**2 Funções de hash**

Atribuir identificadores globais únicos a qualquer coisa — essa é a promessa da primeira construção criptográfica que você aprenderá neste capítulo: a função de hash.

As funções de hash estão por toda parte na criptografia — em toda parte! Informalmente, elas recebem como entrada quaisquer dados que você quiser e produzem uma string única de bytes em retorno. Dada a mesma entrada, a função de hash sempre reproduz a mesma string de bytes. Isso pode parecer simples, mas essa construção é extremamente útil para construir muitas outras na criptografia. Neste capítulo, você aprenderá tudo sobre funções de hash e por que elas são tão versáteis.

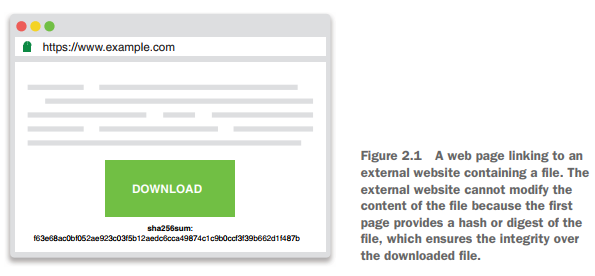
**2.1 O que é uma função de hash?**

Diante de você, um botão de download ocupa boa parte da página. Você pode ler as letras DOWNLOAD, e clicar nele parece redirecioná-lo a um site diferente contendo um arquivo. Abaixo disso, está uma longa string de letras ininteligíveis:

f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b

Este capítulo cobre:

* Funções de hash e suas propriedades de segurança
* As funções de hash amplamente adotadas atualmente
* Outros tipos de hashing que existem



Isso é seguido por algo que parece uma sigla: sha256sum. Soa familiar? Provavelmente você já baixou algo no passado que também estava acompanhado por uma string estranha assim.

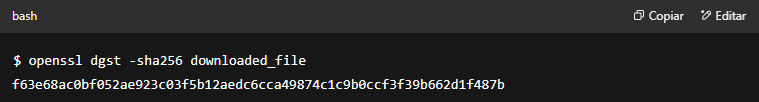
Se você já se perguntou o que fazer com aquela longa string:

1. Clique no botão para baixar o arquivo
2. Use o algoritmo SHA-256 para fazer o hash do arquivo baixado
3. Compare a saída (o resumo) com a string longa exibida na página da web

Isso permite verificar se você baixou o arquivo correto.

NOTA: A saída de uma função de hash é frequentemente chamada de “digest” ou “hash”. Eu uso as duas palavras de forma intercambiável ao longo deste livro. Outros podem chamá-la de “checksum” ou “soma”, termos que evito, pois são usados principalmente por funções de hash não criptográficas e podem gerar confusão. Apenas tenha isso em mente quando bases de código ou documentos diferentes usarem termos diferentes.

Para tentar fazer hash de algo, você pode usar a popular biblioteca OpenSSL. Ela oferece uma interface de linha de comando (CLI) multifuncional que já vem por padrão em diversos sistemas, incluindo o macOS. Por exemplo, isso pode ser feito abrindo o terminal e digitando a seguinte linha:

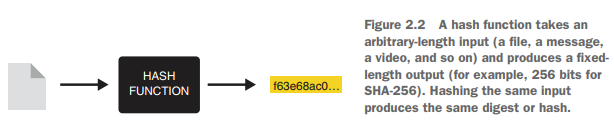


Com esse comando, usamos a função de hash SHA-256 para transformar a entrada (o arquivo baixado) em um identificador único (o valor exibido pelo comando). O que esses passos extras fornecem? Eles fornecem **integridade** e **autenticidade**. Eles dizem que o que você baixou é de fato o arquivo que você deveria ter baixado.

Tudo isso funciona graças a uma propriedade de segurança da função de hash chamada **resistência à segunda pré-imagem**. Esse termo inspirado na matemática significa que, a partir da longa saída do hash f63e..., você não pode encontrar outro arquivo que produza a mesma saída f63e.... Na prática, isso significa que esse resumo está intimamente ligado ao arquivo que você está baixando e que nenhum invasor deveria ser capaz de enganá-lo fornecendo um arquivo diferente.

Observe que esse longo resumo é controlado pelo(s) dono(s) da página da web, e poderia facilmente ser substituído por qualquer um que pudesse modificar a página. (Se você não está convencido, pare um momento para pensar nisso.) Isso significa que precisamos confiar na página que nos deu o resumo, em seus donos e no mecanismo usado para recuperar a página (embora não precisemos confiar na página que nos forneceu o arquivo baixado). Nesse sentido, a função de hash **sozinha** não fornece integridade. A integridade e a autenticidade do arquivo baixado vêm do **resumo combinado com o mecanismo confiável que nos deu o resumo** (HTTPS neste caso). Falaremos sobre HTTPS no capítulo 9, mas por enquanto, imagine que ele permite comunicar-se com segurança com um site.

Voltando à nossa função de hash, que pode ser visualizada como a caixa preta na figura a seguir:



A entrada dessa função pode ter qualquer tamanho. Pode até estar vazia. A saída é sempre do mesmo comprimento e **determinística**: ela sempre produz o mesmo resultado se for dada a mesma entrada. Em nosso exemplo, SHA-256 sempre fornece uma saída de 256 bits (32 bytes), que é sempre codificada como 64 caracteres alfanuméricos em hexadecimal. Uma propriedade importante de uma função de hash é que **não se pode reverter o algoritmo**, ou seja, **não se deve ser capaz de encontrar a entrada apenas com a saída**. Dizemos que funções de hash são **unidirecionais**.

**A notação hexadecimal**

A propósito, a longa string de saída f63e... representa dados binários exibidos em hexadecimal (uma codificação em base 16, usando números de 0 a 9 e letras de a a f para representar diversos bits de dados). Poderíamos ter exibido os dados binários com 0s e 1s (base 2), mas isso ocuparia mais espaço. Em vez disso, a codificação hexadecimal nos permite escrever dois caracteres alfanuméricos para cada 8 bits (1 byte) encontrados. Ela é razoavelmente legível por humanos e ocupa menos espaço. Existem outras formas de codificar dados binários para consumo humano, mas as duas codificações mais amplamente usadas são o hexadecimal e o base64. Quanto maior a base, menos espaço ela ocupa para exibir uma string binária, mas em algum ponto, ficamos sem caracteres legíveis por humanos.

Para ilustrar como uma função de hash funciona na prática, faremos o hash de diferentes entradas com a função de hash SHA-256 usando a mesma CLI do OpenSSL. A sessão de terminal a seguir mostra isso:

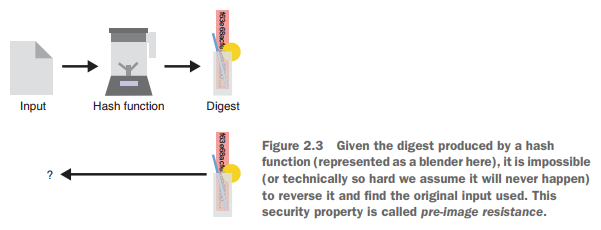


Na próxima seção, veremos quais são exatamente as propriedades de segurança das funções de hash.

**2.2 Propriedades de segurança de uma função de hash**

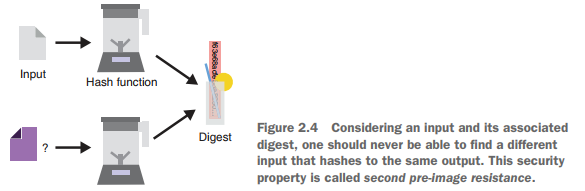
Funções de hash na criptografia aplicada são construções comumente definidas para fornecer **três propriedades específicas de segurança**. Essa definição mudou com o tempo, como veremos nas próximas seções. Mas por agora, vamos definir as três bases fortes que compõem uma função de hash. Isso é importante, pois você precisa entender onde funções de hash podem ser úteis e onde elas **não funcionarão**.

A primeira é a **resistência à pré-imagem** (*pre-image resistance*). Essa propriedade garante que ninguém deve ser capaz de **reverter a função de hash** para recuperar a entrada a partir de uma saída. Na <IMAGEM>, ilustramos essa “unidirecionalidade” imaginando que nossa função de hash é como um liquidificador, tornando impossível recuperar os ingredientes do smoothie produzido.



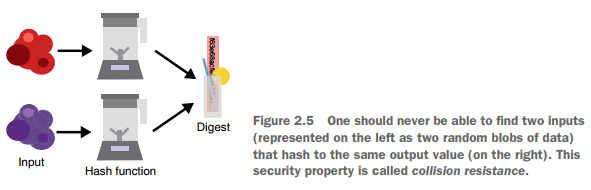
**AVISO** Isso é verdade se sua entrada for pequena? Suponha que ela seja “oui” ou “non” — então é fácil para alguém fazer o hash de todas as palavras de três letras possíveis e descobrir qual foi a entrada. E se o espaço de entrada for pequeno? Ou seja, você sempre faz o hash de variações da frase “Estarei em casa na segunda-feira às 3h da manhã”, por exemplo. Aqui, quem consegue prever isso, mas não sabe exatamente o dia da semana ou a hora, ainda pode fazer o hash de todas as frases possíveis até produzir a saída correta. Assim, essa primeira propriedade de segurança da pré-imagem tem uma ressalva óbvia: **você não pode esconder algo que é pequeno demais ou que é previsível**.

A segunda propriedade é a **resistência à segunda pré-imagem** (*second pre-image resistance*). Já vimos essa propriedade de segurança quando queríamos proteger a integridade de um arquivo. A propriedade diz o seguinte: **se eu te der uma entrada e o resumo a que ela faz hash, você não deve ser capaz de encontrar uma entrada diferente que faça hash para o mesmo resumo**. A <IMAGEM> ilustra esse princípio.



Observe que **não controlamos a primeira entrada**. Essa ênfase é importante para entender a próxima propriedade de segurança das funções de hash.

Por fim, a terceira propriedade é a **resistência a colisões** (*collision resistance*). Ela garante que **ninguém deve ser capaz de produzir duas entradas diferentes que gerem a mesma saída** (como visto na <IMAGEM>). Aqui o atacante **pode escolher as duas entradas**, ao contrário da propriedade anterior, que fixa uma das entradas.



As pessoas frequentemente confundem resistência a colisões com resistência à segunda pré-imagem. Leve um momento para entender as diferenças.

**O oráculo aleatório**

Além disso, funções de hash geralmente são projetadas de forma que seus resumos sejam imprevisíveis e aleatórios. Isso é útil porque **nem sempre se pode provar um protocolo como seguro** com base em alguma propriedade de segurança de função de hash (como resistência a colisões, por exemplo). Muitos protocolos, em vez disso, são provados seguros no **modelo de oráculo aleatório**, onde um participante fictício e ideal chamado oráculo aleatório é usado. Nesse tipo de protocolo, pode-se enviar qualquer entrada como solicitação a esse oráculo aleatório, que retorna saídas completamente aleatórias em resposta e, como uma função de hash, ao receber a mesma entrada duas vezes, retorna a mesma saída.

As provas nesse modelo são às vezes controversas, já que **não sabemos ao certo se podemos substituir esses oráculos aleatórios por funções de hash reais (na prática)**. Ainda assim, muitos protocolos legítimos são provados seguros usando esse método, no qual as funções de hash são vistas como mais ideais do que provavelmente são.

**2.3 Considerações de segurança para funções de hash**

Até agora, vimos três propriedades de segurança de uma função de hash:

* Resistência à pré-imagem
* Resistência à segunda pré-imagem
* Resistência a colisões

Essas propriedades de segurança são frequentemente **sem sentido por si só**; tudo depende de como você **utiliza** a função de hash. No entanto, é importante compreendermos algumas **limitações** aqui antes de analisarmos algumas das funções de hash do mundo real.

Primeiro, essas propriedades de segurança presumem que você **usa a função de hash de forma razoável**. Imagine que eu faça o hash da palavra *yes* ou da palavra *no*, e então publique o resumo. Se você tiver alguma ideia do que estou fazendo, pode simplesmente fazer o hash das duas palavras e comparar o resultado com o que eu forneci. Como **não há segredos envolvidos** e como o algoritmo de hashing que usamos é público, você está livre para fazer isso. E, de fato, alguém poderia pensar que isso quebra a **resistência à pré-imagem** da função de hash, mas eu argumentaria que **sua entrada não era aleatória o suficiente**. Além disso, como uma função de hash aceita entradas de comprimento arbitrário e sempre produz uma saída de mesmo tamanho, também existem um número infinito de entradas que fazem hash para a mesma saída. Mais uma vez, você poderia dizer: “Bem, isso não quebra a **resistência à segunda pré-imagem**?” A resistência à segunda pré-imagem está apenas dizendo que é extremamente difícil encontrar **outra** entrada — tão difícil que **assumimos que, na prática, é impossível**, mas **não teoricamente impossível**.

Segundo, **o tamanho do resumo importa**. Isso não é uma peculiaridade apenas das funções de hash. **Todos os algoritmos criptográficos devem se preocupar com o tamanho de seus parâmetros na prática.** Vamos imaginar o seguinte exemplo extremo. Temos uma função de hash que produz saídas de comprimento **2 bits** de forma uniformemente aleatória (ou seja, ela gera 00 25% das vezes, 01 25% das vezes, e assim por diante). Você **não precisará de muito trabalho para produzir uma colisão**: após fazer o hash de algumas strings de entrada aleatórias, você deverá conseguir encontrar duas que fazem hash para a mesma saída.

Por essa razão, existe um tamanho mínimo de saída que uma função de hash deve produzir na prática: **256 bits (ou 32 bytes)**. Com uma saída desse tamanho, **colisões devem estar fora de alcance**, a menos que ocorra um avanço significativo em computação.

Como esse número foi obtido? Na criptografia do mundo real, os algoritmos visam **pelo menos 128 bits de segurança**. Isso significa que um atacante que queira quebrar um algoritmo (com segurança de 128 bits) precisa realizar cerca de 2^128 operações (por exemplo, tentar todas as strings de entrada possíveis de 128 bits de comprimento levaria 2^128 operações).

Para que uma função de hash forneça todas as três propriedades de segurança mencionadas anteriormente, ela precisa fornecer **pelo menos 128 bits de segurança contra os três ataques**. O ataque mais fácil geralmente é **encontrar colisões**, devido ao **limite do aniversário** (*birthday bound*).

Se nossa função de hash gera saídas aleatórias de 256 bits, o espaço de todas as saídas é de tamanho 2^256. Isso significa que **colisões podem ser encontradas com boa probabilidade após gerar 2^128 resumos** (devido ao limite do aniversário). Esse é o número que estamos almejando, e é por isso que funções de hash devem fornecer **no mínimo saídas de 256 bits**.

Certas restrições às vezes levam desenvolvedores a reduzir o tamanho de um resumo **truncando-o** (removendo alguns de seus bytes). Em teoria, isso é possível, mas pode reduzir muito a segurança. Para alcançar 128 bits de segurança no mínimo, **um resumo não deve ser truncado abaixo de:**

* 256 bits para **resistência a colisões**
* 128 bits para **resistência à pré-imagem** e **à segunda pré-imagem**

Isso significa que, dependendo da propriedade da qual se depende, a saída de uma função de hash pode ser truncada para obter um resumo mais curto.

**2.4 Funções de hash na prática**

Como dissemos anteriormente, na prática, as funções de hash raramente são usadas isoladamente. Elas são mais frequentemente combinadas com outros elementos para criar um **primitivo criptográfico** ou um **protocolo criptográfico**. Veremos muitos exemplos de uso de funções de hash para construir objetos mais complexos neste livro, mas esta seção descreve algumas das formas como as funções de hash têm sido usadas no mundo real.

**O limite do aniversário**

O **limite do aniversário** (*birthday bound*) tem origem na teoria das probabilidades, na qual o **problema do aniversário** revela alguns resultados contraintuitivos. Quantas pessoas são necessárias em uma sala para que haja **pelo menos 50% de chance** de que duas delas compartilhem o mesmo aniversário (isto é, uma colisão)? Acontece que **23 pessoas escolhidas aleatoriamente já são suficientes** para alcançar essa probabilidade! Estranho, não é?

Isso é chamado de **paradoxo do aniversário**. Na prática, quando geramos aleatoriamente strings de um espaço com 2^N possibilidades, pode-se esperar, com 50% de chance, encontrar uma colisão após gerar aproximadamente 2^(N/2) strings.

**2.4.1 Comprometimentos (*Commitments*)**

Imagine que você sabe que uma ação no mercado vai subir de valor e atingir US$ 50 no próximo mês, mas você realmente **não pode contar isso aos seus amigos** (por alguma razão legal, talvez). Você ainda quer ser capaz de dizer a eles que **já sabia disso depois que acontecer**, porque você é convencido (não negue!). O que você pode fazer é **se comprometer com uma frase** como: “Ação X atingirá US$ 50 no próximo mês.” Para isso, **faça o hash da frase** e forneça aos seus amigos apenas a saída. Um mês depois, revele a frase. Seus amigos poderão fazer o hash da frase e observar que, de fato, ela produz a mesma saída.

Isso é o que chamamos de **esquema de comprometimento** (*commitment scheme*). Comprometimentos, na criptografia, geralmente tentam atingir duas propriedades:

* **Sigilo** (*Hiding*) — Um comprometimento deve esconder o valor subjacente.
* **Vinculação** (*Binding*) — Um comprometimento deve esconder **um único valor**. Em outras palavras, se você se compromete com um valor x, **não deve ser capaz de mais tarde revelar com sucesso um valor diferente y**.

**2.4.2 Integridade de sub-recurso (*Subresource Integrity*)**

Acontece (com frequência) de páginas da web importarem arquivos JavaScript externos. Por exemplo, muitos sites usam **CDNs (Content Delivery Networks)** para importar bibliotecas JavaScript ou arquivos relacionados a frameworks web em suas páginas. Esses CDNs são colocados em locais estratégicos para entregar rapidamente esses arquivos aos visitantes. No entanto, se o CDN agir de má-fé e decidir fornecer arquivos JavaScript maliciosos, isso pode ser um problema sério. Para contornar isso, as páginas da web podem usar um recurso chamado **subresource integrity**, que permite incluir um **resumo (digest)** na tag de importação:



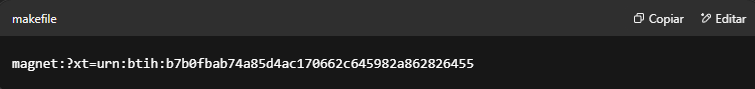
Esse é exatamente o mesmo cenário que discutimos na introdução deste capítulo. Assim que o arquivo JavaScript é recuperado, o **navegador faz o hash dele (usando SHA-256)** e verifica se ele corresponde ao resumo que foi incluído no código da página. Se estiver correto, o arquivo JavaScript é executado, pois sua integridade foi verificada.

**2.4.3 BitTorrent**

Usuários (chamados de *peers*) ao redor do mundo utilizam o **protocolo BitTorrent** para compartilhar arquivos diretamente entre si (também conhecido como **peer-to-peer**). Para distribuir um arquivo, ele é **dividido em pedaços** (*chunks*) e **cada pedaço é hasheado individualmente**. Esses hashes são então compartilhados como uma fonte de confiança para representar o arquivo a ser baixado.

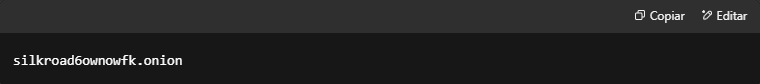
**Exercício**  
Você consegue dizer se uma função de hash fornece *sigilo* e *vinculação* se usada como um esquema de comprometimento?

O BitTorrent possui vários mecanismos que permitem a um peer obter diferentes pedaços de um arquivo a partir de diferentes peers. No final, a **integridade do arquivo completo é verificada** ao se fazer o hash de cada um dos pedaços baixados e comparando as saídas com seus respectivos resumos conhecidos (antes de remontar o arquivo a partir dos pedaços). Por exemplo, o seguinte **link magnético** representa o sistema operacional Ubuntu, versão 19.04. Ele é um **resumo (em hexadecimal)** obtido a partir do hash dos metadados do arquivo, bem como dos resumos de todos os pedaços:



**2.4.4 Tor**

O objetivo do **navegador Tor** é dar às pessoas a capacidade de **navegar na internet anonimamente**. Outra funcionalidade é a de permitir que alguém crie **páginas da web ocultas**, cujas localizações físicas são difíceis de rastrear. As conexões com essas páginas são protegidas por meio de um protocolo que **usa a chave pública da página web**. (Veremos mais sobre como isso funciona no capítulo 9, quando falarmos sobre criptografia de sessão.) Por exemplo, o *Silk Road*, que costumava ser o “eBay das drogas” até ser apreendido pelo FBI, era acessado por meio de:



no navegador Tor. Essa **string base32** na verdade representa o **hash da chave pública** do Silk Road. Assim, **ao conhecer o endereço .onion**, você pode **autenticar a chave pública** da página web oculta que está visitando e ter certeza de que está se comunicando com a página correta (e não com um falsificador). Se isso não estiver claro agora, não se preocupe — voltarei a mencionar esse assunto no capítulo 9.

Em todos os exemplos desta seção, uma **função de hash forneceu integridade ou autenticidade de conteúdo** em situações onde:

* Alguém poderia adulterar o conteúdo sendo hasheado;
* O hash é comunicado de forma segura a você.

Às vezes também dizemos que **autenticamos algo ou alguém**. É importante entender que, **se o hash não for obtido com segurança, qualquer um pode substituí-lo pelo hash de outra coisa!** Assim, ele **não fornece integridade por si só**. O próximo capítulo, sobre **códigos de autenticação de mensagens**, corrigirá isso ao introduzir **segredos**. Agora vamos ver **quais algoritmos de função de hash realmente se pode usar**.

**Exercício**  
A propósito, não há como essa string representar 256 bits (32 bytes), certo? Então como isso é seguro, de acordo com o que você aprendeu na seção 2.3?  
Além disso, consegue adivinhar **como o Dread Pirate Roberts** (o pseudônimo do webmaster do Silk Road) conseguiu obter um hash que continha o nome do site?

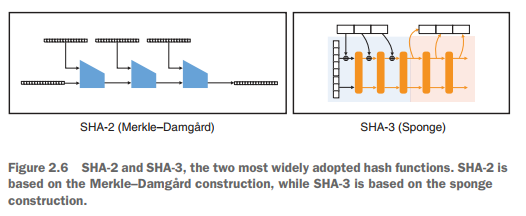
**2.5 Funções de hash padronizadas**

Mencionamos o SHA-256 em nosso exemplo anterior, que é apenas uma das funções de hash que podemos usar. Antes de listarmos as **funções de hash recomendadas atualmente**, vamos primeiro mencionar **outros algoritmos usados em aplicações do mundo real que não são considerados funções de hash criptográficas**.

Primeiro, funções como **CRC32** não são funções de hash criptográficas, mas sim **funções de código de detecção de erros**. Embora ajudem a detectar alguns erros simples, **elas não fornecem nenhuma das propriedades de segurança mencionadas anteriormente** e não devem ser confundidas com as funções de hash de que estamos falando (embora às vezes compartilhem o nome). A saída delas é geralmente chamada de **checksum**.

Segundo, funções de hash populares como **MD5** e **SHA-1** são hoje consideradas **quebradas**. Embora ambas tenham sido padronizadas e amplamente aceitas na década de 1990, **MD5 e SHA-1 foram demonstradas como vulneráveis** em 2004 e 2016, respectivamente, quando colisões foram publicadas por diferentes grupos de pesquisa. Esses ataques foram bem-sucedidos **em parte por avanços na computação, mas principalmente por falhas no design dessas funções**.

As próximas duas seções apresentam o **SHA-2 e o SHA-3**, as duas funções de hash mais amplamente utilizadas. A <IMAGEM> introduz essas funções.



**A descontinuação é difícil**

Tanto o **MD5 quanto o SHA-1** eram considerados boas funções de hash **até que pesquisadores demonstraram sua falta de resistência a colisões**. Ainda assim, até hoje, a resistência à pré-imagem e à segunda pré-imagem dessas funções **não foi afetada por nenhum ataque conhecido**. Isso **não importa para nós**, pois neste livro queremos **falar apenas sobre algoritmos seguros**. No entanto, **você ainda verá pessoas usando MD5 e SHA-1** em sistemas que dependem apenas da resistência à pré-imagem desses algoritmos, e não da resistência a colisões. Esses infratores frequentemente argumentam que **não podem atualizar as funções de hash para versões mais seguras devido a questões de legado e compatibilidade com versões antigas**.

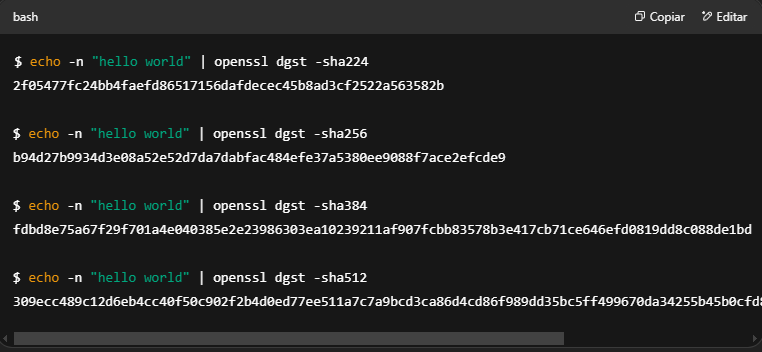
Como este livro é destinado a durar no tempo e ser **um farol de luz para o futuro da criptografia do mundo real**, esta será **a última vez que mencionarei essas funções de hash**.

**2.5.1 A função de hash SHA-2**

Agora que vimos o que são funções de hash e tivemos um vislumbre de seus casos de uso, resta saber **quais funções de hash podemos usar na prática**. Nas próximas duas seções, apresento **duas funções de hash amplamente aceitas**, e também forneço **explicações de alto nível sobre como elas funcionam por dentro**. Essas explicações de alto nível **não devem alterar a forma como usamos funções de hash**, pois as descrições como caixa preta que forneci anteriormente já devem ser suficientes. Mas, ainda assim, é interessante ver **como essas primitivas criptográficas foram concebidas por criptógrafos**.

A função de hash mais amplamente adotada é o **SHA-2 (Secure Hash Algorithm 2)**. O SHA-2 foi inventado pela **NSA** e padronizado pela **NIST** em 2001. Foi criado para **suceder o SHA-1**, já envelhecido, também padronizado pela NIST. O SHA-2 fornece **4 versões diferentes**, produzindo saídas de **224, 256, 384 ou 512 bits**. Seus respectivos nomes omitem o nome completo do algoritmo: **SHA-224**, **SHA-256**, **SHA-384** e **SHA-512**. Além disso, **duas outras versões**, **SHA-512/224** e **SHA-512/256**, fornecem saídas de 224 e 256 bits, respectivamente, truncando o resultado do SHA-512.

Na seguinte sessão de terminal, chamamos cada variante do SHA-2 com a CLI do OpenSSL. Observe que **chamar as diferentes variantes com a mesma entrada produz saídas dos comprimentos especificados, mas completamente diferentes**:

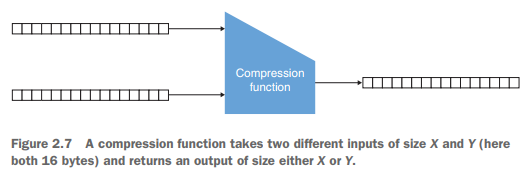


Atualmente, as pessoas usam principalmente o **SHA-256**, que fornece os **mínimos 128 bits de segurança** necessários para as três propriedades de segurança. Aplicações mais paranoicas fazem uso do **SHA-512**. Agora, vejamos uma explicação simplificada de como o SHA-2 funciona.

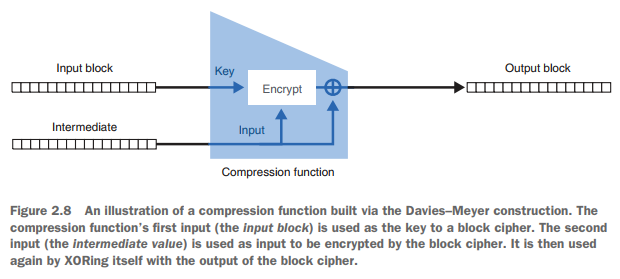
**A operação OU exclusivo**

Para entender o que vem a seguir, você precisa entender a operação **XOR** (*exclusive OR*). **XOR** é uma operação **bit a bit**, ou seja, ela opera sobre bits. A <IMAGEM> a seguir mostra como isso funciona. **XOR é onipresente na criptografia**, então certifique-se de se lembrar disso:

Tudo começa com uma função especial chamada **função de compressão** (*compression function*). Uma função de compressão **recebe duas entradas de determinado tamanho e produz uma saída do tamanho de uma das entradas**. Simplificando: ela **recebe alguns dados e retorna menos dados**. A <IMAGEM> ilustra isso.



Embora existam diferentes formas de construir uma função de compressão, o **SHA-2 usa o método de Davies–Meyer** (veja <IMAGEM>), que se baseia em um **cifra de bloco** (*block cipher*) — uma cifra que pode criptografar um bloco de dados de tamanho fixo. Mencionei a cifra de bloco **AES** no capítulo 1, mas você ainda não aprendeu sobre ela. Por enquanto, aceite a função de compressão como uma **caixa preta** até você ler o capítulo 4 sobre **criptografia autenticada**.



O **SHA-2 é uma construção de Merkle–Damgård**, que é um algoritmo (inventado independentemente por Ralph Merkle e Ivan Damgård) que faz o hash de uma mensagem **chamando iterativamente essa função de compressão**. Especificamente, ele funciona seguindo os dois passos a seguir.

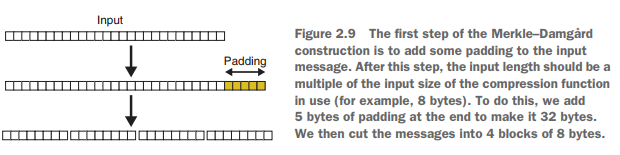
**Exclusivo-OU** ou **XOR** (geralmente denotado como ⊕) opera sobre 2 bits. É semelhante à operação OR, exceto no caso em que **ambos os operandos são 1**, o que retorna 0.

**Função de compressão**: recebe dois blocos de entrada de tamanho X e Y (aqui ambos com 16 bytes) e retorna uma saída de tamanho X ou Y.

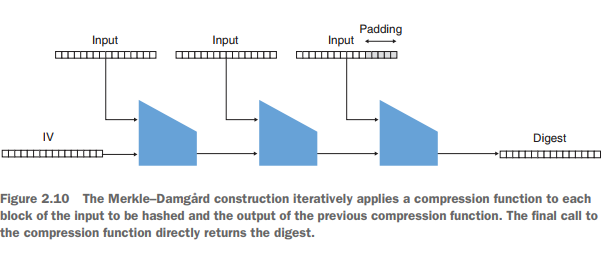
**Ilustração da função de compressão construída via a construção Davies–Meyer**:

* O primeiro argumento da função (bloco de entrada) é usado como **chave** para uma cifra de bloco.
* O segundo argumento (valor intermediário) é **criptografado pela cifra de bloco**.
* O resultado é **XORado** com a entrada original para produzir a saída da função de compressão.

Primeiro, aplica-se um **preenchimento (padding)** à entrada que queremos hashear, depois ela é **cortada em blocos que cabem na função de compressão**. Preenchimento significa **acrescentar bytes específicos ao final da entrada para que seu comprimento seja múltiplo de algum tamanho de bloco**. Cortar a entrada preenchida em blocos de tamanho igual nos permite ajustá-los no primeiro argumento da função de compressão. Por exemplo, o **SHA-256 tem um tamanho de bloco de 512 bits**. A <IMAGEM> ilustra esse passo.

**Passo 1 da construção Merkle–Damgård**: adicionar preenchimento à mensagem de entrada. Após esse passo, o comprimento da entrada deve ser múltiplo do tamanho de entrada da função de compressão (por exemplo, 8 bytes). Para isso, adicionamos 5 bytes de preenchimento ao final, tornando-a com 32 bytes. Em seguida, cortamos a mensagem em 4 blocos de 8 bytes.

Depois, aplica-se **iterativamente a função de compressão aos blocos da mensagem**, usando a **saída anterior da função de compressão como segundo argumento** para a próxima chamada. A saída final é o **resumo (digest)**. A <IMAGEM> ilustra esse passo.

****

**Passo 2 da construção Merkle–Damgård**: aplicar iterativamente uma função de compressão a cada bloco da entrada (previamente preenchida) e a saída da função de compressão anterior. A chamada final da função de compressão retorna diretamente o resumo.

E é assim que o **SHA-2 funciona**: **chamando iterativamente sua função de compressão** sobre fragmentos da entrada até que tudo seja processado em um resumo final.

**NOTA**  
A construção Merkle–Damgård é **comprovadamente resistente a colisões** se a própria função de compressão for resistente a colisões. Assim, a segurança da função de hash (com entrada de comprimento arbitrário) é **reduzida à segurança da função de compressão de tamanho fixo**, que é mais fácil de projetar e analisar. Aí está a engenhosidade da construção Merkle–Damgård.

No começo, o segundo argumento da função de compressão é geralmente **fixo e padronizado** como um valor “sem carta na manga” (*nothing-up-my-sleeve*). Especificamente, o **SHA-256 usa as raízes quadradas dos primeiros números primos** para derivar esse valor. Um valor “sem carta na manga” serve para convencer a comunidade criptográfica de que **ele não foi escolhido para enfraquecer a função de hash** (por exemplo, para criar uma porta dos fundos). Esse é um conceito popular na criptografia.

**AVISO**  
Embora o SHA-2 seja uma função de hash perfeitamente aceitável para uso, **ele não é adequado para fazer hash de segredos**. Isso ocorre devido a uma limitação da construção Merkle–Damgård, que torna o SHA-2 **vulnerável a um ataque chamado “length-extension”** se for usado para fazer hash de segredos. Falaremos mais sobre isso no próximo capítulo.

**2.5.2 A função de hash SHA-3**

Como mencionei anteriormente, tanto as funções de hash **MD5** quanto **SHA-1** foram quebradas relativamente recentemente. Essas duas funções utilizavam a mesma **construção Merkle–Damgård** que descrevi na seção anterior. Por conta disso — e também do fato de que o **SHA-2 é vulnerável a ataques de extensão de comprimento** (*length-extension attacks*) —, o **NIST decidiu, em 2007, organizar uma competição aberta** para definir um novo padrão: o **SHA-3**. Esta seção apresenta o novo padrão e tenta oferecer uma explicação de alto nível de seu funcionamento interno.

Em 2007, **64 candidatos diferentes** de diversos grupos internacionais de pesquisa participaram do concurso do SHA-3. Cinco anos depois, o **Keccak**, uma das submissões, foi nomeado vencedor e passou a se chamar **SHA-3**. Em 2015, o SHA-3 foi padronizado na publicação **FIPS 202**  
(<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>).

O SHA-3 respeita as três propriedades de segurança que discutimos anteriormente e **fornece tanta segurança quanto as variantes do SHA-2**. Além disso, ele **não é vulnerável a ataques de extensão de comprimento** e pode ser **usado para fazer hash de segredos**. Por essa razão, ele é agora a **função de hash recomendada para uso**. Ele oferece as mesmas variantes do SHA-2, desta vez indicando o nome completo SHA-3 em suas versões: **SHA-3-224**, **SHA-3-256**, **SHA-3-384** e **SHA-3-512**. Assim, de forma semelhante ao SHA-2, o **SHA-3-256** fornece uma saída de **256 bits**, por exemplo.

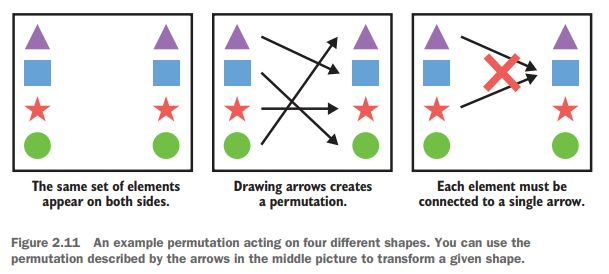
Agora, vou tomar algumas páginas para explicar como o SHA-3 funciona.

O SHA-3 é um algoritmo criptográfico **baseado em uma permutação**. A maneira mais simples de entender uma permutação é imaginar o seguinte: você tem um conjunto de elementos à esquerda e o mesmo conjunto de elementos à direita. Agora desenhe **setas ligando cada elemento da esquerda a um da direita**. Cada elemento pode ter **apenas uma seta** saindo e **terminando em um**. Pronto — você tem uma permutação. A <IMAGEM> ilustra esse princípio.

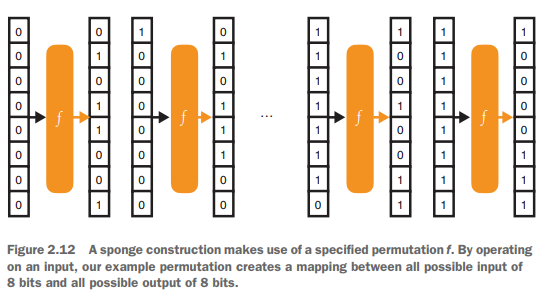
Por definição, **qualquer permutação também é reversível**, o que significa que, a partir da saída, podemos encontrar a entrada.

O **SHA-3 é construído com uma construção de esponja** (*sponge construction*), uma construção diferente da Merkle–Damgård, inventada como parte da competição do SHA-3. Ela é baseada em uma permutação específica chamada **keccak-f**, que recebe uma entrada e retorna uma saída do mesmo tamanho.

**NOTA**  
Não explicaremos como o keccak-f foi projetado, mas você terá uma ideia no capítulo 4, porque ele se assemelha substancialmente ao algoritmo **AES** (com a exceção de que **não possui uma chave**). Isso não é coincidência — **um dos inventores do AES também foi um dos inventores do SHA-3**.

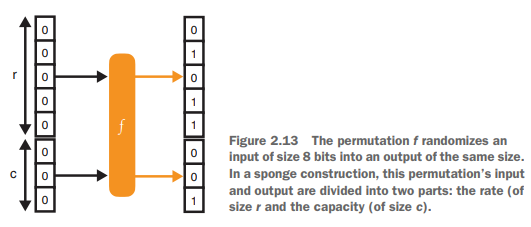


Nos próximos parágrafos, uso uma **permutação de 8 bits** para ilustrar como a construção de esponja funciona. Como a permutação é definida, você pode imaginar que a <IMAGEM> é uma boa ilustração do mapeamento criado por essa permutação sobre **todas as entradas possíveis de 8 bits**. Comparado à explicação anterior sobre permutação, você também pode imaginar que **cada string binária de 8 bits** é representada por formas diferentes (por exemplo, 000... é um triângulo, 100... é um quadrado, e assim por diante).

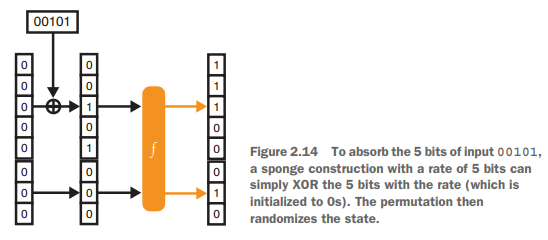


Para usar uma permutação em nossa construção de esponja, também precisamos definir uma **divisão arbitrária da entrada e da saída** em duas partes: a **taxa** (*rate*) e a **capacidade** (*capacity*). É um pouco estranho, mas acompanhe. A <IMAGEM> ilustra esse processo.

Onde colocamos o limite entre a taxa e a capacidade é arbitrário. **Diferentes versões do SHA-3 usam parâmetros diferentes**. Informalmente, apontamos que **a capacidade deve ser tratada como um segredo**, e quanto **maior** ela for, mais **segura** será a construção da esponja.



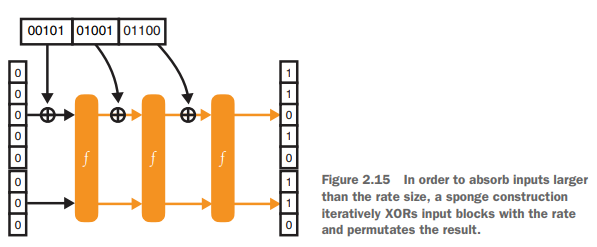
Agora, como toda boa função de hash, precisamos ser capazes de **fazer hash de alguma coisa**, certo? Caso contrário, ela seria inútil. Para isso, nós simplesmente fazemos um **XOR (⊕)** da entrada com a **parte da taxa (rate)** da entrada da permutação. No início, isso é apenas um monte de **0s**. Como apontamos anteriormente, a **capacidade (capacity)** é tratada como um **segredo**, então **não fazemos XOR com ela**. A <IMAGEM> ilustra isso.



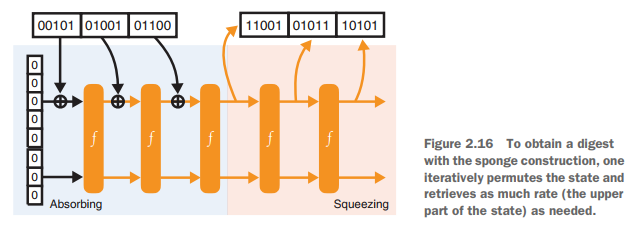
A saída obtida agora deve parecer aleatória (embora possamos trivialmente encontrar qual foi a entrada, já que uma permutação é, por definição, **reversível**). E se quisermos ingerir uma entrada maior? Bem, de forma semelhante ao que fizemos com o SHA-2, devemos:

1. Fazer o **preenchimento (padding)** da entrada, se necessário, e depois dividi-la em blocos de tamanho igual à taxa.
2. Chamar iterativamente a **permutação**, fazendo XOR de cada bloco com a entrada da permutação e permutando o estado (isto é, o valor intermediário) após cada bloco ter sido aplicado com XOR.

Ignorei o preenchimento no restante dessas explicações por uma questão de simplificação, mas **o preenchimento é uma etapa importante do processo** para distinguir entre entradas como 0 e 00, por exemplo. A <IMAGEM> ilustra essas duas etapas.



Até agora tudo bem, mas **ainda não produzimos um resumo (digest)**. Para fazer isso, podemos simplesmente **usar a parte da taxa do último estado da esponja** (mais uma vez, **sem tocar na capacidade**). Para obter um **resumo mais longo**, podemos continuar permutando e **lendo a parte da taxa** do estado, como mostra a <IMAGEM>.



E é assim que o **SHA-3 funciona**. Como ele usa uma construção de esponja, o ato de ingerir a entrada é naturalmente chamado de **absorção** (*absorbing*), e a geração do resumo é chamada de **extração** (*squeezing*). A esponja é especificada com uma permutação de **1600 bits**, utilizando diferentes valores para r (taxa) e c (capacidade), dependendo do nível de segurança anunciado pelas diferentes variantes do SHA-3.

**SHA-3 é um oráculo aleatório**

Falamos antes sobre **oráculos aleatórios**: uma construção ideal e fictícia que **retorna respostas perfeitamente aleatórias** para consultas, repetindo-se se a mesma entrada for usada duas vezes. Acontece que a **construção de esponja se comporta de forma semelhante a um oráculo aleatório**, desde que a **permutação usada** por ela **pareça suficientemente aleatória**.

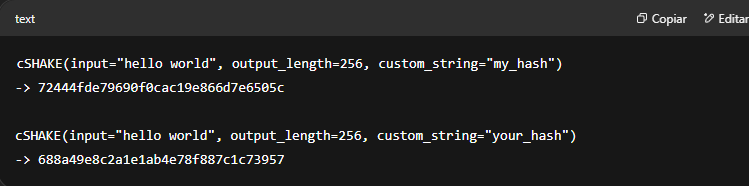
Como provamos tais propriedades de segurança da permutação? **Nossa melhor abordagem é tentar quebrá-la, muitas vezes**, até adquirirmos **forte confiança em seu design** (o que foi feito durante a competição do SHA-3). O fato de que o **SHA-3 pode ser modelado como um oráculo aleatório** imediatamente lhe confere as propriedades de segurança que esperaríamos de uma função de hash.

**2.5.3 SHAKE e cSHAKE: duas funções de saída extensível (XOF)**

Apresentei os dois principais padrões de função de hash: **SHA-2** e **SHA-3**. Essas são **funções de hash bem definidas** que aceitam **entradas de comprimento arbitrário** e produzem **saídas aleatórias e de comprimento fixo**. Como você verá nos próximos capítulos, **protocolos criptográficos frequentemente precisam desse tipo de primitiva**, mas **não querem ser limitados pelo tamanho fixo do resumo** de uma função de hash. Por essa razão, o padrão SHA-3 introduziu uma **primitiva mais versátil** chamada **função de saída extensível** (*extendable output function* ou **XOF**, pronunciado *zoff*). Esta seção apresenta as duas XOFs padronizadas: **SHAKE** e **cSHAKE**.

**SHAKE**, especificada na **FIPS 202** junto com o SHA-3, pode ser vista como uma **função de hash que retorna uma saída de comprimento arbitrário**. A SHAKE é fundamentalmente **a mesma construção que o SHA-3**, exceto que ela é **mais rápida** e **permuta quantas vezes for necessário durante a fase de extração (squeezing)**. Produzir saídas de tamanhos diferentes é bastante útil — **não apenas para criar um resumo**, mas também para **gerar números aleatórios**, **derivar chaves** e assim por diante. Voltarei a falar sobre as aplicações da SHAKE mais adiante neste livro; por enquanto, imagine que a SHAKE é como o SHA-3, exceto que ela fornece uma **saída de qualquer tamanho que você desejar**.

Essa construção é **tão útil** na criptografia que **um ano após a padronização do SHA-3**, o NIST publicou sua **Publicação Especial 800-185**, contendo uma versão personalizável da SHAKE chamada **cSHAKE**. A **cSHAKE** é praticamente **idêntica à SHAKE**, exceto que **também recebe uma string de personalização** (*customization string*). Essa string pode ser vazia ou pode ser **qualquer string que você quiser**. Vejamos um exemplo de uso da **cSHAKE em pseudocódigo**:



Como você pode ver, os dois resumos são diferentes **mesmo que a entrada da cSHAKE seja a mesma**. Isso ocorre porque uma **string de personalização diferente foi usada**. Uma string de personalização permite que você **personalize sua XOF**! Isso é útil em certos protocolos onde, por exemplo, diferentes funções de hash precisam ser usadas para que **uma prova funcione**. Chamamos isso de **separação de domínio** (*domain separation*).

**Regra de ouro na criptografia:**  
Se a **mesma primitiva criptográfica** for usada em **casos de uso diferentes**, **não a use com a mesma chave** (se ela aceitar uma chave) **e/ou aplique separação de domínio**. Você verá mais exemplos de separação de domínio conforme estudarmos protocolos criptográficos nos próximos capítulos.

**AVISO**  
O **NIST tende a especificar algoritmos que recebem parâmetros em bits em vez de bytes**.  
No exemplo acima, foi solicitado um comprimento de 256 bits.  
Imagine se você tivesse solicitado um comprimento de **16 bytes**, mas o programa pensasse que você queria **16 bits** de saída — o que retornaria apenas **2 bytes**.  
Esse tipo de problema às vezes é chamado de **bit attack**.

Como tudo na criptografia, o **tamanho das strings criptográficas**, como **chaves**, **parâmetros** e **saídas**, está **fortemente relacionado à segurança do sistema**. É importante que **não se solicitem saídas muito curtas da SHAKE ou da cSHAKE**. **Você nunca erra ao usar uma saída de 256 bits**, pois ela fornece **128 bits de segurança contra ataques de colisão**. Mas a criptografia do mundo real às vezes opera em **ambientes com restrições**, que podem exigir **valores criptográficos mais curtos**. Isso pode ser feito **se a segurança do sistema for analisada com cuidado**.

Por exemplo, **se a resistência a colisões não for importante** no protocolo que usa o valor, **a resistência à pré-imagem só exige saídas de 128 bits** da SHAKE ou da cSHAKE.

**2.5.4 Evite hash ambíguo com TupleHash**

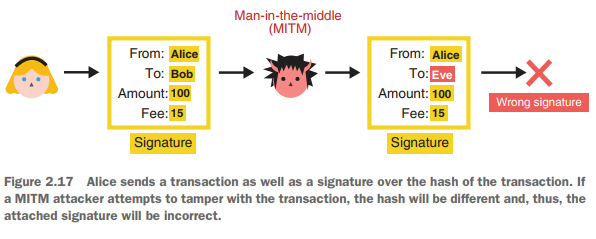
Neste capítulo, falei sobre diferentes tipos de **primitivas criptográficas** e **algoritmos criptográficos**. Isso incluiu:

* A função de hash **SHA-2**, que é vulnerável a ataques de extensão de comprimento, mas ainda amplamente utilizada quando **nenhum segredo é hasheado**;
* A função de hash **SHA-3**, que é a função recomendada atualmente;
* As **XOFs SHAKE e cSHAKE**, que são ferramentas mais versáteis do que funções de hash por oferecerem **comprimento de saída variável**.

Falarei agora sobre mais uma função útil: a **TupleHash**, que é baseada na **cSHAKE** e especificada no mesmo padrão que a cSHAKE. A TupleHash é uma função interessante que permite **fazer o hash de uma tupla (uma lista de elementos)**. Para explicar o que é a TupleHash e por que ela é útil, deixe-me contar uma história.

Alguns anos atrás, fui encarregado de **auditar uma criptomoeda** como parte do meu trabalho. Ela incluía as funcionalidades básicas que se espera de uma criptomoeda: contas, pagamentos etc. As **transações entre usuários** continham metadados sobre **quem está enviando quanto para quem**. Elas também incluíam uma **pequena taxa** para compensar a rede pelo processamento da transação.

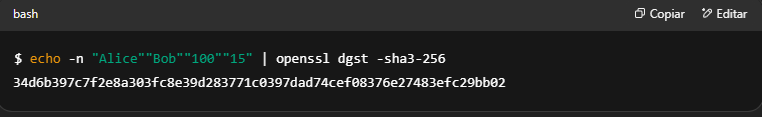
**Alice**, por exemplo, pode enviar transações à rede, mas, para que elas sejam aceitas, ela precisa incluir **prova de que a transação veio dela**. Para isso, ela pode **fazer hash da transação e assiná-la** (dei um exemplo semelhante no capítulo 1). Qualquer pessoa pode fazer o hash da transação e **verificar a assinatura sobre o hash** para ver que aquela era realmente a transação que Alice pretendia enviar. A <IMAGEM> ilustra que um atacante do tipo **man-in-the-middle (MITM)** que intercepte a transação **antes que ela chegue à rede** não poderá adulterá-la.

****

Alice envia uma transação, bem como uma assinatura sobre o hash da transação.  
Se um atacante MITM tentar adulterar a transação, o hash mudará e, portanto, a assinatura anexa será inválida.

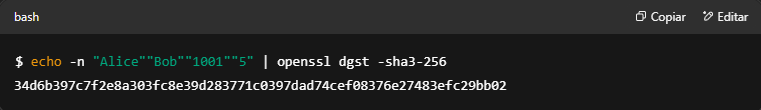
Você verá no capítulo 7 que tal atacante **não consegue forjar a assinatura de Alice** sobre um novo hash. E, graças à **resistência à segunda pré-imagem** da função de hash usada, o atacante também **não pode encontrar uma transação totalmente diferente que gere o mesmo hash**.

Nosso atacante MITM está inofensivo? **Ainda não estamos seguros**. Infelizmente, na criptomoeda que eu estava auditando, a transação era hasheada **simplesmente concatenando cada campo**:



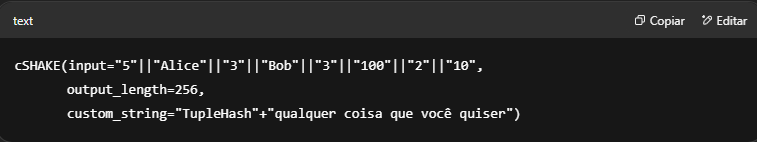
O que parecia totalmente aceitável, na verdade **quebrava completamente o sistema de pagamentos da criptomoeda**. Fazer isso **permite trivialmente a um atacante quebrar a resistência à segunda pré-imagem** da função de hash. Reserve alguns instantes para pensar em como você poderia encontrar **uma transação diferente** que gere o mesmo hash 34d6....

O que acontece se movermos **um dígito do campo taxa para o campo valor**? Veja que a seguinte transação **gera o mesmo hash** que a transação original assinada por Alice:



Assim, um atacante MITM que queira que **Bob receba um pouco mais de dinheiro** conseguiria modificar a transação **sem invalidar a assinatura**. Como você deve ter adivinhado, **é isso que a TupleHash resolve**. Ela permite que você **faça o hash de uma lista de campos de forma não ambígua**, usando codificação clara.

O que acontece na prática é algo parecido com o seguinte (usando || como operação de concatenação de strings):



A entrada é, desta vez, construída **prefixando cada campo da transação com seu comprimento**. Dedique um minuto para entender **por que isso resolve o problema**.

De forma geral, é possível usar **qualquer função de hash com segurança**, certificando-se sempre de **serializar a entrada antes de hasheá-la**. Serializar a entrada significa que sempre existe uma maneira de **desserializá-la** (ou seja, recuperar a entrada original). Se for possível desserializar os dados, então **não há ambiguidade na separação dos campos**.

**2.6 Hashing de senhas**

Você viu várias funções úteis neste capítulo que são **funções de hash ou extensões de funções de hash**. Mas antes que você possa pular para o próximo capítulo, preciso mencionar o **hashing de senhas**.

Imagine o seguinte cenário: você tem um site (o que faz de você um webmaster) e quer que seus usuários possam **se registrar e fazer login** no site. Então você cria duas páginas da web para essas duas funcionalidades. De repente, você se pergunta: **como armazenar as senhas dos usuários?** Você as armazena em texto puro em um banco de dados? A princípio, isso parece não ter nada de errado, você pensa.

Mas isso **não é nada ideal**. As pessoas tendem a **reutilizar a mesma senha em todo lugar** e, se (ou melhor, quando) **seu site for invadido** e os atacantes conseguirem extrair todas as senhas dos usuários, **isso será péssimo para seus usuários e para a reputação da sua plataforma**. Você pensa um pouco mais e percebe que um atacante que conseguir roubar esse banco de dados **poderá fazer login como qualquer usuário**. Armazenar senhas em texto puro já não parece mais uma boa ideia, e você gostaria de encontrar uma forma melhor de lidar com isso.

Uma solução seria **fazer hash das senhas** e **armazenar apenas os resumos**. Quando alguém fizer login no seu site, o fluxo seria semelhante ao seguinte:

1. Você recebe a senha do usuário.
2. Você faz o hash da senha recebida e **descarta a senha original**.
3. Você compara o resumo com o que estava armazenado anteriormente; se corresponder, o usuário está autenticado.

Esse fluxo permite que você **manipule senhas de usuários por um tempo limitado**. Ainda assim, **um atacante que invadir seus servidores poderá permanecer oculto**, coletando senhas desse fluxo **até que você perceba sua presença**. Reconhecemos que essa **ainda não é uma situação perfeita**, mas ela **melhora a segurança do site**.

Na segurança, chamamos isso de **defesa em profundidade** (*defense in depth*), que é o ato de empilhar defesas imperfeitas na esperança de que **um atacante não consiga ultrapassar todas elas**. Isso também é parte do que chamamos de **criptografia do mundo real**. Mas outros problemas ainda existem com essa solução:

* Se um atacante recuperar as senhas hasheadas, **um ataque de força bruta ou uma busca exaustiva** (testando todas as senhas possíveis) pode ser realizado. Isso testaria cada tentativa contra **todo o banco de dados**. Idealmente, **queremos que o atacante só consiga atacar uma senha por vez**.
* Funções de hash **devem ser rápidas**. Os atacantes podem tirar proveito disso para realizar **muitos ataques de força bruta por segundo**. Idealmente, **gostaríamos de ter um mecanismo que tornasse esses ataques mais lentos**.

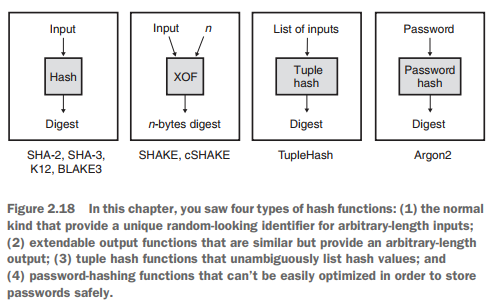
O primeiro problema tem sido comumente resolvido com o uso de **salts** — valores aleatórios que são **públicos e diferentes para cada usuário**. Usamos um **salt junto com a senha do usuário ao fazer o hash**, o que, de certo modo, é como usar **uma string de personalização por usuário com a cSHAKE**: isso efetivamente cria **uma função de hash diferente para cada usuário**. Como cada usuário usa uma função diferente, **um atacante não pode pré-computar grandes tabelas de senhas** (chamadas de **tabelas arco-íris**) esperando testar essas contra todo o banco de dados de senhas roubadas.

O segundo problema é resolvido com **funções de hash para senhas**, que são projetadas para serem **lentas**. A melhor opção atual é o **Argon2**, vencedor da **Password Hashing Competition**  
(<https://password-hashing.net>) que ocorreu de 2013 a 2015. No momento da redação deste livro (2021), o **Argon2 está prestes a ser padronizado como um RFC**  
(<https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/>).

Na prática, outros algoritmos não padronizados como **PBKDF2**, **bcrypt** e **scrypt** também são usados. O problema é que **eles podem ser utilizados com parâmetros inseguros** e, portanto, **não são fáceis de configurar corretamente**.

Além disso, **somente o Argon2 e o scrypt** defendem contra **grandes otimizações feitas por atacantes**, pois outros esquemas **não são resistentes à memória** (*not memory hard*). O termo **resistente à memória** significa que o algoritmo **só pode ser otimizado melhorando o acesso à memória**. Em outras palavras, otimizar o restante da execução **não traz muito ganho**. Como **otimizar acesso à memória é difícil até mesmo com hardware dedicado** (há um limite para o quanto de cache você pode colocar em torno de uma CPU), **funções resistentes à memória são lentas de rodar em qualquer tipo de dispositivo**.

Essa é uma propriedade desejada quando se quer impedir que atacantes obtenham **uma vantagem significativa em velocidade** ao executar uma função.

****

A figura acima resume os diferentes tipos de funções de hash que você viu neste capítulo:

1. A função de hash comum, que fornece um identificador aleatório e único para entradas de comprimento arbitrário.
2. Funções de saída extensível (XOFs), semelhantes, mas com saídas de comprimento arbitrário.
3. Funções de hash de tuplas, que listam valores de hash de maneira não ambígua.
4. Funções de hash para senhas, que **não podem ser facilmente otimizadas** e, portanto, **armazenam senhas com segurança**.

**Resumo**

* Uma **função de hash** oferece:
  + **Resistência à colisão**
  + **Resistência à pré-imagem**
  + **Resistência à segunda pré-imagem**
  + Resistência à pré-imagem significa que **não se deve conseguir encontrar a entrada que produziu um resumo**.
  + Resistência à segunda pré-imagem significa que **a partir de uma entrada e seu resumo, não se deve conseguir encontrar outra entrada que gere o mesmo resumo**.
  + Resistência à colisão significa que **não se deve conseguir encontrar duas entradas aleatórias que gerem a mesma saída**.
* A função de hash mais amplamente adotada é a **SHA-2**, enquanto a **SHA-3 é a recomendada**, devido à vulnerabilidade do SHA-2 a ataques de extensão de comprimento.
* **SHAKE** é uma função de saída extensível (XOF) que age como uma função de hash, mas **fornece um resumo de comprimento arbitrário**.
* **cSHAKE** (SHAKE personalizável) permite criar instâncias da SHAKE que se comportam como **XOFs distintas**. Isso se chama **separação de domínio**.
* Objetos devem ser **serializados antes de serem hasheados**, para evitar quebrar a resistência à segunda pré-imagem da função de hash. **TupleHash** faz isso automaticamente.
* O hashing de senhas usa **funções de hash lentas**, projetadas especificamente para esse propósito. **Argon2 é a melhor opção disponível atualmente**.