado no diagrama anterior, o SHA-256 é uma construção Merkle-Damgård que pega a mensagem de entrada e a divide em blocos iguais (partes dos dados) de 512 bits. Os valores iniciais (ou valores de hash iniciais), ou o vetor de inicialização, são compostos por oito palavras constantes de 32 bits (a, b, c, d, e, f, g — cada uma com 256 bits) que são alimentadas na função de compressão com o primeiro bloco de mensagem. Blocos subsequentes são alimentados na função de compressão até que todos os blocos sejam processados, e por fim o hash de saída é produzido.

A função de compressão do SHA-256 é mostrada no diagrama a seguir:



**Figura 3.5: Função de compressão do SHA-2**

No diagrama anterior, **a, b, c, d, e, f, g e h** são os registradores para oito constantes iniciais pré-determinadas e, em seguida, para valores de hash intermediários para os próximos blocos. As funções **Maj** e **Ch** são definidas pelas fórmulas mostradas abaixo:

**Maj(x, y, z) = (x ∧ y) ⊕ (x ∧ z) ⊕ (y ∧ z)**  
**Ch(x, y, z) = (x ∧ y) ⊕ (¬x ∧ z)**

Onde:

* ∧ é o operador **E lógico bit a bit** (AND),
* ⊕ é o **XOR bit a bit**,
* ¬ é o **NÃO lógico bit a bit** (NOT).

O XOR pode ser substituído por OR bit a bit sem alteração na saída. As funções operam sobre vetores de 32 bits.

**Maj** é a função de "maioria", onde a saída produzida se baseia na maioria dos bits de entrada. Em outras palavras, se a maioria das entradas for 1, então a saída é 1; caso contrário, é 0. Aqui, 3 bits de entrada x, y e z produzem a saída.

**Ch** é a função de escolha (*choice*), na qual uma escolha é feita entre os bits de entrada dependendo do valor de uma das entradas. É como uma construção “if..then..else” em programação, ou um seletor de dados. Ela funciona com o seguinte princípio:

**Ch(x, y, z) = y se x = 1; z se x = 0**

Ou seja, x, y e z são entradas, e dependendo se x = 1 ou x = 0, ou y ou z será selecionado.

Como mostrado na Figura 3.5, as funções **Maj** e **Ch** são aplicadas bit a bit. **Σ₀** e **Σ₁** realizam rotações bit a bit. Por exemplo:

* **Σ₀(x)** retorna: (x rotado à direita por 2) ⊕ (x rotado à direita por 13) ⊕ (x rotado à direita por 22)
* **Σ₁(e)** retorna: (e rotado à direita por 6) ⊕ (e rotado à direita por 11) ⊕ (e rotado à direita por 25)

O símbolo **⊞** significa adição módulo 2³², usada no SHA-256.

As constantes de mistura são **Wᵢ** e **Kⱼ**, que são adicionadas no laço principal (função de compressão) da função de hash, a qual roda 64 vezes.

**SHA-3 (Keccak)**

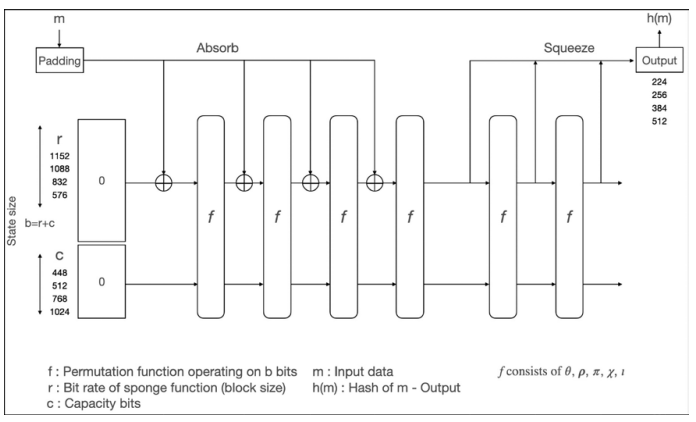
A estrutura do SHA-3 é muito diferente da do SHA-1 e SHA-2. A ideia principal por trás do SHA-3 baseia-se em **permutações sem chave**, ao contrário de outras construções típicas de funções de hash que utilizam **permutações com chave**. O **Keccak** também **não** utiliza a **transformação Merkle-Damgård**, que é comumente usada para lidar com mensagens de entrada de comprimento arbitrário em funções de hash. Uma abordagem mais nova, chamada de **construção esponja e compressão** (*sponge and squeeze construction*), é usada no Keccak. Trata-se de um modelo de permutação aleatória.

Diferentes variantes do SHA-3 foram padronizadas, como:

* **SHA3-224**
* **SHA3-256**
* **SHA3-384**
* **SHA3-512**
* **SHAKE128**
* **SHAKE256**

**SHAKE128** e **SHAKE256** são funções de saída extensível (*XOFs – extendable-output functions*), que permitem que a saída seja estendida para qualquer comprimento desejado.

O diagrama a seguir mostra o modelo esponja e compressão, que é a base do SHA-3 ou Keccak. De forma análoga a uma esponja, os dados (*dados de entrada m*) são primeiro absorvidos pela esponja após a aplicação de preenchimento (*padding*). Em seguida, esses dados são transformados em um subconjunto do estado de permutação usando XOR (*ou exclusivo*) e, então, a saída é extraída (*squeezed*) da função esponja que representa o estado transformado. A **taxa r** (*rate*) é o tamanho do bloco de entrada da função esponja, enquanto a **capacidade c** determina o nível de segurança:



**Figura 3.6: A função de absorção e compressão no SHA-3**

No diagrama anterior, o **tamanho do estado** *b* é calculado somando-se os bits da **taxa r** e da **capacidade c**. *r* e *c* podem ser quaisquer valores, contanto que *r + c* pertença a um dos seguintes tamanhos permitidos: 25, 50, 100, 200, 400, 800 ou 1600. O estado é uma **matriz de bits tridimensional**. O estado inicial é definido como zero.

Os dados *m* são inseridos na fase de **absorção** bloco a bloco via XOR ⊕ após o preenchimento.

A tabela a seguir mostra o valor da **taxa de bits r (tamanho do bloco)** e a **capacidade c** necessária para alcançar o tamanho de hash de saída desejado sob a configuração mais eficiente, com *r + c = 1600*:

| **r (tamanho do bloco)** | **c (capacidade)** | **Tamanho do hash de saída** |
| --- | --- | --- |
| 1152 | 448 | 224 |
| 1088 | 512 | 256 |
| 832 | 768 | 384 |
| 576 | 1024 | 512 |

A função *f* é uma **função de permutação**. Ela contém cinco operações de transformação:

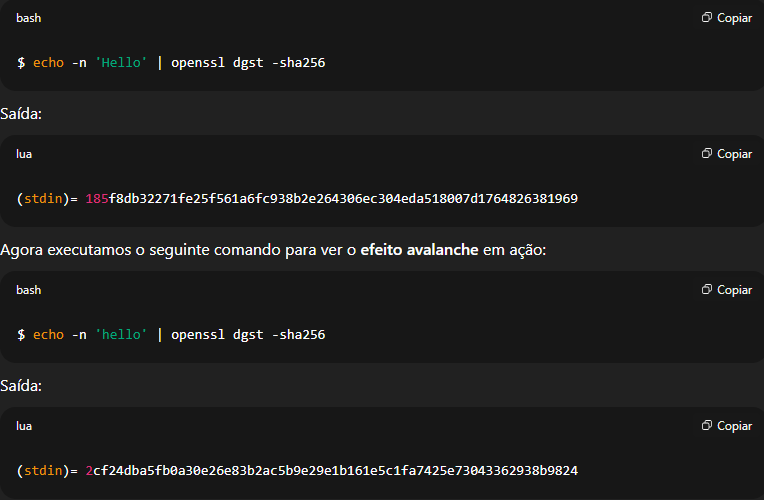
* **𝜃 (Theta)**: XOR de bits no estado, usada para mistura.
* **𝜌 (Rho)**: Função de difusão que realiza rotação de bits.
* **𝜋 (Pi)**: Função de difusão.
* **𝜒 (Chi)**: Cada bit é combinado bit a bit.
* **𝜄 (Iota)**: Combinação com constantes de rodada.

Os detalhes de cada operação de transformação estão fora do escopo deste livro. A ideia principal é aplicar essas transformações para alcançar o **efeito avalanche**, que foi introduzido anteriormente neste capítulo. Essas cinco operações combinadas formam uma **rodada** (*round*). No padrão SHA-3, o número de rodadas é **24** para alcançar o nível de segurança desejado.

Veremos algumas aplicações e exemplos de construções feitas usando funções de hash, como **árvores de Merkle**, no Capítulo 9 – *Arquitetura do Ethereum*.

**Criptografando mensagens com SHA-256**

O seguinte comando produzirá um hash de 256 bits da mensagem “Hello” usando o algoritmo SHA-256:



Observe que mesmo uma pequena alteração no texto, como mudar a letra **H** de maiúscula para minúscula, resulta em uma **grande mudança** na saída do hash:

**Hello:**  
18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:30:4e:da:51:80:07:  
d1:76:48:26:38:19:69

**hello:**  
2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:5c:1f:a7:42:5e:73:  
04:33:62:93:8b:98:24

Geralmente, as funções de hash não utilizam uma chave. No entanto, se forem usadas com uma chave, elas podem ser utilizadas para criar **MACs**.

**Aplicações das funções de hash criptográficas**

Existem várias construções que foram desenvolvidas usando parâmetros criptográficos básicos para resolver diferentes problemas computacionais. Essas construções também são utilizadas em blockchains para fornecer diversos serviços específicos de protocolo. Por exemplo, funções de hash são usadas para construir **árvores de Merkle**, que são utilizadas para verificar de forma eficiente e segura grandes quantidades de dados em sistemas distribuídos. Outras aplicações das funções de hash em blockchains são para fornecer diversos serviços de segurança.

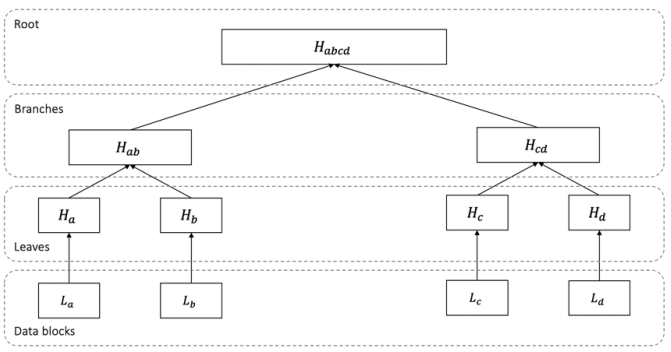
Esses serviços estão listados abaixo:

* Funções de hash são utilizadas em **quebra-cabeças criptográficos**, como o mecanismo de **prova de trabalho (PoW)** no Bitcoin. O PoW do Bitcoin utiliza a função de hash criptográfica **SHA-256**.
* Geração de **endereços** em blockchains. Por exemplo, no Ethereum, contas blockchain são representadas como endereços. Esses endereços são obtidos ao se aplicar o hash da chave pública com o algoritmo **Keccak-256**, e depois utilizando os últimos 20 bytes desse valor com hash.
* **Resumos de mensagem** em assinaturas digitais.
* Criação de **árvores de Merkle** para garantir a integridade da estrutura de transações no blockchain. Especificamente, essa estrutura é usada para verificar rapidamente se uma transação está incluída em um bloco ou não. No entanto, observe que **árvores de Merkle** por si só **não** são uma ideia nova; elas apenas se tornaram mais populares com o advento da tecnologia blockchain.

As **árvores de Merkle** são os blocos de construção centrais de todos os blockchains, por exemplo, Bitcoin e Ethereum. Exploraremos as árvores de Merkle em detalhes agora.

**Árvore de Merkle**

A **árvore de Merkle** foi introduzida por Ralph Merkle. Árvores de Merkle permitem a verificação segura e eficiente de conjuntos de dados de grande porte. Um diagrama de uma árvore de Merkle é mostrado abaixo:



**Figura 3.7: Uma árvore de Merkle**

Uma árvore de Merkle é uma **árvore binária** na qual as entradas são primeiro colocadas nas **folhas** (nós sem filhos), e então os valores de pares de nós filhos são **combinados com uma função de hash** para produzir um valor para o **nó pai** (nó interno), até que um único valor de hash conhecido como **raiz de Merkle** seja alcançado. Essa estrutura ajuda a verificar rapidamente a integridade de toda a árvore (todo o conjunto de dados), apenas verificando a raiz de Merkle no topo da árvore, pois se ocorrer qualquer alteração em algum dos hashes da árvore, a raiz de Merkle também será alterada.

Uma árvore de Merkle é usada para provar que um bloco de dados **Bi** é membro de um conjunto de **N** blocos de dados, formalmente:  
{B₁, B₂, ..., Bₙ}.

Com uma raiz de Merkle e um bloco de dados candidato, utilizando árvores de Merkle podemos provar que um bloco de dados existe dentro de um conjunto. Essa prova é chamada de **prova de Merkle** (*Merkle proof*) e envolve obter um conjunto de hashes chamado de **caminho de Merkle** (*Merkle path*) para um dado bloco e a raiz de Merkle.

Em outras palavras, o caminho de Merkle para um elemento (ou bloco) de dados é a **cadeia mínima de hashes necessária para recriar a raiz de Merkle** por meio de hashing e concatenação repetida até que a raiz seja criada. Por exemplo, na Figura 3.7, se a existência de **La** precisa ser provada, então o caminho de Merkle seria **Ha → Hab → Habcd** até a raiz. Podemos provar a existência do bloco de dados **La** recriando a raiz de Merkle através desse caminho e comparando com a raiz fornecida. Se as raízes de Merkle não coincidirem, então podemos afirmar que o elemento de dados em questão **não** está presente na árvore de Merkle — e o contrário também é verdadeiro. O **caminho de Merkle** também é conhecido como **caminho de autenticação**.

Essa é a razão pela qual a integridade do sistema pode ser verificada rapidamente apenas observando a **raiz de Merkle**. Outra vantagem das árvores de Merkle é que **não há necessidade de armazenar grandes quantidades de dados**, apenas os **hashes dos dados**, que são resumos de tamanho fixo do conjunto de dados. Devido a essa propriedade, o armazenamento e gerenciamento de árvores de Merkle são fáceis e eficientes, pois ocupam **pouquíssimo espaço** de armazenamento. Além disso, como a árvore é eficiente em termos de armazenamento, as provas relevantes de integridade também são menores em tamanho e rápidas para serem transmitidas pela rede, tornando-as **eficientes em largura de banda** (*bandwidth-efficient*).

**Árvore Merkle-Patricia**

Para entender as árvores Patricia, precisamos primeiro entender o que é uma **trie**. Uma **trie**, ou **árvore digital**, é uma estrutura de dados em árvore ordenada usada para armazenar um conjunto de dados.

A árvore **PATRICIA** (*Practical Algorithm to Retrieve Information Coded in Alphanumeric*), também conhecida como **árvore radix**, é uma representação compacta de uma trie em que um nó que é o único filho de um pai é **mesclado** com esse pai. As **chaves** representam o **caminho** para alcançar um nó. Os nós que compartilham a mesma chave podem compartilhar o mesmo caminho, tornando assim uma forma eficiente de encontrar **prefixos comuns** utilizando uma pequena quantidade de memória.

Uma **árvore Merkle-Patricia** é uma árvore que possui um **nó raiz** contendo o valor de hash de toda a estrutura de dados. A árvore Merkle-Patricia combina as árvores de **Merkle** e **Patricia**, onde a Patricia é usada para **armazenamento eficiente** e a Merkle fornece **validação à prova de violação** (*tamper-proof*). A árvore Patricia também é modificada para armazenar **cadeias hexadecimais** em vez de bits e suporta **16 ramificações**.

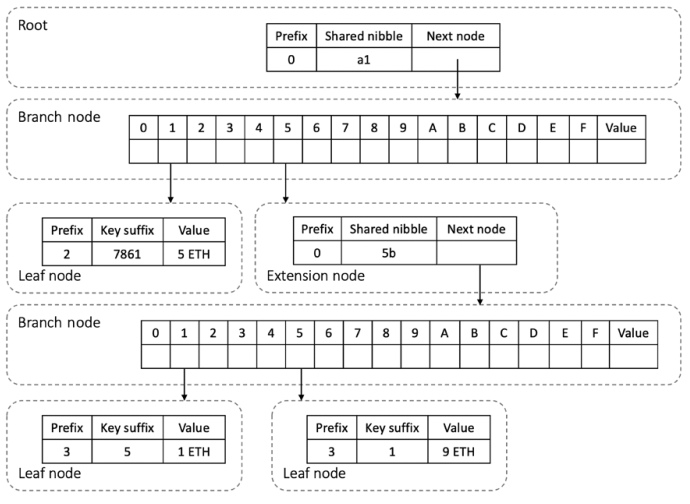
A árvore Merkle-Patricia possui quatro tipos de nós:

* **Nós nulos**, que são nós inexistentes representados por cadeias vazias.
* **Nós de ramificação** (*branch nodes*), que são nós com 17 itens usados para ramificação. Eles têm **16 ramificações possíveis** de 0 a F, e o **17º item** (valor) armazena um valor apenas se o nó for de terminação.
* **Nós de extensão** (*extension nodes*), que são nós com 2 itens contendo um **caminho e um valor**. São usados para **compressão**, onde uma parte comum de múltiplas chaves é armazenada, ou seja, um nibble compartilhado.
* **Nós folha** (*leaf nodes*), que também são nós com 2 itens contendo um **caminho e um valor**. O nó folha é o **nó de terminação** no caminho, sem filhos, e agrupa os **sufixos comuns** de chaves no caminho como técnica de compressão.

**Nós folha** e **nós de extensão** são distinguidos por um **valor de prefixo hexadecimal**:

* Um nó folha com um número par de nibbles tem prefixo **2**;
* Com número ímpar de nibbles, o prefixo é **3**.
* Nós de extensão com número par de nibbles têm prefixo **0**;
* Com número ímpar, o prefixo é **1**.

A figura abaixo mostra como uma árvore Merkle-Patricia é usada para armazenar chaves e valores (contas e saldos no Ethereum):



**Figura 3.8: Árvore Merkle-Patricia**

No diagrama acima:

* A chave **a117861** tem valor de **5 ETH**
* A chave **a155b15** tem valor de **1 ETH**
* A chave **a155b51** tem valor de **9 ETH**

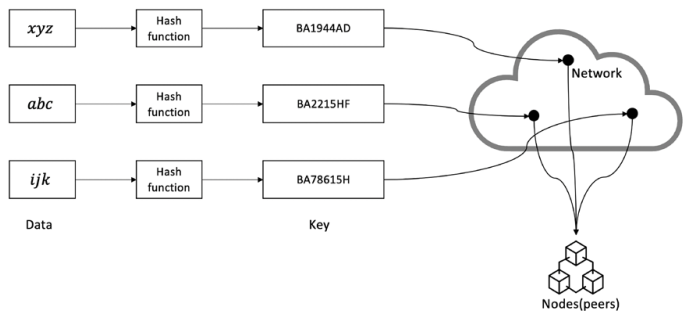
Observe que **a1** é um **prefixo comum** (um nibble compartilhado) e é agrupado no nó de extensão da raiz.  
Então, um **nó folha** (sem filhos) com o sufixo da chave **7861** se ramifica a partir do nó de ramificação.  
Além disso, outro nó de extensão com o nibble compartilhado (prefixo) **5b** se ramifica e leva a outro nó de ramificação, que por fim termina com dois nós folha com os sufixos de chave **5** e **1**.

As **árvores Merkle-Patricia** são usadas no blockchain do Ethereum para **armazenar pares chave-valor**, onde a **raiz** é **hasheada com SHA3** e incluída no **cabeçalho do bloco**.

**Tabelas hash distribuídas**

Uma **tabela hash** é uma estrutura de dados usada para mapear chaves a valores. Internamente, uma função de hash é usada para calcular um índice em um **array de compartimentos** (*buckets*), a partir dos quais o valor requerido pode ser encontrado. Os compartimentos armazenam registros usando uma chave de hash e são organizados em uma ordem específica.

Com essa definição em mente, podemos pensar em uma **tabela hash distribuída (DHT)** como uma estrutura de dados onde os dados estão **espalhados por vários nós**, e os **nós** são equivalentes a compartimentos em uma **rede P2P** (peer-to-peer). O diagrama a seguir mostra como uma DHT funciona:



**Figura 3.9: Tabela hash distribuída**

No diagrama acima, os dados são passados por uma função de hash, que então gera uma **chave compacta**. Essa chave é então vinculada aos dados (valores) na rede P2P. Quando usuários na rede solicitam os dados (via nome do arquivo), o nome do arquivo pode ser hasheado novamente para produzir a mesma chave, e qualquer nó da rede pode então ser solicitado a encontrar os dados correspondentes.

Uma DHT fornece **descentralização**, **tolerância a falhas** e **escalabilidade**.

Nesta seção, abordamos várias aplicações das funções de hash criptográficas. Funções de hash são de particular importância em blockchains, pois são essenciais para a construção de **árvores de Merkle**, que são utilizadas em blockchains para a verificação **eficiente e rápida de grandes conjuntos de dados**.

Veremos como **MACs** e **HMACs** funcionam após introduzirmos a criptografia de chave simétrica, pois essas construções fazem uso de chaves para criptografia.

**Primitivas de chave simétrica**

**Criptografia simétrica** refere-se a um tipo de criptografia onde a chave usada para **criptografar** os dados é a mesma usada para **descriptografá-los**. Assim, ela também é conhecida como **criptografia de chave compartilhada**. A chave deve ser estabelecida ou acordada **antes** da troca de dados entre as partes comunicantes. Essa é a razão pela qual também é chamada de **criptografia de chave secreta**.

Outros tipos de chave são as **chaves públicas** e **chaves privadas**, que são geradas em pares para a criptografia de chave pública ou **criptografia assimétrica**. As chaves públicas são usadas para criptografar o texto plano, enquanto as chaves privadas são usadas para a descriptografia e devem ser mantidas em segredo pelo receptor.

As chaves também podem ser **efêmeras (temporárias)** ou **estáticas**. Chaves efêmeras são destinadas ao uso em um curto período de tempo, como em uma única sessão entre os participantes, enquanto as chaves estáticas são destinadas a uso de longo prazo. Outro tipo de chave é chamada de **chave mestra** (*master key*), que é usada para proteção, criptografia, descriptografia e geração de outras chaves.

Existem diferentes métodos para gerar chaves. Esses métodos estão listados a seguir:

* **Aleatório**: onde um gerador de números aleatórios é usado para gerar um conjunto aleatório de bytes que pode ser usado como chave.
* **Baseado em derivação de chave**: onde uma única chave ou múltiplas chaves são derivadas de uma senha. Uma **função de derivação de chave (KDF)** é usada para esse propósito, tomando a senha como entrada e convertendo-a em uma chave. Funções de derivação de chave comumente usadas são:
  + PBKDF1 (*Password-Based Key Derivation Function 1*)
  + PBKDF2
  + Argon2
  + Scrypt
* **Protocolo de acordo de chave**: onde dois ou mais participantes executam um protocolo que produz uma chave, e essa chave é então compartilhada entre os participantes. Em esquemas de acordo de chave, **todos os participantes contribuem igualmente** no esforço para gerar a chave secreta compartilhada. O protocolo de acordo de chave mais utilizado é o **protocolo de troca de chaves Diffie-Hellman**.

Dentro dos esquemas de criptografia, também existem alguns **números aleatórios** que desempenham um papel vital durante a operação do processo de criptografia. Esses números aleatórios são explicados a seguir:

* **Nonce**: é um número que pode ser usado **apenas uma vez** em um protocolo criptográfico. Não deve ser reutilizado. Nonces podem ser gerados de um grande conjunto de números aleatórios ou também podem ser sequenciais. O uso mais comum dos nonces é prevenir **ataques de repetição** (*replay attacks*) em protocolos criptográficos.
* **Valor inicial** ou **vetor de inicialização (IV)**: é um número aleatório, que basicamente é um nonce, mas deve ser escolhido de maneira **imprevisível**. Isso significa que não pode ser sequencial. IVs são amplamente utilizados em algoritmos de criptografia para proporcionar maior segurança.
* **Sal (salt)**: é um valor aleatório criptograficamente forte que é tipicamente usado em funções de hash para fornecer defesa contra **ataques de dicionário** ou **ataques rainbow**. Usando ataques de dicionário, esquemas de senha baseados em hash podem ser quebrados ao se tentar os hashes de milhões de palavras de um dicionário de forma bruta e comparando com a senha hasheada. Se um sal for usado, então um ataque de dicionário se torna difícil de executar porque um sal aleatório torna cada senha única; além disso, o invasor teria que executar um ataque separado para cada sal aleatório, o que é bastante inviável.

**Códigos de autenticação de mensagem**

**Códigos de autenticação de mensagem (MACs)** são às vezes chamados de **funções de hash com chave** (*keyed hash functions*), e podem ser usados para fornecer **integridade de mensagem** e **autenticação**. Mais especificamente, eles são usados para fornecer **autenticação da origem dos dados**. São primitivas criptográficas simétricas que usam uma **chave compartilhada entre o remetente e o receptor**. Os MACs podem ser construídos usando **cifras de bloco** ou **funções de hash**.

As **KDFs** são usadas em carteiras Ethereum (arquivos keystore) para gerar uma chave simétrica **AES** que é usada para criptografar as carteiras. A função KDF usada em carteiras Ethereum é a **Scrypt**. Explicaremos tudo isso em detalhes no Capítulo 9, *Arquitetura do Ethereum*.

Mais informações sobre PBKDFs podem ser encontradas em:  
<https://tools.ietf.org/html/rfc8018>

**Códigos de autenticação de mensagem baseados em hash**

Assim como a função de hash, os **MACs baseados em hash (HMACs)** produzem uma saída de comprimento fixo e recebem como entrada uma mensagem de comprimento arbitrário. Neste esquema, o remetente assina uma mensagem usando o MAC, e o receptor a verifica usando a chave compartilhada.

A **chave é combinada com a mensagem** utilizando um dos dois métodos conhecidos como **prefixo secreto** ou **sufixo secreto**:

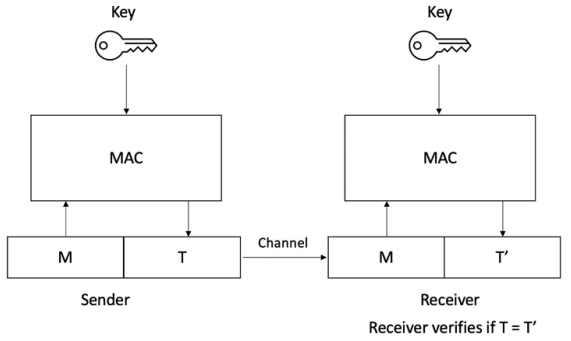
* No método de **prefixo secreto**, a chave é concatenada **antes** da mensagem.
* No método de **sufixo secreto**, a chave vem **depois** da mensagem.

Equações conceituais:

Prefixo: H(K || M)

Sufixo: H(M || K)

Existem prós e contras em ambos os métodos. Alguns ataques contra ambos os esquemas foram relatados. Por isso, foram propostas construções de HMAC que utilizam técnicas como **ipad** (*inner padding*) e **opad** (*outer padding*), consideradas seguras sob certas suposições.



**Figura 3.10: Operação de uma função MAC**

**Cifras de chave secreta**

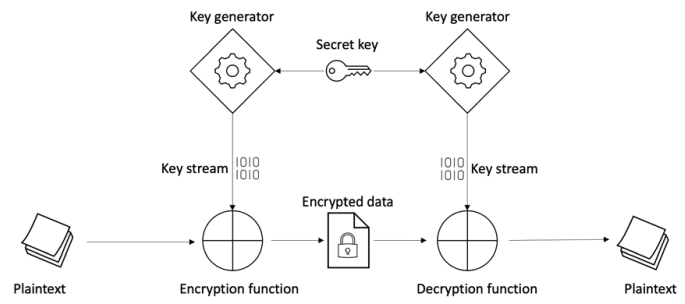
Existem dois tipos de cifras de chave secreta ou **cifras simétricas**:

1. **Cifras de fluxo** (*stream ciphers*)
2. **Cifras de bloco** (*block ciphers*)

**Cifras de fluxo**

As **cifras de fluxo** são algoritmos de criptografia que aplicam a cifra bit a bit (um bit por vez) ao texto plano usando um **fluxo de chave** (*keystream*).

Na cifra de fluxo, **a criptografia e a descriptografia são as mesmas funções**, pois se baseiam em operações simples de adição módulo 2 ou operações XOR. O requisito fundamental em cifras de fluxo é a **segurança e aleatoriedade** dos fluxos de chave. Diversas técnicas — desde PRNGs até RNGs reais baseados em hardware — foram desenvolvidas para gerar números aleatórios, e é vital que todos os geradores de chave sejam **criptograficamente seguros**.

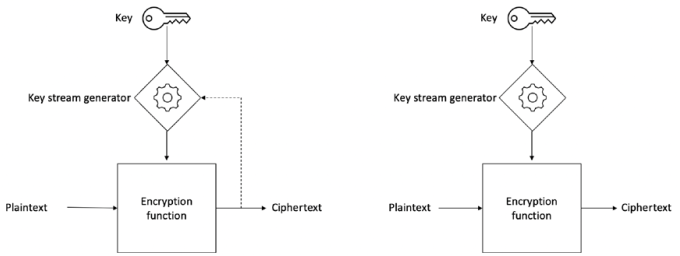


**Figura 3.11: Operação de uma cifra de fluxo**

Existem dois tipos de cifras de fluxo:

* **Cifras de fluxo síncronas**: o fluxo de chave depende **somente da chave**.
* **Cifras de fluxo assíncronas**: o fluxo de chave também depende dos **dados criptografados**.

Essa ideia pode ser visualizada na **Figura 3.12** abaixo.



**Figura 3.12: Cifra de fluxo assíncrona (à esquerda) vs cifra de fluxo síncrona (à direita)**

Algumas cifras de fluxo comumente usadas são:

* **RC4**
* **A5**

**Cifras de bloco**

As **cifras de bloco** são algoritmos de criptografia que **quebram o texto a ser criptografado (texto plano)** em blocos de comprimento fixo e aplicam a cifra **bloco a bloco**. As cifras de bloco são geralmente construídas usando uma estratégia de design conhecida como **cifra de Feistel**. Cifras de bloco mais recentes, como o **AES (Rijndael)**, foram construídas usando uma combinação de substituição e permutação, chamada de **rede de substituição-permutação** (*substitution-permutation network – SPN*). O **DES** (*Data Encryption Standard*) e o **AES** são exemplos típicos de cifras de bloco.

As **cifras de Feistel** são baseadas na **rede de Feistel**, uma estrutura desenvolvida por **Horst Feistel**. Essa estrutura se baseia na ideia de combinar várias rodadas de operações repetidas para atingir propriedades criptográficas desejáveis, conhecidas como **confusão** e **difusão**.

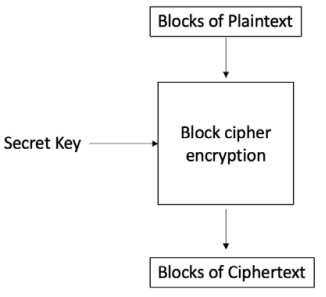
As redes de Feistel operam dividindo os dados em dois blocos (**esquerdo e direito**) e processando esses blocos por meio de **funções de rodada com chave** em iterações, de modo a fornecer **permutações pseudorrandômicas suficientes**.

**Confusão** adiciona complexidade ao relacionamento entre o texto cifrado e o texto plano. Isso é alcançado por meio de substituições. Na prática, a letra A no texto plano é substituída por X no texto cifrado. Em algoritmos criptográficos modernos, a substituição é realizada por meio de **tabelas de consulta** chamadas **S-boxes**.

A **difusão** espalha estatisticamente o texto plano sobre os dados cifrados. Isso garante que, mesmo se um único bit for alterado no texto de entrada, **pelo menos metade** (em média) dos bits no texto cifrado também será alterada.

A confusão é necessária para dificultar a descoberta da chave de criptografia, mesmo se muitos pares de dados cifrados e decifrados forem criados com a mesma chave. Na prática, isso é alcançado por meio de **transposição ou permutação**.

Uma vantagem importante do uso de uma **cifra de Feistel** é que **as operações de criptografia e descriptografia são quase idênticas**, exigindo apenas a **reversão do processo de criptografia** para realizar a descriptografia. O **DES** é um exemplo clássico de cifra baseada em Feistel.



**Figura 3.13: Operação simplificada de uma cifra de bloco**

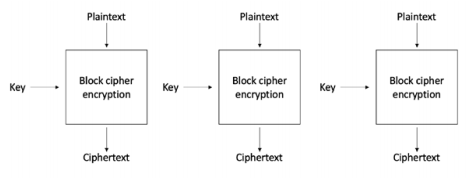
**Modos de operação da cifra de bloco**

Nos **modos de operação de cifra de bloco**, o texto plano é dividido em blocos de comprimento fixo, dependendo do tipo de cifra usada. Em seguida, a função de criptografia é aplicada a **cada bloco**. Os modos de operação de cifra de bloco mais comuns são:

* **ECB (Electronic Code Book)**
* **CBC (Cipher Block Chaining)**
* **CTR (Counter Mode)**

**ECB – Modo Livro de Códigos Eletrônico**

**ECB** é um modo básico de operação no qual os dados cifrados são produzidos como resultado da aplicação do algoritmo de criptografia a cada bloco de texto plano, um por um. Esse é o modo mais direto, mas **não deve ser usado na prática**, pois **não é seguro** e pode revelar informações:

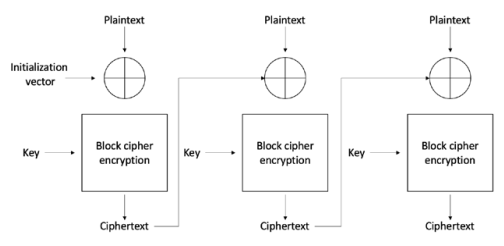


**Figura 3.14: Modo de livro de códigos eletrônico para cifras de bloco**

No diagrama anterior, temos o texto plano **P** fornecido como entrada para a função de criptografia de bloco, junto com uma chave, e o texto cifrado **C** é produzido como saída.

**CBC – Encadeamento de Blocos de Cifra**

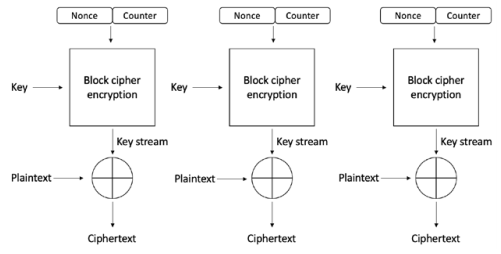
No modo **CBC**, cada bloco de texto plano é **XOR** com o bloco **cifrado anterior**. O CBC utiliza um **vetor de inicialização (IV)** para cifrar o primeiro bloco. Recomenda-se que o IV seja escolhido **aleatoriamente**:



**Figura 3.15: Modo de encadeamento de blocos de cifra**

**CTR – Modo Contador**

O **modo contador (CTR)** usa efetivamente uma cifra de bloco como uma cifra de fluxo. Nesse caso, um **nonce exclusivo** é fornecido, que é **concatenado com um valor de contador** para produzir um **fluxo de chave**:



**Figura 3.16: Modo contador**

O modo **CTR**, como mostrado no diagrama anterior, funciona utilizando um **nonce (N)** e um **contador (C)** que são alimentados na função de criptografia da cifra de bloco. Essa função, usando uma chave secreta (**KEY**), produz um **fluxo de chave** (uma cadeia de caracteres pseudorrandômicos ou randômicos), que ao ser XORado com o texto plano (**P**), produz o texto cifrado (**C**).

Até aqui, discutimos modos usados para **produzir textos cifrados** (dados criptografados). No entanto, existem outros modos que podem ser usados para **diferentes propósitos**, listados a seguir:

* **Modo de geração de fluxo de chave**: a função de criptografia gera um fluxo de chave usado em cifras de fluxo. Especificamente, o fluxo de chave é geralmente **XORado com o fluxo de texto plano** para produzir o texto cifrado.
* **Modo de autenticação de mensagens**: um **MAC** é produzido a partir de uma função de criptografia. Um MAC é um **código de verificação criptográfico** que fornece um serviço de integridade. O método mais comum para gerar um MAC usando cifras de bloco é o **CBC-MAC**, onde uma parte do **último bloco da cadeia** é usada como o MAC.

Em outras palavras, cifras de bloco são usadas no modo **CBC** para gerar um MAC. Um MAC pode ser usado para verificar se uma mensagem foi modificada por uma entidade não autorizada. Isso pode ser feito criptografando a mensagem com uma chave usando a função MAC. A mensagem resultante e o MAC da mensagem, uma vez recebidos pelo destinatário, são verificados criptografando novamente a mensagem com a mesma chave e comparando com o MAC recebido do remetente.  
Se forem iguais, isso significa que a mensagem **não foi modificada** por uma entidade não autorizada, fornecendo assim o serviço de **integridade**. Se não coincidirem, isso significa que a mensagem foi **alterada durante a transmissão**.

* **Qualquer cifra de bloco**, por exemplo, o **AES** no modo CBC, pode ser usada para gerar um MAC. O MAC da mensagem é, de fato, a **saída da última rodada da operação CBC**. O comprimento da saída do MAC é o mesmo que o comprimento do bloco da cifra de bloco usada.  
  Deve-se notar que os MACs funcionam como **assinaturas digitais**, mas **não podem fornecer o serviço de não repúdio** devido à sua **natureza simétrica**.
* **Modo hash criptográfico**: funções de hash são usadas principalmente para comprimir uma mensagem em um *digest* de comprimento fixo. Nesse modo, cifras de bloco são usadas como **funções de compressão** para produzir um **hash do texto plano**.

Também existem outros modos, como:

* **Modo de retroalimentação de cifra (CFB – Cipher Feedback)**
* **Modo de contador de Galois (GCM – Galois Counter Mode)**
* **Modo de retroalimentação de saída (OFB – Output Feedback)**

Esses modos também são usados em diversos cenários. A discussão de todos esses diferentes modos está além do escopo deste livro. Leitores interessados podem consultar qualquer livro-texto padrão sobre criptografia para obter mais detalhes.

Agora apresentaremos alguns conceitos que são relevantes para a criptografia e amplamente utilizados em diversas aplicações.

**Padrão de Criptografia Avançado (AES)**

Nesta seção, apresentaremos o design e o funcionamento de uma cifra de bloco atualmente dominante no mercado, conhecida como **AES**.

**Padrão de Criptografia de Dados (DES)**

O **DES** foi introduzido pelo **NIST** como um algoritmo padrão para criptografia e foi amplamente utilizado nas décadas de 1980 e 1990. No entanto, ele não se mostrou muito resistente a **ataques de força bruta**, devido aos avanços na tecnologia e na pesquisa criptográfica.

Por exemplo, em julho de 1998, a **Electronic Frontier Foundation (EFF)** quebrou o DES usando uma máquina dedicada chamada **EFF DES cracker** (ou **Deep Crack**).

O DES usa uma chave de apenas **56 bits**, o que gerou preocupações. Esse problema foi abordado com a introdução do **Triple DES (3DES)**, que propôs aplicar a cifra DES **três vezes** a cada bloco, aumentando assim o tamanho da chave para **168 bits**. Essa técnica torna os ataques de força bruta praticamente impossíveis.

Contudo, outras limitações, como o **desempenho lento** e o **pequeno tamanho de bloco de 64 bits**, também se mostraram indesejáveis.

**Como o AES funciona**

Em 2001, após uma competição aberta, um algoritmo de criptografia chamado **Rijndael**, inventado pelos criptógrafos **Joan Daemen** e **Vincent Rijmen**, foi padronizado como **AES** com pequenas modificações pelo **NIST**. Até o momento, **nenhum ataque** foi encontrado contra o AES que seja mais eficiente que o método de força bruta.

A versão original do Rijndael permitia diferentes tamanhos de chave e de bloco: **128 bits**, **192 bits** e **256 bits**.  
No padrão AES, no entanto, apenas o **tamanho de bloco de 128 bits** é permitido.  
Mas os **tamanhos de chave de 128, 192 e 256 bits** continuam sendo válidos.

Durante o processamento do algoritmo AES, um **array 4×4 de bytes** conhecido como **estado** é modificado através de múltiplas rodadas.  
A **criptografia completa** requer de **10 a 14 rodadas**, dependendo do tamanho da chave.

A tabela a seguir mostra os tamanhos de chave e o número de rodadas necessárias:

| **Tamanho da chave** | **Número de rodadas requerido** |
| --- | --- |
| 128 bits | 10 rodadas |
| 192 bits | 12 rodadas |
| 256 bits | 14 rodadas |

Uma vez que o estado é inicializado com a entrada para a cifra, as seguintes quatro operações são realizadas sequencialmente, passo a passo, para criptografar a entrada:

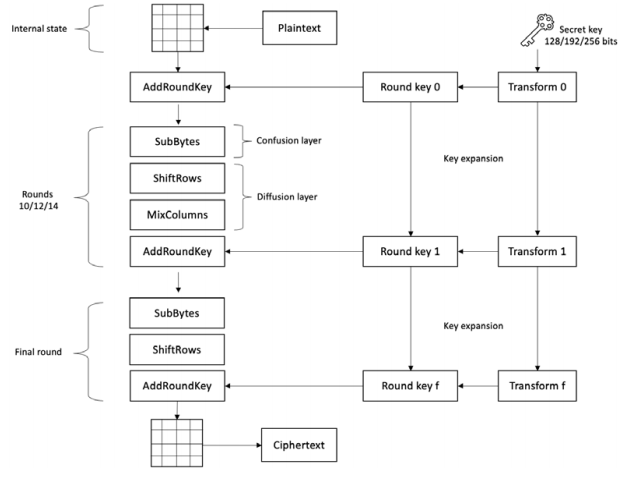
a. **AddRoundKey**: Nesta etapa, o array de estado é combinado via XOR com uma subchave, que é derivada da chave mestra.

b. **SubBytes**: Esta é a etapa de substituição, onde uma tabela de consulta (*S-box*) é usada para substituir todos os bytes do array de estado.

c. **ShiftRows**: Esta etapa é usada para deslocar cada linha para a esquerda, exceto a primeira, no array de estado, de forma cíclica e incremental.

d. **MixColumns**: Finalmente, todos os bytes são misturados linearmente, coluna por coluna.

Essa é uma rodada do AES. Na rodada final (que pode ser a 10ª, 12ª ou 14ª rodada, dependendo do tamanho da chave), a etapa 4 é substituída pela operação **AddRoundKey**, para garantir que as três primeiras etapas não possam ser simplesmente revertidas, conforme mostrado no diagrama a seguir:



**Figura 3.17: Diagrama do AES, mostrando a primeira rodada de criptografia AES. Na rodada final, a etapa de mistura não é realizada**

Diversas carteiras de criptomoedas usam criptografia AES para criptografar dados armazenados localmente. Carteiras Bitcoin usam **AES-256 no modo CBC** para criptografar as chaves privadas. Nas carteiras Ethereum, usa-se **AES-128-CTR**, ou seja, AES de 128 bits em modo contador para criptografar a chave privada. Pares no Ethereum também usam AES em modo contador (**AES CTR**) para criptografar sua comunicação P2P.

**Criptografando e descriptografando usando AES**

Podemos usar a ferramenta de linha de comando do OpenSSL para realizar operações de criptografia e descriptografia. Um exemplo é apresentado a seguir.

Primeiro, criamos um arquivo de texto plano para ser criptografado:

$ echo Datatoencrypt > message.txt

Agora que o arquivo foi criado, podemos executar o OpenSSL com os parâmetros adequados para criptografar o arquivo **message.txt** usando AES de 256 bits no modo CBC:

$ openssl enc -aes-256-cbc -in message.txt -out message.bin

Será solicitado que você informe a senha:

enter aes-256-cbc encryption password:

Verifying - enter aes-256-cbc encryption password:

Após a operação ser concluída, será gerado um arquivo **message.bin** contendo os dados criptografados do arquivo **message.txt**. Podemos visualizar esse arquivo, que mostrará o conteúdo criptografado de **message.txt**:

$ cat message.bin

Isso exibirá a saída criptografada, como abaixo:

Salted\_\_?W?~?׈;??G+??"f??%

Observe que **message.bin** é um arquivo binário. Às vezes, é desejável codificar esse arquivo binário em um formato de texto para compatibilidade ou interoperabilidade. Um formato comum de codificação de texto é o **base64**. Os seguintes comandos podem ser usados para criar uma mensagem codificada em base64:

$ openssl enc -base64 -in message.bin -out message.b64

Para visualizar o arquivo:

$ cat message.b64

Isso mostrará a saída codificada em base64, como abaixo:

U2FsdGVkX1/tEFeZfszXiB47pOt/RyuN/CJm/x/KBBw=

**Descriptografando usando AES com OpenSSL**

Para descriptografar o arquivo criptografado, usamos o comando OpenSSL a seguir, fornecendo o arquivo de entrada criptografado e especificando a saída descriptografada:

$ openssl enc -aes-256-cbc -d -in message.bin -out message\_decrypted.txt

Será solicitado que você informe a senha usada para criptografar o arquivo:

enter aes-256-cbc decryption password:

Após fornecer a senha correta, o arquivo message\_decrypted.txt conterá os dados originais:

$ cat message\_decrypted.txt

Saída:

Datatoencrypt

**Resumo do capítulo**

Este capítulo introduziu a criptografia de chave simétrica. Começamos com definições matemáticas básicas e primitivas criptográficas. Em seguida, introduzimos os conceitos de cifras de fluxo e de bloco, juntamente com os modos de funcionamento das cifras de bloco. Além disso, introduzimos alguns exercícios práticos usando OpenSSL para complementar os conceitos teóricos abordados. No próximo capítulo, apresentaremos a criptografia de chave pública, amplamente utilizada na tecnologia blockchain e com propriedades muito interessantes.

Neste capítulo, você aprendeu sobre os conceitos fundamentais da criptografia simétrica, incluindo:

* A importância da confidencialidade, integridade, autenticação e não repúdio.
* Primitivas criptográficas, incluindo números aleatórios, funções de hash, e criptografia simétrica.
* As propriedades importantes das funções de hash e seu uso nas árvores de Merkle, fundamentais para a tecnologia blockchain.
* Os detalhes dos algoritmos SHA-256 e SHA-3 (Keccak), amplamente utilizados em sistemas blockchain.
* Os princípios da criptografia simétrica, com foco em cifras de fluxo e cifras de bloco, além dos modos de operação como ECB, CBC e CTR.
* O funcionamento e a importância do AES como padrão dominante de criptografia simétrica.
* Como usar o OpenSSL para criptografar e descriptografar arquivos usando AES na prática.

Este conhecimento serve como base para entender as primitivas criptográficas usadas no blockchain, que serão exploradas em capítulos posteriores.