**Capítulo 7**

Assinaturas e provas de conhecimento zero

Você está prestes a aprender um dos primitivos criptográficos mais onipresentes e poderosos — assinaturas digitais. Simplificando, assinaturas digitais são semelhantes às assinaturas da vida real com as quais você está acostumado, aquelas que você assina em cheques e contratos. Exceto, é claro, que assinaturas digitais são criptográficas e, portanto, fornecem muito mais garantias do que suas equivalentes em papel e caneta.

No mundo dos protocolos, assinaturas digitais desbloqueiam tantas possibilidades diferentes que você as encontrará repetidamente na segunda parte deste livro. Neste capítulo, apresentarei o que é esse novo primitivo, como ele pode ser usado no mundo real e quais são os padrões modernos de assinaturas digitais. Por fim, falarei sobre considerações de segurança e os perigos do uso de assinaturas digitais.

Este capítulo cobre:

* Provas de conhecimento zero e assinaturas criptográficas
* Os padrões existentes para assinaturas criptográficas
* Os comportamentos sutis das assinaturas e como evitar suas armadilhas

**NOTA**: Assinaturas em criptografia são frequentemente chamadas de assinaturas digitais ou esquemas de assinatura. Neste livro, uso esses termos de forma intercambiável.

Para este capítulo, você precisará ter lido:

* Capítulo 2 sobre funções de hash
* Capítulo 5 sobre trocas de chaves
* Capítulo 6 sobre criptografia assimétrica

**7.1 O que é uma assinatura?**

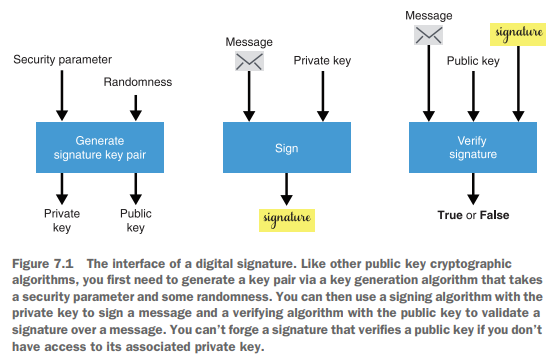
Expliquei no capítulo 1 que assinaturas criptográficas são bastante semelhantes às assinaturas da vida real. Por esse motivo, elas costumam ser um dos primitivos criptográficos mais intuitivos de se entender:

* Somente você pode usar sua assinatura para assinar mensagens arbitrárias.
* Qualquer pessoa pode verificar sua assinatura em uma mensagem.

Como estamos no reino da criptografia assimétrica, você provavelmente pode adivinhar como essa assimetria ocorrerá. Um esquema de assinatura normalmente consiste em três algoritmos diferentes:

* Um algoritmo de geração de par de chaves que um signatário usa para criar uma nova chave privada e pública (a chave pública pode então ser compartilhada com qualquer pessoa).
* Um algoritmo de assinatura que usa uma chave privada e uma mensagem para produzir uma assinatura.
* Um algoritmo de verificação que usa uma chave pública, uma mensagem e uma assinatura e retorna uma mensagem de sucesso ou erro.

Às vezes, a chave privada também é chamada de chave de assinatura, e a chave pública é chamada de chave de verificação. Faz sentido, certo? Recapitulei esses três algoritmos na figura 7.1.



Como as assinaturas são úteis? Elas servem para autenticar a origem de uma mensagem, bem como sua integridade:

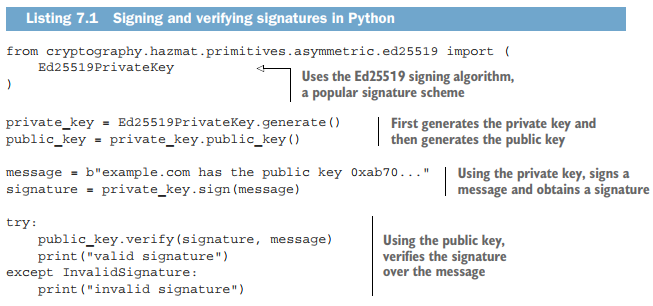
* **Origem** — Se minha assinatura está nela, veio de mim.
* **Integridade** — Se alguém modificar a mensagem, a assinatura perde a validade.

**NOTA**: Embora essas duas propriedades estejam ligadas à autenticação, muitas vezes são distinguidas como duas propriedades separadas: autenticação de origem e autenticação de mensagem (ou integridade).

De certo modo, as assinaturas são semelhantes aos códigos de autenticação de mensagens (MACs) que você aprendeu no capítulo 3. Mas, ao contrário dos MACs, permitem autenticar mensagens de forma assimétrica: um participante pode verificar se uma mensagem não foi adulterada sem conhecimento da chave privada ou de assinatura. A seguir, mostrarei como esses algoritmos podem ser usados na prática.

**7.1.1 Como assinar e verificar assinaturas na prática**

Vejamos um exemplo prático. Para isso, uso o pyca/cryptography (<https://cryptography.io>), uma biblioteca Python bem respeitada. A listagem a seguir simplesmente gera um par de chaves, assina uma mensagem usando a parte da chave privada e depois verifica a assinatura usando a parte da chave pública.





* Usa o algoritmo de assinatura Ed25519, um esquema de assinatura popular
* Primeiro gera a chave privada e depois gera a chave pública
* Usando a chave privada, assina uma mensagem e obtém uma assinatura
* Usando a chave pública, verifica a assinatura sobre a mensagem

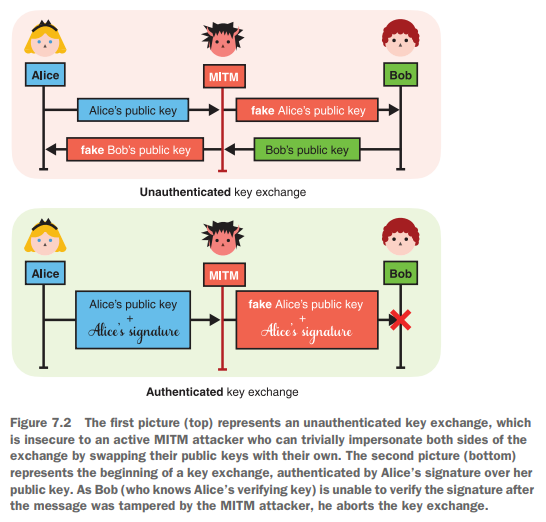
**Exercício**: Como você viu no capítulo 3, etiquetas de autenticação produzidas por MACs devem ser verificadas em tempo constante para evitar ataques de temporização. Você acha que devemos fazer o mesmo para verificar assinaturas?

**7.1.2 Um caso de uso principal para assinaturas: Trocas de chaves autenticadas**

Os capítulos 5 e 6 introduziram diferentes formas de realizar trocas de chaves entre dois participantes. Nesses capítulos, você aprendeu que essas trocas de chaves são úteis para negociar um segredo compartilhado, que pode então ser usado para proteger comunicações com um algoritmo de criptografia autenticada. No entanto, as trocas de chaves não resolveram completamente o problema de estabelecer uma conexão segura entre dois participantes, pois um atacante ativo do tipo *man-in-the-middle* (MITM) pode facilmente se passar por ambos os lados de uma troca de chaves. É aqui que as assinaturas entram em cena.

Imagine que Alice e Bob estão tentando estabelecer um canal de comunicação seguro entre si e que Bob conhece a chave de verificação de Alice. Sabendo disso, Alice pode usar sua chave de assinatura para autenticar seu lado da troca de chaves: ela gera um par de chaves de troca de chaves, assina a parte da chave pública com sua chave de assinatura e depois envia a chave pública de troca de chaves junto com a assinatura. Bob pode verificar que a assinatura é válida usando a chave de verificação associada que ele já conhece e, em seguida, usar a chave pública de troca de chaves para realizar a troca de chaves.

Chamamos essa troca de chaves de troca de chaves autenticada. Se a assinatura for inválida, Bob pode perceber que alguém está ativamente realizando um ataque MITM na troca de chaves. Ilustro trocas de chaves autenticadas na figura 7.2.



Note que, neste exemplo, a troca de chaves é autenticada apenas de um lado: enquanto Alice não pode ser personificada, Bob pode. Se ambos os lados forem autenticados (Bob também assinaria sua parte da troca de chaves), chamamos a troca de chaves de troca de chaves mutuamente autenticada. Assinar trocas de chaves pode não parecer muito útil ainda. Parece que apenas movemos o problema de não conhecer a chave pública de troca de chaves de Alice para o problema de não conhecer sua chave de verificação com antecedência. A próxima seção introduz um uso no mundo real de trocas de chaves autenticadas que fará muito mais sentido.

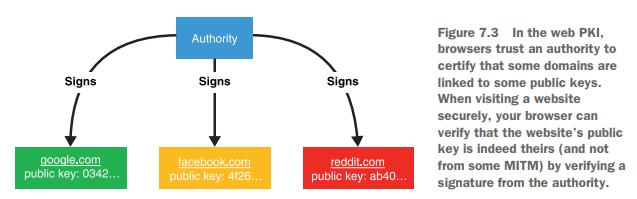
**7.1.3 Um uso no mundo real: Infraestruturas de chaves públicas**

As assinaturas tornam-se muito mais poderosas se você assumir que a confiança é transitiva. Com isso, quero dizer que, se você confia em mim e eu confio em Alice, então você pode confiar em Alice. Ela é confiável.

A transitividade da confiança permite escalar a confiança em sistemas de formas extremas. Imagine que você confia em alguma autoridade e na chave de verificação dela. Além disso, imagine que essa autoridade assinou mensagens indicando qual é a chave pública de Charles, qual é a chave pública de David, e assim por diante. Então, você pode optar por confiar nesse mapeamento! Tal mapeamento é chamado de infraestrutura de chaves públicas (*public key infrastructure* - PKI). Por exemplo, se você tentar realizar uma troca de chaves com Charles e ele afirmar que sua chave pública é um número grande que parece 3848..., você pode verificar isso checando se sua “amada” autoridade assinou alguma mensagem que diga “a chave pública de Charles é 3848...”.

Uma aplicação real desse conceito é a infraestrutura de chaves públicas da web (*web PKI*). A web PKI é o que seu navegador usa para autenticar as trocas de chaves que realiza com a infinidade de sites que você visita todos os dias. Uma explicação simplificada da web PKI (ilustrada na figura 7.3) é a seguinte: quando você baixa um navegador, ele vem com alguma chave de verificação embutida no programa. Essa chave de verificação está ligada a uma autoridade cuja responsabilidade é assinar milhares e milhares de chaves públicas de sites para que você possa confiar nelas sem conhecê-las previamente. O que você não vê é que esses sites precisam provar para a autoridade que realmente possuem seu nome de domínio antes de obterem uma assinatura sobre sua chave pública. (Na realidade, seu navegador confia em muitas autoridades para realizar esse trabalho, não apenas em uma única.)

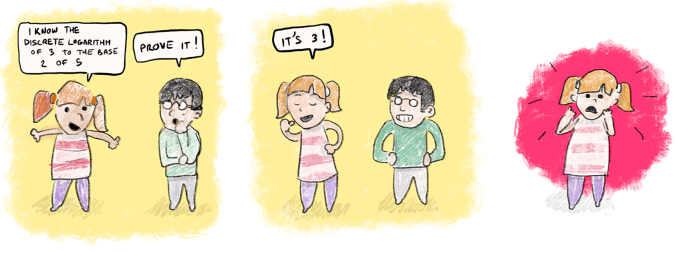
Nesta seção, você aprendeu sobre assinaturas de um ponto de vista de alto nível. Vamos nos aprofundar em como as assinaturas realmente funcionam. Mas, para isso, primeiro precisamos fazer um desvio e olhar para algo chamado prova de conhecimento zero (*zero-knowledge proof* - ZKP).



**7.2 Provas de conhecimento zero (ZKPs): A origem das assinaturas**

A melhor maneira de entender como as assinaturas funcionam na criptografia é entender de onde elas vêm. Por esse motivo, vamos tirar um momento para introduzir brevemente as ZKPs e depois voltarei às assinaturas.

Imagine que Peggy quer provar algo para Victor. Por exemplo, ela quer provar que conhece o logaritmo discreto na base de algum elemento de grupo. Em outras palavras, ela quer provar que conhece **x** dado **Y = g^x**, com **g** sendo o gerador de algum grupo.



Claro, a solução mais simples é Peggy simplesmente enviar o valor **x** (chamado de testemunha). Essa solução seria uma simples prova de conhecimento, e isso seria aceitável, a menos que Peggy não queira que Victor o descubra.

**NOTA**: Em termos teóricos, dizemos que o protocolo para produzir uma prova é **completo** se Peggy pode usá-lo para provar a Victor que ela conhece a testemunha. Se ela não puder provar o que sabe, então o esquema é inútil, certo?

Em criptografia, estamos principalmente interessados em provas de conhecimento que não revelem a testemunha ao verificador. Tais provas são chamadas de provas de conhecimento zero (*zero-knowledge proofs* - ZKPs).

**7.2.1 Protocolo de Identificação de Schnorr: Uma prova de conhecimento zero interativa**

Nas próximas páginas, construirei uma ZKP incrementalmente a partir de protocolos quebrados para mostrar como Alice pode provar que conhece **x** sem revelar **x**.

A maneira típica de abordar esse tipo de problema em criptografia é "esconder" o valor com alguma aleatoriedade (por exemplo, criptografando-o). Mas estamos fazendo mais do que apenas esconder: também queremos provar que ele está lá. Para isso, precisamos de uma maneira algébrica de escondê-lo. Uma solução simples é simplesmente adicionar um valor aleatório **k** gerado aleatoriamente à testemunha:

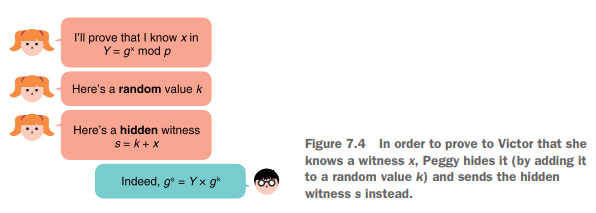
**s = k + x**

Peggy pode então enviar a testemunha escondida **s** junto com o valor aleatório **k** para Victor. Neste ponto, Victor não tem motivo para confiar que Peggy de fato escondeu a testemunha em **s**. De fato, se ela não conhece a testemunha **x**, então **s** provavelmente é apenas algum valor aleatório. O que Victor sabe é que a testemunha **x** está escondida no expoente de **g** porque ele conhece **Y = g^x**.

Para ver se Peggy realmente conhece a testemunha, Victor pode verificar se o que ela forneceu corresponde ao que ele sabe, e isso precisa ser feito no expoente de **g** (pois é onde está a testemunha). Em outras palavras, Victor verifica se esses dois números são iguais:

* **g^s (= g^{k+x})**
* **Y × g^k (= g^x × g^k = g^{x+k})**

A ideia é que somente alguém que conhece a testemunha **x** poderia ter construído uma testemunha "cegada" **s** que satisfaça essa equação. E assim, isso é uma prova de conhecimento. Recapitulei este sistema de ZKP na figura 7.4.



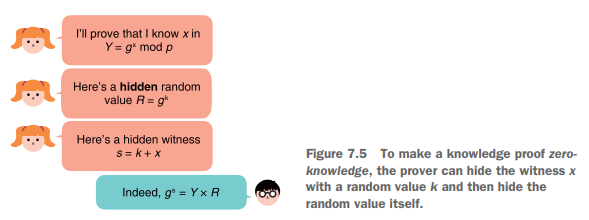
Não tão rápido. Há um problema com este esquema — ele obviamente não é seguro! De fato, porque a equação que esconde a testemunha **x** tem apenas uma incógnita (**x**), Victor pode simplesmente reverter a equação para recuperar a testemunha:

**x = s – k**

Para corrigir isso, Peggy pode esconder o próprio valor aleatório **k**! Desta vez, ela deve esconder o valor aleatório no expoente (em vez de adicioná-lo a outro valor aleatório) para garantir que a equação de Victor ainda funcione:

**R = g^k**

Dessa forma, Victor não aprende o valor **k** (este é o problema do logaritmo discreto abordado no capítulo 5) e, portanto, não pode recuperar a testemunha **x**. Ainda assim, ele tem informação suficiente para verificar que Peggy conhece **x**! Victor simplesmente tem que verificar se **g^s (= g^{k+x} = g^k × g^x)** é igual a **Y × R (= g^x × g^k)**. Revisarei essa segunda tentativa de protocolo ZKP na figura 7.5.



Existe ainda um último problema com nosso esquema — Peggy pode trapacear. Ela pode convencer Victor de que conhece **x** sem realmente conhecer **x**! Tudo o que ela precisa fazer é inverter o passo em que calcula sua prova. Ela primeiro gera um valor aleatório **s** e então calcula o valor **R** com base em **s**:

**R = g^s × Y^(-1)**

Victor então calcula **Y × R = Y × g^s × Y^(-1)**, o que de fato corresponde a **g^s**. (O truque de Peggy de usar um inverso para calcular um valor é usado em muitos ataques em criptografia.)

**NOTA**: Em termos teóricos, dizemos que o esquema é "sólido" se Peggy não pode trapacear (se ela não conhece **x**, então não pode enganar Victor).

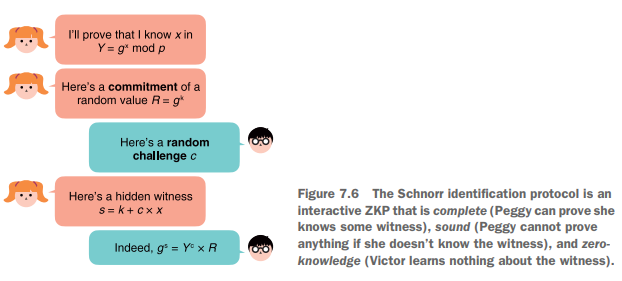
Para tornar o protocolo ZKP sólido, Victor deve garantir que Peggy compute **s** a partir de **R** e não o contrário. Para fazer isso, Victor torna o protocolo interativo:

1. Peggy deve se comprometer com seu valor aleatório **k** para que não possa alterá-lo depois.
2. Após receber o compromisso de Peggy, Victor introduz um pouco de sua própria aleatoriedade no protocolo. Ele gera um valor aleatório **c** (chamado de *desafio*) e o envia para Peggy.
3. Peggy então pode calcular seu compromisso escondido com base no valor aleatório **k** e no desafio **c**.

**NOTA**: Você aprendeu sobre esquemas de compromisso no capítulo 2, onde usamos uma função de hash para se comprometer com um valor que podemos revelar posteriormente. Mas esquemas de compromisso baseados em funções de hash não nos permitem fazer aritmética interessante no valor escondido. Em vez disso, podemos simplesmente elevar nosso gerador ao valor, **g^k**, o que já estamos fazendo.

Como Peggy não pode realizar o último passo sem o desafio **c** de Victor, e Victor não o enviará sem ver um compromisso no valor aleatório **k**, Peggy é forçada a computar **s** com base em **k**. O protocolo obtido, que ilustro na figura 7.6, é frequentemente chamado de protocolo de identificação de Schnorr.

Os chamados sistemas ZKP interativos que seguem um padrão de três movimentos (compromisso, desafio e prova) são frequentemente chamados de protocolos Sigma na literatura e às vezes escritos como **Σ** devido ao formato ilustrativo da letra grega.



**NOTA**: O protocolo de identificação de Schnorr funciona no modelo de conhecimento zero com verificador honesto (HVZK): se o verificador (Victor) agir de forma desonesta e não escolher o desafio de forma aleatória, ele pode aprender algo sobre a testemunha. Alguns esquemas de ZKP mais fortes são conhecimento zero mesmo quando o verificador é malicioso.

**7.2.2 Assinaturas como provas de conhecimento zero não interativas**

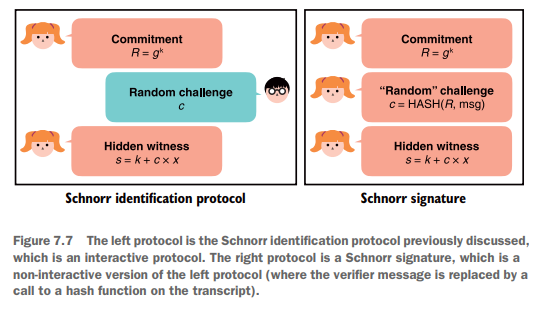
O problema com a ZKP interativa anterior é que, bem, ela é interativa, e protocolos do mundo real geralmente não gostam de interatividade. Protocolos interativos adicionam um overhead não desprezível, pois requerem várias mensagens (potencialmente através da rede) e introduzem atrasos indefinidos, a menos que os dois participantes estejam online ao mesmo tempo. Por isso, ZKPs interativas estão praticamente ausentes do mundo da criptografia aplicada.

Mas toda essa discussão não é em vão! Em 1986, Amos Fiat e Adi Shamir publicaram uma técnica que permitia facilmente converter uma ZKP interativa em uma ZKP não interativa. O truque que eles introduziram (conhecido como heurística de Fiat-Shamir ou transformação de Fiat-Shamir) foi fazer o próprio provador calcular o desafio de forma que ele não pudesse controlá-lo.

Aqui está o truque — calcular o desafio como o hash de todas as mensagens enviadas e recebidas como parte do protocolo até aquele ponto (o que chamamos de *transcrição*). Se assumirmos que a função hash gera saídas indistinguíveis de números verdadeiramente aleatórios (em outras palavras, que parecem aleatórios), então ela pode simular com sucesso o papel do verificador.

Schnorr foi um passo além. Ele percebeu que qualquer coisa pode ser incluída nesse hash! Por exemplo, e se incluíssemos uma mensagem ali? O que obtemos não é apenas uma prova de que conhecemos alguma testemunha **x**, mas um compromisso com uma mensagem que está criptograficamente ligada à prova. Em outras palavras, se a prova for correta, então somente alguém com o conhecimento da testemunha (que se torna a chave de assinatura) poderia ter comprometido aquela mensagem.

Isso é uma assinatura! Assinaturas digitais são apenas ZKPs não interativas. Aplicando a transformação de Fiat-Shamir ao protocolo de identificação de Schnorr, obtemos o esquema de assinatura de Schnorr, que ilustro na figura 7.7.



Recapitulando, uma assinatura de Schnorr é essencialmente dois valores, **R** e **s**, onde:

* **R** é um compromisso com algum valor aleatório secreto (frequentemente chamado de *nonce*, pois precisa ser único por assinatura);
* **s** é um valor computado com a ajuda do compromisso **R**, da chave privada (a testemunha **x**) e de uma mensagem.

Em seguida, vamos analisar os padrões modernos de algoritmos de assinatura.

**7.3 Os algoritmos de assinatura que você deve usar (ou não)**

Como em outros campos da criptografia, assinaturas digitais possuem muitos padrões, e às vezes é difícil entender qual deles usar. É por isso que estou aqui! Felizmente, os tipos de algoritmos para assinaturas são semelhantes aos de trocas de chaves: há algoritmos baseados em aritmética módulo um número grande, como Diffie-Hellman (DH) e RSA, e há algoritmos baseados em curvas elípticas, como Elliptic Curve Diffie-Hellman (ECDH).

Certifique-se de entender bem os algoritmos nos capítulos 5 e 6, pois agora vamos construir a partir deles. Curiosamente, o artigo que introduziu a troca de chaves DH também propôs o conceito de assinaturas digitais (sem apresentar uma solução):

Para desenvolver um sistema capaz de substituir o atual contrato escrito por alguma forma puramente eletrônica de comunicação, devemos descobrir um fenômeno digital com as mesmas propriedades de uma assinatura escrita. Deve ser fácil para qualquer pessoa reconhecer a assinatura como autêntica, mas impossível para qualquer um além do signatário legítimo produzi-la. Chamaremos tal técnica de autenticação unidirecional. Como qualquer sinal digital pode ser copiado precisamente, uma verdadeira assinatura digital deve ser reconhecível sem ser conhecida.

— *Diffie e Hellman (“New Directions in Cryptography,” 1976)*

Um ano depois (em 1977), o primeiro algoritmo de assinatura (chamado RSA) foi introduzido juntamente com o algoritmo de criptografia assimétrica RSA (que você aprendeu no capítulo 6). O RSA para assinatura é o primeiro algoritmo que veremos nesta seção.

Em 1991, o NIST propôs o Digital Signature Algorithm (DSA) como uma tentativa de evitar as patentes sobre as assinaturas de Schnorr. Por esse motivo, o DSA é uma variante estranha das assinaturas de Schnorr, publicada sem uma prova de segurança (embora nenhum ataque tenha sido encontrado até agora). O algoritmo foi adotado por muitos, mas rapidamente substituído por uma versão de curvas elípticas chamada ECDSA (Elliptic Curve Digital Signature Algorithm), da mesma forma que o Elliptic Curve Diffie-Hellman (ECDH) substituiu o Diffie-Hellman (DH), graças às suas chaves menores (veja o capítulo 5). O ECDSA é o segundo algoritmo de assinatura que abordarei nesta seção.

Após a expiração das patentes sobre as assinaturas de Schnorr em 2008, Daniel J. Bernstein, o inventor do ChaCha20-Poly1305 (abordado no capítulo 4) e do X25519 (abordado no capítulo 5), introduziu um novo esquema de assinatura chamado EdDSA (Edwards-curve Digital Signature Algorithm), baseado nas assinaturas de Schnorr. Desde sua invenção, o EdDSA rapidamente ganhou adoção e hoje é considerado o estado da arte em termos de assinatura digital para aplicações do mundo real. O EdDSA é o terceiro e último algoritmo de assinatura que abordarei nesta seção.

**7.3.1 RSA PKCS#1 v1.5: Um padrão ruim**

Assinaturas RSA ainda são amplamente usadas, embora não devessem ser (como você verá nesta seção, elas apresentam muitos problemas). Isso ocorre porque o algoritmo foi o primeiro esquema de assinatura a ser padronizado, além de as aplicações do mundo real serem lentas para adotar algoritmos mais novos e melhores. Por causa disso, você muito provavelmente encontrará assinaturas RSA em sua jornada, e não posso deixar de explicar como elas funcionam e quais padrões são os adotados. Mas deixe-me dizer que, se você entendeu como a criptografia RSA funciona no capítulo 6, então esta seção deve ser simples porque assinar com RSA é o oposto de criptografar com RSA:

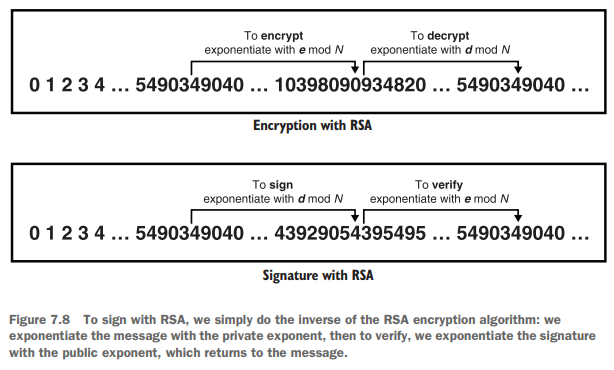
* Para assinar, você criptografa a mensagem com a chave privada (em vez da chave pública), o que produz uma assinatura (um elemento aleatório no grupo).
* Para verificar uma assinatura, você descriptografa a assinatura com a chave pública (em vez da chave privada). Se isso lhe devolver a mensagem original, então a assinatura é válida.

**NOTA**: Na realidade, uma mensagem é frequentemente hashada antes de ser assinada, pois ocupará menos espaço (o RSA só pode assinar mensagens menores que seu módulo). O resultado também é interpretado como um número grande para que possa ser usado em operações matemáticas.

Se sua chave privada é o expoente privado **d**, e sua chave pública é o expoente público **e** e o módulo público **N**, você pode:

* Assinar uma mensagem calculando:  
  **assinatura = mensagem^d mod N**
* Verificar uma assinatura calculando:  
  **assinatura^e mod N** e verificando se é igual à mensagem.

Ilustro isso visualmente na figura 7.8.

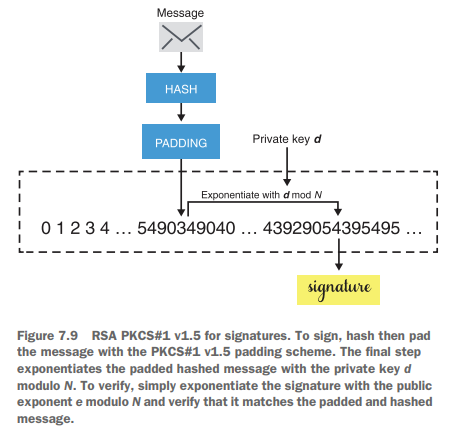


Isso funciona porque somente quem conhece o expoente privado **d** pode produzir uma assinatura sobre uma mensagem. E, assim como na criptografia RSA, a segurança está intimamente ligada à dificuldade do problema de fatoração.

E quanto aos padrões para usar RSA em assinaturas? Felizmente, eles seguem o mesmo padrão que o RSA para criptografia:

* O RSA para criptografia foi vagamente padronizado no documento PKCS#1 v1.5. O mesmo documento continha uma especificação para assinatura com RSA (sem uma prova de segurança).
* O RSA foi então novamente padronizado no documento PKCS#1 v2 com uma construção melhor (chamada RSA-OAEP). O mesmo aconteceu com as assinaturas RSA com o RSA-PSS sendo padronizado no mesmo documento (com uma prova de segurança).

Falei sobre RSA PKCS#1 v1.5 no capítulo 6 sobre criptografia assimétrica. O esquema de assinatura padronizado nesse documento é praticamente o mesmo que o esquema de criptografia. Para assinar, primeiro faça o hash da mensagem com uma função hash de sua escolha, depois preencha (pad) de acordo com o preenchimento PKCS#1 v1.5 para assinaturas (que é semelhante ao preenchimento para criptografia no mesmo padrão). Depois, criptografe a mensagem hashada e preenchida com seu expoente privado. Ilustro isso na figura 7.9.



No capítulo 6, mencionei que houve ataques devastadores contra o RSA PKCS#1 v1.5 para criptografia; o mesmo infelizmente é verdadeiro para assinaturas RSA PKCS#1 v1.5. Em 1998, depois que Bleichenbacher encontrou um ataque devastador contra o RSA PKCS#1 v1.5 para criptografia, ele decidiu dar uma olhada no padrão de assinatura. Bleichenbacher voltou em 2006 com um ataque de falsificação de assinatura contra o RSA PKCS#1 v1.5, um dos tipos de ataque mais catastróficos em assinaturas — atacantes podem forjar assinaturas sem conhecimento da chave privada! Ao contrário do primeiro ataque que quebrou diretamente o algoritmo de criptografia, o segundo ataque foi um ataque de implementação. Isso significava que, se o esquema de assinatura fosse implementado corretamente (de acordo com a especificação), o ataque não funcionaria.

Um erro de implementação não soa tão ruim quanto um erro de algoritmo, isto é, se for fácil de evitar e não afetar muitas implementações. Infelizmente, foi demonstrado em 2019 que um número embaraçoso de implementações open source do RSA PKCS#1 v1.5 para assinaturas caiu nessa armadilha e implementou incorretamente o padrão (veja “Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification” de Chau et al.). As diversas falhas de implementação acabaram permitindo diferentes variantes do ataque de falsificação de Bleichenbacher.

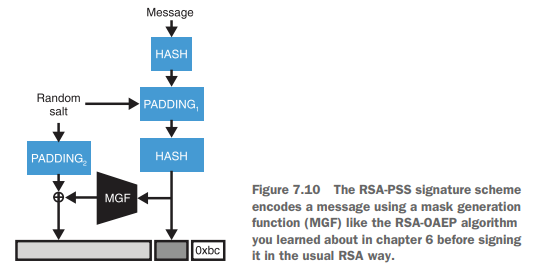
Infelizmente, o RSA PKCS#1 v1.5 para assinaturas ainda é amplamente usado. Esteja ciente desses problemas caso realmente precise usar esse algoritmo por motivos de compatibilidade retroativa. Dito isso, isso não significa que o RSA para assinaturas é inseguro. A história não termina aqui.

**7.3.2 RSA-PSS: Um padrão melhor**

O RSA-PSS foi padronizado na atualização PKCS#1 v2.1 e incluiu uma prova de segurança (ao contrário do esquema de assinatura padronizado no PKCS#1 v1.5). A especificação mais recente funciona assim:

* Codifica a mensagem usando o algoritmo de codificação PSS.
* Assina a mensagem codificada usando o RSA (como era feito no padrão PKCS#1 v1.5).

A codificação PSS é um pouco mais elaborada e semelhante ao OAEP (Optimal Asymmetric Encryption Padding). Eu ilustro isso na figura 7.10.



Verificar uma assinatura produzida pelo RSA-PSS é apenas uma questão de inverter a codificação uma vez que a assinatura tenha sido elevada ao expoente público módulo o módulo público.

Se você se lembra, também falei sobre um terceiro algoritmo no capítulo 6 para criptografia RSA (chamado RSA-KEM) — um algoritmo mais simples que não é usado por ninguém e ainda assim é provado ser seguro. Curiosamente, o RSA para assinaturas também reflete essa parte da história da criptografia RSA e possui um algoritmo muito mais simples que praticamente ninguém usa; ele é chamado de Full Domain Hash (FDH). O FDH funciona simplesmente fazendo o hash de uma mensagem e depois assinando-a (interpretando o digest como um número) usando o RSA.

Apesar do fato de que tanto o RSA-PSS quanto o FDH possuem provas de segurança e são muito mais fáceis de implementar corretamente, hoje a maioria dos protocolos ainda utiliza o RSA PKCS#1 v1.5 para assinaturas. Este é apenas mais um exemplo da lentidão que geralmente ocorre ao descontinuar algoritmos criptográficos. Como implementações antigas ainda precisam funcionar com implementações mais novas, é difícil remover ou substituir algoritmos. Pense em usuários que não atualizam aplicativos, fornecedores que não fornecem novas versões de seus softwares, dispositivos de hardware que não podem ser atualizados, e assim por diante. Em seguida, vejamos um algoritmo mais moderno.

**7.3.3 O Algoritmo de Assinatura Digital de Curvas Elípticas (ECDSA)**

Nesta seção, vamos analisar o ECDSA, uma variante em curvas elípticas do DSA que foi inventada apenas para contornar as patentes das assinaturas de Schnorr. O esquema de assinatura é especificado em muitos padrões incluindo ISO 14888-3, ANSI X9.62, FIPS 186-2 do NIST, IEEE P1363, e assim por diante. Nem todos os padrões são compatíveis, e aplicações que desejam interoperar precisam garantir que estão usando o mesmo padrão.

Infelizmente, o ECDSA, assim como o DSA, não possui uma prova de segurança, enquanto as assinaturas de Schnorr possuíam. No entanto, o ECDSA foi amplamente adotado e é um dos esquemas de assinatura mais usados. Nesta seção, explicarei como o ECDSA funciona e como pode ser utilizado. Como em todos esses esquemas, a chave pública é quase sempre gerada de acordo com a mesma fórmula:

* A chave privada é um número grande **x** gerado aleatoriamente.
* A chave pública é obtida visualizando **x** como um índice em um grupo criado por um gerador (chamado de ponto base na criptografia de curvas elípticas).

Mais especificamente, no ECDSA, a chave pública é computada usando **[x]G**, que é uma multiplicação escalar do escalar **x** com o ponto base **G**.

**Segurança provável para o PSS**

PSS (Probabilistic Signature Scheme) é provadamente seguro, o que significa que ninguém deveria ser capaz de forjar uma assinatura sem conhecimento da chave privada. Em vez de provar que se o RSA é seguro então o RSA-PSS é seguro, o RSA-PSS prova o contrarrecíproco: se alguém pode quebrar o RSA-PSS, então esse alguém também pode quebrar o RSA. Essa é uma forma comum de se provar coisas em criptografia. Claro, isso só funciona se o RSA for seguro, o que assumimos na prova.

Para calcular uma assinatura ECDSA, você precisa dos mesmos insumos exigidos por uma assinatura de Schnorr: um hash da mensagem que você está assinando (**H(m)**), sua chave privada **x**, e um número aleatório **k** que deve ser único por assinatura. Uma assinatura ECDSA é composta por dois inteiros, **r** e **s**, calculados da seguinte forma:

* **r** é a coordenada x de **[k]G**
* **s** é igual a **k⁻¹ (H(m) + x·r) mod p**

Para verificar uma assinatura ECDSA, o verificador precisa usar o mesmo hash da mensagem **H(m)**, a chave pública do assinante e os valores de assinatura **r** e **s**. O verificador então:

1. Calcula **[H(m)·s⁻¹]G + [r·s⁻¹]public\_key**
2. Valida que a coordenada x do ponto obtido é igual ao valor **r** da assinatura

Você certamente pode perceber que há algumas semelhanças com as assinaturas de Schnorr. O número aleatório **k** às vezes é chamado de *nonce* porque é um número que deve ser usado apenas uma vez, e também às vezes é chamado de chave efêmera porque deve permanecer secreto.

**AVISO**: Vou enfatizar isso: **k** nunca deve ser repetido nem previsível! Sem isso, é trivial recuperar a chave privada.

Em geral, bibliotecas criptográficas realizam a geração deste nonce (o valor **k**) nos bastidores, mas às vezes não o fazem e deixam o usuário fornecê-lo. Isso, claro, é uma receita para o desastre. Por exemplo, em 2010, foi descoberto que o Playstation 3 da Sony usava ECDSA com nonces repetidos (o que vazou suas chaves privadas).

**AVISO**: Ainda mais sutil, se o nonce **k** não for escolhido de forma uniforme e aleatória (especificamente, se for possível prever os primeiros bits), ainda existem ataques poderosos que podem recuperar a chave privada rapidamente (os chamados ataques de reticulado, *lattice attacks*). Em teoria, chamamos esses ataques de quebras totais (*total breaks*), pois comprometem tudo! Tais quebras totais são bastante raras na prática, o que faz do ECDSA um algoritmo que pode falhar de forma espetacular.

Tentativas de evitar problemas com nonces existem. Por exemplo, o RFC 6979 especifica um esquema determinístico de ECDSA que gera um nonce com base na mensagem e na chave privada. Isso significa que assinar a mesma mensagem duas vezes envolve o mesmo nonce duas vezes e, assim, produz a mesma assinatura duas vezes (o que obviamente não é um problema).

**Notação aditiva ou multiplicativa?**

Note que estou usando a notação aditiva (com a sintaxe de curvas elípticas de colocar colchetes em torno do escalar), mas eu poderia ter escrito **public\_key = Gˣ** se quisesse usar a notação multiplicativa. Essas diferenças não importam na prática. Na maioria das vezes, protocolos criptográficos que não se importam com a natureza subjacente do grupo são escritos usando a notação multiplicativa, enquanto protocolos que são definidos especificamente em grupos de curvas elípticas tendem a ser escritos usando a notação aditiva.

Na prática, as curvas elípticas que tendem a ser usadas com ECDSA são praticamente as mesmas curvas populares no algoritmo Elliptic Curve Diffie-Hellman (ECDH) (veja o capítulo 5), com uma exceção notável: **Secp256k1**. A curva Secp256k1 é definida em SEC 2: "Recommended Elliptic Curve Domain Parameters" (<https://secg.org/sec2-v2.pdf>), escrito pelo Standards for Efficient Cryptography Group (SECG). Ela ganhou muita tração depois que o Bitcoin decidiu usá-la em vez das curvas NIST mais populares, devido à falta de confiança nas curvas NIST mencionada no capítulo 5.

Secp256k1 é um tipo de curva elíptica chamada de **curva de Koblitz**. Uma curva de Koblitz é simplesmente uma curva elíptica com algumas restrições em seus parâmetros que permitem otimizações em algumas operações na curva. A curva elíptica tem a seguinte equação:

**y² = x³ + ax + b**

onde **a = 0** e **b = 7** são constantes, e **x** e **y** são definidos sobre os números módulo o primo **p**:

**p = 2²⁵⁶ – 2³² – 2¹² – 2⁸ – 2⁷ – 2⁶ – 2³ – 1**

Isso define um grupo de ordem prima, como as curvas NIST. Hoje, temos fórmulas eficientes para calcular o número de pontos em uma curva elíptica. Aqui está o número primo que representa a quantidade de pontos na curva Secp256k1 (incluindo o ponto no infinito):

115792089237316195423570985008687907852837564279074904382605163141518161494337

E usamos como gerador (ou ponto base) o ponto fixo **G** de coordenadas:

**x = 55066263022277343669578718895168534326250603453777594175500187360389116729240**

e

**y = 32670510020758816978083085130507043184471273380659243275938904335757337482424**

Hoje, o ECDSA é usado principalmente com a curva NIST P-256 (às vezes chamada de **Secp256r1**; note a diferença).

Em seguida, vamos analisar outro esquema de assinatura amplamente popular.

**7.3.4 O Algoritmo de Assinatura Digital de Curva de Edwards (EdDSA)**

Permita-me apresentar o último algoritmo de assinatura deste capítulo, o Algoritmo de Assinatura Digital de Curva de Edwards (EdDSA), publicado em 2011 por Daniel J. Bernstein em resposta à falta de confiança nas curvas NIST e em outras curvas criadas por agências governamentais.

O nome **EdDSA** parece indicar que ele é baseado no algoritmo DSA como o ECDSA é, mas isso é enganoso. O EdDSA é, na verdade, baseado nas assinaturas de Schnorr, o que foi possível devido à expiração da patente das assinaturas de Schnorr em 2008.

Uma particularidade do EdDSA é que o esquema não requer nova aleatoriedade para cada operação de assinatura. O EdDSA produz assinaturas de forma determinística. Isso tornou o algoritmo bastante atraente, e ele tem sido adotado por muitos protocolos e padrões.

O EdDSA está a caminho de ser incluído na próxima atualização do padrão FIPS 186-5 do NIST (ainda em rascunho no início de 2021). O padrão oficial atual é o RFC 8032, que define duas curvas de diferentes níveis de segurança a serem usadas com o EdDSA. Ambas as curvas definidas são curvas de Edwards torcidas (*twisted Edwards curves*, um tipo de curva elíptica que permite otimizações interessantes na implementação):

* **Edwards25519** é baseada na Curve25519 de Daniel J. Bernstein (abordada no capítulo 5). Suas operações na curva podem ser implementadas mais rapidamente do que as da Curve25519, graças às otimizações permitidas pelo tipo de curva elíptica. Como foi inventada depois da Curve25519, a troca de chaves X25519 baseada na Curve25519 não se beneficiou dessas melhorias de velocidade. Assim como a Curve25519, a Edwards25519 fornece segurança de 128 bits.
* **Edwards448** é baseada na curva Ed448-Goldilocks de Mike Hamburg. Ela fornece segurança de 224 bits.

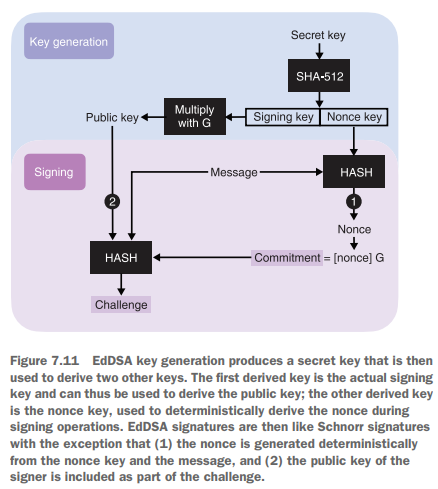
Na prática, o EdDSA é majoritariamente instanciado com a curva Edwards25519 e essa combinação é chamada de **Ed25519** (enquanto o EdDSA com Edwards448 é abreviado como **Ed448**). A geração de chaves com o EdDSA é um pouco diferente dos outros esquemas existentes. Em vez de gerar uma chave de assinatura diretamente, o EdDSA gera uma chave secreta que é então usada para derivar a chave de assinatura propriamente dita e uma outra chave que chamamos de **chave de nonce**. Essa chave de nonce é importante! É ela que será usada para gerar deterministicamente o nonce necessário por assinatura.

**NOTA**: Dependendo da biblioteca criptográfica que você estiver usando, pode ser que você esteja armazenando a chave secreta ou as duas chaves derivadas: a chave de assinatura e a chave de nonce. Isso não faz muita diferença na prática, mas, se você não souber disso, pode ficar confuso ao se deparar com chaves secretas Ed25519 sendo armazenadas com 32 ou 64 bytes, dependendo da implementação utilizada.

Para assinar, o EdDSA primeiro gera determinísticamente o nonce, fazendo o hash da chave de nonce com a mensagem a ser assinada. Depois disso, um processo semelhante ao das assinaturas de Schnorr é seguido:

1. Calcula o nonce como **HASH(chave de nonce || mensagem)**
2. Calcula o compromisso **R** como **[nonce]G**, onde **G** é o ponto base do grupo
3. Calcula o desafio como **HASH(compromisso || chave pública || mensagem)**
4. Calcula a prova **S** como **nonce + desafio × chave de assinatura**

A assinatura é o par **(R, S)**. Eu ilustro as partes importantes do EdDSA na figura 7.11.



Note como o nonce (ou chave efêmera) é derivado de forma determinística e não probabilística a partir da chave de nonce e da mensagem dada. Isso significa que assinar duas mensagens diferentes envolve dois nonces diferentes, prevenindo engenhosamente o signatário de reutilizar nonces e, consequentemente, vazar a chave (como pode acontecer com o ECDSA). Assinar a mesma mensagem duas vezes produz o mesmo nonce duas vezes, o que produz a mesma assinatura duas vezes também. Isso obviamente não é um problema.

Uma assinatura pode ser verificada computando as seguintes duas expressões:

* **[S]G**
* **R + [HASH(R || chave pública || mensagem)] × chave pública**

A assinatura é válida se os dois valores coincidirem. Isso é exatamente como funcionam as assinaturas de Schnorr, exceto que agora estamos em um grupo de curva elíptica e estou usando aqui a notação aditiva.

A instância mais amplamente utilizada do EdDSA, o **Ed25519**, é definida com a curva **Edwards25519** e o hash **SHA-512** como função de hash. A curva Edwards25519 é definida com todos os pontos que satisfazem esta equação:

**–x² + y² = 1 + d × x² × y² mod p**

onde o valor **d** é o número grande:

37095705934669439343138083508754565189542113879843219016388785533085940283555

e as variáveis **x** e **y** são tomadas módulo o primo **p = 2²⁵⁵ – 19** (o mesmo primo usado para a Curve25519). O ponto base **G** tem as coordenadas:

**x = 15112221349535400772501151409588531511454012693041857206046113283949847762202**

e

**y = 46316835694926478169428394003475163141307993866256225615783033603165251855960**

O RFC 8032 define, na verdade, três variantes do EdDSA usando a curva Edwards25519. Todas as três variantes seguem o mesmo algoritmo de geração de chaves, mas com algoritmos de assinatura e verificação diferentes:

* **Ed25519 (ou pureEd25519)** — É o algoritmo que expliquei anteriormente.
* **Ed25519ctx** — Este algoritmo introduz uma string de personalização obrigatória e é raramente implementado, se é que é utilizado na prática. A única diferença é que algum prefixo escolhido pelo usuário é adicionado a cada chamada da função de hash.
* **Ed25519ph (ou HashEd25519)** — Permite que aplicações pré-hashem a mensagem antes de assiná-la (daí o *ph* no nome). Também é baseado no Ed25519ctx, permitindo que o chamador inclua uma string personalizada opcional.

A adição de uma string de personalização é bastante comum em criptografia, como você viu com algumas funções de hash no capítulo 2 ou verá com funções de derivação de chaves no capítulo 8. É uma adição útil quando um participante de um protocolo usa a mesma chave para assinar mensagens em contextos diferentes. Por exemplo, você pode imaginar uma aplicação que permitiria assinar transações usando sua chave privada e também assinar mensagens privadas para pessoas com quem você conversa. Se você, por engano, assinar e enviar uma mensagem que parece uma transação para sua amiga maliciosa Eve, ela poderia tentar republicá-la como uma transação válida se não houver uma forma de distinguir os dois tipos de mensagens que você está assinando.

O Ed25519ph foi introduzido unicamente para atender chamadores que precisam assinar mensagens grandes. Como você viu no capítulo 2, funções de hash frequentemente fornecem uma interface “inicializar-atualizar-finalizar” que permite fazer o hash contínuo de um fluxo de dados sem precisar manter toda a entrada na memória.

Você agora terminou sua jornada pelos esquemas de assinatura utilizados em aplicações do mundo real.  
Em seguida, vamos ver como você pode, possivelmente, se colocar em apuros ao utilizar esses algoritmos de assinatura.

Mas antes, um resumo:

**7.4 Comportamentos sutis dos esquemas de assinatura**

Existem várias propriedades sutis que os esquemas de assinatura podem apresentar. Embora elas possam não importar na maioria dos protocolos, não estar ciente dessas “pegadinhas” pode acabar lhe causando problemas ao trabalhar com protocolos mais complexos e não convencionais.

O final deste capítulo foca em problemas conhecidos com assinaturas digitais.

**7.4.1 Ataques de substituição em assinaturas**

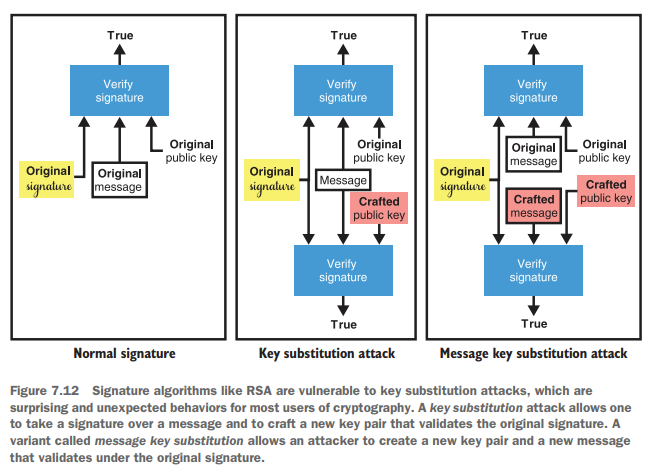
Uma assinatura digital não identifica de forma única uma chave ou uma mensagem.

— *Andrew Ayer (“Duplicate Signature Key Selection Attack in Let’s Encrypt,” 2015)*

Ataques de substituição, também chamados de **duplicate signature key selection (DSKS)**, são possíveis tanto no RSA PKCS#1 v1.5 quanto no RSA-PSS. Existem duas variantes do DSKS:

* **Ataques de substituição de chave (Key substitution attacks)** — Um par de chaves diferente ou uma chave pública diferente é usado para validar uma determinada assinatura sobre uma determinada mensagem.
* **Ataques de substituição de chave e mensagem (Message key substitution attacks)** — Um par de chaves diferente ou uma chave pública diferente é usado para validar uma determinada assinatura sobre uma nova mensagem.

Mais uma vez: o primeiro ataque fixa tanto a mensagem quanto a assinatura; o segundo fixa apenas a assinatura. Recapitulei isso na figura 7.12.



**Inviolabilidade existencial sob ataque adaptativo de mensagem escolhida (EUF-CMA)**

Ataques de substituição são um sintoma da lacuna entre a criptografia teórica e a criptografia aplicada. Assinaturas em criptografia são geralmente analisadas sob o modelo **EUF-CMA** (*existential unforgeability under adaptive chosen message attack*).

Neste modelo, você gera um par de chaves e, em seguida, eu solicito que você assine várias mensagens arbitrárias. Enquanto observo as assinaturas que você produz, eu venço se conseguir, em algum momento, produzir uma assinatura válida sobre uma mensagem que não havia solicitado antes.

Infelizmente, esse modelo EUF-CMA não parece abranger todos os casos extremos, e sutilezas perigosas como os ataques de substituição não são levadas em conta.

**7.4.2 Maleabilidade de assinaturas**

Em fevereiro de 2014, o MtGox, que já foi a maior exchange de Bitcoin, fechou e pediu falência alegando que atacantes usaram ataques de maleabilidade para esvaziar suas contas.

— *Christian Decker e Roger Wattenhofer (“Bitcoin Transaction Malleability and MtGox,” 2014)*

A maioria dos esquemas de assinatura é **maleável**: se você me der uma assinatura válida, eu posso modificá-la de forma que ela se torne uma assinatura diferente, mas ainda válida. Eu não faço ideia de qual era a chave de assinatura, e mesmo assim consigo criar uma nova assinatura válida.

A não maleabilidade não significa necessariamente que as assinaturas são únicas: se eu for o assinante, geralmente posso criar diferentes assinaturas para a mesma mensagem e isso normalmente não é um problema.

Algumas construções, como **funções randômicas verificáveis** (*verifiable random functions*), que você verá mais adiante no capítulo 8, dependem da unicidade das assinaturas, e por isso precisam lidar com isso ou utilizar esquemas de assinatura que tenham assinaturas únicas (como as assinaturas de Boneh–Lynn–Shacham, ou BLS).

O que fazer com todas essas informações?

Fique tranquilo, os esquemas de assinatura definitivamente não estão quebrados, e você provavelmente não deve se preocupar se seu uso de assinaturas não for muito fora do comum. Mas se você estiver projetando protocolos criptográficos ou implementando um protocolo mais complexo do que a criptografia do dia a dia, pode ser interessante manter essas propriedades sutis em mente.

**O forte EUF-CMA**

Um modelo de segurança mais recente chamado **SUF-CMA** (*strong EUF-CMA*) tenta incluir a não maleabilidade (ou resistência à maleabilidade) na definição de segurança de esquemas de assinatura.

Alguns padrões recentes como o **RFC 8032**, que especifica o Ed25519, incluem proteções contra ataques de maleabilidade. Como essas proteções nem sempre estão presentes ou não são comuns, você nunca deve confiar na não maleabilidade das assinaturas em seus protocolos.

**Resumo**

* Assinaturas digitais são semelhantes às assinaturas em papel, mas respaldadas pela criptografia, tornando-as infalsificáveis por qualquer um que não controle a chave de assinatura (privada).
* Assinaturas digitais podem ser úteis para autenticar origens (por exemplo, um lado de uma troca de chaves), além de fornecer confiança transitiva (se eu confio na Alice e ela confia no Bob, eu posso confiar no Bob).
* Provas de conhecimento zero (ZKPs) permitem que um provador prove o conhecimento de uma determinada informação (chamada de testemunha), sem revelar essa informação. Assinaturas podem ser vistas como ZKPs não interativas, pois não exigem que o verificador esteja online durante a operação de assinatura.
* Você pode usar vários padrões para assinar:
  + **RSA PKCS#1 v1.5** ainda é amplamente usado, mas não recomendado, pois é difícil de implementar corretamente.
  + **RSA-PSS** é um esquema de assinatura melhor, mais fácil de implementar e possui prova de segurança. Infelizmente, hoje não é muito popular devido às variantes em curvas elípticas, que suportam chaves menores e, portanto, são mais atraentes para protocolos de rede.
  + Os esquemas de assinatura mais populares atualmente são baseados em curvas elípticas: **ECDSA** e **EdDSA**. O ECDSA é frequentemente usado com a curva **P-256** do NIST, enquanto o EdDSA é frequentemente usado com a curva **Edwards25519** (esta combinação é chamada de **Ed25519**).
* Algumas propriedades sutis podem ser perigosas se as assinaturas forem usadas de forma não convencional:
  + Sempre evite ambiguidade sobre quem assinou uma mensagem, pois alguns esquemas de assinatura são vulneráveis a ataques de substituição de chave. Atacantes externos podem criar um novo par de chaves que validaria uma assinatura já existente sobre uma mensagem, ou criar um novo par de chaves e uma nova mensagem que validariam uma assinatura existente.
  + Não confie na unicidade das assinaturas. Primeiro, na maioria dos esquemas de assinatura, o signatário pode criar várias assinaturas distintas para a mesma mensagem. Segundo, a maioria dos esquemas de assinatura é maleável, o que significa que terceiros podem pegar uma assinatura e criar outra assinatura válida para a mesma mensagem.