Zero a Monero : Segunda Edição

Um Guía Técnico para uma moeda digital privada; para íniciados, amadores e especialistas .

Publicado a 20 de maio de 2023 (v2.0.0)

koe[1], Kurt M. Alonso[2], Sarang Noether[3]

Traduzido por `slave_blocker` [4]

[1] ukoe@protonmail.com
[2] kurt@oktav.se
[3] sarang.noether@protonmail.com
[4] dollner@gmx.de

Abstracto

Criptografia.[a] Pode parecer que somente matemáticos e cientistas da computação tenham acesso a este obscuro, esotérico, poderoso e elegante tópico. De facto, muitos tipos de criptografia são simples o suficiente, tal que qualquer leitor possa aprender esses conceitos fundamentais.

É conhecimento geral que a criptografia é usada para manter comunicações seguras, sejam essas interações cartas codificadas ou interacções digitais privadas. Uma das aplicações chama-se cripto-moeda. Essa moeda digital usa criptografia para transferir ou definir posse de fundos. Para garantir que a transferencia de fundos nao seja duplicada, ou que a criação de moedas não seja arbítrária, as cripto-moedas fazem uso de assim chamadas 'blockchains'. Ou seja um livro público e distribuído que mantem a contabilidade das transacções efectuadas, e que pode ser verificado por terceiros [**?**].

Á primeira vista assume-se talvez que estas transacções tenham que ser enviadas e guardadas num formato de texto simples e nao codificado tal que a verificação pública seja possível. De facto é possível não só ocultar os participantes envolvidos numa transacção como também o montante transferido. E ao mesmo tempo garantir que entidades terceiras sejam aptas de verificar e concordar nessas transaccões efectuadas [**?**]. Isto é exemplificado na cripto-moeda Monero.

O empenho aqui feito é de ensinar a qualquer leitor que saiba algebra linear básica e simples ciencia da computação não só como Monero funciona a um nível profundo e detalhado, e também mostrar como a criptografia pode ser práctica e elegante.

Para os leitores experientes: Monero é uma cripto-moeda numa blockchain; grafo standard uni-dimensional distribuído e acíclico [**?**]. Em que as transacções são baseadas em criptografia de curva elíptica. A curva usada é a Ed25519 [**?**]. Entradas de transacção são assinados com Assinaturas espontâneas anónimas ligadas de grupo com múltiplas camadas (MLSAG) [**?**], e montantes nas saídas são ocultados com compromissos de pedersen [**?**] e prova-se que estes estão num domínio legítimo com uma prova de domínio *Bulletproof* [**?**]. Grande parte da primeira metade deste relatório esplica estas ideias.

[a] Este documento é um trabalho em progresso, a parte II foi traduzida á letra sem expôr o conteúdo de forma verbatim. Faltam 2 capítulos : *RandomX* e *Bulletproof*

Conteúdo

CAPÍTULO 1

Introdução

Em termos digitais é trivial fazer inúmeras cópias de informação, e também com alterações arbitrárias. Para que uma moeda exista digitalmente e para que esta seja adoptada, os seus utilizadores têm de saber que a quantidade monetária total é limitada. Um recepiente de dinheiro têm de poder verificar que não está a receber moedas contrafeitas, ou moedas que já foram enviadas a outro destinatário. Para alcancar isto, sem requerer a colaboração de alguma entidade terceira central, a quantidade monetária e a completa história de transacções tem de ser públicamente verificável.

Cripto-moedas guardam as transacções na lista de blocos. Esta lista de blocos é imutável e não falsificável com uma legítimidade que não pode ser disputada. A maioria das cripto-moedas guardam as transacções em texto claro e não encriptado, para facilitar a verificação das mesmas pela comunidade de utilizadores. Claramente uma lista de blocos desta natureza desafia qualquer conceito básico de privacidade ou de fungíbilidade. Visto que as transacções de qualquer pessoa estão literalmente abertas para que o público as veja [1] .
Para evitar a falta de privacidade, utilizadores de cripto-moedas como Bitcoin podem obfuscar transacções ao utilizar endereços intermediáros [**?**]. Mesmo assim com certas ferramentas é possivel analizar e muitas vezes ligar remetentes com destinatários [**?**, **?**, **?**, **?**].

Em contraste, a cripto-moeda Monero (Moe-neh-row), utiliza endereços ocultos para receber montantes e assinaturas em anel para enviar montantes na lista de blocos. Com estes methodos não

[1] **Fungível** significa capaz de mútua substituição no uso ou satisfação de um contrato. Exemplos : ... Dinheiro, etc [**?**]. Numa lista de blocos aberta como em Bitcoin, as moedas possuídas pela alice podem ser diferenciadas daquelas possuídas pelo bob, com base no histórico de transacções dessas moedas. Se o histórico de transacções incluí transacções relacionadas com actores supostamente nefastos, então essas moedas podem estar manchadas [**?**]. Como tal, tais moedas tornam-se menos valiosas do que outras, para mais diz que Bitcoins acabados de minerar (vírgens), são mais caros pois não têm histórico [**?**].

existem caminhos conhecidos para desvendar a ligação entre remetentes e destinatários. Adicionalmente os montantes em sí são escondidos por construções criptográficas, o que torno o fluxo de montantes opáco.
O resultado é uma cripto-moeda com um elevado grau de privacidade e fungibilidade.

[2]

1.1 Objectivos

Monero é uma cripto-moeda com mais de cinco anos de desenvolvimento [**?**, **?**], e mantêm um nível crescente de adopção [**?**]. [3]

[4] [5] Ainda pior, partes esênciais da estrutura teorética de Monero têm sido publicados em artigos não revisados por profissionais na matéria. Como tal só o código fonte núcleo, é fidedigno como fonte de informação. [6]

Para mais, para aqueles sem experiência em matemática, aprender os básicos de criptográfia de curva elíptica, que Monero utiliza de forma extensiva, pode ser difícil e frustrante. [7] Tenta-se introduzir os conceitos fundamentais necessários para compreender criptografia de curva elíptica, rever algoritmos e esquemas criptográficos e coleccionar informações detalhadas sobre o funcionamento interno de Monero. Para oferecer a melhor experiência aos nossos leitores, contruiu-se uma descrição passo-a-passo da cripto-moeda Monero. Na segunda edição deste relatório a atenção foi focada na versão 12 do protocolo de Monero o que corresponde á versão 0.15.x.x do pacote de software de Monero. Todos os mecanismos, relacionados ás transacções e a lista de blocos, descri-

[2] Dependendo do comportamento dos utilizadores, podem haver casos em que as transacções são analisáveis até um certo grau. Para um exemplo veja-se este artigo : [**?**].

[3] Em termos de capitalização de mercado, Monero têm-se mantido firme em comparação com outras criptomoedas. Foi número 14 em Junho de 2018, e número 12 no quinto de Janeiro de 2020; veja-se https://coinmarketcap.com/.

[4] Um esforço de documentação, em https://monerodocs.org/, tem umas entradas de auxílio relacionadas com a interface da linha de comandos. A *cli* é uma carteira de monero acessível através de um terminal. É a carteira de monero com o maior número de funcionalidades, ao custo de não ser tão fácil de usar como outras carteiras que benificiam de uma superfície gráfica.

[5] Uma outra documentação mais geral chamada *Mastering Monero* pode ser encontrada aqui : [**?**].

[6] Sr. Seguias crio a série excelente *Monero Building Blocks* [**?**], que contêm as provas de segurança criptográficas que justificam os esquemas de assinaturas em Monero. Bem como zero a Monero, primeira edição [**?**], a série de Seguias baseia-se na versão n° 7 do protocolo.

[7] Uma tentativa prévia de esplicar como Monero funciona [**?**] não elucidou criptografia de curva elíptica, estava incompleta, e está agora obsoleta

tos aqui pertencem a estas versões. [8] [9] [10] Esquemas de transacções descontinuados não foram explorados, mesmo que estes ainda sejam parcialmente supportados por causa da compatibilidade com versões anteriores. O mesmo para características descontinuadas da lista de blocos. A primeira edição [**?**] corresponde á versão 7 do protocolo, e á versão 0.12.x.x do pacote de software de Monero.

1.2 Aos leitores

Antecipa-se que muitos leitores irão encontrar este relatório com nenhum a pouco conhecimento de matemática discreta, estruturas algebraícas, criptografia e listas de blocos. Tentou-se ser o mais detalhado possível para que mesmo leigos de todas as perspectivas possam aprender Monero sem precisar de pesquisa externa. [11]

Omitiu-se intencionalmente, ou foi delegado para notas de rodapé, alguns detalhes matemáticos, que tornariam o texto menos claro. Foram também omitidos detalhes concretos de implementação quando estão não pareciam ser essenciais. O objectivo aqui é de apresentar a matéria a meio caminho entre a criptografia e a programação, de forma completa e clara. [12]

1.3 Origens da cripto-moeda Monero

A cripto-moeda Monero, initialmente conhecida como BitMonero, foi criada em abril de 2014 e deriva da prova de conceito de *CryptoNote* [**?**]. Monero significa *moeda* na língua Esperanto, e o plural é Moneroj. A cripto-moeda CryptoNote foi desenvolvida por vários indivíduos. O primeiro abstracto académico que a descreve foi públicado debaixo do pseudónimo de *Nicolas van Saberhagen* em Outubro de 2013 [**?**]. Nessa altura a anônimidade do destinatário foi garantida com endereços ocultos, e a ambuiguidade do remetente com assinaturas em anel. Desde a sua incepção, Monero fortaleceu a sua privacidade ainda mais, ao implementar montantes ocultos, como descrito por Greg Maxwell (entre outros) em [**?**] e integrado em assinaturas de anel baseado em

[8] O 'protocolo' é o conjunto de regras que cada novo bloco tem de seguit antes de ser adicionado á lista de blocos. Este conjunto de regras incluí 'o protocolo de transacção' (actualmente na versão 2, RingCT), que são regras gerais que definem como uma transacção é construída. Regras específicas para as transacções podem mudar e mudam de facto. Sem que a versão do protocolo de transacção mude. Só mudanças em grande escala da estrutura de transacção permitem mover o número da versão.

[9] A integridade do código fonte em Monero parte do princípio de que bastantes pessoas o leram e que todos os erros, ou pelo menos os mais relevantes foram removidos. Espera-se que os nossos leitores não tomem as nossas esplicações como garantidas, e que verifiquem por eles próprios que o código fonte faz, o que deve fazer. Se não, espera-se que o leitor faça uma divulgação responsável. Em que se usa (`https://hackerone.com/monero`) para os maiores problemas, e um *pull request* no github em (`https://github.com/monero-project/monero`) para detalhes.

[10] Diversos protocolos que valem a pena serem considerados para a próxima geração das transacções em Monero estão a ser investigadas e pesquisadas. Estes incluem Triptych [**?**], RingCT3.0 [**?**], Omniring [**?**], e Lelantus [**?**].

[11] Um livro extensivo em criptografia aplicada pode ser encontrado aqui : [**?**].

[12] Algumas notas de rodapé, especialmente nos capítulos relacionados com o protocolo, estragam os capítulos ou secções futuras. A intenção destas, é de fazerem mais sentido numa segunda leitura, pois estas involvem detalhes de implementação que usualmente só fazem sentido para aqueles que já compreendem como Monero funciona.

recomendações por Shen Noether [**?**]. O que se tornou depois ainda mais eficiente com *Bulletproofs* [**?**].

1.3.1 Parte 1: 'Essenciais'

Nesta saga de conhecimento, são apresentados os elementos básicos de criptografia necessários para perceber as complexidades de Monero.
No capítulo 2 são desenvolvidos os aspectos essentiais de criptografia de curva elíptica.
No capítulo 3 expande-se o esquema de assinaturas tipo Schnorr do capítulo anterior. E esplica-se os algoritmos de assinaturas em anel que são aplicados para alcançar transacções confidenciais.

No capítulo 4 esplora-se como Monero utiliza endereços para controlar a posse de montantes, e os diferentes tipos de endereços.

No capítulo 5 introduzimos os mecanismos criptográficos usados para ocultar montantes.

Com todos os componentes dados, esplica-se o esquema de transacções usado em Monero no capítulo 6.

A lista de blocos de Monero é apresentada no capítulo 7.

1.3.2 Part 2: 'Extenções'

Uma cripto-moeda é mais do que o próprio protocolo, e em 'extensões' fala-se sobre um número de ideias diferentes, muitas das quais ainda não foram implementadas. [13] Várias informações sobre uma transacção pode ser provadas a observadores, e esses methodos são o conteúdo do capítulo 8. Enquanto não essencial para a operação de Monero, existe muita utilidade em multi-assinaturas o que permite múltiplas pessoas enviar e receber fundos colaborativamente. O capítulo 9 descreve o esquema actual de multi-assinaturas de Monero, e também os seus futuros possíveis desenvolvimentos. O capítulo 10 constitui a architectura de mercados de garantia online com multi-assinaturas. Primeiro apresentado aqui, TxTangle, descrito no capítulo 11, é um protocolo decentralizado para reunir transacções de múltiplos individuos para uma só transacção.

1.3.3 Conteúdo adicional

O apéndice A explica a estrutura de uma transacção exemplo da lista de blocos. O apéndice B esplica a estrutura de um bloco (incluindo o cabeçalho e as transacções de mineiro). Finalmente o apéndice C, fecha este relatório ao esplicar a estrutura do bloco génese de Monero. Estes apéndices oferecem uma ligação entre os elementos teóricos descritos nas secções anteriores com a sua implementação na vida real.

[13] Note-se que as versões futuras do protocolo em Monero, especialmente aquelas que mudam a estrutura das transacções, podem fazer estas ideias impossíveis ou impraticáveis.

São usadas notas de margem , para indicar onde é que detalhes de implementação existem dentro do código fonte. [14] Usualmente existe um directório, como por exemplo : src/ringct/rctOps.cpp . E também uma função como por exemplo : `ecdhEncode()`. Note-se : '-' indica texto dividido, como por exemplo : crypto- note → cryptonote.
Normalmente os qualificadores de *namespace* não são incluídos (e.g. `Blockchain::`). isto é útil !

1.4 Aviso

Todos os esquemas de assinatura, aplicações de curva elíptica, e detalhes de implementação devem ser considerados somente descriptivos. Leitores que consideram aplicações sérias, ao invés de explorações para extender o conhecimento, devem consultar fontes primárias e especificações técnicas (que são citadas sempre que possível). Esquemas de assinatura precisam provas com uma segurança bem definida, e detalhes de implementação podem ser encontrados no código fonte de Monero.
Em particular, como se diz em inglés : 'don't roll your own crypto' . O que significa : 'não divulgues a tua própria cripto-moeda.' Código que implemente primitivas criptográficas deve ser revisto por especialistas, e deve ter um longo histórico de desempenho confiável.
Finalmente contribuições originais para este documento podem não ser bem revistas e provávelmente não são testadas, tanto que, leitores devem exercitar cautela.

1.5 A História de Zero a Monero

De zero a Monero é uma expansão da tese de mestrado de Kurt Alonso 'Monero - Privacy in the Blockchain' [**?**], publicado em maio de 2018. A primeira edição foi publicada em junho de 2018 [**?**]. Na segunda edição, melhorou-se a forma de apresentar assinaturas em anel (Chapter 3), reorganizou-se como as transacções são esplicadas (adicionou-se o capítulo 4 em endereços Monero). Modernizou-se o méthodo usado para comunicar montantes de saídas de transacção (secção 5.3). Assinaturas em anel *Borromean* são substituídas com *Bulletproofs* . Descontinou-se `RCTTypeFull` (capítulo 6). Renovou-se o peso dinámico de bloco e o sistema de taxa dos mineiros (capítulo 7). Investigaram-se provas relacionadas com transacções (capítulo 8). Descreveram-se multi-assinaturas de Monero (capítulo 9). Desenvolveram-se soluções para mercados com garantia (capítulo 10). Foi proposto um novo protocolo de transacções chamado TxTangle (capítulo 11). Adicionaram-se vários detalhes para alignar com o protocolo actual (v12) e o pacote de software Monero (v0.15.x.x). E poliu-se o documento inteiro para aumentar a qualidade de leitura. [15]

[14] As notas de margem são precisas para a versão 0.15.x.x do pacote de software de Monero. mas podem se tornar gradualmente imprecisos á medida que o código base muda. O código fonte está guardado num repositório de *git* (`https://github.com/monero-project/monero`), portanto uma história completa de alterações está disponível

[15] O código fonte LaTeX de ambas as edições de zero a Monero pode ser encontrado aqui (a primeira edição está no ramo 'ztm1'): `https://github.com/UkoeHB/Monero-RCT-report`.

1.6 Reconhecimentos

Escrito pelo autor ‘koe’. Este relatório não existiria sem a these de mestrado de Kurt Alonso [**?**], portanto a ele tenho uma enorme dívida de gratidão. O pesquisador Brandon “Surae Noether”Goodell e o pseudónimo ‘Sarang Noether’ do laboratório de pesquisas de Monero (MRL), têm sido fontes de conhecimento confiáveis ao longo do desenvolvimento de ambas as edições de zero a Monero.
O pseudónimo ‘moneromooo’, o programador mais prolífico do código fonte do projecto de Monero, tem provavelmente o conhecimento mais extensivo do código neste planeta, e deu-me númeras vezes as direcções indicadas. E claro outros contribuidores magníficos de Monero que gastaram tempo a esplicarem-me as minhas inúmeras questões. E finalmente os vários leitores que nos contactaram com comentários encoragedores, obrigado !

Parte I

Essenciais

CAPÍTULO 2

Conceitos Básicos

2.1 Algumas palavras sobre a notação

Um dos príncipais objectivos neste texto é de reunir, corrigir, rever e homogenizar toda a informação existente relacionada com os mecanismos internos da cripto-moeda Monero. E ao mesmo tempo oferecer os detalhes suficientes, para apresentar a matéria de uma forma única e construtiva.

Entre outras importantes convenções, o seguinte foi usado :

- letra minúscula para inteiros, representações de bits, strings e valores simples.
- letra maiúscula para pontos em curvas elípticas e ou valores complexos.

Para itens com um significado especial, tentámos, o mais possível, usar sempre os mesmos símbolos. Por exemplo um gerador de curva e sempre denotado como G, e a sua ordem é l. Chaves privadas/públicas são denotadas usualmente por k/K respectivamente.

Para além disso tentámos ser *conceptuais* na nossa presentação de algoritmos e esquemas. Um leitor com experiência em ciência da computação pode sentir que não foram esplicadas questões como a representação de bits de alguns itens, ou como executar certas operações. Para mais estudantes de matemática podem considerar que não foram dadas esplicações de algebra abstracta.

Contudo, isto não é visto como uma perda. Um simples objecto como um inteiro ou uma string (lista de characteres) pode sempre ser representado como uma lista de bits. A assim chamada 'endianness' raramente é relevante, e é maioritariamente uma convenção nos nossos algoritmos.

Pontos de curva elíptica são normalmente denotados por pares (x, y), e podem como tal ser representados por dois inteiros. Contudo, no mundo da criptografia é comum aplicar técnicas de compressão nos pontos. O que permite representar um ponto somente através de uma coordinada. Ao abordar estes conceitos tal informação torna-se secundária, neste case assume-se no entanto compressão de pontos de curva elíptica.

As funções hash são aqui também referidas de forma livre e sem declarar algoritmos específicos. No caso de Monero trata-se de uma variante de *Keccak*, mas como não é relevante para a teoria nao será explicitamente definido[1]. src/crypto/keccak.c

Uma função de dispersão criptográfica ou função hash criptográfica, será daqui em diante simplesmente 'função hash' ou também 'hash'. Tais funções têm como entrada uma mensagem `m` de comprimento variável, e devolvem como resultado uma hash h de comprimento fixo. Para qualquer entrada, qualquer resultado é equiprovável. Funções hash são difíceis de inverter, possuem uma característica interessante conhecida como o *efeito avalanche* o que pode causar que duas entradas muito semelhantes produzam hashes muito diferentes e é também difícil encontrar duas mensagem de entrada que resultem na mesma hash.

Funções hash irão ser aplicadas a inteiros, strings, pontos de curva ou combinações destes objectos. Estas occurências devem ser interpretadas como hashes de representações binárias, ou concatenações de tais representações. Dependendo do contexto, a hash resultante será numérica, uma sequência de bits, ou até um ponto de curva. Mais detalhes sobre o mesmo serão dados quando necessário.

2.2 Aritmética modular

A maioria da criptografia moderna começa com aritmética modular, e assim com a operação de modulus (denotado por 'mod'). Aqui só é de interesse o modulus positivo, que devolve sempre um inteiro positivo.

O modulus positivo é similar ao resto da divisão inteira. Formalmente, o modulus positivo é definido como $c = a \pmod b$ tal que $a = bx + c$, em que $0 \leq c < b$. Sendo x um inteiro que é descartado.

Note-se que, se $a \leq n$, $-a \pmod n$ é o mesmo que $n - a$.

2.2.1 adição e multiplicação modular

Na ciência da computação é importante evitar números enormes ao executar operações aritméticas. Por exemplo se temos $29 + 87 \pmod{99}$ e não são permitidas variáveis com mais de 3 ou 4 digitos (tal como $116 = 29 + 87$), então não podemos calcular $116 \pmod{99} = 17$ directamente.

Seja que $c = a + b \pmod n$, em que a e b são ambos menos do que o modulus n, podemos fazer o seguinte:

[1] O algoritmo da função hash Keccak é a base para o standard NIST *SHA-3* [**?**].

- Calcule $x = n - a$. Se $x > b$ então $c = a + b$, se não $c = b - x$.

Podemos utilizar adição modular para executar multiplicação modular $(a * b \pmod n) = c)$ com um algoritmo chamado 'double-and-add'. Eis um exemplo, queremos calcular $7 * 8 \pmod 9 = 2$. O que é o mesmo que :

$$7 * 8 = 8 + 8 + 8 + 8 + 8 + 8 + 8 \pmod 9$$

Isso é repartido em grupos de dois :

$$(8 + 8) + (8 + 8) + (8 + 8) + 8$$

Novamente, em grupos de dois :

$$[(8 + 8) + (8 + 8)] + (8 + 8) + 8$$

O número total de operações + decresce de 6 a 4 porque só temos de calcular $(8 + 8)$ uma vez. [2]

Duplica-e-adiciona é implementado ao converter o primeiro número a para binário, depois passando pelo vector binário, duplicando e adicionando.

Seja um vector $V = [0111]$ indexado com 3,2,1,0. [3] V[0] = 1 é o primeiro elemento de V e é o bit menos significante. A[0] = 1 é o primeiro elemento de A e é o bit menos significante. Existem mais duas variáveis, uma de resultado, que no início é : $r = 0$. E uma variável de soma, que no início é : $s = b$. Segue-se este algoritmo :

1. Itera-se através de : $i = (0, ..., A_{size} - 1)$
   (a) Se A[i] == 1, então $r = r + s \pmod n$.
   (b) Calcula-se $s = s + s \pmod n$.
2. Usa-se finalmente r: $c = r$.

Por exemplo $7 * 8 \pmod 9$, esta sequência acontece :

1. $i = 0$
   (a) A[0] = 1, então $r = 0 + 8 \pmod 9 = 8$
   (b) $s = 8 + 8 \pmod 9 = 7$
2. $i = 1$
   (a) A[1] = 1, então $r = 8 + 7 \pmod 9 = 6$
   (b) $s = 7 + 7 \pmod 9 = 5$

[2] O efeito de duplica-e-adiciona torna-se aparente com números grandes. Por exemplo, com $2^{15} * 2^{30}$ adição normal iria requerer por volta de 2^{15} + operações, enquanto que duplica-e-adiciona só requer 15!

[3] Isto é conhecido como 'LSB 0', do inglés :*least significant bit*, o bit menos significante que tem o index 0. No resto deste capítulo irá-se usar 'LSB 0', como notação para esse bit. Trata-se aqui de clareza e não de convenções precisas.

3. $i = 2$
   (a) A[2] = 1, então $r = 6 + 5 \pmod 9 = 2$
   (b) $s = 5 + 5 \pmod 9 = 1$
4. $i = 3$
   (a) A[3] = 0, então r fica o mesmo
   (b) $s = 1 + 1 \pmod 9 = 2$
5. e o resultado é : $r = 2$

2.2.2 Exponenciação modular

Claramente $8^7 \pmod 9 = 8 * 8 * 8 * 8 * 8 * 8 * 8 \pmod 9$. Da mesma forma que duplica-e-adiciona, pode-se também levar-ao-quadrado-e-multiplicar. Para $a^e \pmod n$:

1. Itera-se através de : $i = (0, ..., A_{size} - 1)$
   (a) Se A[i] == 1, então $r = r * m \pmod n$.
   (b) Calcula-se $m = m * m \pmod n$.
2. Usa-se o r final, como resultado.

2.2.3 Inversa multiplicativa modular

As vezes precisa-se de $1/a \pmod n$, ou por outras palavras $a^{-1} \pmod n$. A inversa de algo vezes sí próprio é por definição 1 (identidade). Por exemplo : $0.25 = 1/4$, e depois $0.25 * 4 = 1$.

Em aritmética modular, para a e n relativamente primos :

$$c = a^{-1} \pmod n,$$
$$ac \equiv 1 \pmod n,\ 0 \leq c < n.$$

[4] Relativamente primo significa que eles não partilham nenhum divisor comum excepto 1 (a fracção a/n não pode ser reduzida ou simplificada).

Podemos usar leva-ao-quadrado-e-multiplica para calcular a inversa multiplicativa quando n é um número primo por causa do *pequeno teorema de Fermat* : [5][**?**]
Isto significa que se pode fazer : $c = a^{-1}b \pmod n \rightarrow ca \equiv b \pmod n \rightarrow a \equiv c^{-1}b \pmod n$

$$a^{n-1} \equiv 1 \pmod n$$
$$a * a^{n-2} \equiv 1 \pmod n$$
$$c \equiv a^{n-2} \equiv a^{-1} \pmod n$$

[4] Na equação $a \equiv b \pmod n$, a é *congruente* a $b \pmod n$, o que só significa $a \pmod n = b \pmod n$.
[5] A inversa multiplicativa modular tem uma regra que diz : Se $ac \equiv b \pmod n$ com a e n relativamente primos, então a solução para esta congruência linear é dada por $c = a^{-1}b \pmod n$.

Mais geralmente (e mais rapidamente), o algoritmo extendido de euclides [**?**], também é capaz de encontrar inversas modulares.

2.2.4 Equações modulares

Seja uma equação $c = 3 * 4 * 5 \pmod 9$. Dada uma operação binária $\circ$ (por exemplo, $\circ = *$) entre duas espressões A e B:

$$(A \circ B) \pmod n = [A \pmod n] \circ [B \pmod n] \pmod n$$

Neste exemplo, declara-se $A = 3 * 4$, $B = 5$, e $n = 9$:

$$\begin{aligned}(3 * 4 * 5) \pmod 9 &= [3 * 4 \pmod 9] * [5 \pmod 9] \pmod 9 \\ &= [3] * [5] \pmod 9 \\ c &= 6\end{aligned}$$

Agora pode-se também fazer subtração modular :

$$\begin{aligned}A - B \pmod n &\to A + (-B) \pmod n \\ &\to [A \pmod n] + [-B \pmod n] \pmod n\end{aligned}$$

O mesmo princípio aplica-se a algo como $x = (a - b * c * d)^{-1}(e * f + g^h) \pmod n$.

2.3 Criptografia de curva elíptica

2.3.1 O que são curvas elípticas?

Um corpo finito $\mathbb{F}_q$, no qual q é um número primo maior que 3, é o corpo formado pelo conjunto $\{0, 1, 2, ..., q-1\}$. Adição e multiplicação $(+, \cdot)$ e negação $(-)$ são calculados $\pmod q$.

`fe`: field element

"Calculado $\pmod q$"significa que $\pmod q$ é aplicado a qualquer instância de uma operação aritmética entre dois elementos pertencentes ao mesmo corpo, como também á sua negação. Por exemplo, dado um corpo finito $\mathbb{F}_p$ com $p = 29$, $17 + 20 = 8$ porque $37 \pmod{29} = 8$. Como também, $-13 = -13 \pmod{29} = 16$.

Típicamente, uma curva elíptica com um dado par (a, b) é definida como o conjunto de todos os pontos com coordinadas (x, y) que satisfaçam a equação de *Weierstraß* [**?**]:[6]

$$y^2 = x^3 + ax + b \quad \text{em que} \quad a, b, x, y \in \mathbb{F}_q$$

A cripto-moeda Monero utiliza uma curva especial pertencente a categoria de assim chamadas curvas *Twisted Edwards* [**?**], que são espressadas para um dado par (a, d) :

[6] Notação: a frase $a \in \mathbb{F}$ significa que a é um elemento no corpo finito $\mathbb{F}$.

$$ax^2 + y^2 = 1 + dx^2y^2 \quad \text{em que} \quad a, d, x, y \in \mathbb{F}_q$$

Iremos de seguida preferir esta forma de representação, a vantagem dada por esta forma é que certas primitivas criptográficas requerem assim menos operações aritméticas. O que resulta em algoritmos mais rápidos (veja-se Bernstein *et al.* em [**?**] para mais detalhes).

Sejam $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$ dois pontos pertencentes a uma curva elíptica *Twisted Edwards* (daqui em diante representado somente por uma CE). Definimos adição nos pontos $P_1 + P_2 = (x_1, y_1) + (x_2, y_2)$ como o ponto $P_3 = (x_3, y_3)$ tal que :[7].

$$x_3 = \frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2} \pmod q$$

$$y_3 = \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \pmod q$$

Estas formulas para adição também se aplicam para duplicar um ponto, isto é, se $P_1 = P_2$.
Para subtraír um ponto, inverte-se as suas coordinadas sobre o eixo y : $(x, y) \to (-x, y)$ [**?**], e depois aplica-se duplicar o ponto. Note-se que elementos negativos em $\mathbb{F}_q$, $-x$ são : $-x \pmod q$. Sempre que pontos de curva elíptica são adicionados, P_3 é um ponto que se mantêm na curva elíptica. Ou seja $x_3, y_3 \in \mathbb{F}_q$, e satisfazem a equação de CE. Daqui em diante diz-se que o ponto x_3, y_3 está em CE. Ou então, o Ponto P em CE.

Um ponto P em CE pode gerar um subgrupo de ordem (tamanho) u, ao multiplicar-se consigo próprio.
Por exemplo, um ponto P pode formar um subgrupo de ordem 5 e conter os pontos $(0, P, 2P, 3P, 4P)$, cada um dos quais está em CE.

Neste caso em vez de $5P$, acontece o *ponto-no-infinito*, que é como se fosse a posição zero na curva elíptica. [8] Convenientemente, $5P + P = P$. Isto significa que o subgrupo é *cíclico*.
Todos os pontos P em CE geram um subgrupo cíclico. Se P gera um subgrupo cuja ordem é um número primo, então *todos os pontos* incluídos nesse subgrupo geram esse mesmo subgrupo (excepto claro o ponto no infinito). No exemplo anterior tome-se múltiplos do ponto $2P$:

$$2P, 4P, 6P, 8P, 10P \to 2P, 4P, 1P, 3P, 0$$

Outro exemplo: um subgrupo com ordem 6 $(0, P, 2P, 3P, 4P, 5P)$.
Tome-se múltiplos do ponto $2P$:

$$2P, 4P, 6P, 8P, 10P, 12P \to 2P, 4P, 0, 2P, 4P, 0$$

Aqui o subgrupo gerado pelo gerador $2P$ tem ordem 3. Desde que 6 não é primo, nem todos os membros do subgrupo, geram o próprio subgrupo.

[7] Típicamente, por razões de eficiência, pontos de curvas elípticas são convertidos em coordenadas projetivas antes de operações como a adição, de forma a evitar inversões dos pontos no dado corpo finito [**?**]

[8] Acontece que curvas elípticas tem uma estrutura de grupo *abeliana* para a operação de adição descrita aqui, desde que o ponto no infinito é o elemento de identidade. Uma definição concisa desta noção pode ser encontrada aqui : https://brilliant.org/wiki/abelian-group/.

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge.h

Cada curva elíptica tem uma ordem N igual ao número total de pontos que está em CE, isto *incluí* o ponto no infinito, e a ordem de cada subgrupo que acontece, divide N (*sem resto*). Isto é pelo theorema de Lagrange. Por outras palavras, imagine-se então um conjunto de todos os pontos em CE $\{0, P_1, ..., P_{N-1}\}$. Se N não é primo, então só existem subgrupos cuja ordem divide N.
Note-se no parágrafo anterior, que a ordem do subgrupo gerado por $2P$ tem ordem 3, e 3 divide 6.

Para encontrar a ordem, u, do subgrupo de qualquer ponto dado P :

1. Encontra-se N (e.g. usa-se o *algoritmo de Schoof*).
2. Encontram-se todos os divisores de N.
3. para cada divisor n de N, calcula-se nP.
4. O menor n tal que $nP = 0$ é a ordem u do subgrupo.

Curvas elípticas seleccionadas para criptografia tipicamente têm $N = hl$, em que l é um número primo suficientemente grande (com 160 bits) e h é o assim chamado *cofactor* que pode ser tão pequeno como 1 ou 2. [9] Um ponto no subgrupo de tamanho l é usualmente seleccionado para ser o gerador G como uma convenção. Para cada outro ponto P nesse subgrupo existe um inteiro $0 < n \leq l$ que satisfaz $P = nG$.

Calcular o produto escalar entre qualquer inteiro n e qualquer ponto P, nP, não é difícil.
Ao contrário de encontrar n dados dois pontos P_1 e P_2 tal que :

$$P_1 = nP_2$$

o que é consenso, de ser difícil, computacionalmente infazível.
É isto que se chama o *problema do logaritmo discreto* (PLD).
Multiplicação escalar pode como tal ser vista como uma função unidireccional. Isto permite o uso de curvas elípticas para o fim de criptografia. Não existe até á data, alguma equação, esquema ou algoritmo para resolver n na equação $P_1 = nP_2$. Ou seja mesmo com todos os computadores do mundo a tentarem resolver para n, só depois de milhões de anos é que uma solução seria encontrada. Existe um inteiro, γ que é central em Monero, que ninguêm conhece e que está protegido pelo PLD (secção 5.3).

O produto escalar nP é equivalente a $(((P+P)+(P+P))+(P+P))\ldots n\ vezes)$. Apesar de não ser todas as vezes o mais efficiente, pode-se utilizar duplica-e-adiciona como na secção 2.2.1, para obter a soma $R = nP$.

1. Define-se $n_{scalar} \rightarrow n_{binary}$; $A = [n_{binary}]$; $R = 0$, o ponto no infinito; $S = P$
2. Itera-se sobre: $i = (0, ..., A_{size} - 1)$
   (a) Se A[i] == 1, então R += S.

[9] Curvas elípticas com cofactores pequenos permitem uma adição de ponto rápida etc . [**?**].

(b) Computa-se S += S.

3. Usa-se o R final como resultado.

Note-se que escalares para pontos no subgrupo de tamanho l, são membros do corpo finito $\mathbb{F}_l$. Isto significa que a aritmética entre escalares é mod l.

2.3.2 Criptografia de chave pública com curvas elípticas

Seja k um número aleatório escolhido tal que $0 < k < l$. Chama-se a este número uma *chave privada*. Também dito como uma chave secreta, abrevia-se então K = chave pública, k = chave secreta. Calcula-se então a *chave pública* K com o produto escalar $kG = K$. Devido ao PLD não podemos facilmente deduzir k de K sabendo G. Isto permite o uso standard de algoritmos de criptografia de chave pública.

2.3.3 A troca de chaves tipo Diffie-Hellman com curvas elípticas

A troca básica tipo *Diffie-Hellman* [**?**] de um segredo partilhado entre a *alice* e o *bob* acontece da seguinte maneira :

1. A alice e o bob geram as suas próprias chaves (k_A, K_A) and (k_B, K_B). Ambos publicam ou trocam as suas chaves públicas, e mantêm as suas chaves privadas para sí próprios.

2. Claramente é verdade que :

$$S = k_A K_B = k_A k_B G = k_B k_A G = k_B K_A = S$$

A alice pode calcular privadamente $S = k_A K_B$,
e o bob pode calcular privadamente $S = k_B K_A$,
ambos podem utilizar este valor S como um segredo partilhado.

Se a alice quer enviar uma mensagem m ao bob,
ela calcula a hash do segredo partilhado $h = \mathcal{H}(S)$,
depois $x = m + h$, e envia x ao bob .

O bob também calcula $h = \mathcal{H}(S)$,
depois $m = x - h$, e como tal, o bob aprende m.

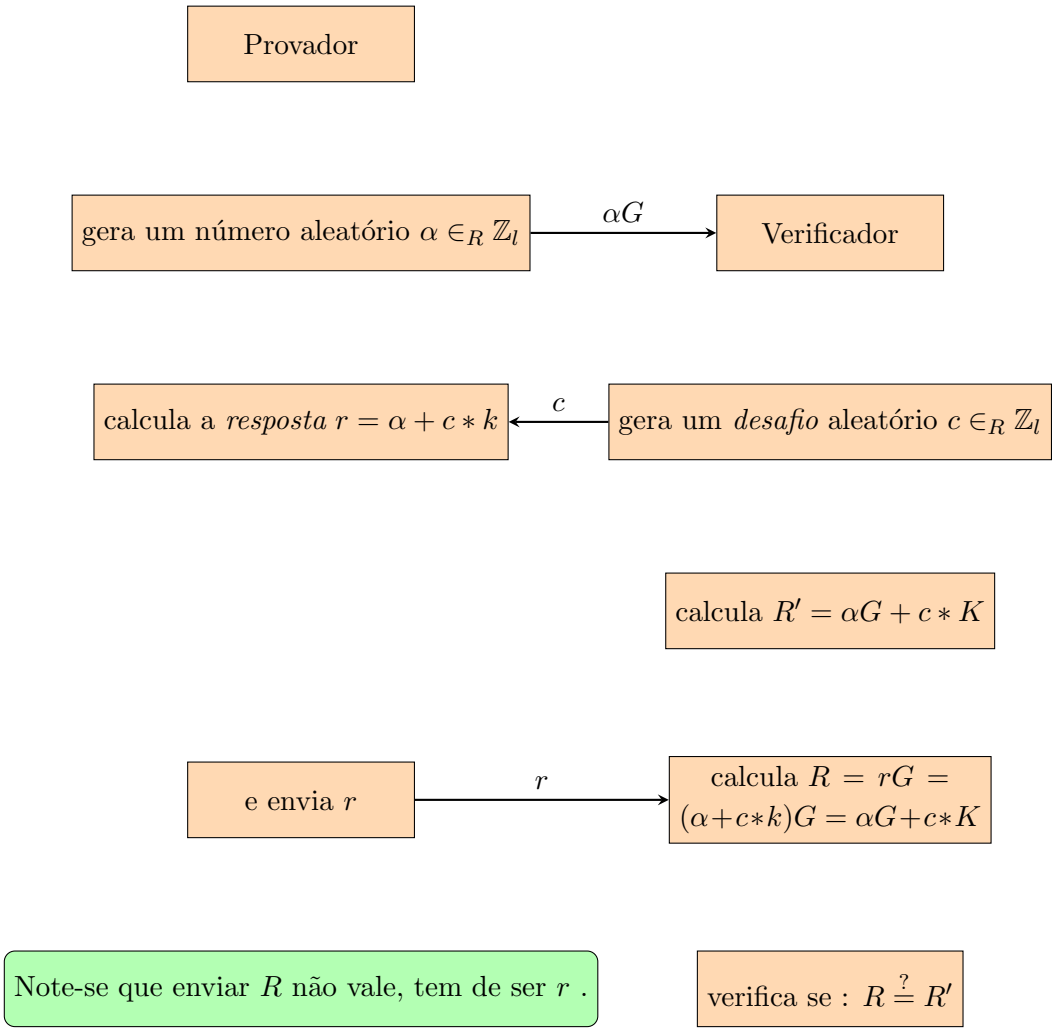
Um observador externo não seria capaz de facilmente calcular o segredo partilhado devido ao 'Problema Diffie-Hellman' (PDH), que diz que encontrar S de K_A e K_B é consenso, de ser difícil, computacionalmente infazível. Da mesma forma o DLP previne que se possa descobrir k_A ou k_B sabendo K_A e K_B [10] .

[10] Diz que o PDH é pelo menos tão difícil como o DLP, mas isto ainda não foi provado. [**?**]

2.3.4 Assinaturas tipo Schnorr e a transformada Fiat-Shamir

Em 1989 Claus-Peter Schnorr, publicou um agora famoso protocolo de autenticação interactivo [**?**]. Este foi generalizado por Maurer em 2009 [**?**]. O protocolo permite ao signatário provar que dada uma chave pública K tem conhecimento da chave privada k, sem revelar alguma informação sobre ela [**?**].

Ambas as partes conhecem a chave pública K e o gerador G bem como o grupo pertencente. Acontece da seguinte forma :



Ou seja se $R = R'$, então o desafio foi *vencido* o que implica que o provador tinha de conhecer k .
O verificador pode calcular $R' = \alpha G + c * K$ antes do provador, assim oferecer c é como dizer : "eu desafio-te a responderes com o logaritmo discreto de R'".
Este desafio só pode ser vencido pelo provador que conhece k, que envia r, o que resolve a equação do lado do verificador (enp).

Se α foi escolhido de forma aleatória então r está distribuído de forma aleatória [**?**] e k está

seguro em termos da teoria da informação dentro de r .
Segurança em termos da teoria da informação significa que mesmo um adversário com um poder de computação infinito não é capaz de a quebrar. Contudo, se um provador reutiliza α para provar o seu conhecimento de k...
Quem souber ambos os desafios em $r = \alpha + c * k$ and $r' = \alpha + c' * k$, pode calcular k [11] .

$$k = \frac{r - r'}{c - c'}$$

No seu passo de desafio, o verificador oferece um número aleatório depois de receber αG, o que o torna equivalente a uma *função aleatória*. Funções aleatórias, como funções de hash, são conhecidas como oráculos aleatórios. Em vez de ser o verificador a oferecer um desafio usa-se uma função hash.
Isto é conhecido como a *transformada de Fiat-Shamir* [**?**], a prova torna-se não-interactiva e publicamente verificável [**?**]. Da mesma forma como antes, o público conhece a chave pública K o gerador G e o grupo pertencente. A única diferença é que o desafio provêm de uma função hash conhecida também ao público.
Acontece da seguinte forma : [12] [13] [14]

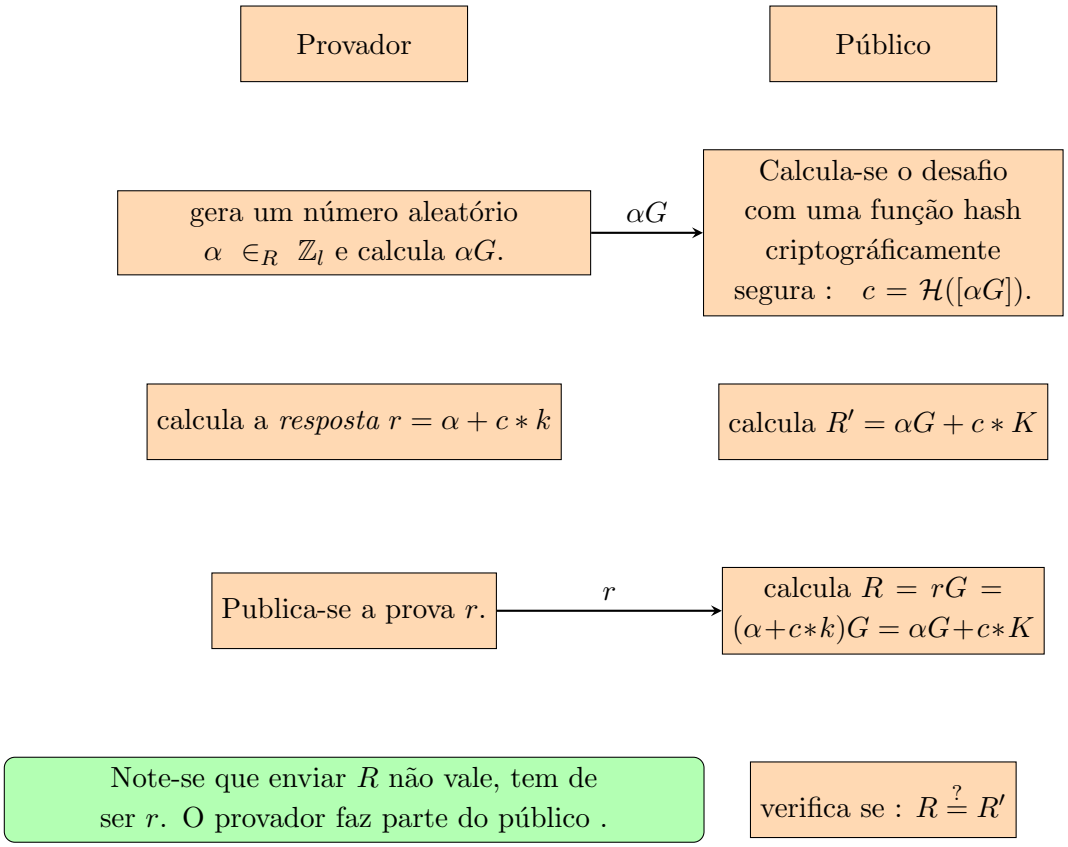
[11] Se o provador é um computador, o leitor pode imaginar que alguêm faz uma cópia do computador depois deste gerar α, e depois apresentar cada cópia com um desafio diferente.

[12] Mais geralmente, "[N]a criptografia... um oráculo é qualquer sistema que dá uma informação sobre um outro sistema, que de outra forma não estaria disponível."[**?**]

[13] O resultado de uma função hash criptográfica $\mathcal{H}$ está uniformemente distribuído entre o domínio de todos os resultados possíveis. Isto significa que, para um argumento A, $\mathcal{H}(A) \in_R^D \mathbb{S}_H$ em que $\mathbb{S}_H$ é o conjunto de resultados possíveis de $\mathcal{H}$. Usa-se $\in_R^D$ para indicar que a função é deterministicamente aleatória. $\mathcal{H}(A)$ produz sempre o mesmo resultado e esse resultado depende de um número aleatório.

[14] Note-se que provas não interactivas do tipo Schnorr (e as assinaturas) requerem o uso de um gerador fixo G, ou a inclusão desse gerador no desafio hash. Isto é conhecido como prefixo de chave e será apresentado mais tarde (secção 3.4 e 9.2.3).

Prova não interactiva



Ou seja se $R = R'$, então o desafio foi *vencido* o que implica que o provador tinha de conhecer k . O provador faz parte do público, ou seja ele podia fazer este esquema sozinho. O que não faz muito sentido, provar a sí próprio que conhece k, é claro que o público não conhece k.

Parte importante de cada esquema de prova/assinatura são os requisitos computacionais para os verificar. Isto incluí o espaco para guardar as provas, e o tempo gasto para verificar. Neste esquema guarda-se um ponto de CE e um inteiro, mais a chave pública - outro ponto de CE. O tempo de verificação depende das operações de curva elíptica, desde que funções hash são rápidas de executar. src/ringct/rctOps.cpp

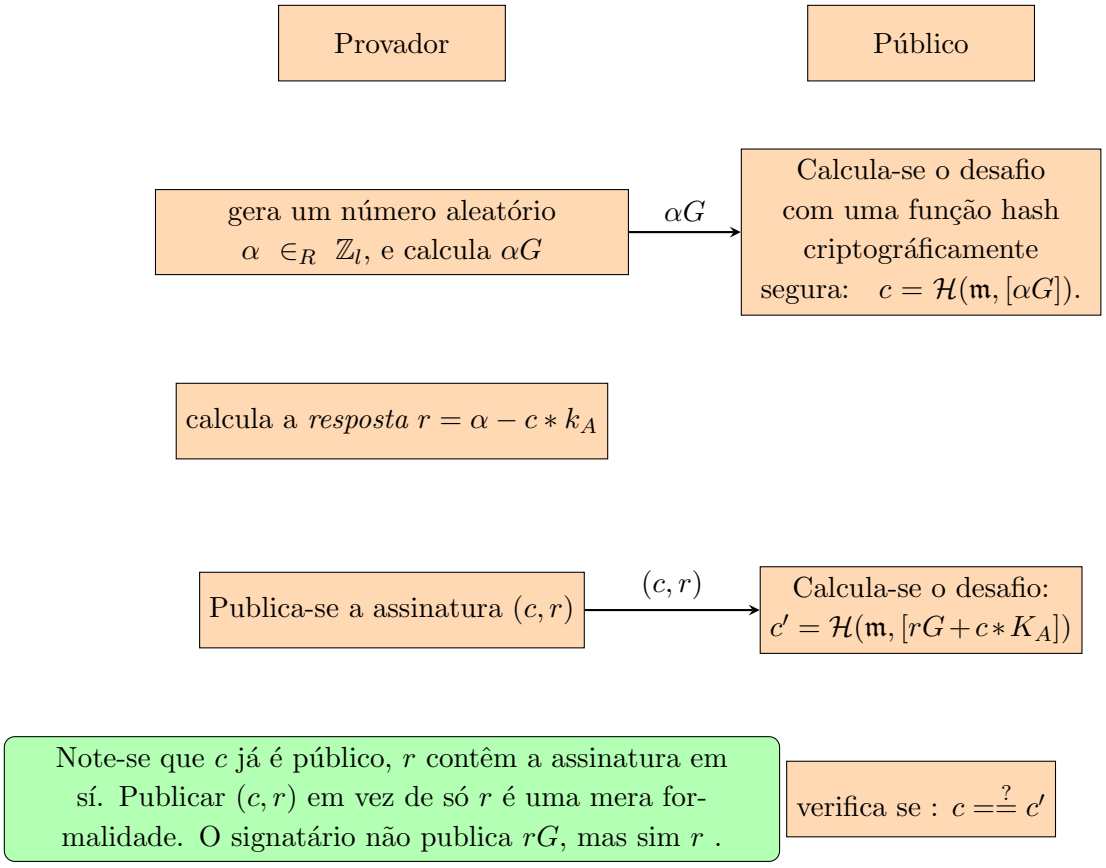
2.3.5 Assinar mensagens

Típicamente, uma assinatura criptográfica é executada numa hash criptográfica de uma mensagem em vez de se assinar a própria mensagem. Isto facilita assinar mensagens de tamanho variável. Contudo aqui refere-se a mensagem com o seu símbolo $\mathfrak{m}$, de forma vaga, excepto quando definido se isso é o valor hash ou não.

Assinar mensagens faz parte da segurança na internet, o que deixa um recipiente de uma mensagem confiar que o seu conteúdo foi intendido pelo signatário. Um esquema comum de

assinaturas é o de ECDSA. Veja-se [**?**], ANSI X9.62, e [**?**].

O esquema de assinaturas apresentado aqui é uma formulação alternativa da prova de Schnorr transformada. Pensar em assinaturas desta forma irá preparar-nos para explorar assinaturas em anel no próximo capítulo.



Se $c = c'$ então a assinatura é válida.

Funciona porque

Isto provém do facto que

$$
\begin{aligned}
r &= \alpha - c * k_A \\
rG &= (\alpha - c * k_A)G \\
rG &= \alpha G - c * K_A \\
\alpha G &= rG + c * K_A \\
\mathcal{H}_n(\mathfrak{m}, [\alpha G]) &= \mathcal{H}_n(\mathfrak{m}, [rG + c * K_A]) \\
c &= c' \\
\alpha G &= (\alpha - c * k_A)G + c * K_A \\
\alpha G &= \alpha G - c * K_A + c * K_A \\
\alpha G &= \alpha G
\end{aligned}
$$

"dabra pokus!"

Portanto a alice criou (c, r) em dependência de $\mathfrak{m}$ com posse de k_A :
ela assinou a mensagem.
A probabilidade que outra pessoa sem posse de k_A, podia ter produzido r é negligível, portanto um verificador confia que a mensagem não foi falsificada.

Requisitos :

Neste esquema de assinatura guardam-se dois escalares, e uma chave pública de CE.

2.4 Curva Ed25519

Monero utiliza uma curva elíptica particular tipo *Twisted Edwards* para operações criptográficas, *Ed25519* o *equivalente birational* da curva de *Montgomery* : *Curve25519*. [15] Ambas as curvas *Curve25519* e *Ed25519* foram públicadas por Bernstein *et al.* [**?**, **?**, **?**]. [16]

A curva é definida sobre o corpo finito primo :

$$\mathbb{F}_{2^{255}-19}$$

através da seguinte equação : src/crypto/crypto_ops_builder/ref10CommentedCombined/ge.h

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

Esta curva faz juz a algumas preocupações da comunidade de criptografia. [17]

É bem sabido que algoritmos standard do NIST têm problemas[18] .
Por exemplo, tornou-se recentemente claro que o algoritmo standard para gerar números aleatórios, a versão baseada em curvas elípticas, contêm uma potential porta de trás [**?**]. Visto de uma perspectiva mais ampla, autoridades que emitem standards como o NIST levam a uma monocultura na criptografia, introdizindo um ponto de centralização. Um grande exemplo disto foi quando a NSA utilizou a sua influência sobre o NIST para enfraquecer um standard international de criptografia [**?**]. A curva *Ed25519* não está sujeita a nenhums patentes (veja-se [**?**]), e a sua equipa desenvolveu e adaptou algoritmos criptográficos com foco na eficiência[**?**]. Curvas do tipo *Twisted Edwards* têm uma ordem que se expressa como : src/crypto/crypto_ops_builder/

$$N = 2^c l,$$

[15] Sem dar mais detalhes, equivalência birational pode ser vista como um isomorfismo que usa termos rationais.
[16] Dr. Bernstein também desenvolveu um esquema de encriptação conhecido como ChaCha [**?**, **?**], que a implementação primária de Monero utiliza para encriptar certas informações sensitivas relacionadas com a carteira. src/wallet/ringdb.cpp
[17] Mesmo se uma curva aparentemente não tem nenhums problemas de segurança criptográfica, é possível que a pessoa ou organização que criou essa curva conheça um aspecto ou característica que só acontece em curvas raras... Esta pessoa pode gerar imensas curvas diferentes que não têm nenhums erros conhecidos, mas que tem um erro escondido a terceiros. Se adicionalmente são necessárias explicações para cada parametro de curva, para que esta curva seja aceite na comunidade criptográfica, então torna-se ainda mais difícil encontrar curvas com tais fraquezas. A curva Ed25519 é conhecida como uma curva 'fully rigid', o que significa que o processo de geração desta curva está completamente descrito. [**?**]
[18] National Institute of Standards and Technology, https://www.nist.gov/

em que l é um número primo e c é um inteiro positivo. No caso de Monero, a ordem é um número com 76 digitos, l tem 253 bits [19]:

$$2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$$

src/ringct/rctOps.h `curveOrder()`

2.4.1 Representação binária

Elementos de $\mathbb{F}_{2^{255}-19}$ estão codificados como inteiros de 256-bits, portanto podem ser representados com 32 bytes. Desde que cada elemento só requer 255 bits, o bit mais significante é sempre zero.

2.4.2 Compressão de ponto elíptico

A curva Ed25519 tem a propriedade que os seus pontos podem ser facilmente comprimidos, portanto representar um ponto é possível com o espaço de só uma coordinada. Não iremos detalhar a matemática necessária para justificar isto, mas podemos dar uma breve esplicação de como isto funciona [**?**]. A compressão de ponto foi primeiro descrito em [**?**], enquanto que o conceito foi primeiro introduzido em [**?**].

Este esquema de compressão de ponto segue-se de uma transformação da equação de curva *Twisted Edwards*, (assumindo $a = -1$, o que é verdade para Monero) :

$$x^2 = \frac{y^2 - 1}{dy^2 + 1},$$

em que $d = -\frac{121665}{121666}$. Como tal existem dois valores possíveis para x para cada y. Os elementos do corpo finito x e y são calculados $\pmod q$, por isso não existem valores negativos. No entanto, $\pmod q$ de $-x$ muda o valor entre par e impar desde que q é impar. Por exemplo : $3 \pmod 5 = 3$, $-3 \pmod 5 = 2$. Por outras palavras, os elementos do corpo finito x e $-x$ têm valores pares e impares diferentes. Seja um ponto de CE em que o seu x é par, mas dado o seu valor y, a transformada da equação de curva resulta num número impar, então sabe-se que negar esse número irá dar-nos o x correcto. Um bit consegue conter esta informação, e convenientemente a coordinada de y tem um bit extra.

Seja que se quer comprimir um ponto (x, y).

Codificar O bit mais significante de y é posto a zero se x é par, se não é posto a 1. O valor resultante y' representa o ponto de curva CE.

Descodificar

[19] Isto significa que as chaves privadas em Ed25519 têm 253 bits.

(a) Do ponto comprimido y' copia-se o bit mais significante para uma variável b, Agora põe-se o bit mais significante de y' de volta a zero, e obtêm-se outra vez o valor y original.

(b) Seja $u = y^2 - 1 \pmod q$ e $v = dy^2 + 1 \pmod q$. Isto significa $x^2 = u/v \pmod q$.

(c) Calcule[20] $z = uv^3(uv^7)^{(q-5)/8} \pmod q$.

i. Se $vz^2 = u \pmod q$ então $x' = z$.

ii. Se $vz^2 = -u \pmod q$ então calcula-se $x' = z * 2^{(q-1)/4} \pmod q$.

ge_from-bytes.c

(d) Usa-se o bit de paridade, se $b \neq$ do bit menos significante de x' então $x = -x' \pmod q$, se não $x = x'$.

(e) Resulta então o ponto descomprimido (x, y).

Implementações de Ed25519 (como em Monero) típicamente usam o gerador $G = (x, 4/5)$ [**?**], em que x é par.

2.4.3 Algoritmo de assinatura EdDSA

Com a intenção de ilustrar descreve-se uma alternativa optimizada e segura do esquema de ECDSA, que segundo os autores, permite produzir mais de 100 000 assinaturas por segundo com um processador Intel Xeon [**?**]. O algoritmo também pode ser lido em *RFC8032* [**?**]. Note-se que este esquema de assinaturas é do estilo Schnorr. Entre outras coisas, em vez de gerar inteiros aleatórios cada vez, é usado um valor hash derivado da chave privada do signatário e da própria mensagem. Isto evita problemas de segurança relacionados com a implementação de geradores de números aleatórios. Outro objectivo deste algoritmo é de evitar o acesso a espaços secretos ou imprevisíveis em memória ram, o que previne assim chamados *ataques de cache temporais* [**?**]. Aqui são esplicados por alto os passos executados pelo algoritmo. Uma descrição completa e uma implementação exemplo em Python pode ser encontrada em [**?**].

Assinatura

(a) Seja h_k uma hash $\mathcal{H}(k)$ da chave privada do signatário k. Calcula-se α como uma hash $\alpha = \mathcal{H}(h_k, \mathfrak{m})$ da chave privada e da mensagem. Dependendo da implementação, $\mathfrak{m}$ pode ser a mensagem ou a sua hash [**?**].

(b) Calcula-se αG e o desafio $ch = \mathcal{H}([\alpha G], K, \mathfrak{m})$.

(c) Calcula-se a resposta $r = \alpha + ch \cdot k$.

(d) A assinatura é o par $(\alpha G, r)$.

[20] Desde que $q = 2^{255} - 19 \equiv 5 \pmod 8$, $(q-5)/8$ e $(q-1)/4$ são inteiros.

Verificação

A verificação procede da seguinte forma :

(a) Calcula-se $ch' = \mathcal{H}([\alpha G], K, \mathfrak{m})$.

(b) Se a seguinte equação é verdade :

$$2^c rG \stackrel{?}{=} 2^c \alpha G + 2^c ch' * K,$$

então a assinatura é válida.

O termo 2^c vem da forma geral do algoritmo EdDSA (Bernstein *et al.*[**?**]). A chave pública K pode ser qualquer ponto de CE, mas só se quer usar os pontos no subgrupo do gerador G. Multiplicar pelo cofactor 2^c garante que todos os pontos estão nesse subgrupo. Alternativamente, pode-se verificar se $lK \stackrel{?}{=} 0$ o que só funciona se K está dentro do subgrupo. Não se sabe de alguma fraqueza causada por estas precauções, e como iremos ver mais tarde, isto é um méthodo importante em Monero (secção 3.4).

Funciona porque

$$\begin{aligned} 2^c rG &= 2^c(\alpha + \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot k) \cdot G \\ &= 2^c \alpha G + 2^c \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot K \\ 2^c(\alpha + ch \cdot k) \cdot G &= 2^c \alpha G + 2^c \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot K \\ 2^c(\alpha + \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot k) \cdot G &= 2^c \alpha G + 2^c \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot K \end{aligned}$$

Requisitos :

Neste esquema de assinatura guarda-se um ponto de CE e um escalar, e uma chave pública de CE.

2.5 Operador binário XOR

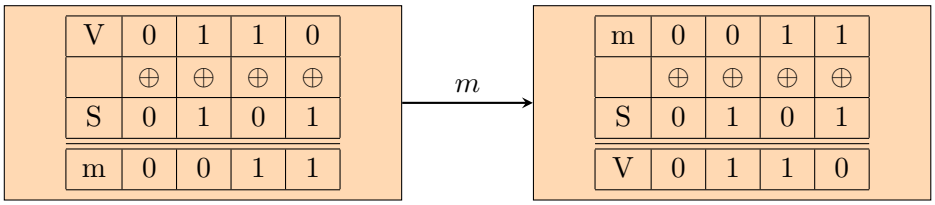
O operador binário XOR é uma ferramenta útil que irá aparecer nas secções 4.4 e 5.3. Leva dois argumentos e devolve verdade se um, mas não ambos são verdade [**?**]. Eis a sua tabela de verdade :

A	B	A XOR B
T	T	F
T	F	T
F	T	T
F	F	F

No contexto da ciência da computação, XOR é equivalente a adição de bits modulo 2. Por exemplo o XOR de dois pares de bits :

$$\begin{aligned} \text{XOR}(\{1,1\},\{1,0\}) &= \{1+1, 1+0\} \pmod 2 \\ &= \{0,1\} \end{aligned}$$

Seja que a alice quer encriptar um vector privado de n bits V, usa-se então um segredo partilhado, um vector S do mesmo comprimento. Aplica-se a operação XOR a cada par de bits, entre V e S. O resultado m é enviado para bob. O bob faz a mesma operação invertida, e obtêm o mesmo vector privado V. Note-se que um observador que desconhece S, e só conhece m, é incapaz de descobrir V.
Existem pois 2^n combinações distintas de V e S que resultam no mesmo m [21] .



[21] Uma outra aplicação de XOR é trocar o conteúdo de dois registos sem um terceiro registo. $\{A, B\} \rightarrow \{[A \oplus B], B\} \rightarrow \{[A \oplus B], B \oplus [A \oplus B]\} = \{[A \oplus B], A \oplus 0\} = \{[A \oplus B], A\} \rightarrow \{[A \oplus B] \oplus A, A\} = \{B, A\}$.

CAPÍTULO 3

Assinaturas tipo Schnorr avançadas

Uma assinatura tipo Schnorr tem uma chave signatária. Acontece que é possivel aplicar estes conceitos para criar uma variedade de assinaturas progressivamente mais complexas. Um desses esquemas, MLSAG, irá ter uma importância central no protocolo de transacção de Monero.

3.1 Provar o conhecimento do logaritmo discreto através de múltiplas bases

É útil provar que a mesma chave privada, foi usada para construir chaves públicas, com chaves *base* diferentes (i.e. geradores diferentes). Por exemplo, seja uma chave pública normal kG, e um segredo partilhado tipo Diffie-Hellman kR respectivo á chave pública de outra pessoa, ou seja as chaves base são G e R. É possível provar conhecimento do logaritmo discreto de k em kG, como também de k in kR, e que k é o mesmo. E isto tudo sem revelar algo sobre k em sí.

Prova não interactiva

Suponha-se que temos uma chave privada k, e d chaves *base* :

$$\mathcal{J} = \{J_1, ...J_d\}.$$

As chaves públicas correspondentes são :

$$\mathcal{K} = \{K_1, ...K_d\}.$$

Em que $K_i = kJ_i$, $i \in \{1, ...d\}$. Faz-se então uma prova tipo Schnorr (secção 2.3.4), através de todas as bases. Assume-se a existência de uma função hash $\mathcal{H}_n$, que para cada inteiro de 0 a $l-1$, devolve uma hash única.

[1] [2]

Assinatura

(a) Gera-se um número aleatório $\alpha \in_R \mathbb{Z}_l$, e calcula-se para cada $i \in (1, ..., d)$, αJ_i. Ou seja : $\alpha\mathcal{J} = \{\alpha J_1, ...\alpha J_d\}$.

(b) Calcula-se o desafio,

$$c = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha J_1], [\alpha J_2], ..., [\alpha J_d])$$

(c) Define-se a resposta $r = \alpha - c * k$.

(d) Publica-se a assinatura (c, r).

Verificação

Assumindo que o verificador sabe $\mathcal{J}$ e $\mathcal{K}$, ele faz o seguinte :

(a) Calcula-se o desafio:

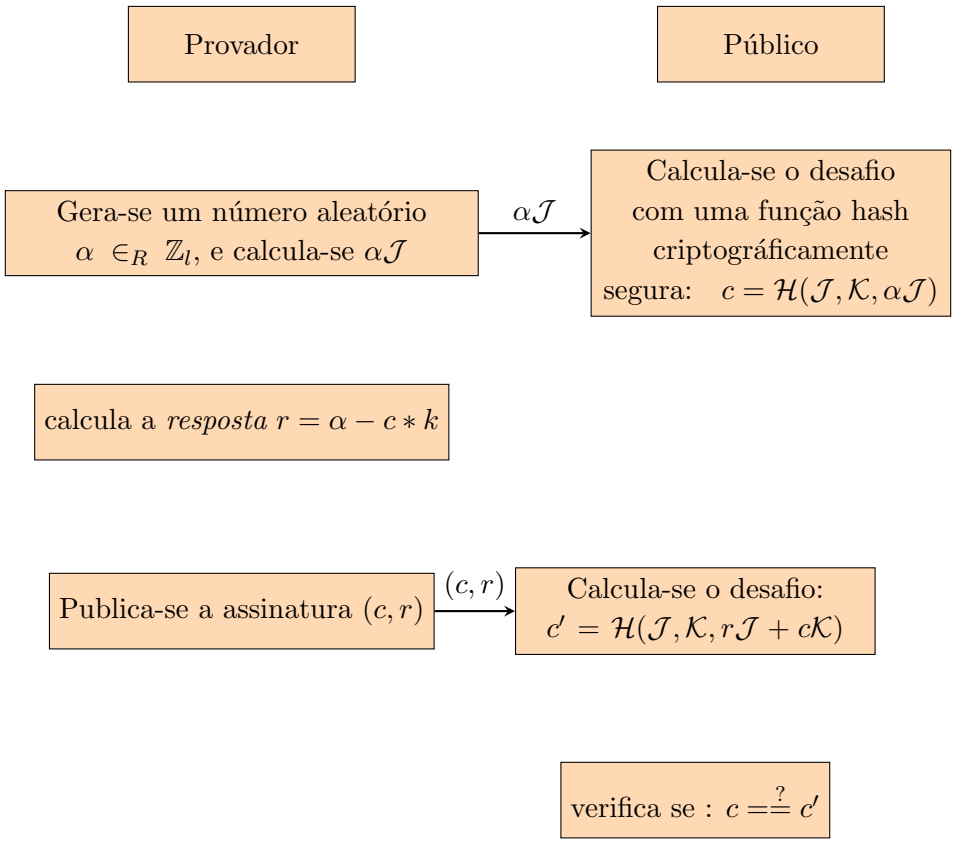
$$c' = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, r\mathcal{J} + c\mathcal{K})$$
$$c' = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [rJ_1 + c * K_1], [rJ_2 + c * K_2], ..., [rJ_d + c * K_d])$$

(b) Se $c = c'$ então o signatário tem de saber o logaritmo discreto através de todas as bases, e é o mesmo em cada caso (enp).

[1] Uma assinatura pode trivialmente ser feita ao incluír uma mensagem $\mathfrak{m}$ no desafio hash.

[2] Em Monero, a função hash $\mathcal{H}_n(x) = \text{sc\_reduce32}(Keccak(x))$, em que $Keccak$ é a base de SHA3 e sc_reduce32() põem o resultado de 256 bits no domínio de 0 a $l-1$ (embora devesse ser de 1 a $l-1$).

Funciona porque



Se $c = c'$ então a assinatura é válida. Se em vez de d chaves base fosse só uma, esta prova seria claramente a mesma que foi apresentada na secção 2.3.4. A sequência de equações que se seguem resolvem-se da mesma maneira :

$$\begin{aligned} r &= \alpha - c * k \\ r\mathcal{J} &= (\alpha - c * k)\mathcal{J} \\ r\mathcal{J} &= \alpha\mathcal{J} - c * k\mathcal{J} \\ \alpha\mathcal{J} &= r\mathcal{J} + c * \mathcal{K} \\ \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha\mathcal{J}]) &= \mathcal{H}_n(\mathcal{J}, \mathcal{K}, r\mathcal{J} + c * \mathcal{K}) \\ c &= c' \\ \alpha\mathcal{J} &= (\alpha - c * k)\mathcal{J} + c * \mathcal{K} \\ \alpha\mathcal{J} &= \alpha\mathcal{J} - c * \mathcal{K} + c * \mathcal{K} \\ \alpha\mathcal{J} &= \alpha\mathcal{J} \end{aligned}$$

Desta vez o signatário tem de conhecer o logaritmo discreto através de todas as bases para que a assinatura seja válida (enp). Diz-se que o logaritmo discreto é o mesmo em cada caso

ou para cada base. Pois k é sempre o mesmo em $K_i = kJ_i$, $i \in \{1, ...d\}$.

3.2 Uma prova com múltiplas chaves privadas

Bem como na prova através de múltiplas bases, podem-se combinar muitas provas tipo Schnorr que usam chaves privadas diferentes. Isto faz com que se prove a posse de todas as chaves privadas para um conjunto de chaves públicas. E todas as provas juntas só requerem um desafio.

Prova não interactiva

Sejam d chaves privadas $k_1, ..., k_d$, e chaves base $\mathcal{J} = \{J_1, ..., J_d\}$. [3] As chaves públicas correspondentes são $\mathcal{K} = \{K_1, ..., K_d\}$. Faz-se uma prova tipo Schnorr de todas as chaves ao mesmo tempo.

Assinatura

(a) Geram-se $\alpha_i \in_R \mathbb{Z}_l$ para cada $i \in (1, ..., d)$, e calculam-se todos os $\alpha_i J_i$.

(b) Calcula-se o desafio,

$$c = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha_1 J_1], [\alpha_2 J_2], ..., [\alpha_d J_d])$$

(c) Define-se cada resposta $r_i = \alpha_i - c * k_i$.

(d) Publica-se a assinatura $(c, r_1, ...r_d)$.

Verificação

Assumindo que o verificador sabe $\mathcal{J}$ e $\mathcal{K}$, ele faz o seguinte :

(a) Calcula-se o desafio:

$$c' = \mathcal{H}(\mathcal{J}, \mathcal{K}, [r_1 J_1 + c * K_1], [r_2 J_2 + c * K_2], ..., [r_d J_d + c * K_d])$$

(b) Se $c = c'$, então a assinatura é válida. O signatário tem de saber as chaves privadas de todas as chaves públicas em $\mathcal{K}$ (enp).

[3] Note-se que $\mathcal{J}$ poderia conter chaves base duplicadas, ou que todas as chaves bases fossem iguais (e.g. G). Duplicados seriam redundantes para provas com múltiplas bases, mas aqui usam-se chaves privadas todas distintas.

Funciona porque

$$
\begin{aligned}
r_i &= \alpha_i - c * k_i \\
r_i J_i &= (\alpha_i - c * k_i) J_i \\
r_i J_i &= \alpha_i J_i - c * k_i J_i \\
\alpha_i J_i &= r_i J_i + c * K_i \\
\alpha_i J_i &= (\alpha_i - c * k) J_i + c * K_i \\
\alpha_i J_i &= \alpha_i J_i - c * K_i + c * K_i \\
\alpha_i J_i &= \alpha_i J_i
\end{aligned}
$$

Em que $i \in \{1, ...d\}$. É claro que :

$$\mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha_i J_i]) = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [r_i J_i + c * K_i])$$

mas isto *não* tem algo a ver com c. Portanto cabe ao leitor neste caso imaginar que para cada argumento na função hash depois de $\mathcal{H}_n(\mathcal{J}, \mathcal{K},$
existe um :

$$\alpha_i J_i$$

para um :

$$r_i J_i + c * K_i.$$

Cada tal par de argumentos na função hash dos dois lados da equação é igual. Como tal é também igual a hash destes pares concatenados ao longo do index i. Tal que :

$$\mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha_1 J_1], ...[\alpha_d J_d]) = \mathcal{H}(\mathcal{J}, \mathcal{K}, [r_1 J_1 + c * K_1], ...[r_d J_d + c * K_d])$$

$$c = c'$$

3.3 Assinaturas espontâneas anónimas de grupo

Assinaturas de grupo são uma maneira de provar que um signatário pertence a um grupo, sem necessariamente ter de o identificar. Originalmente (Chaum em [**?**]), esquemas de assinatura de grupo requeriam que o sistema fosse construído e por vezes até gerido por um terceiro de confiança. De forma a prevenir assinaturas ilegítimas e também para adjudicar disputas. Esses esquemas dependiam de um *segredo de grupo* o que não é desejável, pois é criado um risco de divulgação que podia acabar com a anonimidade. Além disso, requer coordinação entre membros do grupo (i.e. para a configuração e gestão) o que não é escalável para além de pequenos grupos ou para empresas.

Liu *et al.* apresentaram um esquema mais interessante em [**?**] construíndo sobre o trabalho de Rivest *et al.* in [**?**]. Os autores então detalharam um algoritmo chamado LSAG que é caracterizado por trés propriedades : *anonimidade, ligação* e *espontaínidade*. Aqui apresenta-mos SAG, a versão sem interligação de LSAG para a clareza conceptual. A ideia de interligação será apresentada em seccões posteriores.

Estes esquemas com anonimidade e espontaínidade chamam-se 'assinaturas de anel'. No contexto de Monero estas assinaturas irão ultimamente permitir transacções infalsificáveis e em que o signatário é ambíguo. O que deixa o movimento de fundos largamente indetectável.

Assinatura

Assinaturas em anel são compostas de um anel e uma assinatura. Cada *anel* é um conjunto de chaves públicas, uma das quais pertence ao signatário, e as restantes são aleatórias. A assinatura é gerada com esse anel de chaves e com a mensagem $\mathfrak{m}$.

A nossa assinatura tipo Schnorr na secção 2.3.5 pode ser considerada uma assinatura em anel com uma só chave. Podemos extender este esquema, em vez de definir-mos r directamente, gera-se uma resposta desvio r' bem como um segundo desafio, de forma a definir r indirectamente. Note-se que á primeira vista o que se segue é confuso. Porque de facto existem dois aneís. O anel $\mathcal{R} = \{K_1, K_2, ..., K_n\}$, é incluído explícitamente na assinatura como parâmetro da função hash. Se $\mathcal{R}$ fosse público, seria possível ao inspector verificar a assinatura á mesma, pois as chaves em $\mathcal{R}$ seriam implícitas.

Á parte de $\mathcal{R}$, é também o esquema em sí da assinatura em anel, circular!

Seja $\mathfrak{m}$ a mensagem para ser assinada, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ um conjunto de chaves públicas distintas, e k_π a chave privada do signatário correspondente á respectiva chave pública $K_\pi \in \mathcal{R}$, em que π é o index secreto :

(a) Gera-se um número aleatório $\alpha \in_R \mathbb{Z}_l$ e respostas desvio $r_i \in_R \mathbb{Z}_l$ para todo o $i \in \{1, 2, ..., n\}$ excepto $i = \pi$.

(b) Calcula-se
$$c_{\pi+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [\alpha G])$$

(c) Para $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcula-se, substituindo $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$

(d) Define-se a resposta verdadeira r_π tal que $r_\pi = \alpha - c_\pi k_\pi \pmod{l}$, ou seja : $\alpha = r_\pi + c_\pi k_\pi \pmod{l}$.

A assinatura em anel contêm a assinatura $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, e o anel $\mathcal{R}$.

Verificação

Verificação significa provar que $\sigma(\mathfrak{m})$ é uma assinatura válida, creada por uma chave privada correspondente a uma chave pública em $\mathcal{R}$ (sem necessariamente saber qual), e é feito da seguinte maneira:

(a) Para $i = 1, 2, ..., n$ calcule, substituíndo $n + 1 \to 1$,

$$c'_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$

(b) Se $c_1 = c'_1$ então a assinatura é válida. Note-se que c'_1 é o último termo calculado.

Neste esquema guardam-se $(1+n)$ inteiros, e usam-se n chaves públicas.

Funciona porque

Podemos informalmente convencer-nos que o algoritmo funciona indo através de um exemplo. Considere-se um anel $R = \{K_1, K_2, K_3\}$ com $k_\pi = k_2$. Primeiro a assinatura :

(a) Gere-se números aleatórios α, r_1, r_3

(b) Inicia-se o anel :

$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [\alpha G])$$

(c) Calcula-se :

$$c_1 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_3 G + c_3 K_3])$$
$$c_2 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_1 G + c_1 K_1])$$

(d) Fecha-se o anel respondendo : $r_2 = \alpha - c_2 k_2 \pmod{l}$, ou seja : $\alpha = r_2 + c_2 k_2 \pmod{l}$

Substituimos então α em c_3 para reconhecer o conceito 'em anel'

$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [(r_2 + c_2 k_2) G])$$
$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_2 G + c_2 K_2\ ])$$

Depois a verificação usando $\mathcal{R}$, e $\sigma(\mathfrak{m}) = (c_1, r_1, r_2, r_3)$:

(a) usamos r_1 e c_1 para calcular

$$c'_2 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_1 G + c_1 K_1])$$

(b) incluindo c'_2, onde ao assinar estava c_2, obtêm-se :

$$c'_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_2 G + c'_2 K_2])$$

(c) e finalmente :

$$c'_1 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_3 G + c'_3 K_3])$$

Não há surpresas : $c'_1 = c_1$. Como tal a assinatura é válida.

3.4 Assinaturas ligadas espontâneas anónimas de grupo de Adam Back

Os esquemas de assinaturas em anel apresentados daqui em diante, contêm certas propriedades que irão ser úteis para produzir transacções confidenciais. [4] Note-se que 'inforjabilidade' e a 'ambiguidade do signatário' também se aplicam a assinaturas SAG.

Ambiguídade do signatário um observador deveria ser capaz de determinar que o signatário é membro do anel, mas incapaz de o identificar (enp). [5] Em Monero é assim que se obfusca a origem dos fundos em cada transacção.

Ligação Se uma chave privada é utilizada para assinar duas mensagens distintas, então essas mensagens estão ligadas. Esta propriedade não se aplica a chaves públicas não signatárias. Ou seja, um membro de mistura que tenha sido utilizado em diferentes assinaturas de anel, não causa ligações. Como iremos ver esta propriedade previne ataques de duplo gasto (enp).

Autenticidade Nenhum atacante consegue falsificar uma assinatura (enp). Certos esquemas de assinaturas em anel, incluindo esta aqui presente, são imunes a ataques adaptáveis perante a mensagem como também para a chave pública. Um atacante que obtenha assinaturas legítimas para certas mensagens correspondendo a certas chaves públicas em aneis, é incapaz de falsificar qualquer mensagem. Isto é chamado infalsificabilidade existencial, e é utilizado em Monero para previnir o roubo de fundos.

No esquema de assinatura LSAG [**?**], o proprietário de uma chave privada consegue produzir uma assinatura anónima para um anel. Nesse esquema a propriedade de *ligação* só se aplica a aneis que utilizem, para mensagens distintas, os mesmos membros e na mesma ordem. Nesta secção apresentamos uma versão melhorada do esquema LSAG em que a propriedade de *ligação* é independente dos membros do anel.

A modificação foi descoberta em [**?**] baseado numa publicação por Adam Back [**?**] em respeito ao algoritmo de assinaturas em anel, CryptoNote [**?**] (este algoritmo está agora descontinuado; veja-se a secção 8.1.2, que por sua vez foi inspirado por Fujisaki e Suzuki em [**?**]).

Assinatura

Como em SAG, seja $\mathfrak{m}$ a mensagem para assinar, o anel $\mathcal{R} = \{K_1, K_2, ..., K_n\}$, um conjunto de chaves públicas distintas, e k_π a chave privada do signatário e a correspondente chave pública $K_\pi \in \mathcal{R}$. Em que π é o index secreto. Assuma-se a existência de uma

[4] Note-se que todos os esquemas de assinatura robustos têm modelos de segurança que contêm diversas propriedades. As propriedades mencionadas aqui são talvez o mais relevantes para perceber o propósito das assinaturas em anel de Monero, mas não são uma visão geral das propriedades das assinaturas ligadas em anel.

[5] A anonimidade usualmente é em termos do 'conjunto de anonimidade', que é 'todas as pessoas que podiam ter agido de tal forma'. O maior conjunto de anonimidade é a 'humanidade', e em Monero é o tamanho de anel, ou o assim chamado *nível de mistura* v, mais o verdadeiro signatário. Expandir o conjunto de anonimidade torna mais e mais difícil conseguir encontrar o actor verdadeiro.

função hash $\mathcal{H}_p$ que dado um argumento qualquer, devolve como resultado um ponto de curva elíptica.

(a) Calcula-se a *imagem de chave* $\tilde{K} = k_\pi \mathcal{H}_p(K_\pi)$.[6]

(b) Gera-se um número aleatório $\alpha \in_R \mathbb{Z}_l$ e números aleatórios $r_i \in_R \mathbb{Z}_l$ para $i \in \{1, 2, ..., n\}$ mas excluíndo $i = \pi$.

(c) Calcula-se

$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha G], [\alpha \mathcal{H}_p(K_\pi)])$$

(d) Para $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcula-se, substituindo $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$

(e) Define-se $r_\pi = \alpha - c_\pi k_\pi \pmod{l}$.

A assinatura é $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, com a *imagem de chave* $\tilde{K}$ e anel $\mathcal{R}$.

Verificação

Verificação significa provar que $\sigma(\mathfrak{m})$ é uma assinatura válida creada por uma chave privada correspondente a uma chave pública presente em $\mathcal{R}$. Isto é feito da seguinte maneira :

(a) Verifique-se $l\tilde{K} \stackrel{?}{=} 0$.

(b) Para $i = 1, 2, ..., n$ calcule-se iterativamente, substituindo $n + 1 \to 1$,

$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$

(c) Se $c_1 = c'_1$ então a assinatura é válida.

Neste esquema são guardados (1+n) inteiros, com uma *imagem de chave* e n chaves públicas. Há que verificar que $l\tilde{K} \stackrel{?}{=} 0$ porque é possível adicionar um ponto de CE de um subgrupo de ordem h (o cofactor) a $\tilde{K}$ e, se todos os c_i forem múltiplos de h (o que seria conseguido depois de muitas tentativas com diferentes valores para α e r_i), fazem-se h assinaturas não-ligadas mas válidas que usa o mesmo anel e chave signatária. [7] Isto é porque um ponto de CE multiplicado pela ordem do seu subgrupo é igual a zero. [8]

src/cryptonote_core/cryptonote_core.cpp `check_tx_inputs_key-images_domain()`

Dado um ponto K no subgrupo de ordem l, um outro ponto K^h no subgrupo de ordem h, e um inteiro c divisível por h:

$$\begin{aligned} c * (K + K^h) &= cK + cK^h \\ &= cK + 0 \end{aligned}$$

[6] Em Monero é importante usar a função que mapeia hashes para pontos de CE, para as imagens de chave. Em vez de usar um outro ponto base, tal que a linearidade não leve á ligação de assinaturas, criadas pela mesmo endereço (mesmo se para endereços ocultos diferentes). Veja-se [**?**] page 18.

[7] Não nos preocupamos com pontos de outros subgrupos porque o resultado de $\mathcal{H}_n$ está confinado a $\mathbb{Z}_l$. Para a ordem de CE $N = hl$, todos os divisores de N, e como tal subgrupos possíveis, são múltiplos de l (número primo) ou divisores de h.

[8] No início de Monero isto não era verificado. Felizmente, isto não foi explorado, antes de uma correcção ter sido implementada em abril de 2017 (v5 do protocolo) [**?**].

Esta descrição tenta fazer juz á esplicação original de assinaturas bLSAG, que não incluêm $\mathcal{R}$ na hash que calcula c_i. Incluír chaves na hash é conhecido como 'prefixo de chave'. Pesquisas recentes sugerem que isto não é necessário, apesar de que adicionar o prefixo é uma práctica standard para esquemas de assinatura semelhantes (LSAG usa prefixo de chave).

Ligação

Dadas duas assinaturas válidas que são diferentes de alguma forma (por exemplo : mensagens distintas, membros de anel distintos)

$$\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n) \text{ com } \tilde{K}, \text{ and}$$
$$\sigma'(\mathfrak{m}') = (c'_1, r'_1, ..., r'_{n'}) \text{ com } \tilde{K}',$$

Se $\tilde{K} = \tilde{K}'$ Então ambas as assinaturas provêm da mesma chave privada.

Porque

$$\tilde{K} = k_\pi \mathcal{H}_p(K_\pi).$$

Um observador pode ver uma *ligação* entre σ e σ', mas ele não é capaz de saber qual K_i em $\mathcal{R}$ ou $\mathcal{R}'$, é o culpado. A não ser que só haja uma chave pública em comum entre $\mathcal{R}$ e $\mathcal{R}'$.

3.5 Assinaturas espontâneas anónimas ligadas de grupo com múltiplas camadas

Para assinar transacções, são utilizadas múltiplas chaves privadas. Em [**?**], Shen Noether *et al.* descreve uma generalisação do esquema de assinaturas bLSAG, aplicável ao conjunto de $n \cdot m$ chaves. Ou seja o conjunto

$$\mathcal{R} = \{K_{i,j}\} \quad \text{para} \quad i \in \{1, 2, ..., n\} \quad \text{e} \quad j \in \{1, 2, ..., m\}$$

Em que sabemos as m chaves privadas $\{k_{\pi,j}\}$ correspondentes ao subconjunto $\{K_{\pi,j}\}$ para algum index secreto $i = \pi$. Para que tal algoritmo satisfaca os nossos requisitos, geralizamos a noção de *ligação*.

Ligação Se qualquer chave privada $k_{\pi,j}$ é utilizada em duas assinaturas diferentes, então essas assinaturas estão *ligadas*.

Assinatura

(a) Calculam-se as imagens de chave $\tilde{K_j} = k_{\pi,j}\mathcal{H}_p(K_{\pi,j})$ para cada $j \in \{1, 2, ..., m\}$.

(b) Geram-se números aleatórios $\alpha_j \in_R \mathbb{Z}_l$, e $r_{i,j} \in_R \mathbb{Z}_l$ para cada $i \in \{1, 2, ..., n\}$ (excepto $i = \pi$) e para cada $j \in \{1, 2, ..., m\}$.

(c) Calcula-se :[9]

[9] As assinaturas MLSAG usam o *prefixo de chave*. Cada desafio contêm as chaves públicas esplícitas :
$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, K_{\pi,1}, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], ..., K_{\pi,m}, [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})])$$

(d) Para $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcula-se, substituindo $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K}_1], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K}_m])$$

(e) Definem-se todas as respostas : $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j} \pmod{l}$.

A assinatura é $\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m}, ..., r_{n,1}, ..., r_{n,m})$, com imagens de chave $(\tilde{K}_1, ..., \tilde{K}_m)$.

Verificação

A verificação de uma assinatura é feita da seguinte maneira :

(a) Para cada $j \in \{1, ..., m\}$ verifique $l\tilde{K}_j \stackrel{?}{=} 0$.

(b) Para $i = 1, ..., n$ compute, substituindo $n + 1 \to 1$,

$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K}_1], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K}_m])$$

(c) Se $c_1 = c'_1$ então a assinatura é válida.

Funciona porque

Da mesma forma que o algoritmo SAG, observamos que :

- Se $i \neq \pi$, então os valores c'_{i+1} são calculados como descrito no algoritmo de assinatura.
- Se $i = \pi$ então, desde que $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j}$ fecha o anel,

$$r_{\pi,j}G + c_\pi K_{\pi,j} = (\alpha_j - c_\pi k_{\pi,j})G + c_\pi K_{\pi,j} = \alpha_j G$$

e

$$r_{\pi,j}\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j = (\alpha_j - c_\pi k_{\pi,j})\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j = \alpha_j \mathcal{H}_p(K_{\pi,j})$$

Por outras palavras, mantém-se $c'_{\pi+1} = c_{\pi+1}$.

Ligação

Se alguma chave privada $k_{\pi,j}$ é re-utilizada para fazer qualquer assinatura, a correspondente *imagem de chave*, fornecida com a assinatura, revela a falsidade. Esta observação significa em sí, o conceito generalizado de *ligação*. Bem como em bLSAG, assinaturas MLSAG *ligadas* não indicam qual foi a chave pública usada como signatária [10].

Requisitos de espaço

Neste esquema guardam-se $(1+m*n)$ inteiros, m imagens de chave, e $m*n$ chaves públicas.

[10] Bem como em bLSAG, assinaturas MLSAG ligadas não indicam qual foi a chave pública, cuja chave privada fez a assinatura. Mesmo assim, se a mesma imagem de chave é usada em diferentes aneis, e estes aneis só têm uma chave pública em comum então o culpado é óbvio. Se o culpado é identificado, todos os outros membros signatários do anel de ambas as assinaturas são revelados visto que estes têm todos o mesmo índice secreto do culpado.

3.6 Assinaturas espontâneas anónimas ligadas de grupo concisas

CLSAG [?] [11] é uma espécie entre bLSAG e MLSAG. Existe uma chave privada *primária*, e associada com esta existem outras, chamadas de chaves privadas *auxiliares*. É importante provar a posse de todas as chaves privadas, e em termos de *ligação* só a chave privada *primária* é relevante. Em comparação com MLSAG estas assinaturas são mais curtas e mais rápidas. Temos então também um conjunto de $n \cdot m$ chaves (n é o comprimento de cada anel, m é o número de chaves signatárias).

$$\mathcal{R} = \{K_{i,j}\} \quad \text{para} \quad i \in \{1, 2, ..., n\} \quad \text{e} \quad j \in \{1, 2, ..., m\}$$

As chaves privadas *primárias* têm o index de $j = 1$.

Existem então para $j = 1$, n chaves *primárias*, em que assina a $\pi -$ ésima. E existem para $j > 1$, n chaves *auxíliares*, em que também assina a $\pi -$ ésima.

Temos posse do conjunto de chaves privadas $\{k_{\pi,j}\}$ pertencentes ao conjunto de chaves públicas $\{K_{\pi,j}\}$ para o index secreto $i = \pi$.

Assinatura

(a) Calcula-se *imagens de chave* $\tilde{K_j} = k_{\pi,j}\mathcal{H}_p(K_{\pi,1})$ para cada $j \in \{1, 2, ..., m\}$. Note-se que a chave base é sempre a mesma. Assim *imagens de chave* com $j > 1$ são *auxiliares*. Para simplificar a notação, *imagens de chave* representam-se por : $\tilde{K_j}$.

(b) Geram-se números aleatórios $\alpha \in_R \mathbb{Z}_l$, e $r_i \in_R \mathbb{Z}_l$ para cada $i \in \{1, 2, ..., n\}$ (excepto $i = \pi$).

(c) Calculam-se chaves públicas aggregadas W_i para cada $i \in \{1, 2, ..., n\}$, e uma *imagem de chave* aggregada $\tilde{W}$ [12]

$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K_1}, ..., \tilde{K_m}) * K_{i,j}$$

$$\tilde{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K_1}, ..., \tilde{K_m}) * \tilde{K_j}$$

em que $w_\pi = \sum_j \mathcal{H}_n(T_j, ...) * k_{\pi,j}$ é a chave privada aggregada.

(d) Compute

$$c_{\pi+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [\alpha G], [\alpha \mathcal{H}_p(K_{\pi,1})])$$

[11] Esta secção baseia-se num relatório preliminar que está a ser finalizado para uma revisão externa. CLSAG é promissor como um substituto de MLSAG em futuras versões do protocolo, mas ainda não foi implementado e talvez não venha a ser implementado no futuro.

[12] O abstracto CLSAG diz para utilizar funções hash diferentes, para existir a separação de domínios. O que é representado aqui por um *tag* [?], e.g. T_1 = "CLSAG_1", T_c = "CLSAG_c", etc. Cada função hash distinta terá naturalmente resultados distintos para argumentos iguais. A separação de domínios é um novo hábito no desenvolvimento de monero. Nas assinaturas são também fornecidos os *prefixos*, ou seja todas as chaves envolvidas na assinatura. De forma explícita o que é convenção comum (neste caso aqui incluíndo $\mathcal{R}$, o que contêm todas as chaves, e é argumento da função hash).

(e) Para cada $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcule, substituíndo $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [r_i G + c_i W_i], [r_i \mathcal{H}_p(K_{i,1}) + c_i \tilde{W}])$$

(f) Define-se $r_\pi = \alpha - c_\pi w_\pi \pmod{l}$.

Assim $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, com *imagem de chave* primária $\tilde{K}_1$, e *imagens de chave* auxiliares $(\tilde{K}_2, ..., \tilde{K}_m)$.

Verificação

A verificação de uma assinatura é feita da seguinte maneira :

(a) Para cada $j \in \{1, ..., m\}$ verifique $l\tilde{K}_j \stackrel{?}{=} 0$.[13]

(b) Calcule chaves públicas aggregadas W_i para cada $i \in \{1, 2, ..., n\}$, *imagem de chave* aggregada $\tilde{W}$

$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * K_{i,j}$$

$$\tilde{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * \tilde{K}_j$$

(c) Para cada $i = 1, ..., n$ compute, substituíndo $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [r_i G + c_i W_i], [r_i \mathcal{H}_p(K_{i,1}) + c_i \tilde{W}])$$

(d) Se $c_1 = c_1'$ então a assinatura é válida.

Funciona porque

O maior perigo em assinaturas concisas como esta, é o cancelamento de chave. Em que as *imagens de chave* dadas nao são legítimas. Porém a sua soma resulta num valor aggregado legítimo. é aqui que os coefficientes aggregados $\mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m)$ são relevantes. Pois cada chave pertence ao seu valor. As repercussões circulares que advêm de falsificar uma imagen de chave deixa-se como exercício ao leitor. Imagens de chave auxiliares são um artefacto para provar que a imagem de chave primária é legítima. Desde que a chave privada agregada w_π, que contêm todas as chaves privadas, é aplicada ao ponto base $\mathcal{H}_p(K_{\pi,1})$.

Ligação

Se uma chave privada $k_{\pi,1}$ é re-utilisada em qualquer assinatura, a *imagem de chave* $\tilde{K}_1$ correspondente, e fornecida com a assinatura irá revelá-lo. *Imagens de chave* auxiliares são ignoradas.

[13] Em Monero só se verifica $l * \tilde{K}_1 \stackrel{?}{=} 0$ para a imagen de chave primária. Chaves auxiliares seriam guardadas como $(1/8) * \tilde{K}_j$, e durante a verificação multiplicadas por 8 (secção 2.3.1), o que é mais eficiente. Mas isto seria uma opção de implementação.

Requisitos de espaço

Guardamos $(1+n)$ inteiros, temos m *imagens de chave*, e usamos $m*n$ chaves públicas.

CAPÍTULO 4

Endereços

A posse de moeda digital, guardada na lista de blocos, é definida pela posse de endereços. Somente o proprietário do endereço, pode gastar o valor digital alí contido (enp). Mais especificamente, para dadas transacções um *endereço* possuí certas *saídas*. Ou seja imagine-se o proprietário do endereço, com o direito a gastar valor digital. Para gastar de uma *saída*, o proprietário do endereço (que tem posse dessa saída), faz referência a essa saída como uma *entrada* numa nova transacção. Esta nova transacção têm então novas *saídas* controladas por novos endereços. É importante de notar que o proprietário de uma saída só a pode gastar uma vez. Numa transacção a soma do valor das *entradas* é igual á soma do valor das *saídas*. O montante de um endereço é a soma de todos os valores contidos nas *saídas* não gastas. Programas conhecidos como *carteiras* são então usados, para encontrar e organizar as saídas controladas por um endereço. Como também para manter a custódia das respectivas chaves privadas, criar novas transacções e submeter estas á rede de Monero. Para que finalmente essa transacção válida seja incluída num bloco.

4.1 Chaves

Um moneriano tem então dois pares de chaves privadas e públicas, (k^v, K^v) e (k^s, K^s). Estas chaves são geradas como descrito na Secção 2.3.2.

Um *endereço* é o par de chaves públicas (K^s, K^v).

Ao qual corresponde o par de chaves privadas (k^s, k^v).[1]

src/common/base58.cpp `encode_addr()`

[1] Para comunicar um endereço a outros utilizadores, é usual codificá-lo na base-58, um esquema de codificação para converter binário para texto. Este esquema foi criado para Bitcoin [**?**]. Veja-se [**?**] para mais detalhes.

Usar dois conjuntos de chaves permite ter funções distintas.
A chave privada k^s é a *chave de gasto*.
A chave privada k^v é a *chave de ver* .
Com a chave k^v é possível determinar se um endereço possuí uma saída.
E com a chave k^s, é então possível gastar essa saída numa nova transacção. [2]

4.2 Endereços ocultos

Para receber dinheiro, um moneriano pode distribuír o seu endereço a outros. Estes por sua vez enviam-lhe dinheiro contido nas *saídas* das transacções
(ou contido numa saída de uma transacção, no caso singular de ele só conhecer um outro moneriano. E de este só lhe enviar uma vez dinheiro numa transacção com só uma saída). [3]

Um endereço nunca está na lista de blocos.

Em vez disso é aplicada uma troca tipo Diffie-Hellman ao endereço, cada vez que se faz uma transacção para esse endereço. O que gera assim um *endereço oculto* único, para cada *saída* de transacção.
Desta forma, observadores externos, que conhecem todos os endereços de um certo moneriano, são incapazes de verificar quais são as *saídas* de transacções que lhe pertencem (enp).
Comecemos com o caso base, uma transacção simples da alice para o bob, contendo só uma *saída*, com o montante de '0.123' xmr.
O bob tem as chaves (k_B^s, k_B^v) e (K_B^s, K_B^v), e a alice sabe o seu endereço. Esta transacção prosegue da seguinte maneira :[**?**]:

(a) A alice gera um número aleatório $r \in_R \mathbb{Z}_l$, e calcula o *endereço oculto* : [4]

$$K^o = \mathcal{H}_n(rK_B^v)G + K_B^s$$

src/crypto/crypto.cpp `derive_public_key()`

[2] É actualmente comum para que a chave de ver k^v seja igual a $\mathcal{H}_n(k^s)$. Isto significa que uma pessoa só precisa de guardar a sua chave de gasto k^s para aceder ás saídas que possui (gastas e não gastas). A chave de gasto é típicamente representada como uma série de 25 palavras (em que a última palavra serve como soma de verificação). Outros métodos menos populares incluem: gerar k^v e k^s como números aleatórios separados, ou gerar uma mnemónica aleatória de 12 palavras a, em que $k^s = \texttt{sc\_reduce32}(Keccak(a))$ e $k^v = \texttt{sc\_reduce32}(Keccak(Keccak(a)))$. [**?**]

[3] O método descrito aqui não é enforcado pelo protocolo, só por implementações standard de carteira. Isto significa que uma carteira alternativa pode seguir o estilo de Bitcoin em que os endereços dos recipientes estão incluídos directamente nos dados de transacção. Tal carteira alternativa iria produzir saídas de transacção que não são usáveis por outras carteiras, e cada tal endereço tipo Bitcoin só poderia ser usado uma vez porque as imagens de chave têm de ser únicas.

[4] Em Monero cada instância de rk^vG é multiplicada pelo cofactor 8, assim neste caso a alice computa $8 * rK_B^v$ e o bob computa $8 * k_B^v rG$. Multiplicar pelo cofactor de 8 garante que o ponto resultante está no subgrupo de G. Se R e K^v não estão dentro do mesmo subgrupo, então os logaritmos discretos de r e k^v não podem ser usados para fazer um segredo partilhado.

(b) A alice pôe K^o como o destinatário do pagamento, adiciona o montante da *saída* '0.123' xmr, e o valor rG aos dados da transacção, e submete isso á rede monero.

(c) O bob recebe esses dados e vê rG e K^o. src/crypto/crypto.cpp `derive_subaddress_public_key()`
Ele pode calcular :

$$rGk_B^v = rK_B^v.$$

Ele depois calcula :

$$K_B'^s = K^o - \mathcal{H}_n(rK_B^v)G.$$

Ao ver que :

$$K_B'^s = K_B^s.$$

O bob sabe que essa saída lhe pertence.[5]
Note-se aqui que sabendo o endereço do bob, quem possuir também a chave privada k_B^v, pode calcular $K_B'^s$ para cada *saída* de transacção na lista de blocos, e ver quais pertencem ao bob.

(d) O *endereço oculto* e a *chave oculta* são :

$$K^o = \mathcal{H}_n(rK_B^v)G + K_B^s = (\mathcal{H}_n(rK_B^v) + k_B^s)G$$
$$k^o = \mathcal{H}_n(rK_B^v) + k_B^s$$

Para gastar o seu montante '0.123' numa nova transacção, o bob precisa de fazer uma assinatura com a sua chave privada oculta k^o. Sem esta chave a alice não têm a certeza se o bob gastou a saída que ela lhe enviou. [6]

O bob pode dar a sua *chave de ver* a qualquer entidade terceira, como por exemplo um auditor, entidade contabilística ou autoridade tributária etc... Ou seja a alguêm que tenha direitos de ler o seu histórico de transacções. Ao fazer isto o bob está a confiar que essa chave privada não vá parar ás maos erradas. Pois a chave de ver também permite ver o montante nas *saídas* de transacções. (O que será explicado na secção 5.3). Veja o capítulo 8 para outros meios com os quais o bob pode fornecer informações sobre o seu histórico de transacções.

[5] $K_B'^s$ é calculado com `derive_subaddress_public_key()` porque as chaves de gasto de endereços normais são guardados numa *tabela de chaves de gasto* com o index 0, enquanto que os sub-endereços começam com 1,2,3 ... isto irá fazer sentido brevemente : veja-se a secção 4.3.

[6] Imagine-se que a alice produz duas transacções, cada uma delas contêm o mesmo endereço oculto K^o que o bob pode gastar. Desde que K^o só depende de r e K_B^v, ela consegue de facto fazer isto se ela quisesse. O bob so pode gastar uma dessas saídas, porque cada endereço oculto só tem uma imagem de chave. A alice faz então uma transacção 1 com um montante enorme e uma transacção 2 com um montante pequeno. Se ele gasta o montante de 2, nunca poderá gastar o montante em 1. Aliás ninguêm mais pode gastar o dinheiro em 1, ou seja esse montante 'arde'. As carteiras em Monero foram feitas para ignorar o montante mais pequeno nesta situação.

4.2.1 transacções de múltiplas saídas

A maioria das transacções irá conter mais do que uma saída, por exepmlo duas saídas em que uma delas serve para que o remetente receba de volta o seu troco. [7] [8]

Os remetentes de Monero usualmente geram só um valor aleatório r.

O ponto de curva elíptica rG chama-se a *chave pública de transacção* e é publicada juntamente com outros dados dentro da transacção.

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

Para garantir que todos os endereços ocultos dentro de uma transacção com p saídas, são únicos mesmo quando um destinatário é usado duas vezes,
existe um *index de saída*.
Cada saída de transacção possui um *index* $t \in \{0, ..., p-1\}$.
Ao concatenar este valor, ao argumento da função hash, garante-se que o endereço oculto é único :

$$K_t^o = \mathcal{H}_n(rK_t^v, t)G + K_t^s = (\mathcal{H}_n(rK_t^v, t) + k_t^s)G$$
$$k_t^o = \mathcal{H}_n(rK_t^v, t) + k_t^s$$

src/crypto/crypto.cpp `derive_public_key()`

4.3 Sub-endereços

Monerianos podem gerar sub-endereços a partir de cada endereço príncipal.
Montantes enviados para um sub-endereço podem ser vistos e gastos usando a chaves de ver e de gasto do endereço principal [**?**]. Como analogia : uma conta online no banco pode ter vários saldos correspondentes a cartões de crédito e depósitos, e o proprietário da conta é capaz de gastar e ver todos esses saldos.

Sub-endereços são convenientes para receber fundos para o mesmo sítio quando um utilisador não quer que a sua actividade esteja ligada ao mesmo endereço. Como veremos, observadores externos teriam de quebrar o PLD para determinarem que um subendereço pertence a um endereço particular [**?**]. [9] Sub-endereços também são úteis para diferenciar entre saídas recebidas. Por exemplo, se a alice quer comprar uma maçã do bob numa terça-feira, o bob podia escrever um recibo descrevendo a compra, e fazendo um subendereço para esse recibo. A alice depois envia o dinheiro para esse sub-endereço, desta forma o bob consegue associar o dinheiro que ele recebeu com a maça que ele vendeu. Uma outra forma de explorar entre saídas recebidas irá ser explorada na próxima secção.

[7] Actualmente, cada transacção *requer* sempre duas saídas, mesmo que uma delas seja com o montante de 0 (`HF_VERSION_MIN_2_OUTPUTS`). Excepto as transacções de mineiro. Isto melhora a anonimidade das transacções pois elas parecem todas ainda mais iguais. Antes de v12 a carteira do software núcleo de Monero já criava saídas com um montante nulo. A implementação núcleo envia um montante de 0 para um endereço aleatório .

[8] Depois de *Bulletproofs* terem sido implementados no protocolo v8, cada transacção está limitada a um máximo de 16 saídas (`BULLETPROOF_MAX_OUTPUTS`). Antes disso não havia tal limite, aparte de um limite do tamanho da transacção (em bytes)

src/wallet/wallet2.cpp `transfer_selected_rct()`

[9] Antes do uso de sub-endereços (adicionado no update de software do protocolo v7 [**?**]), os monerianos podiam simplesmente gerar muitos endereços normais. E para ver o saldo de cada um desses endereços era necessário fazer um scan separado á lista de blocos. Isto era muito ineficiente, com sub-endereços, mantêm-se uma tabela hash das chaves de gasto. Assim um scan da lista de blocos leva o mesmo tempo para 1 sub-endereço, ou 10,000 sub-endereços.

[10]

[11] O bob gera o seu *i-ésimo* sub-endereço ($i = 1, 2, ...$) do seu endereço como um par de chaves públicas $(K^{s,i}, K^{v,i})$: [12]

$$K^{s,i} = K^s + \mathcal{H}_n(k^v, i)G$$
$$K^{v,i} = k^v K^{s,i}$$

Assim,

$$K^{s,i} = \quad (k^s + \mathcal{H}_n(k^v, i))G$$
$$K^{v,i} = k^v(k^s + \mathcal{H}_n(k^v, i))G$$

4.3.1 Enviar para um sub-endereço

Seja que a alice envia ao bob o montante 0.123 outra vez, desta vez para o seu sub-endereço $(K_B^{v,1}, K_B^{s,1})$.

(a) a alice gera um número aleatório $r \in_R \mathbb{Z}_l$, e calcula o endereço oculto :

$$K^o = \mathcal{H}_n(rK_B^{v,1}, 0)G + K_B^{s,1}$$

(b) A alice põem K^o como o destinatário do pagamento, adiciona o montante '0.123' e o valor $rK_B^{s,1}$ aos dados de transacção, e submete isso á rede.

(c) O bob recebe os dados e vê os valores :

$$rK_B^{s,1},\ K^o.$$

(d) Ele calcula então :

$$k_B^v rK_B^{s,1} = rK_B^{v,1}$$
$$K_B'^s = K^o - \mathcal{H}_n(rK_B^{v,1}, 0)G$$

(e) Quando ele vê que :

$$K_B'^s = K_B^{s,1},$$

então o bob sabe que a transacção é dirigida para ele.

O bob só precisa a sua chave privada de ver k_B^v e o sub-endereço público de gasto $K_B^{s,1}$ para encontrar saídas de transacção enviadas ao seu sub-endereço.

(f) As chaves ocultas para a saída são :

$$K^o = \mathcal{H}_n(rK_B^{v,1}, 0)G + K_B^{s,1} = (\mathcal{H}_n(rK_B^{v,1}, 0) + k_B^{s,1})G$$
$$k^o = \mathcal{H}_n(rK_B^{v,1}, 0) + k_B^{s,1}$$

[10] Antes de sub-endereços terem sido implementados em abril de 2018, os programadores de Monero suportavam um método chamado ID's de pagamento, o que permitia os recipientes de diferenciar entre as várias saídas recebidas. ID's de pagamento eram strings incluídas nos dados do campo *extra* de uma transacção. Os recipientes podiam pedir aos remetentes para incluir um ID de pagamento nos dados de transacção. ID's de transacção também podiam ser incluídos em endereços integrados. ID's de pagamento e endereços integrados não afectam a verificação da transacção. Portanto isso ainda pode ser implementado por qualquer pessoa, mas desde que isso está presentemente descontinuado nas carteiras mais populares escolheu-se não esplicar isso aqui.

[11] Note-se que o index i podia também simplesmente ser uma hash gerada de um password x tal que : $\mathcal{H}_n(x)$. Isto iria permitir que o proprietário de um endereço principal possa ver todos os seus montantes dos sub-endereços, mas só os consegue gastar ao escrever o password. Actualmente não existem carteiras que protejam os sub-endereços com passwords e também não existem planos para implementar tais carteiras.

[12] Acontece que a hash do sub-endereço está separada por domínios, portanto é realmente $\mathcal{H}_n(T_{sub}, k^v, i)$ em que T_{sub} ="SubAddr". Omite-se T_{sub} ao longo deste documento para brevidade.

A chave pública de transacção é particular ao bob e em vez de ser rG é :

$$rK_B^{s,1}.$$

Por outras palavras seja que a alice quer enviar numa só transacção dinheiro para p saídas, se pelo menos uma das saídas é para um sub-endereço, então é necessário fazer uma chave pública de transacção para cada saída.

Por exemplo a alice envia xmr para um sub-endereço $(K_B^{v,1}, K_B^{s,1})$ de bob e para um endereço (K_C^v, K_C^s) da carol, então ela põe $\{r_1 K_B^{s,1}, r_2 G\}$ nos dados de transacção. [13] [14]

4.4 Endereços integrados

Para diferenciar entre saídas recebidas, um destinatário pode requisitar dos remetentes que incluam um *ID de pagamento* nos dados de transacção. [15] Por exemplo se a alice quer comprar uma maçã ao bob numa terça-feira, o bob pode escrever um recibo que descreve a compra, e pedir a alice que inclua tal número de ID do recibo nos dados de transacção, quando ela lhe envia o dinheiro. Desta forma o bob consegue associar o dinheiro que recebe com a maça que vendeu.

A um dado ponto os remetentes podiam comunicar os ID's de pagamento em texto claro e não encriptado. Mas incluir os ID's de pagamento de forma manual é inconveniente, e texto claro é um perigo para a privacidade dos destinatários, que desta forma podem expor as suas actividades. [16] Em Monero, os destinatários podem integrar os ID's de pagamento nos seus endereços, e oferecer estes *endereços integrados*, que contêm : (K^s, K^v, ID), aos remetentes. Estes ID's de pagamento podem ser integrados em qualquer tipo de endereço, endereços normais, sub-endereços e endereços de multi-assinatura. [17]

Remetentes que enviam saídas para endereços integrados codificam ID's de pagamento usando o segredo partilhado rK_t^v e uma operação XOR (secção 2.5), os recipientes por

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_account_integrated_address_as_str()`

[13] Em Monero os sub-endereços levam o prefixo de 8 enquanto que os endereços levam o prefixo de 4. Isto ajuda aos remetentes escolherem o processo correcto ao construir transacções.

[14] Existem aspectos intrincados, com o uso de chaves públicas adicionais de transacção, ao redor de saídas de troco em que o autor da transacção sabe a chave de ver do destinatário (desde que é ele próprio; também o caso para saídas de troco em que o montante é nulo).
Sempre que existem pelo menos duas saídas que não são de troco, e pelo menos um dos destinatários é um sub-endereço, procede-se da froma normal como descrito acima (actualmente um erro na implementação núcleo adiciona uma chave pública de transacção, aos dados da transacção, o que é inútil).
Se somente a saída de troco é para um sub-endereço, ou só existe uma saída na transacção (sem troco) e esta é para um sub-endereço, então só se gera uma chave pública de transacção. No caso anterior, a chave pública de transacção é rG e o sub-endereço oculto de troco é : $K^o = \mathcal{H}_n(k_c^v rG, t)G + K_c^{s,1}$. O que usa a chave de ver normal e a chave de gasto do sub-endereço. No caso posterior a chave pública de transacção $rK^{v,1}$ baseia-se da chave de ver do sub-endereço, e o endereço oculto de troco é : $K^o = \mathcal{H}_n(k_c^v * rK^{v,1}, t)G + K_c^s$. Estes detalhes, ajudam a misturar uma porção de transacções de sub-endereços entre as transacções mais comuns. Transacções com duas saídas de endereços normais perfazem pelo menos 95% do volume de transacções.

[15] Actualmente, as implementações de Monero só permitem o suporte de um ID de pagamento por transacção, independentemente do número de saídas.

[16] Estes ID's de pagamento de longo formato (256 bits) foram descontinuados na v0.15 do software núcleo (o que coíncide com o protocolo v12 em Novembro de 2019). Enquanto que outras carteiras talvez ainda suportem a inclusão destes ID's nos dados de transacção, a carteira núcleo irá ignorá-los.

[17] Tanto quanto o autor saiba, endereços integrados só foram implementados para endereços normais.

sua vez descodificam isso com a chave pública de transacção e outra operação XOR [**?**]. Isto permite a remetentes provar que criaram certas saídas de transacção (e.g. para auditorias, reembolsos, etc...). [18]

Codificar

O remetente codifica o ID de pagamento para inclusão nos dados de transacção [19]

src/device/device_default.cpp `encrypt_payment_id()`

$$k_{\text{máscara}} = \mathcal{H}_n(rK_t^v, \text{pid\_tag})$$
$$k_{\text{ID}} = k_{\text{máscara}} \rightarrow \text{reduzido a 64 bits}$$
$$\text{ID codificado} = \text{XOR}(k_{\text{ID}}, \text{ID})$$

Incliu-se o pid_tag para garantir que k_{mask} é diferente do componente $\mathcal{H}_n(rK_t^v, t)$ em endereços ocultos. [20]

Descodificar

O destinatário usa a chave pública de transacção rG, e a sua chave de ver para encontrar um ID de pagamento. [21]

$$k_{\text{máscara}} = \mathcal{H}_n(k_t^v rG, \text{pid\_tag})$$
$$k_{\text{ID}} = k_{\text{máscara}} \rightarrow \text{reduzido a 64 bits}$$
$$\text{ID} = \text{XOR}(k_{\text{ID}}, \text{ID codificado})$$

[18] Desde que um observador consegue reconhecer a diferença entre transacções com e sem ID's de pagamento, pensa-se que usar ID's torna o histórico de transacções em Monero menos uniforme. Por causa disto, desde o protocolo v10 a implementação núcleo adiciona um ID de pagamento codificado desvio, para todas as transacções com duas saídas. Estas descodificadas revelam só zeros (isto não é requisito do protocolo, somente melhor prática).

[19] Em Monero os ID's de pagamento usualmente são 64 bits.

[20] Em Monero, pid_tag = `cauda_do_ID_codificado` = 141.

[21] Os dados de transacção não explicitam para qual saída pertence um ID de pagamento. Os destinatários têm de identificar os seus próprios ID's

CAPÍTULO 5

Montantes ocultos

Em outras cripto-moedas, as saídas de transacções, são comunicadas de forma clara e transparente. Isto permite a observadores terçeiros verificar que as entradas das transacções são igauis ás saídas (menos a taxa aos mineiros). Monero utiliza *compromissos* para ocultar os montantes de quaisquer pessoas, excepto do remetente e destinatário. Enquanto que ao mesmo tempo, terceiros verificam que uma transacção não gasta nem mais nem menos, do que é verdade.

Como iremos ver, *compromissos a montantes* requerem também *provas de domínio*, o que garante que o montante oculto está dentro de um certo domínio.

5.1 Compromissos

Um esquema de compromisso criptográfico, é em termos gerais, uma forma de comprometer a um certo valor sem ter de o revelar.

Por exemplo, dois monerianos numa tirada de cara ou coroa apostam um xmr de forma oculta. A alice escolhe uma palavra secreta e junta esta á palavra "cara". Ela depois calcula a hash desse par :

$$h = \mathcal{H}(segredo, "cara").$$

E compromete-se a h ao mundo, neste caso bob. O bob atira uma moeda ao ar. Depois de saír cara ou coroa, a alice revela a palavra secreta. O bob faz duas hashes $\mathcal{H}(segredo, "cara")$ e $\mathcal{H}(segredo, "coroa")$ e deduz o compromisso da alice.

5.2 Compromissos de Pedersen

Um compromisso de Pedersen tem a propriedade de ser *additivamente homomórfico*. Se $C(a)$ e $C(b)$ são os compromissos respectivos aos valores a e b, então :

$$C(a+b) = C(a) + C(b).$$

Felizmente, compromissos de pedersen são fáceis de implementar com criptografia de curva elíptica. O seguinte é trivial :

$$(a+b)G = aG + bG$$

Ao definir compromissos tão simples como $C(a) = aG$, era possível ter tabelas de valores comuns para a. Ou seja seria fácil de deduzir para que a, se tinha comprometido.

Para atingir privacidade ao nível da teoria da informação, é necessário adicionar um segredo, o *factor ofuscante* e um outro gerador H .

O gerador H é escolhido, tal que é desconhecido para que valor de γ seja o seguinte :

$$H = \gamma G$$

A dificuldade do problema do logaritmo discreto, garante que calcular γ de H não é fazível. Em Monero : src/ringct/rctTypes.h

$$H = 8 * to\_point(\mathcal{H}_n(G)).$$

Note-se que nenhum moneriano conhece γ, e se alguêm descobre γ, então é capaz de forjar xmr (γ diz-se: "gama"). Ou seja sabendo γ é possivel roubar xmr não existentes nas entradas de transacção.

Define-se um compromisso a um montante b, de saída como :

$$C(y,b) = yG + bH$$ src/ringct/rctOps.cpp `addKeys2()`

Em que y é o *factor ofuscante*, que previne a observadores deduzirem b.

O compromisso $C(x,b)$ é privado ao nível da teoria da informação porque existem muitas possíveis combinações de x e b que resultam no mesmo $C(x,b)$.
Existem muitos x' e b', tal que :

$$x' + b'\gamma = x + b\gamma$$

Quem se compromete conhece uma combinação, mas um atacante é incapaz de saber qual. Se x é verdadeiramente aleatório, um atacante não teria literalmente maneira de descobrir b [**?**, **?**].

Quem se compromete é incapaz de encontrar uma outra combinação, sem resolver o PLD para γ, esta propriedade diz-se *vínculo perfeito*.

5.3 Montantes ocultos

Um montante numa saída controlada por um endereço em Monero, está presente de duas formas numa transacção. Primeiro existe como montante oculto, somente visível para o remetente e destinatário. Segundo existe como compromisso de Pedersen, para os mineiros.

Antes de mais, os destinatários deviam ser capazes de saber quanto dinheiro está em cada saída que lhes pertence. Ou seja o montante b e o factor ofuscante y têm de ser comunicados ao recipiente.

A solução adoptada é um segredo comum tipo Diffie-Hellman rK_B^v que usa a 'chave pública de transacção'. Para cada transacção de Monero, cada saída $t \in \{0, ..., p-1\}$ tem um *factor ofuscante* y_t que remetentes e destinatários podem calcular privadamente. E também um montante b_t.

Enquanto que y_t é um ponto de curva elíptica que ocupa 32 bytes, b_t ocupa somente 8 bytes. [1]

src/ringct/rctOps.cpp `ecdh-Encode()`

$$y_t = \mathcal{H}_n(\text{“}factor\ ofuscante\text{”}, \mathcal{H}_n(rK_B^v, t))$$
$$montante_t = b_t \oplus_8 \mathcal{H}_n(\text{“}montante\text{”}, \mathcal{H}_n(rK_B^v, t))$$

Em que $\oplus_8$ significa a operação XOR (Secção 2.5) entre os primeiros 8 bytes de cada operando.

A remetente alice é capaz de calcular o *factor ofuscante* y_t e o $montante_t$ usando a 'chave privada de transacção' r e a *chave de ver* pública K_B^v do bob.

O destinatário bob é capaz de calcular o *factor ofuscante* y_t usando a 'chave pública de transacção' rG e a sua *chave de ver* privada k_B^v.

A alice incluí o $montante_t$ oculto, nos dados de transacção, como tal o destinatário bob pode calcular b_t ao executar a operação XOR invertida :

$$b_t = montante_t \oplus_8 \mathcal{H}_n(\text{“}montante\text{”}, \mathcal{H}_n(rK_B^v, t))$$

Note-se que o $montante_t$ é o valor b_t encriptado, e seguro em termos da teoria da informação.

Neste caso, trata-se de um valor entre 0 e $2^{64}-1$, e para quem não tenha a *chave de ver* privada k_B^v, não sabe qual desses valores é o verdadeiro. E ter um poder de computação infinito também não serve de algo.

[1] Esta solução (implementada em v10 do protocolo) substitui um método anterior que usava mais dados, o que fez com que o tipo de transacção mudasse de v3 (`RCTTypeBulletproof`) a v4 (`RCTTypeBulletproof2`). A primeira edição deste relatório discutiu o método v3 [**?**].

5.4 Introdução a RingCT

Primeiro uma transacção contêm como entradas, saídas de outras transacções anteriores. E segundo, as suas próprias saídas.
Uma saída de transacção contêm um endereço oculto e um compromisso de saída. O endereço oculto serve como propriedade. O compromisso de saída serve para esconder o montante. Quem verifica as transacções não sabe quanto dinheiro está contido nas entradas e nas saídas.

Para construír uma transacção é preciso primeiro provar que o montante em cada entrada é igual ao montante da saída anterior correspondente.

Seja uma transacção composta por m entradas com montantes $a_1, ...a_m$, em que j é o indíce de entrada $j \in \{1, ...m\}$. Cada saída anterior tem como tal o seguinte compromisso ao montante a :

$$C_j^a = x_j G + a_j H$$

O remetente cria um novo compromisso ao mesmo montante com um factor ofuscante diferente :

$$C_j'^a = x_j' G + a_j H$$

Seja $C_j'^a$ um *pseudo* compromisso de saída, estes estão incluídos nos dados de transacção. Existe um para cada entrada.
O remetente sabe a chave privada da diferenca entre os dois compromissos :

$$C_j^a - C_j'^a = x_j G + a_j H - (x_j' G + a_j H)$$
$$C_j^a - C_j'^a = x_j G - x_j' G$$
$$C_j^a - C_j'^a = (x_j - x_j')G$$

Esta chave :

$$(x_j - x_j') = z_j$$

é utilizada como um *compromisso a zero* .

Ou seja, para provar que não existe um componente H na soma :

$$C_j^a - C_j'^a = z_j G + 0H,$$

o que implica que os montantes de ambos os compromissos são iguais, faz-se uma assinatura com z_j. Isto será feito em maior detalhe no capítulo 6.
Seja que a mesma transacção tenha p saídas com montantes $b_1, ..., b_p$. O remetente cria então para a sua actual transacção um compromisso de saída, para cada montante b_t:

$$C_t^b = y_t G + b_t H$$

É então de esperar que :

$$\sum_{j=1}^{m} a_j - \sum_{t=1}^{p} b_t = 0$$

Outra *prova necessária* na construção de uma transacção em Monero, é tal que a soma dos pseudo compromissos menos a soma dos compromissos das saídas tem de ser igual a zero. Isto é possível pois os compromissos são *additivamente homomórficos* e não se sabe γ.

$$\sum_{j=1}^{m} C_j'^a - \sum_{t=1}^{p} C_t^b = 0$$

Note-se que os pseudo compromissos e os compromissos das saídas são visíveis ao público, ou seja aos mineiros. Se a equação anterior é igual a zero então os mineiros sabem que a dada transacção não gasta nem mais nem menos do que deve ser. Na realidade, isto só é possível se a transacção não paga nenhuma taxa ao mineiro (veja-se a secção 6.1.1).

Os factores ofuscantes para *compromissos de saída* (pseudo e normais), são seleccionados tal que :

$$\sum_{j=1}^{m} x'_j - \sum_{t=1}^{p} y_t = 0$$

Isto permite provar que os montantes de entrada são iguais aos montantes de saída.

Felizmente, escolher tais factores ofuscantes é fácil. Na versão actual de Monero, todos os factores ofuscantes são aleatórios. Excepto o *m-ésimo pseudo compromisso de saída*, em que :

genRctSimple()

$$x'_m = \sum_{t=1}^{p} y_t - \sum_{j=1}^{m-1} x'_j$$

5.5 Provas de Domínio

Um problema com compromissos additivos é o de valores negativos. Sejam $C(a_1)$, $C(a_2)$, $C(b_1)$, $C(b_2)$ e pretende-se provar que :

$$(a_1 + a_2) - (b_1 + b_2) = 0$$

Uma solução é :

$$a_1 = 6, a_2 = 5, b_1 = 21, and b_2 = -10$$

$$(6 + 5) - (21 + -10) = 0$$

em que

$$21G + -10G = 21G + (l - 10)G = (l + 11)G = 11G$$

Visto que $-10 = l - 10$, efectivamente foram creados l mais Moneroj do que existiam nas entradas (mais de $7.2\text{x}10^{74}$ xmr).

A solução para isto é provar que cada montante de saída está dentro de um certo

domínio (de 0 a $2^{64} - 1$). O esquema de provas que é utilizado chama-se *Bulletproofs* (Benedikt Bünz *et al.*[**?**]).

5.6 Assinaturas em anel Borromean

Seja o seguinte compromisso ao montante b :

$$C_a = xG + bH$$

Primeiro faz-se uma partição aleatória de x tal que :

$$x = \sum_{i=0}^{k-1} x_i$$

Segundo descreve-se b em termos binários :

$$b = \sum_{i=0}^{k-1} b_i 2^i$$

O compromisso C_a define-se então como :

$$C_a = \sum_{i=0}^{k-1} C_i = \sum_{i=0}^{k-1} x_i G + b_i 2^i H$$

O objectivo aqui é de provar que o compromisso C_a é válido. Mas C_a, só por si não prova algo pois é somente um ponto de CE. A prova de domínio advém de que cada C_i satisfaz um certo constrangimento na assinatura to tipo *Borromean*.
Se cada C_i é :

$$C_i = x_i G + b_i 2^i H$$

então, se $b_i = 1$:

$$x_i G = C_i - b_i 2^i H$$

e se $b_i = 0$:

$$x_i G = C_i$$

Como tal, a chave privada que é usada é o escalar x_i, e o seu ponto de CE $x_i G$. Há que provar este contrangimento sem revelar b_i.

Para isso fixa-se C_i e $C_i - 2^i H$, como argumentos para a prova de domínio.
Seja que $b_i = 0$, então o caso base é:

$$\begin{aligned}
\alpha &= r() \\
\beta_1 &= r() \\
\omega &= H_s(\alpha G) \\
\psi_g &= \beta_1 G + \omega(\mathbf{C_i} - \mathbf{2^i H}) \\
\Psi_g &= H_s(\psi_g) \\
\lambda &= r() \\
\beta_0 &= \alpha - x_i * \lambda
\end{aligned}$$

agora passa-se Ψ_g, β_0, β_1 e λ para o verificador.

$$\begin{aligned}
\mu &= \beta_0 G + \lambda \mathbf{C_i} \\
\mu &= (\alpha - x_i * \lambda) G + \lambda \mathbf{C_i} \\
\mu &= \alpha G - \lambda(x_i G) + \lambda \mathbf{C_i} \\
\mu &= \alpha G \\
\theta &= H_s(\mu) \\
\psi_c &= \beta_1 G + \theta(\mathbf{C_i} - \mathbf{2^i H}) \\
\Psi_c &= H_s(\psi_c)
\end{aligned}$$

Note-se que Ψ_g é igual a Ψ_c e como tal a prova é válida.
Em que $r()$ é um escalar aleatório no corpo finito $\mathbb{F}_l$. E $H_s(\alpha G)$ é um certo escalar aleatório, que resulta do argumento dado á função *hash* $H_s(\alpha G)$ (aqui αG, H_s aceita qualquer tipo de dados como argumento). O operador binário * é uma multiplicação entre dois escalares no corpo finito $\mathbb{F}_l$. E αG por exemplo é adicionar o ponto de CE G, consigo próprio α vezes.

Seja que $b_i = 1$, então o caso base é:

$$\begin{aligned}
\alpha &= r() \\
\psi_g &= \alpha G \\
\Psi_g &= H_s(\psi_g) \\
\lambda &= r() \\
\beta_0 &= r() \\
\phi &= \beta_0 G + \lambda \mathbf{C_i} \\
\epsilon &= H_s(\phi) \\
\beta_1 &= \alpha - x_i * \epsilon
\end{aligned}$$

agora passa-se Ψ_g, β_0, β_1 e λ para o verificador.

$$\begin{aligned}
\mu &= \beta_0 G + \lambda \mathbf{C_i} \\
\theta &= H_s(\mu) \\
\psi_c &= \beta_1 G + \theta(\mathbf{C_i} - \mathbf{2^i H}) \\
\psi_c &= (\alpha - x_i * \epsilon)G + \theta(\mathbf{C_i} - \mathbf{2^i H}) \\
\psi_c &= \alpha G - \epsilon(x_i G) + \theta(\mathbf{C_i} - \mathbf{2^i H}) \\
\psi_c &= \alpha G - H_s(\beta_0 G + \lambda \mathbf{C_i})(x_i G) \\
&\quad + H_s(\beta_0 G + \lambda \mathbf{C_i})(\mathbf{C_i} - \mathbf{2^i H}) \\
\psi_c &= \alpha G \\
\Psi_c &= H_s(\psi_c)
\end{aligned}$$

Note-se que Ψ_g é igual a Ψ_c e como tal a prova é válida.

As provas apresentadas funcionam, porque :

os argumentos precalculados, $C_i - 2^i H$ e C_i só permitem que b seja 1 ou 0. Á parte de que a geração de uma assinatura deste tipo só se define para que b seja um bit. Mesmo que o algoritmo seja alterado para tentar fazer assinaturas com outros valores para b, as equações dadas não se resolveriam na parte da verificação. Quem verifica sabe de certeza que o provador usou um bit mas não sabe se esse bit tem o valor de 0 ou 1.

Para mais, este caso base cobre todos os índices de um vector de 64 bits. visto que a verificação é sempre a mesma :

$$\begin{aligned}
\mu &= \beta_0 G + \lambda C_i \\
\theta &= H_s(\mu) \\
\psi_c &= \beta_1 G + \theta(C_i - 2^i H) \\
\Psi_c &= H_s(\psi_c)
\end{aligned}$$

O bob podia gerar cada C_i e repetir o processo descrito em cima. A alice para cada

C_i gerava $C_i - 2^i H$, recebia também os argumentos para cada prova Ψ_g, β_0, β_1 e λ, e verificava que a prova é válida. A alice sabe portanto que :

$$C_a = \sum_{i=0}^{k-1} C_i$$

e tem a certeza que b é um vector de 64 bits, ou seja que o montante está dentro do domínio de 0 a $2^{64} - 1$. Estas provas constituem uma assinatura em anel, mas não se trata própriamente de um anel, não é um processo circular. Para mais veja-se [**?**] .[2]

É aparente que 2^i pode ser qualquer $k_i > 0 \in \mathbb{F}_l$. Seja que :

$$b = \sum_{i=1}^{m} b_i k_i$$

Os termos de 2^i são conhecidos pelo verificador de antemão, como tal não são transmitidos pelo provador. Se fosse k_i em vez de 2^i, o provador teria de enviar mais m escalares em $\mathbb{F}_l$. pois então o constrangimento passaria a ser :
Se cada C_i é :

$$C_i = x_i G + b_i k_i H$$

então, se $b_i = 1$:

$$x_i G = C_i - b_i k_i H$$

e se $b_i = 0$:

$$x_i G = C_i$$

De seguida faz-se exactamente o mesmo processo descrito em cima, e as provas acontecem á mesma. Em termos de 2^i, com 64 bits o valor máximo é $(2^{64}) - 1$. Em que cada montante b nesse domínio é equiprovável. Ou seja existem 2^{64} combinações distintas possíveis. No caso de serem m escalares, o constrangimento adicional teria de ser válido :

$$\sum_{i=1}^{m} k_i \leq (2^{64}) - 1$$

[2] No código em *python*, evita-se enviar λ, e usa-se Ψ_g como parámetro para a verificação final, bem como variável supostamente aleatória! Ou seja Ψ_g é utilizado para calcular Ψ_c (através de bbs0 e bbs1) como resultado final... O que não deixa de ser elegante!

Se isto não fosse o caso o verificador nem se dava ao trabalho de olhar para as provas. E o número de combinações distintas possíveis seria 2^m. Se $m = 2$, imagine-se $k_1 = 2$ e $k_2 = 3$. O bob envia 5 xmr para a alice e quem verifica só sabe que o montante enviado é zero, dois, trés ou cinco. O que seria mau em termos de privacidade.

5.7 Bulletproofs

Sou mais rápido quando me mexo.
Sundance

Em português : provas de bala.

Á partida não se percebe sequer como é que alguém chega a descobrir o algoritmo descrito anteriormente, das assinaturas borromeanas. Não se esplica aqui também todos os vectores de ataque contra tais assinaturas ou se as assinaturas borromeanas para o seu custo de comunicação, são as mais compactas. O leitor que pensa portanto como é que se chega a tal coisa? Ou também : "não se pode forjar tais assinaturas?"permanece sem explicação dada. Não só isso mas mais, o que se segue é muito mais complicado e será explicado em termos verbatim. Ou seja quem segue o método consegue compreender o porque dele funcionar, mas não como é que se chega a tal coisa, nem como se forja uma tal prova, ou onde ele poderá falhar se implementada mal.

Se v está entre 0 e $2^{64} - 1$

Primeiro passa-se v para a sua representação binária :

$\underline{a}_L \leftarrow bits$ v

Ou seja :

$v = \langle \underline{a}_L, \underline{2}^n \rangle$

em que $\underline{2}^n = \{2^0, 2^1, 2^2, ..., 2^{n-1}\}$ Depois põe-se dois constrangimentos adicionais :

$$v = \langle \underline{a}_L, \underline{2}^n \rangle$$
$$\underline{a}_R = \underline{a}_L - \underline{1}$$
$$\underline{a}_L \circ \underline{a}_R = 0$$

Depois faz-se o seguinte :

$$v = \langle \underline{a}_L, \underline{2}^n \rangle$$
$$0 = \langle \underline{a}_L - \underline{1} - \underline{a}_R, \underline{y}^n \rangle$$
$$0 = \langle \underline{a}_L \circ \underline{a}_R, \underline{y}^n \rangle$$

Note-se que isto são tudo afirmações equivalentes, mas a introdução de $\underline{y}^n$, já é : "excepto uma negligível probabilidade de falhar"(enp). Porque $\underline{y}^n = \{y^0, y^1, y^2, ..., y^{n-1}\}$, com qualquer $\underline{c} = \{c_0, c_1, c_2, ...c_{n-1}\}$ (neste caso $\underline{c} = \underline{a}_L - \underline{1} - \underline{a}_R$ ou $\underline{c} = \underline{a}_L \circ \underline{a}_R$) constitui o polinómio :

$$F(y) = c_0 y^0 + c_1 y^1 + c_2 y^2 + ... + c_{n-1} y^{n-1}$$

E este polinómio tem no máximo $n - 1$ raízes ($F(y) = 0$). Em que $\underline{c} = \underline{0}$ é uma delas. A probabilidade de um atacante encontrar outro $\underline{c}'$ tal que $F(y) = 0$ é : n/l, e como tal

negligível.

Continua-se então a representar o mesmo de outra forma ainda :

$$\begin{aligned} vz^2 &= \langle \underline{a}_L, \underline{2}^n \rangle z^2 \\ 0z &= \langle \underline{a}_L - \underline{1} - \underline{a}_R, \underline{y}^n \rangle z \\ 0 &= \langle \underline{a}_L \circ \underline{a}_R, \underline{y}^n \rangle \end{aligned}$$

o que depois de muita manipulação resulta na seguinte equação em que $\underline{a}_L$ e $\underline{a}_R$ se encontram de lados opostos no produto interno :

$$z^2 v + \delta(y,z) = \langle \underline{a}_L - \underline{1}z, \underline{y}^n \circ (\underline{a}_R + \underline{1}z) + z^2 \underline{2}^n \rangle \tag{5.1}$$

$$\delta(y,z) = (z - z^2)\langle \underline{1}, \underline{y}^n \rangle - z^3 \langle \underline{1}, \underline{2}^n \rangle$$

Para ver como se chega ás equações anteriores veja-se o apêndice D. Enfim á primeira vista o que se conseguio até agora foi representar que v está constrangido segundo ás trés regras iniciais, de uma forma altamente elaborada.
O provador não pode enviar o lado direito da equação (5.1) ao verificador senão este fica a saber os bits de v.
Como tal há que obfuscar $\underline{a}_L$ e $\underline{a}_R$:

$(\underline{a}_L + \dot{\underline{s}}_L) - \underline{1}z \quad *^5$
$\underline{y}^n \circ ((\underline{a}_R + \dot{\underline{s}}_R) + \underline{1}z) + z^2 \underline{2}^n \quad *^6$
Note-se que o que se alcanço agora foi algo diferente, tanto que :

$$z^2 v + \delta(y,z) \neq \langle (\underline{a}_L + \dot{\underline{s}}_L) - \underline{1}z, \underline{y}^n \circ ((\underline{a}_R + \dot{\underline{s}}_R) + \underline{1}z) + z^2 \underline{2}^n \rangle$$

Define-se então o conseguido anteriormente como :
$\underline{l}_0 = \underline{a}_L - \underline{1}z$
$\underline{r}_0 = \underline{y}^n \circ (\underline{a}_R + \underline{1}z) + z^2 \underline{2}^n$
ou seja :

$$z^2 v + \delta(y,z) = \langle \underline{l}_0, \underline{r}_0 \rangle = t_0$$

e também os factores ofuscantes :
$\underline{l}_1 = \dot{\underline{s}}_L$
$\underline{r}_1 = \underline{y}^n \circ \dot{\underline{s}}_R$
força-se então novas igualdades : $*^2 *^3$
$\underline{l}(x) = \underline{l}_0 + \underline{l}_1 x$
$\underline{r}(x) = \underline{r}_0 + \underline{r}_1 x$
Finalmente : $*^4$

$$t(x) = \langle \underline{l}(x), \underline{r}(x) \rangle = t_0 + t_1 x + t_2 x^2$$

O que se obtêm como tal, é um polinómio que em t_0 se compromete a V. O que o provador faz é enviar um número de variáveis ao verificador, que fazem parte de um circuito aritmético. Em que o conjunto dessas variáveis, nesse sistema, implica que v satisfaz a condição do domínio requerido.

calculam-se também os coefficientes de t(x): $*^1$

$$t(x) = \langle(\underline{l}_0 + \underline{l}_1 x), (\underline{r}_0 + \underline{r}_1 x)\rangle = \langle \underline{l}_0, \underline{r}_0\rangle + \langle \underline{l}_0, \underline{r}_1 x\rangle + \langle \underline{l}_1 x, \underline{r}_0\rangle + \langle \underline{l}_1 x, \underline{r}_1 x\rangle$$

$t_0 = \langle \underline{l}_0, \underline{r}_0\rangle$
$t_1 x = \langle \underline{l}_0, \underline{r}_1 x\rangle + \langle \underline{l}_1 x, \underline{r}_0\rangle = \langle \underline{l}_0, \underline{r}_1\rangle x + \langle \underline{l}_1, \underline{r}_0\rangle x = (\langle \underline{l}_0, \underline{r}_1\rangle + \langle \underline{l}_1, \underline{r}_0\rangle)x$
$t_2 x^2 = \langle \underline{l}_1, \underline{r}_1\rangle x^2$

G, H, $\underline{G}$, $\underline{H}$ ${}^{0}\perp[init]$

$\mathrm{V} = \gamma G + vH$ ${}^{1}\perp_U(V)$

$\underline{\mathrm{a}}_L \leftarrow bits$ v

$\underline{a}_R = \underline{a}_L - \underline{1}$

$\dot{\alpha} \leftarrow R()$

$A = \langle \underline{G}, \underline{a}_L \rangle + \langle \underline{H}, \underline{a}_R \rangle + \alpha G$ ${}^{2}\perp_U(A)$

$\dot{\rho} \leftarrow R()$

$\underline{\dot{s}}_L \leftarrow R()$

$\underline{\dot{s}}_R \leftarrow R()$

$S = \langle \underline{G}, \underline{\dot{s}}_L \rangle + \langle \underline{H}, \underline{\dot{s}}_R \rangle + \dot{\rho} G$ ${}^{3}\perp_U(S)$

y$\leftarrow {}^{4}\perp_C()$, y^{-1}

$z \leftarrow {}^{5}\perp_C()$

$$(t_0, t_1, t_2)*^{1}$$

$\dot{\tau_1} \leftarrow R()$

$\dot{\tau_2} \leftarrow R()$

$T_1 = t_1 H + \dot{\tau_1} G$ ${}^{6}\perp_U(T_1)$

$T_2 = t_2 H + \dot{\tau_2} G$ ${}^{7}\perp_U(T_2)$

$x \leftarrow {}^{8}\perp_C()$

$\tau_x = \dot{\tau_1} x + \dot{\tau_2} x^2 + \gamma z^2$ ${}^{9}\perp_U(\tau_x)$

$\mu = x\dot{\rho} + \dot{\alpha}$ ${}^{10}\perp_U(\mu)$

$$(\underline{l}(x) = \underline{l}_0 + \underline{l}_1 x)*^{2}$$
$$(\underline{r}(x) = \underline{r}_0 + \underline{r}_1 x)*^{3}$$

$$t(x) = \langle \underline{l}(x), \underline{r}(x) \rangle = (t_0 + t_1 x + t_2 x^2)*^{4}$$

${}^{11}\perp_U(t(x))$

$x_{ip} \leftarrow {}^{12}\perp_C()$

$U = x_{ip} H$

IP$\leftarrow$\{G,H',U,l(x),r(x)\}

o símbolo $\perp$ significa um traslado. Ou seja durante este processo sequencial, este traslado devolve hashes. Estas por sua vez são transformadas em escalares ou pontos de CE. Em que $\perp_U()$ é renovar o estado do traslado e $\perp_C()$ devolve um desafio. O j em ${}^{j}\perp$ marca o estado do traslado, como base, no tempo. Vectores são sublinhados como $\underline{a}$ para escalares ou $\underline{G}$ para pontos de CE. Para simbolizar variáveis livres usa-se um ponto, acima da letra($\dot{\alpha}$, $\dot{S}_L$, ...), que recebe um escalar ou ponto de CE aleatório da função R(). Estes valores como sendo livres, têm que ser enviados para o verificador. Ao invés de variáveis como $\underline{G}$, que apesar de serem aleatórias, são valores públicos e bem definidos, e como tal podem e são precalculados pelo verificador de forma independente. Note-se que os vectores têm sempre 64 bits. O protocolo do produto interno, IP, reduz

o tamanho dos vectores á ordem de $\log_2(64)$. E para isso acontecer, o provador constroi os seguintes circuitos aritméticos, em que o primeiro trata do lado esquerdo da equação (5.1) e o segundo do lado direito :

$$\begin{array}{ccccccccc} t(x)H & = & z^2vH & + & \delta(y,z)H & + & t_1xH & + & t_2x^2H \\ + & & + & & + & & + & & + \\ \dot{\tau}_xG & = & \gamma z^2G & + & 0G & + & \dot{\tau}_1xG & + & \dot{\tau}_2x^2G \\ \| & & \| & & \| & & \| & & \| \end{array}$$

$$= \underbrace{z^2V + \delta(y,z)H}_{\text{público}} + xT_1 + x^2T_2$$

As mentes ávidas ao lerem isto dizem logo :"Ah e tal mas T_1 e T_2 pode ser qualquer coisa, até porque está ali o $\dot{\tau}_x$, isso são tudo variáveis aleatórias... Como é que isso tem alguma expressão sobre v?" Primeiro t(x) não pode ser directamente igual a t_0, pois como z^2 e $\delta(y,z)$ são públicos, o verificador conseguiria extraír v. Assim t_1 e t_2 são factores ofuscantes estruturados para manter t(x) secreto ao provador. Mais adiante na altura da *execução*, irá ser necessário que $t(x) = \langle \underline{l}(x), \underline{r}(x) \rangle$, mais uma razão para que t(x) não possa ser só igual a t_0. O que torna T_1 e T_2 necessários ao circuito. Aqui "público"significa a estrutura formal, y e z têm de ser extraídos do traslado pelo verificador.

$$\begin{array}{ccccccccc} \langle \underline{l}(x), \underline{G} \rangle & = & \langle \underline{G}, \underline{a}_L \rangle & + & \langle \underline{G}, \dot{\underline{s}}_L \rangle & + & \langle -z\underline{1}, \underline{G} \rangle & * 5 \\ + & & + & & + & & + & \\ \langle \underline{r}(x), \underline{H'} \rangle & = & \langle \underline{H}, \underline{a}_R \rangle & + & x\langle \underline{H}, \dot{\underline{s}}_R \rangle & + & \langle z\underline{y}^n + z^2\underline{2}^n, \underline{H'} \rangle & * 6 \\ + & & + & & + & & \| & \\ \mu G & = & \dot{\alpha}G & + & x\dot{\rho}G & & & \\ \| & & \| & & \| & & & \end{array}$$

$$= A + xS + \underbrace{\langle z\underline{y}^n + z^2\underline{2}^n, \underline{H'} \rangle + \langle -z\underline{1}, \underline{G} \rangle}_{\text{público}}$$

Em que $\underline{H'} = \underline{y}^{-n} \circ \underline{H} \leftrightarrow \underline{H} = \underline{y}^n \circ \underline{H'}$

Esta é uma das partes mais elegantes, o provador não possui $\underline{y}^n$ á sua disposição para inserir no circuito aritmético. Portanto usa $\underline{H'}$ para fazer os compromissos de forma implícita/indirecta! Ou seja o provador tem de inserir $\underline{y}^n$ contra $\underline{a}_R$ e $\dot{\underline{s}}_R$, só que por causa da forma como o metodo é implementado nessa altura da comunicação entre provador e verificador, $\underline{y}^n$ ainda não existia para o provador, veja-se na página anterior como y só existe depois do compromisso a S ...

Imagine-se que o provador envia :

$t(x)$, τ_x, $\underline{l}(x)$,$\underline{r}(x)$, μ, T_1, T_2, A, S

ao verificador. Note-se que o verificador só através de $\underline{l}(x)$ e $\underline{r}(x)$ não consegue extraír t_0, por causa dos factores ofuscantes $\dot{\underline{s}}_L$ e $\dot{\underline{s}}_R$.

5.7.1 Verificação

Agora se o verificador receber as duas equações anteriores, ele consegue provar que V está no domínio requerido. Mas por agora ainda estamos numa comunicação com dois

vectores de 64 bits, $\underline{l}(x)$ e $\underline{r}(x)$:

$t(x)H + \tau_x G + \langle \underline{l}(x), \underline{G} \rangle + \langle \underline{r}(x), \underline{H'} \rangle + \mu G =$
$= z^2 V + \delta(y,z)H \ + xT_1 + \ x^2 T_2 + A + xS + \langle z\underline{y}^n + z^2 \underline{2}^n, \underline{H'} \rangle + \langle -z\underline{1}, \underline{G} \rangle$

a sorte é que :

$$\langle \underline{l}(x), \underline{G} \rangle + \langle \underline{r}(x), \underline{H'} \rangle + \langle \underline{l}(x), \underline{r}(x) \rangle U = P_0 - \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k \qquad (5.2)$$

em que o $k = \log_2 64 = 6$

o que depois implica :
$t(x)H + \tau_x G + P_0 - \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k + \mu G =$
$= \langle \underline{l}(x), \underline{r}(x) \rangle U + z^2 V + \delta(y,z)H \ + xT_1 + \ x^2 T_2 + A + xS + \langle z\underline{y}^n + z^2 \underline{2}^n, \underline{H'} \rangle + \langle -z\underline{1}, \underline{G} \rangle$
o provador aproveita o facto de que :

$$t(x) = \langle \underline{l}(x), \underline{r}(x) \rangle$$

e só envia t(x) :

$t(x)H + \tau_x G + P_0 - \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k + \mu G =$ pow!
$= t(x)U + z^2 V + \delta(y,z)H \ + xT_1 + \ x^2 T_2 + A + xS + \langle z\underline{y}^n + z^2 \underline{2}^n, \underline{H'} \rangle + \langle -z\underline{1}, \underline{G} \rangle$ ■
ou seja nem sequer se enviam os u_j^{-2} nem os u_j^2.
o que para o custo de comunicação ainda é melhor, enviam-se então 2*6 + 9, escalares e pontos de CE:
$t(x)$, τ_x, a, b, μ, T_1, T_2, A, S, $L_{1,2,3,...6}$ e $R_{1,2,3,...6}$
Em vez de 2*64 pelo menos, como nas assinaturas borromeanas.

mais detalhes do provador

Aqui continua o processo do provador, no protocolo do produto interno IP. Têm-se que P_k é o argumento de começo para o protocolo :
$P_k = \langle \underline{l}_k(x), \underline{G}_k \rangle + \langle \underline{r}_k(x), \underline{H'}_k \rangle + \langle \underline{l}_k(x), \underline{r}_k(x) \rangle U$
Aqui $isto^{\rightarrow}$ significa do início do vector até metade, e $isto^{\leftarrow}$ significa do fim do vector até metade.

Enquanto $n > 1$ (começa-se com $k = 6$)
$L_k = \langle \underline{l}_k^{\rightarrow}(x), \underline{G}_k^{\leftarrow} \rangle + \langle \underline{r}_k^{\leftarrow}(x), \underline{H'}_k^{\rightarrow} \rangle + \langle \underline{l}_k^{\rightarrow}(x), \underline{r}_k^{\leftarrow}(x) \rangle U$
$R_k = \langle \underline{l}_k^{\leftarrow}(x), \underline{G}_k^{\rightarrow} \rangle + \langle \underline{r}_k^{\rightarrow}(x), \underline{H'}_k^{\leftarrow} \rangle + \langle \underline{l}_k^{\leftarrow}(x), \underline{r}_k^{\rightarrow}(x) \rangle U$

$^{13}\perp_U(L_k)$
$^{14}\perp_U(R_k)$

$u_k \leftarrow {}^{15}\perp_C()$

$$\underline{l}_{k-1}(x) = u_k \underline{l}_k^{\rightarrow}(x) + u_k^{-1} \underline{l}_k^{\leftarrow}(x)$$
$$\underline{r}_{k-1}(x) = u_k^{-1} \underline{r}_k^{\rightarrow}(x) + u_k \underline{r}_k^{\leftarrow}(x)$$
$$\underline{G}_{k-1} = u_k^{-1} \underline{G}_k^{\rightarrow} + u_k \underline{G}_k^{\leftarrow}$$
$$\underline{H'}_{k-1} = u_k \underline{H'}_k^{\rightarrow} + u_k^{-1} \underline{H'}_k^{\leftarrow}$$

$n = n/2$
se $n = 1$ devolve-se a e b.
Em que o 13 em $^{13}\perp_U()$ marca só a primeira ronda. O verificador não se irá perder, e no passo n° 13 do seu traslado, irá colher o mesmo. Passamos agora á magia em volta de P_k. Primeiro é notável como os u_j's podem ser quaisquer valores em $\mathbb{F}_l$. Para chegar a P_k, começa-se a partir de P_{k-1}:
$P_{k-1} = \langle \underline{l}_{k-1}(x), \underline{G}_{k-1} \rangle + \langle \underline{r}_{k-1}(x), \underline{H'}_{k-1} \rangle + \langle \underline{l}_{k-1}(x), \underline{r}_{k-1}(x) \rangle U$

$$\begin{aligned} P_{k-1} = \langle & u_k \underline{l}_k^{\rightarrow}(x) + u_k^{-1} \underline{l}_k^{\leftarrow}(x), u_k^{-1} \underline{G}_k^{\rightarrow} + u_k \underline{G}_k^{\leftarrow} \rangle + \\ & \langle u_k^{-1} \underline{r}_k^{\rightarrow}(x) + u_k \underline{r}_k^{\leftarrow}(x), u_k \underline{H'}_k^{\rightarrow} + u_k^{-1} \underline{H'}_k^{\leftarrow} \rangle + \\ & \langle u_k \underline{l}_k^{\rightarrow}(x) + u_k^{-1} \underline{l}_k^{\leftarrow}(x), u_k^{-1} \underline{r}_k^{\rightarrow}(x) + u_k \underline{r}_k^{\leftarrow}(x) \rangle U \end{aligned}$$

$$\begin{aligned} P_{k-1} = & \langle u_k \underline{l}_k^{\rightarrow}(x), u_k^{-1} \underline{G}_k^{\rightarrow} \rangle + \langle u_k \underline{l}_k^{\rightarrow}(x), u_k \underline{G}_k^{\leftarrow} \rangle \\ & \langle u_k^{-1} \underline{l}_k^{\leftarrow}(x), u_k \underline{G}_k^{\leftarrow} \rangle + \langle u_k^{-1} \underline{l}_k^{\leftarrow}(x), u_k^{-1} \underline{G}_k^{\rightarrow} \rangle + \\ & \langle u_k^{-1} \underline{r}_k^{\rightarrow}(x), u_k \underline{H'}_k^{\rightarrow} \rangle + \langle u_k^{-1} \underline{r}_k^{\rightarrow}(x), u_k^{-1} \underline{H'}_k^{\leftarrow} \rangle + \\ & \langle u_k \underline{r}_k^{\leftarrow}(x), u_k^{-1} \underline{H'}_k^{\leftarrow} \rangle + \langle u_k \underline{r}_k^{\leftarrow}(x), u_k \underline{H'}_k^{\rightarrow} \rangle + \\ & \langle u_k \underline{l}_k^{\rightarrow}(x), u_k^{-1} \underline{r}_k^{\rightarrow}(x) \rangle U + \langle u_k \underline{l}_k^{\rightarrow}(x), u_k \underline{r}_k^{\leftarrow}(x) \rangle U + \\ & \langle u_k^{-1} \underline{l}_k^{\leftarrow}(x), u_k \underline{r}_k^{\leftarrow}(x) \rangle U + \langle u_k^{-1} \underline{l}_k^{\leftarrow}(x), u_k^{-1} \underline{r}_k^{\rightarrow}(x) \rangle U \end{aligned}$$

$P_{k-1} =$
$\langle \underline{l}_k^{\rightarrow}(x), \underline{G}_k^{\rightarrow} \rangle \quad + u_k^2 \langle \underline{l}_k^{\rightarrow}(x), \underline{G}_k^{\leftarrow} \rangle$
$\langle \underline{l}_k^{\leftarrow}(x), \underline{G}_k^{\leftarrow} \rangle \quad + u_k^{-2} \langle \underline{l}_k^{\leftarrow}(x), \underline{G}_k^{\rightarrow} \rangle +$
$\langle \underline{r}_k^{\rightarrow}(x), \underline{H'}_k^{\rightarrow} \rangle \quad + u_k^{-2} \langle \underline{r}_k^{\rightarrow}(x), \underline{H'}_k^{\leftarrow} \rangle +$
$\langle \underline{r}_k^{\leftarrow}(x), \underline{H'}_k^{\leftarrow} \rangle \quad + u_k^2 \langle \underline{r}_k^{\leftarrow}(x), \underline{H'}_k^{\rightarrow} \rangle +$
$\langle \underline{l}_k^{\rightarrow}(x), \underline{r}_k^{\rightarrow}(x) \rangle U + u_k^2 \langle \underline{l}_k^{\rightarrow}(x), \underline{r}_k^{\leftarrow}(x) \rangle U +$
$\underbrace{\langle \underline{l}_k^{\leftarrow}(x), \underline{r}_k^{\leftarrow}(x) \rangle U}_{P_k} + \underbrace{u_k^{-2} \langle \underline{l}_k^{\leftarrow}(x), \underline{r}_k^{\rightarrow}(x) \rangle U}_{u_k^2 L_k + u_k^{-2} R_k}$

clik!

E como tal possibilita-se a seguinte recursão :
$P_k = P_{k-1} - (u_k^2 L_k + u_k^{-2} R_k)$
$P_{k-1} = P_{k-2} - (u_{k-1}^2 L_{k-1} + u_{k-1}^{-2} R_{k-1})$
.
.
$P_1 = P_0 - (u_1^2 L_1 + u_1^{-2} R_1)$
tal que :

$$P_k = P_0 - \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k$$

$$P_0 = {}^0\underline{l}(x)^0\underline{G} + {}^0\underline{r}(x)^0\underline{H'} + {}^0\underline{l}(x)^0\underline{r}(x)U$$

Em que ${}^0\underline{l}(x)$ e ${}^0\underline{r}(x)$ são só dois escalares (em Monero representa-se isto como a e b respectivamente na transacção).

o verificador e o fim da batalha

Enquanto que ${}^0\underline{l}(x)$ e ${}^0\underline{r}(x)$ têm de ser transmitidos ao verificador, ${}^0\underline{G}$ e ${}^0\underline{H'}$ pode ser, e é calculado independentemente pelo verificador. Se o $\underline{G}_k$ inicial tivesse só 8 elementos (em vez de 64):

$$\begin{bmatrix} {}_0\underline{G}_3 u_3^{-1} \\ {}_1\underline{G}_3 u_3^{-1} \\ {}_2\underline{G}_3 u_3^{-1} \\ {}_3\underline{G}_3 u_3^{-1} \\ {}_4\underline{G}_3 u_3 \\ {}_5\underline{G}_3 u_3 \\ {}_6\underline{G}_3 u_3 \\ {}_7\underline{G}_3 u_3 \end{bmatrix} \rightarrow \begin{bmatrix} ({}_0\underline{G}_2 = {}_0\underline{G}_3 u_3^{-1} + {}_4\underline{G}_3 u_3) u_2^{-1} \\ ({}_1\underline{G}_2 = {}_1\underline{G}_3 u_3^{-1} + {}_5\underline{G}_3 u_3) u_2^{-1} \\ ({}_2\underline{G}_2 = {}_2\underline{G}_3 u_3^{-1} + {}_6\underline{G}_3 u_3) u_2 \\ ({}_3\underline{G}_2 = {}_3\underline{G}_3 u_3^{-1} + {}_7\underline{G}_3 u_3) u_2 \end{bmatrix} \rightarrow \begin{bmatrix} {}_0\underline{G}_1 u_1^{-1} \\ {}_1\underline{G}_1 u_1 \end{bmatrix} \rightarrow \begin{bmatrix} {}^0\underline{G} \end{bmatrix}$$

O padrão que ${}_2\underline{G}_3$ segue por exemplo é :

$${}_2\underline{G}_3 u_3^{-1} u_2 u_1^{-1}$$
porque $2 = 010_2$

E ${}_7\underline{G}_3 u_3 u_2 u_1$, porque $7 = 111_2$ ...

E genericamente para ${}_i\underline{G}_k$, segue-se :

$$ {}_i\underline{G}_k \rightarrow {}_i\underline{G}_k u_k^{b(i,k)} u_{k-1}^{b(i,k-1)} u_{k-2}^{b(i,k-2)} ... u_1^{b(i,1)} $$

$$ b(i,j) = \begin{cases} \text{-1, se o j-ésimo bit de i for 0} \\ \text{1, se o j-ésimo bit de i for 1} \end{cases} $$

O que acontece então é que o verificador forma um vector $\underline{\phi}$ com 64 elementos tal que cada elemento :

$$ \underline{\phi}[i] = \prod_{m=1}^{k} u_m^{b(i,m)} $$

O mesmo é feito para ${}^0\underline{H'}$, em que o verificador forma outro vector $\underline{\varphi}$ tal que :

$$ \underline{\varphi}[i] = \prod_{m=1}^{k} u_m^{\overline{b(i,m)}} $$

em que :

$$ \overline{b(i,m)} = \begin{cases} \text{1, se o j-ésimo bit de i for 1} \\ \text{-1, se o j-ésimo bit de i for 0} \end{cases} $$

$G, H, \underline{G}, \underline{H}$ $\quad {}^0\perp[init]$

$\quad {}^1\perp_U(V)$

$\quad {}^2\perp_U(A)$

$\quad {}^3\perp_U(S)$

$y \leftarrow {}^4\perp_C(), y^{-1}$

$z \leftarrow {}^5\perp_C()$

$$\delta(y,z) = (z - z^2)\langle \underline{1}, \underline{y}^n \rangle - z^3 \langle \underline{1}, \underline{2}^n \rangle$$

$\quad {}^6\perp_U(T_1)$

$\quad {}^7\perp_U(T_2)$

$x \leftarrow {}^8\perp_C()$

$\quad {}^9\perp_U(\tau_x)$

$\quad {}^{10}\perp_U(\mu)$

$\quad {}^{11}\perp_U(t(x))$

$x_{ip} \leftarrow {}^{12}\perp_C()$

$k = 6$

enquanto $k \geq 1$

$\quad {}^{13}\perp_U(L_k)$

$\quad {}^{14}\perp_U(R_k)$

$u \leftarrow {}^{15}\perp_C()$

$k = k - 1$

Agora o verificador está armado com tudo o que precisa para fazer a verificação final. Voltamos então á equação (5.1) e simplificamos:

$$t(x)H + \tau_x G + P_0 - \sum_{j=1}^{k} u_j^{-2} L_k + u_j^2 R_k + \mu G =$$
$$= t(x)U + z^2 V + \delta(y,z)H \ + xT_1 + \ x^2 T_2 + A + xS + \langle z\underline{y}^n + z^2 \underline{2}^n, \underline{H'} \rangle + \langle -z\underline{1}, \underline{G} \rangle$$

$$t(x)H - t(x)U - \delta(y,z)H + \tau_x G + a^0 \underline{G} + b^0 \underline{H'} + abU + \mu G =$$
$$= z^2 V \ + xT_1 + \ x^2 T_2 + A + xS + \langle z\underline{y}^n + z^2 \underline{2}^n, \underline{H'} \rangle + \langle -z\underline{1}, \underline{G} \rangle$$
$$+ \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k$$

$$\begin{aligned}(t(x)-\delta(y,z))H - t(x)U + \tau_x G + a^0\underline{G} + b^0\underline{H'} + abU + \mu G = \\ = z^2V \ \ + xT_1 + \ \ x^2T_2 + A + xS + \langle z\underline{y}^n + z^2\underline{2}^n, \underline{H'}\rangle + \langle -z\underline{1}, \underline{G}\rangle \\ + \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k\end{aligned}$$

$$\begin{aligned}(t(x)-\delta(y,z))H - t(x)U + \tau_x G + abU + \mu G = \\ = z^2V \ \ + xT_1 + \ \ x^2T_2 + A + xS + \langle z\underline{y}^n + z^2\underline{2}^n - b\underline{\varphi}, \underline{H'}\rangle + \langle -z\underline{1} - a\underline{\phi}, \underline{G}\rangle \\ + \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k\end{aligned}$$

$$\begin{aligned}(t(x)-\delta(y,z))H - t(x)x_{ip}H + (\tau_x + \mu)G + abx_{ip}H = \\ = z^2V \ \ + xT_1 + \ \ x^2T_2 + A + xS + \langle z\underline{y}^n + z^2\underline{2}^n - b\underline{\varphi}, \underline{H'}\rangle + \langle -z\underline{1} - a\underline{\phi}, \underline{G}\rangle \\ + \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k\end{aligned}$$

$$\begin{aligned}((t(x)-\delta(y,z)) - ((t(x)-ab)x_{ip}))H = \\ = z^2V + (-\tau_x - \mu)G + xT_1 + x^2T_2 + A + xS \\ + \langle z\underline{y}^n + z^2\underline{2}^n - b\underline{\varphi}, \underline{H'}\rangle \\ + \langle -z\underline{1} - a\underline{\phi}, \underline{G}\rangle \\ + \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k\end{aligned}$$

Como os escalares usados para multiplicar os pontos de CE estão em $\mathbb{F}_l$:
Existe para cada $1 \leq y \leq l$ tal que $yH = H'$ também um $1/y$ (aqui y^{-1}) tal que $(1/y)H' = H$.

$$\begin{aligned}(0,1) = z^2V + (-\tau_x - \mu)G + xT_1 + x^2T_2 + A + xS \\ + ((\delta(y,z) - t(x)) + ((t(x)-ab)x_{ip}))H \\ + \langle z + y^{-n} \circ (z^2\underline{2}^n - b\underline{\varphi}), \underline{H}\rangle \\ + \langle -z\underline{1} - a\underline{\phi}, \underline{G}\rangle \\ + \sum_{j=1}^{k} u_j^2 L_k + u_j^{-2} R_k\end{aligned}$$

Se esta equação passa, então $0 \leq v \leq 2^{64} - 1$

CAPÍTULO 6

Transacções Confidenciais em Anel (RingCT)

Depois dos capítulos 4 e 5 uma transacção de um remetente anónimo para um destinatário anónimo soa algo como :

"A minha transacção utiliza uma chave pública de transacção rG. Irá ser gasta uma saída X (note-se que tem um montante oculto A_X, comprometido em C_X, com um pseudo compromisso de saída C'_X). Gera-se uma saída Y, que pode ser gasta pelo endereço oculto K^o_Y. Esta saída tem um montante oculto A_Y, comprometido em C_Y, encriptado para o destinatário e dentro de um certo domínio (*Bulletproofs*). Note-se que $C'_X - C_Y = 0$."

Algumas questões permanecem. O remetente tinha posse de X? O pseudo compromisso de saída C'_X corresponde a C_X, tal que $A_X = A'_X = A_Y$? Será que alguêm alterou a transacção de forma a mudar o destinatário?

Como mencionado na secção 4.2, nota-se a posse de uma saída de um endereço oculto, com a chave privada de *ver*. Como veremos neste capítulo, prova-se a posse da saída de um endereço oculto ao assinar uma mensagem com a chave privada de *gasto*. Prova-se também que o montante no compromisso de saída anterior, entrada actual de transacção, é igual ao montante do pseudo compromisso de saída, com a chave privada do *compromisso a zero*.

Assinaturas MLSAG permitem fazer tudo isto e ao mesmo tempo ocultar a saída real entre outras saídas desvio provenientes da lista de blocos, tal que observadores não podem ter a certeza qual saída está a ser gasta.

6.1 RCTTypeBulletproof2

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

Monero é uma criptomoeda em constante desenvolvimento. Estruturas de transacção, protocolos, e esquemas criptográficos estão sempre sujeitos a evoluir a medida que novas inovações, objectivos e ameaças são encontradas.

RingCT é mandatório para todas as novas transacções de Monero, portanto não irão ser descritos esquemas de transacção antigos, mesmo sendo estes ainda parcialmente suportados.[1] O tipo de transacção aqui descrito chama-se `RCTTypeBulletproof2`. [2]

Um resumo conceptual de transacções é apresentado na secção 6.2.

Actualmente (protocolo v12) todas as novas transacções utilizam este tipo de transacção. Cada entrada é assinada separadamente. Um exemplo de uma transacção `RCTTypeBulletproof2` com todos os seus componentes, pode ser inspectado no Apêndice **??**.

6.1.1 compromissos a montantes e taxas de transacção

Um moneriano recebeu várias saídas com montantes $a_1, ..., a_m$, em endereços ocultos $K^o_{\pi,1}, ..., K^o_{\pi,m}$ e com compromissos a montantes $C^a_{\pi,1}, ..., C^a_{\pi,m}$.

Ele sabe as chaves privadas $k^o_{\pi,1}, ..., k^o_{\pi,m}$ correspondentes aos endereços ocultos (Secção 4.2). E também os factores ofuscantes x_j usados nos compromissos $C^a_{\pi,j}$ (Secção 5.3).

Típicamente a soma dos montantes das saídas de transacção são *menores* do que a soma dos montantes das respectivas entradas. Assim, a diferença é a taxa que é paga aos mineiros, para que estes estejam incentivados a incluír a transacção na lista de blocos. [3]

Os montantes das taxas de transacção f são guardados em texto claro e não encriptado, nos dados de transacção. Os mineiros podem criar uma saída adicional na *transacção de mineiro* que inluí todas as taxas do bloco (veja-se 7.3.6).

Uma transacção consiste de entradas $a_1, ...a_m$ e saídas $b_1, ...b_p$ tal que :

$$\sum_{j=1}^{m} a_j - \sum_{t=1}^{p} b_t - f = 0,$$

em que f é a taxa de mineiro.

[1] RingCT foi implementado pela primeira vez em Janeiro de 2017 (v4 do protocolo). Depois foi feito obrigatório para todas as transacções a partir de Setembro de 2017 (v6 do protocolo) [**?**]. RingCT é a segunda versão do protocolo de transacções em Monero.

[2] Dentro da era de RingCT existem três tipos de transacção descontinuados : `RCTTypeFull`, `RCTTypeSimple`, e `RCTTypeBulletproof`. Os primeiros dois coexistiram na primeira iteração de RingCT e são explorados na primeira edição deste relatório [**?**]. Depois com o advento de Bulletproofs (v8 do protocolo) `RCTTypeFull` foi descontinuado e `RCTTypeSimple` foi actualizado para `RCTTypeBulletproof`. Devido a melhorias em detalhes com a encriptação dos montantes e das máscaras nos compromissos de saídas (v10), surgiu `RCTTypeBulletproof2`.

[3] Em Monero existe uma taxa base mínima que escala com o peso de transacção. É *semi-mandatória* porque enquanto que é possível criar novos blocos que contenham taxas de transacção mínimas, a maioria dos nós, não propagam tais transacções pela rede. Por outro lado é possível criar transacções com taxas de mineiro acima do mínimo. Para mais veja-se a secção 7.3.4.

[4]

O remetente calcula pseudo compromissos de saída para os montantes nas entradas, $C'^a_{\pi,1}, ..., C'^a_{\pi,m}$ e gera compromissos para os montantes nas saídas $b_1, ..., b_p$. Sejam estes novos compromissos $C^b_1, ..., C^b_p$.

Ele sabe as chaves privadas $z_1, ..., z_m$ para os *compromissos a zero* :

$$(C^a_{\pi,1} - C'^a_{\pi,1}), ...(C^a_{\pi,m} - C'^a_{\pi,m})$$

Como mencionado na secção 5.4, é necessário provar que a soma dos montantes nas entradas menos a soma dos montantes nas saídas, menos a taxa de mineiro é igual a zero. Mas enquanto que os compromissos são pontos de CE, a taxa sendo também um montante só ocupa 8 bytes. Como tal é necessário converter a taxa f para um ponto de CE. A solução é calcular o compromisso á taxa f sem usar um factor ofuscante. Isto é :

$$C(f) = fH$$

A soma dos montantes nas entradas menos a soma dos montantes nas saídas menos a taxa é igual a zero :

src/ringct/rctSigs.cpp verRctSemanticsSimple()

$$(\sum_{j=1}^{m} C'^a_j - \sum_{t=1}^{p} C^b_t) - fH = 0$$

6.1.2 Assinatura

O remetente selecciona m conjuntos de tamanho v, de quaisquer endereços ocultos adicionais da lista de blocos. No total a assinatura τ é constituida por m assinaturas individuais :

$$\tau = \{\sigma_1(\mathfrak{m}), \sigma_2(\mathfrak{m}), ...\sigma_m(\mathfrak{m})\}$$

Cada uma das quais tem de ser verificada individualmente. A assinatura total τ só é válida se *todas* as assinaturas $\sigma_j(\mathfrak{m})$, $j \in \{1, ...m\}$ forem válidas.

[5,6]

O signatário para assinar uma das m entradas, mistura um conjunto de tamanho $v+1$ de entradas dentro de um *anel*. A entrada própria está no index secreto π.

[4] As saídas são misturadas aleatóriamente pela implementação núcleo e só depois é que são indexadas. Assim observadores não conseguem construir heurísticas para a ordem das mesmas. As entradas são ordenadas pelas suas imagens chave.

[5] Em Monero é standard que o conjunto de *saídas desvio*, de RingCT, seja seleccionado por uma distribuição gamma no domínio de todos as saídas passadas (para saídas pre RingCT, é usada uma distribuição *triângulo*). Este método usa um processo de armazenamento para suavizar as diferenças das densidades de bloco. Primeiro calcula-se o tempo médio entre saídas de RingCT num periodo de um ano (tempo médio = [#saídas/#blocos]*tempo_entre_blocos). Selecciona-se uma saída através da distribuição gamma, depois olha-se para dentro do bloco em que está essa saída, e escolhe-se uma outra saída aleatória para ser uma saída desvio [**?**].

src/wallet/wallet2.cpp `get_outs()`

`gamma_picker::pick()`

[6] Desde o protocolo v12, todas as entradas de transacção têm de ter uma idade mínima de 10 blocos (`CRYPTONOTE_DEFAULT_TX _SPENDABLE_AGE`). Antes de v12, não existia este requisito e outras carteiras pelos vistos faziam escolhas diferentes [**?**].

$$\begin{aligned}\mathcal{R}_j = \{&\{K^o_{1,j}, (C_{1,j} - C'^a_{\pi,j})\},\\ &...\\ &\{K^o_{\pi,j}, (C^a_{\pi,j} - C'^a_{\pi,j})\},\\ &...\\ &\{K^o_{v+1,j}, (C_{v+1,j} - C'^a_{\pi,j})\}\}\end{aligned}$$

Note-se que as entradas desvio contêm compromissos a zero que são falsos. Ou seja $C^a_{i,j} - C'^a_{\pi,j}$, para $i \neq \pi$ é somente um ponto de CE. O componente H dessa subtracção é irrelevante. Para mais, o ponto de CE resultante dessa subtracção só é relevante para a construção formal do anel.

A alice usa uma assinatura tipo MLSAG para assinar este anel, em que ela conhece as chaves privadas $k^o_{\pi,j}$ para $K^o_{\pi,j}$, e z_j para o compromisso a zero ($C^a_{\pi,j}$ - $C'^a_{\pi,j}$).

(a) Calcula-se a imagem de chave $\tilde{K^o_{\pi,j}} = k_{\pi,j}\mathcal{H}_p(K^o_{\pi,j})$.

(b) Geram-se números aleatórios $\alpha_1, \alpha_2 \in_R \mathbb{Z}_l$, e $r_{i,1}, r_{i,2} \in_R \mathbb{Z}_l$ para cada $i \in \{1, 2, ..., v+1\}$ (excepto $i = \pi$).

(c) O anel é iniciado :

$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, \tilde{K^o_j}, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K^o_{\pi,j})], [\alpha_2 G])$$

(d) Para $i = \pi + 1, \pi + 2, ...v + 1, 1, 2, ...\pi - 1$ calcula-se, substituindo $v + 2 \to 1$,

$$c_i + 1 = \mathcal{H}_n(\mathfrak{m}, \tilde{K^o_j}, [r_{i,1}G + c_i K^o_{i,j}], [r_{i,1}\mathcal{H}_p(K^o_{i,j}) + c_i \tilde{K^o_j}], [r_{i,2}G + c_i(C^a_{i,j} - C'^a_{\pi,j})])$$

(e) Definem-se as respostas :

$$r_{\pi,1} = \alpha_1 - c_\pi k_{\pi,j} \pmod l$$

e

$$r_{\pi,2} = \alpha_2 - c_\pi z_j \pmod l$$

(f) A assinatura é $\sigma_j(\mathfrak{m}) = (c_1, (r_{1,1}, r_{1,2}), ...(r_{v+1,1}, r_{v+1,2}))$, com a imagem de chave $\tilde{K^o_j}$.

6.1.3 Verificação

A verificação de uma assinatura é feita da seguinte maneira :

(a) Verifique $l\tilde{K^o_j} \stackrel{?}{=} 0$.

(b) Para $i = 1, ..., v + 1$ compute, substituindo $v + 2 \to 1$,
$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, \tilde{K^o_j}, [r_{i,1}G + c'_i K^o_{i,j}], [r_{i,1}\mathcal{H}_p(K^o_{i,j}) + c'_i \tilde{K^o_j}], [r_{i,2}G + c'_i(C^a_{i,j} - C'^a_{\pi,j})])$
Para c'_2 usa-se á direita da equação c_1.
Para c'_i, $i > 2$ usa-se á direita da equação c'_{i-1}.

(c) Se $c_1 = c'_1$ então a assinatura é válida.

Cada anel R_j é iniciado em duas partes, primeiro com α_1 e depois com α_2. Note-se que a assinatura com z_j, o *compromisso a zero* acontece da seguinte forma :

$$\begin{aligned}\alpha_2 G &= (r_{\pi,2} + c_\pi z_j)G \\ &= r_{\pi,2}G + c_\pi z_j G \\ &= r_{\pi,2}G + c_\pi (C^a_{\pi,j} - C'^a_{\pi,j})\end{aligned}$$

Desde que cada anel contêm um compromisso a zero, o pseudo compromisso de saída usado têm de conter um montante igual á entrada que está a ser própriamente gasta [7,8].

src/ringct/rctSigs.cpp proveRctMGSimple()

A mensagem $\mathfrak{m}$ assinada por $\sigma_j(\mathfrak{m})$, é *literalmente* uma hash de todos os outros dados da transacção actual, *excepto* as assinaturas MLSAG.[9] Assim é garantido que as transacções são á prova de qualquer falsificação, da perspectiva do autor e dos verificadores. A chave privada oculta $k^o_{\pi,j}$ é a essência do modelo de transacção em Monero. Quem assina $\mathfrak{m}$ com $k^o_{\pi,j}$ prova que tem posse do montante no pseudo compromisso em $C'^a_{\pi,j}$.

6.1.4 Evitar o gasto duplo

Uma assinatura MLSAG contêm *imagens de chave* $\tilde{K}^o_j$ de chaves privadas k^o_j. Uma propriedade importante de um esquema de assinatura criptográfico é de ser infalsificável (enp).

Assim efectivamente as *imagens de chave* de uma assinatura têm de ter sido produzidas deterministicamente de chaves privadas legítimas. A rede Monero só precisa de verificar que *imagens de chave* incluídas em assinaturas MLSAG não apareceram antes, em outras transacções. Se isto é o caso então o remetente da assinatura está a tentar fazer um gasto duplo de uma saída $(K^o_{\pi,j}, C^a_{\pi,j})$.[10]

src/cryptonote_core/blockchain.cpp `have_tx_keyimges_as_spent()`

[7] A construção e verificação da assinatura excluí o termo $r_{i,2}\mathcal{H}_p(C_{i,j} - C'^a_{\pi,j}) + c_i\tilde{K}_{z_j}$.

[8] A vantagem de assinar entradas individualmente é que as entradas verdadeiras com os seus compromissos a zero, não têm de estar todas no mesmo index π, ao invés de como seria no caso agregado. Isto significa que mesmo se a origem de uma entrada fosse identificada, as origens/índices das outras entradas mantêm-se ocultas. O tipo de transacção `RCTTypeFull` foi descontinuado porque usava assinaturas em anel agregadas, o que combinava todos os aneis para um só.

[9] A actual mensagem é :$\mathfrak{m} = \mathcal{H}(\mathcal{H}(tx\_prefix), \mathcal{H}(ss), \mathcal{H}(\text{range proofs}))$ em que :

tx_prefix ={Versão da era da transacção (i.e. RingCT = 2), entradas {offsets dos membros de anel, imagens de chave}, saídas {endereços ocultos}, extra {chave pública de transacção, ID de pagamento, ou ID de pagamento codificado, misc ...}}

ss ={tipo de transacção (`RCTTypeBulletproof2` = '4'), taxa de transacção, pseudo compromissos de saída, ecdhInfo (montantes encriptados), compromissos de saída}.

Veja-se o Apêndice A.

src/ringct/rctSigs.cpp `get_pre_mlsag_hash()`

[10] Verifiers must also check the key image is a member of the generator's subgroup (recall Section 3.4).

6.1.5 requisitos de espaço

MLSAG signature (inputs)

Uma assinatura MLSAG (secção 3.5), assumindo compressão de ponto (secção 2.4.2), cada anel $\mathcal{R}_j$ contêm $(v+1)\cdot 2$ chaves. Adicionalmente, a *imagem de chave* $\tilde{K}_j^o$ e o pseudo compromisso de saída $C'^a_{\pi,j}$, resultam num total de $(2(v+1)+3)\cdot 32$ bytes para cada assinatura de uma entrada. A este valor é necessário adicionar espaço para guardar os *offsets* dos membros de anel. Estes *offsets* são utilizados pelos verificadores para encontrar as chaves das saídas e para encontrar os compromissos na lista de blocos. Estes offsets são guardados como inteiros de comprimento variável e como tal não é possível precisar o espaço requerido.[11]

src/ringct/rctSigs.cpp verRctMGSimple() verRctSemanticsSimple()

[12] Verificar todas as assinaturas MLSAG de uma transacção `RCTTypeBulletproof2` incluí a computação de $(C_{i,j} - C'^a_{\pi,j})$ e $(\sum_j C'^a_j \stackrel{?}{=} \sum_t C^b_t + fH)$, e verificar que as imagens de chave estão no subgrupo de G com $l\tilde{K} \stackrel{?}{=} 0$.

6.1.6 o segredo público γ

Note-se que, nenhum moneriano conhece γ. E isto é fundamental, porque quem conhece gamma pode também imprimir dinheiro á vontade. Assume-se então que óscar quer começar a imprimir xmr que ele não possui nas suas saídas. De alguma forma milagrosa ele conhece γ. Ele compra ou recebe xmr para um endereço oculto K^o. Ele conhece o montante b, como também o factor ofuscante y. Que corresponde ao compromisso de saída C^a. Ele quer de imediato, mudar o montante de 0.1 xmr para 100 xmr. Para isso ele precisa de encontrar um novo C^a_{mau}, tal que $C^a_{mau} = C^a$:

$$\begin{aligned} C^a &= yG + bH \\ C^a &= yG + b(\gamma G) \\ C^a_{mau} &= y_{mau}G + b_{mau}(\gamma G) \end{aligned}$$

Ou seja ele faz :

$$\begin{aligned} C^a &= C^a_{mau} \\ yG + b(\gamma G) &= y_{mau}G + b_{mau}(\gamma G) \\ G(y + b\gamma) &= G(y_{mau} + b_{mau}\gamma) \\ y + b\gamma &= y_{mau} + b_{mau}\gamma \\ y + b\gamma - b_{mau}\gamma &= y_{mau} \end{aligned}$$

[11] Veja-se [**?**] ou [**?**] para uma esplicação do tipo de dados *varint* em Monero . É um tipo de inteiro que usa até 9 bytes e guarda até 63 bits de informação. src/common/varint.h

[12] Uma transacção com 10 entradas que use aneis, cada anel com um total de 11 membros, precisa de $((11 \cdot 2 + 3) \cdot 32) \cdot 10 = 8000$ bytes para as suas entradas, com por volta de 110 a 330 bytes para os offsets.

Em que $b_{mau} = 100$xmr ou mais. Efectivamente o oscar agora possui um novo b e um novo y para C^a. Ele pode agora continuar com o processo da transacção como se nada fosse !

Note-se como não conhecer γ proíbe esta burla :

$$\begin{aligned} C^a &= C^a_{mau} \\ yG + bH &= y_{mau}G + b_{mau}H \\ yG + bH - b_{mau}H &= y_{mau}G, \end{aligned}$$

e agora "ah e tal", queremos conhecer y_{mau} ...

Mas isto é o mesmo que dizer :

$$\log_G(yG + bH - b_{mau}H) = y_{mau},$$

o que não se pode fazer por causa do PLD. O que é interessante aqui é que não é só γ que é secreto. O segredo de γ implica uma serie de logaritmos discretos desconhecidos... Ou seja, cada combinação de valores á esquerda da equação implica um logaritmo discreto desconhecido e que se quebrado, permite "gamar".

provas de domínio (saídas)

Uma prova de domínio *Bulletproof* requer :

$$(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$$

6.2 Resumo conceptual

Para resumir, apresenta-se aqui o conteúdo principal de uma transacção, organizada para a clareza conceptual. Um exemplo real pode ser encontrado no apêndice A.

- Tipo: '0' é `RCTTypeNull` (para mineiros), '4' é `RCTTypeBulletproof2`
- Entradas: para cada entrada $j \in \{1, ..., m\}$ gasta pelo remetente da transacção :
  - **offsets dos membros do anel**: uma lista de 'offsets' que indicam onde se encontram os membros do anel $i \in \{1, ..., v+1\}$, da entrada j na lista de blocos (incluí a entrada própria)
  - **Assinatura MLSAG**: σ_j termos c_1, e $r_{i,1}$ & $r_{i,2}$ para cada $i \in \{1, ..., v+1\}$
  - **Imagem de chave**: a imagem de chave $\tilde{K}_j^{o,a}$ para a entrada j
  - **Compromisso de pseudo saída**: $C_j^{'a}$ para a entrada j
- Saídas: para cada saída $t \in \{1, ..., p\}$ para um endereço ou sub-endereço (K_t^s, K_t^v)
  - **Endereço oculto**: $K_t^{o,b}$ para a saída t
  - **compromisso de saída**: C_t^b para a saída t
  - **montante em código**: o proprietário da saída t, pode calcular b_t.
    - *Montante*: $b_t \oplus_8 \mathcal{H}_n(\text{“}montante\text{”}, \mathcal{H}_n(rK_B^v, t))$
  - **Prova de domínio**: uma prova de domínio agregada Bulletproof para cada montante de saída b_t
    - *Prova*: $\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$
- Taxa: texto não encriptado, multiplicado por 10^{12} (i.e. unidades atómicas, picо-nero. Veja-se a secção 7.3.1), assim uma taxa de 1 xmr tem aqui o valor 10000 0000 0000.
- Extra: inclui a chave pública de transacção rG, se pelo menos uma saída é direccionada a um sub-endereço inclui-se também $r_t K_t^{s,i}$ para cada saída de sub-endereço t, e $r_t G$ para cada endereço normal t. Aqui pode estar incluído também um ID de pagamento.
  [13]

[13] Nenhuma informação dentro do campo 'extra' é verificada, mas esta é *assinada* pelas entradas em MLSAGs, portanto nenhuma falsificação é possivel (enp). O campo extra não tem nenhum limite máximo de dados que pode guardar, desde que o peso de transacção máximo seja respeitado. Veja-se [**?**] para mais detalhes.

Finalmente uma transacção exemplo com uma entrada e uma saída soa da seguinte forma : " A transacção utiliza uma chave pública de transacção rG. Uma saída no conjunto $\mathbb{X}$ é gasta. Note-se que têm um montante oculto A_X, ao qual é comprometido em C_X. Foi feita uma assinatura MSLAG pelo proprietário da saída aos endereços ocultos em $\mathbb{X}$. Esta saída não foi gasta antes, porque a *imagem de chave* $\tilde{K}$ não se encontra na lista de blocos. Adiciona-se um pseudo compromisso de saída C'_X. Os fundos são para uma saída Y, o que por sua vez será gasto através do endereço oculto K^o_Y. Têm um montante oculto A_Y comprometido em C_Y, encriptado para o destinatário, e através de uma prova de domínio Bulletproof, dentro de um certo domínio de valor positivo. Inclui uma taxa f. Note-se que $C'_X - (C_Y + C_f) = 0$, existe também uma assinatura ao compromisso a zero $C'_X - C_X = zG$ o que implica que o montante da entrada é igual ao montante da saída mais a taxa ($A_X = A'_X = A_Y + f$). a assinatura MLSAG assinou todos os dados da transacção, assim os verificadores podem garantir que não houve algo dentro dos dados da transacção que tenha sido modificado."

6.2.1 Requisitos de espaço

Para `RCTTypeBulletproof2` são necessários :

$$(2(v+1)+2)\cdot m\cdot 32bytes$$

A prova de domínio Bulletproof requer :

$$(2\cdot\lceil\log_2(64\cdot p)\rceil+9)\cdot 32bytes$$

[14]

Requisitos diversos :

- Imagens de chave de entrada : $m*32$ bytes
- Endereços ocultos de saída : $p*32$ bytes
- Compromissos de saída: $p*32$ bytes
- Montantes de saída em código : $p*8$ bytes
- Chave pública de transacção: 32 bytes, $p*32$ bytes ao enviar para pelo menos um sub-endereço.
- ID de pagamento : é único, 8 bytes para um endereço integrado.
- Taxa : inteiro de comprimento variável ≤ 9 bytes.
- Offsets de entrada : inteiros de comprimento variável, assim ≤ 9 bytes para cada offset, para cada $m*(v+1)$ membro de anel.
- tempo de desbloqueio: inteiro de comprimento variável ≤ 9 bytes .[15]
- Extra : No campo 'extra' os dados são marcados por um byte, cujos bits estão todos postos a um. Veja-se o apêndice A para mais detalhes.

[14] O tamanho de uma transacção está limitado por um assim chamado 'peso de transacção'. Antes de Bulletproofs terem sido adicionados, no protocolo v8, o peso de transacção e o tamanho em bytes era equivalente (o que ainda é o caso para transacções com 2 saídas). O peso máximo é (0.5*300kB - `CRYPTONOTE_COINBASE_BLOB_RESERVED_SIZE`), em que a reserva de *blob* de 600 bytes, existe para deixar espaço para as transacções de mineiro. Antes de v8 o multiplicador 0.5x não estava incluído, e o termo de 300kB era menor em versões anteriores do protocolo (20kB v1, 60kB v2, 300kB v5). Estes tópicos são elaborados na secção 7.3.2.

src/cryptonote_core/tx_pool.cpp `get_transaction_weight_limit()`

[15] Um autor de uma transacção pode bloquear as suas saídas, até a uma altura de bloco específica, e só depois é que tais saídas podem ser gastas. Ou então, até que uma data seja alcançada (carimbo no tempo de UNIX). Só é possivel que todas as saídas sejam bloqueadas até á mesma altura de bloco. Não é claro se isto oferece alguma utilidade aos autores de transacção (talvez para *smart contracts*). Transacções de mineiro têm um tempo de bloqueio mandatório de 60 blocos. Desde o protocolo v12 saídas normais não podem ser gastas até depois de terem uma idade de 10 blocos (isto chama-se a *idade standard de gasto*). Em termos funcionais isto é o mesmo que um um tempo de bloqueio mandatório mínimo de 10 blocos. Se uma transacção é publicada no décimo bloco com um tempo de desbloqueio de 25, esta pode ser gasta no bloco 25 ou depois. O tempo de desbloqueio é provavelmente a funcionalidade menos utilizada para transacções normais.

CAPÍTULO 7

A Blockchain

Em português : a lista de blocos.

A experiência humana ganho uma nova dimensão através da internet. Nós podemos corresponder com pessoas em todos os cantos do planeta, e uma inimaginável quantidade de informação está disponível. A troca de bens e serviços é fundamental para uma sociedade próspera e pacífica [**?**], e no reino digital é possível contribuír valor para o mundo inteiro.

Meios de câmbio (dinheiros) são essenciais, dando-nos um ponto de referência para uma immensa diversidade de bens económicos que de outra forma seriam impossíveis de avaliar, e permitindo interacções mutuamente benéficas entre pessoas que nada têm em comum [**?**]. Ao longo da história houve muitos tipos differentes de dinheiro, de conchas a papel e ouro. Estes são trocados á mão, e agora o dinheiro pode ser trocados electronicamente.

No modelo actual, e de longe o mais adoptado, as transacções electrónicas são controladas por instituições financeiras. Estas entidades terceiras possuem a custódia do dinheiro, e as transacções acontecem com requesitos com confiança. Tais instituições têm de mediar disputas, os pagamentos são reversíveis, e podem ser censurados ou controlados por organizações poderosas.[**?**]

Para aleviar estas inconveniências, surgiram moedas digitais e descentralizadas.
[1]

[1] Este capítulo incluí mais detalhes de implementação do que os anteriores, pois a natureza da blockchain depende muito da sua estrutura específica.

7.1 Moedas digitais

Desenvolver uma moeda digital não é trivial. Existem três tipos: pessoal, centralizada e distribuída. Uma moeda digital é somente uma collecção de mensagens, e os 'montantes' guardados nessas mensagens são interpretados como quantidades monetárias.

No **modelo de e-mail** qualquer pessoa pode fazer moedas. Por exemplo dizer 'eu tenho 5 moedas', e enviar esta mensagem a todas as pessoas que tenham um endereço de e-mail. A quantidade total deste dinheiro não é limitada, e é possível gastar as mesmas moedas mais que uma vez (gasto duplo).

No **modelo de video jogo**, as moedas são guardadas numa base de dados centralizada. Os utilizadores confiam na honestidade do guardião. A quantidade total não pode ser verificada por observadores externos, e o guardião pode alterar as regras a qualquer altura, ou ser censurado por terceiros poderosos.

7.1.1 Versão de eventos comuns e distribuídos

No dinheiro digital 'comum', muitos computadores possuem o registo de cada transacção. Quando uma nova transacção é feita por um computador, esta é transmitida pela rede inteira dos outros computadores, e é aceite se segue certas regras predifinidas.

Os utilizadores só beneficiam de moedas se outros as aceitam em troca. Para maximizar a utilidade das moedas, existe uma tendência natural para definir um conjunto de regras comuns, sem a presença de uma autoridade central.
[2]

Regra n° 1: Dinheiro so pode ser creado em cenários bem definidos.

Regra n° 2: Transacções gastam dinheiro que já existe.

Regra n° 3: Cada transacção só ocorre uma vez.

Regra n° 4: Somente o proprietário do montante o pode gastar.

Regra n° 5: As saídas de transacção representam o dinheiro gasto.

Regra n° 6: As transacções seguem a sintaxe correcta.

As regras 2-6 estão contidas no esquema de transacção discutido no capítulo 6, o que implica fungibilidade e privacidade em relação ao signatário ambîguo, destinatário anônimo, e montantes ocultos a terceiros. A primeira regra irá ser explicada mais tarde neste capítulo. As transacções fazem uso da criptografia e como tal trata-se de uma *cripto-moeda*.

Seja que dois computadores distantes recebem cada um, uma transacção para gastar do mesmo montante (da mesma entrada), mas para destinatários distintos. Dado que depois de um curto espaço de tempo os dois computadores teriam as duas transacções em questão, como é que cada um deles decide qual é a transacção válida? Acontece então uma divisão do histórico de transacções, porque existem duas cópias distintas que seguem as mesmas regras ('fork' do inglés : "garfo").

[2] Na ciência política diz-se a isto um contrato social.

Claramente a transacção que ocorreu antes deve ser considerada canônica. Isto é mais fácil dito do que feito. Como veremos, obter o consenso para o histórico de transacções constitui a *razão de ser* da tecnologia blockchain.

7.1.2 Blockchain simples

Primeiro é necessário que todos os computadores, doravante ditos como *nodes*, concordem com a ordem das transacções.

Diga-mos que uma cripto-moeda começa com uma declaração de génese : "A moeda exemplo começa!"

Esta mensagem é, neste caso, um 'bloco', e a hash do bloco é :

$$BH_G = \mathcal{H}(\text{“A moeda exemplo começa!)}$$

Cada vez que um node recebe transacções, estas passam por uma função hash TH. Bem como cada bloco também passa por uma função hash BH. E cada bloco "aponta"para o bloco anterior, como tal uma blockchain é uma lista ou vector unidimensional de blocos. Em que o primeiro bloco é o bloco de genese :

$$BH_1 = \mathcal{H}(BH_G, TH_1, TH_2, ...)$$
$$BH_2 = \mathcal{H}(BH_1, TH_3, TH_4, ...)$$

Desta forma existe uma clara ordem de eventos que se extende do bloco actual para o passado até ao bloco genese. Uma blockchain é técnicamente um grafo acíclico dirigido, em que listas de blocos tipo Monero são uma variante unidimensional. Grafos deste estilo contêm um número finito de *nós*, e pares de nós que se chamam *arestas*. Neste caso as arestas são como setas unidireccionais entre nós. Começando num nó não se volta ao mesmo nó, vai-se em frente, até se chegar ás *folhas* do grafo [**?**].

Os nodes podem incluir a data e hora nos seus blocos. Se a maioria dos nodes são honestos a fazer isto então a blockchain serve como uma base de dados decente para cada transacção. Se blocos diferentes referenciam o mesmo bloco anterior, e ambos são propagados pela rede ao mesmo tempo, poderá existir um problema. Imagine-se que metade da rede possui um bloco actual, e a outra metade possui o outro bloco actual. Acontece então um "garfo"na rede, esta situação não é grave e será esplicada posteriormente.

7.2 Dificuldade

Se os nodes podem publicar novos blocos quando eles querem, a rede de nodes pode dividir-se e divergir para diferentes, igualmente legítimas listas de blocos. Além disso, se demora 30 segundos para que o novo bloco se propague por toda a rede, o que acontece se novos blocos são encontrados todos os 31 segundos ou todos os 10 segundos, etc ? É possível controlar a velocidade com a qual a rede inteira cria novos blocos. Se o tempo que é necessário para gerar um novo bloco é muito maior do que o tempo necessário

para que o bloco anterior atinga a maioria dos nodes. Então a rede irá permanecer intacta.

7.2.1 Mineração de um bloco

O resultado de uma função hash criptográfica é uniformemente distribuída e aparentemente independente do argumento dado como entrada. Isto significa, dado qualquer argumento como entrada, qualquer resultado é igualmente provável. Além disso, demora um certo tempo para que uma hash seja calculada. Imagine-se uma função hash $\mathcal{H}_i(x)$ que tem como resultado um número de 1 a 100:

$$\mathcal{H}_i(x) \in_R^D \{1, ..., 100\}$$

[3]

Para um dado x, $\mathcal{H}_i(x)$ selecciona o mesmo número aleatório de $\{1, ..., 100\}$ cada vez que é calculado. Demora 1 minuto para calcular $\mathcal{H}_i(x)$. Seja uma mensagem $\mathfrak{m}$, e o objectivo é encontrar um argumento n (um inteiro) tal que $\mathcal{H}_i(\mathfrak{m}, n)$ tem como resultado um número menor ou igual do que o *alvo* $t = 5$.

$$\mathcal{H}_i(\mathfrak{m}, n) \in \{1, ..., 5\}$$

Desde que só 1/20-ésimo dos resultados de $\mathcal{H}_i(x)$ irá estar dentro do alvo, deve levar por volta de 20 tentativas para encontrar um n que funcione. Ou seja por volta de 20 minutos de tempo de computação. Estar á procura de um tal argumento *nonce* útil é chamado *mineração*, e publicar tal mensagem com n é uma *prova de trabalho*. Note-se que qualquer pessoa pode verificar isto ao calcular $\mathcal{H}_i(\mathfrak{m}, n)$.

Seja uma função hash utilizada para provas de trabalho :

$$\mathcal{H}_{PoW} \in_R^D \{0, ...l\}.$$

Em que l é o resultado máximo. Dada uma mensagem $\mathfrak{m}$, um argumento n para minerar e um alvo t, podemos definir a média do número de hashes calculadas :

$$d = l/t.$$

Em que d é a *dificuldade*.

Se $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d \leq l$, então : $\mathcal{H}_{PoW}(\mathfrak{m}, n) \leq t$, e n é aceitável.[4] Com alvos mais pequenos a dificuldade aumenta e leva a um computador mais e mais hashes, e como tal mais e mais tempo para encontrar argumentos aceitáveis. Note-se que verificar demora sempre o mesmo tempo, uma computação de $\mathcal{H}_{PoW}$, é independente da dificuldade ($\mathcal{O}(1)$).

src/cryptonote_basic/difficulty.cpp `check_hash()`

7.2.2 Velocidade de mineração

Assuma-se que todos os nodes estão a minerar ao mesmo tempo, mas páram com o bloco actual quando recebem um novo da rede. Eles comecam imediatamente a minerar um novo que referência o último encontrado.

[3] Usa-se $\in_R^D$, para dizer que o resultado é aleatório

[4] Em Monero só as dificuldades são calculadas, desde que $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d \leq m$ não precisa de t.

São colleccionados um número b de blocos recentes da lista de blocos, com index $u \in \{1, ..., b\}$. Cada um destes blocos teve uma dificuldade d_u. Por agora assuma-se que os nodes mineiros foram honestos, portanto cada data/hora marcada no bloco está acertada. Então o tempo total entre o bloco mais velho e o mais novo é :

$$Tempo \quad total = TS_b - TS_1.$$

Em que o número aproximado de hashes necessárias para minerar todos os blocos em questão foi : [5]

$$Dificuldade\ total = \sum_{u=1}^{b} d_u.$$

É possivel calcular com que velocidade a rede inteira de nodes, consegue computar hashes. Se a velocidade actual não mudou muito enquanto o conjunto b de blocos foi produzido, deveria ser efectivamente : [6]

$$velocidade\ de\ hash \approx dificuldade\ total / Tempo\ total$$

Seja o tempo alvo entre blocos :

$$um\ bloco / Tempo\ alvo,$$

então calcula-se quantas hashes é que a rede leva para calcular um bloco :

$$nova\ dificuldade = velocidade\ de\ hash * Tempo\ alvo,$$

Note-se que aqui arredonda-se, tal que a dificuldade nunca é igual a zero.

Não existe nenhuma garantia que o próximo bloco irá levar um número total de hashes da rede igual a *nova dificuldade*. Mas ao longo do tempo a dificuldade irá manter-se a par com a velocidade real de hashes da rede, e os blocos iram tender a aparecer cada *Tempo alvo*. [7]

7.2.3 Consenso: maior dificuldade comulativa

Agora é possível resolver conflitos entre garfos de rede.

Por convenção, a lista de blocos com a maior dificuldade comulativa, de todos os blocos, e como tal com o maior trabalho gasto, é considerada a lista de blocos verdadeira e legítima. Se há uma divisão da rede (fork, do inglés : garfo), e cada garfo (parte da rede), tem a mesma dificuldade comulativa, então os nodes continuam a minerar na versão da blockchain que receberam primeiro. Quando um garfo tem uma lista de blocos maior do que o outra, a versão mais fraca é discardada. O que faz com que a rede inteira esteja outra vez em consenso a minerar uma só versão da lista de blocos.

[5] Carimbos no tempo são determinados quando um mineiro *começa* a minerar um bloco. O carimbo no tempo do próximo bloco indica quanto tempo foi gasto no bloco anterior.

[6] Se o node 1 tenta o argumento $n = 23$ e depois o node 2 também tenta $n = 23$, o esforço do node 2 é desperdiçado porque a rede já 'sabe' que $n = 23$ não funciona. A velocidade *efectiva* de hash depende do número de hashes resultantes de nonces *únicas*, para um dado bloco de transacções. Desde que mineiros incluem uma transacção de mineiro com o endereço oculto $K^o \in_{ER} \mathbb{Z}_l$ (ER = efectivamente aleatório), os blocos são sempre únicos entre mineiros excepto com uma probabilidade neglegível, portanto tentar as mesmas nonces faz sentido.

[7] Assumindo que a quantidade total de hashes é constante, e gradualmente a subir, então como novas dificuldades dependem de hashes *passadas*, é de esperar que os tempos entre blocos são, em média, um pouco menos do que *Tempo alvo*. O efeito disto na emissão monetária podia ser cancelado por penalidades de pesos de bloco crescente, o que será explorado na secção 7.3.3.

Se os nodes querem alterar ou fazer upgrade to protocolo, eles podem optar por fazer isto, o que resulta num garfo da rede. Se uma nova versão da rede tem um impacto ou não nos monerianos, depende na quantidade de nodes que mudam e da quantidade de software que é alterada.
[8]

Para que um atacante convença nodes honestos a alterar a história de transacções, ele tem de criar um garfo da lista de blocos com uma dificuldade comulativa maior do que a da rede principal. O que é muito difícil de conseguir, só com o controlo de mais de 50% da velocidade de mineração, tal que o resto dos mineiros não consigam acompanhar tal trabalho. [**?**]

7.2.4 Minerar em Monero

Para garantir que os garfos da lista de blocos não geram conflitos com a calculação da dificuldade, não se usam os blocos mais recentes para este cálculo. Por exemplo se existem 29 blocos na lista, e se o conjunto $b = 10$, e $l = 5$, usam-se os blocos 15-24 para calcular a dificuldade do bloco n° 30.

Se os nodes mineiros são dishonestos eles podem manipular os carimbos no tempo, tal que as novas dificuldades não estão a par com a velocidade efectiva de hash. Isto evita-se ao ordenar os carimbos de tempo cronológicamente e depois cortar os primeiros e últimos o extremos. Agora existe uma *janela de blocos* $w = b - 2 * o$. Do exemplo anterior, se $o = 3$ e os carimbos no tempo são dishonestos então cortam-se os blocos n° 15-17 e n° 22-24, o que deixa os blocos n° 18-21 para calcular a dificuldade do bloco n° 30.

Antes de cortar os extremos ordenam-se os carimbos no tempo, mas *só* os carimbos. As dificuldades dos blocos mantêm-se como estão. Usa-se a dificuldade comulativa para cada bloco, que é a dificuldade do próprio bloco mais a dificuldade de todos os blocos anteriores na lista de blocos.

src/cryptonote_core/blockchain.cpp `get_difficulty_for_next_block()`

Usando o vector ordenado de w carimbos no tempo e dificuldades comulativas (indexadas de $1, ..., w$), define-se :
$Tempo\ total = carimbos\ cortados[w] - carimbos\ cortados[1]$
$Dificuldade\ total = Dificuldades\ comulativas\ cortadas[w] -$
$Dificuldades\ comulativas\ cortadas[1]$

Em Monero o tempo alvo entre blocos, é de 120 segundos, $l = 15$ (30 mins), $b = 720$ (um dia), e $o = 60$ (2 horas). [9] [10]

src/cryptonote_config.h

[8] Os desenvolvedores de Monero conseguiram alterar o protocolo 11 vezes , com uma adopção de utilizadores e mineiros quase unánima todas as vezes : v1 Abril 18, 2014 (versão génese) [**?**]; v2 Março de 2016; v3 Setembro de 2016; v4 Janeiro de 2017; v5 Abril de 2017; v6 Setembro de 2017; v7 Abril de 2018; v8 e v9 Outubro de 2018; v10 e v11 Março de 2019; v12 Novembro de 2019. O repositório núcleo no *git* tem um ficheiro README contêm um resumo das mudanças de protocolo para cada versão.

src/hardforks/hardforks.cpp `mainnet_hard_forks[]`

[9] Em março de 2016 (v2 do protocolo), Monero mudou de 1 minuto entre blocos para 2 minutos [**?**]. Outros parâmetros de dificuldade sempre foram os mesmos.

[10] O algoritmo de dificuldade em Monero é talvez sub-optimal comparado com outros que são o último grito [**?**]. Felizmente é 'relativamente resiliente á mineração egoísta' [**?**], uma característica essencial.

As dificuldades de cada bloco não são guardadas na lista de blocos, alguêm que baixe uma cópia da lista de blocos e que verifique que todos os blocos são legítimos precisa de recalcular as dificuldades dos carimbos no tempo lá presentes. Existem algumas regras a considerar para os primeiros $b + l = 735$ blocos.

Regra n° 1: O bloco de génese é ignorado (bloco 0, com $d = 1$). Os blocos 1 e 2 têm $d = 1$.

Regra n° 2: Tenta-se obter a janela w com a qual se calculam os totais.

Regra n° 3: Depois de w blocos, cortam-se os extremos altos e baixos, isto é feito para b blocos. Se o montante dos blocos anteriores (menos w) é ímpar, remove-se mais um extremo baixo.

Regra n° 4: Depois de b blocos, obtêm-se os primeiros b blocos até ao n° $b + l$. Depois tudo continua normalmente - atrasado por l blocos.

Prova de trabalho em Monero

Monero tem usado diferentes algoritmos para a prova de trabalho, versões diferentes de funções hash com resultados de 32 bytes, em diferentes versões do protocolo. O original, chamado *Cryptonight* foi desenvolvido para ser relativamente inefficiente em placas gráficas (GPU), fpga, e architecturas de asic [**?**], quando comparadas a funções hash standard como o SHA256. Em abril de 2018 (v7 do protocolo), o algoritmo foi alterado um pouco, dando o começo a máquinas asic Cryptonight [**?**]. Em outubro de 2018 (v8) era Cryptonight V2 [**?**]. Em março de 2019 (v10) era Cryptonight-R [**?**]. Desde novembro de 2019 (v12), está a ser utilizado um algoritmo radicalmente novo chamado RandomX [**?**], que até a data é resistente a máquinas asic [**?**].

7.3 Quantidade monetária

Existem dois mequanismos básicos para criar dinheiro numa cripto-moeda que tenha como base uma lista de blocos. Primeiro, os criadores da moeda podem conjurar moedas e distribui-las a pessoas na mensagem de génese. Isto é muitas vezes chamado como 'airdrop'. Outras vezes os criadores dão a sí próprios um grande montante naquilo que se chama uma 'pre-mineração'. [**?**] Segundo, a moeda pode ser automáticamente distribuída como recompensa para minerar um bloco, á semelhança de minerar ouro. Existem dois tipos aqui. No modelo de bitcoin a quantidade monetária é limitada (21 milhões de btc). As recompensas dos blocos declinam lentamente para zero, e depois mais nenhum dinheiro é criado.

Monero baseia-se numa moeda chamada de Bytecoin que teve uma pre-mineração substancial, seguida de recompensas de bloco [**?**].

Monero não teve nenhuma pre-mineração, e como veremos as recompensas de bloco declinam lentamente para um pequeno valor, depois do qual todos os novos blocos têm a mesma recompensa (0.6 xmr). O que torna monero inflationário. Mas cuja inflação

decresce em termos percentuais ao ano, visto que a quantidade de xmr adicional ao já existente é cada vez mais irrelevante, no entanto os mineiros individuais sentem-se motivados.

7.3.1 Recompensa de bloco

Mineiros de blocos, antes de minerarem para um argumento, fazem um transacção para o mineiro, sem entradas e com pelo menos uma saída. [11]

O montante total de saída é igual á recompensa de bloco. Adicionalmente o mineiro recebe também as taxas de cada transacção presente no bloco. A recompensa está presente no bloco de forma clara e não encriptada. Nodes que recebem um bloco minerado têm de verificar que a recompensa do bloco está correcta. É também possível calcular a quantidade monetária actual ao somar todas as recompensas de todos os blocos. Note-se aqui como o argumento contra Monero, de que a quantidade de xmr não pode ser auditada não se aplica ás taxas, nem ás recompensas de base.

Para além de distribuir dinheiro, a recompensa de bloco incentiviza a mineração. Se não houvesse recompensas de blocos, porque que alguêm iria minerar? Talvez por altruísmo ou curiosidade. Porém, se existem poucos mineiros, é mais fácil para um atacante ter mais de 50% da velocidade de hash da rede. O que implica que o atacante é capaz de reescrever os blocos recentes na lista de blocos (ataque de 51%).

[12] É também por causa disto que as recompensas dos blocos não cheguam a ser zero. Com as recompensas de blocos, a competição entre mineiros leva ao aumento da velocidade de mineração total, até que o custo marginal de adicionar mais velocidade de mineração é maior do que a recompensa marginal de obter essa proporção de blocos minerados. Isto significa que enquanto uma cripto-moeda se torna mais valiosa, a sua velocidade total de mineração aumenta e torna-se progressivamente mais difícil e caro possuir >50% de todas as hashes a serem calculadas.

Bit shifting

Bit shifting é utilizado para calcular a recompensa base de bloco (secção 7.3.3, a recompensa de bloco pode ás vezes ser reduzida abaixo do montante base). Seja um inteiro A = 13 com a representação de bits [1101]. Se esses bits são deslocados por duas posições com o operador >> ou seja, A >> 2, obtêm-se [0011].01, que é igual a 3,25. Na realidade a parte depois da vírgula não conta, ou seja obtêm-se [0011] = 3. A operação de shift para a direita por n bits é equivalente á divisão inteira por 2^n.

[11] Uma transacção de mineiro pode ter qualquer número de saídas, porém actualmente a implementação núcleo só é capaz de fazer uma. Para mais, ao contrário de transacções normais, não existem nenhumas restrições explicitas no peso da transacção de mineiro. Estas estão limitadas pelo peso máximo de bloco.

[12] Á medida que um atacante possuí uma percentagem maior de hash da rede, para além de 50%, leva menos tempo para reescrever blocos passados. Dado um bloco com uma idade de x dias, uma velocidade de hash v, e uma velocidade de hash "honesta" v_h ($v > v_h$), irá levar $y = x * (v_h/(v - v_h))$ dias para reescrever a lista de blocos.

Calcular a recompensa base de bloco em Monero

Seja a quantidade total monetária M, e o seu 'limite' L = $2^{64}-1$. No início, a recompensa base de bloco era :

$$\text{B} = \text{(L-M)} \text{ ¿¿ } 20,$$

se M=0, então, em formato decimal :

$$\text{L} = 18,446,744,073,709,551,615$$
$$\text{B}_0 = (L-0) >> 20 = 17,592,186,044,415$$

. Estes números são unidades atómicas. Uma unidade atómica de Monero não pode ser dividida. Claramente unidades atómicas são ridiculas, L é mais do que 18 quintilões. Divide-se então L por 10^{12} o que move a vírgula para a esquerda, o que resulta nas unidades standard de Monero.

$$\frac{\text{L}}{10^{12}} = 18,446,744.073709551615$$
$$\text{B}_0 = \frac{(L-0) >> 20}{10^{12}} = 17.592186044415$$

A primeira recompensa base de bloco, dispersada ao pseudónimo thankful_for_today (que foi responsável pelo lançamento do projecto de Monero) no bloco génese [**?**], foram por volta de 17,6 xmr (apêndice C)! Os montantes em Monero são guardados na lista de blocos no formato de unidades atómicas. Á medida que os blocos são minerados M aumenta, o que reduz a recompensa base de bloco. Inicialmente, desde o bloco génese em Abril de 2014) os blocos de Monero eram minerados cada minuto. Em março de 2016, tornou-se dois minutos por bloco [**?**]. Para manter o mesmo ritmo de emissão [13] , a recompensa base de bloco foi duplicada. Isto só significa, que depois da mudança faz-se (L-M) >> 19, em vez de >> 20 para novos blocos. Actualmente a recompensa base de bloco é :

$$\text{B} = \frac{(L-M) >> 19}{10^{12}}$$

7.3.2 peso dinâmico de bloco

Seria bom poder minerar cada nova transacção para um bloco de imediato. O que acontece se alguêm submete imensas transacções maliciosamente? A lista de blocos, que guarda cada transacção, tornaria-se rápidamente enorme. Uma mitigação é um tamanho fixo de bloco (em bytes), assim o número de transacções por bloco é limitado. E se o número de transacções honestas aumenta? Cada autor de transacções iria competir para ter um lugar nos próximos blocos ao oferecer uma taxa aos mineiros. Os mineiros iriam como tal primeiro minerar as transacções com as taxas mais elevadas. Á medida que o volume total de transacções aumenta, as taxas tornam-se assim prohibitivamente elevadas para montantes de transacção pequenos. Somente aquele que paga mais é que tem o direito de incluír a sua transacção na lista de blocos.[14]

[13] Para uma comparação interessante entre as emissões de Bitcoin e Monero veja-se [**?**].

[14] Bitcoin tem um historial de um volume de transacções sobrecarregado. Este website (https://bitcoinfees.info/) mostra os montantes ridículos de taxas. Em Monero estes extremos, fixo vs sem limite, são evitados com o peso dinâmico de bloco.

Tamanho vs Peso

Desde que *Bulletproofs* foi adicionado (v8), os tamanhos de bloco e de transacção já não são considerados de forma estrita. O termo utilizado agora é *peso de transacção*. Na secção 5.5, uma *Bulletproof* occupa :

$$(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32,$$

portanto á medida que mais saídas são adicionadas, o espaço requisitado para as provas de domínio cresce de forma logarítmica. No entanto a verificação das provas de domínio é linear, portanto aumentar artificialmente os pesos de transacção "incluí"esse tempo extra de verificação.

O peso de um bloco é igual á soma dos pesos das suas transacções mais o peso da transacção de mineiro.

O peso de bloco a longo termo

Se blocos dinâmicos são permitidos de crescer rapidamente, então a lista de blocos pode rápidamente, tornar-se enorme [**?**]. Para mitigar isto, pesos de bloco máximo são amarrados por *pesos de bloco de longo termo*. Cada bloco tem, em adição ao seu peso normal, uma segunda métrica, que é calculada com base nos blocos anteriores. Ou seja é calculada a mediana do peso em longo termo. [15] A efectiva mediana de peso de longo termo está relacionada com a mediana dos 100000 blocos mais recentes. [16] [17]

src/cryptonote_core/blockchain.cpp `update_next_cumulative_weight_limit()`

peso de_bloco de_longo termo = min\{peso de bloco,
1.4 $*$ a anterior efectiva mediana de longo termo\}
efectiva mediana de longo termo = max\{300kB,
a mediana dos pesos de longo termo de 100000 blocos\}

Se os pesos de bloco normais se mantêm elevados durante um longo periodo de tempo, então irá levar pelo menos 50,000 blocos (por volta de 69 dias) para que a mediana effectiva de longo termo cresca por 40%. É esse o tempo que leva para que o peso de longo termo se torne na mediana.

A mediana de peso comulativo

O volume de transacções pode mudar dramaticamente num curto periodo de tempo, especialmente por volta das férias [**?**]. Para accomodar isto, permite-se uma flexibilidade a curto prazo dos pesos de bloco. Para suavizar essa variabilidade a mediana comulativa

[15] Bem como as dificuldades de bloco, os pesos de bloco normais e os de longo termo são calculados e guardados pelos nodes completos em vez de serem incluídos na lista de blocos.

[16] Os blocos feitos antes dos pesos de longo termo terem sido implementados têm pesos de longo termo iguais aos pesos normais. Portanto não existe preocupação sobre os detalhes de bloco de génese ou blocos antigos. Uma lista de blocos nova pode fazer decisões sensatas.

`CRYPTONOTE_REWARD_BLOCKS_WINDOW`

[17] No início, o termo de '300kB' era de 20kB, depois aumento para 60kB em março de 2016 (v2 do protocolo) [**?**], e tem sido 300kB desde abril de 2017 (v5 do protocolo) [**?**]. Este 'chão' não nulo na mediana do peso dinámico de bloco ajuda nas mudanças voláteis do volume de transacções transientes, quando este volume é baixo, especialmente no começo de adopção de Monero.

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_min_block_weight()`

de um bloco usa a mediana dos pesos normais dos últimos 100 blocos (incluíndo a própria).
Mediana de peso comulativo = max{300kB,
$min\{max$\{300kB,
mediana dos pesos dos 100blocos passados\},
50 ∗ mediana efectiva de longo termo\}\}
O próximo bloco a ser incluído na lista de blocos está constrangido da seguinte maneira: [18]

$$\text{peso máximo do bloco seguinte} = 2 * \text{Mediana de peso comulativo}$$

src/cryptonote basic/cryptonote_basic_impl.cpp `get_block_reward()`

Enquanto que o peso máximo de bloco pode subir 100 vezes acima da efectiva mediana do peso a longo termo depois de umas centenas de blocos, não pode subir mais do que 40% para além disso nos próximos 50,000 blocos. Como tal o crescimento do peso de bloco de longo termo está amarrado aos pesos de longo termo.
A curto prazo, os pesos podem surgir para além dos seus valores constantes.

7.3.3 Recompensa de bloco com multa

Para minerar blocos maiores do que a mediana comulativa, os mineiros têm de pagar uma multa, na forma de uma recompensa de bloco reduzida. Isto significa que existem duas zonas dentro do peso máximo do bloco : a zona sem multa, e a zona com multa. A mediana pode subir lentamente, permitindo blocos progressivamente maiores sem multa.

Se o peso de bloco pretendido é maior do que a mediana comulativa, então dada a recompensa base de bloco B, a multa P é :

$$\mathrm{P} = \mathrm{B} * ((\text{block\_weight}/\text{cumulative\_weights\_median}) - 1)^2$$

A recompensa de bloco é portanto : [19]

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

$$\mathrm{B}^{\text{actual}} = \mathrm{B} - \mathrm{P}$$

Ao usar a operação ˆ2 significa que as multas são sub-proporcional ao peso de bloco. Por exemplos : Um peso de bloco 10% maior do que as medianas comulativas anteriores só têm 1% de multa.
Um peso de bloco 50% maior têm uma multa de 25% .
Um peso de bloco 90% maior têm uma multa de 81% , etc [**?**]

[18] A mediana comulativa substituiu 'M100' (uma mediana semelhante) no protocolo v8. Multas e taxas descritas na primeira edição deste relatório [**?**] usavam M100.

[19] Antes de transacções confidenciais (RingCT) serem implementadas (v4), todos os montantes eram comunicados em texto claro e não encriptado, e em versões antigas os montantes eram separados em partes de valor igual, por exemplo 1244 → 1000 + 200 + 40 + 4. Para reduzir o tamanho de uma transacção de mineiro, a implementação do código fonte, cortava os bits menos significantes da recompensa de bloco (veja-se `BASE_REWARD_CLAMP_THRESHOLD`) em v2-v3). A quantidade pequena que era cortada não se perdia mas estava disponível para futuras recompensas de bloco.

É de esperar que os mineiros criem blocos maiores que a mediana comulativa, para a zona de multa, desde que o ganho com as taxas de transacção o justifique.

7.3.4 Taxa mínima dinámica

Actores maliciosos podem inundar a lista de blocos com transacções que possam ser usadas para poluír assinaturas em anel. O que gasta espaço na lista de blocos inútilmente. Para prevenir isto, Monero tem uma taxa mínima por byte de dados de transacção. src/cryptonote_core/blockchain.cpp `check_fee()`

[20] No começo era 0.01 XMR/KiB (durante a primeira versão do protocolo) [**?**], depois mudou para 0.002 XMR/KiB em Setembro de 2016 (v3) Em janeiro de 2017 (v4), uma taxa dinámica por KiB foi introduzida [**?**, **?**, **?**, **?**]. Depois com a introdução de *Bulletproofs* (v8), mudou de KiB para byte. A característica mais importante do algoritmo actual, é que previne taxas totais mínimas de exceder a recompensa de bloco (mesmo com recompensas de bloco pequenas e um peso do bloco grande)[**?**, **?**, **?**], pois pensa-se que isso causa instabilidade. [21]
[22]

O algoritmo da taxa

O algoritmo da taxa baseia-se á volta de uma transacção referência com um peso de 3000 bytes. Á semelhança de uma transacção básica `RCTTypeBulletproof2` com duas entradas e duas saídas, que normalmente tem por volta de 2600 bytes.[23] E com taxas consideradas para activar a multa quando a mediana está no seu mínimo (a zona sem multa mínima, ou seja 300kB) [**?**]. Por outras palavras, a multa induzida por um bloco com um peso de 303kB.

Primeiro, a taxa T que iguala a multa marginal MM que provém de adicionar uma transacção com peso PT a um bloco com um peso PB é :

$$\mathrm{T} = \mathrm{MM} = \mathrm{B} * ((\mathrm{[PB + PT]/mediana\_cumulativa} - 1)^2 - \mathrm{B} * (\mathrm{(PB/mediana\_cumulativa} - 1)^2$$

Define-se o factor de peso do bloco :

$$\mathrm{FP}_b = (\mathrm{BW/mediana\_cumulativa} - 1)$$

e o factor de peso de uma transacção :

$$\mathrm{FP}_t = (\mathrm{PT/mediana\_cumulativa})$$

[20] Este mínimo é enforçado pelo protocolo de consenso dos nodes, e não pelo protocolo da lista de blocos. A maioria dos nodes não propagam uma transacção a outros nodes se esta tem uma taxa menor do que o mínimo . Assim somente transacções que irão provavelmente ser mineradas é que são distribuídas pela rede. No entanto a rede aceita um blocos que contenham transacções com uma taxa menor do que o mínimo . O que implica que não é necessário manter a compatibilidade com algoritmos antigos de taxa. src/cryptonote_core/tx_pool.cpp `add_tx()`

[21] Créditos para os conceitos apresentados nesta secção são largamente devidos a Francisco Cabañas (a.k.a. 'ArticMine'), o arquitecto do sistema dinâmico de bloco e do sistema de taxa [**?**, **?**, **?**].

[22] A unidade KiB (kibibyte, 1 KiB = 1024 bytes) é diferente de kB (kilobyte, 1 kB = 1000 bytes).

[23] Uma transacção básica de uma entrada e duas saídas em Bitcoin pesa 250 bytes [**?**], ou 430 bytes para uma transacção com duas entradas e duas saídas.

De forma simples :

$$\mathrm{T} = \mathrm{B} * (2 * \mathrm{FP}_b * \mathrm{FP}_t + \mathrm{FP}_t^2)$$

Usando um bloco que pesa 300kB, com uma mediana comulativa standard de 300kB, e a transacção de referência com 3000 bytes,

$$\begin{aligned}
\mathrm{T}_{\text{ref}} &= \mathrm{B} * (2 * 0 * \mathrm{FP}_t + \mathrm{FP}_t^2) \\
\mathrm{T}_{\text{ref}} &= \mathrm{B} * \mathrm{FP}_t^2 \\
\mathrm{T}_{\text{ref}} &= \mathrm{B} * (\frac{\mathrm{PT}_{\text{ref}}}{\text{mediana_cumulativa}_{\text{ref}}})^2
\end{aligned}$$

Esta taxa é extendida sobre 1% da zona de multa (3000 de 300000). Extende-se a mesma taza sobre 1% de qualquer zona de multa com uma transacção de referência geralizada.

$$\begin{aligned}
\frac{\mathrm{PT}_{\text{ref}}}{\text{mediana_cumulativa}_{\text{ref}}} &= \frac{\mathrm{PT}_{\text{general-ref}}}{\text{mediana_cumulativa}_{\text{general}}} \\
1 &= (\frac{\mathrm{PT}_{\text{general-ref}}}{\text{mediana_cumulativa}_{\text{general}}}) * (\frac{\text{mediana_cumulativa}_{\text{ref}}}{\mathrm{PT}_{\text{ref}}}) \\
\mathrm{T}_{\text{general-ref}} &= \mathrm{T}_{\text{ref}} \\
&= \mathrm{T}_{\text{ref}} * (\frac{\mathrm{PT}_{\text{general-ref}}}{\text{mediana_cumulativa}_{\text{general}}}) * (\frac{\text{mediana_cumulativa}_{\text{ref}}}{\mathrm{PT}_{\text{ref}}}) \\
\mathrm{T}_{\text{general-ref}} &= \mathrm{B} * (\frac{\mathrm{PT}_{\text{general-ref}}}{\text{mediana_cumulativa}_{\text{general}}}) * (\frac{\mathrm{PT}_{\text{ref}}}{\text{mediana_cumulativa}_{\text{ref}}})
\end{aligned}$$

É possivel escalar a taxa com base no peso de uma transacção real para uma dada mediana, então por exemplo se a transacção é 2% da zona de multa, a taxa é duplicada.

$$\begin{aligned}
\mathrm{T}_{\text{geral}} &= \mathrm{T}_{\text{geral-ref}} * \frac{\mathrm{PT}_{\text{geral}}}{\mathrm{PT}_{\text{geral-ref}}} \\
\mathrm{T}_{\text{geral}} &= \mathrm{B} * (\frac{\mathrm{PT}_{\text{geral}}}{\text{mediana_cumulativa}_{\text{geral}}}) * (\frac{\mathrm{PT}_{\text{ref}}}{\text{mediana_cumulativa}_{\text{ref}}})
\end{aligned}$$

Isto equivale á taxa standard por byte :

$$\begin{aligned}
f^B_{standard} &= \mathrm{T}_{\text{geral}}/\mathrm{PT}_{\text{geral}} \\
f^B_{standard} &= \mathrm{B} * (\frac{1}{\text{mediana_cumulativa}_{\text{geral}}}) * (\frac{3000}{300000})
\end{aligned}$$

Quando o volume de transacções está abaixo da mediana, não existe nenhuma razão real para que as taxas estejam no nível de referência [**?**]. A taxa mínima é 1/5 do valor standard.

$$\begin{aligned}
f^B_{min} &= \mathrm{B} * (\frac{1}{\text{mediana_cumulativa}}) * (\frac{3000}{300000}) * (\frac{1}{5}) \\
f^B_{min} &= \mathrm{B} * (\frac{1}{\text{mediana_cumulativa}}) * 0.002
\end{aligned}$$

A mediana de taxa

Acontece que usar a mediana comulativa para as taxas permite um ataque de *spam*. Ao elevar a mediana de curto prazo ao seu valor mais elevado (50 x mediana de longo

termo), um ataquante pode usar as taxas mínimas para manter pesos de bloco elevados (relativo ao volume de transacções consequentes), com um custo bastante baixo. Para evitar este cenário as taxas são limitadas, para que a transacção entre no próximo bloco, com a menor mediana disponível, o que irá favorecer taxas elevadas, em todos os casos. [24]

mediana_mínima = $max\{300\text{kB}, min\{\text{mediana\_dos\_pesos\_de\_100\_blocos}, \text{mediana\_efectiva\_de\_longo\_ter}$

Favorecer taxas elevadas durante um crescente volume de transacções também facilita ajustar a mediana de curto prazo o que garante que as transacções não ficam á espera na mempool. Visto que os mineiros irão com uma maior probabilidade minerar para dentro da zona de multa.

src/crypto-note_core block-chain.cpp `verifica_tax`

A taxa mínima actual é portanto :[25]

$$f^{B}_{min-actual} = \text{B} * (\frac{1}{\text{mediana\_mínima}}) * 0.002$$

src/crypto-note_core block-chain.cpp `get_dyna-mic_base_ fee()`

Transaction fees

Como Cabañas disse na sua presentação perspicaz sobre este tópico [**?**], "As taxas dizem ao mineiro qual é a multa que os remetentes estão dispostos a pagar, para que a transacção seja minerada."Os mineiros enchem os blocos, com transacções com uma taxa elevada primeiro e depois com taxas decrescentes [**?**], em que se assume que todas as transacções têm um peso igual. Assim para que se entre dentro da zona de multa têm de haver muitas transacções com taxas elevadas. Isto significa que, é provável que o limite do peso de bloco, só é atingido se as taxas totais são pelo menos 3 até 4 vezes a recompensa base de bloco. Nesta altura a multa é tão grande que, a recompensa de bloco é nula. [26]

[24] Um ataquante pode gastar somente o suficiente em taxas para que a mediana de curto prazo atinga 50 x a mediana de longo termo. Com as recompensas base de bloco actuais de 2 xmr, um atacante sofisticado pode aumentar a mediana de curto prazo por 17% todos os 50 blocos, e alcançar o limite superior depois de 1300 blocos (por volta de 43 horas), gastando 0.39*2 XMR por bloco, para um custo total de 1000 xmr (ou por volta de 65k usd, ao preço actual), e depois voltar á taxa mínima. Quando a mediana de taxa é igual á taxa paga na zona sem multa, a taxa mínima total para encher a zona sem multa é 0.004 xmr (por volta de 0.26 usd, ao preço actual). Se a mediana é igual á mediana de longo termo, seria no cenário de *spam* 1/50 da zona de multa. Portanto seria só 50x a mediana de curto prazo, para 0.2 xmr por bloco (13 usd por bloco). Isto resulta em 2.88 xmr por dia vs 144 xmr por dia (durante 69 dias, até que a mediana de longo termo sobe por 40%) para manter cada bloco com um peso igual a 50 x a mediana de longo termo do peso de bloco. O caso de 1000xmr iria valer a pena no caso anterior, mas não no posterior. Isto reduz para 300xmr, 43xmr de manutenção, na cauda de emissão.

[25] Para verificar que uma dada taxa é correcta, permite-se uma tolerância de 2% em $f^{B}_{min-actual}$. Por causa de conflitos com o tipo 'inteiro' (é preciso calcular as taxas antes que os pesos estejam determinados). Isto significa que uma taxa effectiva mínima é $0.98 * f^{B}_{min-actual}$.

src/crypto-note_core block-chain.cpp `check_fee()`

[26] A multa marginal incorrida por encher os últimos bytes de um bloco, pode ser considerada uma transacção comparável a outra transacções. Para que uma transacção ou um conjunto de transacções preencha esse espaço restante de um bloco, cada transacção indiviual tem de ter uma taxa maior do que a multa senão o mineiro irá optar por manter a recompensa marginal, e discartar tais transacções. Esta última recompensa marginal, assumindo um bloco cheio de pequenas transacções, precisa de pelo menos 4 vezes a recompensa base de bloco no total, em taxas para ser superada. Se os pesos de transacção são maximizados (50% da zona livre de multa, i.e. 150kB), então a mediana é minimizada (300kB). A última transacção marginal requer pelo menos 3x no total de taxas.

Para calcular as taxas de uma transacção, a implementação núcleo de Monero utiliza multiplicadores de 'prioridade' . Uma transacção 'lenta' usa uma taxa mínima directamente, 'normal' é a taxa standard 5x, 'rápida' é 25x e 'super urgente' é 1000x a taxa mínima. Se todas as transacções são 'rápidas' estas podem atingir 2.5% da zona de multa, e um bloco só com transacções 'super urgentes', enche 100% da zona de multa. Uma consequência importante de pesos de bloco dinámicos é que a média de todas as taxas de um bloco irá tender a uma mágnitude menor que, ou pelo menos igual á recompensa do bloco. Transacções que competem para um espaço no bloco com taxas elevadas, lideram a uma maior oferta de espaço de bloco e transacções mais baixas.[27] Este mecanismo de feedback é uma boa defesa contra a ameaça do 'mineiro egoísta' [**?**].

src/wallet/wallet2.cpp `get_fee_multiplier()`

7.3.5 Cauda de emissão

Suponha-se uma cripto-moeda com uma quantidade monetária fixa e um peso de bloco dinámico. Depois de algum tempo as recompensas de bloco declinam para zero. Sem mais multas no crescimento do peso de blocos, os mineiros adicionam qualquer transacção com uma taxa não nula aos seu blocos.

O peso dos blocos stabiliza á volta da média da quantidade de transacções submetida á rede, e os remetentes usam sempre as taxas mínimas possíveis, o que seria zero segundo a secção 7.3.4.

Isto introduz uma situação instável e insegura. Mineiros têm pouco a nenhum incentivo para minar novos blocos, á medida que os retornos de investimentos declinam, a quantidade de hashes da rede cai. O tempo entre blocos permanece o mesmo á medida que a dificuldade ajusta, mas o custo de executar um ataque de 51% pode se tornar fazível. Se as taxas mínimas de transacção são forçadas a serem não nulas, então a ameaça do 'mineiro egoísta' torna-se realista. [28]

Monero previne isto de acontecer, ao não permitir que a recompensa de blocos caia abaixo de 0.6 XMR (0.3 XMR por minuto). Quando a seguinte condição é mantida :

$$0.6 > ((L - M) >> 19)/10^{12}$$
$$\mathrm{M} > \mathrm{L} - 0.6 * 2^{19} * 10^{12}$$
$$\mathrm{M}/10^{12} > \mathrm{L}/10^{12} - 0.6 * 2^{19}$$
$$\mathrm{M}/10^{12} > 18,132,171.273709551615$$

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

A lista de blocos de Monero irá entrar numa assim chamada 'cauda de emissão', com uma recompensa de bloco constante de 0.6 xmr (0.3 xmr/minuto). Isto corresponde a uma inflação inicial de 0.9% anualmente, que decresce ano após ano, e tende para zero. [29]

[27] A medida que as recompensas de bloco declinam, e a mediana aumenta devido a uma adopção crescente, as taxas deviam tornar-se gradualmente mais e mais baixas. Em termos do poder de compra real, isto poderá ter menos impacto no custo de transacções se o valor de Monero aumenta devido á adopção e a deflação económica.

[28] O caso de uma quantidade monetária limitada e um tamanho de bloco fixo, como em Bitcoin, também é considerado como instável [**?**].

[29] A emissão de cauda de Monero, começa aproximadamente em Maio de 2022 [**?**]. A quantidade monetária L

7.3.6 Transacção de mineiro: `RCTTypeNull`

O mineiro de um bloco tem o direito de posse das taxas de transacção e da recompensa do bloco. Isto é feito através de uma transacção de mineiro (também conhecido como *coinbase transaction*), que é similar a uma transacção normal.
[30]

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_miner_tx()`

Os montantes das saídas de uma transacção de mineiro não pode ser mais do que a soma das taxas de transacção mais a recompensa de bloco e são comunicadas em texto claro e não encriptado. A posse destes fundos é enviada a um endereço oculto standard, com uma chave pública de transacção incluída no campo extra. Os fundos estão presos, sem a possibilidade de serem gastos durante 60 blocos [**?**]. Em vez de uma ou mais entradas, a altura de bloco é guardada (i.e. "Eu tenho posse da recompensa deste bloco e de suas taxas"). Em que a *altura* é o número do bloco, na lista de blocos. [31]
[32].

Desde que *RingCT* foi implementado em Janeiro de 2017 (v4) [**?**], quem saca uma nova cópia da lista de blocos calcula um compromisso para o montante de transacção de mineiro a, em $C = 1G + aH$, e guarda isso para referência (cada a pesa 32 bytes). Isto permite aos mineiros gastar as suas próprias saídas de transacção como se fossem saídas de transacção normais. Estas são postas em aneis com outras saídas de transacção normais ou de mineiro.

src/blockchain_db/blockchain_db.cpp `add_transaction()`

7.4 A estrutura da Blockchain

O estilo da lista de blocos de Monero é simples.

Começa com uma mensagem de génese de qualquer tipo (neste caso, básicamente uma transacção de mineiro que dispersa a primeira recompensa de bloco), que constitui o bloco de génese (apêndice C). O próximo bloco contêm uma referência ao bloco anterior, na forma de um ID de bloco.

O ID de bloco é uma hash do cabeçalho de um bloco anterior (uma lista de informação sobre o bloco). Ou seja, uma raíz de Merkle, e o número de transacções (incluí a transacção de mineiro). A raíz de Merkle junta todos os ID's de transacção do bloco. O ID de uma transacção é simplesmente a hash dessa transacção. [33]

src/cryptonote_core/cryptonote_tx_utils.cpp `generate_genesis_block()`

src/cryptonote_basic/cryptonote_format_utils.cpp `get_block_hashing_blob()`

src/cryptonote_basic/cryptonote_format_utils.cpp `calculate_block_`

src/cryptonote_core/cryptonote_chain.cpp `is_tx_spendtime_unlocked()`

irá ser alcançada em Maio de 2024. Devido á prova de domínio *Bulletproof*, irá ser impossível enviar mais do que a quantidade L de uma só saída (e assumindo que o software da carteira consegue lidar com tal montante).

[30] Aparentemente a uma dada altura as transacções de mineiro podiam ser construídas de tal forma, que usavam versões de transacção descontinuadas, o que permitia incluír componentes de transacções normais (RingCT). Esta questão foi resolvida na versão de protocolo v12 depois deste relatório *hackerone* ter sido publicado [**?**].

[31] Na versão actual os mineiros podem optar por não recolherem todas as taxas. Desta forma os restos são adicionados ao plano de emissão para mineiros futuros.

[32] A transacção de mineiro não pode ser bloqueada, para mais ou menos de 60 blocos. Se o bloco em questão é o décimo, a sua altura de desbloqueio é 70, ou seja a saída presente na transacção de mineiro pode ser utilizada como entrada no bloco n° 70 ou mais tarde.

[33] +1 accounts for the miner tx.

$$\text{ID de bloco} = \mathcal{H}_n(\text{cabeçalho do bloco}, \text{raíz de Merkle}, \#\text{transacções} + 1)$$

Para produzir um novo bloco, é preciso produzir hashes, que são prova de trabalho (do inglés: *proof of work*), ao alterar o valor de *nonce*. Isto é feito até que a hash satisfaça a condição da dificuldade alvo [34]. A prova de trabalho e o ID de bloco fazem uma hash da mesma informação, porém com funções de hash diferentes. Blocos são minerados por :
enquanto que :

$$(PoW_{resultado} * dificuldade) > 2^{256} - 1,$$

continua-se a alterar a *nonce* para recalcular :

$$PoW_{resultado} = \mathcal{H}_{PoW}(\text{cabeçalho de Bloco}, \text{raíz de Merkle}, \#\text{transacções} + 1)$$

7.4.1 ID de transacção

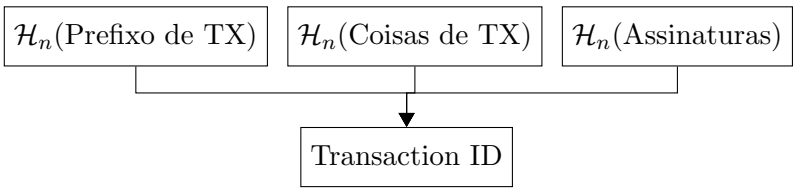
ID's de transacção são parecidos á mensagem assinada pelas entradas de assinatura tipo MLSAG (secção 6.1.2), mas incluem a assinatura MLSAG.

Faz-se uma hash da seguinte informação :

- Prefixo de transacção = {Versão da era da transacção (e.g. ringCT = 2), entradas {offsets de chave, imagens de chave}, saídas {endereços ocultos}, extra {chave pública de transacção, ID de pagamento codificado, misc.}}
- Coisas de transacção = {tipo de assinatura (`RCTTypeNull` ou `RCTTypeBulletproof2`), taxa de transacção, compromissos a pseudo saídas para entradas, ecdhInfo (montantes ecriptados ou em texto claro), e compromissos de saída}
- Assinaturas = {MLSAGs, provas de domínio}

src/cryptonote_basic/cryptonote_format_utils.cpp `calculate_transaction_hash()`

Neste diagrama de árvore a seta preta indica uma hash de entradas.



Em vez de uma 'entrada', uma transacção de mineiro contêm a altura do bloco actual. Isto garante que o ID de transacção de mineiro, é sempre único, para uma procura de ID mais fácil. Note-se que uma transacção de mineiro não contêm assinaturas; $\mathcal{H}_n(\text{Signatures}) \to 0$.

[34] Em Monero um mineiro típico (de `https://monerobenchmarks.info/` na altura deste relatório pode fazer menos que 50,000 hashes por segundo, portanto menos do que 6 milhões de hashes por bloco. Isto significa que o argumento de nonce não precisa de ser assim tão grande. A nonce de Monero tem 4 bytes de comprimento, e seria estranho se um mineiro usasse todos os bits.

7.4.2 Árvore merkle

Alguns monerianos poderam querer livrar-se de dados da sua cópia da lista de blocos. Por exemplo, uma vez que as provas de domínio e as assinaturas de entrada foram verificadas, a única razão para manter essa informação é para poder propagar isso a outros nodes. E por sua vez estes também podem verificar o mesmo.

Para facilitar a 'poda' de dados de transacção, e para que estes estejam mais organizados dentro de um bloco, usa-se uma árvore de merkle (Ralph C. Merkle)[**?**], que é uma árvore binária de hashes. Qualquer ramo numa árvore de merkle pode ser podado, se a hash da raíz da árvore é mantida.
[35]

Um exemplo de uma árvore de merkle, com quatro folhas como transacções apresenta-se na Figura 7.1. [36]

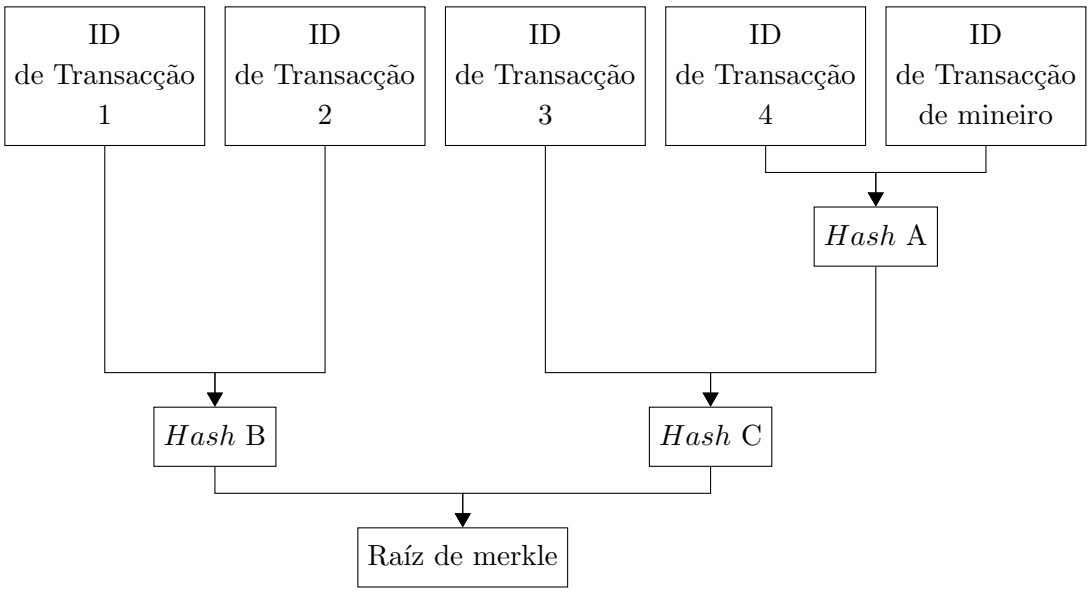


Figura 7.1: árvore de merkle

Uma raíz de merkle é inherentemente uma referência a todas as transacções incluídas.

[35] O primeiro método de poda foi adicionado na versão 0.14.1 da implementação núcleo de Monero (Março de 2019, coincidente com o protocolo v10). Depois de verificar uma transacção, nodes completos podem apagar todos os dados de assinatura. Isto incluí *Bulletproofs*, MLSAG's, e pseudo compromissos de saída. A hash das assinaturas $\mathcal{H}_n$(assinaturas) para computar a ID de transacção. Em cada 8 transacções isto é feito a 7 delas. Portanto cada transacção está guardada por pelo menos por 1 em cada 8 nodes. Isto reduz o espaço da lista de blocos por volta de 2/3. [**?**]

[36] Um erro no código fonte da raíz de Merkle levou a um sério mas aparentemente não crítico ataque no dia 4 de setembro de 2014 [**?**].

7.4.3 Blocos

Um bloco é basicamente um cabeçalho e algumas transacções. O cabeçalho de bloco tem informação importante sobre cada bloco. As transacções de um bloco são referenciadas com a raíz de merkle. Apresenta-se aqui o conteúdo de um bloco. Os nossos leitores podem encontrar um exemplo de um bloco real no apêndice B.

- cabeçalho de bloco:
  - **Versão maior**: para marcar mudanças no protocolo.
  - **Versão menor**: foi usado para votar, agora só mostra a versão maior outra vez.
  - **carimbo temporal**: UTC (Coordinated Universal Time) tempo do bloco. Escrito pelo mineiro, não é verificado mas não será aceite se for menor do que a mediana dos últimos 60 blocos.
  - **ID do bloco anterior**: Referência ao bloco anterior, característica essencial de uma lista de blocos.
  - **Inteiro**: Um inteiro de 4 bytes que os mineiros utilizam para minerar. Quem verifica o bloco utiliza este valor para calcular a hash da prova de trabalho.

  src/cryptonote_core/blockchain.cpp `check_block_timestamp()`
- transacção de mineiro: Envia a recompensa de bloco e as taxas das transacções para o mineiro do bloco.
- ID's de transacção: Referência as transacções normais. Estas juntamente com a transacção de mineiro podem ser utilizadas para calcular a raíz de merkle.

Adicionalmente aos dados em cada transacção (secção 6.2), a seguinte informação é guardada :

- Versão maior e Versão menor: Inteiros variáveis ≤ 9 bytes.
- Carimbo temporal: Inteiro variável ≤ 9 bytes.
- ID do bloco anterior: 32 bytes.
- *Nonce*: 4 bytes. Este tamanho pode ser extendido através do campo extra na transacção do mineiro.[37]
- Transacção de mineiro : 32 bytes para uma chave pública de transacção (+ 1 byte para o tag 'extra'). Inteiros variáveis para o tempo de desbloqueio, ou seja a altura de bloco e o montante.

  Depois de baixar a lista de blocos, são também precisos 32 bytes para guardar um compromisso de montante $C = 1G + aH$ (só para montantes de transacção de mineiro pós-RingCT)
- ID's de transacção: 32 bytes cada

[37] Dentro de cada transacção existe um campo extra que pode conter mais ou menos dados arbitrários. Se um mineiro precisa de um domínio maior para o argumento nonce, então é possível adicionar ou alterar dados no campo extra da transacção de mineiro, para 'extender' o domínio do argumento nonce [**?**].

Parte II

Extensões

CAPÍTULO 8

Provas de conhecimento de transacções

Monero é uma cripto-moeda, e como qualquer moeda os seus usos são complexos. Desde contabilidade empresarial, á presença na bolsa e arbitragem legal, diferentes entidades estão interessadas nas transacções efectuadas. Como é possivel determinar que dinheiro recebido veio de uma pessoa específica? Ou como se prova que uma certa transacção foi efectuada para alguêm, apesar de acusações contrárias? Remetentes e destinatários no registro público de Monero são ambiguos. Visto que os montantes em Monero estão escondidos do público, como é possível provar a posse de um montante, sem que as chaves privadas sejam comprometidas?

São considerados diferentes tipos de asserções sobre transacções, algumas das quais estão implementadas em Monero e disponíveis com ferramentas da carteira. Apresentam-se também estruturas para auditar o saldo de uma pessoa ou organização, o que não requer vazar informação sobre transacções futuras.

8.1 Provas de transacção em Monero

As provas de transacção em Monero estão no processo de serem renovadas [**?**]. As provas actualmente implementadas são todas 'versão 1', e não incluem a separação de domínios. Aqui só irão ser descritas as provas mais avancadas, porque estão implementadas, poderão ser implementadas [**?**], ou provas hipotéticas que servem só de teoria (secções 8.1.5 [**?**], e 8.2.1).

8.1.1 Provas de transacção com múltiplas bases

Existem alguns detalhes a ter em conta ao prosseguir. A maioria das provas de transacção em Monero involvem provas com múltiplas bases (secção 3.1). Sempre que

relevante o separador de domínio é :

$$T_{txprf2} = \mathcal{H}_n(\text{“TXPROOF\_V2”}).$$

A mensagem que é assinada é usualmente (salvo indicação em contrário) :

$$\mathtt{m} = \mathcal{H}_n(\mathtt{tx\_hash,\ message}).$$

Em que `tx_hash` é a ID de transacção relevante ( ) e `message` é uma mensagem opcional que os provadores ou entidades terceiras podem fornecer para dar a certeza que a prova foi de facto feita e não roubada [1] .

As provas estão codificadas em base-58, um esquema de código de binário para texto, primeiro introduzido para Bitcoin [**?**]. Verificar estas provas envolve sempre primeiro descodificar da base-58 de volta a binário. Note-se que os verificadores também precisam de ter acesso á lista de blocos, para que através da ID de transacção possam extraír informação como endereços ocultos. [2]

A estrutura de prefixo de chave em provas está algo torta, em parte por causa de desenvolvimentos acumulados que ainda não reorganizaram isto. Desafios para provas, base-2 ‘versão 2’, são construídas com este formato, em que se ‘chave 1 base’ é G então a sua posição no desafio é preenchida com 32 bytes de zero.

$$c = \mathcal{H}_n(\mathtt{m}, \text{chave pública 2, prova parte 1, prova parte 2, } T_{txprf2},\\ \text{chave pública 1, chave base 2, chave base 1}).$$

8.1.2 Provar a criação de uma entrada de transacção (`SpendProofV1`)

Seja que um remetente fez uma transacção, e agora precisa de o provar. Claramente, ao refazer a assinatura á entrada de transacção numa outra mensagem, qualquer verificador teria de concluír que o remetente também fez a assinatura original. Ao refazer *todas* as assinaturas para cada entrada de uma transacção significa que o remetente tem de ter feito a transacção inteira.

Alguêm que fez uma assinatura a uma entrada, não implica que tenha assinado todas as entradas (capítulo 11).

[3]

Uma assim chamada ‘SpendProof’ (do inglés : *prova de gasto*) contêm novas assinaturas para cada entrada de transacção. Importante é que a prova de gasto re-utiliza os membros de anel originais para evitar a identificação do verdadeiro signatário através da intersecção de aneis.

src/wallet/wallet2.cpp `get_spend_proof()`

Provas de gasto são implementadas em Monero, o provador concatena o prefixo “`SpendProofV1`”com a lista de assinaturas. Note-se que este prefixo está em texto claro e não encriptado

[1] Bem como na secção 3.6, funções hash deviam ter um separador de domínio, através de um prefixo com tags. As provas de transacção actualmente implementadas em Monero não têm nenhuma separação de domínio. Portanto todos os tags neste capítulo são características ainda *não* implementadas.

[2] Os ID’s de transacção são usualmente comunicados separadamente das provas.

[3] Como veremos no capítulo 11, alguêm ter assinado uma entrada não implica que tenha assinado todas as entradas.

visto que o seu propósito é de ser lido por uma pessoa.

Prova de gasto

Provas de gasto inesperadamente não utilizam MLSAG, mas o esquema original de assinaturas em anel que foi utilizado no primeiro protocolo de transacção (pre-RingCT) [**?**].

(a) Calcule a imagem de chave $\tilde{K} = k_\pi^o \mathcal{H}_p(K_\pi^o)$.

(b) Gera-se o número aleatório $\alpha \in_R \mathbb{Z}_l$ e $c_i, r_i \in_R \mathbb{Z}_l$ para cada $i \in \{1, 2, ..., n\}$ mas excluíndo $i = \pi$.

(c) Computa-se
$$c_{tot} = \mathcal{H}_n(\mathfrak{m}, [r_1 G + c_1 K_1^o], [r_1 \mathcal{H}_p(K_1^o) + c_1 \tilde{K}], ..., [\alpha G], [\alpha \mathcal{H}_p(K_\pi^o)], ..., \text{etc.})$$

(d) Define-se o desafio verdadeiro
$$c_\pi = c_{tot} - \sum_{i=1, i\neq\pi}^{n} c_i$$

(e) Define-se $r_\pi = \alpha - c_\pi * k_\pi^o \pmod{l}$.

A assinatura é :
$$\sigma = (c_1, r_1, c_2, r_2, ..., c_n, r_n).$$

Verificação

Para verificar uma prova de gasto de uma dada transacção, o verificador confirma que todas as assinaturas em anel são válidas usando informação encontrada na transacção original em questão (imagens de chave, offsets de saída etc)

src/wallet/wallet2.cpp `check_spend_proof()`

(a) Compute-se
$c_{tot} = \mathcal{H}_n(\mathfrak{m}, [r_1 G + c_1 K_1^o], [r_1 \mathcal{H}_p(K_1^o) + c_1 \tilde{K}], ..., [r_n G + c_n K_n^o], [r_n \mathcal{H}_p(K_n^o) + c_n \tilde{K}])$

(b) Verifica-se que
$$c_{tot} \stackrel{?}{=} \sum_{i=1}^{n} c_i$$

Funciona porque

Note-se como este esquema é o mesmo como o bLSAG (Secção 3.4) quando só existe um membro de anel. Para adicionar um membro desvio, em vez de pôr o desafio $c_{\pi+1}$ dentro de uma nova hash de desafio, o membro é adicionado para a hash original. Como a equação seguinte :
$$c_s = c_{tot} - \sum_{i=1, i\neq s}^{n} c_i$$
é verdade para cada index s, um verificador não terá forma de identificar o desafio verdadeiro. Para mais, sem o conhecimento de k_π^o o provador não teria conseguido definir r_π .

8.1.3 Provar a criação de uma saída de transacção (`OutProofV2`)

Agora suponha-se que um remetente enviou dinheiro a alguêm (uma saída) e quer prová-lo. Saídas de transacção contêm principalmente 3 componentes : o endereço do destinatário, o montante enviado e a chave privada de transacção. Os montantes estão em código, portanto só é necessário o endereço e a chave privada de transacção. Se a chave privada de transacção é perdida será impossivel fazer uma prova de saída. [4]

A tarefa aqui é de mostrar que o endereço oculto foi feito pelo endereço do destinatário, e deixar que verificadores reconstruam o compromisso de saída. Isto é feito começando pelo segredo partilhado entre o remetente e o destinatário :

$$rK^v.$$

Prova-se a criação do mesmo, e que corresponde á chave pública de transacção e ao endereço do destinatário. Isto é conseguido ao fazer uma assinatura com duas bases, nomeadamente G e K^v. Os verificadores usam o segredo partilhado para verificar o recipiente (secção 4.2), descodificar o montante (secção 5.3), e reconstruir o compromisso de saída (secção 5.3). São presentados aqui detalhes para endereços normais bem como para sub-endereços.

src/wallet/wallet2.cpp
`check_tx_proof()`

Prova de saída (`OutProof`)

Para gerar uma prova de saída dirigida a um endereço

$$(K^s, K^v)$$

ou a um sub-endereço

$$(K^{s,i}, K^{v,i})$$

com uma chave privada de transacção r em que o segredo partilhado entre o remetente e o destinatário é rK^v. Note-se que uma chave pública de transacção guardada nos dados de transacção é rG para um endereço normal ou então $rK^{s,i}$ para um sub-endereço.

src/crypto/crypto.cpp
`generate_tx_proof()`

(a) Gera-se um número aleatório $\alpha \in_R \mathbb{Z}_l$, e calcula-se
   i. *Endereço normal*: αG e αK^v
   ii. *Sub-endereço*: $\alpha K^{s,i}$ e $\alpha K^{v,i}$

(b) Calcula-se o desafio
   i. *Endereço normal*:[5]
   $$c = \mathcal{H}_n(\mathfrak{m}, [rK^v], [\alpha G], [\alpha K^v], [T_{txprf2}], [rG], [K^v], [0])$$
   ii. *Sub-endereço*:
   $$c = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [\alpha K^{s,i}], [\alpha K^{v,i}], [T_{txprf2}], [rK^{s,i}], [K^{v,i}], [K^{s,i}])$$

(c) Define-se a resposta[6] $r^{resp} = \alpha - c * r$.

(d) A assinatura é : $\sigma^{outproof} = (c, r^{resp})$.

[4] Uma 'prova de saída' (OutProof) que mostra que uma saída de transacção, *saí* do provador. Uma 'prova de entrada' (secção 8.1.4) mostra que uma saída de transacção, *entra* para o endereço do provador.

[5] Aqui o valor '0' são 32 bytes a zero.

[6] Due to the limited number of available symbols, we unfortunately used r for both responses and the transaction private key. Superscript 'resp' for 'response' will be used to differentiate the two when necessary.

Um provador gera várias *OutProofs*, e envia estas para o verificador. Ele concatena o string prefixo "`OutProofV2`"com a lista de provas, em que cada item (codificado na base-58) consiste no segredo partilhado entre o remetente e o destinatário rK^v (ou $rK^{v,i}$ para um sub-endereço), e o correspondente $\sigma^{outproof}$. Assume-se que o verificador sabe o endereço respectivo para cada prova.

src/wallet/wallet2.cpp `get_tx_proof()`

Verificação

(a) Calcula-se o desafio

i. *Endereço normal*:

$$c' = \mathcal{H}_n(\mathfrak{m}, [rK^v], [r^{resp}G + c * rG], [r^{resp}K^v + c * rK^v], [T_{txprf2}], [rG], [K^v], [0])$$

ii. *Sub-endereço*:

$$c' = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [r^{resp}K^{s,i} + c*rK^{s,i}], [r^{resp}K^{v,i} + c*rK^{v,i}], [T_{txprf2}], [rK^{s,i}], [K^{v,i}], [K^{s,i}])$$

src/crypto/crypto.cpp `check_tx_proof()`

(b) Se $c = c'$ então o provador sabe r, e rK^v é um segredo partilhado legítimo entre rG e K^v (enp).

(c) O verificador devia verificar que o dado endereço do destinatário pode ser usado para criar um endereço oculto a partir da transacção relevante (trata-se da mesma computação para endereços normais bem como para sub-endereços)

$$K^s \stackrel{?}{=} K_t^o - \mathcal{H}_n(rK^v, t)$$

(d) Eles também deviam descodificar o montante b_t, calcular a máscara y_t, e tentar reconstruir o compromisso de saída correspondente. [7]

$$C_t^b \stackrel{?}{=} y_t G + b_t H$$

8.1.4 Provar a posse de uma saída (`InProofV2`)

Uma prova de saída (*OutProof*) mostra que o provador enviou uma saída para um endereço, enquanto que uma prova de entrada (*InProof*) mostra que uma saída foi recebida por um certo endereço. É essencialmente o outro 'lado' do segredo partilhado entre o remetente e o destinatário, rK^v. Desta vez o provador prova conhecimento de k^v em K^v, e que em combinação com a chave pública rG, apareçe o segredo partilhado, $k^v * rG$.

Uma vez que o verificador tem rK^v, ele verifica se o endereço oculto correspondente é

[7] Uma prova de saída válida não significa necessáriamente que o destinatário considerado, é o destinatário verdadeiro. Um provador malicioso podia gerar uma chave de ver aleatória K'^v, calcular $K'^s = K^o - \mathcal{H}_n(rK'^v, t) * G$, e oferecer (K'^s, K'^v) como o recipiente nominal. Ao recalcular o compromisso de saída, os verificadores podem ter mais confiança que o endereço de destinatário em questão é legítimo. Contudo, um provador e um destinatário podiam colaborar para codificar o compromisso de saída usando K'^v, enquanto que o endereço oculto usa (K^s, K^v). Desde que o destinatário teria de saber a chave privada k'^v (assumindo que o montante na saída ainda é suposto de ser gasto), é questionável o qual a utilidade para esse tipo de decepção. Porque que o destinatário não usaria simplesmente (K'^s, K'^v) (ou outro endereço) para a saída inteira? Desde que a computação de C_t^b está relacionada com o recipiente, considera-se o processo de verificação *OutProof* adequado. Por outras palavras, neste processo, o provador não pode enganar os verificadores sem uma coordinação com o destinatário.

possuído pelo endereço do provador através de :

$$K^o - \mathcal{H}_n(k^v * rG, t) * G \stackrel{?}{=} K^s.$$

Ao fazer uma prova de entrada para cada chave pública de transacção na lista de blocos, o provador revela todas as saídas que possui.Dar a chave de ver directamente ao verificador têm o mesmo efeito, mas com a diferença de que o verificador seria também capaz de identificar a posse de saídas criadas no futuro. Com as provas de entrada o provador retêm o controlo das suas chaves privadas, ao custo do tempo que leva a provar e verificar se cada saída lhe pertence ou não.

A prova de entrada (`InProof`)

Uma prova de entrada, é construída da mesma forma que uma prova de saída, excepto que as chaves base agora são :

$$\mathcal{J} = \{G, rG\},$$

as chaves públicas são :

$$\mathcal{K} = \{K^v, rK^v\},$$

e a chave signatária é : k^v (em vez de ser r). Mostra-se só o passo de verificação para clarificar o que isto significa. Note-se que a ordem do prefixo de chave muda, rG e K^v trocam de posições para coincidir com a função que têm.É possivel enviar uma multitude de provas de entrada ao verificador, estas são respectivas a várias saídas possuídas pelo mesmo endereço. Estas provas tem o prefixo "`InProofV2`", e cada uma delas (codificada na base-58) contêm o segredo partilhado entre o destinatário e o remetente rK^v (ou $rK^{v,i}$), e a prova de entrada $\sigma^{inproof}$.

src/wallet/wallet2.cpp `get_tx_proof()`

Verificação

(a) Calcula-se o desafio

i. *Endereço normal*:

$$c' = \mathcal{H}_n(\mathfrak{m}, [rK^v], [r^{resp}G + c*K^v], [r^{resp}*rG + c*k^v*rG], [T_{txprf2}], [K^v], [rG], [0])$$

ii. *Sub-endereço*:

$$c' = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [r^{resp}K^{s,i} + c*K^{v,i}], [r^{resp}*rK^{s,i} + c*k^v*rK^{s,i}], [T_{txprf2}], [K^{v,i}], [rK^{s,i}], [K^{s,i}])$$

src/crypto/crypto.cpp `check_tx_proof()`

(b) Se $c = c'$ então o provador sabe que k^v e $k^v * rG$ é um segredo partilhado entre K^v e rG (enp).

Provar a posse de uma saída através do endereço oculto

Enquanto que uma prova de entrada mostra que um endereço oculto foi construído por um endereço específico, isso não quer dizer que o provador possa *gastar* essa saída. Só quem pode gastar uma saída realmente a possuí.

Provar a posse de uma saída, uma vez que a prova de entrada está completa, é tão simples como assinar uma mensagem com a *chave de gasto*.

[8]

8.1.5 Prova de não-gasto de uma saída numa transacção

Para provar que uma saída está gasta ou não gasta pode-se recriar a imagem de chave com uma prova de múltiplas bases em :

$$\mathcal{J} = \{G, \mathcal{H}_p(K^o)\}$$

e :

$$\mathcal{K} = \{K^o, \tilde{K}\}.$$

Enquanto que isto funciona, os verificadores têm de descobrir a *imagem de chave*, o que também revela quando uma saída não gasta irá ser gasta no futuro.

Acontece que pode-se provar que uma saída não foi gasta numa transacção específica sem revelar a imagem chave. Para além disso pode-se provar que a saída actualmente não está gasta, ao extender esta prova de não-gasto [**?**] a 'todas as transacções em que esta saída foi incluída como membro de anel'.

[9]

Mais especificamente, as provas de não-gasto revelam que uma dada imagem de chave de uma transacção na lista de blocos corresponde ou não a um endereço oculto específico do seu anel correspondente.

Construir uma prova de não-gasto (`UnspentProof`)

O verificador de uma prova de não-gasto tem de saber rK^v, o segredo partilhado entre remetente e destinatário de uma dada saída com o endereço oculto K^o e a chave pública de transacção rG. Ele sabe a chave de ver k^v, que lhe permitiu calcular :

$$k^v * rG$$

e verificar :

$$K^o - \mathcal{H}_n(k^v * rG, t) * G \stackrel{?}{=} K^s$$

e desta forma ele sabe que a saída em questão pertence ao provador (secção 4.2).

Ou o provador forneceu rK^v. É aqui que as provas de entradas são usadas, pois com uma prova de entrada o verificador pode estar assegurado que rK^v veio da chave de ver do provador. E que corresponde com uma saída lhe pertencente sem que a chave privada de ver tenha sido fornecida.

Antes de mais, um verificador irá descobrir a imagem de chave a ser testada $\tilde{K}_?$, e verifica que o anel correspondente inclui um endereço oculto K^o, que possui uma saída pertencente ao provador. Ele depois calcula a imagem parcial de *gasto* $\tilde{K}^s_?$.

$$\tilde{K}^s_? = \tilde{K}_? - \mathcal{H}_n(rK^v, t) * \mathcal{H}_p(K^o)$$

Se a imagem de chave em questão foi criada com K^o então o ponto resultante irá ser :

$$\tilde{K}^s_? = k^s * \mathcal{H}_p(K^o).$$

[8] A possibilidade de oferecer tal assinatura directamente não existe em Monero. Mas como veremos isto está incluido na prova de reserva *ReserveProofs* (secção 8.1.6).

[9] Provas de não-gasto não estão implementadas em Monero

O provador cria duas provas de múltipla-base (secção 3.1). O seu endereço, que possui a saída em questão é (K^v, K^s) or $(K^{v,i}, K^{s,i})$. Provas de não-gasto são feitas da mesma forma para endereços normais como para sub-endereços. A chave de gasto do sub-endereço é necessária, ou seja :

$$k^{s,i} = k^s + \mathcal{H}_n(k^v, i) 4.3$$

Juntamente com as provas $\sigma_3^{unspent}$ e $\sigma_2^{unspent}$, o provador também fornece as chaves públicas $k^s * K^s$, $k^s * \tilde{K}_?^s$ e $k^s * k^s * \mathcal{H}_p(K^o)$.

Verificação

(a) Confirmar que as provas $\sigma_3^{\text{nãogasto}}$ e $\sigma_2^{\text{nãogasto}}$ são legitimas.

(b) Ter a certeza de que a mesma chave pública $k^s * K^s$ foi utilizada em ambas as provas.

(c) Verificar se $k^s * \tilde{K}_?^s$ e $k^s * k^s * \mathcal{H}_p(K^o)$ são iguais. Se são então a saída está gasta, e se não então a saída não está gasta (enp).

Funciona porque

Este meio previne que o verificador discubra $k^s * H_p(K^o)$ para uma saída não gasta, que ele poderia utilizar em combinação com rK^v para calcular a imagem de chave verdadeira. Enquanto que ele está confiante que a imagem de chave testada não corresponde a essa saída.

A prova $\sigma_2^{unspent}$ pode ser reutilizada para qualquer número de provas de não-gasto que involvam a mesma saída. Se essa saída foi de facto gasta então

8.1.6 Provar que um endereço tem um saldo mínimo não gasto (`ReserveProofV2`)

Apesar da fuga de privacidade, ao revelar a imagem de chave de uma saída quando esta ainda não foi gasta, isto ainda é um método algo útil e foi implementado em Monero [**?**] antes das provas de não-gasto terem sido inventadas [**?**]. A prova de 'reserva' é utilizada para provar que um endereço possui um montante mínimo ao criar imagens de chave para algumas saídas de transacção. Mais especificamente, dado um saldo mínimo, o provador encontra suficientes saídas não gastas que o cubra, e demonstra a posse destas com provas de entrada. Depois cria imagens de chave para estas saídas, e prova que são legítimas, para com essas saídas com provas de duas bases (com um formato de prefixo de chave diferente). Depois prova o conhecimento das chaves privadas de gasto usadas, com assinaturas normais tipo Schnorr (pode haver mais de uma se algumas saídas são possuídas por sub-endereços distintos). Um verificador verifica que as imagens de chave não apareceram na lista de blocos, e assim essas saídas têm de ser não gastas.

A prova de reserva

Todas as sub-provas dentro de uma prova de reserva *ReserveProof* assinam uma mensagem diferente do que os outros tipos de provas (e.g. OutProofs, InProofs, or SpendProofs). Desta vez é :

$$\mathfrak{m} = \mathcal{H}_n(\texttt{message}, \texttt{address}, \tilde{K}_1^o, ...\tilde{K}_n^o),$$

em que `address` é o endereço normal (K^s, K^v), de forma codificada (veja-se [**?**]). E as imagens de chave correspondem a saídas não gastas a serem incluídas na prova.

(a) Cada saída tem uma prova de entrada, o que mostra que o endereço do provador (ou um dos seus sub-endereços) possui essa saída.

(b) Cada imagem de chave de uma saída é assinada com uma prova de duas bases, em que o desafio está formatado da seguinte forma :

$$c = \mathcal{H}_n(\mathfrak{m}, [rG + c * K^o], [r\mathcal{H}_p(K^o) + c * \tilde{K}])$$

(c) Cada endereço (e sub-endereço) que possui pelo menos uma saída tem uma assinatura normal tipo Schnorr (secção 2.3.5), e o desafio (que é igual para endereços normais e sub-endereços) é :

$$c = \mathcal{H}_n(\mathfrak{m}, K^{s,i}, [rG + c * K^{s,i}]),$$

src/crypto/crypto.cpp `generate_signature()`

Para enviar uma prova de reserva a alguêm, o provador concatena o string de prefixo "`ReserveProofV2`"com duas listas codificadas em base-58 (e.g. "`ReserveProofV2`, list 1, list 2"). Cada item na lista 1 está relacionada a uma saída específica e contêm o seu hash de transacção (secção 7.4.1), o index de saída dentro dessa transacção (secção 4.2.1), o segredo partilhado relevante rK^v, a sua imagem de chave, a sua prova de entrada ($\sigma^{inproof}$), e a sua prova de imagem de chave. A lista 2 contêm items que são os endereços que possuem essas saídas juntamente com as suas assinaturas tipo Schnorr.

src/wallet/wallet2.cpp `get_reserve_proof()`

Verificação

(a) Verifica-se se as imagens de chave das provas de reserva não apareceram na lista de blocos.

(b) Verifica-se a prova de entrada para cada saída, e que um dos endereços dados possui uma saída.

(c) Verificam-se as assinaturas (de duas bases), das imagens de chave.

(d) Usam-se os segredos partilhados entre remetente e destinatário para descodificar os montantes nas saídas (secção 5.3).

(e) Verifica-se a assinatura de cada endereço.

src/wallet/wallet2.cpp `check_reserve_proof()`

Se tudo é legítimo, então o provador tem de possuir pelo menos o montante total contido nas saídas da prova de reserva (enp). [10]

[10] Uma prova de reserva, enquanto que demonstra a posse de fundos, não inclui a prova de que dados sub-endereços pertencem a um endereço normal.

8.2 Estrutura de auditoria

Nos estados unidos da america a maioria das empresas são sujeitas a auditorias financeiras [**?**]. Isto incluí a declaração de rendimentos, de cash flow e de saldo. Os primeiros dois involvem em larga parte a contabilidade interna de uma empresa enquanto que o saldo involve cada transacção, que afecta quanto dinheiro é que uma empresa actualmente possui. Cripto-moedas são como cash, portanto qualquer auditoria ao cash-flow de um utilizador de uma cripto-moeda, está relacionada com as transacções na lista de blocos.

A primeira tarefa ao auditar uma pessoa, é de identificar todas as saídas que esta possui (gastas e não gastas). Isto pode ser feito com provas de entrada usando todos os endereços em causa. Uma grande empresa pode ter uma multitude de sub-endereços, especialmente vendedores que operem em mercados online (ver capítulo 10). Criar provas de entrada para todas as transacções para cada sub-endereço pode resultar em requisitos computacionais e de espaço enormes para ambos o provador e o verificador.

Em vez disso, pode-se fazer uma prova de entrada só para o endereço normal do provador para todas as transacções. O auditor usa o segredo partilhado entre o remetente e o destinatário para verificar se alguma saída é possuída pelo endereço principal do provador ou pelos sub-endereços respectivos. Note-se que a *chave de ver* é suficiente para identificar todas as saídas possuídas pelos sub-endereços de um endereço.

Para garantir que um provador não está a tentar enganar um auditor, ao esconder um endereço normal para alguns dos seus sub-endereços, ele tem de provar também que todos os sub-endereços dados correspondem com um endereço normal.

8.2.1 Provar que um sub-endereço corresponde a um endereço

É possivel mostrar com este tipo de prova que a chave de ver de um endereço normal pode ser utilizada para identificar saídas possuídas por um dado sub-endereço [11].

Prova de sub-endereço

As provas de sub-endereço podem ser feitas á semelhança de provas de saída ou de entrada. Aqui as chaves base são :

$$\mathcal{J} = \{G, K^{s,i}\}$$

as chaves públicas são :

$$\mathcal{K} = \{K^v, K^{v,i}\}$$

e a chave que assina é k^v.

Verificação

Um verificador sabe o endereço do provador (K^v, K^s), o sub-endereço $(K^{v,i}, K^{s,i})$ e tem a prova de sub-endereço $\sigma^{subproof} = (c, r)$.

[11] Este tipo de prova ainda não foi implementado em Monero.

(a) Calcula-se o desafio

$$c' = \mathcal{H}_n(\mathfrak{m}, [K^{v,i}], [rG + c * K^v], [rK^{s,i} + c * K^{v,i}], [T_{txprf2}], [K^v], [K^{s,i}], [0])$$

(b) Se $c = c'$ então o provador sabe k^v para K^v, e $K^{s,i}$ em combinação com essa chave de ver faz $K^{v,i}$ (enp).

8.2.2 a estrutura de auditoria

Agora é possível aprender o máximo possível sobre a história de transacção de uma pessoa. [12]

(a) O provador junta uma lista de todas as suas contas, em que cada conta consiste de um endereço normal e vários sub-endereços. Ele faz uma prova de sub-endereços a todos os seus sub-endereços. Depois faz uma assinatura com a chave de gasto de cada endereço e sub-endereço, demonstrando que possui o direito de gastar de todas as saídas pertencentes a esses endereços.

(b) O provador gera, para cada um dos seus endereços normais, provas de entrada para cada transacção na lista de blocos. Isto revela ao auditor todas as saídas possuídas pelo endereço do provador. Pois é possível verificar todos os endereços ocultos com o segredo partilhado entre o remetente e o destinatário. Há a certeza que as saídas pertencentes a sub-endereços irão ser identificadas por causa das provas de sub-endereço.[13]

(c) O provador gera, para cada uma das saídas que possui, provas de não-gasto para todas as entradas de transacção em que estas saídas sejam membros de anel. Agora o auditor saberá o saldo do provador e pode passar a investigar saídas gastas. [14]

(d) *Opcional*: O provador gera, para cada transacção em que ele gastou uma saída, uma prova de saída para mostrar ao auditor o destinatário e o montante. Este passo só é possível para transacções em que o provador salvaguardou as chaves privadas de transacção.

Note-se que um provador não tem maneira de mostrar as origens de fundos directamente. O seu único recurso é requisitar um conjunto de provas de pessoas que lhe enviam dinheiro.

(a) Para uma transacção que envia dinheiro para o provador, o seu autor faz uma prova de gasto o que demonstra que o dinheiro foi de facto enviado.

(b) Quem envia dinheiro ao provador também faz uma assinatura com uma chave pública identificável, por exemplo a chave de gasto do endereço normal. A prova de gasto e esta assinatura assina uma mensagem que contêm essa chave pública

[12] Esta estrutura de auditoria não está completamente disponível em Monero. Provas de sub-endereço e provas de não-gasto não estão implementadas. Provas de entrada não estão preparadas para a optimização relacionada com sub-endereços. E não existe um verdadeira estrutura para facilmente obter e organizar todas as informações para os provadores e verificadores.

[13] Este passo também pode ser completado com as chaves privadas de ver, o que implica certas fugas de privacidade.

[14] Alternativamente, ele podia fazer provas de reserva para todas as saídas que possui. Mas revelar as imagens de chave das saídas não gastas, implica fugas de privacidade.

identificável. Assim garante-se que a prova de gasto não foi roubada ou de facto feita por outra pessoa.

CAPÍTULO 9

Multi-assinaturas

Transacções de cripto-moedas não são reversíveis. Se alguêm rouba as chaves privadas ou tem sucesso numa fraude, o dinheiro perdido pode nunca ser recuperado. Dividir o poder signatário entre pessoas pode enfraquecer o potential de algo correr mal.

Por exemplo alguem deposita dinheiro numa conta com uma empresa de segurança que vai monitorar actividade suspeita relacionada com essa conta. As transacções só podem ser assinadas se ambos os partidos cooperam. Se alguêm rouba as chaves, esta empresa de segurança pode ser alarmada, e como tal esta pára de assinar essas transacções. Isto é usualmente um serviço de garantia.

[1]

Cripto-moedas utilizam uma técnica de 'multi-assinatura' para alcançar assinaturas colaborativas chamada 'M-de-N multisig'. Em M-de-N, N pessoas cooperam para fazer uma chave junta, e só M ($M \leq N$) destas precisam de assinar com esta chave. Primeiro será introduzido o básico de 'N-de-N multisig', depois 'N-de-N multisig' em Monero, depois a geralização para 'M-de-N multisig' e por fim será esplicado como pôr chaves de multi-assinatura dentro de outras chaves de multi-assinatura.

Neste capítulo apresenta-se como as Multi-assinaturas *deviam* ser feitas, baseado nas recomendações em [**?**], e várias observações sobre implementações eficientes. Tenta-se mostrar em notas de rodapé quando a implementação actual desvia daquilo que é descrito.

[2]

[1] Multi-assinaturas têm uma diversidade de aplicações, desde contas de corporações, subscrições de jornais até aos mercados online.

[2] Actualmente existem 3 tipos de implementações de multi-assinaturas. A primeira é um processo muito básico e manual que usa a *cli* (interface da linha de comando)[**?**]. Segundo existe o excelente MMS (sistema de mensagens

9.1 Comunicação entre co-signatários

Construir chaves juntas e transacções juntas requere comunicar informação secreta entre pessoas que podem estar em qualquer parte do globo. Para manter essa informação segura de observadores, co-signatários precisam de encriptar as mensagens que são enviadas mutuamente.

Uma troca tipo diffie hellman com curvas elípticas é uma maneira muito simples de encriptar mensagens com pontos de curva elíptica. Isto já foi mencionado na secção 5.3, em que montantes de saída são comunicados ao destinatário através do segredo partilhado rK^v :

$$montante_t = b_t \oplus_8 \mathcal{H}_n(\text{“}montante\text{”}, \mathcal{H}_n(rK_B^v, t))$$

Isto pode facilmente ser extendido a qualquer mensagem. Primeiro a mensagem é codificada como uma série de bits, depois particiona-se isso em partes iguais dependendo do comprimento de $\mathcal{H}_n$. Gera-se um número aleatório $r \in \mathbb{Z}_l$ e faz-se uma troca tipo Diffie-Hellman em todas as partes usando a chave pública do destinatário K. As partes, agora encriptadas, são enviadas ao destinatário com a chave pública rG. Este desencripta a mensagem com o segredo partilhado krG.

Remetentes de mensagens deviam também criar uma assinatura na mensagem encriptada (ou só a hash da mensagem encriptada), tal que destinatários possam confirmar que as mensagens não foram alteradas (uma assinatura só é verificável com a mensagem correcta $\mathfrak{m}$).

Leitores curiosos podem olhar para este resumo conceptual : [?], ou ver uma descrição técnica do esquema de encriptação popular AES aqui : [?]. Dr. Bernstein desenvolveu um esquema de encriptação conhecido como ChaCha [?, ?], que a primeira implementação de Monero utiliza para encriptar certas informações sensitivas relacionada com as carteiras dos utilizadores (tal como as imagens de chave de saídas de transacção).

src/wallet/wallet2.cpp `export_multisig()` src/wallet/ringdb.cpp

9.2 Agregação de chaves para endereços

9.2.1 Abordagem ingénua

Seja que N pessoas querem criar um endereço de multi-assinatura, que é :

$$(K^{v,grp}, K^{s,grp}).$$

Montantes podem ser enviados a esse endereço como para cada endereço normal, mas para gastar esses fundos, todas as N pessoas tem de trabalhar juntamente para assinar transacções.

de multi-assinatura, do inglés : *Multisig Messaging System*), que é altamente automatizado através da *cli* [?, ?]. Terçeiro existe a carteira comercialmente disponível que se chama ‘Exa Wallet’, cuja código fonte inicial está em : https://github.com/exantech. Todas estas implementações baseiam-se no mesmo código fonte, da equipa núcleo de Monero, ou seja práticamente estas são todas a mesma.

Desde que todos os N participantes deviam poder ver os montantes recebidos pelo endereço de grupo, a chave de ver é dada a cada participante :

$$k^{v,grp}.$$

Para que cada participante tenha o mesmo poder, a chave de ver pode ser a soma de componentes que todos os participantes enviam uns aos outros de forma segura. Para o participante $e \in \{1, ..., N\}$, a chave de ver é a componente base :

$$k_e^{v,base} \in_R \mathbb{Z}_l.$$

Todos os participantes podem calcular a chave de ver privada do grupo :

$$k^{v,grp} = \sum_{e=1}^{N} k_e^{v,base}.$$

De forma similar, a chave de gasto do grupo :

$$K^{s,grp} = k^{s,grp}G,$$

poderia ser a soma de componentes base de chaves privadas de gasto. Contudo, se alguêm sabe todas essas componentes, então sabe a chave privada total de gasto. E como tal pode assinar transacções. Não seria uma multi-assinatura, só uma assinatura normal.

Em vez disso, obtem-se o mesmo efeito se a chave de gasto do grupo é a soma de chaves públicas de gasto. Seja que os participantes têm chaves base de gasto públicas $K_e^{s,base}$ que eles enviam uns aos outros de forma segura. Então cada um calcula :

$$K^{s,grp} = \sum_{e} K_e^{s,base}.$$

O que é o mesmo que :

$$K^{s,grp} = (\sum_{e} k_e^{s,base}) * G.$$

src/multisig/multisig.cpp `generate_multisig_view_secret_key()`

src/multisig/multisig.cpp `generate_multisig_N_N()`

9.2.2 Desvantagens da abordagem ingénua

Usar a soma de chaves públicas de gasto é intuitivo, mas leva a uma serie de questões.

Teste de agregação a uma chave

Um adversário exterior que conheça todas as chaves base públicas de gasto $K_e^{s,base}$, pode testar um dado endereço público (K^v, K^s) para a agregação de chave ao calcular :

$$K^{s,grp} = \sum_{e} K_e^{s,base},$$

e verificar se :

$$K^s \stackrel{?}{=} K^{s,grp}.$$

Isto une-se a um requisito mais genérico que chaves agregadas sejam indistingíveis de chaves normais, tal que observadores externos não possam ganhar conhecimento das actividades dos monerianos dependendo do tipo de endereço publicado. [3]

[3] Se pelo menos existe um participante honesto, que utiliza componentes seleccionados de forma aleatória seguindo uma distribuição aleatória, então as chaves agregadas por uma soma simples são indistinguíveis [**?**] de chaves normais.

Pode-se evitar isto ao criar novas chaves base de gasto para cada endereço de multi-assinatura. O que é fácil mas pode ser inconveniente. A segunda opção é mascarar chaves velhas. Procede-se da seguinte forma :
dado um participante e com um par de chaves públicas :

$$(K_e^v, K_e^s),$$

com chaves privadas :

$$(k_e^v, k_e^s)$$

e máscaras aleatórias μ_e^v, μ_e^s. Sejam os componentes de chave base privada para o endereço de grupo :

$$k_e^{v,base} = \mathcal{H}_n(k_e^v, \mu_e^v)$$
$$k_e^{s,base} = \mathcal{H}_n(k_e^s, \mu_e^s)$$

[4] Se os participantes não querem que observadores reúnam as novas chaves e as testem para agregação de chave, eles teriam de comunicar os seus novos componentes de chave uns aos outros de forma segura. [5] Se testes de agregação de chave não são uma preocupação, então os componentes de chave base pública :

$$(K_e^{v,base}, K_e^{s,base})$$

podem ser publicados como endereços normais. Qualquer entidade terceira podia então a partir desses endereços individuais, calcular e enviar montantes para o endereço de grupo. Sem interacção com os destinatários do grupo [**?**].

Cancelamento de chave

Se a chave de gasto do grupo é uma soma de chaves públicas, um participante deshonesto que descobre os componentes de chaves base de gasto, dos outros participantes, pode cancelar estes componentes.

Por exemplo, seja que a alice e o bob querem fazer um endereço de grupo. A alice bem intencionada, diz ao bob os seus componentes de chave :

$$(k_A^{v,base}, K_A^{s,base}).$$

O bob calcula os seus componentes privadamente :

$$(k_B^{v,base}, K_B^{s,base}),$$

mas não diz algo a alice. Em vez disso ele calcula :

$$K_B'^{s,base} = K_B^{s,base} - K_A^{s,base},$$

e diz a alice :

$$(k_B^{v,base}, K_B'^{s,base}).$$

[4] As máscaras aleatórias são fácilmente derivadas de um password. Por exemplo, $\mu^s = \mathcal{H}_n(password)$ e $\mu^v = \mathcal{H}_n(\mu^s)$. Ou então como é feito em Monero, as chaves de ver e de gasto são mascaradas com uma string e.g. $\mu^s, \mu^v =$ "Multisig". Isto implica que Monero só suporta uma chave de base de multi-assinatura para cada endereço. Na realidade tornar uma carteira normal para uma carteira de multi-assinatura faz com que o moneriano perca acesso á carteira normal [**?**]. É necessário criar uma nova carteira para voltar a aceder os fundos desse endereço.

[5] Como veremos na secção 9.6, a agregação de chave não funciona em multi-assinaturas em que M < N, devido á presença de segredos partilhados.

src/multisig/multisig.cpp `get_multisig_blinded_secret_key()`

O endereço de grupo é :

$$\begin{aligned} K^{v,grp} &= (k_A^{v,base} + k_B^{v,base})G \\ &= k^{v,grp}G \\ K^{s,grp} &= K_A^{s,base} + K_B'^{s,base} \\ &= K_A^{s,base} + (K_B^{s,base} - K_A^{s,base}) \\ &= K_B^{s,base}. \end{aligned}$$

Isto resulta num endereço de grupo :

$$(k^{v,grp}G, K_B^{s,base}).$$

Em que a alice sabe a chave de ver privada do grupo, e o bob sabe essa e também a chave privada de gasto !

O bob pode assinar as transacções por ele próprio, enganando a alice, que acredita que os montantes enviados para esse endereço só podem ser gastos, com a permissão dela.

Isto poderia ser resolvido ao requerer que cada participante, antes de agregar chaves, faça uma assinatura que prova que este sabe a chave privada correspondente ao componente da chave de gasto [**?**].Isto é inconveniente e vulnerável a erros de implementação. Felizmente existe uma sólida alternativa. [6]

9.2.3 Agregação de chave robusta

Para facilmente resistir ao cancelamento de chave faz-se uma pequena alteração á agregação das chaves de gasto (deixando a agregação das chaves de ver igual). Seja que o conjunto de N signatários tenham componentes de chave base de gasto :

$$\mathbb{S}^{base} = \{K_1^{s,base}, ..., K_N^{s,base}\},$$

que estão ordenados seguindo uma convenção qualquer, de mínimo para máximo, ou de forma lexicográfica.

A robusta chave de gasto agregada é :

$$K^{s,grp} = \sum_e \mathcal{H}_n(T_{agg}, \mathbb{S}^{base}, K_e^{s,base}) K_e^{s,base}$$

Se o bob tenta cancelar a chave de gasto da alice, ele fica preso com um problema díficil :

$$\begin{aligned} K^{s,grp} &= \mathcal{H}_n(T_{agg}, \mathbb{S}, K_A^s)K_A^s + \mathcal{H}_n(T_{agg}, \mathbb{S}, K_B'^s)K_B'^s \\ &= \mathcal{H}_n(T_{agg}, \mathbb{S}, K_A^s)K_A^s + \mathcal{H}_n(T_{agg}, \mathbb{S}, K_B'^s)K_B^s - \mathcal{H}_n(T_{agg}, \mathbb{S}, K_B'^s)K_A^s \\ &= [\mathcal{H}_n(T_{agg}, \mathbb{S}, K_A^s) - \mathcal{H}_n(T_{agg}, \mathbb{S}, K_B'^s)]K_A^s + \mathcal{H}_n(T_{agg}, \mathbb{S}, K_B'^s)K_B^s \end{aligned}$$

Da mesma forma que com a abordagem ingénua, qualquer terceiro que saiba $\mathbb{S}^{base}$ e as chaves públicas correspondentes, pode calcular o endereço de grupo.

Como os participantes não precisam de provar que sabem as chaves privadas de gasto, e também não precisam de interagir de todo antes de assinar as transacções, esta

[6] A primeira iteração (ainda a mesma actualmente) de multi-assinaturas, disponibilizada em abril de 2018 [**?**] (com a integração de M-de-N em outubro de 2018 [**?**]) Monero's current (and first) iteration of multisig, made available in April 2018 [**?**] (with M-of-N integration following in October 2018 [**?**]), used this naive key aggregation, and required users sign their spend key components.

src/wallet/wallet2.cpp `get_multisig_info()`

agregação de chave robusta segue o modelo *simples de chave-pública*. O único requisito neste modelo é que cada participante e potencial signatário tenha uma chave pública [**?**].
[7]

Funções `premerge` e `merge`

Mais formalmente e para ser mais claro, diz-se que existe uma operação `premerge`, que tem como argumento um conjunto de chaves base $\mathbb{S}^{base}$, e tem como resultado um conjunto de chaves agregadas $\mathbb{K}^{agg}$ de igual tamanho. Em que $\mathbb{K}^{agg}[e]$ é o n-ésimo elemento do conjunto :

$$\mathbb{K}^{agg}[e] = \mathcal{H}_n(T_{agg}, \mathbb{S}^{base}, K_e^{s,base})K_e^{s,base}$$

As chaves privadas de agregação k_e^{agg}, são usadas em assinaturas de grupo. [8] Existe uma outra operação `merge`, que usa as chaves de agregação de `premerge` e constroi a chave signatária de grupo (de gasto):

$$K^{grp} = \sum_e \mathbb{K}^{agg}[e]$$

Estas funções são generalizadas para (N-1)-de-N e M-de-N na secção 9.6.2, e também para multi-assinaturas inclusivas na secção 9.7.2.

9.3 Assinaturas tipo Schnorr com limite

É preciso um certo número de signatários para que uma multi-assinatura funcione, portanto diz-se que existe um 'limite' de signatários abaixo do qual a assinatura não pode ser produzida. Uma multi-assinatura com N participantes que requer que todos assinem, tem um limite de N, e chama-se *multi-assinatura N-de-N*.

Todos os esquemas de assinatura neste documento, baseiam-se na prova de conhecimento genérica de *Maurer* (com conhecimento nulo [**?**]). Portanto mostra-se a forma essencial de uma assinatura com limite tipo Schnorr (secção 2.3.5) [**?**].

Assinatura

Sejam N pessoas em que cada uma tem uma chave pública no conjunto $\mathbb{K}^{agg}$. Cada pessoa $e \in \{1, ..., N\}$, também sabe a chave privada k_e^{agg}.
A chave de grupo pública N-de-N que irá ser usada para assinar mensagens é K^{grp}.
Seja que todos os participantes querem assinar uma mensagem $\mathtt{m}$. Uma assinatura tipo Schnorr básica é feita com colaboração da seguinte forma :

(a) Cada participante $e \in \{1, ..., N\}$ faz o seguinte :

[7] A agregação de chave só satisfaz o modelo de chave-pública para multi-assinaturas N-de-N e 1-de-N.
[8] A agregação robusta de chave ainda não foi implementada em Monero, mas como os participantes guardam e usam a chave privada k_e^{agg} (para a agregação de chave ingénua : $k_e^{agg} = k_e^{base}$), actualizar Monero para usar a agregação robusta de chave só irá alterar o processo de *premerge*.

i. escolhe um componente aleatório $\alpha_e \in_R \mathbb{Z}_l$,

ii. calcula $\alpha_e G$

iii. compromete-se através de $C_e^\alpha = \mathcal{H}_n(T_{com}, \alpha_e G)$,

iv. e envia C_e^α aos outros participantes de forma segura.

(b) uma vez que todos os compromissos C_e^α foram coleccionados, cada participante envia o seu $\alpha_e G$ aos outros participantes de forma segura.
Cada participante verifica que :

$$C_e^\alpha \stackrel{?}{=} \mathcal{H}_n(T_{com}, \alpha_e G),$$

para cada um dos outros participantes.

(c) Cada participante calcula

$$\alpha G = \sum_e \alpha_e G$$

(d) Cada participante $e \in \{1, ..., N\}$ faz o seguinte : [9]

i. calcula o desafio $c = \mathcal{H}_n(\mathfrak{m}, [\alpha G])$,

ii. define a componente resposta $r_e = \alpha_e - c * k_e^{agg} \pmod{l}$,

iii. e envia r_e aos outros participantes de forma segura.

(e) Cada participante calcula

$$r = \sum_e r_e$$

(f) Qualquer participante pode publicar a assinatura :

$$\sigma(\mathfrak{m}) = (c, r).$$

Verificação

Dado K^{grp}, $\mathfrak{m}$, e $\sigma(\mathfrak{m}) = (c, r)$:

(a) Calcula-se o desafio $c' = \mathcal{H}_n(\mathfrak{m}, [rG + c * K^{grp}])$.

(b) Se $c = c'$, então a assinatura é legítima (enp).

O sobrescrito *grp* foi incluído para a claridade, mas na realidade o verificador não tem maneira alguma de saber que K^{grp} é uma chave junta. Só se ele sabe os componentes base ou de agregação.

Funciona porque

A resposta r é central nesta assinatura. O participante e sabe dois segredos em r_e (α_e and k_e^{agg}), assim a sua chave privada k_e^{agg} está segura em termos da teoria da informação, dos outros participantes (assumindo que ele nunca re-usa α_e). Para mais, os verificadores usam a chave de grupo pública K^{grp}, assim todos os componentes de

[9] É importante não re-utilizar α_e para diferentes desafios c. Isto significa que recomeçar um processo de multi-assinatura em que as respostas já foram enviadas, um recomeço implica um iniciar de novo com novos valores α_e.

chave são necessários para construir assinaturas.

$$\begin{aligned} rG &= (\sum_e r_e)G \\ &= (\sum_e (\alpha_e - c * k_e^{agg}))G \\ &= (\sum_e \alpha_e)G - c * (\sum_e k_e^{agg})G \\ &= \alpha G - c * K^{grp} \\ \alpha G &= rG + c * K^{grp} \\ \mathcal{H}_n(\mathfrak{m}, [\alpha G]) &= \mathcal{H}_n(\mathfrak{m}, [rG + c * K^{grp}]) \\ c &= c' \end{aligned}$$

Passo adicional de comprometer e revelar

O leitor pode estar a questionar-se de onde veio o segundo passo. Sem comprometer e revelar [**?**], um co-signatário malicioso podia aprender todos os $\alpha_e G$ *antes* do desafio ser calculado.

Isto permite-lhe controlar o desafio produzido a um certo grau, ao modificar o seu próprio $\alpha_e G$ antes de o enviar para fora.

Ele pode usar as componentes de resposta coleccionados de múltiplas assinaturas controladas para derivar outras chaves privadas k_e^{agg} em tempo sub-exponential[**?**]. O que é uma ameaça séria de segurança.

Esta ameaça baseia-se na generalização [**?**] (veja também [**?**] para uma explicação mais intuitiva) do problema do dia de anos [**?**] [10].

9.4 MLSTAG assinaturas confidenciais em anel de Monero

Transacções confidenciais em aneis com limite, adicionam alguma complexidade porque chaves signatárias MLSTAG (MLSAG com o T de *threshold* do inglés : limite, barreira) são endereços ocultos e compromissos a zero para os montantes de entrada.

Um endereço oculto que dá posse á t-ésima saída a quem tiver o endereço público (K_t^s, K_t^v) funciona da seguinte forma :

$$\begin{aligned} K_t^o &= \mathcal{H}_n(rK_t^v, t)G + K_t^s = (\mathcal{H}_n(rK_t^v, t) + k_t^s)G \\ k_t^o &= \mathcal{H}_n(rK_t^v, t) + k_t^s \end{aligned}$$

Altera-se a notação para saídas recebidas por um endereço de grupo $(K_t^{v,grp}, K_t^{s,grp})$:

$$\begin{aligned} K_t^{o,grp} &= \mathcal{H}_n(rK_t^{v,grp}, t)G + K_t^{s,grp} \\ k_t^{o,grp} &= \mathcal{H}_n(rK_t^{v,grp}, t) + k_t^{s,grp} \end{aligned}$$

[10] Comprometer e revelar não é usado na implementação actual de multi-assinaturas em Monero, no entanto está a ser considerado para um lançamento futuro [**?**].

Qualquer pessoa que tenha $k_t^{v,grp}$ e $K_t^{s,grp}$ pode descobrir $K_t^{o,grp}$, e pode descodificar o termo de Diffie-Hellman para o montante de saída e reconstruir o compromisso de máscara correspondente (secção 5.3).

Isto também significa que sub-endereços de multi-assinatura são possíveis (secção 4.3). Transacções de multi-assinatura que utilizem fundos provenientes de um sub-endereço requerem algumas modificações aos seguintes algoritmos. Sub-endereços de multi-assinatura são suportados em Monero.

9.4.1 `RCTTypeBulletproof2` com multi-assinaturas N-de-N

A maior parte de uma transacção de multi-assinatura pode ser completada por quem a iniciou. Só as assinaturas de MLSTAG requerem colaboração. Um iniciador deve fazer estes passos para preparar uma transacção `RCTTypeBulletproof2` (lembrar secção 6.1):

(a) Gera-se uma chave privada de transacção $r \in_R \mathbb{Z}_l$ (secção 4.2) e calcula-se a correspondente chave pública rG (ou múltiplas chaves destas se o destinatário for um sub-endereço; secção 4.3).

(b) Escolhem-se as entradas a serem gastas ($j \in \{1, ..., m\}$ saídas possuídas com endereços ocultos $K_j^{o,grp}$ e montantes $a_1, ..., a_m$), e os destinatários que recebem os fundos ($t \in \{0, ..., p-1\}$ novas saídas com montantes $b_0, ..., b_{p-1}$ e endereços ocultos K_t^o). Isto incluí a taxa do mineiro f e o seu compromisso fH. Escolhe-se também o conjunto de membros desvio do anel.

(c) Codifica-se cada montante de saída $montante_t$ (secção 5.3), e calcula-se os compromissos de saída C_t^b.

(d) Selecciona-se para cada entrada $j \in \{1, ..., m-1\}$, componentes máscara de pseudo compromissos de saída :

$$x'_j \in_R \mathbb{Z}_l.$$

Calcula-se a *m-ésima* máscara como na secção 5.4 :

$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$

Calcule o compromisso da pseudo saída C'^a_j.

(e) É feita a prova de domínio agregada tipo *Bulletproof* para todas as saídas. Lembrar secção 5.5.

(f) Preparação para as assinaturas MLSTAG, ao gerar para os compromissos a zero, componentes semente :

$$\alpha_j^z \in_R \mathbb{Z}_l,$$

e calcular $\alpha_j^z G$. Não há necessidade de comprometer e revelar desde que estes compromissos a zero são conhecidos por todos os signatários.

Depois o iniciador envia toda esta informação aos outros participantes de forma segura. Agora o grupo de signatários está pronto a construír assinaturas nas entradas com as suas chaves privadas $k_e^{s,agg}$, e os compromissos a zero :

$$C_{\pi,j}^a - C'^a_{\pi,j} = z_j G$$

MLSTAG RingCT

Primeiro são construidas as imagens de chave de grupo para todas as entradas $j \in \{1, ..., m\}$ com endereços ocultos $K^{o,grp}_{\pi,j}$ [11].

(a) Para cada entrada j cada participante e faz o seguinte :

i. calcula a imagem de chave parcial :
$$\tilde{K}^{o}_{j,e} = k^{s,agg}_{e} \mathcal{H}_p(K^{o,grp}_{\pi,j}),$$
ii. e envia $\tilde{K}^{o}_{j,e}$, aos outros participantes de forma segura.

(b) Cada participante, usa u_j como o index de saída na transacção em que $K^{o,grp}_{\pi,j}$ foi enviado ao endereço de multi-assinatura, e calcula : [12]
$$\tilde{K}^{o,grp}_{j} = \mathcal{H}_n(k^{v,grp} rG, u_j)\mathcal{H}_p(K^{o,grp}_{\pi,j}) + \sum_{e} \tilde{K}^{o}_{j,e}$$

Depois é construída uma assinatura MLSTAG para cada entrada j.

(a) Cada participante e faz o seguinte :

i. gera componentes semente :
$$\alpha_{j,e} \in_R \mathbb{Z}_l$$
componentes semente $\alpha_{j,e} \in_R \mathbb{Z}_l$ e calcula $\alpha_{j,e}G$, e $\alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$, src/wallet/wallet2.cpp
ii. gera para $i \in \{1, ..., v+1\}$ excepto $i = \pi$, componentes aleatórios $r_{i,j,e}$ e $r^z_{i,j,e}$, `get_multisig_kLRki()`
iii. calcula o compromisso
$$C^{\alpha}_{j,e} = \mathcal{H}_n(T_{com}, \alpha_{j,e}G, \alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j}), r_{1,j,e}, ..., r_{v+1,j,e}, r^z_{1,j,e}, ..., r^z_{v+1,j,e})$$
iv. e envia $C^{\alpha}_{j,e}$ aos outros participantes de forma segura.

(b) Depois de receber todos os $C^{\alpha}_{j,e}$ dos outros participantes, envie todos os $\alpha_{j,e}G$, $\alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$ e $r_{i,j,e}$ e $r^z_{i,j,e}$, e verifique que cada compromisso original de cada participante é válido.

(c) Cada participante calcula
$$\alpha_j G = \sum_e \alpha_{j,e} G$$
$$\alpha_j \mathcal{H}_p(K^{o,grp}_{\pi,j}) = \sum_e \alpha_{j,e} \mathcal{H}_p(K^{o,grp}_{\pi,j})$$
$$r_{i,j} = \sum_e r_{i,j,e}$$
$$r^z_{i,j} = \sum_e r^z_{i,j,e}$$

[11] Se $K^{o,grp}_{\pi,j}$ é construido de um sub-endereço de multi-assinatura indexado por i, então (secção 4.3) a chave privada é a composta :
$$k^{o,grp}_{\pi,j} = \mathcal{H}_n(k^{v,grp} r_{u_j} K^{s,grp,i}, u_j) + \sum_e k^{s,agg}_e + \mathcal{H}_n(k^{v,grp}, i)$$

[12] Se o endereço oculto corresponde a um sub-endereço de multi-assinatura indexado por i, adiciona-se : src/cryptonote_basic/cryptonote_format_utils.cpp `generate_key_image_helper_precomp()`
$$\tilde{K}^{o,grp}_{j} = ... + \mathcal{H}_n(k^{v,grp}, i)\mathcal{H}_p(K^{o,grp}_{\pi,j})$$

(d) Cada participante constroí o ciclo de assinatura (veja-se a secção 3.5).

(e) Para fechar a assinatura, cada participante e faz o seguinte:
   i. define $r_{\pi,j,e} = \alpha_{j,e} - c_\pi k_e^{s,agg} \pmod l$,
   ii. e envia $r_{\pi,j,e}$ aos outros participantes de forma segura.

(f) Qualquer participante pode calcular (lembre-se que $\alpha_{j,e}^z$ foi criado pelo iniciador) [13] src/ringct/rctSigs.cpp `signMultisig()`

$$r_{\pi,j} = \sum_e r_{\pi,j,e} - c_\pi * \mathcal{H}_n(k^{v,grp} rG, u_j)$$
$$r_{\pi,j}^z = \alpha_{j,e}^z - c_\pi z_j \pmod l$$

A assinatura para a entrada j é $\sigma_j(\mathfrak{m}) = (c_1, r_{1,j}, r_{1,j}^z, ..., r_{v+1,j}, r_{v+1,j}^z)$ com $\tilde{K}_j^{o,grp}$.
Desde que em Monero a mensagem $\mathfrak{m}$ e o desafio c_π dependem em todas as outras partes da transacção, cada participante que tenta enganar os seus co-signatários, ao enviar o valor errado, em qualquer ponto do processo inteiro, irá causar que a assinatura falhe. A resposta $r_{\pi,j}$ só é válida para a mensagem $\mathfrak{m}$ e para o desafio c_π para o qual foi definido.

9.4.2 Comunicação simplificada

Em Monero, são necessários muitos passos para construír uma transacção de multi-assinatura. É possível simplificar e reorganizar alguns deles, tal que as interações entre signatários acontecam em duas partes com um total de 5 rondas.

(a) *Agregação de chave para um endereço público de multi-assinatura*: Qualquer pessoa com um conjunto de endereços públicos pode executar `premerge`, e depois `merge` o que resulta num endereço N-de-N. Mas nenhum participante irá saber a chave de ver de grupo excepto se eles aprendam todos os componentes, portanto o grupo começa por enviar k_e^v e $K_e^{s,base}$ uns aos outros de forma segura. . Qualquer participante pode executar `premerge` e `merge` e publicar $(K^{v,grp}, K^{s,grp})$, o que permite ao grupo receber fundos através do endereço de grupo. A agregação M-de-N requer mais passos, que nós descrevemos na secção 9.6. src/wallet/wallet2.cpp `pack_multisignature_keys()`

(b) *Transacções*:
   i. Algum participante ou sub-coalição (o iniciador) decide escrever uma transacção. Eles então escolhem m entradas com endereços ocultos $K_j^{o,grp}$ e compromissos a montantes C_j^a, m conjuntos de v endereços ocultos adicionais e compromissos para serem usados como membros desvio de anel. Os participantes escolhem também p destinatários com endereços públicos (K_t^s, K_t^v) e montantes b_t para lhes enviar, decidir uma taxa de transacção f, e gerar uma chave privada de transacção r, [14] gerar máscaras de pseudo compromisso de src/wallet/wallet2.cpp `transfer_selected_rct()`

[13] Se o endereço oculto $K_{\pi,j}^{o,grp}$ corresponde a um sub-endereço de multi-assinatura indexado por i, inclui-se :

$$r_{\pi,j} = ... - c_\pi * \mathcal{H}_n(k^{v,grp}, i)$$

src/cryptonote_basic/cryptonote_format_utils.cpp `generate_key_image_helper_precomp()`

[14] Ou chaves privadas de transacção r_t ao enviar para pelo menos um sub-endereço.

saída x'_j com $j \neq m$, construír o termo ECDH $montante_t$ para cada saída, produzir uma prova de domínio agregada, e gerar aberturas de assinatura α^z_j para todos os compromissos a zero das entradas e escalares aleatórios $r_{i,j}$ e $r^z_{i,j}$ com $i \neq \pi_j$. generate pseudo output commitment masks x'_j with $j \neq m$, construct the ECDH term $amount_t$ for each output, produce an aggregate range proof, and generate signature openers α^z_j for all inputs' commitments to zero and random scalars $r_{i,j}$ and $r^z_{i,j}$ with $i \neq \pi_j$. [15] Os participantes também preparam a contribuição para a próxima ronda de comunicação.

O iniciador envia toda esta informação aos outros participantes de forma segura. [16] Os outros participantes podem sinalizar consentimento ao enviar a parte deles da próxima ronda de comunicação, ou negociar mudanças. src/wallet/wallet2.cpp

ii. Cada participante escolhe os seus componentes de abertura para as assinaturas MLSTAG, compromete-se ás mesmas, calcula as imagens de chave parciais, e envia esses compromissos e imagens parciais aos outros participantes de forma segura. `save_multisig_tx()`

Assinatura(s) MLSTAG : A imagem de chave $\tilde{K}^o_{j,e}$, a entropia de assinatura $\alpha_{j,e}G$, e $\alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$. Imagens de chave parciais não precisam de estar nos dados comprometidos, desde que esses dados não podem ser usados para extraír as chaves privadas dos signatários. As imagens de chave parciais também são úteis para observar quais saídas foram gastas, portanto para uma arquitectura modular estas devem ser tratadas individualmente.

iii. Depois de receber todos os compromissos de assinatura, cada participante envia a informação comprometida aos outros participantes de forma segura.

iv. Cada participante fecha a sua parte das assinaturas MLSTAG, e envia todos os $r_{\pi_j,j,e}$ aos outros participantes de forma segura. [17]

Assumindo que o processo correu bem, todos os participantes podem acabar de escrever a transacção e envia-la á rede. As transacções feitas por uma coalição de multi-assinatura são indistinguíveis de transacções normais, feitas por só um signatário.

[15] Note-se que o processo signatário é simplificado ao deixar que o iniciador gere escalares aleatórios $r_{i,j}$ e $r^z_{i,j}$, em vez de que cada co-signatário gere componentes que eventualmente sejam somados juntos.

[16] Ele não precisa de enviar os montantes de saída b_t directamente, pois estes podem ser calculados a partir de $amount_t$. Em Monero faz-se a abordagem razoável de criar uma transacção parcial, que contêm informação seleccionada pelo iniciador, isto envia-se para outros co-signatários juntamente com uma lista de informação relevante; como chaves privadas de transacção, endereços de destino, entradas verdadeiras, etc.

[17] É imperativo que cada tentativa de assinar por um certo signatário, use um $\alpha_{j,e}$ único, para evitar a fuga da chave privada a outros signatários (secção 2.3.4) [**?**]. As carteiras deviam por princípio forçar isto ao apagarem sempre $\alpha_{j,e}$, cada vez que uma resposta que use isso seja enviada para fora da carteira.

9.5 Recalcular imagens de chaves

Se alguêm perde os seus dados e quer calcular o saldo de um dos seus endereços (montantes recebidos menos montantes gastos), então precisa de sacar dados da lista de blocos. As chaves de ver só são úteis para ler quais os montantes recebidos, assim um moneriano precisa de calcular imagens de chave para todas as saídas que possui, para ver se essas foram gastas, ao comparar com imagens de chave guardadas na lista de blocos. Desde que os membros de um endereço de grupo nao podem calcular as imagens de chave individualmente, estes requerem a ajuda do outros membros.

Calcular imagens de chave de uma simples soma de componentes pode falhar se participantes deshonestos reportam chaves falsas. [18] Dada uma saída recebida com o endereço oculto $K^{o,grp}$, o grupo pode produzir uma simples prova tipo Schnorr, para que a imagem de chave $\tilde{K}^{o,grp}$ é legitima sem que componentes de chaves privadas sejam revelados, nem a necessidade de confiarem uns nos outros.

Prova

(a) Cada participante e faz o seguinte :
   i. calcula $\tilde{K}_e^o = k_e^{s,agg}\mathcal{H}_p(K^{o,grp})$,
   ii. gera a componente semente$\alpha_e \in_R \mathbb{Z}_l$ e calcula $\alpha_e G$ e $\alpha_e\mathcal{H}_p(K^{o,grp})$,
   iii. compromete-se aos dados com : $C_e^\alpha = \mathcal{H}_n(T_{com}, \alpha_e G, \alpha_e\mathcal{H}_p(K^{o,grp}))$,
   iv. e envia C_e^α e $\tilde{K}_e^o$ aos outros participantes de forma segura.

(b) Depois de receber todos os C_e^α, cada participante envia a informação comprometida e verifica os compromissos dos outros participantes.

(c) Cada participante pode calcular : [19]

$$\tilde{K}^{o,grp} = \mathcal{H}_n(k^{v,grp}rG, u)\mathcal{H}_p(K^{o,grp}) + \sum_e \tilde{K}_e^o$$

$$\alpha G = \sum_e \alpha_e G$$

$$\alpha\mathcal{H}_p(K^{o,grp}) = \sum_e \alpha_e\mathcal{H}_p(K^{o,grp})$$

(d) Cada participante calcula o desafio [20]

$$c = \mathcal{H}_n([\alpha G], [\alpha\mathcal{H}_p(K^{o,grp})])$$

(e) Cada participante faz o seguinte :
   i. define-se $r_e = \alpha_e - c * k_e^{s,agg} \pmod l$,
   ii. e envia-se r_e aos outros participantes de forma segura.

[18] Actualmente em Monero, usa-se uma soma simples .

[19] Se o endereço oculto corresponde a um sub-endereço de multi-assinatura indexado por i, adicione-se :

$$\tilde{K}^{o,grp} = ... + \mathcal{H}_n(k^{v,grp}, i)\mathcal{H}_p(K^{o,grp})$$

src/wallet/wallet2.cpp `export_multisig()`

[20] Esta prova devia incluir separação de domínio e prefixo de chave, o que é omitido por brevidade.

(f) Cada participante pode calcular : [21]

$$r^{resp} = \sum_e r_e - c * \mathcal{H}_n(k^{v,grp} rG, u)$$

A prova é (c, r^{resp}) com $\tilde{K}^{o,grp}$.

Verificação

(a) Verifica $l\tilde{K}^{o,grp} \stackrel{?}{=} 0$.

(b) Calcula
$c' = \mathcal{H}_n([r^{resp}G + c * K^{o,grp}], [r^{resp}\mathcal{H}_p(K^{o,grp}) + c * \tilde{K}^{o,grp}])$.

(c) Se $c = c'$ então a imagem chave $\tilde{K}^{o,grp}$ corresponde ao endereço oculto $K^{o,grp}$ (enp).

9.6 M < N

No início deste capítulo foram discutidos servicos de garantia, que usam uma multi-assinatura 2-de-2, para separar o poder signatário entre um moneriano e uma empresa de segurança. Esse sistema não é ideal, pois se a empresa de segurança for comprometida, ou se recusa cooperar, então os montantes podem ficar bloqueados.

Isto tem solução, usa-se um endereço de multi-assinatura 2-de-3, em que há 3 participantes mas só são necessários dois signatários para efectuar uma transacção. Um serviço de garantia guarda uma chave e os outros participantes guardam as outras.
Os participantes podem guardar uma chave num lugar seguro, e usar a outra chave para o uso quotidiano. Se o serviço de garantia falha, o participante pode usar a chave do quotidiano e a chave segura para tirar os montantes do endereço de multi-assinatura.
Multi-assinaturas com limites abaixo de N, têm vários usos.

9.6.1 Agregação de chave 1-de-N

Seja que um grupo de pessoas querem fazer uma chave de multi-assinatura K^{grp}, com a qual todos podem assinar. A solução é trivial: deixa-se que cada participante saiba a chave privada k^{grp}. Existem 3 formas de fazer isto.

(a) Um participante ou uma sub-coalição selecciona uma chave e envia esta a todos os outros participantes de forma segura.

(b) Todos os participantes calculam componentes de chaves privadas, e enviam estas de forma segura a todos os outros participantes. Em que a soma é uma chave junta. [22]

[21] Se o endereço oculto $K^{o,grp}$ corresponde a um sub-endereço de multi-assinatura indexado por i, inclui-se :

$$r^{resp} = ... - c * \mathcal{H}_n(k^{v,grp}, i)$$

[22] Note-se que aqui o cancelamento de chave, não é muito relevante visto que cada participante conhece a chave privada inteira.

(c) Os participantes extendem multi-assinaturas de M-de-N a 1-de-N. Isto pode ser útil se um adversário tem acesso ás comunicações de grupo.

Neste caso, para Monero, cada participante saberia as chaves privadas :

$$(k^{v,grp,\text{1xN}}, k^{s,grp,\text{1xN}}).$$

Antes desta secção todas as chaves de grupo eram N-de-N, mas agora usa-se o sobrescrito 1xN para denotar chaves relacionadas com multi-assinaturas 1-de-N.

9.6.2 Agregação de chave (N-1)-de-N

Numa chave de grupo (N-1)-de-N, como 2-de-3 ou 4-de-5, cada conjunto de (N-1) participantes pode efectuar uma transacção. Isto é conseguido com segredos partilhados do tipo Diffie-Hellman. Seja que existem participantes $e \in \{1, ..., N\}$, cada um com uma chave base pública K_e^{base}.

Cada participante e calcula para $w \in \{1, ..., N\}$ excepto $w \neq e$.

$$k_{e,w}^{sh,\text{(N-1)xN}} = \mathcal{H}_n(k_e^{base} K_w^{base})$$

Depois cada participante calcula todos os :

$$K_{e,w}^{sh,\text{(N-1)xN}} = k_{e,w}^{sh,\text{(N-1)xN}} G,$$

e envia isso aos outros participantes de forma segura. Agora utiliza-se o sobrescrito sh para denotar chaves partilhadas por um sub-grupo dos participantes.

src/multisig/multisig.cpp `generate_multisig_N1_N()`

Cada participante irá ter (N-1) componentes de chaves privadas partilhadas, correspondentes a cada um dos outros participantes. O que perfaz para todos os participantes, um total de N*(N-1) componentes de chaves privadas partilhadas.

Cada chave é partilhada por dois parceiros que executam a troca Diffie-Hellman, portanto só existem [N*(N-1)]/2 chaves únicas. Estas chaves únicas compõem o conjunto :

$$\mathbb{S}^{\text{(N-1)xN}}.$$

Generalizar `premerge` and `merge`

É aqui que renovamos a definição de `premerge` da secção 9.2.3. O seu argumento irá ser o conjunto $\mathbb{S}^{\text{MxN}}$, em que M é o limite do conjunto de chaves. Quando $M = N$:

$$\mathbb{S}^{\text{NxN}} = \mathbb{S}^{base},$$

e quando $M < N$ it contains shared keys. O resultado é :

$$\mathbb{K}^{agg,\text{MxN}}.$$

Os [N*(N-1)]/2 componentes de chave em $\mathbb{K}^{agg,\text{(N-1)xN}}$ podem ser enviados para `merge`, resultando em $K^{grp,\text{(N-1)xN}}$. Todos os componentes de chave privada podem ser construídos com só (N-1) participantes, desde que cada participante partilha um segredo partilhado tipo Diffie-Hellman com o n-ésimo participante.

Um exemplo 2-de-3

Seja que existem 3 pessoas com chaves públicas $\{K_1^{base}, K_2^{base}, K_3^{base}\}$, para as quais cada uma sabe uma chave privada, e estas querem fazer uma chave de multi-assinatura

2-de-3. Depois da troca tipo Diffie-Hellman, e de enviarem uns aos outros as chaves públicas, cada um dos participantes sabe o seguinte :

(a) Participante 1: $k_{1,2}^{sh,2x3}$, $k_{1,3}^{sh,2x3}$, $K_{2,3}^{sh,2x3}$

(b) Participante 2: $k_{2,1}^{sh,2x3}$, $k_{2,3}^{sh,2x3}$, $K_{1,3}^{sh,2x3}$

(c) Participante 3: $k_{3,1}^{sh,2x3}$, $k_{3,2}^{sh,2x3}$, $K_{1,2}^{sh,2x3}$

Em que $k_{1,2}^{sh,2x3} = k_{2,1}^{sh,2x3}$, e assim por diante. O conjunto $\mathbb{S}^{2x3} = \{K_{1,2}^{sh,2x3}, K_{1,3}^{sh,2x3}, K_{2,3}^{sh,2x3}\}$. Executar `premerge` e `merge` gera a chave de grupo : [23]

$$\begin{aligned} K^{grp,2x3} =& \mathcal{H}_n(T_{agg}, \mathbb{S}^{2x3}, K_{1,2}^{sh,2x3}) K_{1,2}^{sh,2x3} + \\ & \mathcal{H}_n(T_{agg}, \mathbb{S}^{2x3}, K_{1,3}^{sh,2x3}) K_{1,3}^{sh,2x3} + \\ & \mathcal{H}_n(T_{agg}, \mathbb{S}^{2x3}, K_{2,3}^{sh,2x3}) K_{2,3}^{sh,2x3} \end{aligned}$$

Seja que o participante 1 e 2 querem assinar a mensagem $\mathfrak{m}$. Iremos usar uma assinatura básica tipo Schnorr para demonstrar.

(a) Cada participante $e \in \{1, 2\}$ faz o seguinte :

  i. escolhe um componente aleatório $\alpha_e \in_R \mathbb{Z}_l$,

  ii. calcula $\alpha_e G$,

  iii. compromete-se com $C_e^{\alpha} = \mathcal{H}_n(T_{com}, \alpha_e G)$,

  iv. e envia C_e^{α} aos outros participantes de forma segura.

(b) Depois de receber C_e^{α}, cada participante envia $\alpha_e G$ e verifica os outros compromissos.

(c) cada participante calcula

$$\alpha G = \sum_e \alpha_e G$$
$$c = \mathcal{H}_n(\mathfrak{m}, [\alpha G])$$

(d) Participante 1 faz o seguinte :

  i. calcula $r_1 = \alpha_1 - c * [k_{1,3}^{agg,2x3} + k_{1,2}^{agg,2x3}]$,

  ii. e envia r_1 ao participante 2 de forma segura.

(e) Participante 2 faz o seguinte :

  i. calcula $r_2 = \alpha_2 - c * k_{2,3}^{agg,2x3}$,

  ii. e envia r_2 ao participante 1 de forma segura.

(f) Cada participante calcula

$$r = \sum_e r_e$$

(g) Cada participante pode publicar a assinatura $\sigma(\mathfrak{m}) = (c, r)$.

A única mudança com multi-assinaturas de limite sub-N, é como 'fechar o círculo' ao definir $r_{\pi,e}$ (no caso de assinaturas em anel, com index secreto π). Cada participante tem de incluir o segredo partilhado correspondente á 'pessoa desaparecida', mas desde que todos os outros segredos partilhados são duplicados, existe um truque.

[23] Desde que a chave junta é composta de segredos partilhados, um observador que saiba só as chaves base originais não seria capaz de as agregar (secção 9.2.2), e de identificar os membros da chave junta.

Dado o conjunto $\mathbb{S}^{base}$ de chaves originais de todos os participantes, só a *primeira pessoa* - ordenada por index em $\mathbb{S}^{base}$ - com uma cópia de um segredo partilhado uses it to calculate his $r_{\pi,e}$. [24] [25]

No exemplo prévio, o participante 1 calcula

$$r_1 = \alpha_1 - c * [k_{1,3}^{agg,2x3} + k_{1,2}^{agg,2x3}]$$

enquanto o participante 2 só calcula :

$$r_2 = \alpha_2 - c * k_{2,3}^{agg,2x3}$$

Em Monero o mesmo princípio aplica-se a calcular a imagem de chave de grupo em transacções de multi-assinatura com limites sub-N.

9.6.3 Agregação de chave M-de-N

Em (N-1)-de-N cada segredo partilhado entre duas chaves públicas, tal como K_1^{base} e K_2^{base} contêm duas chaves privadas $k_1^{base}k_2^{base}G$.
É um segredo porque só o participante 1 pode calcular $k_1^{base}K_2^{base}$, e só o participante 2 pode calcular $k_2^{base}K_1^{base}$. E se existe uma terceira pessoa com K_3^{base}, existem segredos partilhados

$$k_1^{base}k_2^{base}G, k_1^{base}k_3^{base}G, k_2^{base}k_3^{base}G.$$

E se os participantes enviam essas chaves públicas uns aos outros? Cada um contribui uma chave privada a duas chaves públicas. Agora diga-se que os participantes fazem um novo segredo partilhado com essa terceira chave pública.

O participante 1 calcula o segredo partilhado :

$$k_1^{base} * (k_2^{base}k_3^{base}G).$$

O participante 2 calcula o segredo partilhado :

$$k_2^{base} * (k_1^{base}k_3^{base}G).$$

O participante 3 calcula o segredo partilhado :

$$k_3^{base} * (k_1^{base}k_2^{base}G).$$

Agora cada um dos participantes sabe :

$$k_1^{base}k_2^{base}k_3^{base}G.$$

O que é um segredo partilhado entre os 3 participantes (desde que nenhum deles publique o segredo).

src/multisig/multisig.cpp `generate_multisig_derivations()`

[24] Na prática isto significa que um iniciador devia determinar qual o sub-conjunto de M signatários irá assinar uma dada mensagem. Se ele descobre que nenhum signatário quer assinar, ou seja (O ≥ M), ele pode orchestrar múltiplas tentativas simultâneas de assinar para cada sub-conjunto de tamanho M dentro de O para aumentar a probabilidade que uma funcione. Parece que Monero utiliza esta abordagem. Também acontece que (um ponto esotérico) a lista *original* de saídas destino, criadas pelo iniciador são misturadas de forma aleatória, e essa lista é então usada por cada tentativa de assinar, e todos os outros co-signatários (isto está relacionado com o `flag` obscuro `shuffle_outs`).

[25] Actualmente Monero usa um método signatário de *round-robin* , em que o iniciador assina com todas as suas chaves privadas, esta transacção parcialmente assinada é então enviada a outro signatário que assina com todas as *suas* chaves privadas (que ainda não foram utilizadas para assinar), que por sua vez envia esta transacção parcialmente assinada ao próximo signatário, e por assim diante, até ao signatário final. Este então pode enviar a transacção agora totalmente assinada á rede, ou a outro signatário para que este a envie á rede.

src/wallet/wallet2.cpp `sign_multisig_tx()`

Este grupo podia usar :

$$k^{sh,1x3} = \mathcal{H}_n(k_1^{base} k_2^{base} k_3^{base} G),$$

como uma chave privada partilhada. Publica-se :

$$K^{grp,1x3} = \mathcal{H}_n(T_{agg}, \mathbb{S}^{1x3}, K^{sh,1x3}) K^{sh,1x3},$$

como um endereço de multi-assinatura 1-de-3.

Numa multi-assinatura 3-de-3 cada participante tem um segredo.
Numa multi-assinatura 2-de-3 cada sub-conjunto de 2 participantes partilha um segredo.

Para generalizar para M-de-N : cada possivel sub-conjunto de (N-M+1) participantes, partilha um segredo [**?**]. Se (N-M) pessoas

algoritmo M-de-N

Sejam participantes $e \in \{1, ..., N\}$ com chaves privadas iniciais $k_1^{base}, ..., k_N^{base}$, que querem produzir uma chave M-de-N junta (M $\leq$ N; M $\geq$ 1 and N $\geq$ 2). Utiliza-se então um algoritmo interactivo.

src/wallet/wallet2.cpp `exchange_multisig_keys()`

Utiliza-se $\mathbb{S}_s$ para denotar todas as chaves públicas *únicas* na iteração $s \in \{0, ..., (N - M)\}$.
O conjunto final $\mathbb{S}_{N-M}$ está ordenado convencionalmente com números ascendentes ou lexicográficamente). Esta notação é uma conveniência, e $\mathbb{S}_s$ é o mesmo que $\mathbb{S}^{(N-s)xN}$ das secções anteriores.

O conjunto de chaves públicas que cada participante gera na iteração s do algoritmo é denotado por :

$$\mathbb{S}_{s,e}^K.$$

No início :

$$\mathbb{S}_{0,e}^K = \{K_e^{base}\}.$$

O conjunto :

$$\mathbb{S}_e^k.$$

Irá no fim, conter as chaves privadas de agregação de cada participante.

(a) Cada participante e envia o conjunto original de chaves públicas :

$$\mathbb{S}_{0,e}^K = \{K_e^{base}\},$$

aos outros participantes de forma segura.

(b) cada participante constroí $\mathbb{S}_0$ ao coleccionar todos os $\mathbb{S}_{0,e}^K$ e remover duplicados.

(c) Na iteração $s \in \{1, ..., (N - M)\}$ (excepto M = N)

i. Cada participante e faz o seguinte :

A. Para cada elemento g_{s-1} de
$\mathbb{S}_{s-1} \notin \mathbb{S}_{s-1,e}^K$,
calcula-se um novo segredo partilhado :

$$k_e^{base} * \mathbb{S}_{s-1}[g_{s-1}]$$

B. Todos os novos segredos partilhados são postos em :

$$\mathbb{S}_{s,e}^K.$$

C. Se $s = (N - M)$,
calcula-se a chave privada partilhada para cada elemento :
$$xem \mathbb{S}^K_{N-M,e}.\mathbb{S}^k_e[x] = \mathcal{H}_n(\mathbb{S}^K_{N-M,e}[x])$$
e sobre escrever a chave pública ao pôr :
$$\mathbb{S}^K_{N-M,e}[x] = \mathbb{S}^k_e[x] * G.$$
D. Envia-se aos outros participantes :
$$\mathbb{S}^K_{s,e}.$$

ii. Cada participante constroi $\mathbb{S}_s$ ao coleccionar todos os $\mathbb{S}^K_{s,e}$, removendo duplicados. [26]

(d) Cada participante ordena $\mathbb{S}_{N-M}$ segundo a convenção.

(e) a função `premerge` tem como argumento $\mathbb{S}_{(N-M)}$, e cada chave de agregação é para $g \in \{1, ..., (\text{size of } \mathbb{S}_{N-M})\}$:
$$\mathbb{K}^{agg,\text{MxN}}[g] = \mathcal{H}_n(T_{agg}, \mathbb{S}_{(N-M)}, \mathbb{S}_{(N-M)}[g]) * \mathbb{S}_{(N-M)}[g]$$

(f) A função `merge` tem como argumento $\mathbb{K}^{agg,\text{MxN}}$, e a chave de grupo é :
$$K^{grp,\text{MxN}} = \sum_{g=1}^{\text{size of } \mathbb{S}_{N-M}} \mathbb{K}^{agg,\text{MxN}}[g]$$

(g) Cada participante e substitui cada elemento x em $\mathbb{S}^k_e$ com a sua chave privada de agregação :
$$\mathbb{S}^k_e[x] = \mathcal{H}_n(T_{agg}, \mathbb{S}_{(N-M)}, \mathbb{S}^k_e[x]G) * \mathbb{S}^k_e[x]$$

9.7 Famílias de chaves

Até aqui foram consideradas agregações de chave entre um simples grupo de signatários. Por exemplo, a alice o bob e a carol cada um contribui componentes de chave a um endereço de multi-assinatura 2-de-3.

Agora imagine-se que a alice quer assinar *todas* as transacções desse endereço, e não quer que o bob e a carol assinem sem ela. Por outras palavras, (alice + bob) ou (alice + carol) são pares aceitáveis, mas não (bob + carol).

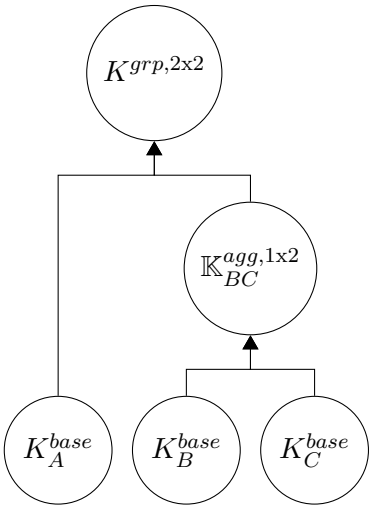
Isto pode ser conseguido com duas camadas de agregação de chave. Primeiro uma agregação de multi-assinatura 1-de-2 $\mathbb{K}^{agg,1\text{x}2}_{BC}$ entre o bob e a carol. Depois uma chave de grupo 2-de-2 $K^{grp,2\text{x}2}$ entre a alice e $\mathbb{K}^{agg,1\text{x}2}_{BC}$. Básicamente um endereço de multi-assinatura (2-de-([1-de-1] e [1-de-2])).

Isto implica que estruturas de acesso a direitos signatários podem ficar inacabados.

9.7.1 Árvores de família

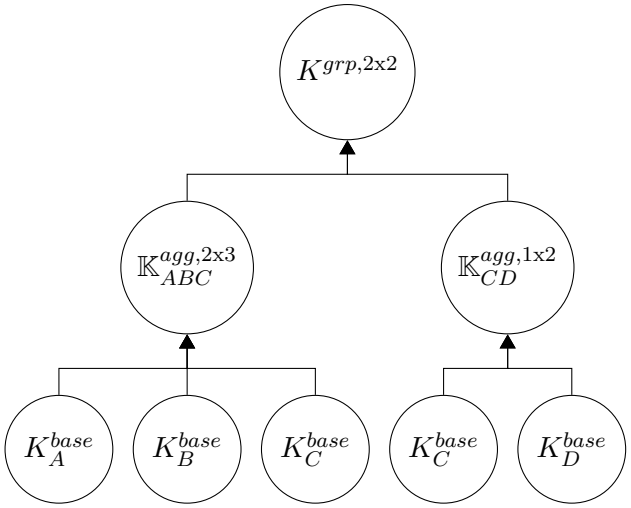
O endereço de multi-assinatura (2-de-([1-de-1] e [1-de-2])) pode ser representado no seguinte diagrama :

[26] Os participantes deviam saber quais participantes têm quais chaves no último passo ($s = N - M$), para facilitar as assinaturas colaborativas, em que só a primeira pessoa em $\mathbb{S}_0$ com uma certa chave privada é que assina. Veja-se secção 9.6.2.



As chaves $K^{base}_{A}, K^{base}_{B}, K^{base}_{C}$ são consideradas *raízes ancestrais*, enquanto que $\mathbb{K}^{agg,1x2}_{BC}$ é *filho* dos *pais* K^{base}_{B} e K^{base}_{C}. Pais podem ter mais de um filho, mas para a clareza conceptual nós consideramos cada cópia de um pai como distinta. Isto significa que podem haver múltiplas raízes ancestrais que são a mesma chave.

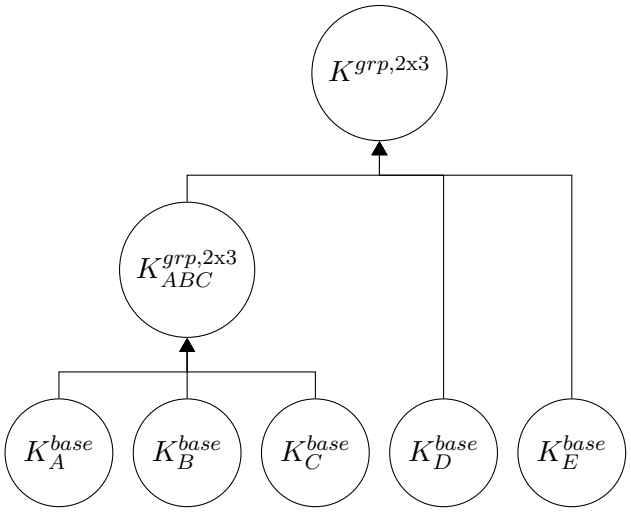
Por exemplo, neste 2-de-3 ou 1-de-2 junto a um 2-de-2, a chave da carol K^{base}_{C} é usada duas vezes e representada duas vezes :



Conjuntos distintos $\mathbb{S}$ são definidos para cada sub-coalição de multi-assinatura. Existem trés conjuntos de premerge no exemplo anterior :

9.7.2 Chaves de multi-assinatura inclusivas

Seja a seguinte família de chaves



Se juntar-mos as chaves em $\mathbb{S}^{2x3}_{ABC}$ correspondentes aos primeiros 2-de-3, irá acontecer uma situação no próximo nível. Escolha-se só um segredo partilhado, entre $K^{grp,2x3}_{ABC}$ e K^{base}_{D} :

$$k_{ABC,D} = \mathcal{H}_n(k^{grp,2x3}_{ABC} K^{base}_{D})$$

Agora, duas pessoas de ABC podem fácilmente contribuir componentes de agregação de chave, tal que a sub-coalição possa calcular :

$$k^{grp,2x3}_{ABC} K^{base}_{D} = \sum k^{agg,2x3}_{ABC} K^{base}_{D}$$

O problema é que qualquer participante de ABC pode calcular :

$$k^{sh,2x3}_{ABC,D} = \mathcal{H}_n(k^{grp,2x3}_{ABC} K^{base}_{D})!$$

Se todos os participantes de uma camada de multi-assinatura mais baixa, sabem todas as chaves privadas de uma camada de multi-assinatura mais elevada, então o esquema é escusado e é como se a camada de baixo fosse 1-de-N.

Isto é resolvido ao não juntar completamente as chaves até á chave final.
Em vez disso faz-se `premerge` para todas as chaves das camadas mais baixas.

Solução para a inclusão

Para usar $\mathbb{K}^{agg,MxN}$ numa nova multi-assinatura, esta é passada entre os participantes, com uma alteração. Operações que involvam $\mathbb{K}^{agg,MxN}$ usam cada uma das suas chaves membro, em vez de uma chave junta de grupo. Por exemplo, a 'chave' pública de um segredo partilhado entre $\mathbb{K}^{agg,2x3}_{x}$ e K^{base}_{A} iria ser :

$$\mathbb{K}^{sh,1x2}_{x,A} = \{[\mathcal{H}_n(k^{base}_{A} \mathbb{K}^{agg,2x3}_{x}[1]) * G], [\mathcal{H}_n(k^{base}_{A} \mathbb{K}^{agg,2x3}_{x}[2]) * G], ...\}$$

Desta forma todos os membros de :

$$\mathbb{K}^{agg,2x3}_{x},$$

só sabem os segredos partilhados correspondentes ás suas chaves privadas da camada abaixo de multi-assinatura 2-de-3. Uma operação entre um conjunto de chaves de tamanho 2 ${}^{2}\mathbb{K}_{A}$ e um conjunto de chaves de tamanho 3 ${}^{3}\mathbb{K}_{B}$ iria produzir um conjunto de chaves de tamanho 6 ${}^{6}\mathbb{K}_{AB}$.

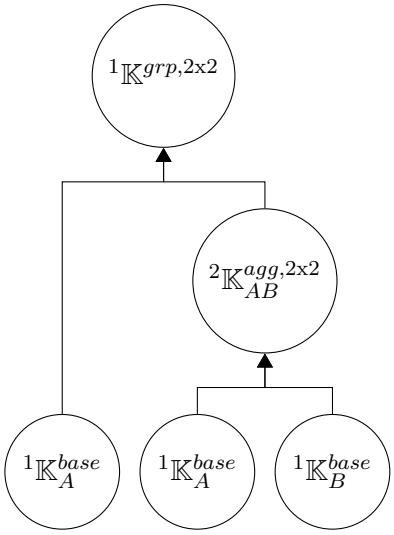
Podemos generalizar todas as chaves numa família de chaves como conjuntos de chaves, em que chaves individuais são denotadas ${}^{1}\mathbb{K}$.
Elementos num conjunto de chaves são ordenados segundo uma convenção (por exemplo lexicográfica), e conjuntos que contenham conjuntos de chave (e.g. conjuntos $\mathbb{S}$), são ordenados pelo primeiro elemento em cada conjunto de chave. Deixa-se que os conjuntos de chave se propaguem pela estrutura de família, em que cada grupo de multi-assinatura inclusiva envia o seu conjunto de agregação `premerge` como a 'chave base' para a próxima camada. [27] Até que aparece o conjunto de agregação do último filho, em que finalmente é utilizado `merge`. Monerianos devem guardar as suas chaves privadas base, as chaves privadas de agregação para todas as camadas de uma estrutura de família de multi-assinatura, e as chaves públicas para todas as camadas. Ao fazer isto é mais fácil criar novas estruturas, fazer `merge` de multi-assinaturas inclusivas, e colaborar com outros signatários para reconstruír uma estrutura de família caso haja um problema.

9.7.3 Implicações

Cada sub-coalição que contribui para a chave final precisa de contribuir componentes para uma transacção de Monero (tal como os valores iniciais αG), e como tal cada sub-sub-coalição precisa de contribuir para a sub-coalição filho.

Isto significa que cada raíz ancestral, mesmo quando existem múltiplas cópias da mesma chave na estrutura da família, tem de contribuir uma componente raíz ao filho do próximo nível. E cada filho por sua vez tem de contribuir uma componente ao seu filho. E isto para cada camada. São usadas somas simples a cada nível.

Por exemplo, seja esta família :



Seja que é necessário calcular um valor de grupo x para uma assinatura. Raízes ances-

[27] Note-se que `premerge` precisa de ser feito ás saídas de *todas* as multi-assinaturas inclusivas, mesmo quando uma multi-assinatura N-de-N está incluída em outra N-de-N, porque o conjunto $\mathbb{S}$ muda.

trais contribuem :

$$x_{A,1}, x_{A,2}, x_B.$$

O total é :

$$x = x_{A,1} + x_{A,2} + x_B.$$

Não existem actualmente implementações de multi-assinaturas inclusivas em Monero.

CAPÍTULO 10

Mercados garantidos por terceiros

Na maioria das vezes, as compras online acontecem sem problemas. O comprador envia dinheiro ao vendedor, e o produto esperado aparece em casa. Se o produto tem um defeito, ou se o comprador muda de ideias, o produto pode ser devolvido, e o comprador é reembolsado. É difícil confiar numa pessoa ou organização descohnecida, e muitos compradores podem-se sentir seguros sabendo que as companhias de cartões de crédito irão possivelmente reverter as transacções [**?**].

As transacções de cripto-moedas não são reversíveis, e existe um recurso legal limitado para o comprador ou vendedor quando algo corre mal. Especialmente com Monero, porque as transacções não são analisadas facilmente por entidades terceiras [**?**].

Um aspecto central a fazer compras online são mercados garantidos por terceiros, em que são usadas multi-assinaturas 2-de-3. Assim entidades terceiras são capazes de mediar disputas. Ao confiar nessas entidades, vendedores e compradores anónimos conseguem interagir sem formalidades.

Em Monero as interacções com multi-assinaturas são complexas. Neste capítulo é descrito um mercado online efficiente garantido por terceiros [1,2].

[1] René Brunner "rbrunner7", um contribuidor de Monero que crio o MMS [**?**, **?**], investigou integrar as multi-assinaturas em Monero no mercado digital baseado em cripto-moedas chamado *OpenBazaar* https://openbazaar.org/. Os conceitos aí apresentados são inspirados pelas dificuldades que René encontrou [**?**] (his 'earlier analysis').

[2] A implementação actual de multi-assinaturas em Monero, já tem a sequência de passos na criação de transacção, desejada para mercados online garantidos por terçeiros. O que é bom, mas note-se que multi-assinaturas precisam de actualizações de segurança, antes de serem utilizadas em grande escala. [**?**].

10.1 Características essenciais

Existem vários requisitos básicos para as interacções entre vendedores e compradores em mercados online. Estes pontos são da investigação de René acerca de OpenBazaar [**?**].

- *venda offline*: Um comprador devia poder aceder á loja online, enquanto o vendedor está offline, e fazer uma compra. É claro que o vendedor tem de ir online para receber a compra e finalizar a venda. Ou seja enviar o produto ao comprador.
- *Compras com pagamentos ordenados*: Os endereços do vendedor são únicos para cada venda, de forma a ligar cada venda com o seu pagamento.
- *Compras de alta confiança*: Um comprador pode, se confia no vendedor, pagar por um produto mesmo antes deste estar disponível ou ser ser entregue.
  - *pagamento online directo*: Depois de confirmar que o vendedor está online, e que o produto listado está disponível, o comprador envia o dinheiro numa só transacção a um endereço do vendedor, e este depois signaliza que a compra está a ser efectuada.
  - *Pagamento offline*: Se o vendedor está offline, o comprador cria e envia um montante para um endereço multi-assinatura 1-de-2, com dinheiro suficiente para finalizar a compra. Quando o vendedor volta a estar online, ele pode tirar o dinheiro do endereço de multi-assinatura, e executar a venda. Se o vendedor nunca mais volta a estar online (ou depois de um certo tempo de espera), ou se o comprador muda de ideias antes dele voltar a estar online, o comprador pode tirar o dinheiro desse endereço de multi-assinatura de volta para a sua carteira pessoal.
- *Compra através de um moderador*: Um endereço de multi-assinatura 2-de-3 é construido entre o comprador o vendedor e o moderador. O moderador é aceite pelo comprador e pelo vendedor. O comprador envia dinheiro para este endereço antes do vendedor enviar o produto. Uma vez que o produto tenha sido entregue ao comprador, os dois partidos cooperam para que os fundos sejam libertados. Se o comprador e o vendedor não chegam a um acordo, estes podem delegar o julgamento ao moderador.

10.1.1 Sequência de passos na compra

Todas as compras deviam caber dentro do mesmo conjunto de passos, assumindo que todos os partidos agem com a diligência devida. Um moderador antes de servir dois partidos, espera um número de passos já executados, por exemplo : "antes de involverem um moderador, um dos partidos já pediu um reembolso ?"

10.2 Multi-assinatura Monero

Uma interacção de multi-assinatura 2-de-3 pode ser feita quase completamente ao longo da sequência de passos na compra sem que os participantes notem. Existe um passo extra a fazer pelo vendedor no fim. Em que ele tem de assinar e submeter a transacção final para receber o pagamento. O que é comparável a retirar o dinheiro todo da caixa registadora. [3]

10.2.1 Os básicos de uma interacção de multi-assinatura

Todas as interacções de multi-assinatura 2-de-3 contêm o mesmo conjunto de rondas de comunicação, o que involve a construção do endereço e depois da construção da transacção. De forma a cumprir com a sequência de passos usa-se um outro processo de construção do que aquele dado no capítulo de multi-assinatura 9.4.2 e 9.6.2.
[4]

(a) *Construír o endereço*

i. Todos os participantes têm de primeiro conhecer as chaves base dos outros participantes, o que irá ser usado para gerar segredos partilhados.
As chaves públicas desses segredos partilhados :
$$K^{sh} = \mathcal{H}_n(k_A^{base} * k_B^{base} G)G.$$
são enviados aos outros participantes.

ii. Depois de um participante ter aprendido todas as chaves públicas de segredo partilhado, ele pode executar a função `premerge` e depois `merge`, o que gera a chave de gasto do endereço de grupo. As chaves agregadas privadas de gasto, irão ser usadas para assinar transacções.
Uma hash da chave privada do segredo partilhado :
$$k^{sh} = \mathcal{H}_n(k_A^{base} * k_B^{base} G),$$
irá ser usada como a chave de ver :
$$k^{v} = \mathcal{H}_n(k^{sh}).$$

(b) *Construção da Transacção* : Seja que o endereço já possui pelo menos uma saída, e as imagens de chave são desconhecidas.
Existem dois signatários, o iniciador e o cosignatário.

i. *Iniciar uma transacção*: O iniciador decide começar uma transacção. Ele gera valores de abertura (e.g. αG) para todas as saídas possuídas, e compromete-se a estas saídas.
Ele depois cria imagens de chave parciais para essas saídas possuídas, e assina essas imagens com uma prova de legitimidade (secção 9.5).
Ele depois envia essa informação, juntamente com o seu endereço para receber montantes, ao cosignatário.

[3] Extender isto para além de (N-1)-de-N é provavelmente infazível sem adicionar mais passos, devido ás rondas adicionais necessárias para construir um endereço de multi-assinatura com uma barreira (`M`)mais baixa.
[4] Este processo é actualmente bastante similar á organisação de transacções de multi-assinaturas em Monero.

ii. *Fazer uma transacção parcial*: O cosignatário verifica que toda a informação na transacção inicial é válida. Ele depois decide os montantes e as saídas destinatárias, bem como as entradas e os membros desvio de anel, e a taxa de transacção.
Ele gera uma chave privada de transacção, cria endereços ocultos, compromissos de saída, montantes ocultos, máscaras de compromisso de pseudo saídas, e valores iniciais para os compromissos a zero.
Para provar que os montantes estão dentro do domínio, ele constroi a prova de domínio tipo Bulletproof para cada saída.
Ele também gera valores de abertura para a sua assinatura sem compromissos, escalares aleatórios para a assinatura MLSAG, imagens de chave parciais para saídas possuídas, e provas de legitimidade para essas imagens parciais. Tudo isto é enviado ao iniciador.

iii. *Assinatura parcial do iniciador* : O iniciador verifica que a informação da transacção parcial é válida, e é conforme as suas expectativas (os montantes e os recipientes são o que devem ser). O iniciador completa as assinaturas MLSAG e assina-as com as suas chaves privadas, depois envia esta transacção parcialmente assinada ao cosignatário juntamente com os valores de abertura revelados.

iv. *Finalizar a transacção*: O cosignatário finaliza assinar a transacção, e submite isso á rede.

Assinaturas de compromisso único

Ao contrário daquilo que é recomendado no capítulo de multi-assinatura, só um compromisso é dado por cada transacção parcial (pelo iniciador da transacção). E este compromisso é revelado depois do cosignatário enviar para fora esse valor de abertura explicitamente. O propósito de comprometer a valores de abertura (e.g. αG) é de prevenir que um cosignatário malicioso possa usar o seu próprio valor de abertura para alterar o desafio que é produzido. Isto permitiria ao cosignatário malicioso de descobrir as chaves de agregação dos outros cosignatários (secção 9.3).
É necessário somente um valor de abertura desconhecido, quando o actor malicioso gera o seu valor de abertura, para que lhe seja impossível controlar o desafio. [5] [6]

[5] Isto é também o porque do iniciador só revelar os seus valores de abertura depois de toda a informação de transacção ter sido determinada, assim nenhum signatário consegue alterar a mensagem MLSAG e influenciar o desafio.

[6] Assinar com só um compromisso, pode ser geralizado a assinar com (M-1) compromissos em que só o autor da transacção parcial é que não compromete-e-revela, e os restantes co-signatários só revelam quando a transacção está inteiramente determinada. Por exemplo, suponha-se que existe um endereço 3-de-3 com os co-signatários (A,B,C), que irão tentar assinar com só um compromisso. Os signatários B e C estão numa coalição maliciosa contra A, enquanto que C é o iniciador e B é o autor parcial da transacção. C inicia com um compromisso, e depois A oferece o seu valor de abertura (sem compromisso). Quando B constroi a transacção parcial, ele pode conspirar com C para controlar o desafio da assinatura de forma a expor a chave privada de A. Note-se também que assinar com (M-1) compromissos é um conceito original apresentado primeiro aqui. Ou seja não foi pesquisado nem investigado em outro lado por relatórios abstractos, nem existe implementação disto em código. Como tal, isto poderá vir a ser

[7]

Simplificar desta forma remove uma ronda de comunicação, o que tem consequências importantes para a experiência de interação entre o comprador e o vendedor.

10.2.2 Experiência com garantia para o utilizador

Aqui está uma compra online que involve uma multi-assinatura 2-de-3, apresentada em detalhe e passo a passo. As interacções seguintes são entre o comprador, o vendedor e o moderador.

Existem quatro optimizações instaladas.

moderadores pre-seleccionados

Ao escolher moderadores, vendedores podem criar um segredo partilhado com cada um deles, e para cada produto e publicar essa chave pública com a informação do produto. [8] Desta forma os compradores podem construir um endereço junto de multi-assinatura num só passo, assim que decidam comprar algo. Os compradores podem optar entre vários moderadores, o que lhes permite escolher aquele em que mais confiam.

Os compradores podem também, se não confiam nos moderadores aceites pelo vendedor, escolher e cooperar com um outro moderador online para fazer um endereço de multi-assinatura com a chave base do produto em causa. Depois de receber a ordem de compra, o vendedor pode aceitar esse novo moderador ou rejeitar a venda. [9]

Espera-se que a necessidade mútua que compradores e vendedores têm para utilizarem bons moderadores, gere ao longo do tempo uma hierarquia de moderadores organizados pela qualidade e justiça dos seus serviços. Moderadores com uma reputação menor irão receber menos dinheiro e servir para menos transacções e também de montantes mais baixos. [10]

totalmente erróneo.

[7] Uma forma de pensar sobre isto é de considerar o significado e propósito de um 'compromisso' (secção 5.1). Uma vez que a alice se compromete ao valor A, ela fica presa a esse compromisso, e não pode ter vantagens de informações do evento B (causado por bob) que acontece mais tarde. Para mais, se A ainda não foi revelado, então B não é influenciado por A. A alice e o bob têm a certeza de que A e B são independentes. Uma Assinatura de só um compromisso satisfaz esse standard, e é equivalente a assinaturas com todos os compromissos. Se o compromisso c é uma função unidireccional de valores de abertura $\alpha_A G$ e $\alpha_B G$ (e.g. $c = \mathcal{H}_n(\alpha_A G, \alpha_B G)$), então se inicialmente é comprometido a $\alpha_A G$, e $\alpha_B G$ é revelado depois de $C(\alpha_A G)$ aparecer. E $\alpha_A G$ é revelado depois de $\alpha_B G$ aparecer, então $\alpha_B G$ e $\alpha_A G$ são independentes. Como tal c, irá ser aleatório da perspectiva da alice e do bob, excepto se ambos colaborarem, e (enp).

[8] Estes endereços de multi-assinatura ainda são resistentes a testes de agregação de chave, porque os segredos partilhados do comprador são desconhecidos aos observadores.

[9] Para uma experiência mais fluída, um servico de garantia poderia estar 'sempre online', e em vez de ser usado um moderador pre-seleccionado, todos os endereços de multi-assinatura 2-de-3 são activamente construídos com esse serviço, quando uma ordem de compra é feita. Uma outra opção é utilizar multi-assinaturas inclusivas (secção 9.7), em que o moderador pre-seleccionado é actualmente um grupo de multi-assinatura 1-de-N. Desta forma sempre que há uma disputa, qualquer moderador desse grupo, que está disponível nesse momento pode intervenir. Isto provávelmente requer um esforço substancial para implementar.

[10] Nãe é claro qual será o melhor método de financiamento para os moderadores. Talvez eles recebem uma taxa fixa ou uma percentagem de cada transacção que moderam, ou então cada vez que são seleccionados como

Subendereços e ID's de produto

Vendedores criam uma nova chave base para cada ID de um produto, e essas chaves são usadas para construir endereços de multi-assinatura 2-de-3. [11] Quando um vendedor serve uma ordem de compra, este cria um subendereço para receber os fundos, o que pode ser usado para ligar ordens de compra aos pagamentos recebidos. O requisito para 'pagamentos baseados na ordem de compra' é satisfeito de forma efficaz, também porque fundos dirigidos a sub-endereços diferentes são trivialmente acessíveis da mesma carteira (secção 4.3).

Transacções parciais antecipadas

Transacções de multi-assinatura levam várias rondas, portanto estas são feitas assim que possível. Para a conveniência do utilizador, transacções parciais que só são usadas raramente (e.g. reembolsos) são feitas de forma antecipada. E assim estas estão disponíveis de imediato para assinar, quando for preciso.

Acesso condicional do moderador

Para a eficiência e privacidade, os moderadores só precisam de acesso aos detalhes de um negócio quando há a necessidade de resolver disputas. Para alcançar isto, a chave privada de ver de multi-assinatura, é uma hash da chave privada do segredo partilhado :

$$k^{v,grp}_{purchase-order} = \mathcal{H}_n(T_{mv}, k^{sh,2x3}_{AB}).$$

Em que T_{mv} é a chave de ver que separa domínios no mercado, e A e B correspondem com o vendedor e comprador, e

$$k^{sh,2x3}_{AB} = \mathcal{H}_n(k^{base}_A * k^{base}_B G).$$

Extende-se o que pode *ver* á transcrição de comunicação entre comprador e vendedor. Por outras palavras, a chave que encripta essa comunicação é :

$$k^{ce}_{ordem-de-compra} = \mathcal{H}_n(T_{me}, k^{v,grp}_{ordem-de-compra}),$$

e T_{me} é a chave que encripta o separador de domínio do mercado . [12] [13] Os moderadores ganham acesso á comunicação entre o comprador e o vendedor, e a abilidade de autorizar pagamentos, só quando um dos partidos lhes liberta a chave de ver. [14]

moderadores (e se a taxa não está presente na transacção original parcial, o moderador recusa a disputa). Ou então os utilizadores destes mercados com garantia estabelecem contratos com os moderadores.

[11] Esta chave base também é utilizada para comprar com multi-assinaturas 1-de-2. É importante não expor a chave privada de gasto no canal de comunicação, portanto usar um segredo partilhado entre o comprador e o vendedor, faz sentido.

[12] Separar a chave de ver e a chave de encriptação permite oferecer direitos de ver ao histórico de comunicações sem dar direitos de ver ao histórico de transacções do endereço de multi-assinaturas.

[13] Este método também é utilizado para endereços de multi-assinatura 1-de-2.

[14] É importante que os moderadores verifiquem que o histórico de comunicação que eles recebem não foi falsificado. Uma forma seria que cada co-signatário incluísse uma hash assinada da mensagem com o histórico, sempre que este é enviado para fora. Os moderadores podem olhar para esta comunicação, com as hashes dos históricos para identificar discrepâncias. Também ajudaria aos co-signatários identificar mensagens que falharam na transmissão, e alternativamente criar a evidência que certos co-signatários receberam de facto certas mensagens.

Para mais, vendedores podem verificar que o anfitrião do mercado online (que pode também ser o único moderador disponível, dependendo da forma como isto é implementado) não é MITM ('homem no meio') das suas comunicações com os clientes. Os vendedores verificam que as chaves base que eles publicam para cada produto é igual aquele que é mostrado publicamente no mercado. Como a chave base do comprador, que é usada para criar o endereço de multi-assinatura, também é parte da chave de encriptação, um anfitrião malicioso não consegue orquestrar um ataque MITM (enp).

CAPÍTULO 11

Transacções juntas (TxTangle)

Existem algumas heurísticas para construir grafos de transacções, que são inevitáveis e que resultam da natureza de certos scenários e entidades. Em particular, o comportamento de mineiros, piscinas (do inglés : pools; secção 5.1 de [**?**]), mercados online e bolsas de câmbio apresentam padrões abertos a análise apesar do protocolo privado de Monero.

Descreve-se aqui *TxTangle*, análogo ao *CoinJoin* de Bitcoin um méthodo para confundir essas heuristicas.
[1]

Em essência, várias transacções são juntas para formar uma transacção, tal que os padrões de cada participante, derretam juntamente.

Para alcançar tal obfuscação, tem de ser praticamente impossivel para que observadores usem essa informação contida numa transacção junta para associar entradas e saídas a participantes individuais ou até de saber quantos participantes estão envolvidos.
[2]

Para além disso, mesmo os próprios participantes não deviam estar a par desse número. Nem de associar entradas e saídas a outros participantes (excepto em certos cenários). [3]
Finalmente devia ser possível de construír transacções juntas sem ter de depender numa autoridade central [**?**]. Felizmente todos esses requisitos são alcancados em Monero.

[1] Este capítulo constitui a proposta para um protocolo de transacções juntas. Isto até á data ainda não foi implementado. Uma proposta prévia, chamada *MoJoin* e criada pelo co-autor, investigador do laboratório de pesquisas de Monero (MRL), de pseudónimo Sarang Noether, requeria uma entidade terçeira de confiança para funcionar. Como tal *MoJoin*, não foi mais desenvolvido, e não está implementado.

[2] Como em Bitcoin os montantes são claramente visíveis, é muitas vezes possível agrupar saídas e entradas de transacções com base na soma dos montantes [**?**].

[3] Poluír transacções juntas de forma maliciosa é um potencial ataque a este método, e que foi primeiro identificado para CoinJoin. [**?**]

11.1 Construir transacções juntas

Numa transacção normal, entradas e saídas são construídas usando a prova de que os montantes se igualam. Da secção 6.1.1, a soma de pseudo compromissos de saída é igual á soma de compromissos de saída (mais o compromisso da taxa).

$$\sum_j C'^a_j - (\sum_t C^b_t + fH) = 0$$

Uma transacção junta trivial podia pegar todo o conteúdo de múltiplas transacções, e juntá-las. As mensagens MLSAG assinariam todos os dados das sub-transacções, e os montantes iriam-se igualar $(0 + 0 + 0 = 0)$.

[4] Porém, grupos de entradas e saídas seriam identificados trivialmente, porque seria possível testar se conjuntos de entradas e saídas têm montantes que se igualam. [5]

É possivel dar a volta a isto ao calcular segredos partilhados entre pares de participantes, ao adicionar estes offsets ás máscaras de pseudo compromissos de saída (Section 5.4). Em cada par, um participante adiciona o segredo partilhado a um dos seus pseudo compromissos de saída, e um outro participante subtrai isso de um dos *seus* pseudo compromissos. Na sua soma, os segredos partilhados cancelam-se mutuamente.

E desde que cada par de participantes tem um segredo partilhado, o montante só aparece depois de todos os compromissos serem combinados. [6]

11.1.1 Canal de comunicação em grupo

O número máximo de participantes para um *TxTangle* é o número de entradas ou de saídas, dependendo de qual é menor. Isto é modelado através de cada participante real pretender ser uma pessoa different para cada saída que ele envia. Isto é feito para estabelecer um canal de comunicação em grupo com outros potenciais participantes, sem que seja revelado quantos participantes existem.

Imagine-se n $(2 \leq n \leq 16$, apesar de que pelo menos 3 é recomendado) pessoas supostamente não relacionadas juntam-se a um *chatroom*. Agendado para abrir no tempo t_0 e fechar no tempo t_1. Só 16 pessoas podem estar no *chatroom* ao mesmo tempo. E existem condições como prioridades de taxa, taxa base por cada byte e tipos de transacção aceites. Ao tempo t_1 todos os membros signalam que querem proceder ao publicarem uma chave pública, e o *chatroom* é convertido para um canal de comunicação de grupo ao construir um segredo partilhado entre todos os membros desvio. [7] [8] Este segredo partilhado é usado para encriptar conteúdos de mensagens, enquanto que membros desvio assinam mensagens relacionadas a entradas com assinaturas SAG (secção 3.3).

[4] Desde que provas de domínio do tipo *Bulletproof* são agregadas para uma só (secção 5.5), os participantes teriam de colaborar entre sí até um certo grau, mesmo no caso trivial.

[5] Os participantes poderiam tentar dividir a taxa de uma forma esperta para confundir os observadores, mas isto iria falhar á face da *força bruta*, pois as taxas não são assim tão grandes (por volta de 32 bits ou menos).

[6] O *offset* é feito dos pseudo compromissos de saída, em vez dos compromissos de saída, pois as máscaras dos compromissos de saída são construídas a partir do endereço do destinatário (secção 5.3).

[7] Actualmente uma transacção pode ter no máximo 16 saídas.

[8] O método de multi-assinatura da secção 9.6.3 é uma forma, que extende M-de-N até 1-de-N.

Portanto não é claro quem é que enviou uma dada mensagem.
As mensagens relacionadas com as saídas usam uma assinatura tipo bLSAG (secção 3.4) no conjunto das chaves públicas dos membros desvio tal que as saídas não estão associadas aos membros desvio. [9]

11.1.2 Rondas de mensagens para construir uma transacção junta

Depois do canal estar pronto, transacções de *TxTangle* podem ser construídas em cinco rondas de comunicação, em que a próxima ronda só pode começar se a ronda prévia foi finalizada. Em cada ronda as mensagens são publicadas dentro de um certo intervalo de tempo limite, e o tempo de publicação de cada uma é aleatório. Isto serve para prevenir que grupos de mensagens revelem conjuntos de saída/entrada.

(a) Cada membro desvio gera privadamente um escalar aleatório para cada saída, e assina esta com uma assinatura tipo bLSAG. Uma lista ordenada destes escalares serve para determinar indexes de saída. O menor escalar recebe o index $t = 0$ (secção 4.2.1).
[10] Essas assinaturas são publicadas, e também assinaturas SAG que assinam o número da versão de entradas de transacção. Depois desta ronda os participantes podem calcular o peso de transacção baseado no número de entradas e saídas, e estimar precisamente a taxa necessária. [11] [12] [13]

(b) Cada membro desvio utiliza a lista de chaves públicas para construir o segredo partilhado com cada outro membro. Isto é feito para fazer um offset dos pseudo compromissos de saída. É decidido quem adiciona ou subtrai baseado na chave pública menor entre cada par.
Cada membro desvio tem de pagar $1/n$ da taxa estimada (usando divisão inteira). O membro desvio com o index de saída menor, tem a responsabilidade de pagar o resto da divisão inteira (o que deve ser um montante infinitésimal, mas tem de ser posto em conta para evitar que a transacção seja reconhecida como junta).
Eles geram privadamente chaves públicas de transacção para cada uma das suas

[9] Cada conjunto separado de assinaturas TxTangle bLSAG, devia usar as mesmas imagens de chave, desde que tudo relacionado com uma dada saída está ligado.
[10] A selecção do index de saída devia ser igual a outras implementações que constroiem transacções, para evitar que software diferente seja identificado. Utiliza-se efectivamente uma abordagem aleatória para alinhar com a implementação núcleo, que também mistura as saídas de forma aleatória.
[11] Se acontece que só podem haver dois participantes reais, baseado na comparação do número de entradas e saídas total, com o número de entradas/saídas do próprio participante, o TxTangle pode ser abandonado. Recomenda-se que cada participante tenha pelo menos duas entradas e duas saídas, no caso de actores maliciosos que não abandonam TxTangles mesmo quando reconhecem que só existem dois participantes. Esta recomendação está aberta ao debate, pois usar mais entradas e saídas não é heuristicamente neutral.
[12] Transacções de TxTangle não deviam ter informação adicional no campo *extra* (e.g. nenhum ID de pagamento, a não ser que seja um TxTangle com só duas saídas, que deve ter pelo menos um ID de pagamento desvio encriptado)
[13] A estimação da taxa devia ser baseada numa abordagem standard, assim cada participante calcula a mesma coisa. De outra forma as saídas podem ser agrupadas com base no método do cálculo da taxa. Este standard da taxa devia ser implementado fora de TxTangle, para promover que transacções de TxTangle se parecam iguais a transacções normais.

saídas (por enquanto não se envia isto a outros membros), e constroíem os compromissos de saída, montantes codificados, a parte A de provas parciais para serem utilizadas para a prova agregada de domínio *Bulletproof*, e assinar isto tudo com bLSAGs (um compromisso, um montante encriptado, uma prova parcial por mensagem bLSAG, e a imagem de chave liga estes dados á lista original de escalares aleatórios que foi utilizado para especificar indíces de saída). Pseudo compromissos de saída são gerados normalmente (secção 5.4), depois o offset com o segredo partilhado, e assinado com SAGs. Depois de bLSAGs e SAGs serem publicados, e assumindo que os participantes estimaram o total em taxas da mesma forma, eles podem agora verificar que os montantes em geral se igualam.
14 15

(c) Se os montantes se igualam própriamente, começa uma ronda para construir as provas de domínio agregadas, o que prova que todas os montantes de saída estão no domínio.
Cada membro desvio usa as provas parciais e os compromissos de saída da ronda parte A, e calcula privadamente o desafio agregado A. Depois constrõem a prova parcial parte B que depois é enviada para o canal usando uma assinatura tipo bLSAG.

(d) Os participantes começam a preencher a mensagem para ser assinada com assinaturas do tipo MLSAG. Dois tipos de mensagens são publicados em ordem aleatória dentro do intervalo de comunicação.
Cada offset de entrada que pertence a um anel, e as imagens de chave são assinadas com uma assinatura tipo SAG, e associadas com o pseudo compromisso de saída correcto.
Cada endereço oculto de uma saída, chave pública de transacção, e a prova parcial parte C são assinados com uma assinatura do tipo bLSAG. A prova parcial parte C é calculada com base nas provas parciais da parte B, e com o desafio agregado B. Pode também existir uma componente aleatória que é uma chave pública de transacção, esta serve para mitigar o "Janus falso".

(e) Os participantes usam as provas parciais para completar a prova de domínio agregada tipo *Bulletproof*. E cada um aplica uma técnica de produto interno logarítmico para que a compressão e a prova final seja incluída nos dados de transacção. Uma vez que todas as informações a serem assinadas por assinaturas do tipo MLSAG estão presentes. Cada participante completa as assinaturas MSLAG ás entradas e envia isso (com uma assinatura SAG para cada) para o canal de comunicação.
Cada participante pode submeter a transacção quando quiser.

[14] Visto que os pontos de CE são comprimidos (secção 2.4.2), as chaves são interpretadas como inteiros de 32 bytes. É convenção que o dono da chave menor adiciona, e o dono da chave maior subtrai.

[15] Não se publicam os montantes das taxas separadas pagas, no caso em que um participante se enganou no cálculo da taxa. O que pode revelar um grupo de saídas, devido aos montantes estranhos que sobressaem. Se os montantes não...?

Chaves públicas de transacção e a mitigação de Janus

Se cada participante de um *TxTangle* sabe a chave privada de transacção r (Section 4.2), então qualquer um deles pode testar os endereços ocultos de saída, contra uma lista de endereços conhecidos. Por causa disto é necessário construír transacções de *TxTangle* como se existisse um recipiente com um sub-endereço (secção 4.3), e também diferentes chaves públicas de transacção para cada saída.

Para complementar uma implementação possível de uma mitigação do ataque de Janus a sub-endereços, em que uma adicional chave pública de transacção 'base' é incluída no campo extra [**?**], transacções do tipo *TxTangle* também devem ter uma falsa chave pública composta por uma soma de chaves aleatórias geradas por cada membro desvio. Muitos participantes de uma transacção *TxTangle* que enviam dinheiro para um sub-endereço irão ter provavelmente duas saídas. Uma das quais diverte o troco de volta para o participante. Isto significa que cada participante pode activar uma mitigação de Janus ao tornarem a chave pública de transacção do troco também a chave 'base' para o recipiente do sub-endereço.

[16] [17] O recipiente do sub-endereço poderá realizar que a transacção é do tipo *TxTangle*, e que a chave 'base' provavelmente corresponde á saída de transacção do troco do participante. [18]

11.1.3 Fraquezas

Actores maliciosos têm duas opções para derrotar o propósito de transacções do tipo *TxTangle*, que é de esconder conjuntos de saídas/entradas de transacção de analistas ou adversários. As transacções podem ser poluídas, tal que o conjunto de participantes é menor ou até não existente [**?**]. Eles podem também tentar fazer com que tentativas de transacção *TxTangle* falhem. E usar as tentativas subsequentes pelos mesmos participantes para estimar conjuntos de saídas/entradas.

O primeiro caso não é fácil de mitigar, especialmente no caso decentralizado em que nenhum participante tem uma reputação. Uma possível aplicação de *TxTangle* é a de piscinas colaborativas, que podem esconder a qual piscina pertencem os mineiros. Tais piscinas saberiam os conjuntos de entradas/saídas, mas desde que o propósito

[16] O cancelamento de chave (secção 9.2.2) não devia ser um problema, desde que é só uma chave falsa e devia idealmente ser indexada de forma aleatória, entre a lista de chaves públicas de transacção.

[17] Se alguêm envia para o seu próprio sub-endereço, não existe a necessidade para uma mitigação de Janus. As carteiras que estão activadas para a mitigação de Janus deviam reconhecer que o montante gasto numa transacção TxTangle, é igual ao montante recebido pelo próprio sub-endereço. Para que o utilizador não seja notificado, de forma errónea, de um problema.

[18] Isto assume que as chaves públicas de transacção são 1:1 com as saídas, como aparenta ser o caso actualmente. Se fosse standard para as chaves públicas de transacção estarem numa ordem aleatória ou estrita dentro do campo *extra*, então as transacções TxTangle e não TxTangle seriam largamente indistinguíveis para os destinatários com sub-endereços. Existem casos específicos em que os participantes de TxTangle são incapazes de incluir uma chave 'base' (e.g. quando todas as saídas são para sub-endereços), ou quando se trata claramente de uma transacção não TxTangle, pois o destinatário do sub-endereço recebe a maioria ou todas as saídas. Note-se que transacções TxTangle geralmente têm muito mais saídas do que uma transacção típica, esta observação pode ser usada para diferenciar transacções normais de TxTangles.

é ajudar os mineiros a elas ligados, existe uma motivação de os ajudar e de manter esta informação secreta. Para mais, tais transacções *TxTangle* não permitiriam maus actores, assumindo que as piscinas são honestas.

O segundo caso pode ser evitado ao só tentar participar em transacções *TxTangle* poucas vezes antes de fazer uma pausa. E sempre que novas tentativas são feitas utilizar os elementos constituintes da transacção, regenerados de forma aleatória. Estes elementos são : as chaves públicas de transacção, máscaras de pseudo compromissos, escalares da prova de domínio, e escalares MLSAG. Em particular, o conjunto de membros desvio de anel para cada entrada devia permanecer o mesmo para prevenir comparações cruzadas que revelem a verdadeira entrada. Se possível, diferentes entradas verdadeiras deveriam ser utilizadas para tentativas de *TxTangle* diferentes. Desde que esta fraqueza é inevitável, torna a próxima secção mais atrativa.

11.2 Servidor TxTangle

TxTangle decentralizado tem algumas questões abertas. Como é que as rondas são iniciadas e enforçadas? Como é que são criados *chatrooms*, para que os participantes se encontrem mútuamente? A maneira mais simples é através de um servidor de *TxTangle*, que gera e mantêm esses *chatrooms*.

Um servidor desses parece desafiar o objectivo da participação obfuscada, desde que cada individuo tem de se ligar, e enviar mensagens que podem ser usadas para correlar entradas e saídas. Pode-se usar uma rede do tipo I2P para fazer com que cada mensagem recebida pelo servidor aparente provenir de um individuo único.
[19]

11.2.1 Comunicação básica com um servidor sobre I2P, e outras características

Com I2P, utilisadores fazem assim chamados 'túneis' que passam mensagens encriptadas por outros clientes antes de chegarem á destinação. Daquilo que se percebe, estes túneis podem transportar múltiplas mensagens antes de serem destruídos e recriados É essencial para este uso, que haja um controlo cuidado sobre a abertura e o fecho dos túneis, bem como quais mensagens saem do mesmo túnel. [20]

(a) *Inscrever-se para TxTangles*: Na nossa proposta original com n direcções (secção 11.1.1) os participantes gradualmente adicionam os seus membros desvio a quartos de TxTangle disponíveis antes destes estarem agendados para fechar. Contudo

[19] O projecto invisível da internet (https://geti2p.net/en/).

[20] Em I2P existem túneis de 'entrada' e de 'saída' (see https://geti2p.net/en/docs/how/tunnel-routing). Tudo o que é recebido através de um túnel de entrada aparenta ser da mesma fonte mesmo que existam múltiplas fontes. Portanto á superficie parece que utilizadores de TxTangle não precisam de criar túneis distintos para cada um dos seus usos. Contudo se o anfitrião TxTangle faz com que ele próprio seja o ponto de entrada para o seu próprio tunel de entrada, então ele ganha o acesso directo aos túneis de saída dos participantes de TxTangle.

se um volume suficientemente grande de utilisadores tenta fazer um TxTangle ao mesmo tempo, é provável de haver muitas falhas. Os utilizadores tentam de forma aleatória pór todas as suas saídas no mesmo 'quarto' de TxTangle, mas assim os quartos enchem-se demasiadamente rápido, e como tal os utilizadores teriam de abortar o TxTangle.

Pode-se fazer uma optimização ao dizer ao anfitrião quantas saídas estão em causa (e.g. dando-lhe uma lista das chaves públicas dos membros desvio), e deixar que ele junte os participantes para o TxTangle. Desde que os participantes retêm o protocolo de mensagens SAG e bLSAG, o anfitrião não será capaz de identificar conjuntos de saídas na transacção final. Tudo o que ele sabe é o número de participantes, e quantas saídas cada um tinha. Para mais, neste cenário os observadores não conseguem monitorizar quartos de TxTangle abertos para inferir qualquer informação sobre os participantes, uma melhoria importante na privacidade. Note-se que o poder do anfitrião de polluir os TxTangles não é significamente diferente do modelo sem anfitrião, portanto esta mudança é neutral para esse vector de ataque.

(b) *Método de comunicação*: Como o anfitrião já actua como o lugar de transporte das mensagens, é simples que seja ele a gerir a comunicação de TxTangle. Durante cada ronda o anfitrião colecciona mensagens dos membros desvio (estas comunicações ocorrem sobre intervalos de tempo aleatórios). E no fim de uma ronda existe uma curta phase de distribuição em que ele envia os dados coleccionados a cada participante, a isto dá-se um certo tempo de espera antes da próxima ronda para que se garanta que as mensagens foram recebidas e que houve tempo suficiente para as processar.

(c) *Túneis e grupos de saídas/entradas*: Quando um TxTangle é iniciado, os participantes devem desasociar as identidades dos membros desvio das saídas verdadeiras. Isto significa criar novos túneis para mensagens assinadas tipo bLSAG. Cada túnel destes so pode transmitir mensagens relativamente a uma certa saída (a informação sobre uma saída vem sempre da mesma fonte). Os participantes devem também criar novos túneis para mensagens assinadas tipo SAG, respectivas a certas entradas.

(d) *Ameaça de um ataque MITM pelo anfitrião*: O anfitrião pode enganar um participante ao pretender ser um outro participante, visto que ele controla o envio da lista de membros desvio para construir bLSAGs e SAGs. Por outras palavras a lista que ele envia ao participante A, pode conter os membros desvio do participante A, e todo o resto são os seus próprios. As mensagens recebidas pelo participante B são assinadas outra vez com a lista de A antes de ser re-transmitida a A. Como todas as mensagens assinadas por A pertencem a A, o anfitrião teria visibilidade directa para os agrupamentos de entradas/saídas de A.

É possível prevenir que o anfitrião aja como MITM a partir de interações honestas entre os participantes, ao modificar como as chaves públicas de transacção são feitas. Os participantes enviam uns aos outros as chaves públicas de transacção (com uma assinatura bLSAG), depois, bem como na agregação robusta de chave

da secção 9.2.3, as chaves que de facto são incluídas nos dados de transacção (e que são utilizados para fazer máscaras de compromissos de saída, etc.) são prefixos com uma hash da lista dos membros desvio. Por outras palavras, a *t-ésima* chave pública de transacção é :

$$\mathcal{H}_n(T_{agg}, \mathbb{S}_{mock}, r_t G) * r_t G$$

Pôr a lista de membros desvio dentro da transacção faz com que seja muito difícil completar TxTangles sem que haja uma comunicação directa entre todos os actuais participantes. [21]

11.2.2 Um anfitrião como um serviço

É importante para a sustentabilidade e para a melhoria continua que um serviço de TxTangle opere com lucro. [22] Em vez de comprometer as identidades dos utilizadores com um modelo de conta, o anfitrião pode participar em cada TxTangle com uma só saída, e requerer que os participantes financiem essa saída. Quando os participantes acedem ao serviço/'eepsite' do anfitrião, são informados dessa taxa actual. Esta taxa do anfitrião deve ser paga por cada saída.

Participantes seriam responsáveis por pagar fracções da taxa de mineiro e a taxa do anfitrião. Desta vez, a chave pública mais pequena do membro desvio (excluindo a chave do anfitrião) pagaria o resto de ambas as taxas. [23] Desde que o anfitrião não tem nenhuma entrada, ele também não tem nenhum pseudo compromisso de saída para cancelar a sua máscara do compromisso de saída. Em vez disso ele cria segredos partilhados com os outros membros desvio, depois separa a sua máscara de compromisso em partes de tamanho aleatório e divide estas partes pelos segredos partilhados. Ele pública uma lista desses escalares, assina com a sua chave tal que os participantes saibam que se trata do anfitrião. Esta lista, ao aparecer assinála o começo da ronda n° 1 da secção 11.1.2 (e.g. o fim da ronda '0'). Membros desvio irão multiplicar o seu escalar do anfitrião pelo segredo partilhado apropriado, e adicionam isso á sua máscara de pseudo compromisso. Desta forma, mesmo a saída do anfitrião não pode ser identificada por qualquer participante na transacção final sem que haja uma coalição total contra ele.

Para simplificar as calculações da taxa, o anfitrião pode distribuir a taxa total a ser usada na transacção no fim da ronda n° 1, pois ele irá saber qual é o peso da transacção relativamente cedo. Os participantes podem verificar que esse montante é semelhante ao montante esperado, e pagar uma fracção do mesmo.

[21] Se a mitigação de Janus for implementada, esta defesa ao ataque MITM devia em vez disso ser feita com a chave base falsa de Janus. Cada membro desvio oferece uma chave aleatória $r_{desvio}G$, depois a actual chave base é :

$$\sum_{desvio} \mathcal{H}_n(T_{agg}, \mathbb{S}_{desvio}, r_{desvio} G) * r_{desvio} G$$

[22] Enquanto que um serviço de TxTangle opere com lucro, o próprio código pode ser de código aberto. Isto é importante para auditar o software de carteira que interage com um tal serviço.

[23] É necessário utilizar aqui as chaves do membros desvio, pois o anfitrião não paga taxa, e o seu index de saída é desconhecido.

Se os participantes colaboram para enganar, e não querem pagar o preço do anfitrião, então o anfitrião pode terminar o TxTangle na ronda n° 3. Ou então se as mensagens que aparecem no canal não deviam estar ali ou são inválidas.

No fim da ronda 5 o anfitrião completa a transacção e submete isso á rede para verificação, como parte do seu serviço. Ele incluí a hash da transacção na mensagem que é distribuida no fim.

11.3 Usar um administrador á confiança

Existem aspectos negativos para o TxTangle decentralizado. Requer que os participantes se encontrem uns aos outros, e que comuniquem activamente dentro de um tempo planeado. Isto é também difícil de implementar em código.

Usar um administrador central, que é responsável para coleccionar informação de transacção de cada participante e obfuscar os grupos de entrada/saída, simplifica o processo. O custo é ter uma entidade terçeira á confiança, que sabe (no mínimo) esses grupos de entrada/saída. [24]

11.3.1 Processo baseado num administrador

O administrador revela que ele está disponível para gerir TxTangles, e colecciona inscrições de participantes potenciais (que consistem do número de entradas [com os seus tipos] e saídas). Ele pode se quiser participar com o seu próprio conjunto de saídas/entradas.

Depois de quase 16 saídas terem sido juntadas num grupo (têm de existir pelo menos dois ou mais participantes, e nenhum participante deve ter todas as entradas ou saídas), o administrador começa a primeira de 5 rondas. Em cada ronda ele acumula informação de cada participante, faz algumas decisões, e envia mensagens que significam o começo de uma nova ronda.

(a) Para começar, o administrador gera, para cada par de participantes, um escalar aleatório, e decide quem nesse par recebe a versão negativa desse escalar. Ele usa o número e tipo de entradas e saídas, para estimar a taxa total necessária. Ele faz a soma dos escalares dos participantes, e de forma privada envia para cada um essa soma, juntamente com a taxa que eles devem pagar e os indíces das suas saídas (escolhidas de forma aleatória). Estas mensagens constituem um sinal para os participantes que um TxTangle está a começar.

(b) Cada participante constroí a sua sub-transacção como se fosse uma transacção normal, com chaves públicas de transacção para as suas saídas (com uma mitigação de Janus se necessário), calcular endereços ocultos de saída, codificar montantes de saída, fazer pseudo compromissos de saída que igualam os compromissos de saída mais a taxa para os mineiros, fazer uma lista de offsets de membros de anel para

[24] Esta secção é inspirada pelo protocolo *MoJoin*.

o uso em assinaturas MLSAG juntamente com as imagens de chave respectivas, e adicionar a um dos seus pseudo compromissos de saída, o escalar enviado pelo administrador (multiplicado por G). Eles criam a parte A de provas parciais para as suas saídas, e enviam toda esta informação ao administrador. O administrador verifica que os montantes de entrada e saída se igualam, e depois envia a lista completa da parte A de provas parciais a cada participante.

(c) Cada participante calcula o desafio agregado A, e gera a parte B de provas parciais que depois são enviadas ao administrador. Este por sua vez colecciona as provas parciais e distribui estas por todos os participantes.

(d) Cada participante calcula o desafio B, e gera a parte C de provas parciais que eles enviam ao administrador. O administrador por sua vez colecciona estas provas e aplica-lhes a técnica do produto interno logarítmico para as comprimir para a prova final. Assumindo que a prova se verifica como devia, ele gera uma chave pública 'base' de transacção falsa tipo Janus, e envia a mensagem para ser assinada em MLSAGs para cada participante.

(e) Cada participante completa a sua assinatura MLSAG e envia isso ao administrador. Uma vez que este tem todas as peças, ele pode finalizar a construção da transacção, e submeter isso á rede. Ele pode também enviar a ID da transacção a cada participante para que estes confirmem que ela foi publicada.

Appendices

APÊNDICE A

Estrutura de transacção `RCTTypeBulletproof2`

Apresenta-se neste apêndice uma transacção de Monero real do tipo `RCTTypeBulletproof2`, juntamente com as esplicações dos seus campos relevantes.

Esta transacção foi obtida do explorador de blocos https://xmrchain.net. Também se pode obter esta transacção ao executar o commando `print_tx <TransactionID> +hex +json` no *daemon* `monerod` (sem o parámetro *detached*). Em que `<TransactionID>` é uma hash da transacção (secção **??**). A primeira linha mostra o commando que se executou. Para replicar estes resultados, o leitor faz o seguinte :

(a) Precisa-se a *cli* de Monero, que se encontra em https://web.getmonero.org/downloads/. Saque somente a *cli*, para o seu sistema operativo, mova o ficheiro para um directório qualquer, e extrai-a o arquivo.

(b) Abra um terminal/linha de comandos e navegue para o directório criado pela extração.

(c) Corra `monerod` com `./monerod`. Agora a lista de blocos irá ser saquada da rede para a sua máquina local. Infelizmente não existe outra forma de imprimir transacções localmente sem ter posse da lista inteira de blocos.

(d) Depois do processo de sincronização estar feito, é possível executar comandos como `print_tx`. Use `help` para conhecer outros comandos.

O componente `rctsig_prunable`, é podável da lista de blocos. Isto é, uma vez que há consenso sobre um bloco e o comprimento actual da lista de blocos proíbe a possibilidade de um ataque de duplo gasto, este campo inteiro pode ser podado e substituido pela hash da raíz da arvore de *merkle*.

Imagens de chave são guardadas separadamente, na área não podável das transacções. Estas imagens de chave são cruciais para detectar ataques de duplo gasto e não podem ser podadas.

Esta transacção exemplo têm duas entradas e duas saídas, e foi adicionada á lista de blocos com o carimbo no tempo de 2020-03-02 19:01:10 UTC .

```
print_tx 84799c2fc4c18188102041a74cef79486181df96478b717e8703512c7f7f3349
Found in blockchain at height 2045821
{
  "version": 2,
  "unlock_time": 0,
  "vin": [ {
      "key": {
        "amount": 0,
        "key_offsets": [ 14401866, 142824, 615514, 18703, 5949, 22840, 5572, 16439,
        983, 4050, 320
        ],
        "k_image": "c439b9f0da76ca0bb17920ca1f1f3f1d216090751752b091bef9006918cb3db4"
      }
    }, {
      "key": {
        "amount": 0,
        "key_offsets": [ 14515357, 640505, 8794, 1246, 20300, 18577, 17108, 9824, 581
        637, 1023
        ],
        "k_image": "03750c4b23e5be486e62608443151fa63992236910c41fa0c4a0a938bc6f5a37"
      }
    }
  ],
  "vout": [ {
      "amount": 0,
      "target": {
        "key": "d890ba9ebfa1b44d0bd945126ad29a29d8975e7247189e5076c19fa7e3a8cb00"
      }
    }, {
      "amount": 0,
      "target": {
        "key": "dbec330f8a67124860a9bfb86b66db18854986bd540e710365ad6079c8a1c7b0"
      }
    }
  ],
  "extra": [ 1, 3, 39, 58, 185, 169, 82, 229, 226, 22, 101, 230, 254, 20, 143,
  37, 139, 28, 114, 77, 160, 229, 250, 107, 73, 105, 64, 208, 154, 182, 158, 200,
  73, 2, 9, 1, 12, 76, 161, 40, 250, 50, 135, 231
  ],
  "rct_signatures": {
```

```
41      "type": 4,
42      "txnFee": 32460000,
43      "ecdhInfo": [ {
44          "amount": "171f967524e29632"
45        }, {
46          "amount": "5c2a1a9f54ccf40b"
47        }],
48      "outPk": [ "fed8aded6914f789b63c37f9d2eb5ee77149e1aa4700a482aea53f82177b3b41",
49      "670e086e40511a279e0e4be89c9417b4767251c5a68b4fc3deb80fdae7269c17"]
50    },
51    "rctsig_prunable": {
52      "nbp": 1,
53      "bp": [ {
54          "A": "98e5f23484e97bb5b2d453505db79caadf20dc2b69dd3f2b3dbf2a53ca280216",
55          "S": "b791d4bc6a4d71de5a79673ed4a5487a184122321ede0b7341bc3fdc0915a796",
56          "T1": "5d58cfa9b69ecdb2375647729e34e24ce5eb996b5275aa93f9871259f3a1aecd",
57          "T2": "1101994fea209b71a2aa25586e429c4c0f440067e2b197469aa1a9a1512f84b7",
58          "taux": "b0ad39da006404ccacee7f6d4658cf17e0f42419c284bdca03c0250303706c03",
59          "mu": "cacd7ca5404afa28e7c39918d9f80b7fe5e572a92a10696186d029b4923fa200",
60          "L": [ "d06404fc35a60c6c47a04e2e43435cb030267134847f7a49831a61f82307fc32",
61          "c9a5932468839ee0cda1aa2815f156746d4dce79dab3013f4c9946fce6b69eff",
62          "efdae043dcedb79512581480d80871c51e063fe9b7a5451829f7a7824bcc5a0b",
63          "56fd2e74ac6e1766cfd56c8303a90c68165a6b0855fae1d5b51a2e035f333a1d",
64          "81736ed768f57e7f8d440b4b18228d348dce1eca68f969e75fab458f44174c99",
65          "695711950e076f54cf24ad4408d309c1873d0f4bf40c449ef28d577ba74dd86d",
66          "4dc3147619a6c9401fec004652df290800069b776fe31b3c5cf98f64eb13ef2c"
67          ],
68          "R": [ "7650b8da45c705496c26136b4c1104a8da601ea761df8bba07f1249495d8f1ce",
69          "e87789fbe99a44554871fcf811723ee350cba40276ad5f1696a62d91a4363fa6",
70          "ebf663fe9bb580f0154d52ce2a6dae544e7f6fb2d3808531b0b0749f5152ddbf",
71          "5a4152682a1e812b196a265a6ba02e3647a6bd456b7987adff288c5b0b556ec6",
72          "dc0dcb2e696e11e4b62c20b6bfcb6182290748c5de254d64bf7f9e3c38fb46c9",
73          "101e2271ced03b229b88228d74b36088b40c88f26db8b1f9935b85fb3ab96043",
74          "b14aae1d35c9b176ac526c23f31b044559da75cf95bc640d1005bfcc6367040b"
75          ],
76          "a": "4809857de0bd6becdb64b85e9dfbf6085743a8496006b72ceb81e01080965003",
77          "b": "791d8dc3a4ddde5ba2416546127eb194918839ced3dea7399f9c36a17f36150e",
78          "t": "aace86a7a1cbdec3691859fa07fdc83eed9ca84b8a064ca3f0149e7d6b721c03"
79        }
80      ],
81      "MGs": [ {
82          "ss": [[ "d7a9b87cfa74ad5322eedd1bff4c4dca08bcff6f8578a29a8bc4ad6789dee106",
```

```
          "f08e5dfade29d2e60e981cb561d749ea96ddc7e6855f76f9b807842d1a17fe00"],
          ["de0a86d12be2426f605a5183446e3323275fe744f52fb439041ad2d59136ea0b",
          "0028f97976630406e12c54094cbbe23a23fe5098f43bcae37339bfc0c4c74c07"],
          ["f6eef1f99e605372cb1ec2b3dd4c6e56a550fec071c8b1d830b9fda34de5cc05",
          "cd98fc987374a0ac993edf4c9af0a6f2d5b054f2af601b612ea118f405303306"],
          ["5a8437575dae7e2183a1c620efbce655f3d6dc31e64c96276f04976243461e08",
          "5090103f7f73a33024fbda999cd841b99b87c45fa32c4097cdc222fa3d7e9502"],
          ["88d34246afbccbd24d2af2ba29d835813634e619912ea4ca194a32281ac14e0e",
          "eacdf59478f132dd8dbb9580546f96de194092558ffceeff410ee9eb30ce570e"],
          ["571dab8557921bbae30bda9b7e613c8a0cff378d1ec6413f59e4972f30f2470d",
          "5ca78da9a129619299304d9b03186233370023debfdaddcd49c1a338c1f0c50d"],
          ["ac8dbe6bb28839cf98f02908bd1451742a10c713fdd51319f2d42a58bf1d7507",
          "7347bf16cba5ee6a6f2d4f6a59d1ed0c1a43060c3a235531e7f1a75cd8c8530d"],
          ["b8876bd3a5766150f0fbc675ba9c774c2851c04afc4de0b17d3ac4b6de617402",
          "e39f1d2452d76521cbf02b85a6b626eeb5994f6f28ce5cf81adc0ff2b8adb907"],
          ["1309f8ead30b7be8d0c5932743b343ef6c0001cef3a4101eae98ffde53f46300",
          "370693fa86838984e9a7232bca42fd3d6c0c2119d44471d61eee5233ba53c20f"],
          ["80bc2da5fc5951f2c7406fce37a7aa72ffef9cfa21595b1b68dfab4b7b9f9f0c",
          "c37137898234f00bce00746b131790f3223f97960eefe67231eb001092f5510c"],
          ["01c89e07571fd365cac6744b34f1b44e06c1c31cbf3ee4156d08309345fdb20e",
          "a35c8786695a86c0a4e677b102197a11dadc7171dd8c2e1de90d828f050ec00f"]],
          "cc": "0d8b70c600c67714f3e9a0480f1ffc7477023c793752c1152d5df0813f75ff0f"
        }, {
          "ss": [[ "4536e585af58688b69d932ef3436947a13d2908755d1c644ca9d6a978f0f0206",
          "9aab6509f4650482529219a805ee09cd96bb439ee1766ced5d3877bf1518370b"],
          ["5849d6bf0f850fcee7acbef74bd7f02f77ecfaaa16a872f52479ebd27339760f",
          "96a9ec61486b04201313ac8687eaf281af59af9fd10cf450cb26e9dc8f1ce804"],
          ["7fe5dcc4d3eff02fca4fb4fa0a7299d212cd8cd43ec922d536f21f92c8f93f00",
          "d306a62831b49700ae9daad44fcd00c541b959da32c4049d5bdd49be28d96701"],
          ["2edb125a5670d30f6820c01b04b93dd8ff11f4d82d78e2379fe29d7a68d9c103",
          "753ac25628c0dada7779c6f3f13980dfc5b7518fb5855fd0e7274e3075a3410c"],
          ["264de632d9cb867e052f95007dfdf5a199975136c907f1d6ad73061938f49c01",
          "dd7eb6028d0695411f647058f75c42c67660f10e265c83d024c4199bed073d01"],
          ["b2ac07539336954f2e9b9cba298d4e1faa98e13e7039f7ae4234ac801641340f",
          "69e130422516b82b456927b64fe02732a3f12b5ee00c7786fe2a381325bf3004"],
          ["49ea699ca8cf2656d69020492cdfa69815fb69145e8f922bb32e358c23cebb0f",
          "c5706f903c04c7bed9c74844f8e24521b01bc07b8dbf597621cceeeb3afc1d0c"],
          ["a1faf85aa942ba30b9f0511141fcab3218c00953d046680d36e09c35c04be905",
          "7b6b1b6fb23e0ee5ea43c2498ea60f4fcf62f70c7e0e905eb4d9afa1d0a18800"],
          ["785d0993a70f1c2f0ac33c1f7632d64e34dd730d1d8a2fb0606f5770ed633506",
          "e12777c49ffc3f6c35d27a9ccb3d9b8fed7f0864a880f7bae7399e334207280e"],
          ["ab31972bf1d2f904d6b0bf18f4664fa2b16a1fb2644cd4e6278b63ade87b6d09",
```

```
125          "1efb04fe9a75c01a0fe291d0ae00c716e18c64199c1716a086dd6e32f63e0a07"],
126          ["a6f4e21a27bf8d28fc81c873f63f8d78e017666adbf038da0b83c2ad04ef6805",
127          "c02103455f93c2d7ec4b7152db7de00d1c9e806b1945426b6773026b4a85dd03"]],
128          "cc": "d5ac037bb78db41cf924af713b7379c39a4e13901d3eac017238550a1a3b910a"
129        }],
130      "pseudoOuts": [ "b313c1ae9ca06213684fbdefa9412f4966ad192bc0b2f74ed1731381adb7ab58
131      "7148e7ef5cfd156c62a6e285e5712f8ef123575499ff9a11f838289870522423"]
132    }
133  }
```

Componentes de transacção

- (linha 2) - O comando `print_tx` iria reportar o bloco em que foi encontrada a transacção, o que é replicado aqui com o propósito de demonstração.
- `version` (linha 4) - O formato/versão da transacção; '2' corresponde a RingCT.
- `unlock_time` (linha 5) - Previne que a(s) saída(s) de transacção sejam gastas antes que esse tempo passe. Trata-se aqui de uma altura de bloco, ou um carimbo no tempo de UNIX. Se o número é maior do que o tempo UNIX até á data. Este valor torna-se zero se nenhum limite é definido.
- `vin` (linha 6-23) - lista de entradas (existem duas aqui)
- `amount` (linha 8) - Descontinuado (legado); campo para o montante, para transacções do tipo 1.
- `key_offset` (linha 9) - Isto permite que os verificadores encontrem as chaves de membros de anel e compromissos na lista de blocos, o que implica óbviamente que estes membros são legítimos. O primeiro offset é absoluto dentro do histórico da lista de blocos, e cada offset subsequente é relativo ao anterior. Por exemplo, com offsets reais {7,11,15,20}, guarda-se na lista de blocos {7,4,4,5}. Os verificadores calculam o último offset como (7+4+4+5 = 20) (secção 6.1.5).
- `k_image` (linha 12) - Imagem de chave $\tilde{K}_j$ da secção 3.5, em que $j = 1$ desde que esta é a primeira entrada.
- `vout` (linhas 24-35) - Lista de saídas (existem duas aqui)
- `amount` (linha 25) - Descontinuado (legado); campo para o montante, para transacções do tipo 1.
- `key` (linha 27) - chave do endereço oculto de destino para a saída $t = 0$ como descrito na secção 4.2.
- `extra` (linhas 36-39) - Diversos dados, incluíndo a *chave pública de transacção*, i.e. o segredo partilhado rG da secção 4.2, e ID de pagamento da secção 4.4. Típicamente funciona da seguinte forma: cada número é um byte (de 0 a 255), e cada tipo diferente de dados têm um 'tag' e um 'comprimento'. O tag indica qual é a informação que se segue, e o comprimento indica quantos bytes essa informação ocupa. O primeiro número é sempre um tag. Aqui, '1' indica uma chave pública

src/cryptonote_basic/tx_extra.h

de transacção. Estas chaves têm sempre 32 bytes de comprimento, portanto não é necessário incluír o comprimento. Depois de 32 números encontra-se um novo tag : '2' o que significa 'nonce extra', o seu comprimento é '9', e o próximo byte é '1' o que significa um ID de pagamento codificado (o nonce extra pode ter campos próprios por alguma razão). Depois de 8 bytes o campo extra acaba. Veja-se [**?**] para mais detalhes. (Na especificação original de criptonote o primeiro byte indicava o tamanho do campo. Monero não utiliza isso.) [**?**]

- `rct_signatures` (linhas 40-50) - Primeira parte dos dados de assinatura
- `type` (linha 41) - Tipos de assinatura; `RCTTypeBulletproof2` é tipo 4. Tipos descontinuados RingCT `RCTTypeFull` e `RCTTypeSimple` são tipo 1 e 2 respectivamente. Transacções de mineiro são `RCTTypeNull`, tipo 0.
- `txnFee` (linha 42) - Taxa de transacção em texto claro, neste caso 0.00003246 xmr.
- `ecdhInfo` (linhas 43-47) - 'elliptic curve diffie-hellman Info': Montante oculto para cada uma das saídas $t \in \{0, ..., p-1\}$; em que $p = 2$
- `amount` (linha 44) - O campo *amount* com $t = 0$ é descrito na secção 5.3.
- `outPk` (linhas 48-49) - Compromissos para cada saída, secção 5.4.
- `rctsig_prunable` (linhas 51-132) - Segunda parte dos dados da assinatura
- `nbp` (linha 52) - Número de provas de domínio *Bulletproof* nesta transacção.
- `bp` (linhas 53-80) - elementos da prova *Bulletproof*

$$\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$$

- `MGs` (linhas 81-129) - assinaturas MLSAG
- `ss` (linhas 82-103) - Componentes $r_{i,1}$ e $r_{i,2}$ da assinatura MLSAG para a primeira entrada.

$$\sigma_j(\mathfrak{m}) = (c_1, r_{1,1}, r_{1,2}, ..., r_{v+1,1}, r_{v+1,2})$$

- `cc` (linha 104) - Componente c_1 da assinatura MLSAG, para a primeira entrada.
- `pseudoOuts` (linhas 130-131) - Pseudo compromissos de saída $C_j'^a$, como descrito na secção 5.3. Lembre-se que a soma destes compromissos é igual á soma dos dois compromissos de saída mais o compromisso da taxa de mineiro fH.

APÊNDICE B

Conteúdo de bloco

Neste apêndice mostra-se a estrutura de um bloco, nomeadamente o bloco n° 1582196. Este bloco contêm 5 transacções, e foi minerado para a lista de blocos com o carimbo no tempo 2018-05-27 21:56:01 UTC, pelo seu mineiro.

```
print_block 1582196
timestamp: 1527458161
previous hash: 30bb9b475a08f2ea6fe07a1fd591ea209a7f633a400b2673b8835a975348b0eb
nonce: 2147489363
is orphan: 0
height: 1582196
depth: 2
hash: 50c8e5e51453c2ab85ef99d817e166540b40ef5fd2ed15ebc863091ca2a04594
difficulty: 51333809600
reward: 4634817937431
{
  "major_version": 7,
  "minor_version": 7,
  "timestamp": 1527458161,
  "prev_id": "30bb9b475a08f2ea6fe07a1fd591ea209a7f633a400b2673b8835a975348b0eb",
  "nonce": 2147489363,
  "miner_tx": {
    "version": 2,
    "unlock_time": 1582256,
    "vin": [ {
        "gen": {
```

```
22            "height": 1582196
23          }
24        }
25      ],
26      "vout": [ {
27          "amount": 4634817937431,
28          "target": {
29            "key": "39abd5f1c13dc6644d1c43db68691996bb3cd4a8619a37a227667cf2bf055401"
30          }
31        }
32      ],
33      "extra": [ 1, 89, 148, 148, 232, 110, 49, 77, 175, 158, 102, 45, 72, 201, 193,
34      18, 142, 202, 224, 47, 73, 31, 207, 236, 251, 94, 179, 190, 71, 72, 251, 110,
35      134, 2, 8, 1, 242, 62, 180, 82, 253, 252, 0
36      ],
37      "rct_signatures": {
38        "type": 0
39      }
40    },
41    "tx_hashes": [ "e9620db41b6b4e9ee675f7bfdeb9b9774b92aca0c53219247b8f8c7aecf773ae",
42                   "6d31593cd5680b849390f09d7ae70527653abb67d8e7fdca9e0154e5712591bf",
43                   "329e9c0caf6c32b0b7bf60d1c03655156bf33c0b09b6a39889c2ff9a24e94a54",
44                   "447c77a67adeb5dbf402183bc79201d6d6c2f65841ce95cf03621da5a6bffefc",
45                   "90a698b0db89bbb0704a4ffa4179dc149f8f8d01269a85f46ccd7f0007167ee4"
46    ]
47  }
```

Componentes do bloco

- (linhas 2-10) - Informação dada pelo software cliente, isto não faz parte do bloco.
- `is orphan` (linha 5) - Significa que este bloco foi orfão. Os *nodes* usualmente guardam todos os ramos durante uma situação de garfo de rede, e discartam ramos desnecessários quando uma outra dificuldade comulativa vencedora emerge, o que torna os blocos do outro garfo órfaos.
- `depth` (linha 7) - A profundidade de qualquer dado bloco, é a distância desse bloco ao último bloco na lista (distância ao bloco mais recente).
- `hash` (linha 8) - A ID deste bloco.
- `difficulty` (linha 9) - A dificuldade não é guardada num bloco, desde que os utilizadores podem calcular *todas* as dificuldades de bloco a partir dos carimbos no tempo e com as regras na secção 7.2.

- `major_version` (linha 12) - Corresponde á versão do protocolo usada para verificar este bloco.
- `minor_version` (linha 13) - Originalmente destinada para ser um mecanismo entre mineiros, agora simplesmente repete a `major_version`.
- `timestamp` (linha 14) - Uma representação com um inteiro do carimbo no tempo em UTC, posto pelo mineiro.
- `prev_id` (linha 15) - A ID do bloco anterior.
- `nonce` (linha 16) - A nonce usada pelo mineiro do bloco para passar o alvo de dificuldade. É possível verificar que esta prova de trabalho é válida para esta nonce.
- `miner_tx` (linhas 17-40) - A transacção de mineiro
- `version` (linha 18) - O formato/era da transacção; version '2' corresponde a RingCT.
- `unlock_time` (linha 19) - A transacção de mineiro não pode ser gasta até ao bloco 1582256, até que mais 59 blocos tenham sido minerados (é um tempo de bloqueio de 60 blocos pois não pode ser gasto até que 60 intervalos de tempo entre blocos tenham passado, e.g. $2 * 60 = 120$ minutos).
- `vin` (linhas 20-25) - Entradas para as transacção de mineiro, não existem, pois a transacção de mineiro é usada para gerar o subsídio de bloco e coleccionar taxas de transacção.
- `gen` (linha 21) - Forma curta para 'gerar'.
- `height` (linha 22) - A altura de bloco. Cada altura de bloco só pode gerar o subsídio de bloco, uma vez.
- `vout` (linhas 26-32) - Saídas da transacção de mineiro.
- `amount` (linha 27) - Montante da transacção de mineiro, contêm o subsídio do bloco e as taxas de transacção deste bloco. Usa a unidade atómica *piconero* .
- `key` (linha 29) - Endereço oculto que é proprietário do montante na transacção de mineiro.
- `extra` (linhas 33-36) - Informação extra para a transacção de mineiro, inclui a chave pública de transacção.
- `type` (linha 38) - Tipo de transacção, neste caso '0' para `RCTTypeNull`.
- `tx_hashes` (linhas 41-46) - Todas as ID's de transacção incluídas neste bloco (mas não a ID da transacção de mineiro, que é : `06fb3e1cf889bb972774a8535208d98db164394ef2b14ecfe74814170557e6e9`).

APÊNDICE C

Bloco de génese

Neste apêndice mostra-se a estrutura do bloco génese, o bloco tem zero transacções, só envia o primeiro subsídio de bloco para thankful_for_today [**?**]). O fundador de Monero não adicionou um carimbo no tempo, talvez uma marca de Bytecoin, a moeda da qual provêm o código fonte de Monero. Mais sobre a história de Bytecoin pode ser lida aqui [**?**], e aqui [**?**].

O bloco 1 foi adicionado á rede com o carimbo no tempo 2014-04-18 10:49:53 UTC, assume-se que o bloco 0, de génese tenha sido minerado no mesmo dia. Isto corresponde com a data de lançamento anunciada por thankful_for_today [**?**].

```
print_block 0
timestamp: 0
previous hash: 0000000000000000000000000000000000000000000000000000000000000000
nonce: 10000
is orphan: 0
height: 0
depth: 1580975
hash: 418015bb9ae982a1975da7d79277c2705727a56894ba0fb246adaabb1f4632e3
difficulty: 1
reward: 17592186044415
{
  "major_version": 1,
  "minor_version": 0,
  "timestamp": 0,
  "prev_id": "0000000000000000000000000000000000000000000000000000000000000000",
  "nonce": 10000,
```

```
17    "miner_tx": {
18      "version": 1,
19      "unlock_time": 60,
20      "vin": [ {
21          "gen": {
22            "height": 0
23          }
24        }
25      ],
26      "vout": [ {
27          "amount": 17592186044415,
28          "target": {
29            "key": "9b2e4c0281c0b02e7c53291a94d1d0cbff8883f8024f5142ee494ffbbd088071"
30          }
31        }
32      ],
33      "extra": [ 1, 119, 103, 170, 252, 222, 155, 224, 13, 207, 208, 152, 113, 94, 188,
34      247, 244, 16, 218, 235, 197, 130, 253, 166, 157, 36, 162, 142, 157, 11, 200, 144,
35      209
36      ],
37      "signatures": [ ]
38    },
39    "tx_hashes": [ ]
40  }
```

Componentes do bloco de génese

Como foi usado o mesmo software para imprimir o bloco de génese e o bloco do apêndice B, a estrutura parece ser básicamente a mesma. São dadas algumas diferenças.

- `difficulty` (linha 9) - A dificuldade é 1, o que significa que qualquer nonce serve.
- `timestamp` (linha 14) - O bloco de génese não tem um carimbo no tempo significante.
- `prev_id` (linha 15) - Usam-se 32 bytes a zero para o ID anterior, por convenção.
- `nonce` (linha 16) - $n = 10000$ por convenção.
- `amount` (linha 27) - Isto corresponde exactamente ao cálculo para o primeiro subsídio de bloco (17.592186044415 xmr) da secção 7.3.1.
- `key` (linha 29) - Os primeiros monerοj foram dispersados para o fundador de Monero thankful_for_today.
- `extra` (linhas 33-36) - Usa-se a codificação discutida no apêndice o campo `extra` da transacção de mineiro, só contêm uma chave pública de transacção.

- `signatures` (linha 37) - Não existem assinaturas no bloco de génese. Isto aparece aqui devido á função `print_block`. O mesmo é verdade para `tx_hashes` na linha 39.

APÊNDICE D

Uma nota sobre notações

Neste apêndice explica-se a notação dentro de bulletproofs. Existem pois duas operações básicas $a + b = c$ é sobre os escalares em F_p, e $A + B = C$ é sobre todos os pontos de CE disponíveis em ed25519.

sobre os pontos de CE :
$a\underline{B} = \{aB_1, aB_2, aB_3, ..., aB_n\}$
$\underline{a} \circ \underline{B} = \{a_1B_1, a_2B_2, a_3B_3, ..., a_nB_n\}$
$\langle \underline{a}, \underline{B} \rangle = \sum_{j=1}^{n} a_j B_j$
existem optimizações específicas para o seguinte :
$\langle \underline{a}, \underline{B} \rangle + \langle \underline{b}, \underline{C} \rangle + \langle \underline{c}, \underline{D} \rangle ...$
para mais informações sobre algoritmos de factorização sobre pontos de CE (Pippenger et al.) veja-se[].

sobre os escalares :
$a\underline{b} = \{ab_1, ab_2, ab_3, ..., ab_n\} = a\underline{1} \circ \underline{b}$
$a\underline{1} = \underline{a}$
$\underline{a}^n = \{a^0, a^1, a^2, ...a^{n-1}\}$
$\underline{a} \circ \underline{b} = \{a_1b_1, a_2b_2, a_3b_3, ..., a_nb_n\}$
$\langle \underline{a}, \underline{b} \rangle = \sum_{j=1}^{n} a_j b_j$
$\langle \underline{a}, \underline{b} \rangle = \langle \underline{1}, \underline{a} \circ \underline{b} \rangle$
$\langle \underline{a}, \underline{b} \circ \underline{c} \rangle = \langle \underline{a} \circ \underline{c}, \underline{b} \rangle$
$\langle \underline{a}, \underline{b} \rangle + \langle \underline{a}, \underline{c} \rangle = \langle \underline{a}, \underline{b} + \underline{c} \rangle$

para extraír $\delta(y,z)$ de :

$$\begin{aligned} vz^2 &= \langle \underline{a}_L, \underline{2}^n \rangle z^2 \\ 0z &= \langle \underline{a}_L - \underline{1} - \underline{a}_R, \underline{y}^n \rangle z \\ 0 &= \langle \underline{a}_L \circ \underline{a}_R, \underline{y}^n \rangle \end{aligned}$$

$$vz^2 + 0 + 0 = \langle \underline{a}_L, \underline{2}^n \rangle z^2 + \langle \underline{a}_L - \underline{1} - \underline{a}_R, \underline{y}^n \rangle z + \langle \underline{a}_L \circ \underline{a}_R, \underline{y}^n \rangle$$

$$vz^2 = \langle \underline{a}_L, \underline{2}^n \rangle z^2 + \langle \underline{a}_L, \underline{y}^n \rangle z + \langle -\underline{1}, \underline{y}^n \rangle z + \langle -\underline{a}_R, \underline{y}^n \rangle z + \langle \underline{a}_L, \underline{a}_R \circ \underline{y}^n \rangle$$

$$vz^2 + \langle \underline{1}, \underline{y}^n \rangle z = \langle \underline{a}_L, \underline{2}^n \rangle z^2 + \langle \underline{a}_L, \underline{y}^n \rangle z + \langle -\underline{1}, \underline{a}_R \circ \underline{y}^n \rangle z + \langle \underline{a}_L, \underline{a}_R \circ \underline{y}^n \rangle$$

$$vz^2 + \langle \underline{1}, \underline{y}^n \rangle z = \langle \underline{a}_L, \underline{2}^n z^2 + \underline{y}^n z + \underline{a}_R \circ \underline{y}^n \rangle + \langle -\underline{1}, \underline{a}_R \circ \underline{y}^n \rangle z$$

$$\begin{aligned} vz^2 + \langle \underline{1}, \underline{y}^n \rangle z - \langle \underline{1}, \underline{2}^n z^2 + \underline{y}^n z \rangle z &= \langle -\underline{1}, \underline{2}^n z^2 + \underline{y}^n z \rangle z \\ &+ \langle \underline{a}_L, \underline{2}^n z^2 + \underline{y}^n z + \underline{a}_R \circ \underline{y}^n \rangle \\ &+ \langle -\underline{1}, \underline{a}_R \circ \underline{y}^n \rangle z \end{aligned}$$

$$\begin{aligned} vz^2 + \langle \underline{1}, \underline{y}^n \rangle z - \langle \underline{1}, \underline{y}^n z \rangle z - \langle \underline{1}, \underline{2}^n z^2 \rangle z &= \langle -z\underline{1}, \underline{2}^n z^2 + \underline{y}^n z + \underline{a}_R \circ \underline{y}^n \rangle \\ &+ \langle \underline{a}_L, \underline{2}^n z^2 + \underline{y}^n z + \underline{a}_R \circ \underline{y}^n \rangle \end{aligned}$$

$$vz^2 + (z - z^2)\langle \underline{1}, \underline{y}^n \rangle - z^3 \langle \underline{1}, \underline{2}^n \rangle = \langle \underline{a}_L - z\underline{1}, \underline{2}^n z^2 + \underline{y}^n z + \underline{a}_R \circ \underline{y}^n \rangle$$

$$vz^2 + \underbrace{(z - z^2)\langle \underline{1}, \underline{y}^n \rangle - z^3 \langle \underline{1}, \underline{2}^n \rangle}_{\delta(y,z)} = \langle \underline{a}_L - z\underline{1}, \underline{y}^n \circ (\underline{1}z + \underline{a}_R) + \underline{2}^n z^2 \rangle$$