**4 Criptografia autenticada**

Confidencialidade trata de esconder dados de olhos indesejados, e a criptografia é o meio de alcançar isso. A criptografia é aquilo para o qual a ciência da criptografia foi inicialmente inventada; foi o que preocupou a maioria dos primeiros criptógrafos. Eles se perguntavam: "Como podemos impedir que observadores compreendam nossas conversas?"

Enquanto a ciência e seus avanços floresceram inicialmente a portas fechadas, beneficiando apenas governos e seus militares, agora ela está aberta em todo o mundo. Hoje, a criptografia é usada em todos os lugares para adicionar privacidade e segurança em diferentes aspectos de nossas vidas modernas. Neste capítulo, descobriremos o que realmente é a criptografia, que tipos de problemas ela resolve e como as aplicações de hoje fazem amplo uso deste primitivo criptográfico.

**NOTA** Para este capítulo, você precisará ter lido o capítulo 3 sobre códigos de autenticação de mensagens.

Este capítulo aborda:

* Criptografia simétrica vs. criptografia autenticada
* Algoritmos populares de criptografia autenticada
* Outros tipos de criptografia simétrica

**4.1 O que é um cifrador?**

É como quando você usa gírias para falar com seus irmãos sobre o que fará depois da escola, para que sua mãe não saiba o que estão planejando.  
— Natanael L. (2020, <https://twitter.com/Natanael_L>)

Vamos imaginar que nossos dois personagens, Alice e Bob, querem trocar algumas mensagens de forma privada. Na prática, eles têm muitos meios à sua disposição (correio, telefones, internet, etc.), e cada um desses meios é, por padrão, inseguro. O carteiro pode abrir suas cartas; os operadores de telecomunicação podem espionar suas chamadas e mensagens de texto; provedores de internet ou quaisquer servidores na rede que estejam entre Alice e Bob podem acessar o conteúdo dos pacotes sendo trocados.

Sem mais delongas, vamos apresentar o salvador de Alice e Bob: o algoritmo de criptografia (também chamado de cifrador). Por ora, vamos imaginar este novo algoritmo como uma caixa preta que Alice pode usar para criptografar suas mensagens para Bob. Ao criptografar uma mensagem, Alice a transforma em algo que parece aleatório. O algoritmo de criptografia toma:

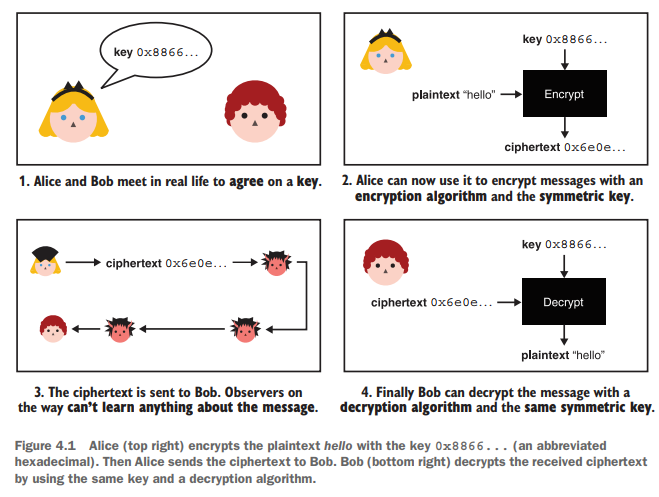
* **Uma chave secreta** — É crucial que este elemento seja imprevisível, aleatório e bem protegido, pois a segurança do algoritmo de criptografia depende diretamente do segredo da chave. Falarei mais sobre isso no capítulo 8, sobre segredos e aleatoriedade.
* **Algum texto em claro (plaintext)** — Isto é o que você quer criptografar. Pode ser algum texto, uma imagem, um vídeo ou qualquer coisa que possa ser traduzida em bits.

Este processo de criptografia produz um **texto cifrado (ciphertext)**, que é o conteúdo criptografado. Alice pode usar com segurança um dos meios listados anteriormente para enviar este texto cifrado para Bob. O texto cifrado parecerá aleatório para qualquer pessoa que não conheça a chave secreta, e nenhuma informação sobre o conteúdo da mensagem (o plaintext) será vazada. Uma vez que Bob recebe este texto cifrado, ele pode usar um algoritmo de descriptografia para revertê-lo ao plaintext original. A descriptografia toma:

* **Uma chave secreta** — É a mesma chave secreta que Alice usou para criar o texto cifrado. Como a mesma chave é usada para ambos os algoritmos, às vezes chamamos a chave de **chave simétrica**. Por isso, às vezes especificamos que estamos usando criptografia simétrica e não apenas criptografia.
* **Algum texto cifrado** — É a mensagem criptografada que Bob recebe de Alice.

O processo então revela o plaintext original. A <IMAGEM> ilustra este fluxo.

A criptografia permite que Alice transforme sua mensagem em algo que parece aleatório e que pode ser transmitido com segurança para Bob. A descriptografia permite que Bob reverta a mensagem criptografada de volta à mensagem original. Este novo primitivo criptográfico fornece confidencialidade (ou segredo, ou privacidade) às suas mensagens.



* 1. Alice e Bob se conheceram na vida real para combinar a chave
  2. Alice agora pode usá-lo para criptografar mensagens com uma chave simétrica de algoritmo de criptografia
  3. O texto cifrado é enviado para Bob. Observadores a caminho não conseguem descobrir nada sobre a mensagem.
  4. Finalmente, Bob pode descriptografar a mensagem com um algoritmo de descriptografia com a mesma chave simétrica

Figura 4.1 Alice (canto superior direito) criptografa o texto simples "hello" com a chave 0x8866... ​​(um hexadecimal abreviado). Em seguida, Alice envia o texto cifrado para Bob. Bob (canto inferior direito) descriptografa o texto cifrado recebido usando a mesma chave e um algoritmo de descriptografia.

**NOTA** Como Alice e Bob concordam em usar a mesma chave simétrica? Por enquanto, assumiremos que um deles teve acesso a um algoritmo que gera chaves imprevisíveis, e que se encontraram pessoalmente para trocar a chave. Na prática, como iniciar tais protocolos com segredos compartilhados é frequentemente um dos grandes desafios que as empresas precisam resolver. Neste livro, você verá muitas soluções diferentes para este problema.

**4.2 O cifrador de bloco AES (Advanced Encryption Standard)**

Em 1997, o NIST iniciou uma competição aberta para um novo Padrão de Criptografia Avançada (AES), com o objetivo de substituir o algoritmo Data Encryption Standard (DES), seu padrão anterior de criptografia que já começava a apresentar sinais de envelhecimento. A competição durou três anos, durante os quais 15 designs diferentes foram submetidos por equipes de criptógrafos de diferentes países. Ao final da competição, apenas uma submissão, Rijndael, de Vincent Rijmen e Joan Daemen, foi nomeada vencedora.

<imagem>

Em 2001, o NIST lançou o AES como parte da publicação FIPS (Federal Information Processing Standards) 197. O AES, o algoritmo descrito no padrão FIPS, ainda é o principal cifrador utilizado atualmente. Nesta seção, explico como o AES funciona.

**4.2.1 Quanta segurança o AES fornece?**

O AES oferece três versões diferentes:

* **AES-128** usa uma chave de 128 bits (16 bytes),
* **AES-192** usa uma chave de 192 bits (24 bytes),
* **AES-256** usa uma chave de 256 bits (32 bytes).

O tamanho da chave dita o nível de segurança — quanto maior, mais forte. Contudo, a maioria das aplicações utiliza o AES-128, pois oferece segurança suficiente (128 bits de segurança).

O termo **segurança em bits** é comumente usado para indicar a segurança de algoritmos criptográficos. Por exemplo, o AES-128 especifica que o melhor ataque que conhecemos hoje exigiria cerca de 2¹²⁸ operações. Este número é gigantesco, e é o nível de segurança que a maioria das aplicações busca.

Quão grande é 2¹²⁸? Observe que a quantidade entre duas potências de 2 dobra a cada passo. Por exemplo, 2³ é o dobro de 2². Se 2¹⁰⁰ operações já são praticamente impossíveis de se alcançar, imagine dobrar isso (2¹⁰¹). Para alcançar 2¹²⁸, você terá dobrado seu valor inicial 128 vezes! Em linguagem simples, 2¹²⁸ é:

**340 undecilhões 282 decilhões 366 nonilhões 920 octilhões 938 septilhões 463 sextilhões 463 quintilhões 374 quadrilhões 607 trilhões 431 bilhões 768 milhões 211 mil 456.**

É bastante difícil imaginar o quão grande é esse número, mas você pode assumir que nunca seremos capazes de atingir tal número na prática. Também não consideramos o espaço necessário para qualquer ataque complexo de larga escala funcionar, o que é igualmente enorme na prática.

Prevê-se que o AES-128 continuará seguro por um longo tempo. A menos que avanços na criptoanálise descubram alguma vulnerabilidade ainda desconhecida que reduza o número de operações necessárias para atacar o algoritmo.

**4.2.2 A interface do AES**

Olhando para a interface do AES para criptografia, vemos o seguinte:

* O algoritmo aceita uma chave de comprimento variável, como discutido anteriormente.
* Também aceita um plaintext de exatamente 128 bits.
* Produz um ciphertext de exatamente 128 bits.

**Segurança em bits é um limite superior**  
O fato de que uma chave de 128 bits oferece 128 bits de segurança é específico ao AES; não é uma regra de ouro. Uma chave de 128 bits usada em algum outro algoritmo poderia, teoricamente, fornecer menos de 128 bits de segurança. Enquanto uma chave de 128 bits pode fornecer menos segurança, nunca fornecerá mais (há sempre o ataque por força bruta). Testar todas as chaves possíveis levaria no máximo 2¹²⁸ operações, reduzindo a segurança para no mínimo 128 bits.

Como o AES criptografa um plaintext de tamanho fixo, o chamamos de **cifrador de bloco** (*block cipher*). Alguns outros cifradores podem criptografar plaintexts de comprimento arbitrário, como você verá mais adiante neste capítulo.

A operação de descriptografia é exatamente o reverso disso: usa a mesma chave, um ciphertext de 128 bits e retorna o plaintext original de 128 bits. Efetivamente, a descriptografia reverte a criptografia. Isso é possível porque as operações de criptografia e descriptografia são **determinísticas**; produzem os mesmos resultados não importa quantas vezes sejam executadas.

Em termos técnicos, um cifrador de bloco com uma chave é uma **permutação**: ele mapeia todos os possíveis plaintexts para todos os possíveis ciphertexts (veja o exemplo na <IMAGEM>). Alterar a chave altera esse mapeamento. Uma permutação também é reversível: a partir de um ciphertext, existe um mapeamento de volta ao plaintext correspondente (caso contrário, a descriptografia não funcionaria).

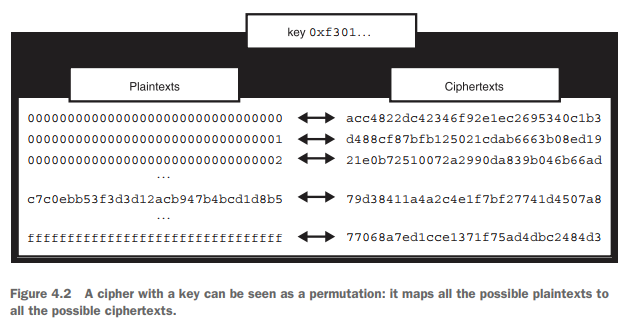


Figura 4.2 Uma cifra com uma chave pode ser vista como uma permutação: ela mapeia todos os textos simples possíveis para todos os textos cifrados possíveis.

Obviamente, não temos espaço para listar todos os possíveis plaintexts e seus ciphertexts associados. Isso seriam 2¹²⁸ mapeamentos para um cifrador de bloco de 128 bits. Em vez disso, projetamos construções como o AES, que se comportam como permutações e são randomizadas por uma chave. Dizemos que são **permutações pseudoaleatórias (PRPs)**.

**4.2.3 O funcionamento interno do AES**

Vamos nos aprofundar um pouco nas entranhas do AES para ver o que há dentro. Note que o AES enxerga o estado do plaintext durante o processo de criptografia como uma matriz 4x4 de bytes (como você pode ver na <IMAGEM>).

Na prática, isso não faz muita diferença, mas é assim que o AES é definido. Nos bastidores, o AES funciona como muitos outros primitivos criptográficos simétricos chamados de **cifradores de bloco**, que criptografam blocos de tamanho fixo. O AES também possui uma **função de rodada (round function)** que itera diversas vezes, partindo da entrada original (o plaintext). Eu ilustro isso na <IMAGEM>.

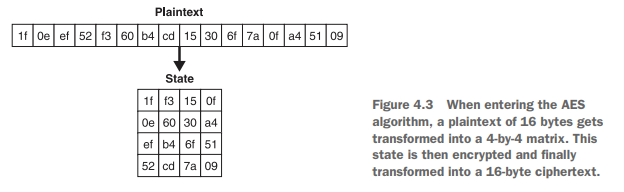


Figura 4.3 Ao inserir o algoritmo AES, um texto simples de 16 bytes é transformado em uma matriz 4 por 4. Esse estado é então criptografado e, finalmente, transformado em um texto cifrado de 16 bytes.

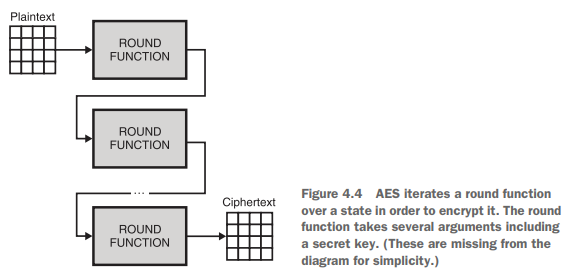


Figura 4.4 O AES itera uma função round sobre um estado para criptografá-lo. A função round recebe vários argumentos, incluindo uma chave secreta. (Estes argumentos estão ausentes no diagrama para simplificar.)

Cada chamada à função de rodada transforma o estado ainda mais, eventualmente produzindo o ciphertext. Cada rodada usa uma **chave de rodada (round key)** diferente, que é derivada da chave simétrica principal (durante o que chamamos de *key schedule*). Isso permite que uma mínima alteração nos bits da chave simétrica gere uma criptografia completamente diferente (um princípio chamado de **difusão**).

A função de rodada consiste em múltiplas operações que misturam e transformam os bytes do estado. A função de rodada do AES usa especificamente quatro subfunções diferentes. Embora evitemos explicar exatamente como as subfunções funcionam (você pode encontrar essas informações em qualquer livro sobre AES), seus nomes são: **SubBytes, ShiftRows, MixColumns e AddRoundKey**. As três primeiras são facilmente reversíveis (é possível encontrar a entrada a partir da saída da operação), mas a última não é. Ela realiza uma operação **XOR (OU-exclusivo)** com a chave de rodada e o estado e, portanto, precisa do conhecimento da chave de rodada para ser revertida. Eu ilustro o que entra em uma rodada na <IMAGEM>.

O número de iterações da função de rodada no AES — que normalmente é pequeno o suficiente para ser prático — foi escolhido para frustrar a criptoanálise.

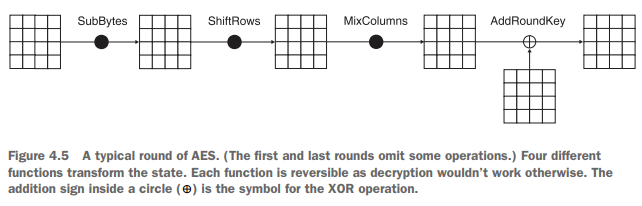


Figura 4.5 Uma rodada típica de AES. (A primeira e a última rodadas omitem algumas operações.) Quatro funções diferentes transformam o estado. Cada função é reversível, pois a descriptografia não funcionaria de outra forma. O sinal de adição dentro de um círculo (⊕) é o símbolo da operação XOR.

Por exemplo, existem ataques extremamente eficientes (que recuperam a chave) em variantes de três rodadas do AES-128. Ao iterar muitas vezes, o cifrador transforma o plaintext em algo que não se assemelha em nada ao original. A menor alteração no plaintext também retorna um ciphertext completamente diferente. Este princípio é chamado de **efeito avalanche**.

**NOTA**  
Os algoritmos criptográficos do mundo real são tipicamente comparados pela segurança, tamanho e velocidade que oferecem. Já falamos sobre a segurança e o tamanho do AES; sua segurança depende do tamanho da chave, e ele pode criptografar blocos de dados de 128 bits por vez. Em termos de velocidade, muitos fabricantes de CPU implementaram o AES por hardware. Por exemplo, as **AES New Instructions (AES-NI)** são um conjunto de instruções disponível em CPUs Intel e AMD, que podem ser usadas para implementar criptografia e descriptografia de AES de forma eficiente. Essas instruções especiais tornam o AES extremamente rápido na prática.

Uma pergunta que você pode ainda ter é: como criptografar mais ou menos que 128 bits com o AES? Vou responder isso a seguir.

**4.3 O pinguim criptografado e o modo de operação CBC**

Agora que apresentamos o cifrador de bloco AES e explicamos um pouco sobre seu funcionamento interno, vamos ver como usá-lo na prática. O problema com um cifrador de bloco é que ele só consegue criptografar um bloco por vez. Para criptografar algo que não tenha exatamente 128 bits, precisamos usar um preenchimento (*padding*) bem como um **modo de operação**. Então, vamos ver do que se tratam esses dois conceitos.

Imagine que você queira criptografar uma mensagem longa. Ingenuamente, você poderia dividi-la em blocos de 16 bytes (o tamanho de bloco do AES). Então, se o último bloco de plaintext for menor que 16 bytes, você poderia adicionar alguns bytes ao final até que o plaintext fique com 16 bytes de comprimento. É disso que se trata o *padding*.

Existem várias maneiras de especificar como escolher esses bytes de preenchimento, mas o aspecto mais importante do *padding* é que ele precisa ser reversível. Após a descriptografia do ciphertext, devemos ser capazes de remover o *padding* para recuperar a mensagem original sem preenchimento. Simplesmente adicionar bytes aleatórios, por exemplo, não funcionaria porque não seria possível discernir se os bytes aleatórios faziam parte da mensagem original ou não.

O mecanismo de preenchimento mais popular é frequentemente chamado de **PKCS#7 padding**, que apareceu pela primeira vez no padrão PKCS#7 publicado pela RSA (uma empresa) no final dos anos 1990. O PKCS#7 padding especifica uma única regra: o valor de cada byte de preenchimento deve ser igual ao comprimento do preenchimento necessário. E se o plaintext já tiver 16 bytes? Então adicionamos um bloco completo de preenchimento com o valor 16. Eu ilustro isso visualmente na <IMAGEM>. Para remover o preenchimento, basta verificar o valor do último byte do plaintext e interpretá-lo como o comprimento do preenchimento a ser removido.

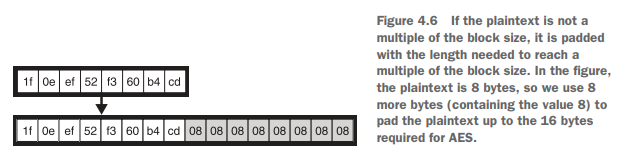


Figura 4.6 Se o texto simples não for um múltiplo do tamanho do bloco, ele será preenchido com o comprimento necessário para atingir um múltiplo do tamanho do bloco. Na figura, o texto simples tem 8 bytes, então usamos mais 8 bytes (contendo o valor 8) para preencher o texto simples até os 16 bytes necessários para o AES.

Agora, há um grande problema que preciso mencionar. Até agora, para criptografar uma mensagem longa, você simplesmente a dividiu em blocos de 16 bytes (e talvez tenha preenchido o último bloco). Essa forma ingênua é chamada de **modo de operação de livro de códigos eletrônico (Electronic Codebook - ECB)**. Como você aprendeu, a criptografia é determinística, e assim, criptografar o mesmo bloco de plaintext duas vezes leva ao mesmo ciphertext. Isso significa que, ao criptografar cada bloco individualmente, o ciphertext resultante pode ter padrões repetitivos.

Isso pode parecer aceitável, mas permitir essas repetições leva a muitos problemas. O mais óbvio é que elas vazam informações sobre o plaintext. A ilustração mais famosa disso é o **pinguim do ECB**, ilustrado na <IMAGEM>.

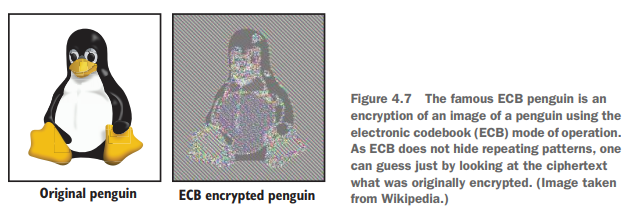


Figura 4.7 O famoso pinguim ECB é uma criptografia de uma imagem de um pinguim usando o modo de operação do livro de códigos eletrônico (ECB). Como o ECB não oculta padrões repetidos, é possível adivinhar, apenas observando o texto cifrado, o que foi originalmente criptografado. (Imagem retirada da Wikipédia.)

Para criptografar mais de 128 bits de plaintext com segurança, existem modos de operação melhores que “randomizam” a criptografia. Um dos modos de operação mais populares para o AES é o **cipher block chaining (CBC)**. O CBC funciona para qualquer cifrador de bloco determinístico (não apenas o AES) ao utilizar um valor adicional chamado **vetor de inicialização (IV – initialization vector)** para randomizar a criptografia. Por causa disso, o IV tem o tamanho do bloco (16 bytes para o AES) e deve ser aleatório e imprevisível.

Para criptografar com o modo CBC, começamos gerando um IV aleatório de 16 bytes (o capítulo 8 mostrará como fazer isso), depois aplicamos o XOR do IV gerado com os primeiros 16 bytes do plaintext antes de criptografá-los. Isso efetivamente randomiza a criptografia. De fato, se o mesmo plaintext for criptografado duas vezes com IVs diferentes, o modo de operação produzirá dois ciphertexts diferentes.

Se houver mais plaintext a ser criptografado, usamos o ciphertext anterior (como usamos o IV anteriormente) para aplicá-lo em XOR com o próximo bloco de plaintext antes de criptificá-lo. Isso randomiza o próximo bloco de criptografia também. Lembre-se: a criptografia de algo deve ser imprevisível e tão boa quanto a aleatoriedade usada para criar nosso verdadeiro IV. A <IMAGEM> ilustra a criptografia CBC.

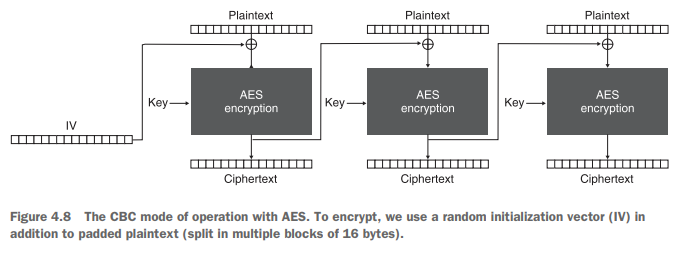


Figura 4.8 Modo de operação CBC com AES. Para criptografar, usamos um vetor de inicialização aleatório (IV) além de texto simples preenchido (dividido em vários blocos de 16 bytes).

Para descriptografar com o modo CBC, revertamos as operações. Como o IV é necessário, ele deve ser transmitido em texto claro junto com o ciphertext. Como o IV deve ser aleatório, nenhuma informação é vazada ao se observar seu valor. Eu ilustro a descriptografia CBC na <IMAGEM>.

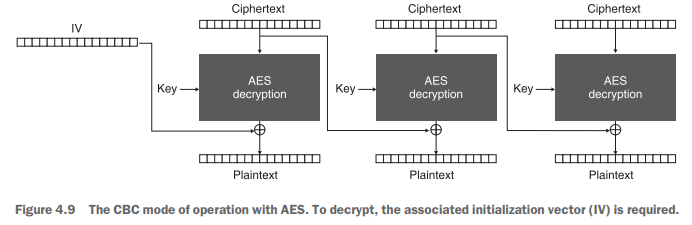


Figura 4.9 Modo de operação CBC com AES. Para descriptografar, é necessário o vetor de inicialização (IV) associado.

Parâmetros adicionais como IVs são prevalentes em criptografia. No entanto, frequentemente são mal compreendidos e são uma grande fonte de vulnerabilidades. Com o modo CBC, um IV precisa ser único (não pode se repetir) além de imprevisível (precisa ser verdadeiramente aleatório). Esses requisitos podem falhar por diversos motivos. Como os desenvolvedores frequentemente ficam confusos com IVs, algumas bibliotecas criptográficas removeram a possibilidade de especificar um IV ao criptografar com CBC, gerando-o automaticamente de forma aleatória.

**AVISO**  
Quando um IV se repete ou é previsível, a criptografia se torna determinística novamente e uma série de ataques engenhosos se tornam possíveis. Este foi o caso do famoso ataque **BEAST (Browser Exploit Against SSL/TLS)** no protocolo TLS. Note também que outros algoritmos podem ter requisitos diferentes para IVs. Por isso é sempre importante ler o manual. Detalhes perigosos estão nas letras miúdas.

Note que um modo de operação e um *padding* ainda não são suficientes para tornar um cifrador utilizável. Você está prestes a ver o porquê na próxima seção.

**4.4 Falta de autenticidade, portanto AES-CBC-HMAC**

Até agora, não abordamos uma falha fundamental: tanto o ciphertext quanto o IV, no caso do CBC, ainda podem ser modificados por um atacante. De fato, não há nenhum mecanismo de integridade para prevenir isso! Alterações no ciphertext ou no IV podem gerar mudanças inesperadas na descriptografia. Por exemplo, no AES-CBC (AES usado com o modo de operação CBC), um atacante pode inverter bits específicos do plaintext invertendo bits no IV e no ciphertext. Eu ilustro este ataque na <IMAGEM>.

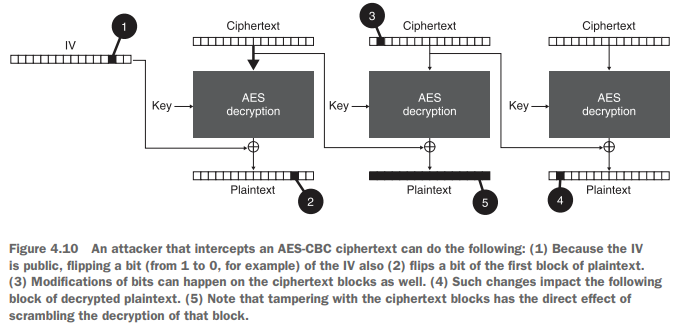


Figura 4.10 Um invasor que intercepta um texto cifrado AES-CBC pode fazer o seguinte: (1) Como o IV é público, inverter um bit (de 1 para 0, por exemplo) do IV também (2) inverte um bit do primeiro bloco de texto simples. (3) Modificações de bits também podem ocorrer nos blocos de texto cifrado. (4) Essas alterações afetam o bloco seguinte de texto simples descriptografado. (5) Observe que a adulteração dos blocos de texto cifrado tem o efeito direto de embaralhar a descriptografia desse bloco.

Consequentemente, um cifrador ou um modo de operação não devem ser usados isoladamente. Eles carecem de algum tipo de proteção de integridade para garantir que um ciphertext e seus parâmetros associados (aqui, o IV) não possam ser modificados sem disparar algum alarme.

Para prevenir modificações no ciphertext, podemos usar os códigos de autenticação de mensagens (MACs) que vimos no capítulo 3. Para o AES-CBC, normalmente usamos o **HMAC (hash-based MAC)** em combinação com a função de hash **SHA-256** para fornecer integridade. Aplicamos o MAC após preencher o plaintext e criptografá-lo, cobrindo tanto o ciphertext quanto o IV; caso contrário, um atacante ainda poderia modificar o IV sem ser detectado.

**AVISO**  
Essa construção é chamada de **Encrypt-then-MAC**. As alternativas (como **MAC-then-Encrypt**) podem, às vezes, levar a ataques engenhosos (como o famoso **ataque oracle de padding de Vaudenay**) e, por isso, são evitadas na prática.

A tag de autenticação criada pode ser transmitida junto com o IV e o ciphertext. Normalmente, todos são concatenados juntos, como ilustra a <IMAGEM>. Além disso, é uma boa prática usar chaves diferentes para o AES-CBC e para o HMAC.

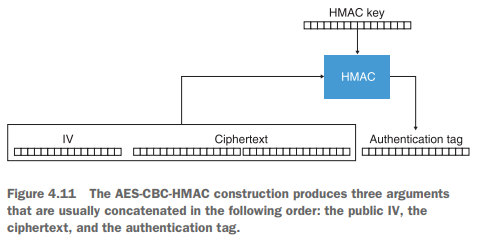


Figura 4.11 A construção AES-CBC-HMAC produz três argumentos que geralmente são concatenados na seguinte ordem: o IV público, o texto cifrado e a tag de autenticação.

Antes da descriptografia, a tag precisa ser verificada (em tempo constante, como você viu no capítulo 3). A combinação de todos esses algoritmos é chamada de **AES-CBC-HMAC** e foi um dos modos de criptografia autenticada mais amplamente utilizados até começarmos a adotar construções modernas tudo-em-um.

**AVISO**  
O AES-CBC-HMAC não é a construção mais amigável para desenvolvedores. Frequentemente é mal implementado e possui algumas armadilhas perigosas quando não usado corretamente (por exemplo, o IV de cada criptografia deve ser imprevisível). Eu dediquei algumas páginas a este algoritmo pois ainda é amplamente usado e continua funcionando, mas recomendo não usá-lo em favor das construções mais modernas que apresento a seguir.

**4.5 Construções tudo-em-um: Criptografia autenticada**

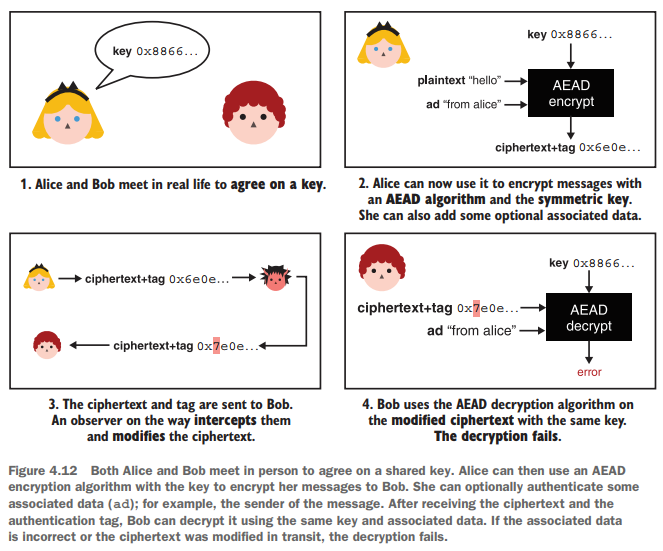
A história da criptografia não é bonita. Não apenas foi mal compreendido que criptografia sem autenticação é perigosa, mas também a aplicação incorreta da autenticação tem sido um erro sistêmico cometido por desenvolvedores. Por essa razão, muita pesquisa surgiu buscando padronizar **construções tudo-em-um** que simplificam o uso da criptografia para os desenvolvedores. No restante desta seção, apresento esse novo conceito bem como dois padrões amplamente adotados: **AES-GCM** e **ChaCha20-Poly1305**.

**4.5.1 O que é criptografia autenticada com dados associados (AEAD)?**

A forma mais atual de criptografar dados é usar uma construção tudo-em-um chamada **Authenticated Encryption with Associated Data (AEAD)** — Criptografia Autenticada com Dados Associados. A construção é extremamente próxima do que o AES-CBC-HMAC fornece, pois também oferece confidencialidade dos seus plaintexts enquanto detecta quaisquer modificações que possam ter ocorrido nos ciphertexts. Além disso, fornece um meio de autenticar dados associados.

O argumento de **dados associados** é opcional e pode ser vazio, ou pode conter metadados relevantes para a criptografia e descriptografia do plaintext. Esses dados não serão criptografados e são ou implícitos ou transmitidos junto com o ciphertext. Além disso, o tamanho do ciphertext é maior que o do plaintext pois agora contém uma **tag de autenticação adicional** (geralmente adicionada ao final do ciphertext).

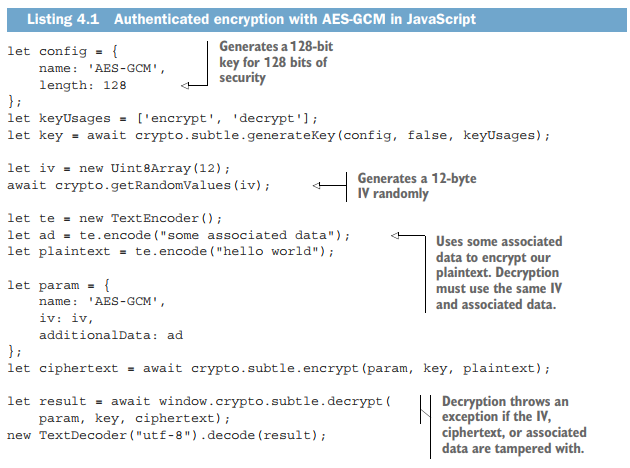
Para descriptografar o ciphertext, é necessário usar os mesmos dados associados (implícitos ou transmitidos). O resultado é ou um erro, indicando que o ciphertext foi modificado em trânsito, ou o plaintext original. Eu ilustro este novo primitivo na <IMAGEM>.

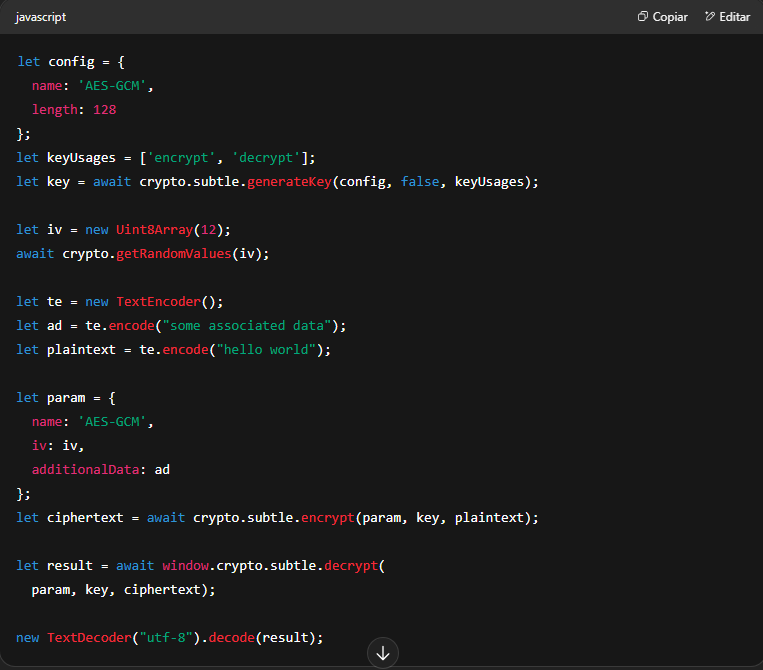


1. Alice e Bob se encontram na vida real para concordar sobre uma chave
2. Alice agora pode usá-lo para criptografar mensagens com um e o algoritmo AEAD chave simétrica. Ela também pode adicionar alguns dados associados opcionais
3. O texto cifrado e a etiqueta são enviados para Bob. Um observador no caminho os intercepta e o texto cifrado
4. Bob usa o algoritmo de descriptografia AEAD com a mesma chave. texto cifrado modificado. A descriptografia falha.

Figura 4.12 Alice e Bob se encontram pessoalmente para concordar com uma chave compartilhada. Alice pode então usar um algoritmo de criptografia AEAD com a chave para criptografar suas mensagens para Bob. Ela pode, opcionalmente, autenticar alguns dados associados (anúncio); por exemplo, o remetente da mensagem. Após receber o texto cifrado e a tag de autenticação, Bob pode descriptografá-lo usando a mesma chave e os dados associados. Se os dados associados estiverem incorretos ou o texto cifrado tiver sido modificado durante o envio, a descriptografia falha.

Vamos ver como usar uma biblioteca criptográfica para criptografar e descriptografar com um primitivo de criptografia autenticada. Para isso, usaremos a linguagem JavaScript e a **Web Crypto API** (uma interface oficial suportada pela maioria dos navegadores que fornece funções criptográficas de baixo nível), como mostra o seguinte código:





**Nota:** A Web Crypto API é uma API de baixo nível e, como tal, não ajuda o desenvolvedor a evitar erros. Por exemplo, ela nos permite especificar um IV, o que é um padrão perigoso. Neste exemplo, usei o **AES-GCM**, que é o AEAD mais amplamente utilizado. A seguir, falaremos mais sobre o AES-GCM.

**4.5.2 O AEAD AES-GCM**

O AEAD mais amplamente utilizado é o **AES com o Galois/Counter Mode (AES-GCM)**. Ele foi projetado para alta performance aproveitando o suporte de hardware ao AES e utilizando um MAC (GMAC) que pode ser implementado de forma eficiente.

O AES-GCM foi incluído na publicação especial do NIST (SP 800-38D) desde 2007, e é o principal cifrador utilizado em protocolos criptográficos, incluindo várias versões do protocolo **TLS** que protege conexões com websites na internet. Podemos afirmar, efetivamente, que o AES-GCM criptografa a web.

O AES-GCM combina o modo de operação **Counter (CTR)** com o código de autenticação de mensagem **GMAC**. Primeiro, vejamos como o modo CTR funciona com o AES. A <IMAGEM> mostra como o AES é usado com o modo CTR.

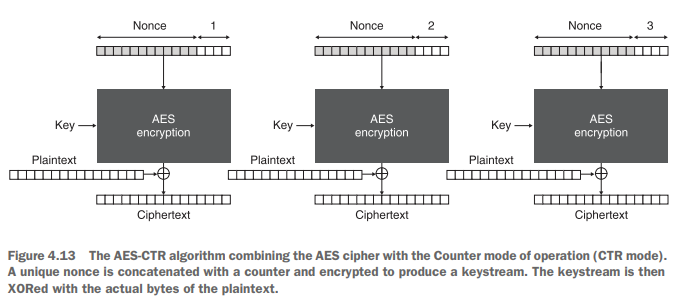


Figura 4.13 O algoritmo AES-CTR combina a cifra AES com o modo de operação Contador (modo CTR). Um nonce único é concatenado com um contador e criptografado para produzir um fluxo de chaves. O fluxo de chaves é então submetido a um XOR com os bytes reais do texto simples.

O AES-CTR usa o AES para criptografar um *nonce* concatenado com um contador (iniciando em 1), em vez de criptografar o plaintext. Este argumento adicional, "*nonce*" (de *number once*), serve ao mesmo propósito do IV: permite que o modo de operação randomize a criptografia AES. Os requisitos, entretanto, são um pouco diferentes do IV no modo CBC. Um *nonce* precisa ser único, mas não imprevisível. Uma vez que esse bloco de 16 bytes é criptografado, o resultado é chamado de **keystream** (fluxo de chave), e é aplicado um XOR com o plaintext real para produzir a criptografia.

**NOTA**  
Assim como IVs, *nonces* são um termo comum em criptografia, e são encontrados em diferentes primitivos criptográficos. *Nonces* podem ter requisitos diferentes, embora o nome normalmente indique que não devem se repetir. Mas, como sempre, o que importa é o que o manual diz, não o que o nome do argumento implica. De fato, o *nonce* do AES-GCM às vezes é chamado de IV.

O *nonce* no AES-CTR é de 96 bits (12 bytes) e ocupa a maior parte dos 16 bytes a serem criptografados. Os 32 bits (4 bytes) restantes servem como contador, iniciando em 1 e sendo incrementado para cada bloco criptografado até atingir seu valor máximo de 2³² – 1 = 4.294.967.295. Isso significa que, no máximo, 4.294.967.295 blocos de 128 bits podem ser criptografados com o mesmo *nonce* (ou seja, menos de 69 GB).

Se o mesmo *nonce* for usado duas vezes, o mesmo *keystream* será criado. Ao aplicar XOR nos dois ciphertexts, o *keystream* é cancelado e pode-se recuperar o XOR dos dois plaintexts. Isso pode ser devastador, especialmente se houver alguma informação sobre o conteúdo de um dos plaintexts.

A <IMAGEM> mostra um aspecto interessante do modo CTR: não é necessário *padding*. Dizemos que ele transforma um cifrador de bloco (AES) em um **cifrador de fluxo** (*stream cipher*). Ele criptografa o plaintext byte a byte.

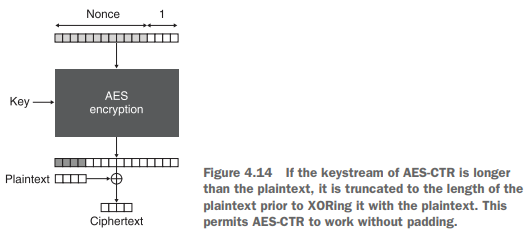


Figura 4.14 Se o fluxo de chaves do AES-CTR for maior que o texto simples, ele será truncado para o comprimento do texto simples antes de ser submetido a um XOR com o texto simples. Isso permite que o AES-CTR funcione sem preenchimento.

A segunda parte do AES-GCM é o **GMAC**. Trata-se de um MAC construído a partir de um hash com chave (chamado de **GHASH**). Em termos técnicos, o GHASH é um hash universal quase XORado (AXU — *almost XORed universal hash*), que também é chamado de **função imprevisível por diferença (DUF — *difference unpredictable function*)**. O requisito de tal função é mais fraco que o de um hash comum. Por exemplo, um AXU não precisa ser resistente a colisões. Graças a isso, o GHASH pode ser significativamente mais rápido. A <IMAGEM> ilustra o algoritmo GHASH.

**Cifradores de fluxo**  
Cifradores de fluxo são outra categoria de cifradores. Eles são diferentes dos cifradores de bloco porque podemos usá-los diretamente para criptografar um ciphertext aplicando XOR com um *keystream*. Não há necessidade de *padding* nem de um modo de operação, permitindo que o ciphertext tenha o mesmo comprimento do plaintext.

Na prática, não há muita diferença entre essas duas categorias de cifradores, pois cifradores de bloco podem ser facilmente transformados em cifradores de fluxo via o modo CTR. Mas, em teoria, os cifradores de bloco têm a vantagem de poderem ser úteis na construção de outras categorias de primitivas (semelhante ao que você viu no capítulo 2 com funções de hash).

Este também é um bom momento para observar que, por padrão, a criptografia não (ou dificilmente) esconde o comprimento do que está sendo criptografado. Por causa disso, o uso de compressão antes da criptografia pode levar a ataques se um atacante puder influenciar partes do que está sendo criptografado.

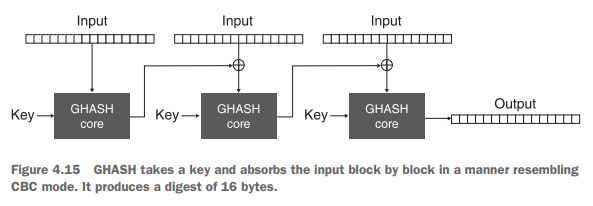


Figura 4.15 O GHASH recebe uma chave e absorve a entrada bloco por bloco, de maneira semelhante ao modo CBC. Ele produz um resumo de 16 bytes.

Para fazer o hash de algo com o GHASH, dividimos a entrada em blocos de 16 bytes e, então, os hashamos de uma forma semelhante ao modo CBC. Como esse hash recebe uma chave como entrada, teoricamente pode ser usado como MAC, mas apenas uma vez (caso contrário, o algoritmo quebra) — é um **MAC de uso único**. Como isso não é ideal para nós, usamos uma técnica (de **Wegman-Carter**) para transformar o GHASH em um MAC de múltiplos usos. Eu ilustro isso na <IMAGEM>.

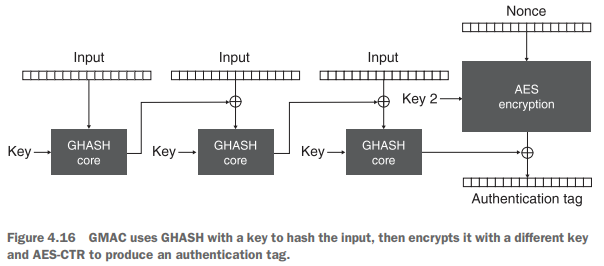


Figura 4.16 O GMAC usa GHASH com uma chave para fazer hash da entrada e, em seguida, criptografa-a com uma chave diferente e AES-CTR para produzir uma tag de autenticação.

O GMAC é, efetivamente, a criptografia da saída do GHASH com o AES-CTR (e uma chave diferente). Novamente, o *nonce* deve ser único; caso contrário, atacantes engenhosos podem recuperar a chave de autenticação usada pelo GHASH, o que seria catastrófico e permitiria forjar facilmente tags de autenticação.

Finalmente, o **AES-GCM** pode ser visto como uma combinação entrelaçada do modo CTR e do GMAC, semelhante à construção **Encrypt-then-MAC** que discutimos anteriormente. Eu ilustro todo o algoritmo na <IMAGEM>.

O contador começa em 1 para a criptografia, deixando o contador 0 reservado para criptografar a tag de autenticação criada pelo GHASH. O GHASH, por sua vez, usa uma chave independente H, que é a criptografia do bloco todo-zero com a chave K. Desta forma, não é necessário carregar duas chaves diferentes, pois a chave K basta para derivar a outra.

Como eu disse anteriormente, o *nonce* de 12 bytes do AES-GCM precisa ser único e, portanto, nunca deve se repetir. Note que ele não precisa ser aleatório. Consequentemente, algumas pessoas preferem usá-lo como um contador, iniciando em 1 e incrementando a cada criptografia. Neste caso, é necessário usar uma biblioteca criptográfica que permita ao usuário escolher o *nonce*. Isso possibilita criptografar 2¹²×⁸ – 1 mensagens antes de alcançar o valor máximo do *nonce*. Basta dizer: esse é um número impossível de ser atingido na prática.

Por outro lado, ter um contador significa que é necessário manter estado. Se uma máquina falhar no momento errado, pode ocorrer o reaproveitamento do *nonce*. Por essa razão, às vezes é preferível usar um *nonce* aleatório. De fato, algumas bibliotecas não permitem que os desenvolvedores escolham o *nonce* e o geram aleatoriamente.

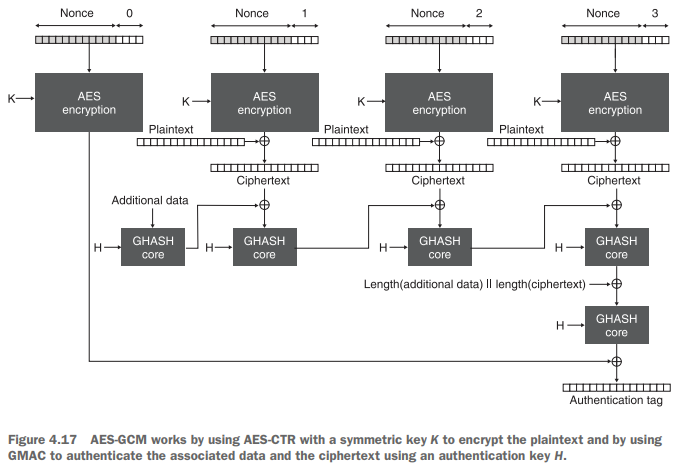


Figura 4.17 O AES-GCM funciona usando AES-CTR com uma chave simétrica K para criptografar o texto simples e usando GMAC para autenticar os dados associados e o texto cifrado usando uma chave de autenticação H.

Isso evita repetições com probabilidades tão pequenas que não deveriam ocorrer na prática. Ainda assim, quanto mais mensagens são criptografadas, mais *nonces* são utilizados e maiores se tornam as chances de colisão. Devido ao **limite do aniversário (birthday bound)**, sobre o qual falamos no capítulo 2, recomenda-se não criptografar mais que 2⁹²⁄³ ≈ 2³⁰ mensagens com a mesma chave ao gerar *nonces* aleatoriamente.

**Segurança além do limite do aniversário**  
2³⁰ mensagens é um número bastante grande. Pode nunca ser atingido em muitos cenários, mas a criptografia do mundo real frequentemente testa o limite do que é considerado razoável. Alguns sistemas de longa duração precisam criptografar muitas, muitas mensagens por segundo, eventualmente alcançando esses limites. A Visa, por exemplo, processa 150 milhões de transações por dia. Se precisasse criptografá-las com uma chave única, atingiria o limite de 2³⁰ mensagens em apenas uma semana. Nestes casos extremos, o **rekeying** (troca da chave usada na criptografia) pode ser uma solução. Existe também um campo de pesquisa chamado **segurança além do limite do aniversário** (*beyond birthday-bound security*) que visa aumentar o número máximo de mensagens que podem ser criptografadas com a mesma chave.

<imagem>

**4.5.3 ChaCha20-Poly1305**

O segundo AEAD de que falarei é o **ChaCha20-Poly1305**. Trata-se da combinação de dois algoritmos: o cifrador de fluxo **ChaCha20** e o MAC **Poly1305**. Ambos foram projetados separadamente por Daniel J. Bernstein para serem rápidos quando usados em software, ao contrário do AES, que é lento quando o suporte de hardware não está disponível. Em 2013, o Google padronizou o AEAD ChaCha20-Poly1305 para utilizá-lo em telefones Android baseados em processadores de baixo desempenho. Atualmente, é amplamente adotado por protocolos da internet como **OpenSSH, TLS e Noise**.

O ChaCha20 é uma modificação do cifrador de fluxo **Salsa20**, originalmente projetado por Daniel J. Bernstein por volta de 2005. Ele foi um dos algoritmos nomeados na competição **ESTREAM** (<https://www.ecrypt.eu.org/stream/>). Como todo cifrador de fluxo, o algoritmo produz um **keystream**, uma sequência de bytes aleatórios com o mesmo comprimento do plaintext. Ele então aplica XOR com o plaintext para criar o ciphertext. Para descriptografar, o mesmo algoritmo é usado para produzir o mesmo *keystream*, que é aplicado com XOR sobre o ciphertext para recuperar o plaintext. Eu ilustro ambos os fluxos na <IMAGEM>.

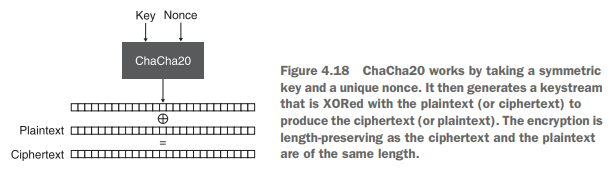


Figura 4.18 O ChaCha20 funciona usando uma chave simétrica e um nonce único. Em seguida, ele gera um fluxo de chaves que é submetido a um XOR com o texto simples (ou texto cifrado) para produzir o texto cifrado (ou texto simples). A criptografia preserva o comprimento, pois o texto cifrado e o texto simples têm o mesmo comprimento.

Por dentro, o ChaCha20 gera um *keystream* chamando repetidamente uma função de bloco que produz múltiplos blocos de 64 bytes de *keystream*. A função de bloco recebe:

* Uma chave de 256 bits (32 bytes), como o AES-256;
* Um *nonce* de 92 bits (12 bytes), como o AES-GCM;
* Um contador de 32 bits (4 bytes), como o AES-GCM.

O processo de criptografia é o mesmo que com o AES-CTR (eu ilustro esse fluxo na <IMAGEM>):

1. Executa-se a função de bloco, incrementando o contador a cada vez, até produzir *keystream* suficiente;
2. O *keystream* é truncado para o comprimento do plaintext;
3. Aplica-se XOR entre o *keystream* e o plaintext.

Devido ao limite superior do contador, o ChaCha20 pode criptografar tantas mensagens quanto o AES-GCM (já que ambos são parametrizados por um *nonce* semelhante). Como a saída criada por esta função de bloco é muito maior, o tamanho de uma mensagem que pode ser criptografada também é impactado. Pode-se criptografar uma mensagem de tamanho 2³² × 64 bytes ≈ 274 GB. Se um *nonce* for reutilizado para criptografar um plaintext, problemas semelhantes aos do AES-GCM surgem.

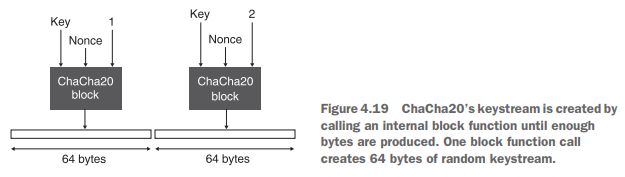


Figura 4.19 O fluxo de chaves do ChaCha20 é criado chamando uma função de bloco interna até que bytes suficientes sejam produzidos. Uma chamada de função de bloco cria 64 bytes de fluxo de chaves aleatório.

Um observador pode obter o XOR dos dois plaintexts aplicando XOR sobre os dois ciphertexts, e também pode recuperar a chave de autenticação para o *nonce*. Estes são problemas graves que podem permitir que um atacante forge mensagens!

Dentro da função de bloco do ChaCha20, forma-se um **estado**. A <IMAGEM> ilustra esse estado.

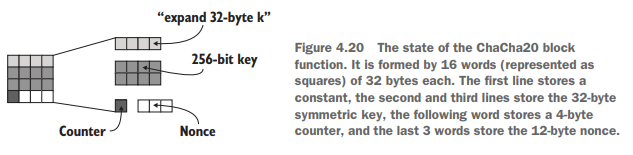


Figura 4.20 O estado da função de bloco ChaCha20. Ela é formada por 16 palavras (representadas como quadrados) de 32 bytes cada. A primeira linha armazena uma constante, a segunda e a terceira linhas armazenam a chave simétrica de 32 bytes, a palavra seguinte armazena um contador de 4 bytes e as últimas 3 palavras armazenam o nonce de 12 bytes.

Este estado é então transformado em 64 bytes de *keystream* iterando uma função de rodada 20 vezes (daí o número 20 no nome do algoritmo). Isto é semelhante ao que foi feito com o AES e sua função de rodada. A função de rodada, por sua vez, chama uma **função Quarter Round (QR)** 4 vezes por rodada, atuando em diferentes palavras do estado interno a cada vez, dependendo se o número da rodada é ímpar ou par. A <IMAGEM> mostra esse processo.

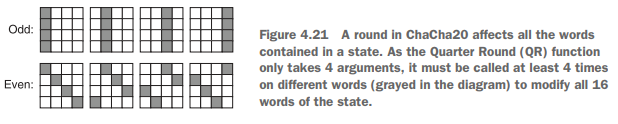


Figura 4.21 Uma rodada no ChaCha20 afeta todas as palavras contidas em um estado. Como a função Quarter Round (QR) aceita apenas 4 argumentos, ela deve ser chamada pelo menos 4 vezes em palavras diferentes (em cinza no diagrama) para modificar todas as 16 palavras do estado.

**O tamanho dos nonces e contadores**  
O tamanho dos *nonces* e dos contadores não é (na prática) sempre o mesmo em todos os lugares (tanto para o AES-GCM quanto para o ChaCha20-Poly1305), mas são os valores recomendados pelos padrões adotados. Ainda assim, algumas bibliotecas criptográficas aceitam tamanhos diferentes de *nonce*, e algumas aplicações aumentam o tamanho do contador (ou do *nonce*) para permitir a criptografia de mensagens maiores (ou mais mensagens). Aumentar o tamanho de um componente necessariamente diminui o tamanho do outro.

Para prevenir isso, mantendo a capacidade de criptografar um grande número de mensagens sob uma única chave, existem outros padrões como o **XChaCha20-Poly1305**. Esses padrões aumentam o tamanho do *nonce* mantendo o restante intacto, o que é importante em casos onde o *nonce* precisa ser gerado aleatoriamente em vez de ser um contador rastreado no sistema.

<imagem>

A função QR recebe quatro argumentos diferentes e os atualiza utilizando apenas operações de soma (Add), rotação (Rotate) e XOR. Dizemos que é um cifrador de fluxo **ARX (Add-Rotate-XOR)**. Isso torna o ChaCha20 extremamente fácil de implementar e rápido em software.

O **Poly1305** é um MAC criado via a técnica de Wegman-Carter, muito parecido com o GMAC de que falamos anteriormente. A <IMAGEM> ilustra este MAC criptográfico.

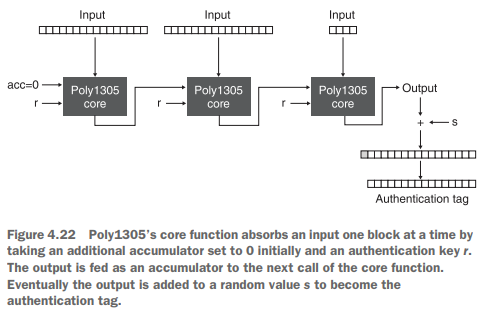


Figura 4.22 A função principal do Poly1305 absorve uma entrada, bloco por bloco, recebendo um acumulador adicional definido como 0 inicialmente e uma chave de autenticação r. A saída é alimentada como um acumulador para a próxima chamada da função principal. Eventualmente, a saída é adicionada a um valor aleatório s para se tornar a tag de autenticação.

Na figura, **r** pode ser visto como a chave de autenticação do esquema, assim como a chave de autenticação **H** no GMAC. E **s** torna o MAC seguro para múltiplos usos ao criptografar o resultado, portanto, deve ser único para cada uso.

A função central do Poly1305 mistura a chave com o acumulador (inicializado em 0) e a mensagem a ser autenticada. As operações são multiplicações simples módulo uma constante **P**.

**NOTA**  
Obviamente, muitos detalhes estão ausentes da nossa descrição. Raramente menciono como codificar dados ou como alguns argumentos devem ser preenchidos (*padded*) antes de serem processados. Todas essas são especificidades de implementação que não importam para nós enquanto buscamos uma intuição de como essas coisas funcionam.

Eventualmente, podemos usar o ChaCha20 e um contador configurado como 0 para gerar um *keystream* e derivar os valores de 16 bytes **r** e **s** de que precisamos para o Poly1305. Eu ilustro o AEAD resultante na <IMAGEM>.

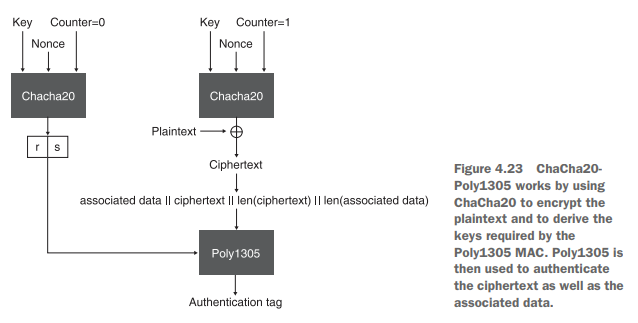


Figura 4.23 ChaCha20 - O Poly1305 funciona usando o ChaCha20 para criptografar o texto simples e derivar as chaves exigidas pelo MAC do Poly1305. O Poly1305 é então usado para autenticar o texto cifrado, bem como os dados associados.

O algoritmo ChaCha20 normal é primeiro usado para derivar os segredos de autenticação **r** e **s** necessários para o Poly1305. O contador é então incrementado, e o ChaCha20 é usado para criptografar o plaintext. Após isso, os dados associados e o ciphertext (e seus respectivos comprimentos) são passados ao Poly1305 para criar uma tag de autenticação.

Para descriptografar, o mesmo processo é aplicado. O ChaCha20 primeiro verifica a autenticação do ciphertext e dos dados associados através da tag recebida. Em seguida, descriptografa o ciphertext.

**4.6 Outros tipos de criptografia simétrica**

Vamos fazer uma pausa e revisar os algoritmos de criptografia simétrica que você aprendeu até agora:

* **Criptografia não autenticada** — AES com um modo de operação, mas sem um MAC. Isso é inseguro na prática, pois os ciphertexts podem ser adulterados.
* **Criptografia autenticada** — AES-GCM e ChaCha20-Poly1305 são os dois cifradores mais amplamente adotados.

O capítulo poderia terminar aqui e tudo bem. No entanto, a criptografia do mundo real nem sempre trata apenas dos padrões estabelecidos; também trata de restrições de tamanho, velocidade, formato, e assim por diante. Para isso, vou dar uma breve visão geral de outros tipos de criptografia simétrica que podem ser úteis quando AES-GCM e ChaCha20-Poly1305 não se encaixam.

**4.6.1 Empacotamento de chaves (*Key wrapping*)**

Um dos problemas dos AEADs baseados em *nonce* é que todos eles exigem um *nonce*, o que ocupa espaço adicional. Note que ao criptografar uma chave, pode não ser necessário randomização, pois o que está sendo criptografado já é aleatório e dificilmente irá se repetir (ou, se se repetir, não será um grande problema). Um padrão bem conhecido para *key wrapping* é a **Publicação Especial 800-38F do NIST: "Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping"**. Esses algoritmos de *key wrapping* não exigem um *nonce* ou IV adicionais e randomizam sua criptografia com base no próprio conteúdo a ser criptografado. Graças a isso, não é necessário armazenar um *nonce* ou IV adicional junto aos ciphertexts.

**4.6.2 Criptografia autenticada resistente a reutilização de nonce**

Em 2006, Phillip Rogaway publicou um novo algoritmo de *key wrapping* chamado **vetor de inicialização sintético (SIV — Synthetic Initialization Vector)**. Como parte da proposta, Rogaway observa que o SIV não é útil apenas para criptografar chaves, mas também como um esquema AEAD geral que é mais tolerante a repetições de *nonce*. Como você aprendeu neste capítulo, a repetição de um *nonce* em AES-GCM ou ChaCha20-Poly1305 pode ter consequências catastróficas. Não apenas revela o XOR dos dois plaintexts, mas também permite que um atacante recupere uma chave de autenticação e forje encriptações válidas de mensagens.

O objetivo de um algoritmo resistente a reutilização de *nonce* é que criptografar dois plaintexts com o mesmo *nonce* apenas revela se os dois plaintexts são iguais ou não, e só isso. Não é o ideal, mas obviamente não é tão ruim quanto vazar uma chave de autenticação. O esquema tem atraído bastante interesse e desde então foi padronizado no **RFC 8452: "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption"**. O truque por trás do SIV é que o *nonce* usado no AEAD para criptografar é gerado a partir do próprio plaintext, o que torna altamente improvável que dois plaintexts diferentes acabem sendo criptografados com o mesmo *nonce*.

**4.6.3 Criptografia de disco**

Criptografar o armazenamento de um laptop ou telefone móvel impõe algumas restrições pesadas: precisa ser rápido (senão o usuário perceberá) e só pode ser feito no próprio lugar (*in-place*) (economizar espaço é importante para um grande número de dispositivos). Como a criptografia não pode expandir o tamanho, AEADs que precisam de *nonce* e uma tag de autenticação não são uma boa opção. Em vez disso, usa-se criptografia não autenticada.

Para proteger contra ataques de inversão de bits (*bitflip*), blocos grandes (pense em milhares de bytes) de dados são criptografados de forma que uma única inversão de bit corrompa a descriptografia de todo o bloco. Dessa forma, um ataque tem mais chances de travar o dispositivo do que alcançar seu objetivo. Essas construções são chamadas de **cifradores de bloco largo (wide-block ciphers)**, embora essa abordagem também tenha sido apelidada de **autenticação do homem pobre**.

Sistemas Linux e alguns dispositivos Android adotaram essa abordagem usando o **Adiantum**, uma construção de bloco largo envolvendo o cifrador ChaCha, padronizada pelo Google em 2019. Ainda assim, a maioria dos dispositivos utiliza soluções não ideais: tanto a Microsoft quanto a Apple fazem uso do **AES-XTS**, que não é autenticado e não é um cifrador de bloco largo.

**4.6.4 Criptografia de banco de dados**

Criptografar dados em um banco de dados é complicado. Como o objetivo é evitar que violações do banco vazem dados, a chave usada para criptografar e descriptografar os dados deve ser armazenada separadamente do servidor de banco de dados. Como os clientes não possuem os dados, eles ficam severamente limitados na forma como podem consultar as informações.

A solução mais simples é chamada de **transparent data encryption (TDE)** e simplesmente criptografa colunas selecionadas. Isso funciona bem em alguns cenários, embora seja necessário tomar cuidado para autenticar dados associados que identifiquem a linha e a coluna sendo criptografadas; caso contrário, conteúdos criptografados podem ser trocados. Ainda assim, não é possível pesquisar por dados criptografados e, portanto, as consultas precisam usar as colunas não criptografadas.

A **criptografia pesquisável (searchable encryption)** é o campo de pesquisa que busca resolver este problema. Muitos esquemas diferentes foram propostos, mas parece não haver uma solução perfeita. Diferentes esquemas propõem diferentes níveis de "pesquisabilidade" assim como diferentes degradações de segurança. O **blind indexing**, por exemplo, permite apenas buscas por correspondência exata, enquanto as criptografias **order-preserving** e **order-revealing** permitem ordenar resultados. A conclusão é que a segurança dessas soluções deve ser analisada cuidadosamente, pois de fato representam compensações.

**Resumo**

* **Criptografia (ou criptografia simétrica)** é um primitivo criptográfico que pode ser usado para proteger a confidencialidade de dados. A segurança depende de uma chave simétrica que precisa permanecer secreta.
* A criptografia simétrica precisa ser autenticada (passando a ser chamada de criptografia autenticada) para ser segura, caso contrário, os ciphertexts podem ser adulterados.
* A criptografia autenticada pode ser construída a partir de um algoritmo de criptografia simétrica utilizando um código de autenticação de mensagens (MAC). Mas a melhor prática é usar um algoritmo **AEAD (Authenticated Encryption with Associated Data)**, pois são construções tudo-em-um mais difíceis de serem usadas incorretamente.
* Duas partes podem usar criptografia autenticada para esconder suas comunicações, desde que ambas conheçam a mesma chave simétrica.
* **AES-GCM** e **ChaCha20-Poly1305** são os dois AEADs mais amplamente adotados. A maioria das aplicações hoje usa um dos dois.
* Reutilizar *nonces* quebra a autenticação do AES-GCM e do ChaCha20-Poly1305. Esquemas como o **AES-GCM-SIV** são resistentes a reutilização de *nonce*, enquanto a criptografia de chaves pode evitar esse problema, pois não exige *nonces*.
* A criptografia do mundo real envolve restrições, e os AEADs nem sempre servem para todos os cenários. Este é o caso da criptografia de banco de dados ou disco, por exemplo, que exigem o desenvolvimento de novas construções.