**É isso?**

Criptografia de próxima geração

Comecei este livro com a ideia de que leitores que passassem pela maioria dos capítulos também estariam interessados no futuro da criptografia do mundo real. Embora você esteja lendo um livro aplicado e prático com foco no que está em uso hoje, o campo da criptografia está mudando rapidamente (como vimos nos últimos anos com as criptomoedas, por exemplo).

Enquanto você lê este livro, vários primitivos e protocolos criptográficos teóricos estão fazendo seu caminho para o mundo da criptografia aplicada — talvez porque esses primitivos teóricos estejam finalmente encontrando um caso de uso ou porque estejam finalmente se tornando eficientes o suficiente para serem usados em aplicações do mundo real. Seja qual for o motivo, o mundo real da criptografia está definitivamente crescendo e se tornando mais empolgante. Neste capítulo, dou a você uma amostra de como pode ser o futuro da criptografia do mundo real (talvez nos próximos 10 a 20 anos), apresentando brevemente três primitivos:

* Computação multipartidária segura (MPC) — Um subcampo da criptografia que permite que diferentes participantes executem um programa juntos sem necessariamente revelar suas próprias entradas para o programa.
* Criptografia totalmente homomórfica (FHE) — O santo graal da criptografia, um primitivo usado para permitir cálculos arbitrários em dados criptografados.
* Provas de conhecimento zero de uso geral (ZKPs) — O primitivo que você aprendeu no capítulo 7 que permite provar que você sabe algo sem revelar esse algo, mas desta vez aplicado de forma mais geral a programas mais complexos.

Este capítulo contém os conceitos mais avançados e complicados do livro. Por essa razão, recomendo que você o leia por alto e depois siga para o capítulo 16 para ler a conclusão. Quando estiver motivado a aprender mais sobre os aspectos internos desses conceitos mais avançados, volte a este capítulo. Vamos começar!

**15.1 Quanto mais, melhor: Computação multipartidária segura (MPC)**

A computação multipartidária segura (MPC) é um campo da criptografia que surgiu em 1982 com o famoso problema dos milionários. Em seu artigo de 1982 "Protocols for Secure Computations", Andrew C. Yao escreveu: "Dois milionários desejam saber quem é mais rico; contudo, não querem descobrir inadvertidamente nenhuma informação adicional sobre a riqueza um do outro. Como podem realizar tal conversa?" Simplificando, MPC é uma forma de múltiplos participantes computarem um programa juntos. Mas antes de aprender mais sobre esse novo primitivo, vejamos por que ele é útil.

Sabemos que, com a ajuda de uma terceira parte confiável, qualquer computação distribuída pode ser facilmente realizada. Essa terceira parte confiável pode talvez manter a privacidade da entrada de cada participante, bem como possivelmente restringir a quantidade de informação revelada pelo cálculo a participantes específicos. No mundo real, contudo, não gostamos muito de terceiros confiáveis; sabemos que são difíceis de encontrar e nem sempre cumprem seus compromissos.

MPC permite eliminar completamente terceiros confiáveis de uma computação distribuída e capacita os participantes a realizarem o cálculo por si mesmos sem revelar suas respectivas entradas uns aos outros. Isso é feito por meio de um protocolo criptográfico. Com isso em mente, usar MPC em um sistema é praticamente equivalente a usar uma terceira parte confiável (veja <IMAGEM>).

Observe que você já viu alguns protocolos de MPC. Assinaturas por limiar e gerações de chaves distribuídas, abordadas no capítulo 8, são exemplos de MPC. Mais especificamente, esses exemplos fazem parte de um subcampo de MPC chamado criptografia de limiar, que tem recebido muita atenção nos últimos anos, com, por exemplo, o NIST em meados de 2019 iniciando um processo de padronização para a criptografia de limiar.

**15.1.1 Interseção de conjuntos privados (PSI)**

Outro subcampo bem conhecido de MPC é o campo da interseção de conjuntos privados (PSI), que coloca o seguinte problema: Alice e Bob têm uma lista de palavras e querem saber quais palavras (ou talvez apenas quantas) têm em comum sem revelar suas respectivas listas de palavras. Uma maneira de resolver este problema é usar a construção de função pseudorrandômica oblívica (OPRF) que você aprendeu no capítulo 11. (Ilustro este protocolo na <IMAGEM>.)

Se você se lembra:

1. Bob gera uma chave para o OPRF.
2. Alice obtém os valores aleatórios, PRF(chave, palavra), para cada palavra em sua lista usando o protocolo OPRF (assim, ela não aprende a chave da PRF e Bob não aprende as palavras).
3. Bob então pode computar a lista de valores PRF(chave, palavra) para suas próprias palavras e enviá-la a Alice, que então é capaz de compará-la com suas próprias saídas de PRF para ver se alguma das saídas de PRF de Bob coincide.

PSI é um campo promissor que tem visto mais adoção nos últimos anos, pois tem se mostrado muito mais prático do que costumava ser. Por exemplo, o recurso Password Checkup do Google, integrado ao navegador Chrome, usa PSI para avisá-lo quando algumas de suas senhas foram detectadas em vazamentos de senhas após invasões, sem realmente ver suas senhas. Curiosamente, a Microsoft também faz isso em seu navegador Edge, mas usa criptografia totalmente homomórfica (que apresentarei na próxima seção) para realizar a interseção de conjuntos privados. Por outro lado, os desenvolvedores do aplicativo de mensagens Signal (discutido no capítulo 10) decidiram que o PSI era muito lento para realizar a descoberta de contatos com base na lista de contatos de seu telefone e, em vez disso, usaram SGX (abordado no capítulo 13) como uma terceira parte confiável.

**15.1.2 MPC de uso geral**

Mais geralmente, o MPC possui muitas soluções diferentes visando a computação de programas arbitrários. As soluções de MPC de uso geral fornecem diferentes níveis de eficiência (de horas a milissegundos) e tipos de propriedades. Por exemplo: quantos participantes desonestos o protocolo pode tolerar? Os participantes são maliciosos ou apenas honestos, mas curiosos (também chamados de semi-honestos, um tipo de participante em protocolos de MPC que está disposto a executar o protocolo corretamente, mas pode tentar aprender as entradas dos outros participantes)? É justo para todos os participantes caso alguns deles encerrem o protocolo prematuramente?

Antes que um programa possa ser computado com segurança por MPC, ele precisa ser traduzido em um circuito aritmético. Circuitos aritméticos são sucessões de adições e multiplicações e, como são Turing-completos, podem representar qualquer programa! Para uma ilustração de um circuito aritmético, veja <IMAGEM>.

Antes de prosseguirmos para o próximo primitivo, deixe-me dar um exemplo simplificado de um MPC de uso geral (com maioria honesta) construído via compartilhamento secreto de Shamir. Muitos outros esquemas existem, mas este é simples o suficiente para caber aqui em uma explicação de três etapas: compartilhar informações suficientes sobre cada entrada no circuito, avaliar cada porta no circuito e reconstruir a saída. Vamos ver cada etapa em mais detalhes.

A primeira etapa é que cada participante tenha informações suficientes sobre cada entrada do circuito. As entradas públicas são compartilhadas publicamente, enquanto as entradas privadas são compartilhadas via o compartilhamento secreto de Shamir (abordado no capítulo 8). Ilustro isso na <IMAGEM>.

A segunda etapa é avaliar cada porta do circuito. Por razões técnicas que omitirei aqui, portas de adição podem ser computadas localmente, enquanto portas de multiplicação devem ser computadas de forma interativa (os participantes devem trocar algumas mensagens). Para uma porta de adição, basta somar as partes de entrada que você possui; para uma porta de multiplicação, multiplique as partes de entrada. O que você obtém é uma parte do resultado, como ilustrado na <IMAGEM>. Neste ponto, as partes podem ser trocadas (para reconstruir a saída) ou mantidas separadas para continuar a computação (se representarem um valor intermediário).

A etapa final é reconstruir a saída. Neste ponto, os participantes devem possuir uma parte da saída, que podem usar para reconstruir o resultado final usando a etapa final do esquema de compartilhamento secreto de Shamir.

**15.1.3 O estado do MPC**

Houve um enorme progresso na última década para tornar o MPC prático. É um campo com muitos casos de uso diferentes, e devemos ficar atentos às aplicações potenciais que podem se beneficiar deste primitivo relativamente novo. Observe que, infelizmente, não há nenhum esforço real de padronização, e embora várias implementações de MPC possam ser consideradas práticas para muitos casos de uso hoje, elas não são fáceis de usar.

Incidentalmente, a construção de MPC de uso geral que expliquei anteriormente nesta seção é baseada em compartilhamento secreto, mas existem outras formas de construir protocolos de MPC. Uma alternativa bem conhecida é chamada de circuitos embaralhados (garbled circuits), um tipo de construção proposta por Yao em seu artigo de 1982 introduzindo o MPC. Outra alternativa é baseada em criptografia totalmente homomórfica, um primitivo que você aprenderá na próxima seção.

**15.2 Criptografia totalmente homomórfica (FHE) e as promessas de uma nuvem criptografada**

Por muito tempo na criptografia, uma pergunta intrigou muitos criptógrafos: é possível computar programas arbitrários em dados criptografados? Imagine que você pudesse criptografar os valores a, b e c separadamente, enviar os textos cifrados a um serviço e pedir que esse serviço retornasse a criptografia de a × 3b + 2c + 3, que você poderia então descriptografar. A ideia importante aqui é que o serviço nunca aprende seus valores e sempre lida com textos cifrados. Este cálculo pode não ser muito útil, mas com adições e multiplicações, pode-se computar programas reais em dados criptografados.

Este conceito interessante, originalmente proposto em 1978 por Rivest, Adleman e Dertouzos, é o que chamamos de criptografia totalmente homomórfica (FHE) (ou, como era chamada, o santo graal da criptografia). Ilustro este primitivo criptográfico na <IMAGEM>.

**15.2.1 Um exemplo de criptografia homomórfica com criptografia RSA**

Aliás, você já viu alguns esquemas criptográficos que devem fazê-lo sentir que sabe do que estou falando. Pense no RSA (abordado no capítulo 6): dado um  
cifrado = mensagemᵉ mod N, alguém pode facilmente computar alguma função restrita do  
texto cifrado:

nᵉ × cifrado = (n × mensagem)ᵉ mod N

para qualquer número n que quiser (embora ele não possa ser muito grande). O resultado é um texto cifrado que decriptografa para:

n × mensagem

Claro, este não é um comportamento desejado para o RSA, e levou a alguns ataques (por exemplo, o ataque de Bleichenbacher mencionado no capítulo 6). Na prática, o RSA quebra a propriedade homomórfica usando um esquema de preenchimento (padding). Observe que o RSA é homomórfico apenas para a multiplicação, o que não é suficiente para computar funções arbitrárias, pois tanto a multiplicação quanto a adição são necessárias para isso. Devido a essa limitação, dizemos que o RSA é parcialmente homomórfico.

**15.2.2 Os diferentes tipos de criptografia homomórfica**

Outros tipos de criptografia homomórfica incluem:

* **Parcialmente homomórfica** — o que significa homomórfica para uma operação (adição ou multiplicação) e homomórfica para a outra operação de forma limitada. Por exemplo, adições são ilimitadas até certo número, mas apenas algumas multiplicações podem ser feitas.
* **Homomórfica em níveis (leveled homomorphic)** — tanto adição quanto multiplicação são possíveis até um certo número de vezes.
* **Totalmente homomórfica (fully homomorphic)** — adição e multiplicação são ilimitadas (é o negócio de verdade).

Antes da invenção da FHE, vários tipos de esquemas de criptografia homomórfica foram propostos, mas nenhum conseguia alcançar o que a criptografia totalmente homomórfica prometia.  
A razão é que, ao avaliar circuitos em dados criptografados, um ruído cresce; após certo ponto, o ruído atinge um limite que torna a decriptação impossível. E, por muitos anos, alguns pesquisadores tentaram provar que talvez houvesse alguma teoria da informação que mostrasse que a criptografia totalmente homomórfica era impossível; isto é, até que foi mostrado que era possível.

**15.2.3 *Bootstrapping*, a chave para a criptografia totalmente homomórfica**

Numa noite, Alice sonha com imensas riquezas, cavernas empilhadas de prata, ouro e diamantes. Então, um dragão gigante devora as riquezas e começa a comer sua própria cauda! Ela acorda com uma sensação de paz. Ao tentar entender seu sonho, percebe que tem a solução para seu problema.

— Craig Gentry (“Computing Arbitrary Functions of Encrypted Data,” 2009)

Em 2009, Craig Gentry, um estudante de PhD de Dan Boneh, propôs a primeira construção de criptografia totalmente homomórfica. A solução de Gentry foi chamada de *bootstrapping*, que, na prática, consistia em avaliar um circuito de decriptação sobre o texto cifrado periodicamente para reduzir o ruído a um limite administrável. Interessantemente, o circuito de decriptação em si não revela a chave privada e pode ser computado pela parte não confiável. O *bootstrapping* permitiu transformar um esquema FHE em níveis em um esquema FHE completo. A construção de Gentry era lenta e bastante impraticável, levando cerca de 30 minutos por operação de bit básica, mas, como todo avanço, só melhorou com o tempo. Também mostrou que a criptografia totalmente homomórfica era possível.

Como o *bootstrapping* funciona? Vejamos se conseguimos obter alguma intuição. Primeiro, preciso mencionar que não usaremos um sistema de criptografia simétrica, mas um sistema de chave pública, onde uma chave pública pode ser usada para criptografar e uma chave privada pode ser usada para decriptografar. Agora, imagine que você executa um certo número de adições e multiplicações sobre um texto cifrado e atinge certo nível de ruído. O ruído é baixo o suficiente para ainda permitir a decriptação correta, mas alto demais para permitir mais operações homomórficas sem destruir o conteúdo criptografado. Ilustro isso em <IMAGEM>.

Você poderia pensar que está preso, mas o *bootstrapping* o liberta removendo o ruído desse texto cifrado. Para fazer isso, você recriptografa o texto cifrado ruidoso sob uma nova chave pública (normalmente chamada de chave de *bootstrapping*) para obter uma criptografia desse texto cifrado ruidoso. Observe que o novo texto cifrado não possui ruído. Ilustro isso em <IMAGEM>.

Agora vem a mágica: você recebe a chave privada inicial, não em texto claro, mas criptografada sob essa chave de *bootstrapping*. Isso significa que você pode usá-la com um circuito de decriptação para decriptar homomorficamente o texto cifrado ruidoso interno. Se o circuito de decriptação produzir uma quantidade aceitável de ruído, então funciona, e você terminará com o resultado da primeira operação homomórfica criptografado sob a chave de *bootstrapping*. Ilustro isso em <IMAGEM>.

Se a quantidade restante de erros permitir fazer pelo menos mais uma operação homomórfica (+ ou ×), então você está em boa forma: você tem um algoritmo de criptografia totalmente homomórfica porque sempre poderá, na prática, executar o *bootstrapping* após ou antes de cada operação. Observe que você pode definir o par de chaves de *bootstrapping* como sendo o mesmo que o par de chaves inicial. É um pouco estranho pois você obtém uma certa "segurança circular", mas parece funcionar e nenhum problema de segurança é conhecido.

**15.2.4 Um esquema FHE baseado no problema *learning with errors***

Antes de prosseguirmos, vejamos um exemplo de um esquema FHE baseado no problema *learning with errors* que vimos no capítulo 14. Vou explicar uma versão simplificada do esquema GSW, nomeado em homenagem aos autores Craig Gentry, Amit Sahai e Brent Waters. Para manter as coisas simples, introduzirei uma versão de chave secreta do algoritmo, mas tenha em mente que é relativamente simples transformar tal esquema em uma variante de chave pública, que precisamos para o *bootstrapping*. Observe a seguinte equação onde C é uma matriz quadrada, s é um vetor e m é um escalar (um número):

Cs = ms

Nesta equação, s é chamado de vetor próprio (*eigenvector*) e m de valor próprio (*eigenvalue*). Se essas palavras são estranhas para você, não se preocupe; elas não importam muito aqui.

A primeira intuição em nosso esquema FHE é obtida ao observar os vetores próprios e os valores próprios. A observação é que se definirmos m como um único bit que queremos criptografar, C como o texto cifrado e s como a chave secreta, então temos um esquema de criptografia homomórfica (inseguro) para criptografar um bit. (Claro, assumimos que há uma maneira de obter um texto cifrado aleatório C a partir de um bit fixo m e uma chave secreta fixa s). Ilustro isso de forma semelhante a um Lego na <IMAGEM>.

Para decriptografar um texto cifrado, você multiplica a matriz com o vetor secreto s e verifica se obtém o vetor secreto de volta ou 0. Você pode verificar que o esquema é totalmente homomórfico ao verificar que a decriptação de dois textos cifrados somados (C₁ + C₂) resulta na soma dos bits associados:

(C₁ + C₂)s = C₁s + C₂s = b₁s + b₂s = (b₁ + b₂)s

Também, a decriptação de dois textos cifrados multiplicados (C₁ × C₂) resulta na multiplicação dos bits associados:

(C₁ × C₂)s = C₁ (C₂s) = C₁ (b₂s) = b₂C₁s = (b₂ × b₁) s

Infelizmente, esse esquema é inseguro, pois é trivial recuperar o vetor próprio (o vetor secreto s) a partir de C. E se adicionássemos um pouco de ruído? Podemos modificar esta equação um pouco para que se assemelhe ao nosso problema de *learning with errors*:

Cs = ms + e

Isso deve parecer mais familiar. Novamente, podemos verificar que a adição ainda é homomórfica:

(C₁ + C₂)s = C₁s + C₂s = b₁s + e₁ + b₂s + e₂ = (b₁ + b₂)s + (e₁ + e₂)

Aqui, note que o erro está crescendo (e₁ + e₂), o que é o que esperávamos. Também podemos verificar que a multiplicação ainda está funcionando:

(C₁ × C₂)s = C₁ (C₂s) = C₁ (b₂s + e₂) = b₂C₁s + C₁e₂ = b₂ (b₁s + e₁) + C₁e₂  
= (b₂ × b₁) s + b₂e₁ + C₁e₂

Aqui, b₂e₁ é pequeno (pois é e₁ ou 0), mas C₁e₂ é potencialmente grande. Isso é obviamente um problema, que vou ignorar para não entrar em muitos detalhes. Se estiver interessado em aprender mais, não deixe de ler o relatório de Shai Halevi "Homomorphic Encryption" (2017), que faz um excelente trabalho explicando tudo isso e muito mais.

**15.2.5 Onde é usado?**

O caso de uso mais alardeado da FHE sempre foi a nuvem. E se eu pudesse continuar armazenando meus dados na nuvem sem que eles fossem vistos? E, adicionalmente, e se a nuvem pudesse realizar computações úteis sobre esses dados criptografados? De fato, pode-se pensar em muitas aplicações onde a FHE poderia ser útil. Alguns exemplos incluem:

* Um detector de spam poderia analisar seus e-mails sem visualizá-los.
* Pesquisas genéticas poderiam ser realizadas em seu DNA sem realmente ter que armazenar e proteger seu código humano sensível à privacidade.
* Um banco de dados poderia ser armazenado criptografado e consultado no servidor sem revelar nenhum dado.

Contudo, Phillip Rogaway, em seu artigo seminal de 2015 sobre “O Caráter Moral do Trabalho Criptográfico”, observa que:

“A FHE [...] gerou uma nova onda de exuberância. Em propostas de financiamento, entrevistas na mídia e palestras, teóricos de destaque falam da FHE [...] como indicações revolucionárias de onde chegamos. Ninguém parece enfatizar o quão especulativo é que qualquer uma dessas coisas tenha algum impacto na prática.”

Embora Rogaway não esteja errado, a FHE ainda é bastante lenta, mas os avanços no campo têm sido empolgantes. No momento em que escrevo (2021), as operações são cerca de um bilhão de vezes mais lentas que operações normais, ainda assim, desde 2009, houve um avanço de 10⁹ vezes na velocidade. Estamos indubitavelmente caminhando para um futuro onde a FHE será possível para pelo menos algumas aplicações limitadas.

Além disso, nem toda aplicação precisa do primitivo completo; a criptografia homomórfica parcial também pode ser usada em uma ampla gama de aplicações e é muito mais eficiente que a FHE. Um bom indicativo de que um primitivo criptográfico teórico está entrando no mundo real é a padronização, e de fato, a FHE não é estranha a isso. O esforço de padronização <https://homomorphicencryption.org> inclui muitas grandes empresas e universidades. Ainda não está claro exatamente quando, onde e em que forma a criptografia homomórfica fará sua entrada no mundo real. O que está claro é que isso acontecerá, então fique atento!

**15.3 Provas de conhecimento zero de uso geral (ZKPs)**

Falei sobre provas de conhecimento zero (ZKPs) no capítulo 7 sobre assinaturas. Lá, apontei que assinaturas são semelhantes a provas não interativas de conhecimento zero para logaritmos discretos. Esses tipos de ZKPs foram inventados em meados da década de 1980 pelos Professores Shafi Goldwasser, Silvio Micali e Charles Rackoff. Pouco depois, Goldreich, Micali e Wigderson descobriram que poderíamos provar muito mais do que apenas logaritmos discretos ou outros tipos de problemas difíceis; poderíamos também provar a execução correta de qualquer programa, mesmo se removêssemos algumas das entradas ou saídas (veja <IMAGEM> para um exemplo). Esta seção foca nesse tipo de ZKP de uso geral.

O campo de ZKP cresceu tremendamente desde seus primeiros anos. Uma das principais razões para esse crescimento é o boom das criptomoedas e a necessidade de prover mais confidencialidade às transações em blockchain, bem como otimizar o espaço. O campo de ZKP ainda está crescendo extremamente rápido no momento em que escrevo, e é bastante difícil acompanhar todos os esquemas modernos que existem e que tipos de ZKPs de uso geral há.

Felizmente para nós, o problema estava se tornando grande o suficiente para cruzar o limite de padronização, uma linha imaginária que, quando atingida, quase sempre motiva algumas pessoas a trabalharem juntas em prol de uma clarificação do campo. Em 2018, atores da indústria e da academia se uniram para formar o esforço de Padronização do ZKProof com o objetivo de "padronizar o uso de provas criptográficas de conhecimento zero." Até hoje, ainda é um esforço em andamento. Você pode ler mais em <https://zkproof.org>.

Você pode usar ZKPs de uso geral em diversas situações, mas até onde sei, eles têm sido principalmente utilizados no espaço das criptomoedas até agora, provavelmente devido ao alto número de pessoas interessadas em criptografia e dispostas a experimentar com as tecnologias mais inovadoras. Contudo, os ZKPs de uso geral têm aplicações potenciais em muitos campos: gerenciamento de identidade (ser capaz de provar sua idade sem revelá-la), compressão (ser capaz de ocultar a maior parte de um cálculo), confidencialidade (ser capaz de ocultar partes de um protocolo) e assim por diante. Os maiores obstáculos para que mais aplicações adotem ZKPs de uso geral parecem ser os seguintes:

* O grande número de esquemas de ZKP e o fato de que a cada ano mais esquemas estão sendo propostos.
* A dificuldade de compreender como esses sistemas funcionam e como usá-los para casos de uso específicos.

As distinções entre os diferentes esquemas propostos são bastante importantes. Como isso é uma grande fonte de confusão, aqui está como alguns desses esquemas são divididos:

* **Com ou sem conhecimento zero** — Se algumas das informações precisam permanecer secretas para alguns dos participantes, então precisamos de conhecimento zero. Observe que provas sem segredos também podem ser úteis. Por exemplo, você pode querer delegar um cálculo intensivo a um serviço que, por sua vez, deve provar que o resultado fornecido é correto.
* **Interativo ou não** — A maioria dos esquemas de ZKP pode ser tornada não interativa (às vezes usando a transformação de Fiat-Shamir de que falei no capítulo 7), e os projetistas de protocolos parecem mais interessados na versão não interativa do esquema. Isso porque a interatividade pode ser demorada nos protocolos e, às vezes, nem sequer é possível. Provas não interativas são frequentemente chamadas de NIZKs, para *non-interactive zero-knowledge proofs*.
* **Provas sucintas ou não** — A maioria dos esquemas de ZKP em evidência são frequentemente referidos como zk-SNARKs, para *Zero-Knowledge Succinct Non-Interactive Argument of Knowledge*. Enquanto a definição pode variar, ela foca no tamanho das provas produzidas por tais sistemas (geralmente na ordem de centenas de bytes) e no tempo necessário para verificá-las (na faixa de milissegundos). Assim, zk-SNARKs são curtas e fáceis de verificar. Note que o fato de um esquema não ser um zk-SNARK não o desqualifica para o mundo real, pois frequentemente diferentes propriedades podem ser úteis em diferentes casos de uso.
* **Configuração transparente ou não** — Como todo primitivo criptográfico, os ZKPs precisam de uma configuração para definir um conjunto de parâmetros e valores comuns. Isso é chamado de *common reference string* (CRS). Mas as configurações para ZKPs podem ser muito mais limitantes ou perigosas do que se pensa inicialmente. Existem três tipos de configuração:
  + **Confiável** — Significa que quem criou o CRS também tem acesso a segredos que lhes permitem forjar provas (por isso esses segredos às vezes são chamados de “lixo tóxico”). Este é um grande problema, pois voltamos a ter uma terceira parte confiável; ainda assim, esquemas que exibem essa propriedade são frequentemente os mais eficientes e têm os menores tamanhos de prova. Para reduzir o risco, o MPC pode ser usado para que muitos participantes ajudem a criar esses parâmetros perigosos. Se um único participante for honesto e apagar suas chaves após a cerimônia, o lixo tóxico é eliminado.
  + **Universal** — Uma configuração confiável é dita universal se pode ser usada para provar a execução de qualquer circuito (limitado por algum tamanho). Caso contrário, é específica para um único circuito.
  + **Transparente** — Felizmente para nós, muitos esquemas também oferecem configurações transparentes, significando que nenhuma terceira parte confiável precisa estar presente para criar os parâmetros do sistema. Esquemas transparentes são, por definição, universais.
* **Resistentes a computação quântica ou não** — Alguns ZKPs fazem uso de criptografia de chave pública e primitivos avançados como emparelhamentos bilineares (que explicarei depois), enquanto outros dependem apenas de criptografia simétrica (como funções de hash), o que os torna naturalmente resistentes a computadores quânticos (geralmente ao custo de provas muito maiores).

Como os zk-SNARKs são o que está em voga no momento em que escrevo, vou apresentar minha percepção de como eles funcionam.

**15.3.1 Como os zk-SNARKs funcionam**

Antes de tudo, há muitos, muitos esquemas de zk-SNARK — muitos mesmo. A maioria deles é construída com base neste tipo de construção:

* Um sistema de prova, permitindo que um provador prove algo a um verificador.
* Uma tradução ou compilação de um programa para algo que o sistema de prova pode provar.

A primeira parte não é tão difícil de entender, enquanto a segunda parte meio que exige um curso de pós-graduação no assunto. Para começar, vamos dar uma olhada na primeira parte.

A ideia principal dos zk-SNARKs é que tudo gira em torno de provar que você conhece algum polinômio f(x) que possui certas raízes. Por raízes, quero dizer que o verificador tem alguns valores em mente (por exemplo, 1 e 2), e o provador deve provar que o polinômio secreto que ele tem em mente avalia a 0 nesses valores (por exemplo, f(1) = f(2) = 0). Aliás, um polinômio que possui 1 e 2 como raízes (como no nosso exemplo) pode ser escrito como f(x) = (x – 1)(x – 2)h(x), para algum polinômio h(x). (Se não está convencido, tente avaliar isso em x = 1 e x = 2). Dizemos que o provador deve provar que conhece um f(x) e h(x) tal que f(x) = t(x)h(x), para algum polinômio alvo t(x) = (x – 1)(x – 2). Neste exemplo, 1 e 2 são as raízes que o verificador quer verificar.

Mas é isso! É isso que os sistemas de prova zk-SNARK geralmente fornecem: algo para provar que você conhece um polinômio. Estou repetindo isso porque, na primeira vez em que aprendi sobre isso, não fazia sentido para mim. Como você pode provar que conhece alguma entrada secreta de um programa se tudo o que pode provar é que conhece um polinômio? Bem, é por isso que a segunda parte de um zk-SNARK é tão difícil. É sobre traduzir um programa em um polinômio. Mas falaremos mais sobre isso depois.

Voltando ao nosso sistema de prova: como alguém prova que conhece tal função f(x)? É preciso provar que conhece um h(x) tal que possa escrever f(x) como f(x) = t(x)h(x). Ugh... não tão rápido aqui. Estamos falando de provas de conhecimento zero, certo? Como podemos provar isso sem revelar f(x)? A resposta está nos seguintes três truques:

* **Comprometimentos homomórficos** — Um esquema de comprometimento semelhante aos que usamos em outros ZKPs (abordados no capítulo 7).
* **Emparelhamentos bilineares** — Uma construção matemática com algumas propriedades interessantes; mais sobre isso depois.
* **O fato de que diferentes polinômios avaliam em diferentes valores na maioria das vezes**

Vamos passar por cada um desses, certo?

**15.3.2 Comprometimentos homomórficos para esconder partes da prova**

O primeiro truque é usar comprometimentos para ocultar os valores que estamos enviando ao verificador. Mas não apenas queremos escondê-los, também queremos permitir que o verificador realize algumas operações sobre eles para que possa verificar a prova. Especificamente, o verificador precisa verificar que, se o provador se comprometer com seu polinômio f(x) assim como com h(x), então temos:

com(f(x)) = com(t(x)) com(h(x)) = com(t(x)h(x))

onde o comprometimento com(t(x)) é computado pelo verificador como a restrição acordada sobre o polinômio. Essas operações são chamadas de operações homomórficas, e não poderíamos tê-las realizado se tivéssemos usado funções hash como mecanismos de comprometimento (como mencionado no capítulo 2). Graças a esses comprometimentos homomórficos, podemos “esconder valores no expoente” (por exemplo, para um valor v, enviar o comprometimento gᵛ mod p) e realizar verificações de identidade úteis:

* **Igualdade de comprometimentos** — A igualdade gᵃ = gᵇ significa que a = b.
* **Adição de comprometimentos** — A igualdade gᵃ = gᵇgᶜ significa que a = b + c.
* **Escalonamento de comprometimentos** — A igualdade gᵃ = (gᵇ)ᶜ significa que a = bc.

Observe que a última verificação só funciona se c for um valor público e não um comprometimento (gᶜ). Com apenas os comprometimentos homomórficos não podemos verificar a multiplicação dos comprometimentos, que é o que precisávamos. Felizmente, a criptografia tem outra ferramenta para obter tais equações escondidas no expoente — os emparelhamentos bilineares.

**15.3.3 Emparelhamentos bilineares para melhorar nossos comprometimentos homomórficos**

Os emparelhamentos bilineares podem ser usados para nos desbloquear, e esta é a única razão pela qual usamos emparelhamentos bilineares em um zk-SNARK (realmente, apenas para conseguir multiplicar os valores dentro dos comprometimentos). Não quero me aprofundar muito no que são emparelhamentos bilineares, mas basta saber que é outra ferramenta no nosso arsenal que permite multiplicar elementos que anteriormente não podiam ser multiplicados movendo-os de um grupo para outro.

Usando **e** como a notação típica de emparelhamento bilinear, temos e(g₁, g₂) = h₃, onde g₁, g₂ e h₃ são geradores de diferentes grupos. Aqui, usaremos o mesmo gerador à esquerda (g₁ = g₂), o que torna o emparelhamento simétrico. Podemos usar um emparelhamento bilinear para realizar multiplicações escondidas no expoente por meio da seguinte equação:

e(gᵇ, gᶜ) = e(g)ᵇᶜ

Novamente, usamos emparelhamentos bilineares para tornar nossos comprometimentos não apenas homomórficos para adição, mas também para multiplicação. (Observe que isso não é um esquema totalmente homomórfico, pois a multiplicação é limitada a uma única vez). Os emparelhamentos bilineares também são usados em outros lugares na criptografia e estão lentamente se tornando um bloco de construção mais comum. Eles podem ser vistos em esquemas de criptografia homomórfica e também em esquemas de assinatura como o BLS (que mencionei no capítulo 8).

**15.3.4 De onde vem a concisão (*succinctness*)?**

Finalmente, a concisão dos zk-SNARKs vem do fato de que duas funções que diferem avaliam em pontos diferentes na maioria das vezes. O que isso significa para nós? Digamos que eu não tenha um polinômio f(x) que realmente possua as raízes que escolhemos com o verificador; isso significa que f(x) não é igual a t(x)h(x). Então, avaliar f(x) e t(x)h(x) em um ponto aleatório r não retornará o mesmo resultado na maioria das vezes. Para quase todo r, f(r) ≠ t(r)h(r). Isso é conhecido como o lema de Schwartz-Zippel, o qual ilustro em <IMAGEM>.

Sabendo disso, basta provar que com(f(r)) = com(t(r)h(r)) para algum ponto aleatório r. É por isso que as provas zk-SNARK são tão pequenas: ao comparar pontos em um grupo, você acaba comparando polinômios muito maiores. Mas isso também é a razão por trás da configuração confiável necessária na maioria das construções zk-SNARK. Se um provador conhece o ponto aleatório r que será usado para verificar a igualdade, então pode forjar um polinômio inválido que ainda verificará a igualdade. Assim, uma configuração confiável envolve:

1. Criar um valor aleatório r.
2. Comprometer diferentes exponenciações de r (por exemplo, g, gʳ, gʳ², gʳ³, ...) para que esses valores possam ser usados pelo provador para computar seu polinômio sem conhecer o ponto r.
3. Destruir o valor r.

**15.3.5 De programas para polinômios**

Até agora, a restrição sobre o polinômio que o provador deve encontrar é que ele precisa ter algumas raízes: alguns valores que avaliam a 0 com nosso polinômio. Mas como transformamos uma afirmação mais geral em uma prova de conhecimento de polinômio? Afirmações típicas nas criptomoedas, que são as aplicações que atualmente mais usam zk-SNARKs hoje em dia, são do tipo:

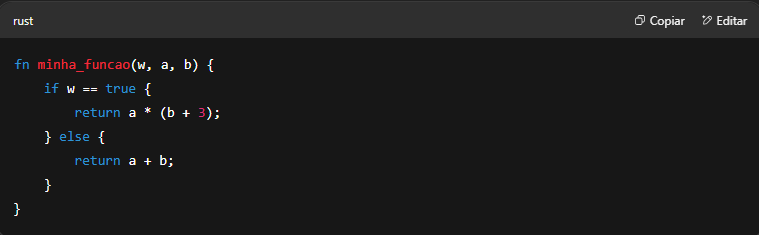
* Provar que um valor está no intervalo [0, 2⁶⁴] (isso é chamado de *range proof* — prova de intervalo)
* Provar que um valor (secreto) está incluído em uma determinada (pública) árvore de Merkle
* Provar que a soma de alguns valores é igual à soma de outros valores
* E assim por diante

E aqui está a parte difícil. Como eu disse anteriormente, converter a execução de um programa no conhecimento de um polinômio é realmente difícil. A boa notícia é que não vou explicar todos os detalhes, mas vou te dizer o suficiente para dar uma ideia de como as coisas funcionam. A partir daí, você deverá ser capaz de entender quais partes estão faltando na minha explicação e preenchê-las como desejar. O que vai acontecer a seguir é o seguinte:

1. Nosso programa será primeiro convertido em um circuito aritmético, como os que vimos na seção sobre MPC.
2. Esse circuito aritmético será convertido em um sistema de equações que estão em uma certa forma (chamada de sistema de restrições de posto 1, ou R1CS).
3. Em seguida, usaremos um truque para converter nosso sistema de equações em um polinômio.

**15.3.6 Programas são para computadores; nós precisamos de circuitos aritméticos**

Primeiro, vamos assumir que quase qualquer programa pode ser reescrito, mais ou menos facilmente, em matemática. O motivo pelo qual gostaríamos de fazer isso deve ser óbvio: não podemos provar código, mas podemos provar matemática. Por exemplo, o seguinte trecho de código apresenta uma função onde toda entrada é pública exceto **a**, que é nossa entrada secreta:



Neste exemplo simples, se todas as entradas e saídas forem públicas exceto **a**, ainda assim pode-se deduzir qual é **a**. Este exemplo também serve como um exemplo do que você não deveria tentar provar em conhecimento zero. De qualquer forma, o programa pode ser reescrito matematicamente com esta equação:

w × (a × (b + 3)) + (1 – w) × (a + b) = v

Onde **v** é a saída e **w** é 0 (falso) ou 1 (verdadeiro). Note que essa equação não é realmente um programa ou um circuito, ela apenas parece uma restrição. Se você executar o programa corretamente e depois preencher as entradas e saídas obtidas na equação, a igualdade deve ser verdadeira. Se a igualdade não for verdadeira, então suas entradas e saídas não correspondem a uma execução válida do programa.

É assim que você deve pensar sobre esses ZKPs de uso geral. Em vez de executar uma função em conhecimento zero (o que não significa muita coisa, na verdade), usamos zk-SNARKs para provar que alguns dados de entrada e saída correspondem corretamente à execução de um programa, mesmo quando algumas das entradas ou saídas são omitidas.

**15.3.7 De um circuito aritmético para um sistema de restrições de posto 1 (R1CS)**

De qualquer modo, estamos apenas um passo no processo de conversão de nossa execução em algo que possamos provar com zk-SNARKs. O próximo passo é converter isso em uma série de restrições, que então podem ser convertidas em uma prova de conhecimento de polinômio. O que os zk-SNARKs querem é um *rank-1 constraint system* (R1CS). Um R1CS é simplesmente uma série de equações da forma L × R = O, onde L, R e O podem ser apenas somas de algumas variáveis, assim a única multiplicação é entre L e R. Não importa muito por que precisamos transformar nosso circuito aritmético em tal sistema de equações, exceto que isso ajuda na conversão para o polinômio final que podemos provar. Se tentarmos fazer isso com a equação que temos, obtemos algo como:

* a × (b + 3) = m
* w × (m – a – b) = v – a – b

Na verdade, esquecemos da restrição de que **w** é 0 ou 1, o que podemos adicionar ao nosso sistema com um truque inteligente:

* a × (b + 3) = m
* w × (m – a – b) = v – a – b
* w × w = w

Faz sentido? Você deve realmente ver esse sistema como um conjunto de restrições: se você me der um conjunto de valores que afirma corresponder às entradas e saídas da execução do meu programa, então eu deveria ser capaz de validar que os valores também verificam corretamente as igualdades. Se alguma das igualdades estiver errada, então significa que o programa não produziu o valor de saída que você forneceu para as entradas que me deu. Outra maneira de pensar nisso é que os zk-SNARKs permitem que você remova de forma verificável entradas ou saídas do histórico da execução correta de um programa.

**15.3.8 De R1CS para um polinômio**

A pergunta ainda é: como transformamos este sistema em um polinômio? Estamos quase lá, e como sempre, a resposta vem com uma série de truques! Como temos três equações diferentes em nosso sistema, o primeiro passo é concordar com três raízes para o nosso polinômio. Podemos simplesmente escolher 1, 2, 3 como raízes, significando que nosso polinômio resolve f(x) = 0 para x = 1, x = 2 e x = 3. Por que fazer isso? Fazendo assim, podemos fazer com que nosso polinômio represente todas as equações em nosso sistema simultaneamente, representando a primeira equação quando avaliada em 1, a segunda equação quando avaliada em 2, e assim por diante. O trabalho do provador agora é criar um polinômio f(x) tal que:

* f(1) = a × (b + 3) – m
* f(2) = w × (m – a – b) – (v – a – b)
* f(3) = w × w – w

Observe que todas essas equações devem avaliar a 0 se os valores corresponderem corretamente à execução do nosso programa original. Em outras palavras, nosso polinômio f(x) tem raízes 1, 2, 3 somente se o criarmos corretamente. Lembre-se, é disso que se trata os zk-SNARKs: temos o protocolo para provar que, de fato, nosso polinômio f(x) tem essas raízes (conhecidas por ambos, provador e verificador).

Seria muito simples se isso fosse o fim da minha explicação porque agora o problema é que o provador tem liberdade demais para escolher seu polinômio f(x). Ele pode simplesmente encontrar um polinômio que tenha as raízes 1, 2, 3 sem se importar com os valores a, b, m, v e w. Ele pode basicamente fazer o que quiser! O que queremos, em vez disso, é um sistema que trave cada parte do polinômio, exceto pelos valores secretos que o verificador não deve conhecer.

**15.3.9 São necessários dois para avaliar um polinômio escondido no expoente**

Recapitulando: queremos um provador que tenha que executar corretamente o programa com seu valor secreto **a** e os valores públicos **b** e **w** e obter a saída **v** que ele pode publicar. O provador então deve criar um polinômio preenchendo apenas as partes que o verificador não deve conhecer: os valores **a** e **m**. Assim, em um protocolo real de zk-SNARK, queremos que o provador tenha a menor liberdade possível ao criar seus polinômios e, em seguida, avaliá-los em um ponto aleatório.

Para fazer isso, o polinômio é criado de forma um pouco dinâmica: o provador apenas preenche sua parte, depois o verificador preenche as outras partes. Por exemplo, vamos pegar a primeira equação, f(1) = a × (b + 3) – m, e representá-la como:

f₁(x) = a L₁(x) × (b + 3) R₁(x) – m O₁(x)

onde L₁(x), R₁(x), O₁(x) são polinômios que avaliam para 1 em x = 1 e para 0 em x = 2 e x = 3. Isso é necessário para que influenciem apenas nossa primeira equação. (Note que é fácil encontrar tais polinômios via algoritmos como a interpolação de Lagrange). Agora, observe mais duas coisas:

* Temos as entradas, valores intermediários e saídas como coeficientes de nossos polinômios.
* O polinômio f(x) é a soma f₁(x) + f₂(x) + f₃(x), onde podemos definir f₂(x) e f₃(x) para representar as equações 2 e 3, de forma semelhante a f₁(x).

Como você pode ver, nossa primeira equação ainda é representada no ponto x = 1:

f(1) = f₁(1) + f₂(1) + f₃(1)  
= f₁(1)  
= a L₁(1) × (b + 3) R₁(1) – m O₁(1)  
= a × (b + 3) – m

Com essa nova forma de representar nossas equações (que, lembre-se, representam a execução do nosso programa), o provador agora pode avaliar as partes do polinômio que lhe são relevantes:

1. Tomando a exponenciação do ponto aleatório **r** escondido no expoente para reconstruir os polinômios L₁(r) e O₁(r).
2. Exponenciando gᴸ¹⁽ʳ⁾ com o valor secreto **a** para obter (gᴸ¹⁽ʳ⁾)ᵃ = gᵃᴸ¹⁽ʳ⁾, o que representa a avaliação de a × L₁(x) no ponto aleatório desconhecido x = r e escondido no expoente.
3. Exponenciando gᴼ¹⁽ʳ⁾ com o valor intermediário secreto **m** para obter (gᴼ¹⁽ʳ⁾)ᵐ = gᵐᴼ¹⁽ʳ⁾, o que representa a avaliação de m O₁(x) no ponto aleatório r e escondido no expoente.

O verificador pode então preencher as partes restantes reconstruindo (gᴿ¹⁽ʳ⁾)ᵇ e (gᴿ⁰⁽ʳ⁾)³ para algum valor acordado **b** com as mesmas técnicas que o provador usou. Somando os dois juntos, o verificador obtém gᵇᴿ¹⁽ʳ⁾ + g³ᴿ¹⁽ʳ⁾, que representa a avaliação (escondida) de (b + 3) × R₁(x) no ponto aleatório desconhecido x = r. Finalmente, o verificador pode reconstruir f₁(r), que está escondido no expoente, usando um emparelhamento bilinear:

e(gᵃᴸ¹⁽ʳ⁾, g⁽ᵇ⁺³⁾ᴿ¹⁽ʳ⁾) – e(g, gᵐᴼ¹⁽ʳ⁾) = e(g, g)ᵃᴸ¹⁽ʳ⁾ × ⁽ᵇ⁺³⁾ᴿ¹⁽ʳ⁾ – ᵐᴼ¹⁽ʳ⁾

Se você extrapolar essas técnicas para todo o polinômio f(x), poderá entender o protocolo final. Claro, isso ainda é uma simplificação grosseira de um protocolo real de zk-SNARK; isso ainda deixa liberdade demais ao provador.

Todos os outros truques usados nos zk-SNARKs servem para restringir ainda mais o que o provador pode fazer, garantindo que ele preencha corretamente e de forma consistente as partes faltantes, bem como para otimizar o que pode ser otimizado. Aliás, a melhor explicação que já li é o artigo “Why and How zk-SNARK Works: Definitive Explanation” de Maksym Petkus, que vai muito mais a fundo e explica todas as partes que deixei de lado.

E isso é tudo para os zk-SNARKs. Isto é realmente apenas uma introdução; na prática, os zk-SNARKs são muito mais complicados de entender e usar! Não só o trabalho de converter um programa em algo que pode ser provado é não trivial, como às vezes impõe novas restrições a um protocolo criptográfico. Por exemplo, as funções hash e esquemas de assinatura mais comuns são frequentemente pesados demais para sistemas de ZKP de uso geral, o que levou muitos projetistas de protocolo a investigar esquemas mais amigáveis a ZKP. Além disso, como eu disse anteriormente, existem muitas construções diferentes de zk-SNARKs, e também muitas construções de não-zk-SNARKs, que podem ser mais relevantes como construções de ZKP de uso geral, dependendo do seu caso de uso.

Mas, infelizmente, não parece existir um esquema de ZKP universal (por exemplo, um esquema com configuração transparente, sucinto, universal e resistente a quântica), e não está claro qual usar em quais casos. O campo ainda é jovem, e a cada ano novos e melhores esquemas estão sendo publicados. Pode ser que, daqui a alguns anos, melhores padrões e bibliotecas mais fáceis de usar surjam, então, se você tem interesse nessa área, continue acompanhando!

**Resumo**

* Na última década, muitos primitivos criptográficos teóricos fizeram enormes progressos em termos de eficiência e praticidade; alguns estão fazendo seu caminho para o mundo real.
* Computação multipartidária segura (MPC) é um primitivo que permite a múltiplos participantes executarem corretamente um programa juntos, sem revelar suas respectivas entradas. Assinaturas de limiar estão começando a ser adotadas em criptomoedas, enquanto protocolos de interseção de conjuntos privados (PSI) estão sendo usados em protocolos modernos e em larga escala como o Password Checkup do Google.
* Criptografia totalmente homomórfica (FHE) permite computar funções arbitrárias em dados criptografados sem decriptá-los. Tem aplicações potenciais na nuvem, onde poderia impedir o acesso aos dados por qualquer um exceto o próprio usuário, ao mesmo tempo em que permitiria à plataforma de nuvem realizar computações úteis sobre os dados para o usuário.
* Provas de conhecimento zero de uso geral (ZKPs) encontraram muitos casos de uso, e tiveram avanços recentes com provas pequenas e rápidas de verificar. São usadas principalmente em criptomoedas para adicionar privacidade ou comprimir o tamanho do *blockchain*. Seus casos de uso parecem mais amplos, entretanto, e à medida que melhores padrões e bibliotecas mais fáceis de usar fizerem seu caminho para o mundo real, poderemos vê-las sendo usadas cada vez mais.