Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Esquemas de privacidade nas moedas criptográficas Monero e Zcash

André Souza Abreu

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Routo Terada

São Paulo
2023

Dedico este trabalho à todos aqueles que buscam compreender assuntos complexos, tais como os sistemas criptográficos aqui apresentados, e que procuram verificar a veracidade das informações apresentadas.

Agradecimentos

We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called "abstraction"; as a result the effective exploitation of the powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worth-while to point out that the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.
— Edsger Dijkstra, The Humble Programmer

Trusted Third Parties are Security Holes
— Nick Szabo, 2001

Agradeço à Satoshi Nakamoto por ter descoberto um mecanismo de verificação de registro históricos sem necessidade de confiança. Agradeço à Nicolas Van Saberhagen por ter descoberto como aperfeiçoar a privacidade neste mecanismo. Agradeço à ambos por terem permanecidos anônimos, humildemente atestando que suas descobertas são mais importantes que os próprios autores.

Resumo

Nos últimos anos, temos presenciado o surgimento de inúmeras moedas digitais baseadas em protocolos criptográficos. Algumas dessas moedas digitais surgiram com a proposta de permitir transações monetárias com alto nível de privacidade, sem revelar informações confidenciais ao mesmo tempo que mantém um registro de dados distribuído, um sistema verificável, e sem necessidade de confiança em terceiros. Neste quesito, duas moedas criptográficas têm se destacado: *Monero e Zcash. O objetivo deste trabalho é expor e explicar os mecanismos pelos quais estas duas moedas proporcionam maior privacidade aos seus usuários: os tipos de privacidade adquirida, o grau de privacidade proporcionado, o funcionamento destes sistemas do ponto de vista criptográfico-computacional, suas vantagens e trade-offs.*

Palavras-chave: *criptografia. monero. zcash. privacidade.*

Abstract

In the past few years, we have seen the raise of numerous digital currencies based on criptographic protocols. Some of these came with the purpose of providing the users with high levels of privacy, allowing monetary transactions to take place without revealing users' confidential information while simultaneously relying on public data distributed and audited accross a P2P network, all without trusted third-parties. In this regard, two criptographic currencies have stood out: *Monero* and *Zcash*. The goal of this paper is to present and explain the mechanisms by which these two currencies provide privacy to their users: the type of privacy acquired, the degree of privacy obtained, the functioning of the system from a cryptographic perspective, and the trade-offs.

Keywords: criptography. monero. zcash. privacy.

Lista de abreviaturas

DLP	*Discrete Logarithm Problem* Problema do Logaritmo Discreto
ECDLP	*Elliptic Curve Discrete Logarithm Problem* Problema do Logaritmo Discreto em Curvas Elípticas
EC	*Elliptic Curve* Curva Elíptica
MAC	*Message Authentication Code* Código de Autenticação de Mensagem
P2P	*Peer-To-Peer* Par-a-Par
PoW	*Proof Of Work* Prova de Trabalho
zk-SNARKs	*zero-knowledge Succinct Non-Interactive Arguments of Knowledge system* Sistema zero-conhecimento de Argumentos de Conhecimento Sucintos Não-Interativos

Lista de símbolos

Símbolos usados nas seções sobre Monero:

α	nonce
$\mathcal{F}_q$	campo finito sobre o primo q
G	gerador público da EC
$\mathcal{H}$	função de hash
$\mathcal{H}_n$	função de hash no campo $\mathcal{F}_q$
$\mathcal{H}_p$	função de hash para ponto na curva
k	chave privada
K	chave pública
k_s	chave privada de gasto (*spending key*)
K_s	chave pública de gasto (*spending key*)
k_v	chave privada de visualização (*view key*)
K_v	chave pública de visualização (*view key*)
l	inteiro primo, ordem do subgrupo de G
π	índice secreto
r	número aleatório
$\mathcal{R}$	anel (conjunto de chaves públicas)
σ	assinatura digital criptográfica

Símbolos usados nas seções sobre Zcash:

`rt`	raiz da Árvore de Merkle
`sn`	*serial number*
`cm`	*coin commitment*
`np`	*note plaintext*
`nf`	*nulifier*
`pp`	parâmetros públicos
`PRF`	*Pseudo-Random Function*
`CRH`	*Collision-Resistant Hashing*
`COM`	*Commitment Scheme*
`KDF`	*Key Derivation Function*
π	prova de zero-conhecimento
σ	assinatura digital
C_i	texto cifrado
h_i	*tag, MAC*
L_C	Linguagem do circuito aritmético C

Sumário

Introdução

Com o avanço da internet e dos meios de comunicação, tornou-se viável o comércio online de produtos e serviços. O pagamento destes bens no meio digital apresenta um problema: não é possível transferir digitalmente bens físicos de valor, tais como dinheiro físico ou outros meios de troca. O que pode ser transferido digitalmente são bens digitais, tais como arquivos e registros financeiros, mas estes podem ser replicados facilmente por computadores, desvalorizando-os devido à abundância de cópias.

A forma convencional de resolver esse problema é utilizar algum intermediário, geralmente bancos ou processadores de pagamentos – PayPal, PagSeguro, PicPay, Visa, MasterCard. O pagador solicita transferir uma quantia para o recipiente; a solicitação é enviada para o intermediário, o qual verifica se o pagador tem o saldo a ser transferido e, se sim, transfere o valor da conta do pagador para o recebedor; se não, o pagamento não ocorre. Dessa forma, o intermediário mantém um registro de quanto dinheiro cada usuário possui, teoricamente impedindo duas fraudes: (1) gasto de dinheiro alheio e (2) criação arbitrária de dinheiro por meio da replica de arquivos digitais.

Esta forma convencional de comércio eletrônico requer que as partes transferindo valor confiem que o intermediário aja honestamente. É também comum que este intermediário, seja banco ou processador de pagamentos, exija uma taxa de serviço e informações pessoais sobre as duas partes envolvidas, comprometendo em certo grau a privacidade e anonimato delas. Ademais, o intermediário pode não efetuar a transferência de valor caso houver alguma ordem judicial, ou até mesmo por uma política interna própria, impedindo a transferência de valor devido à, por exemplo, limite de valor, produto ou serviço banalizado socialmente, data e horário da transferência, e perseguição política de uma das partes.

Diante disso, uma entidade desconhecida sob o pseudônimo Satoshi Nakamoto publicou um artigo teórico e a implementação em código do *Bitcoin*, um sistema de pagamentos digital Par-a-Par (P2P) sem necessidade de terceiros de confiança [Nak08]. Este sistema consiste de uma rede P2P em que cada nó mantém uma cópia de um registro de dados distribuídos, em que as transferências de valor são certificadas por meio de provas criptográficas utilizando criptografia assimétrica, e a validade do histórico de transações é garantida por meio de *Prova-de-Trabalho Acumulada*, um mecanismo no qual é computacionalmente caro criar um registro com uma ordenação cronológica válida das transações, mas é computacionalmente trivial verificar a validade de tais registros.

O sistema Bitcoin utiliza um esquema de criptografia assimétrica para verificar a transferência de valor. Não há associação intrínseca entre o detentor de um par de chaves pública-privada, a qual é usada na prova criptográfica, e sua identidade no mundo real. Essa

associação só existirá se o detentor revelar publicamente que é o dono de tal par de chaves, ou cometer algum deslize que o revele. Portanto, os detentores dos saldos de Bitcoin são anônimos, desde que não se revelem por conta própria ou por descuido. Entretanto, dados de todas as transferências são públicos: a origem, o destino, e o valor. Logo, apesar de haver anonimato (identidade desconhecida), não há nenhuma privacidade nas transações.

Inúmeras ideias foram propostas para reduzir a ausência de privacidade no Bitcoin. Dentre elas, se destacam *Monero* e *Zcash*. Estas duas moedas criptográficas utilizam tecnologias semelhantes ao Bitcoin (rede P2P, registro de dados distribuídos, criptografia assimétrica, etc) ao passo que mantém a privacidade nas transações por meio de uma série de artifícios criptográficos.

O objetivo deste trabalho é, portanto, expor e explicar o funcionamento dos componentes criptográficos utilizados por Monero e Zcash e demonstrar como eles garantem a privacidade dos usuários em um sistema cujo registro de dados é público e distribuído. O trabalho é divido em quatro partes: a primeira parte discorre brevemente sobre as tecnologias de moedas criptográficas, a segunda entra em detalhes nos esquemas de privacidade utilizados em Monero; a terceira nos detalhes de Zcash; e a quarta parte analisa suas características, diferenças e *tradeoffs* entre os dois sistemas.

Capítulo 1

Moedas Criptográficas

Esta seção trata do funcionamento de moedas criptográficas. Os conceitos apresentados aqui tem como base o modelo do Bitcoin e são também utilizados nas moedas Monero e Zcash. Entendê-los é necessário para compreender tópicos mais avançados apresentados posteriormente.

O funcionamento de moedas criptográficas será explorado de modo abstrato, sem especificar implementações particulares de cada componente do sistema. Tais detalhes são apresentados nos capítulos posteriores, que aprofundam nas operações criptográficas específicas.

1.1 Curvas Elípticas

As *Curvas Elípticas (EC)* são amplamente usadas em moedas criptográficas, constituindo-se *pilares de fundação*[1] destes sistemas. São empregadas na geração de endereços de pagamentos, em protocolos de assinatura digital assimétrica e em esquemas de *commitments*. Ambos Bitcoin, Monero e Zcash fazem uso delas.

Em criptografia, uma EC é definida sobre um *campo finito*. Um campo finito $\mathbb{F}_q$ [SP18, p. 272] é o conjunto $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ no qual operações de adição, multiplicação, e negação são aplicadas sob o módulo q, onde $q > 3$ é primo. Uma EC é então definida [SP18, p. 278] como o conjunto de pares (x, y) que satisfazem a equação:

$$y^2 = x^3 + ax + b \qquad a, b, x, y \in \mathbb{F}_q$$

A adição de dois pontos distintos $P_1 = (x_1, y_2)$ e $P_2 = (x_2, y_2)$ da EC resulta em um terceiro ponto P_3 cujas coordenadas são:

[1] Também chamados *cryptographic building blocks* na literatura acadêmica.

$$\lambda = \frac{x_2 - x_1}{y_2 - y_1}$$
$$x_3 = \lambda^2 - x_2 - x_1$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

Se P_1 e P_2 são o mesmo ponto, então:

$$\lambda = \frac{3x^2 + a}{2y}$$

E as definições de x_3 e y_3 continuam iguais.

Em uma EC, o ponto $\mathcal{O}$ tal que $P + \mathcal{O} = P$, para todo ponto P da curva, é chamado de *Ponto de Afinidade*. É o *elemento nulo* da adição em curvas elípticas.

A negação de um ponto P é o ponto $-P$ tal que:

$$P + (-P) = \mathcal{O}$$

A multiplicação de um ponto P da curva por uma constante $k \in \mathbb{Z}$ pode ser definido por recursão:

$$kP = P + (k - 1)P$$

Se k é negativo, pode-se calcular $kP = |k|(-P)$.

O subgrupo de um ponto P é o conjunto de pontos que podem ser gerados multiplicando P por uma constante k. Para todo ponto P, há um certo valor k_p tal que $k_pP = \mathcal{O}$. Isto significa que, após adicionar P mais de k_p vezes consecutivas, os pontos obtidos pela adição irão se repetir, pois $(k_p + k)P = k_pP + kP = kP$.

A ordem de um subgrupo é definida como o número de elementos neste subgrupo. Além disso, cada Curva Elíptica tem uma ordem N, correspondente ao número total de pontos nesta curva. A ordem de cada subgrupo gerado por pontos da curva são divisores de N (teorema de Lagrange). Se N é primo, haverá apenas um subgrupo, correspondente ao próprio conjunto de todos os pontos.

Para encontrar a ordem u do subgrupo de um ponto P da curva, faça:

1. Compute N
2. Compute os divisores de N (fatoração)
3. Para cada divisor d de N, compute $d \times P$
4. O menor d tal que $d \times P = 0$ é a ordem u do subgrupo.

Geralmente, curvas elípticas selecionadas para operações criptográficas são tais que sua ordem $N = hl$, onde l é um primo suficientemente grande para tornar as operações criptograficamente seguras, e h é chamado de *cofator*, que pode ser um valor pequeno tal como 2 ou 3.

É comum escolher um elemento gerador G do subgrupo de ordem l para ser um ponto fixo a ser usado em várias operações. Este elemento é chamado de *Gerador* do grupo porque todos os outros elementos do subgrupo podem ser obtidos a partir de adições sucessivas de G à sim mesmo. Comumente, há outros geradores além de G, mas ele é selecionado para ser um valor público. Ressalta-se a segurança das operações não é comprometido por G ser público, pois esta segurança é derivada a partir de outros fatores.

Em particular, a segurança de muitas operações criptográficas em Curvas Elípticas baseia-se na *segurança computacional* do seguinte problema: encontrar um inteiro k tal que $K = kG$ sabendo apenas os valores dos pontos K e G. Este problema é chamado de *Problema do Logaritmo Discreto em Curvas Elípticas*, por ser análogo ao *Problema do Logaritmo Discreto* em aritmética modular.

Um uso prático destas propriedades é criptografia assimétrica: o valor k sendo a chave pública e K sendo a chave privada. Eventualmente, será mostrado como k e K podem ser usados para proporcionar privacidade por meio de assinaturas digitais, provas não interativas de *zero conhecimento*, e *commitments*.

1.2 Transações

A criptografia elucidada na seção 1.1 é base para que transferências de valor no meio digital possam ter sua legitimidade verificada sem a necessidade de instituições financeiras e terceiros de confiança. No Bitcoin e nas moedas criptográficas a serem estudadas neste trabalho, a legitimidade das transações é comumente realizada por meio de assinaturas digitais em Curvas Elípticas.

Nas próprias palavras[2]de Satoshi Nakamoto:

> *Definimos dinheiro digital como uma sequência de assinaturas digitais. Cada proprietário [do dinheiro] transfere a moeda para o próximo proprietário ao assinar o hash de uma transação anterior com a chave pública do próximo proprietário, adicionando estes valores no final da transação. O recipiente pode verificar as assinaturas para averiguar o histórico de posse do dinheiro [Nak08, p. 2].*

Neste sistema, uma transação é a transferência de valor autenticada por uma assinatura digital do proprietário atual do valor para um endereço vinculado à chave pública do receptor da quantia. O endereço do receptor é o único identificador do destinatário da transação[3]. O novo proprietário deve repetir o procedimento: fornecer uma assinatura digital que ateste que ele é o legítimo possuidor do valor. Esta assinatura digital é gerada com a chave privada correspondente à chave pública do endereço que contém o saldo.

[2] Traduzidas para o português.

[3] No Bitcoin, o endereço é o hash de uma chave pública do recipiente. No Zcash o endereço é a chave pública

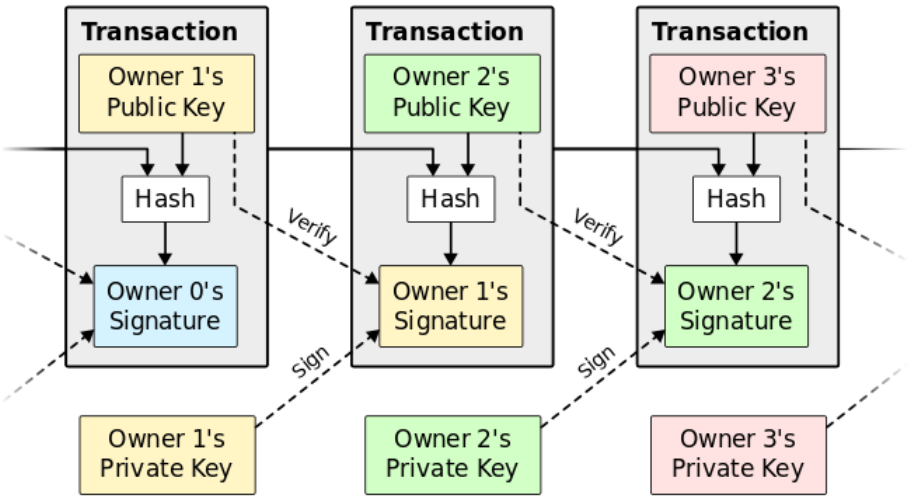

Figura 1.1: *Cadeia de Transações*
Créditos: Bitcoin Whitepaper

1.3 Rede P2P

As transações assinadas digitalmente são agrupadas em blocos. Estes blocos são transmitidos via *broadcast* em uma rede P2P, para que todos os membros possam verificar sua validade. Um nó qualquer não necessita de se conectar à todos os outros nós da rede, mas apenas à alguns, os quais irão repassar informações sobre blocos válidos e sobre outros nós da rede

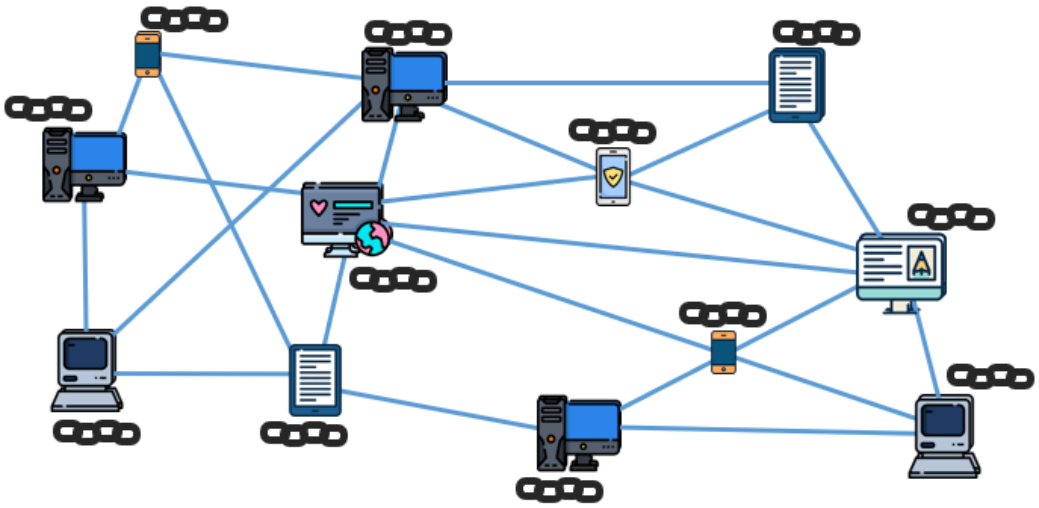

Figura 1.2: *Rede P2P*
Créditos: https://linkedin.com/pulse/blockchain-distributed-p2p-network-saurabh-virdi

Ademais, um nó não precisa confiar nas informações fornecidas por outro nó, pois ele pode verificar por si mesmo a validade das assinaturas digitais dentro de um bloco e pode

do recipiente, mas está encriptada com outros dados. No Monero, o endereço é uma chave pública de uso único.

formar conexões com outros nós para coletar mais informações, descartando aquelas que provarem serem falsas e mantendo as que verificar serem verdadeiras.

A arquitetura P2P é usada com o propósito de evitar centralização do protocolo, pois do contrário haveriam alguns nós com a habilidade de decidir quais transações são válidas. O fato de qualquer nó da rede poder verificar e validar as transações contribui para a manutenção de um sistema decentralizado sem a necessidade de confiança em entidades terceiras.

1.4 Gasto Duplo

O esquema de transações explicado na seção 1.2, conjuntamente com a rede P2P, resolve apenas parte do problema de criar um dinheiro digital descentralizado. Conforme Satoshi elucidou:

> *O problema [com o esquema de transações], obviamente, é que o recebedor não pode verificar que um dos proprietários passados não efetuou um gasto duplo do dinheiro. Uma solução comum é introduzir uma autoridade central, que checa toda transação para prevenir gasto duplo. Após efetuar uma transação, o dinheiro do remetente é enviado para esta entidade, que emite o dinheiro para o destinatário. Apenas valores emitidos diretamente pela entidade são considerados como não tendo sido gastos mais de uma vez.*
>
> *O problema com esta solução é que a credibilidade de todo o sistema monetário depende desta autoridade, com todas as transações passando por ela, tal como um banco. Precisamos de um modo para que o recebedor do dinheiro certifique-se de que proprietários passados não assinaram outras transações as quais transferiram valores para outros [Nak08, p. 2].*

O que Satoshi refere como *gasto duplo* é a situação em que o proprietário de uma quantia de dinheiro tenta gastar o mesmo dinheiro várias vezes. No sistema analisado aqui, isto seria feito ao gerar várias assinaturas digitais diferentes que transferem este dinheiro para diferentes destinatários. Assim, o proprietário de uma quantia X de dinheiro pode, por exemplo, transferir X para 10 comerciantes diferentes, tentando engana-los; caso obtenha sucesso, esta pessoa efetivamente fraudou uma quantia de $9X$, gastando seu dinheiro mais de uma vez. Daí o nome "gasto duplo".

O *gasto duplo* é um problema no meio digital porque arquivos e informações podem ser duplicados e gerados com custo mínimo, mas não no mundo físico porque itens físicos de valor não podem ser "duplicados"(produzidos, confeccionados) sem custo adicional considerável e ao mesmo tempo evitar sua desvalorização.

1.5 Problema dos Generais Bizantinos

O problema do gasto-duplo é uma instância do *Problema dos Generais Bizantinos*. Este termo foi cunhado em 1982 por *Lamport et al* em um trabalho relacionado à confiabilidade de processos em um sistema distribuído [LSP82]. Trata-se de um problema prático que

ocorre em redes distribuídas, que pode ser resumida pela analogia a seguir (usada inclusive pelos próprios autores[4]):

Vários generais do exército Bizantino estão sitiando uma cidade; cada um deles comanda uma tropa. Eles só terão sucesso em sua missão se combinarem de atacar ao mesmo tempo. A única forma de comunicação que eles possuem é por meio de cartas entregue por mensageiros. Porém, há a possibilidade de generais e mensageiros traidores, os quais desejam impedir um curso coordenado de ação. Um procedimento é necessário para garantir que um grupo de traidores não engane os outros generais, que devem chegar à um consenso sobre um curso de ação.

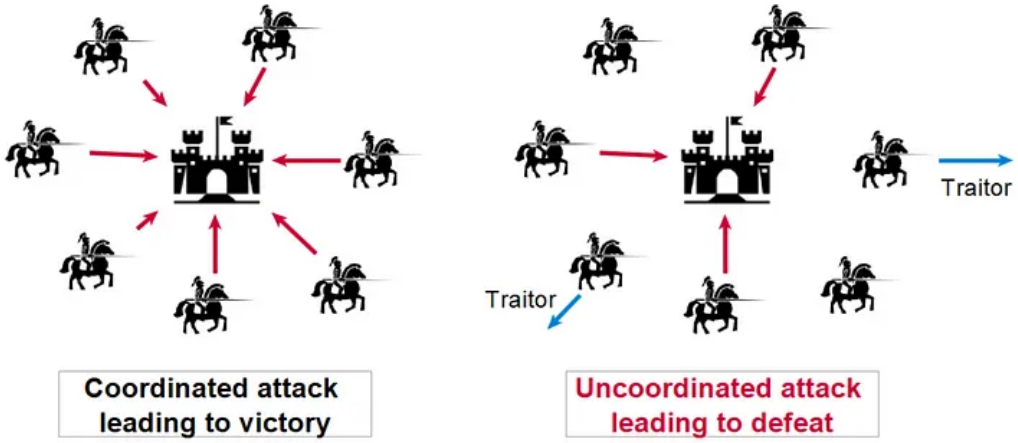

Figura 1.3: *Problema dos Generais Bizantinos*
Créditos: https://medium.com/swlh/bitcoins-proof-of-work-the-problem-of-the-byzantine-generals-33dc4540442

Este problema é relevante para a computação por tratar da questão de consenso e sincronia de informação entre computadores de uma rede distribuída. É ainda mais relevante quando se trata de uma rede de pares, não controlada por uma autoridade central (por exemplo, uma central que controla computadores distribuídas em vários *data centers*), mas composta por nós igualmente legítimos a princípio.

Deve-se imaginar o seguinte: um nó, ao participar de uma rede, deve-se conectar em outros nós. Após isso, passa a trocar informações com eles, incluindo informações sobre a própria rede. Como este nó irá saber que os outros nós não estão repassando informação falsa ou maliciosa? Se a informação assumida como correta for aquela fornecida pelo maior número de nós conectados, então um ator malicioso pode executar um ataque que faça o nó honesto se conectar majoritariamente à nós maliciosos controlados pelo atacante, enganando assim o nó honesto. A solução para este dilema é explorada a seguir.

1.6 Prova De Trabalho

Antes de introduzir a solução descoberta por Satoshi para resolver o Problema dos Generais Bizantinos no contexto de moeda digital, é necessário entender o conceito de

[4] Lamport disse que a analogia teve inspiração no problema dos filósofos pensantes de Dijkstra, problema clássico em temas como escalonamento de processos e paralelismo[LSP82].

Prova de Trabalho (do inglês, *Proof Of Work, PoW*). Este conceito é o pilar do *HashCash*, sistema que inspirou o próprio Satoshi no design do Bitcoin.

O *HashCash* foi proposto como um mecanismo de mitigar *spams* e Ataques de Negação de Serviço (*Denial Of Service, DoS*) [Bac02]. O esquema propõe que em situações de alta demanda de servidores, os clientes de serviços de internet deveriam prover um *token*[5] derivado à partir de funções custosas, de modo que a obtenção do token exija um determinado custo computacional, mas a verificação da validade do token seja trivial.

Formalmente, o esquema do HashCash pode ser resumido como:

1. Cliente requisita um serviço `s`.
2. Servidor cria um desafio $C \leftarrow \text{CHALLENGE}(s, w)$ com expectativa de ser solucionado com um trabalho médio `w`.
3. Servidor envia o desafio para o cliente.
4. Cliente computa uma possível solução para o problema, gerando um token correspondente: $\tau \leftarrow \text{MINT}(C)$
5. Cliente envia o token para o servidor.
6. Servidor valida o token do cliente $V \leftarrow \text{VERIFY}(\tau)$
7. Se o token for válido, o servidor atende a requisição do cliente.

Uma implementação deste algoritmo, definida no próprio *HashCash*, é a seguinte:

Seja $s = \{0,1\}^*$ uma *bitstring*: uma string constituída apenas de 0s e 1s. Denote por $[s]_i$ o i-ésimo bit na posição de s, começando pelo bit mais à esquerda. Assim, s_1 é o bit mais à esquerda e $s_{|s|}$ é o bit mais a direita. Seja $=_b$ um operador de comparação de infixo entre duas strings x e y tal que os b primeiros *bits*, à esquerda, sejam iguais em ambas:

$$x =_b y \iff x_i = y_i \quad \forall_{i=1,2,\ldots,b}$$

As função de custo e verificação do *Hashcash* são definidas como:

- $\text{MINT}(s, w)$: encontrar $x \in \{0,1\}^*$ tal que $\mathcal{H}(s||x) =_w 0^w$, retornando x como token τ.
- $\text{VERIFY}(\tau)$: verificar se $\mathcal{H}(s||x) =_w 0^w$, onde x é o próprio token τ.

Em resumo, o *mint* consiste em encontrar um valor x que ao ser concatenado à string s, resulta em um hash cujos w primeiros *bits* são iguais à zero. A função de hash deve ser criptograficamente segura para garantir que não haja algoritmos que facilitem o cálculo de x. Assim, a única forma de resolver o desafio é tentar adivinhar x testando vários valores.

Sendo o hash criptograficamente seguro, encontrar x demandará realizar um número médio de computações que cresce exponencialmente com w. Ademais, o valor de w não é necessariamente fixo, podendo ser aumentando caso o servidor esteja sofrendo abuso de *spams* ou ataques DoS, e diminuído conforme a demanda pelos seus serviços diminua.

[5] Para os propósitos desta seção, um *token* pode ser entendido como uma *string* com propriedades criptográficas usada para autenticar ou autorizar um usuário do sistema

A Prova De Trabalho não se restringe ao *HashCash* e suas aplicações, o qual foi ilustrado aqui por duas razões: (1) ele provê soluções para problemas reais, (2) é o sistema que fundamentou Bitcoin e outras moedas criptográficas.

1.7 Prova De Trabalho Acumulada

Para eliminar a necessidade de confiança e resolver o gasto duplo, Bitcoin usa um sistema baseado em Prova De Trabalho.

Essencialmente, a solução é que cada bloco de transações contenha o hash do bloco anterior (chamaremos de `prevHash`) e um `nonce`. O `nonce` deve ser tal o hash do bloco obtido (que inclui o `nonce`, as transações, e outros metadados) satisfaz um desafio de Prova De Trabalho semelhante ao do *HashCash*: começar um determinado número de zeros, chamado `target`.

Assim, os blocos formam uma estrutura de dados do tipo Lista Ligada, na qual o *link* é uma referência para o hash do bloco anterior, e na qual cada item da lista deve satisfazer a Prova De Trabalho. A lista sequencial de blocos gera uma ordem cronológica de transações, as quais permitem evitar gasto duplo ao considerar legítima apenas a primeira vez em que um saldo é gasto, descartando-se a segunda vez.

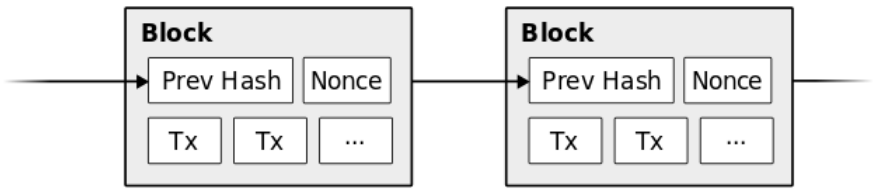

Figura 1.4: *Cadeia de blocos*
Créditos: Bitcoin Whitepaper

Um outro efeito desta lista ligada é que não é possível corromper um bloco passado sem corromper todos os blocos futuros, pois uma ligeira modificação em um bloco mudará o valor de seu hash criptográfico, fazendo com que o próximo bloco tenha uma referência inválida. Assim sendo, é necessário atualizar o campo `prevHash` do próximo bloco, alterando o hash deste mesmo. Isso gera um efeito em cascata, em que é necessário recalcular os hashes dos próximos blocos para que as referências sejam válidas.

Refazer os hashes da lista ligada é uma tarefa fácil para um computador pessoal moderno, o qual consegue computar milhões de hashes por segundo (ver apêndice A). No momento de escrita deste trabalho, Bitcoin tem pouco mais de 800 mil blocos. Se Bitcoin não usasse PoW, um adversário poderia adulterar uma transação de 10 anos atrás para efetuar *gasto duplo* e recomputar os hashes da lista ligada em menos de 1 segundo usando apenas um laptop comum. A sequência de transações adulterada seria válida e não haveria critério objetivo para distingui-la da sequência original, pois as provas criptográficas de ambas seriam válidas.

Por outro lado, com a Prova de Trabalho não basta o adversário recalcular o hash:

o hash do bloco deve satisfazer o requerimento especificado pela `target`, que significa gastar poder computacional para achar algum `nonce` adequado. Desta forma, a Prova De Trabalho individual de cada bloco é acumulada, formando uma Prova De Trabalho Acumulada. É esta Prova De Trabalho Acumulada que previne a corrupção dos blocos, pois é necessário refazer o trabalho todo.

Assim sendo, o *PoW* é um critério objetivo e definitivo: a sequência de transações final é aquela com maior Prova de Trabalho Acumulada. É uma regra do sistema Bitcoin que a sequência de blocos que é considerada definitiva por um nó será a sequência que tiver a maior Prova De Trabalho Acumulada, obtida somando-se o trabalho `w` feito em cada bloco baseado numa `target` dinâmica.

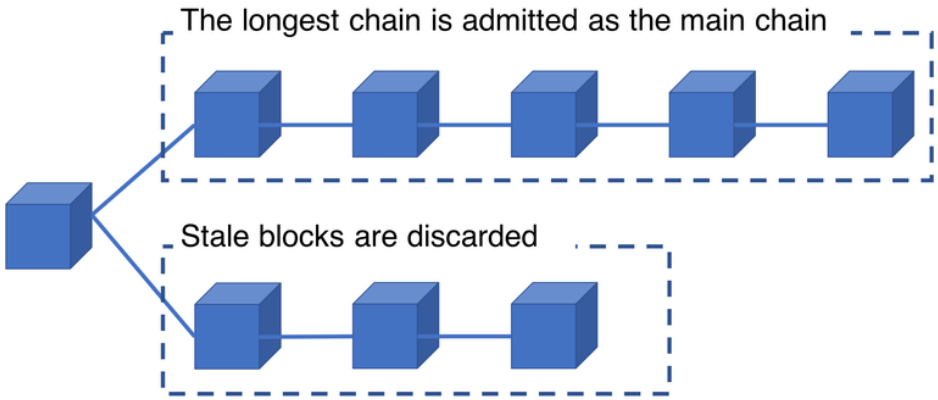

Figura 1.5: *Sequência com maior Prova de Trabalho prevalece*
Créditos: https://www.researchgate.net/figure/fig4_335705491

Um adversário pode tentar refazer todo o trabalho e adulterar os blocos passados, mas enquanto ele gasta tempo e recursos computacionais calculando a Prova De Trabalho dos blocos adulterados, os outros nós da rede estão gerando novos blocos, os quais o adversário terá que adulterar também (para que tenham referência aos blocos adulterados) e refazer o PoW destes. Por outro lado, uma lista de blocos ligada por hash não possui nenhuma segurança contra um adversário se ela não tiver PoW. É justamente a Prova De Trabalho, elemento central do sistema, que garante a incorruptibilidade dos blocos; do contrário, o sistema é facilmente adulterado e torna-se inútil.

Se o adversário conseguir produzir blocos válidos mais rápido do que todos os outros nós da rede, então eventualmente ele irá conseguir adulterar toda uma sequência de blocos. Para isto, é necessário que o adversário possua um poder computacional maior que o resto da rede, o que equivale a pelo menos 51% do poder computacional de todos os nós da rede. Por isso mesmo, este tipo de ataque é chamado de *ataque de 51%*.

Caso contrário, a probabilidade do adversário adulterar os blocos decresce exponencialmente conforme mais blocos são minerados e aumentam o trabalho necessário para adulterar todos os blocos e gerar uma sequência com a maior Prova De Trabalho Acumulada.

Aqui é importante ressaltar que, caso este evento de fato ocorra, o estrago causado pelo adversário é efetuar *gasto duplo*: ele pode reverter uma transação na qual envia dinheiro para alguém, substituindo esta transação por outra transação na qual o dinheiro é enviado

para si mesmo. Um *ataque de 51%* bem sucedido não é capaz de gastar dinheiro alheio, pois gerar uma transação válida requer a chave privada necessária para uma assinatura digital legítima. Este tipo de ataque também não é capaz de criar dinheiro do nada, pois isto é contra as regras do protocolo e um bloco que tente fazer isto será descartado pelos nós da rede.

Para ilustrar a robustez de sistemas cujo consenso entre nós é baseado em Prova de Trabalho Acumulada, considere o exemplo: se um nó honesto X se conectar à 999 nós maliciosos (informando um consenso falso) e a 1 nó honesto (informando o consenso verdadeiro), o nó X irá conseguir verificar matematicamente que este 1 nó honesto está informando a verdade e os outros 999 nós estão errados sobre o consenso da rede. Os nós maliciosos podem transmitir informações erradas para outros nós da rede, mas eles não podem alterar o consenso de um sistema de Prova de Trabalho Acumulada sem refazer todo o trabalho acumulado, o que significa gastar poder computacional para reverter o consenso atual.

A Prova de Trabalho Acumulada, apresentada por Satoshi e usada no Bitcoin, é a primeira solução (e a única conhecida até hoje) para o problema dos Generais Bizantinos que **não** envolve esquemas de confiança, baseando-se inteiramente em provas matemáticas verificáveis.

Existem outras formas de resolver o Problema dos Generais Bizantinos, as quais baseiam-se em sistemas com certo grau confiança. Em alguns destes sistemas, confia-se que uma ou mais entidades são fontes de informações verídicas inquestionáveis. Em outros sistemas, admite-se que algumas entidades podem ser comprometidas, mas assume-se que não é provável que um atacante consiga comprometer um número suficiente de entidades. A confiabilidade destes sistemas está na expectativa que a maioria de um grupo, ou *quorum*, não será comprometida. Um exemplo concreto de tal sistema é descrito em *Design of a Secure Timestamp Server with minimal requirements* [QSM99].

1.8 Incentivos

O Bitcoin, bem como outras moedas criptográficas, utiliza uma recompensa financeira para incentivar comportamento honesto, conforme as regras do protocolo. Esta recompensa consiste em uma quantia monetária obtida ao se produzir um bloco válido.

A produção do bloco é também chamada de *mineração* como uma analogia ao ato de minerar ouro, pois produzir o bloco gera moedas da mesma forma que o ouro minerado pode gerar moedas valiosas. Nesta mesma lógica, os produtores do bloco são chamados de *mineradores*, os quais tentam *minerar* um bloco.

A primeira transação de um bloco é um tipo especial de transação, chamada de `coinbase transaction`. Esta transação cria moedas como forma de recompensa ao *minerador*. As moedas *mineradas* podem ser gastas em blocos subsequentes. Esta recompensa incentiva nós à *minerarem* blocos, o que provê segurança à própria rede, pois aumenta o poder computacional necessário para realizar um *ataque de 51%*.

1.9 Ajuste de dificuldade

Por fim, neste sistema, a Prova De Trabalho de cada bloco é dinâmica, e não fixa. O campo `target`, presente nos blocos, define o quanto de trabalho médio é esperado para se resolver o desafio. Este valor é dinamicamente ajustado para que cada bloco seja produzido em um intervalo de tempo fixo, em média [Ant17, p. 235]. No Bitcoin, cada bloco é produzido a cada 10 minutos, em média, e no Monero este tempo é 2 minutos.

Um tempo médio entre blocos é necessário por várias razões:

- garantir que os nós da rede, os quais estão espalhados pelo modo, entrem em sincronia, incluindo aqueles com pouca *bandwidth*.
- prevenir ataques de *spam* ao impedir que blocos comecem a ser gerados a uma taxa muito rápida.
- restringir o espaço de disco necessário para armazenar os blocos, permitindo que indivíduos comuns consigam rodar o próprio nó, mantendo a rede descentralizada.
- manter a estabilidade da rede P2P, de forma que haja um consenso entre os nós, evitando que mudanças rápidas na rede resultem em uma divisão (*fork*) dos membros da mesma.

1.10 Outras moedas

Este capítulo resumiu o funcionamento genérico de uma moeda criptográfica baseada no modelo do Bitcoin. Alguns dos aspectos apresentados vão variar dependendo do sistema, o qual pode ter diferentes esquemas e implementações tais como: tipos de transações válidas, tamanho dos blocos, ajuste de dificuldade, sistema de recompensas, regras de validação, armazenamento e formato dos dados, codificação e compressão, e protocolos criptográficos específicos. Além disso, a moeda criptográfica pode implementar tecnologias específicas, tais como uma linguagem de programação própria (sistema de *scripts* no caso do Bitcoin) ou *Contratos Inteligentes*. Mesmo assim, muitas delas são similares em design, e os fundamentos aqui ilustrados ajudarão a compreender Monero e Zcash, estudadas a seguir.

Capítulo 2

Monero

Em 2013, uma entidade anônima sob o pseudônimo Nicolas Van Saberhagen publicou o artigo *CryptoNote Protocol* [Sab13], no qual apresenta uma série de esquemas para resolver falhas de privacidade no Bitcoin e, baseada nelas, propõe o protocolo de dinheiro eletrônico *CryptoNote*. O CryptoNote tornou-se a base para inúmeros projetos de moedas criptográficas focadas em privacidade, incluindo Monero.

Em 2014, com base no protocolo CryptoNote, `thankful_for_today`, usuário do fórum de discussão online *BitcoinTalk*, criou a moeda *BitMonero* [tha04]. A moeda foi divulgada no fórum e teve apoio de vários membros para seu desenvolvimento. Eventualmente, a moeda foi renomeada para Monero, que significa *dinheiro* em Esperanto.

O objetivo do Monero foi a criação de uma moeda criptográfica decentralizada que permitisse efetuar transações privadas e anônimas. Isto é possível graças a três tecnologias[1] principais [Ser18, p. 60]:

- *Stealth Addresses*: geração de endereços de pagamentos únicos, na qual um endereço novo (que é uma espécie de identificador do destino do pagamento) é gerado aleatoriamente a cada transação. Dessa forma, saldos enviados ao mesmo proprietário não podem ser vinculados por terem endereços diferentes.
- *Ring Signatures*: assinaturas criptográficas em grupo, na qual um usuário membro de um grupo gera uma assinatura que prova que alguém do grupo gerou a assinatura de forma legítima mas sem revelar quem é esta pessoa. Isto permite provar que um saldo está sendo legitimamente gasto pelo proprietário, que é membro de um grupo, sem revelar quem do grupo é este proprietário.
- *RingCT*: transações de anéis confidenciais, as quais empregam *Pedersen commitments* para provar que os saldos de origem (de quem gasta) são iguais aos saldos de destino (de quem recebe) sem revelar os valores dos saldos para terceiros. Assim, os valores

[1] Monero também faz uso de outras tecnologias de privacidade, tais como a *Kovri*, um protocolo de roteamento e anonimização de trafego baseado no *I2P* cujo objetivo é desvincular o endereço IP de um membro da rede P2P e os dados que este membro transmitiu. Entretanto, este trabalho não tem como objetivo explorar a arquitetura de rede, otimização e compressão de transações, ou outros aspectos do Monero. Portanto, estes assuntos não serão abordados.

das transações são completamente ocultos exceto para quem está enviando o saldo e quem está recebendo.

Os próximos capítulos se destinam a explorar em detalhes como funcionam cada uma dessas tecnologias e quais problemas que elas resolvem. A versão estudada do protocolo é a versão v15, corresponde-te à versão v0.18 do software, apelidada de *Fluorine Fermi*. Esta é a versão mais recente do Monero no momento em que este trabalho foi desenvolvido.

2.1 Primitivos Criptográficos

Monero faz uso de algumas tecnologias criptográficas específicas. Elas não são necessárias para entender os esquemas de privacidade de Monero, que são agnósticos quanto à implementação dos algoritmos. Entretanto, constituem detalhes técnicos relevantes porque são amplamente usadas no protocolo.

2.1.1 Curva Ed25519

Monero utiliza da curva elíptica *Twisted Edwards - Ed25519*, equivalente *biracional* da curva *Montgomery 25519* [Ser18, p. 111].

Está curva é definida sobre o campo $\mathbb{F}_{2^{255}-19}$ aplicada à seguinte equação:

$$-x^2 + y^2 = 1 - \frac{121655}{121666}x^2y^2$$

A curva é utilizada em diversas operações criptográficas, tais como criptografia assimétrica. No Monero, os elementos da curva são representados como inteiros utilizando 256 bits, equivalente a 32 bytes.

Ademais, como cada inteiro da curva tem no máximo 255 bits, o bit extra (no caso, o bit mais significativo) pode ser aproveitado na compressão da curva. Em vez de representar a curva como um par de pontos (x, y), o que gastaria 64 bytes, é possível representar o ponto somente com y e derivar x a partir da equação:

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

onde $d = \frac{121655}{121666}$.

O sinal de x pode ser obtido da seguinte maneira: se o bit mais significativo de y for igual ao bit menos significativo de x (computado pela equação), então x é positivo. Caso contrário é negativo.

2.1.2 Keccak

Keccak é o nome do algoritmo de hash usado no Monero e também o vencedor da competição da NIST (*National Institute for Standards and Technology*, agência americana cuja uma das tarefas é estabelecer padrões criptográficos) para ser o algoritmo `SHA3`.

Monero usa *Keccak-256* com 32 bytes de saída em transações, no hashing dos blocos, na sua Função Geradora de Números Pseudoaleatórios, e em seu sistema PoW, chamado CryptoNight PoW, que emprega um algoritmo de hashing CryptoNote baseado no *Keccak*.

2.2 Stealth Addresses

2.2.1 Contexto

A tecnologia *Stealth Addresses* surgiu em face do seguinte problema: transações no Bitcoin que utilizam o mesmo endereço do recipiente podem ser vinculadas. Isto ocorre por causa da arquitetura do Bitcoin: existem vários tipos de transações, sendo a mais comum a *PayToPubKeyHash*, na qual envia um saldo para um endereço que é o hash da chave pública do recipiente [Ant17, p. 136]. Para gastar o saldo recebido, o recipiente deve prover uma chave pública cujo hash deve ser igual ao endereço e uma assinatura digital comprovando que o recipiente detém a chave privada desta chave pública. O nome deste tipo de transação deriva justamente do fato que ela *"paga para o hash da chave pública"* do recipiente.

Assim sendo, se um vendedor fornece seu endereço para um cliente, o cliente poderá consultar o registro de dados público do Bitcoin, contendo todas as transações, e poderá calcular o saldo que o vendedor possui associado com este endereço. Uma forma de mitigar isto é usando múltiplos endereços: é possível gerar um número arbitrário de endereços, pois a geração de um endereço consiste basicamente na geração de um par de chave pública-privada e a computação do hash correspondente à chave pública.

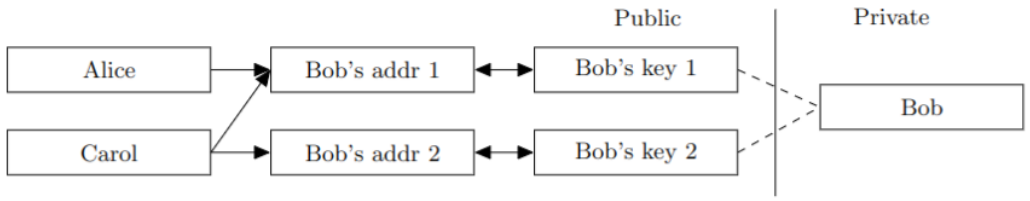

Figura 2.1: *Modelo de transação no Bitcoin*
Créditos: CryptoNote Whitepaper

Entretanto, esta forma de mitigação exige gerar e gerenciar múltiplos endereços. Enquanto o gerenciamento de endereços pode ser facilitado por aplicações gráficas, aliviando os esforços do usuário, ainda assim é necessário gerar um endereço novo para cada cliente. Dito isso, toda vez que se for efetuar uma transação, para preservar a privacidade é necessário gerar um novo endereço para que a outra parte não tenha conhecimento do saldo do recipiente.

Stealth Addresses visa justamente resolver esse impasse. Por meio desta tecnologia [Ser18, p. 62], o pagador gera um endereço de uso único, que pode ser reconhecido e gasto pelo recipiente. Outras partes, que não sejam o pagador ou o recipiente, não são capazes de encontrar uma associação entre os diversos endereços de uso único possuídos pelo recipiente.

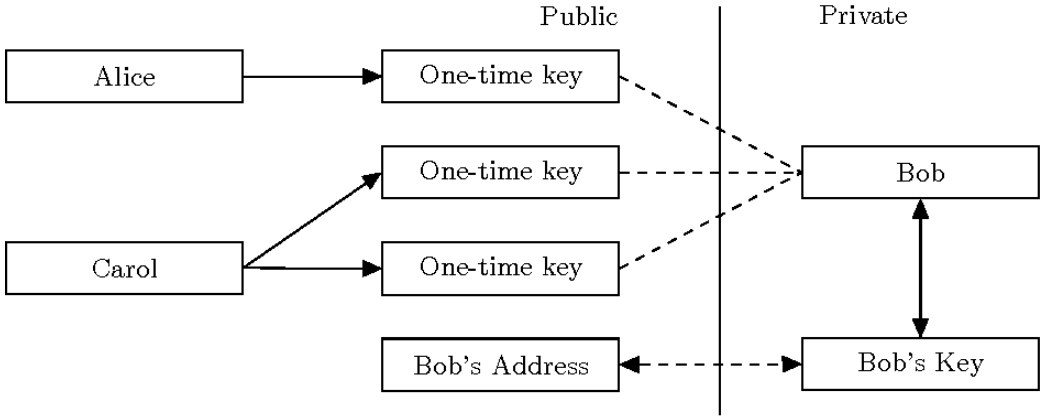

Figura 2.2: *Modelo de transação no CryptoNote*
Créditos: CryptoNote Whitepaper

2.2.2 Geração de endereços

O protocolo CryptoNote descreve um esquema para geração de pagamentos não vinculáveis. Esta especificação é implementada no Monero sob o nome *Stealth Addresses*. Seu funcionamento é o seguinte [Sab13, p. 6]:

1. Alice quer enviar pagamento para Bob, cujo endereço público é a codificação de suas chaves públicas de visualização K_v e gasto K_s. Alice decodifica o endereço e obtém as chaves públicas.

2. Alice gera um número aleatório $r \in \{1, l-1\}$ e computa uma chave pública de uso único $K = H_n(rK_v)G + K_s$

3. Alice utiliza a chave pública K gerada como o destino do valor sendo transferido na transação. Ela inclui $R = rG$ dentro dos dados da transação.

4. Alice envia a transação.

5. Bob checa cada transação e, usando suas chaves privadas (k_v, k_s), e computa $K' = H_n(Rk_v)G + K_s$. Se $K' = K$, a transação é destinada para Bob.

6. Bob pode provar que é proprietário do valor recebido por meio de uma assinatura digital associada à chave pública da transação. A chave pública é $K = H_n(Rk_v)G + K_s = (H_n(Rk_v) + k_s)G$, de modo que a chave privada é $k = H_n(Rk_v) + k_s$, satisfazendo $K = kG$ e podendo ser computada por Bob.

Uma ilustração deste esquema é provida por Saberhagen [Sab13, p. 8]:

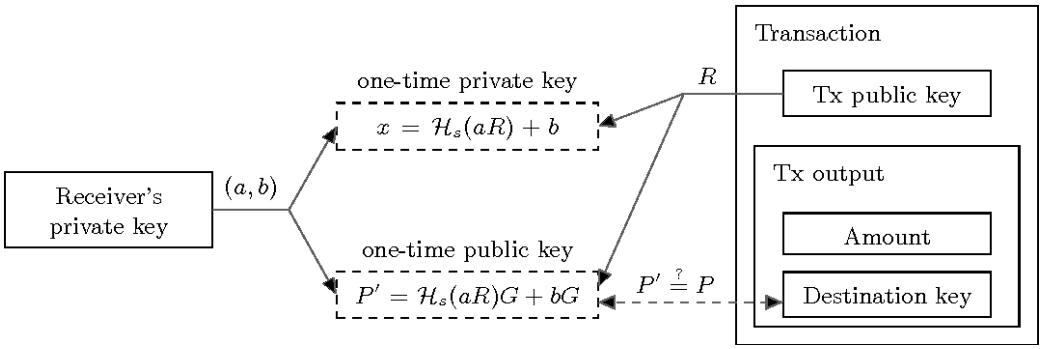

Figura 2.3: *Stealth Address no CryptoNote*
Créditos: CryptoNote Whitepaper

Note que na ilustração é usada uma notação diferente para as variáveis, mas a matemática é a mesma. Veja que $K_s = aG$ e $K_v = bG$ e que substituindo no diagrama acima resulta em equações idênticas ao esquema descrito.

2.2.3 Demonstração matemática da criptografia

Para associar a transação destinada à Bob com o endereço público de Bob é necessário saber a chave privada de visualização k_v. A demonstração é direta: os únicos valores públicos são R, K, K_v, K_s; para associar K com K_v e K_s é necessário computar $H_n(Rk_v) = H_n(rK_v)$, mas somente Alice sabe r e somente Bob sabe k_v.

Para gastar o saldo associado da transação, é necessário saber a chave privada da transação e gerar uma assinatura digital com essa chave. Computar a chave privada $k = H_n(Rk_v) + k_s$ exige saber o valor k_s. Dessa forma, somente quem detém a chave privada de gasto k_s consegue gastar o valor recebido pela transação.

2.3 Ring Signatures

Uma *Ring Signature*, ou Assinatura em Anel, em tradução livre, é um método que permite um membro de um grupo assinar digitalmente uma mensagem provando que alguém do grupo assinou a mensagem sem revelar quem é esta pessoa. Existem variações de *Ring Signatures* com modificações que visam cumprir diferentes propósitos, tais como *SAG*, *LSAG*, *MLSAG* e *CLSAG*. Monero atualmente usa CLSAG, mas é necessário entender as outras variantes porque CLSAG é construída com base em MLSAG, que é construída com base em LSAG.

2.3.1 Contexto

Esta tecnologia foi escolhida e adaptada para resolver o seguinte problema: no Bitcoin, para efetuar uma transação é necessário prover uma assinatura digital que prove posse dos fundos de origem da transação; como o sistema é público, é possível traçar o histórico de posse e transferência dos bitcoins, o que pode ser combinado com informações adicionais para tentar inferir os diferentes donos de bitcoin ao longo do tempo e informações a respeito de suas transferências.

O modo que Monero resolve tal problema é com o uso `Ring Signature` para criar ambiguidade sobre a origem do saldo: sabe-se que alguém do grupo está gastando os fundos de origem legitimamente, mas não se sabe quem é. Desta forma, traçar a origem e o destino dos fundos fica inviável: todos os membros do grupo da assinatura são candidatos equiprováveis de serem o dono dos fundos.

2.3.2 LSAG

Monero usava uma variante de *Ring Signatures* chamada *Concisable Linkable Spontaneous Anonymous Group (CLSAG)*. Será primeiro explicado assinaturas *Linkable Spontaneous Anonymous Group (LSAG)*[2], por ser mais simples de entender, e então explicado *MLSAG*, depois *CLSAG*.

Geração de assinatura LSAG

Seja m a mensagem a ser assinada. Seja $\mathcal{R} = \{K_1, K_2, \dots, K_n\}$ um conjunto de chaves públicas distintas. Seja k_π a chave secreta correspondente à chave pública $K_\pi \in \mathcal{R}$ onde π é um índice secreto aleatoriamente gerado.

Os passos para produzir uma assinatura LSAG são [NAK20, p. 29]:

1. Calcule a *imagem de chave* $\hat{K} = k_\pi \mathcal{H}_p(K_\pi)$
2. Gere um número aleatório $\alpha \in \mathbb{Z}_l$ e números aleatórios $r_i \in \mathbb{Z}_l$ para $i \in \{1, 2, \dots, n\}$, mas excluindo π.
3. Calcule:[3]

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

4. Para $i = \pi + 1, \pi + 2, \dots, n, 1, 2, \dots, \pi - 1$ (nesta exata ordem), calcule:

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

   Quando $i = n$, substituir $i + 1$ por 1.

5. . Calcule $r_\pi = \alpha - c_\pi k_\pi \mod l$

A assinatura gerada é $\sigma(m) = (c_1, r_1, r_2, \dots, r_n)$, com imagem de chave $\hat{K}$ e anel $\mathcal{R}$.

Eis uma figura ilustrando este esquema [Sab13, p. 9]:

[2] Também chamada de Back's LSAG (*bLSAG*) em homenagem ao seu criador

[3] Os colchetes `[]` nas equações são utilizados apenas para facilitar a distinção dos diferentes parâmetros que são passados para a função $\mathcal{H}_n$, portanto não denota alguma operação ou qualquer outra propriedade matemática.

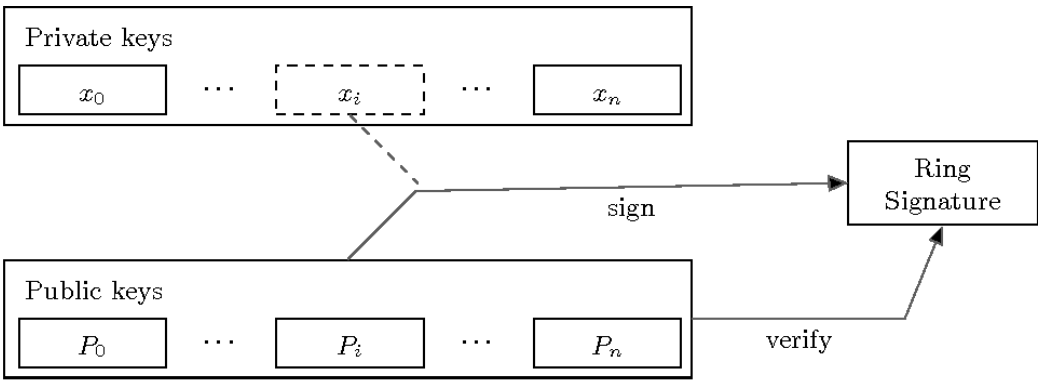

Figura 2.4: *Assinatura de Anel Génerica*
Créditos: CryptoNote Whitepaper

Verificação da assinatura LSAG

Verificar a assinatura de anel significa provar que $\sigma(m)$ é uma assinatura válida sobre a mensagem m gerada por uma chave privada k corresponde à alguma chave pública em $\mathcal{R}$ sem saber qual chave pública é esta.

Isto é feito da seguinte forma [NAK20, p. 31]:

1. Inicialmente, faça $c'_1 = c_1$. Para $i = 1, 2, \ldots, n$, compute:

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

   Quando $i = n$, substituir $i + 1$ por 1.

2. e $c'_1 = c_1$, a assinatura é válida

Demonstração matemática da verificação

Como c'_{i+1} é o valor obtido por uma função de hash criptográfica, cuja entrada depende de c'_i, têm-se que $c'_{i+1} = c_{i+1}$ somente se o valor passado para a função de hash $\mathcal{H}_p$ no cálculo de c'_{i+1} for igual ao usado no hash no cálculo de c_{i+1}, ou se algum atacante conseguiu encontrar uma colisão no Hash.

Assumindo que a implementação da função abstrata $\mathcal{H}_p$ é criptograficamente segura, a chance do atacante encontrar uma colisão é igual ao inverso do número possíveis de saídas para esta função de hash, que é $\frac{n}{l} \approx \frac{n}{2^{255}}$, onde n é o número de tentativas de *brute force* do atacante.

Portanto, achar uma colisão nesta função de Hash é computacionalmente inviável. Logo, se $c'_{i+1} = c_{i+1}$, os valores passados como entrada para a função de hash tem probabilidade extremamente alta de serem iguais.

Estes valores dependem de $m, r_i, K_i, \hat{K}$, que são públicos. Qualquer ligeira modificação destas variáveis iria resultar em um Hash diferente devido ao *efeito avalanche*.

Além disso, quem gerou a assinatura de anel deve necessariamente saber o valor de alguma chave privada associada com uma das chaves públicas. Isso pois, a *imagem de*

chave é derivada da chave privada. Se algum atacante falsificar um valor para a imagem de chave, a verificação irá falhar. Veja que:

$$
\begin{aligned}
r_\pi &= \alpha - c_\pi k_\pi \\
c_{\pi+1} &= \mathcal{H}_n(m, [\alpha G], [\alpha \mathcal{H}_p(K_\pi)]) \\
c_{\pi+1} &= \mathcal{H}_n(m, [(r_\pi + c_\pi k_\pi) G], [(r_\pi + c_\pi k_\pi) \mathcal{H}_p(K_\pi)]) \\
c_{\pi+1} &= \mathcal{H}_n(m, [r_\pi G + c_\pi k_\pi G], [r_\pi \mathcal{H}_p(K_\pi) + c_\pi k_\pi \mathcal{H}_p(K_\pi)]) \\
c_{\pi+1} &= \mathcal{H}_n(m, [r_\pi G + c_\pi K_\pi], [r_\pi \mathcal{H}_p(K_\pi) + c_\pi \hat{K}])
\end{aligned}
$$

Então, caso um atacante gere uma imagem de chave falsa, ele não conseguirá calcular r_π tal que $r_\pi = \alpha - c_\pi k_\pi$ por causa do problema do logaritmo discreto em curvas elípticas.

Enquanto r_i é escolhido aleatoriamente para $i \neq \pi$, o valor de π precisa ser derivado da chave privada k_i para que os valores c_i sejam corretamente computados de forma a garantir a igualdade no passo 1 da verificação.

Se $r_\pi \neq \alpha - c_\pi k_\pi$, teremos que :

$$
c_{\pi+1} \neq \mathcal{H}_n(m, [r_\pi G + c_\pi K_\pi], [r_\pi \mathcal{H}_p(K_\pi) + c_\pi \hat{K}])
$$

Pois nem $\hat{K}$ e nem r_π foram derivados usando alguma chave privada k_π, de modo que a única forma do atacante escolher um valor correto para r_π é chutando este valor, que é o mesmo que adivinhar um ponto aleatório na curva elíptica.

Portanto, este esquema é criptograficamente seguro e um atacante não conseguirá, com alta probabilidade, gerar uma assinatura de anel válida sem conhecer pelo menos uma chave privada correspondente à uma das chaves públicas do anel.

2.3.3 MLSAG

Conforme será explicado mais adiante, uma transação em Monero pode enviar várias entradas e cada entrada tem seu par de chaves pública e privada. Assim sendo, é necessário um esquema que permita provar a posse não apenas de uma chave privada, mas de várias chaves.

O MLSAG alcança esta façanha generalizando o LSAG com ligeiras modificações, permitindo provar que o assinante detém posse de m chaves. Então, em vez do anel $\mathcal{R}$ ser um conjunto de n chaves públicas, $\mathcal{R}$ é um conjunto de n vetores, cada vetor $K_i = (K_{i,1}, K_{i,2}, \ldots, K_{i,m})$ tendo m chaves públicas.

Essencialmente, a assinatura irá provar que o assinante detém todas as chaves privadas das chaves públicas de algum vetor K_π, sem revelar qual vetor é esse (valor de π desconhecido por terceiros).

Geração de MLSAG

Seja π o índice secreto do vetor de chaves públicas do assinante. Os passos para gerar a assinatura MLSAG sobre a mensagem m são [NAK20, p. 32]:

1. Calcule as *imagens de chave*:

$$\hat{K}_j = k_{\pi,j}\mathcal{H}_p(K_{\pi,j}) \quad j \in \{1, 2, \dots, m\}$$

2. Gere números aleatórios $\alpha_i, r_{i,j} \in \mathbb{Z}_l$ para $i = 1, 2, \dots, n$ (exceto quando $i = \pi$) e para $j = 1, 2, \dots, m$

3. Calcule:

$$h_{\pi+1} = \Big([\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], [\alpha_2 G], [\alpha_2 \mathcal{H}_p(K_{\pi,2})], \dots, [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})]\Big)$$
$$c_{\pi+1} = \mathcal{H}_n(m, h)$$

4. Para $i = \pi + 1, \pi + 2, \dots, n, 1, 2, \dots, \pi - 1$ (nesta exata ordem), calcule:

$$h_{i+1} = \Big([r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \hat{K}_1], \dots, [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \hat{K}_m], \Big)$$
$$c_{i+1} = \mathcal{H}_n(m, h_{i+1})$$

   Quando $i = n$, substituir $i + 1$ por 1.

5. Calcule $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j} \mod l$ para $j = 1, 2, \dots, m$

A assinatura gerada é $\sigma(m) = (c_1, r_{1,1}, \dots, r_{1,m}, \dots, r_{n,m}, \dots r_{n,m})$, com imagens de chave $\hat{K}_1, \dots, \hat{K}_m$ e anel $\mathcal{R}$.

Verificação de MLSAG

Isto é feito da seguinte forma [NAK20, p. 33]:

1. Inicialmente, faça $c'_1 = c_1$. Para $i = 1, 2, \dots, n$, compute:

$$c'_{i+1} = \mathcal{H}_n(m, [r_{i,1}G + c'_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c'_i \hat{K}_1] \dots, [r_{i,m}G + c'_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c'_i \hat{K}_m])$$

   Quando $i = n$, substituir $i + 1$ por 1.

2. Se $c'_1 = c_1$, a assinatura é válida

Demonstração matemática da verificação

A demonstração matemática da verificação da assinatura MLSAG é semelhante à da LSAG, pois a primeira é uma generalização da segunda. Veja que:

$$
\begin{aligned}
r_{\pi,j} &= \alpha_j - c_\pi k_{\pi,j} \\
c_{\pi+1} &= \mathcal{H}_n(m, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], \dots, [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})]) \\
c_{\pi+1} &= \mathcal{H}_n(m, \\
&\quad [(r_{\pi,1} + c_\pi k_{\pi,1})G], [(r_{\pi,1} + c_\pi k_{\pi,1})\mathcal{H}_p(K_{\pi,1})], \\
&\quad [(r_{\pi,2} + c_\pi k_{\pi,2})G], [(r_{\pi,2} + c_\pi k_{\pi,2})\mathcal{H}_p(K_{\pi,2})], \\
&\quad \dots, \\
&\quad [(r_{\pi,m} + c_\pi k_{\pi,m})G], [(r_{\pi,m} + c_\pi k_{\pi,m})\mathcal{H}_p(K_{\pi,m})]) \\
c_{\pi+1} &= \mathcal{H}_n(m, \\
&\quad [r_{\pi,1}G + c_\pi k_{\pi,1}G], [r_{\pi,1}\mathcal{H}_p(K_{\pi,1}) + c_\pi k_{\pi,1}\mathcal{H}_p(K_{\pi,1})], \\
&\quad [r_{\pi,2}G + c_\pi k_{\pi,2}G], [r_{\pi,2}\mathcal{H}_p(K_{\pi,2}) + c_\pi k_{\pi,2}\mathcal{H}_p(K_{\pi,2})], \\
&\quad \dots, \\
&\quad [r_{\pi,m}G + c_\pi k_{\pi,m}G], [r_{\pi,m}\mathcal{H}_p(K_{\pi,m}) + c_\pi k_{\pi,m}\mathcal{H}_p(K_{\pi,m})],) \\
c_{\pi+1} &= \mathcal{H}_n(m, \\
&\quad [r_{\pi,1}G + c_\pi K_{\pi,1}], [r_{\pi,1}\mathcal{H}_p(K_{\pi,1}) + c_\pi \hat{K}_1], \\
&\quad [r_{\pi,2}G + c_\pi K_{\pi,2}], [r_{\pi,2}\mathcal{H}_p(K_{\pi,2}) + c_\pi \hat{K}_2], \\
&\quad \dots, \\
&\quad [r_{\pi,m}G + c_\pi K_{\pi,m}], [r_{\pi,m}\mathcal{H}_p(K_{\pi,m}) + c_\pi \hat{K}_m],)
\end{aligned}
$$

Portanto, se a assinatura for válida, o valor c_π é calculado de forma indistinguível dos outros c_i, não sendo possível identificar π.

Para calcular c_π corretamente, é preciso saber as chaves privadas $k_{\pi,j}$ usadas no cálculo $r_{\pi,j}$. Sem saber essas chaves, um adversário teria que adivinhar vários $r_{\pi,j}$ aleatoriamente e de forma que todos eles satisfizessem a relação $r_{pi,j} = \alpha_j - c_\pi k_{\pi,j}$.

Não apenas $k_{\pi,j}$ seria desconhecido pelo adversário, como também c_π depende do hash de $c_{\pi-1}$, que depende do hash de $c_{\pi-2}$, e assim retrogradadamente até $c_{\pi+1}$. Visto que pequenas variações na mensagem de entrada mudam drasticamente a saída de uma função de hash, pelo efeito avalanche, a chance do adversário conseguir providenciar valores $k_{\pi,j}$ é negligenciável.

Logo, a verificação da assinatura fornece uma prova criptográfica que o assinante de fato conhecia todas as chaves privadas $k_{\pi,j}$ para algum vetor de chaves de índice π secreto.

2.3.4 CLSAG

A assinatura *CLSAG* foi introduzida em Monero na versão v14 do protocolo, correspondente à versão v0.17 do software e apelidada de *Oxygen Orion* [sel20], sendo mandatória em todas as transações a partir daí. Esta transação é um meio termo entre *LSAG* e *MLSAG*: prova-se possessão de um conjunto de chaves assim como em *MLSAG*, mas a verificação da *imagem de chave* é feita apenas para uma chave assim como em *LSAG*.

Da mesma forma que em *MLSAG* têm-se um conjunto de $m \times n$ chaves, onde n é o tamanho do anel e m o conjunto de chaves de cada membro do anel. A *chave primária* de cada membro do anel tem índice 1 naquele grupo. Têm-se então:

$$R = K_{i,j},\ i \in \{1, 2, \dots, n\}, j \in \{1, 2, \dots, m\}$$

O gerador da assinatura conhece as chaves privadas $k_{\pi,j}$ correspondentes ao membro $K_\pi = \{K_{\pi,1}, \dots, K_{\pi,m}\}$, onde π é um índice secreto. A assinatura será feita sob a chave primária $k_{\pi,1}$, que irá assinar chaves auxiliares $k_{\pi,2}, \dots, k_{\pi,m}$

Os benefícios desta construção são tempos de verificação mais rápidos e menor requerimentos para armazenamentos, pois em uma das etapas de verificação é usado apenas a *imagem de chave agregada* para garantir a propriedade **vinculabilidade** (descrita na próxima seção).

Geração de CLSAG

A geração de uma assinatura *CLSAG* consiste das seguintes etapas [NAK20, p. 34]:

1. Compute imagens de chave $\hat{K}_j = k_{\pi,j}\mathcal{H}_p(K_{\pi,1})$.

   Note que a função de *hashing para ponto* $\mathcal{H}_p$ é aplicada sempre sob a chave primária, por isto as outras chaves são chamadas de auxiliares.

2. Gere números aleatórios $\alpha \in \mathbb{Z}_l$ e $r_i \in \mathbb{Z}_l$ para $i \in \{1, 2, \dots, \pi - 1, \pi + 1, \dots, n\}$ (isto é, excluindo $i = \pi$).

3. Para $i \in \{1, 2, \dots, n\}$, compute as chaves públicas agregadas W_i:

$$\begin{aligned} W_i &= \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n) K_{i,j} \\ &= \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n) \sum_{j=1}^{m} K_{i,j} \end{aligned}$$

   Onde T_j = `CLSAG_j` é um prefixo usado em Monero para criar, efetivamente, funções de hash diferente para separação de domínios (uma nova regra do Monero). Assim, mesmo que as entradas fossem iguais, os valores W_i seriam diferentes.

4. computar a *imagem de chave agregada* $\hat{W}$:

$$\hat{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n)\hat{K}_j$$

5. computar $c_{\pi+1}$:

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

   Onde $T_c = \mathtt{CLSAG\_c}$ tem o mesmo propósito que T_j.

6. Para $i \in \{\pi+1, \pi+2, \dots, n, 1, 2, \dots, \pi-1\}$, computar:

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

   quando $i = n$, substituir $i + 1 = n + 1$ por 1.

7. Faça $r_\pi = \alpha - c_\pi w_\pi \mod l$

A assinatura σ sob a mensagem $\mathtt{m}$ é $\sigma(m) = (c_1, r_1, \dots, r_n)$, com imagem de chave primária $\hat{K}_1$ e imagens de chave auxiliares $\hat{K}_2, \dots, \hat{K}_m$.

Verificação de CLSAG

A verificação de uma assinatura CLSAG consiste das seguintes etapas [NAK20, p. 35]:

1. Para $j \in \{1, \dots, m\}$, verifique se $l\hat{K}_j$. Se falhar, alguma imagem de chave foi gerada incorretamente e a assinatura é inválida.

2. Para $i \in \{1, 2, \dots, n\}$, compute as chaves públicas agregadas W_i:

$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n)K_{i,j}$$

3. compute a *imagem de chave agregada* $\hat{W}$:

$$\hat{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n)\hat{K}_j$$

   Note que $\hat{W} = W_\pi$, mas somente o gerador da assinatura saberá qual W_i corresponde à $\hat{W}$, pois π é secreto.

4. Para $i \in \{1, 2, \dots, n\}$, compute:

$$c'_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [r_i G + c_i W_i], [r_i \mathcal{H}_p(K_{\pi,1}) + c_i \hat{W}])$$

   quando $i = n$, substituir $i + 1 = n + 1$ por 1.

5. A assinatura é válida se somente se $c_1 = c'_1$

Demonstração da verificação

A demonstração da verificação é parecida com a do MLSAG. Primeiramente, note que na geração das chaves públicas agregadas W_i, para $i = \pi$, têm-se:

$$\begin{aligned} W_\pi &= \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n) \sum_{j=1}^{m} K_{\pi,j} \\ &= \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n) \sum_{j=1}^{m} k_{\pi,j} G \\ &= \left[\mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \hat{K}_2, \dots, \hat{K}_n) \sum_{j=1}^{m} k_{\pi,j}\right] G \\ &= w_\pi G \end{aligned}$$

Onde $w_\pi = \mathcal{H}_n(T_j, \mathcal{R}, \hat{K}_1, \dots, \hat{K}_n)(k_{\pi,1} + \dots + k_{\pi,m})$ é a chave privada de W_π. Como $\hat{W} = W_\pi$, segue que $\hat{w} = w_\pi$, onde $\hat{w}$ é a chave privada da chave pública agregada.

Note também que quando $i = \pi$, computamos:

$$c'_{i+1} = c'_{\pi+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [r_\pi G + c_\pi W_\pi], [r_i \mathcal{H}_p(K_{\pi,1}) + c_\pi \hat{W}])$$

Se a assinatura foi gerada corretamente, $r_\pi = \alpha - c_\pi w_\pi$, têm-se:

$$\begin{aligned} c_\pi &= \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [\alpha G], [\alpha \mathcal{H}_n(K_{\pi,1})]) \\ &= \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [(r_\pi + c_\pi w_\pi) G], [(r_\pi + c_\pi w_\pi) \mathcal{H}_n(K_{\pi,1})]) \\ &= \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [r_\pi G + c_\pi w_\pi G], [r_\pi \mathcal{H}_n(K_{\pi,1}) + c_\pi w_\pi \mathcal{H}_n(K_{\pi,1})]) \\ &= \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [r_\pi G + c_\pi W_\pi], [r_\pi \mathcal{H}_n(K_{\pi,1}) + c_\pi \hat{w} \mathcal{H}_n(K_{\pi,1})]) \\ &= \mathcal{H}_n(T_c, \mathcal{R}, \mathtt{m}, [r_\pi G + c_\pi W_\pi], [r_i \mathcal{H}_p(K_{\pi,1}) + c_\pi \hat{W}]) \\ &= c'_\pi \end{aligned}$$

Note que c_π só será igual à c'_π se o gerador da assinatura tiver calculado corretamente o valor r_π e para isto é necessário ele saber a chave privada w_π, que depende do valor das chaves privadas $k_{\pi,1}, k_{\pi,2}, \dots, k_{\pi,n}$.

Como w_π depende da soma das chaves privadas $k_{\pi,j}$ é possível que um adversário pudesse adivinhar o valor da soma e calcular w_π sem saber o valor de cada chave individual. Entretanto, sem saber o valor de cada chave individual ele não conseguiria calcular as imagens de chave $\hat{K}_j = k_{\pi,j} \mathcal{H}_p(K_{\pi,1})$. As imagens de chaves são entradas para a função de hashing usado nos cálculos de c_i, logo é necessário saber o valor destas imagens de

chave para computar o valor do hash corretamente. Do contrário, é necessário encontrar uma colisão de hash. Assim sendo, a construção descrita é segura sob a hipótese de que a função de hash usada é resistente à colisão.

Assim sendo, a assinatura *CLSAG* funciona como *MLSAG*, provando que o gerador da assinatura sabe m chaves secretas correspondentes às chaves públicas de algum membro do anel, sem revelar qual membro do anel é.

2.3.5 Propriedades de LSAG, MLSAG e CLSAG

As assinaturas LSAG, MLSAG e CLSAG têm as seguintes propriedades [NAK20, p. 30]:

- **Espontaneidade**: a assinatura pode ser gerada sem pedir permissão de outros membros do grupo, pois suas chaves públicas são públicas e não requerem qualquer ação por parte deles. Assim, evita-se a necessidade de uma autoridade certificadora ao qual poderia comprometer a privacidade do assinante.
- **Ambiguidade do assinante**: um observador externo pode perceber que algum membro do anel produziu a assinatura de anel, mas não sabe quem é este membro: não consegue deduzir o dono de qual chave público do grupo.
- **Vinculabilidade**: o esquema de assinaturas de anel gera uma chave pública $\hat{K}$ que é deterministicamente derivada de k_π. Se alguém utilizar a mesma chave privada para gerar duas assinaturas de anéis, mesmo que os seus membros e o índice π sejam distintos, estas duas assinaturas serão vinculáveis: seus valores de $\hat{K}$ serão iguais.
- **Legitimidade**: a assinatura é legítima (foi gerada por algum membro do grupo que detém uma das chaves privadas) exceto com probabilidade negligenciável.

Estas propriedades são cruciais para garantir a privacidade e integridade das transações na rede Monero, como será explicado a seguir.

2.3.6 Ring Signatures no Monero

No Monero, as transações possuem uma ou mais entradas, que são a origem dos saldos da transação, e uma ou mais saídas, que são o destino do saldo. Em termos técnicos, uma entrada consiste de uma *ring signature* e uma saída consiste de um *stealth address*. Na verdade, há dados adicionais na entrada e saída, os quais serão discutidos na seção *RingCT*.

As entradas de uma transação produzirão saídas, as quais servirão de entradas em novas transações. Uma saída tem associada à si um *stealth address*, que é uma chave pública K, e um valor R usado pelo recipiente para derivar a chave privada k correspondente à K. A chave pública K será usada como um dos membros da assinatura de anel [NAK20, p. 48].

Os outros membros do anel são chaves públicas escolhidas aleatoriamente a partir de registros públicos de transações passadas. Estas chaves públicas pertencem à outros usuários da rede Monero. Estes outros usuários não precisam ser conhecidos e nem colaborar: a assinatura de anel pode ser gerada sem sua colaboração, visto que é necessário saber apenas suas chaves públicas, que são dados públicos. É assim que a propriedade *espontaneidade* é usada.

Assim sendo, após coletar um grupo de chaves públicas, Alice pode gerar uma assinatura de anel que provando que alguém do grupo detém posse dos fundos de entrada, sem revelar quem é. É assim que a propriedade *ambiguidade* é usada.

A posse dos fundos é provada pela assinatura de anel, pois requer saber a chave privada associada ao *stealth address* da saída de uma transação passada, o qual está sendo agora usado como entrada. É assim que a propriedade *legitimidade* é usada.

Ademais, Alice não pode tentar gastar o seu saldo várias vezes sem ser detectada. Se ela tentar enviar um saldo que possui para diversas pessoas (criando dinheiro a partir do nada e cometendo fraude), suas assinaturas de anéis serão vinculadas por meio de sua imagem de chave $\hat{K}$, que será igual para toda assinatura de anel, conforme já visto. Diante disso, é uma regra do protocolo Monero que uma transação é inválida se possuir uma entrada cuja imagem de chave ocorreu em transações passadas [4]. É assim que a propriedade *vinculabilidade* é usada.

O diagrama a seguir mostra como *Ring Signatures* e *Stealth Addresses* são combinados em uma transação [Sab13, p. 11]:

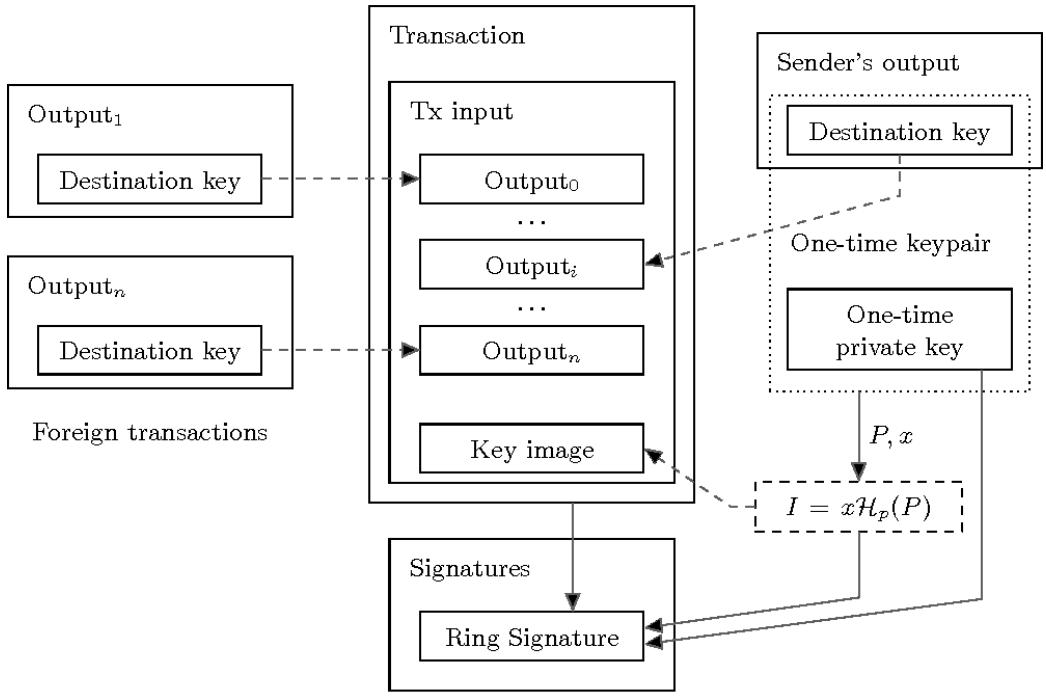

Figura 2.5: *Transação usando Ring Signatures e Stealth Address*
Créditos: CryptoNote Whitepaper

2.3.7 Ring Signature Size

O tamanho padrão das *Ring Signatures* no Monero é 16, conforme as regras da versão v15 do protocolo, correspondente à versão v0.18 do software e apelidada de *Fluorine Fermi* [sel22]. Isto significa que todas as assinaturas em anel devem ter exatamente 16 membros,

[4] Na prática, significa que os nós da rede Monero precisam armazenar todas as imagens de chave para poder efetuar esta checagem.

sendo 1 destes uma *unspent output* controlada pelo usuário efetuando transação e 15 outras *unspent outputs*, chamadas de *decoys*, selecionadas aleatoriamente pelo registro público de transações passadas.

Desde a versão v8 do protocolo, correspondente à versão v0.13 do software, apelidada de *Beryllium Bullet*, as regras do protocolo Monero exigem que todas as transações tenham o mesmo tamanho [flu18]. O objetivo disto é prover anonimidade através de uniformidade: com um tamanho fixo, todos os usuários são vistos de forma igual, porém se os usuários podem escolher um tamanho arbitrário, então alguns deles se destacaram em relação aos outros por escolherem valores incomuns (valores muito altos ou sempre um mesmo valor), comprometendo não apenas o próprio anonimato, mas o anonimato dos outros.

2.4 RingCT

2.4.1 Contexto

Monero utiliza um protocolo chamado RingCT (Transações Confidenciais de Anel, em tradução livre) para ocultar os valores monetários nas transações ao mesmo tempo que garante que os valores são legítimos e não há fraudes. Antes da versão v6 do protocolo, correspondente à versão 0.11 do software, chamada de *Hellium Hydra*, valores nas transações do Monero eram públicos, mas a partir desta versão o protocolo exige o uso de RingCT nas transações [flu17].

Antes, valores das transações eram valores fixos chamados de *denominações*. É análogo ao valor de moedas de ouro ou papel moeda com, por exemplo, denominações fixas de 2, 5, 10, 20, 50 e 100. No Monero, estas denominações eram usadas para realçar a privacidade, pois valores arbitrários (tal como 23.37) significa que seria possível identificar a origem e destino das transações apesar do uso de *Ring Signatures* e *Stealth Address*, pois os valores seriam tipicamente únicos.

2.4.2 Commitments

A ocultação dos valores é feito por meio de *commitments*: esquemas criptográficos no qual uma entidade compromete-se atestar veracidade de uma declaração envolvendo um valor oculto, uma espécie de promessa com garantias, porém sem revelar este valor. Para honrar o compromisso, a entidade fornece provas criptográficas como forma de evidência que a declaração é verídica.

2.4.3 Pedersen commitments

Um *Pedersen commitment* é um tipo de *commitment* que tem a propriedade de ser aditivo [NAK20, p. 52]:

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

Onde $C(x)$ denota um *commitment* ao valor x. Esta propriedade é útil no Monero para

provar que a soma das quantias das entradas é igual a soma das quantias das saídas, sem revelar estes valores.

Pedersen commitment é um categoria de *commitment*, portanto ele não especifica como deve ser sua implementação matemática. No caso de curvas elípticas, sua implementação é trivial graças à propriedade de soma de pontos em curvas elípticas:

$$C(x) = xG \therefore C(a) + C(b) = aG + aB = C(a + b) = (a + b)G$$

Pelo problema do logaritmo discreto em curvas elípticas, é computacionalmente difícil saber o valor de x sabendo apenas o valor de $C(x)$, o que faz com que o esquema apresentado acima seja uma implementação simples de um *Pedersen commitment*.

2.4.4 Commitments em Monero

O esquema descrito na última seção tem um problema do ponto de vista de privacidade: sendo o valor do gerador G público, um atacante pode pre-computar $C(a)$ para vários valores de a, criando uma tabela que lhe permitiria consultar o valor de a com base no valor de $C(a)$ presente em um transação. Transações envolvendo valores comerciais comuns seriam facilmente detectáveis nesta situação.

Para obter mais privacidade, é necessário acrescentar um fator de obscurecimento que previna ataques de pre-computação de tabelas. Como analogia, este fator extra atuará como uma espécie de *salt*, método usado para mitigar ataques de *rainbow tables* [SP18, p. 401].

Tendo isto em mente, define-se o seguinte esquema de *commitment* [NAK20, p. 45]:

$$C(x, a) = xG + aH$$

Neste esquema a é o valor que está sendo *committed* e x é uma espécie de *salt*, um valor aleatório que é introduzido para ofuscar mais ainda o valor de a.

O H é outro gerador, diferente de G. O porquê de se usar outro gerador H em vez de G é o seguinte: nas transações de Monero, precisamos provar que a soma das quantias das entradas é igual a soma das quantias da saída para que não haja criação fraudulenta de dinheiro; se usássemos G em vez de H, qualquer um poderia facilmente criar *commitments* válidos e ao mesmo tempo falsificando o valor a. Veja que se $H = G$:

$$C(x, a) = xG + aH = xG + aG = (x - 1000)G + (a + 1000)G = C(x - 1000, a + 1000)$$

Portanto, para evitar a fraude acima, é necessário que H seja um novo gerador e que $H = \gamma G$ para que H gere os mesmos pontos que G. O valor de γ deve ser desconhecido pois, se não, é fácil ver que $C(x, a) = C(w, b)$ onde $w + b = x + \gamma a$. No caso de γ ser conhecido, valores de w podem ser escolhidos para aumentar o valor de b, que representa a quantia de moedas, criando assim dinheiro do nada.

No caso do Monero, $H = 8 \times \mathcal{H}_p(\mathcal{H}_n(G))$ [5]. A função de hashing para ponto garante que H é um ponto aleatório dentro da curva, portanto o valor de γ é desconhecido pela dificuldade em se resolver o problema do logaritmo discreto em curvas elípticas.

Voltando a discussão da privacidade deste esquema de *commitments*, note que há infinitos pares de inteiros (x, a) que produzem o mesmo *commitment* $C(x, a)$, pois este *commitment* é um ponto na curva elíptica, cujo domínio é um conjunto finito de pontos de um campo finito, mas x e a são inteiros com o domínio em Z.

Desta forma, há inúmeras combinações possíveis, e mesmo se o adversário adivinhe uma dessas combinações, ele não tem meios para averiguar que a combinação que ele achou é realmente o par de valores usados no *commitment*. Independente do poder computacional, o adversário não tem pista para descobrir os valores verdadeiros. Assim, é alcançado *segurança de informação-teórica*.

2.4.5 Ocultação de quantias

O esquema *commitments* descrito é empregado em Monero da seguinte forma [NAK20, p. 45]:

$$\begin{aligned} C(x, a) &= xG + aH \\ h &= \mathcal{H}_n(rK_v, t) \\ x &= \mathcal{H}_n(\text{“}commitment\ mask\text{”}, h) \\ \text{quantia} &= a \oplus_8 \mathcal{H}_n(\text{“}amount\text{”}, h) \end{aligned}$$

Onde:

- a é a quantia verdadeira da transação
- x é um valor pseudoaleatório usada como máscara de ofuscação de a
- r é um *nonce* secreto gerado pelo remetente.
- K_v é a chave pública de visualização do recipiente
- t é um índice numérico da posição desta saída na transação (pode haver múltiplas saídas).[6]
- `quantia` é um valor público pseudoaleatório que o recipiente usará para descobrir o valor verdadeiro da quantia
- $\oplus_8$ é a operação `XOR` aplicada aos primeiros 8 bytes de cada operando.

[5] A multiplicação por 8 é para verificar que H e G sejam geradores do mesmo subgrupo [NAK20, p. 48]. Se $\mathcal{H}_p(\mathcal{H}_n(G))$ não estiver no subgrupo de G, isto significa que está no outro subgrupo, cuja ordem é 8. Neste caso, multiplicar o valor por 8 geraria o ponto de afinidade $\mathcal{O}$. Empiricamente, $H \neq \mathcal{O}$, portanto H e G são de fato do mesmo subgrupo.

[6] Concatenar t na saída é feito para que saídas diferentes que tenham o mesmo destinatário não gerem valores iguais para “quantia” e para a máscara de ofuscação (exceto com probabilidade baixíssima).

O valor rG é anexado aos dados da transação, enquanto r é mantido em segredo e pode ser descartado pelo remetente, caso ele queira. Assim, o recipiente pode calcular x e a da seguinte maneira:

$$
\begin{aligned}
h &= \mathcal{H}_n((rG)k_v, t) \\
x &= \mathcal{H}_n(\text{“commitment mask”}, h) \\
a &= \text{quantia} \oplus_8 \mathcal{H}_n(\text{“amount”}, h)
\end{aligned}
$$

Onde k_v é a chave privada de visualização do recipiente. Assim sendo, somente o remetente (que sabe r) e quem possuir a chave privada de visualização poderão saber o valor verdadeiro a da transação. Terceiros não conseguirão computar h ou x para poder descobrir o real valor de a.

2.4.6 Prova da legitimidade de quantias ocultas

Para provar que as quantias ocultadas são legítimas, primeiramente deve-se garantir que a soma das quantias das entradas é igual a soma das quantias da saída. Usando o esquema de *Pedersen commitments* mostrado anteriormente, é suficiente provar que (assumindo m entradas e n saídas):

$$
\texttt{diff} = \sum a_i - \sum b_j = 0
$$

Onde os a_i são os valores verdadeiros de entrada e b_j são os valores verdadeiros de saída. Para tanto, basta provar que `diff` é igual a zero na seguinte equação:

$$
\begin{aligned}
\sum C_{\in} - \sum C_{\texttt{out}} &= \sum_i^m \Big(x_i G + a_i H \Big) - \sum_j^n \Big(y_j G + b_j H \Big) \\
&= \Big(\sum x_i - \sum y_j \Big) G + \Big(\sum a_i - \sum b_j \Big) H \\
&= \Big(\sum x_i - \sum y_j \Big) G + \text{diff} \times H \\
&= kG + \texttt{diff} \times H
\end{aligned}
$$

Onde $C_{\in}$ denota os *commitments* de entrada e $C_{\texttt{out}}$ os *commitments* de saída. Assim, $\sum C_{\in}$ é a soma dos *commitments* de entrada e $\sum C_{\texttt{out}}$ é a soma dos *commitments* de saída. Para preservar a balança, queremos que a diferença entre estas somas seja 0, que é equivalente a termos $\texttt{diff} = 0$

Nesta equação, um recipiente legítimo deve conhecer x_i, pois ele é o dono das chaves privadas das entradas e pode calcular x_i. O recipiente também deve conhecer y_j, pois é ele que irá gerar os *commitments* $C(y_j, b_j)$.

Seja $K = \sum C_{\in} - \sum C_{\texttt{out}}$. Se $\texttt{diff} = 0$, então $K = kG$. Logo, o recipiente consegue

gerar uma assinatura digital sobre o ponto K, visto que ele consegue computar k. A prova que $\texttt{diff} = 0$ é a própria assinatura digital sobre K, que não poderia ter sido obtida pelo recipiente se $\texttt{diff} \neq 0$ devido ao *ECDLP*.

Esta abordagem funciona, porém ela tem o seguinte problema de privacidade: ela permite terceiros vincularem os *commitments* de saída com *commitment* de entradas. Isto tornaria o esquema de *Ring Signatures* inútil, visto que este último visa justamente ofuscar a origem dos fundos.

A solução do Monero para resolver estes problemas é a seguinte: usa-se novos *commitments* chamados *pseudo commitments* no lugar dos *commitments* originais. Um *pseudo commitment* de $C(x, a)$ é um *commitment* $C(x', a)$, onde a nova máscara x' é um valor aleatório. Como os valores de a são os mesmos para o *commitment* original e o *pseudo commitment*, pode-se usar a esquema anterior para provar que $\texttt{diff} = 0$, mas dessa vez $k = \sum x'_i - \sum y_j$.

É necessário também que x'_i não seja associado à x_i para que não seja possível estabelecer um vínculo entre os *pseudo commitments* e os *commitments* originais. Para fazer isso, utiliza-se CLSAG.

No CLSAG, têm-se um anel $\mathcal{R}$ de n vetores com m chaves públicas. O assinante detém posse de m chaves privadas de algum vetor com índice π desconhecido. Cada chave pública $K_{i,j}$ do anel está associada a um *commitment* $C_{i,j}$ (regra do protocolo Monero). Defina:

$$K'_{i,j} = K_{i,j} + C_{i,j} - C'_{\pi,j}$$

Onde $C'_{\pi,j}$ é o *pseudo commitment* correspondente ao *commitment* $C_{\pi,j}$ da chave pública $K_{\pi,j}$, cujo remetente detém posse da chave privada. O termo $C_{i,j} - C'_{\pi,j}$ é chamado de *commitment para zero*, pois quando $i = \pi$ o *commitment* é uma garantia criptográfica que $a_j - a'_j$ é zero.

Ademais, como π é secreto e $C'_{\pi,j}$ é gerado usando uma máscara aleatória, não há como estabelecer vínculo entre os *pseudo commitments* e os *commitments* originais a partir de $K'_{i,j}$ ou dos próprios *pseudo commitments* [NAK20, p. 46].

Note que:

$$\begin{aligned} K'_{\pi,j} &= K_{\pi,j} + C_{\pi,j} - C'_{\pi,j} \\ &= k_{\pi,j}G + (x_i G + a_j H) - (x'_i G + a_j H) \\ &= (k_{\pi,j} + x_i - x'_i)G \\ &= k'_{\pi,j} G \end{aligned}$$

Portanto, o remetente consegue computar as chaves privadas de todos os $K'_{\pi,j}$. Ele pode então gerar uma assinatura CLSAG sobre o anel $\mathcal{R}'$ composto pelas chaves públicas $K'_{i,j}$. Esta assinatura prova que os *pseudo commitments* $C'(x, a)$ são legítimos *pseudo commitments*, que o valor de a nos *pseudo commitments* não foi fraudado, pois do contrário não seria

possível saber os valores $k'_{\pi,j}$ necessário para gerar a assinatura sobre o anel R'.

Assim sendo, o esquema de assinatura de anel descrito remove o vínculo entre os *pseudo commitments* e os *commitments* originais, ao passo que também prova a legitimidade dos *pseudo commitments*, assegurando que o valor a desses *pseudo commitments* não foi fraudado.

Com base nisso, o remetente pode usar o esquema mostrado anteriormente para provar que $\sum a_i - \sum b_j = 0$. Têm-se:

$$\begin{aligned}
\sum C'_{\texttt{in}} - \sum C_{\texttt{out}} &= \sum_{i}^{m} \Big(x'_i G + a_i H \Big) - \sum_{j}^{n} \Big(y_j G + b_j H \Big) \\
&= \Big(\sum x'_i - \sum y_j \Big) G + \Big(\sum a_i - \sum b_j \Big) H \\
&= \Big(\sum x'_i - \sum y_j \Big) G + \texttt{diff} \times H \\
&= kG + \texttt{diff} \times H
\end{aligned}$$

O remetente pode então prover uma assinatura digital sobre a chave pública do ponto $K = \sum C'_{\texttt{in}} - \sum C_{\texttt{out}}$, onde $C'_{\texttt{in}}$ são os *pseudo commitments* e $C_{\texttt{out}}$ os *commitments* da saída. Isso só será possível se o remetente conseguir computar o valor de k, o que significa que $\texttt{diff} = 0$ e a soma das quantias de entrada é de fato igual a soma das quantias de saída.

2.5 Transações

Após a discussão do funcionamento das três tecnologias empregadas em Monero, o próximo passo é ver como elas são combinadas em uma transação típica.

Uma transação consiste de m entradas X_i, cada entrada com uma quantia oculta a_i e *commitment* C_{x_i}. Estas entradas, *commitments*, e quantias são conhecidas apenas pelo remetente, o qual gera *pseudo commitments* C'_{x_i} públicos correspondentes aos C_{x_i}. Então, ele gera uma assinatura CLSAG para provar que os *pseudo commitments* são legítimos e que ele detém posse de m chaves privadas de algumas entradas, mas sem revelar estas entradas [NAK20, p. 48].

Para cada um dos n destinatários Y_j (podendo serem a mesma pessoa), o remetente usa os n endereços públicos K_{Y_j} para calcular os endereços de recebimento de uso único $K^o_{Y_j}$. Cada endereço $K^o_{Y_j}$ irá receber uma quantia oculta b_j, e terá um *commitment* determinístico C_{Y_j}, que só pode ser revelado pelo recipiente ou pelo remetente. Usando novamente *commitments*, o remetente prova que a soma das entradas a_i é igual a soma das saídas b_j, assegurando que não houve fraude na quantia.

Este tipo de transação é chamada de `RCTTypeBulletproof2`, a qual é a transação padrão no Monero desde a versão v10, liberada na versão v0.14 do software, chamada de *Boron Butterfly*. Também havia a transação `RCTTypeFul` (depreciada por não ocultar as quantias), `RCTTypeSimple` (substituída por *RCTTypeBulletproof* devido à atualizações

e melhorias), e `RCTTypeBulletproof` (sucessor de `RCTTypeSimple`, mas substituída por `RCTTypeBulletproof2`, também devido à melhorias) [NAK20, p. 49].

2.6 Outras tecnologias

Monero é um sistema computacional complexo composto de vários componentes criptográficos, elementos de rede, ferramentas de otimização e bibliotecas de software bem estabelecidas. Assim, vários aspectos do Monero não foram cobertos neste capítulo, mas vale a pena destacar alguns deles:

- **Bulletproofs**: sistema de provas *zero-knowledge* usado na validação de uma transação, provando que cada quantia em uma transação está dentro de um intervalo pré-definido (entre 0 e $2^{64} - 1$), evitando assim a possibilidade de quantias com valores negativos ou valores além do limite do protocolo [NAK20, p. 47]. *Bulletproofs* permitem transações menores e mais eficientes ao reduzir o tamanho das provas criptográficas e usar provas sucintas. Monero tem planos para no futuro melhorar esta tecnologia por meio de seu sucessor *Bulletproofs++*.
- **Multisignatures**: Esquema de assinaturas múltiplas, *multisig*, na qual membros de um grupo detém conjuntamente posse de um saldo que somente pode ser gasto se um subconjunto de membros assinar simultaneamente a transação. Um *multisig* `m-n` significa que são necessários pelo menos m assinaturas de um grupo com n integrantes, com $m \leq n$.
- **Join Transactions**: trata-se de um esquema similar ao *CoinJoin* do Bitcoin, no qual vários usuários distintos misturam suas transações de forma à anonimizar a origem dos saldos e evitar heurísticas construídas em cima do grafo de transações. Monero já provê *Ring Signatures* para ofuscar origem dos fundos e *Join Transactions* visa melhorar isto adicionando mais camadas de ambiguidade e mitigar heurísticas dos dados públicos, tais como padrões do número de transações, frequência e horário.
- **Kovri**: sistema de anonimização de trafego de rede baseado no (*Invisible Internet Project - I2P*) que visa ofuscar endereços IP e outras informações que possam revelar dados sobre os usuários da rede Monero. Além de ofuscar o IP, o *Kovri* quebra o vínculo entre as mensagens enviadas pelo usuário e o próprio usuário, então não é somente a origem do usuário que é protegida mas quais dados este usuário transmitiu.
- **P2Pool**: protocolo P2P de *mineração* em conjunto. É comum *mineradores* formarem uma associação na qual eles mineram em conjunto e repartem os ganhos. Esta associação é chamada de *pool de mineração*. A principal vantagem de se participar de uma *pool* é obter recompensas com maior frequência: em vez de recompensas altas com baixa frequência prefere-se recompensas menores com frequência maior. Entretanto, isto pode levar à uma centralização da *mineração* e aumentar chances de um ataque de %51. O protocolo *P2Pool* visa permitir a *mineração* em conjunto de forma P2P, mitigando ataques e tornando a rede mais eficiente, decentralizada.

Mais detalhes sobre estas tecnologias e seu funcionamento podem ser encontrados em *Zero To Monero* [NAK20], *Mastering Monero* [Ser18], e *Monero Research Labs* [Lab].

Capítulo 3

Zcash

Zcash é uma implementação concreta do *Zerocash*, um protocolo abstrato de um sistema de moeda digital criptográfica, baseado no bitcoin e projetado para providenciar um elevado grau de privacidade e anonimato para seus usuários.

Zerocash foi proposta inicialmente em 2014, por um grupo de seis acadêmicos e pesquisadores da área de criptografia. Zcash foi criado por este mesmo grupo em colaboração com dezenas de outros profissionais nas áreas de criptografia e segurança da informação, além de se beneficiar de auditorias de segurança de empresas e pesquisadores.

Este capítulo está dividido em duas partes: (1) uma explicação abstrata de alto nível dos principais conceitos e funcionamento do protocolo Zerocash, o qual é implementado e melhorado por Zcash; (2) uma descrição da implementação concreta que Zcash emprega para prover privacidade aos seus usuários.

3.1 Zerocoin

O *Zerocoin* é um percursor do Zerocash que tentou resolver o problema de dinheiro eletrônico anônimo e privado.

O Zerocoin é um sistema paralelo construído em cima do Bitcoin com o propósito de ocultar a origem dos fundos, que é uma informação pública no Bitcoin. Então, Zerocoin não tem como objetivo substituir Bitcoin, mas prover um sistema que coexiste com Bitcoin e estende suas funcionalidades para prover privacidade aos usuários do Bitcoin.

A proposta do Zerocoin é introduzir uma nova transação denominada `Zerocoin Mint`, que possibilitaria usuários de Bitcoin destruir seus bitcoins em troca do direito de possuir *zerocoins*. Após um usuário efetuar uma transação destruindo 1 bitcoin, ele teria o direito de reivindicar 1 *zerocoin*, em uma relação um-para-um. O usuário pode então usar o sistema Zerocoin para misturar suas moedas com outras moedas, ofuscando o histórico de origem de sua moeda e obtendo privacidade. Após isso, o usuário poderia destruir seus *zerocoins* e reivindicar novos bitcoins que devem ser criados a partir de uma nova transação especial `Zerocoin spend` (que seria implementada no Bitcoin) que converte *zerocoins* para bitcoins [Mie+13].

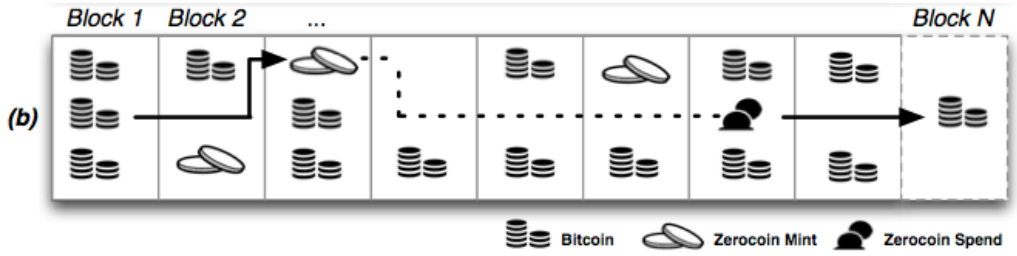

Figura 3.1: *Transações Zerocoin ocultam a origem das moedas*
Créditos: Zerocoin Paper

A privacidade do Zerocoin funciona por meio de uma mistura de moedas. Cada moeda do Zerocoin é validada por meio de uma prova que ela pertence à uma lista públicas de moedas não gastas, mas sem revelar qual moeda é essa. Após ser gasta, o *serial number* da moeda é revelado, prevenindo que esta moeda seja gasta novamente (pois, sabemos pelo *serial number* revelado anteriormente que ela já foi gasta). Entretanto, não é possível associar o *serial number* de uma moeda com a moeda em si, nem deduzir suas origens, devido as *provas de zero conhecimento*.

3.2 Zerocash

O *Zerocash* é o protocolo abstrato implementado e aperfeiçoado por Zcash. Este protocolo define um *DAP scheme (Decentralized Anonymous Payment Scheme)*, que em tradução literal seria Esquema de Pagamento Anônimo Decentralizado. Um DAP, definido no artigo, é constituído de diversos componentes criptográficos, conceitos e algoritmos.

3.2.1 Introdução

O Zerocash é construído com base no Zerocoin, mas com melhorias significativas. O sistema Zerocoin possuem alguns gargalos: Zerocoin ofusca apenas a origem dos fundos, mas revela o destino dos fundos e a quantia transferida. Ademais, seu mecanismo de provas é consideravelmente lento e requer alto espaço de disco para armazenamento das provas. Zerocash propõe um sistema para resolver esses impasses.

O sistema do Zerocash consiste de uma série de estrutura de dados e algoritmos para permitir usuários enviarem e receberem moedas de maneira anônima e privada, bem como criarem moedas segundo alguma regra (pois, em algum momento é necessário que as moedas sejam criada). Por meio de uma combinação de várias ferramentas criptográficas, tais como assinaturas digitais, esquemas de *commitment*, encriptação assimétrica, provas zero-knowledge, geradores pseudo-aleatórios, entre outros, é construído um esquema que define de forma abstrata as regras e parâmetros públicos de um protocolo de pagamentos digitais que usuários podem usar para criar endereços para receber saldo, gastar o saldo de seus endereços, e verificar a legitimidade de transações, ao passo que preservam sua privacidade.

Assim como no Zerocoin, números seriais são usados para prevenir gasto duplo e ofuscar a origem dos fundos. A verificação de uma transação legítima também é feita conferindo se ela pertence à uma lista de moedas válidas, sem revelar qual moeda é,

mas usando provas mais eficientes. O sistema *zero-knowledge* empregado é baseado em *zk-SNARKs* e depende da construção de um Circuito Aritmético para um problema NP chamado `POUR`, o qual consiste em se provar que uma estrutura dados pertencente à um campo finito satisfaz uma série de condições. Estas condições visam impor restrições necessárias ao protocolo, tais como: preservar a balança dos saldos de entrada e saída, prevenir corrupção dos dados nas transações, evitar o gasto duplo, impedir gastos de moedas não autorizados.

3.2.2 Estrutura de dados

Primeiramente, é necessário introduzir o conjunto de estrutura de dados que são usadas pelas operações e algoritmos do protocolo [Sas+14, p. 13]:

Lista Ligada: um registro de dados distribuído representando uma sequência cronológica de transações monetárias, uma espécie de lista ligada no qual é possível apenas adicionar dados, mas nunca editá-los ou deletá-los. É público e acessível à todos os usuários. Em um dado momento T, os usuários tem acesso à L_T, a lista ligada L naquele momento. Se $T' > T$, então L_T é prefixo de $L_{T'}$.

Árvore de Merkle: uma árvore de Merkle binária T com raiz `rt`, a qual será denominada por raiz Merkle daqui em diante, utilizada para representar de forma eficiente uma lista de *commitments* sobre o número serial de uma moeda (definida abaixo). Esta estrutura de dados permite consultar rapidamente se um valor está na árvore e adicionar novos valores em tempo $O(h)$, onde h é a altura da árvore. É semelhante à lista pública do Zerocoin, mas é bem mais eficiente computacionalmente.

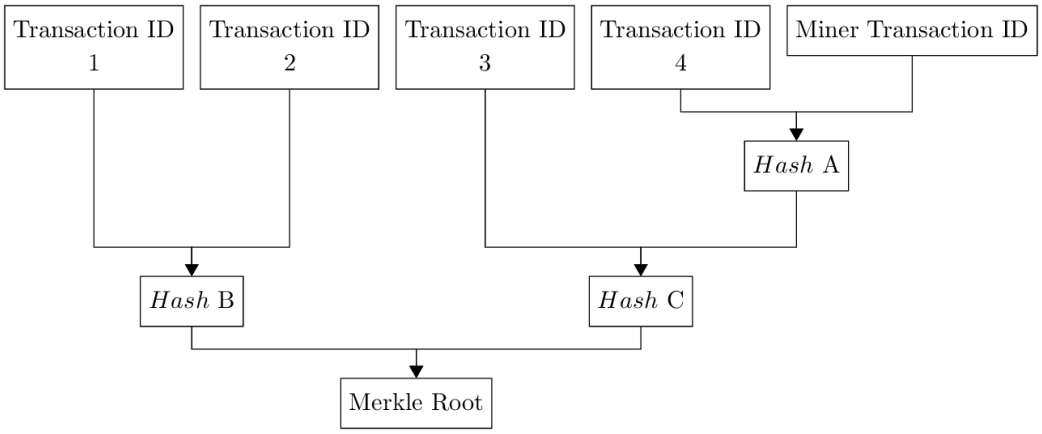

Figura 3.2: *Árvore de Merkle*
Créditos: Zerocoin Paper

Parâmetros Públicos: um conjunto de parâmetros de sistema disponíveis aos usuários e utilizadas nas operações criptográficas. Estes parâmetros são definidos por um Terceiro de Confiança na inicialização do sistema.

Endereços: Um endereço é um par de chaves pública-privada: $(\texttt{addr}_{\texttt{pk}}, \texttt{addr}_{\texttt{sk}})$. A chave pública $\texttt{addr}_{\texttt{pk}}$ é usada para receber moedas, e a chave privada $\texttt{addr}_{\texttt{sk}}$ (também chamada de chave secreta) é usada para gastar o saldo. Um usuário pode ter quantos endereços quiser e pode gerá-los de modo *offline*.

Moedas: Uma moeda é um objeto de dados descrito pelas seguintes informações:

- **valor**: valor monetário da moeda, que pode ir desde 0 até $v_{\max}$, que é um parâmetro do sistema.
- **commitment**: é uma garantia criptográfica que assegura que a moeda existe e é legítima (não foi criada por um processo fraudulento).
- **endereço**: um endereço público representando quem é proprietário da moeda (aquele que souber a chave privada correspondente).
- **Número serial**: uma string aleatória que é utilizada para prevenir o *gasto duplo* (ver seção 1.4).

Transações: Há duas transações principais no esquema DAP: `Mint` e `Pour`.[1]

.

- **Mint**: A transação `Mint`(significando confeccionar, fabricar) é usada para criar novas moedas no sistema.
- **Pour**: A transação `Pour`(significando despejar, separar), é usada para combinar gastar duas moedas que não foram gastas previamente, criando duas novas moedas como resultado. As novas moedas podem então ser gastas pelos seus donos.

Nota: A criação das moedas no sistema pode ser feita, por exemplo, após um usuário comprovar um depósito de bitcoins para obter o direito de ter moedas neste novo sistema, assim como no Zerocoin. Eventualmente, um usuário pode reivindicar os bitcoins, destruindo as moedas dentro do sistema DAP para obter os bitcoins correspondentes. Neste exemplo, haveria uma relação 1-para-1 entre a moeda do sistema DAP e 1 bitcoin. Vale ressaltar que as regras para a criação de moedas (quem pode criar a moeda, quando e porquê) vão depender da implementação do DAP, a qual deve por mais restrições na transação `Mint` para prevenir inflação arbitrária.

Os detalhes das transações serão vistos a seguir, nos algoritmos das mesmas. Por exemplo, a transação `Pour` requer algoritmos e provas criptográficas para permitir a transferência de moedas sem revelar a origem das moedas, seu destino, e o valor correspondente.

3.2.3 Primitivos Criptográficos

Zerocash define um conjunto de Primitivos Criptográficos adaptados ao protocolo [Sas+14, p. 18]. Seja λ um parâmetro de segurança que define o nível de segurança do sistema. Os Primitivos Criptográficos são:

CRH [Collision-Resistant Hashing]: Função de hash resistente à colisões definida como $\texttt{CRH} : \{0,1\}^* \to \{0,1\}^{O(\lambda)}$

PRF [Pseudo-Random Functions]: Família de funções Pseudo-Aleatórias, cada função definida como $\texttt{PRF} = \{\texttt{PRF}_x : \{0,1\}^* \to \{0,1\}^{O(\lambda)}\}$, onde x denota a semente do gerador de números pseudo-aleatórios. A partir de $\texttt{PRF}_x$ defini-se três funções

[1] é possível adicionar mais transações ao protocolo, mas pelo menos estas duas transações são obrigatórias

pseudo-aleatórias distintas auxiliares: $\mathtt{PRF}_x^{\mathtt{addr}}(z) = \mathtt{PRF}_x(00\|z)$ usada na geração de números aleatórios para endereços, $\mathtt{PRF}_x^{\mathtt{sn}}(z) = \mathtt{PRF}_x(01\|z)$ usada na geração de *números seriais* aleatórios, e $\mathtt{PRF}_x^{\mathtt{pk}}(z) = \mathtt{PRF}_x(10\|z)$ usada na geração de números aleatórios para a chave pública. A implementação de PRF deve ser uma função resistente à colisão (não necessariamente uma função de hash).

***Commitments* Estatisticamente Ocultos**: Um esquema de *commitments* COM cuja propriedade de ocultação tem *Segurança Estatística*. Isto significa que até mesmo para um adversário com poder computacional irrestrito teria probabilidade negligenciável de descobrir o valor do *commitment* a menos que este valor seja revelado [Hai+09]. Isto é atingido pelo uso de *commitment trapdoors*, uma máscara usada na ocultação do valor. Denota-se $\mathtt{COM}_x : \{0,1\} \to 0,1^{O(\lambda)}$ o *commitment* que usa x como máscara.

Assinaturas Digitais de Uso Único Fortemente Não Falsificáveis: Esquema de assinaturas digitais com probabilidade negligenciável de falsificação e cujas chaves são usadas uma única vez. É definido como:

$$\mathtt{Sig} = (\mathcal{G}_{\mathtt{sig}}, K_{\mathtt{sig}}, S_{\mathtt{sig}}, V_{\mathtt{sig}})$$

- $\mathcal{G}_{\mathtt{sig}}(1^\lambda) \to \mathtt{pp}_{\mathtt{sig}}$: Dado um parâmetro de segurança λ, o algoritmo $\mathcal{G}_{\mathtt{sig}}$ gera aleatoriamente parâmetros públicos $\mathtt{pp}_{\mathtt{sig}}$ que são usados no esquema de encriptação.
- $K_{\mathtt{sig}}(\mathtt{pp}_{\mathtt{sig}}) \to (\mathtt{pk}_{\mathtt{sig}}, \mathtt{sk}_{\mathtt{sig}})$: Dado parâmetros públicos $\mathtt{pp}_{\mathtt{sig}}$, o algoritmo $K_{\mathtt{sig}}$ gera aleatoriamente uma chave pública e uma chave privada ara serem usadas em assinaturas digitais.
- $S_{\mathtt{sig}}(\mathtt{sk}_{\mathtt{sig}}, m) \to \sigma$: Dado uma chave secreta $\mathtt{sk}_{\mathtt{sig}}$ e uma mensagem m, o algoritmo $S_{\mathtt{sig}}$ gera uma assinatura digital σ.
- $V_{\mathtt{sig}}(\mathtt{pk}_{\mathtt{sig}}, m, \sigma) \to b$: Dado uma chave pública $\mathtt{pk}_{\mathtt{sig}}$, uma mensagem m e uma assinatura σ, o algoritmo $V_{\mathtt{sig}}$ retorna $b = 0$ se a assinatura é inválida e $b = 1$ se a assinatura é válida.

Encriptação Assimétrica: Esquema de encriptação simétrica enc definido como:

$$\mathtt{enc} = (\mathcal{G}_{\mathtt{enc}}, K_{\mathtt{enc}}, \mathcal{E}_{\mathtt{enc}}, D_{\mathtt{enc}})$$

- $\mathcal{G}_{\mathtt{enc}}(1^\lambda) \to \mathtt{pp}_{\mathtt{enc}}$: Dado um parâmetro de segurança λ, o algoritmo $\mathcal{G}_{\mathtt{enc}}$ gera aleatoriamente parâmetros públicos $\mathtt{pp}_{\mathtt{enc}}$ que são usados no esquema de encriptação.
- $K_{\mathtt{enc}}(\mathtt{pp}_{\mathtt{enc}}) \to (\mathtt{pk}_{\mathtt{enc}}, \mathtt{sk}_{\mathtt{enc}})$: Dado parâmetros públicos $\mathtt{pp}_{\mathtt{enc}}$, o algoritmo $K_{\mathtt{enc}}$ gera aleatoriamente uma chave pública e uma chave privada para serem usadas em encriptação e decriptação.
- $\mathcal{E}_{\mathtt{enc}}(\mathtt{pk}_{\mathtt{enc}}, m) \to c$: Dado uma chave pública $\mathtt{pk}_{\mathtt{enc}}$ e uma mensagem m, o algoritmo $Enc_{\mathtt{enc}}$ gera um texto cifrado c.
- $D_{\mathtt{enc}}(\mathtt{sk}_{\mathtt{enc}}, c) \to m$: Dado uma chave privada $\mathtt{sk}_{\mathtt{enc}}$ e um texto cifrado c, o algoritmo $D_{\mathtt{enc}}$ decripta a cifra para produzir a mensagem m correspondente ou então falha.

3.2.4 zk-SNARKs

O sistema zk-SNARKs do Zerocash depende de dois conceitos chaves: *Circuitos Aritméticos* e *Satisfabilidade de Circuitos.*

Circuitos Aritméticos

Circuitos Aritméticos são utilizados para modelarem funções e polinômios em Teoria da Complexidade Computacional [AB09, p. 129]. Tipicamente, são usados para buscarem um algoritmo eficiente para computar um polinômio.

Dito isso, define-se um *Circuito Aritmético C* como um grafo direto acíclico construído sobre um campo finito $\mathbb{F}$ e um conjunto de variáveis $\{x_1, \ldots, x_n\}$. Os nós neste grafo com grau 0 são rotulados por variáveis ou constantes em $\mathbb{F}$, e são chamadas de *Portões de Entrada.* Os outros nós são chamados de *Portões* e são rotulados por um dos símbolos $+, -, \times, /$, denotando as operações aritméticas de soma, subtração, multiplicação e divisão [AB09, p. 322].

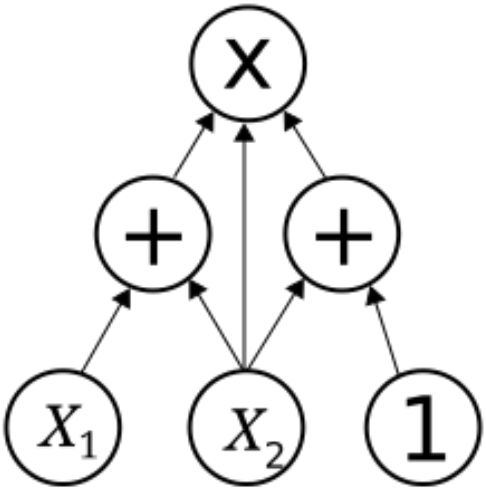

Figura 3.3: *Circuito Arimética*
Créditos: https://commons.wikimedia.org/wiki/File:ArithmeticCircuit.svg

Uma aresta indo de um portão A para o portão B indica que o valor representado por A é usado como entrada para calcular o valor do portão B. No exemplo acima, há três entradas cujos valores são x_1, x_2, e 1. No primeiro portão, combinamos x_1 e x_2 sob a operação $+$, resultando em $x_1 + x_2$. No segundo portão, combinamos x_2 e 1 sob a operação $+$, resultando em $x_2 + 1$. No terceiro portão, combinamos $(x_1 + x_2)$, x_2 e $(x_2 + 1)$ sob a operação $\times$, resultando em $(x_1 + x_2)(x_2)(x_2 + 1)$, que corresponde ao polinômio $x_2^3 + x_2^2 + x_1x_2^2 + x_1x_2$.

A relevância de Circuitos Aritméticos para Zerocash está na modelagem de algoritmos para problemas NP específicos que podem atuar como *trapdoors* criptográficas, sendo difícil calcular a solução do problema NP mas fácil verificar a validade de uma solução mediante um certificado (neste caso, é a própria prova criptográfica) e um algoritmo de verificação. Em particular, Zerocash define *zk-ZNARKs* em função do problema Satisfabilidade de Circuitos Aritméticos.

O problema Satisfabilidade de Circuitos Aritméticos é uma variação do problema

Satisfabilidade de Circuitos Booleanos (CSAT), mas para Circuitos Aritméticos. O CSAT é um problema de decisão que envolve determinar se o Circuito Booleano possui uma atribuição das variáveis de entradas que produz uma saída com valor booleano verdadeiro. Define-se então uma variante deste problema para Circuitos Aritméticos, de forma à aderir aos requerimentos do DAP [Sas+14, p. 10]. Formalmente:

Definição: Seja *Cum* Circuito Aritmético que recebe como entrada $x \in \mathbb{F}^n$ e uma entrada auxiliar $a \in \mathbb{F}^h$, chamada de testemunha, e produz saída em $\mathbb{F}^l$. Em notação matemática, $C : F^n \times F^h \to F^l$. Seja R_C a relação $R_C = \{(x, a) : C(x, a) = 0^l\}$, onde 0^l é o elemento nulo em $\mathbb{F}^l$. Seja L_C a linguagem $L_C = \{x \in \mathbb{F}^n : \exists a \in F^h \wedge (x, a) \in R_C\}$. O problema de Satisfabilidade de Circuitos Aritméticos é determinar se $x \in L_C$, o que significa encontrar uma atribuição para a que satisfaça $C(x, a) = 0^l$.

Definição de zk-SNARKs

Dado um campo $\mathbb{F}$ e um Circuito Aritmético C sobre $\mathbb{F}$, um sistema **zk-SNARKs** é definido [Sas+14, p. 10] como uma tupla de três algoritmos (`KeyGen`, `Prove`, `Verify`) tais que:

- $\texttt{KeyGen}(1^\lambda, C) \to (\texttt{pk}, \texttt{sk})$: dado o parâmetro de segurança λ e um Circuito Aritmético C sobre $\mathbb{F}$, o algoritmo `KeyGen` gera aleatoriamente uma chave provadora `pk` e uma chave verificadora `vk`. Estes valores são divulgados como parâmetros publicados e podem ser usados para provar ou verificar se um dado x pertence à L_C.
- $\texttt{Prove}(\texttt{pk}, x, a) \to \pi$: dado um parâmetro `pk`e $(x, a) \in R_C$, o algoritmo `Prove` gera uma prova π que x pertence à L_C.
- $\texttt{Verify}(\texttt{vk}, x, \pi) \to b$: dado um parâmetro `vk`, uma entrada x, e uma prova π, o algoritmo `Verify` produz $b = 1$ se $x \in L_C$ e $b = 0$ caso contrário, com probabilidade negligenciável de erro.

Segurança

Para ser seguro, a implementação do zk-SNARKs deve cumprir os seguintes requerimentos [Sas+14, p. 11]:

Completude: Dado um provador e um verificador que seguem honestamente o protocolo, o provador (que conhece $(x, a) \in R_C$) é capaz de convencer o verificador que $x \in L_C$, exceto com probabilidade negligenciável (a qual depende do parâmetro de segurança).

Sucinto: Dado um λ fixo, a prova π deve ter tamanho constante $O_\lambda(1)$ e o algoritmo `Verify` deve rodar em tempo linear $O_\lambda(x)$ em relação ao tamanho de x.

Consistência: se um verificador honesto foi convencido que $x \in L_C$, então o provador sabe uma testemunha a tal que $(x, a) \in R_C$, exceto com probabilidade negligenciável (dependente de λ).

Zero Conhecimento: a prova π não deve revelar informação que possa ser usada para inferir qual valor de a foi usada na geração da prova.

3.2.5 zk-SNARKs para POUR

A transação Pour consome duas moedas não gastas previamente, gerando duas novas moedas no processo, as quais podem ser gastas por seus novos donos. Para preservar a privacidade dos usuários, utiliza-se zk-SNARKs para comprovar que a transação é legítima e ao mesmo tempo não revelar nenhuma informação sobre a origem dos fundos, a quantia gasta e o destino dos fundos.

Para tanto, constrói-se um sistema zk-SNARKs para um problema NP chamado POUR. Este problema é baseado na própria transação Pour, a qual é representada como a seguinte estrutura de dados:

$$\mathtt{tx}_{\mathtt{Pour}} = (\mathtt{rt}, \mathtt{sn}_1^{\mathtt{old}}, \mathtt{sn}_2^{\mathtt{old}}, \mathtt{cm}_1^{\mathtt{new}}, \mathtt{cm}_2^{\mathtt{new}}, v_{\mathsf{pub}}, \mathtt{info}, *)$$

Onde:

- rt: raiz Merkle dos *commitments* de números seriais em algum momento após a criação das moedas $c_1^{\mathtt{old}}$ e $c_2^{\mathtt{old}}$
- $\mathtt{sn}_i^{\mathtt{old}}$: número serial da moeda $c_i^{\mathtt{old}}$
- $\mathtt{cm}_i^{\mathtt{new}}$: commitment sobre a moeda $c_i^{\mathtt{new}}$
- v_{pub}: um valor público (possivelmente 0), que poderia ser usado para reivindicar as moedas por Bitcoins, assim como no Zerocoin.
- info: dados extra arbitrários (i.e, uma anotação) anexados pelo remetente
- *: dados dependentes da implementação
- $c^{\mathtt{old}}$: moeda sendo gasta
- $c^{\mathtt{new}}$: nova moeda gerada

Relembrando que a estrutura de uma moeda c é:

$$\begin{aligned} c &= (\mathtt{addr}_{\mathsf{pk}}, v, p, r, s, \mathtt{cm}) \\ \mathtt{addr}_{\mathsf{pk}} &= (a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}}) \\ \mathtt{addr}_{\mathsf{sk}} &= (a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}}) \end{aligned}$$

Em essência, estas estruturas guardam as informações necessárias para efetuar o seguinte procedimento: gastar duas moedas $c_1^{\mathtt{old}}$ e $c_2^{\mathtt{old}}$ cujos números seriais são $\mathtt{sn}_1^{\mathtt{old}}$ e $\mathtt{sn}_2^{\mathtt{old}}$, registrados em uma Árvore de Merkle cuja raiz é rt, gerando no processo duas novas moedas $c_1^{\mathtt{new}}$ e $c_2^{\mathtt{new}}$, as quais possuem *commitments* $\mathtt{cm}_1^{\mathtt{new}}$ e $\mathtt{cm}_2^{\mathtt{new}}$ que serão adicionados à arvore para que possam ser gastas da mesma maneira.

Dito isso, o problema NP POUR é definido assim [Sas+14, p. 19]:

Seja uma instância x da forma:

$$x = (\mathtt{rt}, \mathtt{sn}_1^{\mathtt{old}}, \mathtt{sn}_2^{\mathtt{old}}, \mathtt{cm}_1^{\mathtt{new}}, \mathtt{cm}_2^{\mathtt{new}}, v_{\mathtt{pub}}, h_{\mathtt{sig}}, h_1, h_2)$$

Onde:

- $h_{\mathtt{sig}}, h_1, h_2$ são assinaturas usadas para garantir integridade.
- Os demais valores tem o mesmo significado anterior

Seja uma testemunha a da forma:

$$a = (\mathtt{path}_1, \mathtt{path}_2, c_1^{\mathtt{old}}, c_2^{\mathtt{old}}, \mathtt{addr}_{\mathtt{sk},1}^{old}, \mathtt{addr}^{\mathtt{sk},2}, c_1^{\mathtt{new}}, c_2^{\mathtt{new}})$$

Onde:

- $\mathtt{path}_i$ é um caminho de autentificação usado para localizar a posição do *commitment* $\mathtt{cm}_i^{\mathtt{old}}$ na Árvore Merkle com raiz $\mathtt{rt}$.
- Demais valores tem o mesmo significado que anteriormente

Dada uma instância x do problema POUR, uma testemunha a é válida se:

- Cada *commitment* $\mathtt{cm}_i$ aparece na árvore de raiz $\mathtt{rt}$ conforme $\mathtt{path}_i$
- Cada endereço público satisfaz $a_{\mathtt{pk}} = \mathtt{PRF}_{a_{\mathtt{sk}}}^{\mathtt{addr}}(0)$
- Cada número serial satisfaz $\mathtt{sn} = \mathtt{PRF}_{a_{\mathtt{sk}}}^{\mathtt{sn}}(p)$
- Cada *commitment* satisfaz $\mathtt{cm} = \mathtt{COM}_s(\mathtt{COM}_r(a_{\mathtt{pk}} || p) || v)$
- Cada assinatura satisfaz $h_i = \mathtt{PRF}_{a_{\mathtt{sk},i}^{\mathtt{old}}}^{\mathtt{pk}}(i || h_{\mathtt{sig}})$
- Saldo é preservado: $v_1^{\mathtt{new}} + v_2^{\mathtt{new}} + v_{\mathtt{pub}} = v_1^{\mathtt{old}} + v_2^{\mathtt{old}}$

Com base neste problema abstrato, o protocolo concreto deve implementar um sistema zk-SNARKs para este problema. Note que o sistema *zk-SNAKRs* é definido com base no problema de Satisfabilidade de Circuitos Aritméticos, portanto vai depender do Circuito Aritmético $C_{\mathtt{POUR}}$ construído para o problema POUR.

3.2.6 Algoritmos

Um esquema DAP é definido [Sas+14, p. 14] como uma tupla 6 de algoritmos, com complexidade de tempo polinomial, conforme segue:

$$\mathtt{DAP} = (\mathtt{Setup}, \mathtt{CreateAddress}, \mathtt{Mint}^2, \mathtt{Pour}, \mathtt{VerifyTransaction}, \mathtt{Receive})$$

Nota: As estruturas de dados e primitivos criptográficos utilizados no pseudo-código (que será apresentado) dos algoritmos do DAP são abstratos no que tange a implementação, apesar dos algoritmos serem concretos no sentido que eles especificam suas etapas. A análise dos pseudocódigos auxiliará na compreensão do ZCash, que se fundamenta neles.

Setup

O algoritmo `Setup` recebe como entrada um parâmetro de segurança λ e gera parâmetros de segurança pp, que são divulgados publicamente e utilizados como constantes na implementação do protocolo.

Seus passos são:

1. Construa um circuito C_{POUR} para o problema `POUR`
2. Gere $(\mathsf{pk}_{\mathsf{POUR}}, \mathsf{vk}_{\mathsf{POUR}}) \leftarrow \mathsf{KeyGen}(1^\lambda, C_{\mathsf{POUR}})$
3. Gere $\mathsf{pp}_{\mathsf{enc}} \leftarrow \mathcal{G}_{\mathsf{enc}}(1^\lambda)$
4. Gere $\mathsf{pp}_{\mathsf{sig}} \leftarrow \mathcal{G}_{\mathsf{sig}}(1^\lambda)$
5. Faça $\mathsf{pp} \leftarrow (\mathsf{pk}_{\mathsf{POUR}}, \mathsf{vk}_{\mathsf{POUR}}, \mathsf{pp}_{\mathsf{enc}}, \mathsf{pp}_{\mathsf{sig}})$
6. Retorne pp

A geração dos parâmetros pp é realizada por um Terceiro de Confiança, possivelmente em algum esquema de computação multipartidária. Após a geração, não é necessário a presença de Terceiros de Confiança no protocolo, e os segredos utilizados na geração de pp devem ser completamente destruídos pela entidade que os gerou.

CreateAddress

Dado parâmetros públicos pp, o algoritmo `CreateAddress` gera aleatoriamente um par $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$.

1. Gere $(\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}}) \leftarrow K_{\mathsf{enc}}(\mathsf{pp}_{\mathsf{enc}})$
2. Gere uma *seed* $a_{\mathsf{sk}} \leftarrow \mathsf{PRF}^{\mathsf{addr}}$
3. Compute $a_{\mathsf{pk}} \leftarrow \mathsf{PRF}^{\mathsf{addr}}_{a_{\mathsf{sk}}}$
4. Faça $\mathsf{addr}_{\mathsf{pk}} \leftarrow (a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$
5. Faça $\mathsf{addr}_{\mathsf{sk}} \leftarrow (a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$
6. Faça $\mathsf{addr} = (\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$
7. Retorne addr

O valor $\mathsf{addr}_{\mathsf{pk}}$ é um endereço público usado para receber moedas, e o valor $\mathsf{addr}_{\mathsf{sk}}$ é mantido em segredo e usado para gastar o saldo detido por $\mathsf{addr}_{\mathsf{pk}}$. Cada usuário pode gerar quantos endereços quiser, bastando apenas saber pp e dispor do algoritmo, e isto pode ser feito *offline*.

Mint

O algoritmo `Mint` recebe como entrada:

- parâmetros públicos pp
- valor monetário v $(0 \leq v \leq v_{\max})$

- endereço de destino $\texttt{addr}_{\texttt{pk}}$

Etapas:

1. Faça $a_{\texttt{pk}}, \texttt{pk}_{\texttt{enc}} \leftarrow \texttt{addr}_{\texttt{pk}}$
2. Gere uma *seed* $p \leftarrow \texttt{PRF}^{\texttt{sn}}$
3. Gere aleatoriamente duas *commitment trapdoors* r, s
4. Compute $k \leftarrow \texttt{COM}_r(a_{\texttt{pk}} \| p)$
5. Compute *coin commitment* $\texttt{cm} \leftarrow \texttt{COM}_s(v \| k)$
6. Faça $c \leftarrow (\texttt{addr}_{\texttt{pk}}, v, p, r, s, \texttt{cm})$
7. Faça $\texttt{tx}_{\texttt{Mint}} \leftarrow (\texttt{cm}, v, k, s)$
8. Retorne $(c, \texttt{tx}_{\texttt{Mint}})$

O valor de c é conhecido por quem criou a moeda e não deve ser revelado para preservar a privacidade dos envolvidos. A transação $\texttt{tx}_{\texttt{Mint}}$ é transmitida na rede P2P e registrada na lista ligada.

Ressalta-se que uma implementação do protocolo terá suas próprias restrições extras para auferir a validade de uma transação `Mint` e evitar inflação da moeda. Estas regras definirão em que circunstâncias é legítimo a criação de novas moedas. Um exemplo seria que a criação de moedas é a recompensa obtida por minerar um bloco válido usando PoW, assim como no Bitcoin.

Pour

A transação $\texttt{tx}_{\texttt{Pour}}$ (ver seção 3.2.5) recebe como entrada:

- parâmetros públicos `pp`
- raiz Merkle `rt`
- moedas não gastas $c_1^{\texttt{old}}, c_2^{\texttt{old}}$
- chaves secretas das moedas $\texttt{addr}_{\texttt{sk},1}^{\texttt{old}}, \texttt{addr}_{\texttt{sk},2}^{\texttt{old}}$
- caminhos de autenticação $\texttt{path}_1, \texttt{path}_2$ correspondentes à $\texttt{cm}_1^{\texttt{old}}, \texttt{cm}_2^{\texttt{old}}$
- valores das novas moedas $v_1^{\texttt{new}}, v_2^{\texttt{new}}$,
- endereços públicos dos recipientes $\texttt{addr}_{\texttt{pk},1}^{\texttt{new}}, \texttt{addr}_{\texttt{pk},2}^{\texttt{new}}$,
- valor público $v_{\texttt{pub}}$
- string `info`.

A transação consiste das seguintes etapas:

1. Para $i \in \{1, 2\}$
   (a) Faça $\texttt{addr}_{\texttt{pk},i}^{\texttt{old}}, v_i^{\texttt{old}}, p_i^{\texttt{old}}, r_i^{\texttt{old}}, s_i^{\texttt{old}}, \texttt{cm}_i^{\texttt{old}} \leftarrow c_i^{\texttt{old}}$
   (b) Faça $a_{\texttt{sk},i}^{\texttt{old}}, \texttt{sk}_{\texttt{enc},i}^{\texttt{old}} \leftarrow \texttt{addr}_{\texttt{sk},i}^{\texttt{old}}$

(c) Faça $a^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i} \leftarrow \mathtt{addr}^{\mathsf{new}}_{\mathsf{pk},i}$

(d) Compute $\mathtt{sn}^{\mathsf{old}}_i \leftarrow \mathtt{PRF}^{\mathtt{sn}}_{a^{\mathsf{old}}_{\mathsf{pk},i}}(p^{\mathsf{old}}_i)$

(e) Gere $p^{\mathsf{new}}_i \leftarrow \mathtt{PRF}^{\mathtt{sn}}$

(f) Gere aleatoriamente máscaras de *commitment* $r^{\mathsf{new}}_i, s^{\mathsf{new}}_i$

(g) Compute $k^{\mathsf{new}}_i \leftarrow \mathtt{COM}_{r^{\mathsf{new}}_i}(a^{\mathsf{new}}_{\mathsf{pk},i} || p^{\mathsf{new}}_i)$

(h) Compute $\mathtt{cm}^{\mathsf{new}}_i \leftarrow \mathtt{COM}_{s^{\mathsf{new}}_i}(v^{\mathsf{new}}_i || k^{\mathsf{new}}_i)$

(i) Faça $c^{\mathsf{new}}_i \leftarrow (\mathtt{addr}^{\mathsf{new}}_{\mathsf{pk},i}, v^{\mathsf{new}}_i, p^{\mathsf{new}}_i, r^{\mathsf{new}}_i, s^{\mathsf{new}}_i, \mathtt{cm}^{\mathsf{new}}_i)$

(j) Faça $C_i \leftarrow \mathcal{E}_{\mathsf{enc}}(\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}, (v^{\mathsf{new}}_i, p^{\mathsf{new}}_i, r^{\mathsf{new}}_i, s^{\mathsf{new}}_i))$

(k) Gere $\mathsf{pk}_{\mathtt{sig}}, \mathsf{sk}_{\mathtt{sig}} \leftarrow K_{\mathtt{sig}}(\mathsf{pp}_{\mathtt{sig}})$

2. Compute $h_{\mathtt{sig}} \leftarrow \mathtt{CRH}(\mathsf{pk}_{\mathtt{sig}})$
3. Compute $h_1 \leftarrow \mathtt{PRF}^{\mathsf{pk}}_{a^{\mathsf{old}}_{\mathsf{sk},1}}(1 || h_{\mathtt{sig}})$
4. Compute $h_2 \leftarrow \mathtt{PRF}^{\mathsf{pk}}_{a^{\mathsf{old}}_{\mathsf{sk},2}}(2 || h_{\mathtt{sig}})$
5. Faça $x \leftarrow (\mathtt{rt}, \mathtt{sn}^{\mathsf{old}}_1, \mathtt{sn}^{\mathsf{old}}_2, \mathtt{cm}^{\mathsf{new}}_1, \mathtt{cm}^{\mathsf{new}}_2, v_{\mathsf{pub}}, h_{\mathtt{sig}}, h_1, h_2)$
6. Faça $a \leftarrow (\mathtt{path}_1, \mathtt{path}_2, c^{\mathsf{old}}_1, c^{\mathsf{old}}_2, \mathtt{addr}^{\mathsf{old}}_{\mathsf{sk},1}, \mathtt{addr}^{\mathsf{old}}_{\mathsf{sk},2}, c^{\mathsf{new}}_1, c^{\mathsf{new}}_2)$
7. Compute $\pi_{\mathtt{POUR}} \leftarrow \mathtt{Prove}(\mathsf{pk}_{\mathtt{POUR}}, x, a)$
8. Faça $m \leftarrow (x, \pi_{\mathtt{POUR}}, \mathtt{info}, C_1, C_2)$
9. Compute $\sigma \leftarrow S_{\mathtt{sig}}(\mathsf{sk}_{\mathtt{sig}}, m)$
10. Faça $\mathtt{tx}_{\mathtt{Pour}} \leftarrow \mathtt{rt}, \mathtt{sn}^{\mathsf{old}}_1, \mathtt{sn}^{\mathsf{old}}_2, \mathtt{cm}^{\mathsf{new}}_1, \mathtt{cm}^{\mathsf{new}}_2, v_{pub}, \mathtt{info}, *$
    onde $* = (\mathsf{pk}_{\mathtt{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, C_1, C_2, \sigma)$
11. Retorne $(c_1, c_2, \mathtt{tx}_{\mathtt{Pour}})$

Alguns remarques sobre as etapas:

- *Faça* significa atribuição de variável
- *Gere* significa geração de um valor aleatório
- *Compute* significa computação de um valor determinístico
- C_i é o texto cifrado com a chave pública do destinatário i
- $(\mathsf{pk}_{\mathtt{sig}}, \mathsf{sk}_{\mathtt{sig}})$ é uma chave assimétrica de uso único usada para produzir uma assinatura digital
- $h_{\mathtt{sig}}$ é o hash da chave pública de assinatura $\mathsf{pk}_{\mathtt{sig}}$, usado na autentificação desta chave
- h_i é um valor pseudo-aleatório, determinístico, usado para autentificar o dono de c^{old}_i que sabe o segredo $\mathtt{addr}^{\mathsf{old}}_{\mathsf{sk},i}$
- $\pi_{\mathtt{POUR}}$ é uma prova que $x \in L_{C_{\mathtt{POUR}}}$

- m é a mensagem a ser enviada
- σ é a assinatura sobre m usando $\mathtt{sk}_{\mathtt{sig}}$
- $*$ são valores específicos para esta implementação [3]do algoritmo `Pour`

Outro remarque importante: o texto cifrado C_i é usado para comunicar os valores $(v_i^{\mathsf{new}}, p_i^{\mathsf{new}}, r_i^{\mathsf{new}}, s_i^{\mathsf{new}})$. Estes valores são necessários para derivar o número serial do *commitment* da nova moeda. Sem saber o número serial, quem recebe a nova moeda não consegue gastá-la (mas saber o número serial não é suficiente, pois precisa saber também a chave secreta do endereço). Por isso, é necessário comunicar estes valores de forma privada, para que o destinatário saiba dos valores mas terceiros não consigam identificar sua origem. O destinatário pode então usar sua chave privada para decifrar o texto cifrado.

VerifyTransaction

O algoritmo `VerifyTransaction` é usado pelos participantes do protocolo para verificar se uma transação é válida.

Recebe como entrada:

- Parâmetros públicos `pp`
- Transação `tx` do tipo `Mint` ou `Pour`
- Lista ligada L com registro de transações passadas

Consiste das etapas:

1. Se tx é do tipo `Mint`, faça:
   (a) Faça $\mathtt{cm}, v, * \leftarrow \mathtt{tx}$
   (b) Faça $k, s \leftarrow *$
   (c) Faça $\mathtt{cm}' \leftarrow \mathtt{COM}_s(v||k)$
   (d) Retorne $\mathtt{cm}' == \mathtt{cm}$
2. Se tx é do tipo `Pour`, faça:
   (a) Faça $\mathtt{rt}, \mathtt{sn}_1^{\mathsf{old}}, \mathtt{sn}_2^{\mathsf{old}}, \mathtt{cm}_1^{\mathsf{new}}, \mathtt{cm}_2^{\mathsf{new}}, v_{pub}, \mathtt{info}, * \leftarrow \mathtt{tx}_{\mathtt{Pour}}$
   (b) Faça $\mathtt{pk}_{\mathtt{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, C_1, C_2, \sigma \leftarrow *$
   (c) Retorne 0 se $(\mathtt{sn}_1^{\mathsf{old}} \in L) \vee (\mathtt{sn}_2^{\mathsf{old}} \in L) \vee (\mathtt{sn}_1^{\mathsf{old}} == \mathtt{sn}_2^{\mathsf{old}}) \vee (\mathtt{rt} \notin L)$
   (d) Compute $h_{\mathtt{sig}} = \mathtt{CRH}(\mathtt{pk}_{\mathtt{sig}})$
   (e) Faça $x \leftarrow (\mathtt{rt}, \mathtt{sn}_1^{\mathsf{old}}, \mathtt{sn}_2^{\mathsf{old}}, \mathtt{cm}_1^{\mathsf{new}}, \mathtt{cm}_2^{\mathsf{new}}, v_{\mathtt{pub}}, h_{\mathtt{sig}}, h_1, h_2)$
   (f) Faça $m \leftarrow (x, \pi_{\mathtt{POUR}}, \mathtt{info}, C_1, C_2)$
   (g) Retorne $V_{\mathtt{sig}}(\mathtt{pk}_{\mathtt{sig}}, m, \sigma) \wedge \mathtt{Verify}(\mathtt{vk}_{\mathtt{POUR}}, x, \pi_{\mathtt{POUR}})$

Receive

O algoritmo `Receive` é usado por um usuário para identificar quais moedas foram recebidas (mas não foram gastas ainda) em seu endereço público $\mathtt{addr}_{\mathsf{pk}}$. Desta forma, ele consegue saber o saldo associado ao seu endereço.

Recebe como entrada:

- Parâmetros públicos `pp`
- Endereço do recipiente $\mathtt{addr} = (\mathtt{addr}_{\mathsf{pk}}, \mathtt{addr}_{\mathsf{sk}})$
- Lista ligada com registro das transações passadas

Consiste das etapas:

1. Faça $a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}} \leftarrow \mathtt{addr}_{\mathsf{pk}}$
2. Faça $a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}} \leftarrow \mathtt{addr}_{\mathsf{sk}}$
3. Faça $\mathtt{rc} = \{\}$, conjunto de moedas recebidas
4. Para cada transação $\mathtt{tx}_{\mathtt{Pour}}$ em L, faça:
   (a) Faça $\mathtt{rt}, \mathtt{sn}_1^{\mathtt{old}}, \mathtt{sn}_2^{\mathtt{old}}, \mathtt{cm}_1^{\mathtt{new}}, \mathtt{cm}_2^{\mathtt{new}}, v_{pub}, \mathtt{info}, * \leftarrow \mathtt{tx}_{\mathtt{Pour}}$
   (b) Faça $\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, C_1, C_2, \sigma \leftarrow *$
   (c) Para $i \in \{1, 2\}$, faça:
      i. Compute $(v_i, p_i, s_i, r_i) \leftarrow D_{\mathsf{enc}}(\mathsf{sk}_{\mathsf{enc}}, C_i)$
      ii. Se D_{enc} falhar, proceda para a próxima iteração
      iii. Compute $\mathtt{cm} = \mathtt{COM}_{s_i}(v_i \| \mathtt{COM}_{r_i}(a_{\mathsf{pk}} \| p_i))$
      iv. Se $\mathtt{cm} \neq \mathtt{cm}_i^{\mathtt{new}}$ proceda para a próxima iteração
      v. Compute $\mathtt{sn} = \mathtt{PRF}_{a_{\mathsf{sk}}}^{\mathtt{sn}}(p_i)$
      vi. Se `sn` aparece em L (já foi gasto), proceda para a próxima iteração
      vii. Faça $c = (\mathtt{addr}_{\mathsf{pk}}, v_i, p_i, s_i, r_i)$
      viii. Faça $\mathtt{rc} = rc + \{c\}$
5. Retorne `rc`

3.2.7 Segurança

Para uma implementação de um DAP ser segura, ela deve satisfazer as seguintes propriedades [Sas+14, p. 17]:

Indistiguibilidade da Lista ligada: refere-se à propriedade de impedir um adversário de extrair informações da lista ligada além daquelas que são públicas, constituindo assim um sistema *zero-knowledge*. Esta propriedade é simulada pelo seguinte experimento:

Dado duas listas ligadas L_1 e L_2, correspondentes à duas instâncias do mesmo protocolo, um adversário efetua as mesmas transações (`Mint`, `Pour`) nos dois sistemas via dois oráculos O_A e O_B, cada qual responsável por atualizar L_1 ou L_2 (mas sem o adversário saber inicialmente qual oráculo corresponde a qual lista). Neste cenário, o adversário deveria ser incapaz de distinguir as listas ligada apresentados por O_A e O_B. Isto é, após efetuar um número arbitrário de transações, as quais seriam registradas ambas em L_1 e L_2, o adversário não conseguiria usar as informações apresentadas por O_A e por O_B para decidir se a lista ligada de O_A é L_1 ou L_2. Sua expectativa de adivinhar corretamente é 50%, o mesmo que jogar cara-coroa com uma moeda honesta.

Transações incorruptíveis: refere-se à propriedade de que as transações efetuadas por participantes honestos do protocolo são incluídas na lista ligada somente se elas são íntegras, se não foram corrompidas de alguma forma. Um adversário não conseguiria interceptar uma transação legítima, modificá-la antes que a transação original fosse propagada para outros membros do protocolo, e então fazer com que a transação alterada fosse registrada na lista ligada.

Preservação do saldo: refere-se à propriedade de que os usuários do sistema podem apenas gastar um saldo equivalente ao saldo que receberam ou criaram, descontado o saldo que gastaram. Assim, não seria possível para um adversário criar moedas de forma fraudada, efetuar gasto-duplo, ou gastar dinheiro alheio.

3.3 Zcash

Zcash é um sistema construído com base no Zerocash mas com algumas modificações. No Zcash, existe o conceito de *chain value pool*, tratando-se do conjunto de transações e moedas que obedecem regras de um protocolo específico. Assim, dentro do mesmo sistema coexistem protocolos diferentes para transações monetárias. Há quatro *chain value pools*: *Transparente*, *Sprout*, *Sapling* e *Orchard*. O subsistema *Transparente* funciona como o Bitcoin, pois Zcash é um *fork* do código fonte do Bitcoin e aproveita parte de sua base de código. Os outros três subsistemas são chamados de *shielded protocols* (Protocolos Blindados, em tradução livre) porque atuam como um escudo de proteção da privacidade dos usuários.

As moedas dentro dos Protocolos Blindados são transferidas por meio de *notas*, uma estrutura de dados presentes na transação, específica do protocolo, que contém informações sobre a quantia (ocultada por *commitments*) e sobre o endereço de pagamento (destino do saldo). Para cada *nota*, existe um *commitment de nota*. Este *commitment* é incluído em uma árvore de *commitments* após a transação correspondente ser *minerada*. A posição na árvore é denominada *posição de nota* e é usada nas provas zero-knowledge no momento de gastar o saldo. Cada *nota* possui um *nulificador* único, que só pode ser computado conhecendo-se a chave privada do endereço com posso dos fundos monetários. Assim, uma nota (*note*) em Zcash é o equivalente de uma moeda (*coin*) no Zerocash e o nulificador é equivalente ao número serial.

Não é computacionalmente viável associar o *commitment* de nota com sua posição de nota ou com o seu nulificador exceto com o conhecimento da chave privada. Ao gastar-se o saldo, revela-se o nulificador junto com uma prova zero-knowledge que este nulificador

corresponde à algum *commitment* em alguma posição na árvore de *commitments*, sem revelar estes valores. Assim, não é possível saber a origem dos fundos da transação e sua legitimidade é comprovada. O nulificador é então inserido em uma árvore de nulificadores. Se o nulificador de uma transação estiver nesta árvore, então a transação é descartada pelos nós da rede, pois indica que as moedas já foram gastas antes.

Alguns dados da transação são públicos, pois são informações necessárias para as provas de zero-knowledge. Entretanto, estes dados públicos não revelam informações sobre o destino ou valor do pagamento. Os dados contendo o endereço de destino do pagamento e seu valor são encriptados com a chave pública de encriptação do recipiente. Apenas o recipiente é capaz de decriptar o texto cifrado, pois detém a chave privada correspondente, chamada de *View Key* por ser a chave necessária para identificar um pagamento recebido. Após receber o pagamento, o recipiente poderá gasta-ló com sua chave de gasto, *Spend Key*, que é uma chave privada diferente da *View Key*. Assim, além de esconder a origem das moedas, o destino e o valor também são ocultados e as transações são legitimadas pela criptografia assimétrica (que usa Curvas Elípticas).

Desta forma, transações dentro dos Protocolos Blindados são privadas, mas fora deles não. Uma moeda em Zcash pode estar apenas em uma *chain value pool* em um dado momento e é possível transferir moedas entre as *pools* a qualquer momento caso o usuário queira. Transações dentro dos Protocolos Blindados ocorrem sem revelar as quantias envolvidas, porém transações entre *pools* sempre revelam o valor transferido porque os nós da rede precisam validar a preservação de balança de saldo e não é possível usar esquemas zero-knowledge neste caso porque cada *pool* tem seu próprio esquema e regras diferentes.

Por fim, cada Protocolo Blindado implementa um esquema próprio de endereços de pagamentos, ilustrados na seguinte figura e explicados nas respectivas seções:

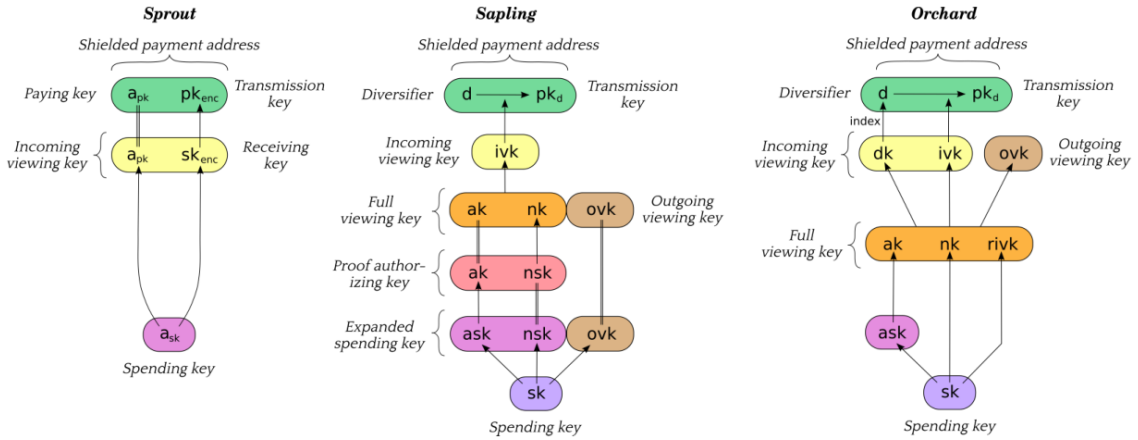

Figura 3.4: *Esquemas de Endereços de Pagamentos em Zcash*
Créditos: Zcash Protocol Specification

3.4 Zcash Sprout

`Sprout` é uma implementação concreta do *Zerocash*, mas com ligeiras modificações. A principal diferença é que as transações `Mint` e `Pour` foram substituídas por uma transação `JoinSplit` que é capaz de encapsular a funcionalidade das duas. Primeiro será discorrido

sobre implementação concreta dos principais componentes do sistema, e então sobre `JoinSplit`, que é a peça central do `Sprout`.

3.4.1 PRF

Funções pseudo-aleatórias (Pseudo Random Functions, `PRF`) são usadas em diversos componentes do Zcash, tais como endereços de pagamentos, assinaturas digitais, códigos de autentificação de mensagem e geração de *nonces*. O `Sprout` utiliza as seguintes funções para diferente propósitos [Hop+22, p. 84]:

$$\begin{aligned}
\mathtt{PRF}^{\mathtt{addr}}_{x}(t) &= \mathtt{sha256}(\mathtt{1100}|x_{[252]}|t_{[8]}|\mathtt{0}_{[248]}) \\
\mathtt{PRF}^{\mathtt{nf}}_{a_{\mathtt{sk}}}(\rho) &= \mathtt{sha256}(\mathtt{1110}|a_{\mathtt{sk}[252]}|\rho_{[256]}) \\
\mathtt{PRF}^{\mathtt{pk}}_{a_{\mathtt{sk}}}(i, h_{\mathtt{Sig}}) &= \mathtt{sha256}(\mathtt{1}\hat{i}\mathtt{00}|a_{\mathtt{sk}[252]}|h_{\mathtt{Sig},[256]}) \\
\mathtt{PRF}^{\rho}_{\varphi}(i, h_{\mathtt{Sig}}) &= \mathtt{sha256}(\mathtt{1}\hat{i}\mathtt{10}|\varphi_{[252]}|h_{\mathtt{Sig},[256]})
\end{aligned}$$

Onde:

1. $\mathtt{PRF}^{\mathtt{addr}}$ é usada na geração de endereços de pagamento.
2. $\mathtt{PRF}^{\mathtt{nf}}$ é usada na derivação de *nuilificadores*.
3. $\mathtt{PRF}^{\mathtt{pk}}$ é usada na geração de códigos de autentificação de mensagens.
4. x denota uma semente aleatória
5. $a_{\mathtt{sk}}$ denota a chave secreta do endereço (ver seção 3.4.2)
6. ρ denota um valor pseudo-aleatório (ver seção 3.4.3)
7. O subscrito $_{[n]}$ indica que são usados os n primeiros bits.
8. $\mathtt{0}_{[248]}$ é uma sequência de 248 bits com valor 0
9. i é o índice da moeda na transação (pode ser 1 ou 2)[4]
10. $\hat{i} = i - 1$ é um único bit (pode ser 0 ou 1)

3.4.2 Endereços

Os endereços de pagamentos são usados para receber moedas e especificam o recipiente do pagamento. O usuário gera um ou mais endereços e então compartilha eles com o remetente. Qualquer um que saiba o endereço poderá enviar moedas para o usuário.

Definições

Seja $l_{a_{\mathtt{sk}}} = 252$ o comprimento da chave privada em bits [Hop+22, p. 72].

[4] `Sprout` suporte até 2 entradas e 2 saídas, assim como Zerocash

Seja $q_{\mathtt{Base}} = 9$ uma sequência de bytes codificando o ponto base na curva 25519 usado no protocolo.

Seja `FormatPrivate` uma função de formatação de chave privada definida como:

$$\mathtt{FormatPrivate}(x) = \mathtt{clamp}_{\mathtt{Curve25519}}(x)$$

Onde $\mathtt{clamp}_{\mathtt{Curve25519}}$ é uma função com entrada de 32 bytes que retorna uma sequência de bytes representando uma chave privada sob a curva 25519 [5].

Seja `DerivePublic` uma função de derivação de chave pública definida como [Hop+22, p. 86]:

$$\mathtt{DerivePublic}(n, q) = \mathtt{Curve25519}(n, q)$$

Onde $\mathtt{Curve25519}(n, q)$ é uma função que computa a multiplicação por n do ponto representado por q na curva 25519.

Geração de endereço

Um endereço de pagamentos é gerado da seguinte forma:

1. Gere aleatoriamente a_{sk} com l_{sk} bits.
2. Compute $a_{\mathsf{pk}} \leftarrow \mathtt{PRF}_x^{\mathtt{addr}}(0)$
3. Compute $\mathsf{sk}_{\mathsf{enc}} \leftarrow \mathtt{FormatPrivate}(\mathtt{PRF}_{a_{\mathsf{sk}}}^{\mathtt{addr}}(1))$
4. Compute $\mathsf{pk}_{\mathsf{enc}} \leftarrow \mathtt{DerivePublic}(\mathsf{sk}_{\mathsf{enc}}, q_{\mathtt{Base}})$

Onde:

- a_{pk} é a chave pública para receber saldo
- a_{sk} é a chave privada correspondente para gastar os saldos
- $\mathsf{pk}_{\mathsf{enc}}$ é a chave pública usada por remetentes para encriptar informações na transação
- $\mathsf{sk}_{\mathsf{enc}}$ é a chave privada correspondente usada pelo recipiente para decriptar as informações enviadas.

Múltiplos endereços

Um usuário pode receber pagamentos de múltiplos remetentes usando apenas um único endereço, sem que estes remetentes ou terceiros saibam que ele está recebendo estes pagamentos, pois o endereço é encriptado na transação. Entretanto, nada impede que os remetentes entrem em coluio e comparem o endereço que lhes foi fornecido pelo usuário, deduzindo assim que se trata da mesma pessoa. Para prevenir isso, o usuário poderia criar um endereço diferente para cada remetente [Hop+22, p. 13].

[5] para detalhes, ver [Hop+22, p. 86].

3.4.3 Notas

As notas em Zcash são uma espécie de *containers* de dados, estruturas cujo propósito é armazenar e transmitir as informações necessárias para efetuar as transferências de valor dentro da rede monetária Zcash.

No `Sprout`, uma nota `n` é uma tupla $(a_{\mathtt{pk}}, v, \rho, \mathtt{rcm})$, onde [Hop+22, p. 13]:

- $a_{\mathtt{pk}}$ é a chave do endereço de pagamento do recipiente.
- v é um inteiro representando o valor da nota em moedas.
- ρ é um valor usado para derivar o nulificador da nota.
- `rcm` (*random commitment trapdoor*) é a máscara usada no *comitment*.

O valor ρ e `rcm` são valores aleatórios gerados pelo remetente da moeda e usados para ofuscar os demais valores. Estes dados são encriptados (ver seção 3.4.7) e transmitidos para o recipiente, o qual poderá usa-los para gastar seus saldos.

Commitment de notas

Como explicado em 3.3, as transações de Protocolos Blindados são efetuadas por meio de *commitments*. O *commitment* de uma nota *n* é calculado seguinte forma [Hop+22, p. 93]:

$$\mathtt{cm} = \mathtt{NoteCommitment(n)} = \mathtt{sha256}(\mathtt{10110000}|a_{\mathtt{pk}}|v|\rho|\mathtt{rcm})$$

Nulificador da nota

O nulificador de uma nota é similar ao papel do número serial em Zerocash. Após ser revelado, é adicionado a um conjunto de nulificadores, associados à uma estrutura de dados com informações sobre o estado de Árvores Merkle [Hop+22, p. 20]. Se o nulificador já apareceu antes, então trata-se de um *gasto duplo*. Se o nulificador não está no conjunto, então é verificado se o *commitment* associado com o nulificador está em alguma Árvore Merkle que armazena *commitments* [Hop+22, p. 19]. Se, e somente se, estiver na árvore especificada, então a transação é legítima.

Seja $\mathtt{PRF}^{\mathtt{nf}}_{x}$ e $\mathtt{PRF}^{\rho}_{\varphi}$ como definidas em 3.4.1.

O nulificador `nf` de uma nota é calculado da seguinte forma [Hop+22, p. 57]:

$$\mathtt{nf} = \mathtt{PRF}^{\mathtt{nf}}_{a_{\mathtt{sk}}}(\rho)$$

O valor ρ, por sua vez, é calculado como [Hop+22, p. 42]:

$$\rho_i = \mathtt{PRF}^{\rho}_{\varphi}(i, h_{\mathtt{Sig}})$$

Onde i é o índice da moeda, $h_{\mathtt{Sig}}$ é o hash da assinatura (ver seção 3.4.4), e φ é uma semente para o gerador de números pseudo-aleatórias gerada pelo remetente.

3.4.4 Hashing

BLAKE2

Zcash usa duas família de algoritmos de hashing: *BLAKE2* e *BLAKE2b*. São usadas para derivar novos algoritmos de hashing, geralmente por meio de um prefixo ou sufixo concatenado na entrada. Dentre os algoritmos estão `hSigCHR` e `KDF`, discutidos a seguir.

Assumindo que os algoritmos originais são criptograficamente seguros, os algoritmos derivados herdam as propriedades de segurança do algoritmo original. É evidente que se o algoritmo original é resistente à colisão, resistente à ataques pré-imagem, concatenar a entrada com prefixo não vai mudar esta propriedade (do contrário, não serie resistente aos ataques), mas irá mudar a saída, configurando assim uma nova função de hashing.

hSigCHR

A função de hashing denotada por `hSigCHR` para gerar um *blueprint* sobre a chave pública da assinatura digital de uma transferência `JoinSplit` [Hop+22, p. 22].

Assinaturas digitais são usadas para garantir a integridade. A assinatura digital sobre a transação somente pode ser gerada após a transação ter sido construída. Construir a transação significa codificar corretamente os dados necessários para a transação em um formato padronizado. Os dados presentes na estrutura da transação são usados em operações criptográficas e uma delas é a verificação da integridade da transação.

Portanto, a própria transação, sobre a qual a assinatura deve ser gerada, deve conter dados que previnam a adulteração da transação (do contrário, um atacante poderia copiar dados da transação, modificá-los, e gerar uma transação assinada pela sua própria chave). Como a transação não pode conter a assinatura digital, pois a assinatura não pode assinar a si mesmo, ela contém o hash $h_{\mathtt{Sig}}$ da chave pública `pk` usada na geração da assinatura digital `JoinSplitSig`.

Ela é definida como [Hop+22, p. 75]:

$$h_{\mathtt{Sig}} = \mathtt{hSigCHR}(s, \mathtt{nf}_1^{\mathtt{old}}, \ldots, \mathtt{nf}_{N^{\mathtt{old}}}^{\mathtt{old}}, \mathtt{pk}) = \mathtt{BLAKE2b\text{-}256}(\text{“}\mathtt{ZcashComputehSig}\text{”}, \mathtt{hSigInput})$$

Onde:

- $h_{\mathtt{Sig}}$ é o valor retornado pela função
- $\mathtt{hSigInput} = s|\mathtt{nf}_1^{\mathtt{old}}|\ldots|\mathtt{nf}_{N^{\mathtt{old}}}^{\mathtt{old}}|\mathtt{pk}$
- s é uma semente aleatória
- $\mathtt{nf}_i^{\mathtt{old}}$ é o nulificador da *i-ésima* moeda sendo gasta[6]
- $N^{\mathtt{old}}$ é o número total de nulificadores da entrada
- `pk` é a chave pública da assinatura da nota, chamada `JoinSplitPubkey` (ver seção3.4.5)

KDF

Zcash usa uma Função de Derivação de Chave `KDF` (*Key Derivation Function*) baseada no `BLAKE2b-256` para derivar chaves assimétricas no esquema de encriptação de notas (ver 3.4.7).

Esta função é definida como [Hop+22, p. 87]:

$$\texttt{KDF}(i, h_{\texttt{Sig}}, \texttt{sharedSecret}, \texttt{pk}, \texttt{pk}_{\texttt{enc},i}^{\texttt{new}}) = \texttt{BLAKE2b-256}(\texttt{kdftag}, \texttt{kdfinput})$$

Onde:

- $\texttt{kdftag} = \text{“}\texttt{ZcashKDF}\text{”}|(i-1)_8|[0]_{56}$
- $\texttt{kdfinput} = h_{\texttt{Sig}}|\texttt{sharedSecret}|\texttt{pk}|\texttt{pk}_{\texttt{enc},i}^{\texttt{new}}$
- i é um índice (8 bits)
- $h_{\texttt{Sig}}$ é o hash sobre a assinatura (256 bits)
- `sharedSecret` é um segredo compartilhado (256 bits)
- `pk` é uma chave pública gerada para encriptação dos dados (256 bits)
- $\texttt{pk}_{\texttt{enc},i}$ é a chave de encriptação do recipiente de índice i (256 bits)

3.4.5 Assinatura

JoinSplitSig

O protocolo `Sprout` usa um esquema de assinatura chamado `JoinSplitSig`, para assinar as transações `JoinSplit`, mantendo assim sua integridade [Hop+22, p. 88]. Este esquema é na verdade uma variante do *EdDSA* e funciona assim [Ber+11, pp. 5–7]:

Seja:

- M a mensagem a ser assinada
- k uma chave secreta
- b o número de bits de k
- $\mathcal{H}$ uma função de hashing que produz uma saída de $2b$ bits
- B um ponto base público da curva usada
- l a ordem do subgrupo de B

A geração da assinatura consista das etapas:

1. Compute $\mathcal{H}(k)$

[6] A moeda “velha” é consumida para gerar uma moeda “nova”, por isso a notação oficial inclui o *old*.

[6] Corresponde ao $h_{\texttt{sig}}$ do Zerocash visto na seção 3.2.6

2. Faça $h_0, h_1, \ldots, h_{2b-1}$ os bits de $\mathcal{H}(k)$
3. Compute $a = 2^{b-2} + sum_{i=3}^{b-3} 2^i h_i$
4. Compute $A = aB$, a chave pública da assinatura
5. Compute $r = \mathcal{H}(h_b, \ldots, h_{2b-1}, M)$
6. Compute $R = rB$
7. Compute $S = (r + \mathcal{H}(R, A, M)a) \mod l$
8. Faça $\sigma_M = (R, S)$, a assinatura sobre M
9. Retorne σ

A verificação é feita da seguinte maneira:

1. Faça $R, S \leftarrow \sigma$
2. Compute $c = \mathcal{H}(R, S, A)$
3. Se $SB = R + cA$, a assinatura é valida

É trivial ver porque a verificação funciona:

$$\begin{aligned} SB &= (r + \mathcal{H}(R, A, M)a)B \\ &= rB + \mathcal{H}(R, A, M)aB \\ &= R + \mathcal{H}(R, A, M)A \\ &= R + cA \end{aligned}$$

No caso do `JoinSplitSig`, têm-se $b = 256$, $\mathcal{H} =$ `sha512` e a curva usada é a Ed25519 [Hop+22, p. 89].

3.4.6 Tags

A assinatura digital `JoinSplit` sobre a transação autentifica que o remetente tem autorização para gastar os fundos, que estão representados por meio das notas $\mathsf{n}^{\mathsf{old}}$. É também necessário autentificar os recipientes das novas moedas, representados por $\mathsf{n}^{\mathsf{new}}$, para que eles e apenas eles possam mover seus fundos do pagamento recebido.

Para tanto, são usadas *tags*, outro termo para se referir aos Códigos de Autentificação de Mensagem (MAC), uma espécie de assinatura simétrica. Cada tag autentifica a chave pública a_{pk} do recipiente, garantindo que esta é a chave pública que pode ser usada para gastar os novos fundos, não podendo ser outra chave (caso contrário, um adversário poderia tentar um ataque que usa suas próprias chaves para gastar fundos alheios).

A tag h_i da nota $\mathsf{n}_i^{\mathsf{new}}$ é definida como [Hop+22, p. 57]:

$$h_i = \mathsf{PRF}^{\mathsf{pk}}_{a^{\mathsf{old}}_{\mathsf{sk},i}}(i, h_{\mathtt{Sig}})$$

As tags são armazenadas e transmitidas em uma Descrição `JoinSplit`, que são os dados públicos necessários para verificar a prova de *zero-conhecimento* que a transação é legítima (ver seção 3.4.8).

3.4.7 Encriptação

Alguns segredos precisam ser transmitidos para o recipiente de uma transação. Um destes segredos é o conjunto de nulificadores das novas moedas, que são gerados pelo remetente da transação e inicialmente desconhecidos pelo destinatário, o qual precisa saber destes valores para gastar as moedas recebidas.

O remetente da transação poderia transmitir estes segredos por canais externos ao protocolo, tais como email usando PGP. Entretanto, isto exige trabalho adicional por parte do usuário final do protocolo, que teria que gerenciar a transmissão destes segredos. Ademais, isto aumenta a chance de possíveis vazamentos de segredos, visto que a maioria dos usuários não possuirá a expertise necessária para transmitir as informações de maneira segura.

Portanto, estes segredos são transmitidos em uma estrutura de dados na própria transação, encriptados usando a chave pública de transmissão $\mathsf{pk}_{\mathsf{enc}}$ do recipiente.

Texto original

O texto original a ser encriptado consiste de [Hop+22, p. 110]:

1. versão de codificação (1 byte)
2. valor v da transação (8 bytes)
3. valor ρ usado para derivar o nulificador (32 bytes)
4. *commitment* de nota `rcm` (32 bytes)
5. `memo` (512 bytes)

O campo `memo` é um espaço que o recipiente pode usar para inserir dados arbitrários, tais como um identificador de pagamento ou simplesmente uma mensagem de texto.

Encriptação

Seja `AgreeSharedSecret` uma função que dado um parâmetro público q e outro secreto n, computa um segredo compartilhado, definida como [Hop+22, p. 86]:

$$\texttt{AgreeSharedSecret}(n, q) = \texttt{Curve25519}(n, q)$$

Seja N o número de recipientes da nota.

Seja $\mathsf{pk}_{\mathsf{enc},1}, \ldots, \mathsf{pk}_{\mathsf{enc},N}$ as chaves públicas de encriptação do endereço de cada recipiente.

Seja $\mathsf{np}_1, \ldots, \mathsf{np}_N$ os *textos originais de nota* de cada recipiente.

Seja `encode` um algoritmo de codificação de `np`.

Seja `Encrypt` um algoritmo de encriptação simétrico, definido como o protocolo de encriptação já existente `AEAD_CHACA20_POLY1305` especificado pela `RFC-7539` [Hop+22, p. 86].

A encriptação consiste das etapas [Hop+22, p. 62]:

1. Gere $(\mathsf{pk}, \mathsf{sk})$, com a chave pública `pk` derivada da chave secreta `sk` usando `DerivePublic`
2. Para $i \in \{1, \ldots, N\}$, faça:
   (a) Compute $P_i \leftarrow \mathtt{encode}(\mathsf{np}_i)$
   (b) Compute $\mathtt{sharedSecret} \leftarrow \mathtt{AgreeSharedSecret}(\mathsf{sk}, \mathsf{pk}_{\mathsf{enc},i})$
   (c) Compute $K_i \leftarrow \mathtt{KDF}(i, h_{\mathtt{Sig}}, \mathtt{sharedSecret}, \mathsf{pk}, \mathsf{pk}_{\mathsf{enc},i})$
   (d) Compute $C_i \leftarrow \mathtt{Encript}(K_i, P_i)$
3. Retorne $\{C_1, C_2, \ldots, C_N\}$

Onde, para cada i:

- P_i é o *plaintext*
- K_i é a chave de encriptação
- C_i é o texto cifrado

A encriptação desses dados garante que somente o recipiente sabe que a transação é destinada para si e qual é seu valor.

Decriptação

Seja $a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}}$ conforme definido em 3.4.2.

Seja $\mathtt{cm}_1, \ldots, \mathtt{cm}_N$ os *commitments* de nota de cada moeda gerada pela transação.

Seja `Decrypt` o algoritmo de decriptação correspondente ao `Encrypt`.

O recipiente tentará decriptar o texto cifrado de cada moeda para descobrir se aquela moeda lhe pertence. Caso não consiga, significa que ele não é dono daquela moeda.

O algoritmo funciona assim [Hop+22, p. 63]:

1. Faça $\mathtt{Dec} = \{\}$
2. Para $i \in \{1, 2, \ldots, N\}$:
   (a) Compute $\mathtt{sharedSecret} \leftarrow \mathtt{AgreeSharedSecret}(\mathsf{sk}_{\mathsf{enc}}, \mathsf{pk})$
   (b) Compute $K_i \leftarrow \mathtt{KDF}(i, h_{\mathtt{Sig}}, \mathtt{sharedSecret}, \mathsf{pk}, \mathsf{pk}_{\mathsf{enc}})$
   (c) Compute $\mathtt{dec} \leftarrow \mathtt{DecryptNote}(K_i, C_i, cm_i, a_{\mathsf{pk}})$
   (d) Se `DecryptNote` falhar, prossiga para $i + 1$
   (e) Faça $\mathtt{Dec} = \mathtt{Dec} \cup \{\mathtt{dec}\}$

3. Retorne `Dec`

O algoritmo $\texttt{DecryptNote}(K, C, \texttt{cm}, a_{\texttt{pk}})$ é definido como:

1. Compute $P \leftarrow \texttt{Decrypt}(K, C)$
2. Se `Decrypt` falhar, falhe
3. Compute $\texttt{np} \leftarrow \texttt{decode}(P)$
4. Faça $v, \rho, \texttt{rcm}, \texttt{memo} \leftarrow \texttt{np}$
5. Faça $\texttt{n} = (a_{\texttt{pk}}, v, \rho, \texttt{rcm})$
6. Se $\texttt{cm} \neq \texttt{NoteCommitment}(\texttt{n})$, falhe
7. retorne $(\texttt{n}, \texttt{memo})$

Observe que para decriptar um pagamento é necessário saber $(a_{\texttt{pk}}, \texttt{sk}_{\texttt{enc}})$, a chave *pública* de *gasto* e a chave *privada* de *encriptação*, mas não é necessário saber a chave *privada* de *gasto* $a_{\texttt{sk}}$. O par $(a_{\texttt{pk}}, \texttt{sk}_{\texttt{enc}})$ é chamado de Chave de Visualização (*View Key*), pois permite detectar pagamentos recebidos mas não permite gastá-los (é necessário $a_{\texttt{sk}}$).

Desta forma, é possível compartilhar a chave de visualização com uma entidade de confiança, tal como membro da família ou tesoureiro, o qual pode gerenciar os pagamentos recebidos mas sem a habilidade de gastar os fundos. É também possível armazenar esta chave em um dispositivo de baixa confiança, tal como um celular, enquanto a chave de gasto $a_{\texttt{sk}}$ é mantida em uma máquina mais segura e protegida, tendo assim a conveniência de monitorar os pagamentos no celular e a segurança de gastá-los na outra máquina.

3.4.8 JoinSplit

A transação `JoinSplit` é uma transferência de valor no Protocolos Blindados `Sprout`. Ela pode ser usada para criar moedas na *chain pool Sprout* vindas de outras *chain pool*, equivalente ao `Mint` do Zerocash. Ela pode transferir valor dentro da *chain pool Sprout*, equivalente ao `Pour` do Zerocash. Assim como a transação `Pour`, uma transação `JoinSplit` pode lidar até com 2 entradas e 2 saídas [Hop+22, p. 8].

O `JoinSplit` consiste de uma estrutura de dados que encapsula as informações da transação, chamado `JoinSplit Description`, e de um sistema de provas zero-knowledge para validar a transação, chamado `JoinSplit zk-SNARKs`.

JoinSplit Descriptions

Cada transferência `JoinSplit` inclui uma sequência (possivelmente vazia) de `JoinSplit Descriptions`, que são dados necessários para a validação das entradas [Hop+22, p. 17]. Quando a sequência não é vazia, inclui a codificação de uma chave de validação e de uma assinatura [Hop+22, p. 37].

Uma `JoinSplit Description` é definida como a tupla [Hop+22, p. 38]:

$$(v_{\text{pub}}^{\text{old}}, v_{\text{pub}}^{\text{new}}, \texttt{rt}, \texttt{nf}_{1,\ldots,N^{\text{old}}}^{\text{old}}, \texttt{cm}_{1,\ldots,N^{\text{new}}}^{\text{new}}, \texttt{pk}, \texttt{randomSeed}, h_{1,\ldots,N^{\text{old}}}, \pi_{\texttt{ZKJoinSplit}}, C_{1,\ldots,N^{\text{new}}}^{\text{enc}})$$

Onde:

- $v_{\text{pub}}^{\text{old}}$ é o valor público gasto vindo da *Pool* Transparente.
- $v_{\text{pub}}^{\text{old}}$ é o valor público sendo depositado na *Pool* Transparente.
- `rt` é raiz de uma árvore Merkle representando o estado dos *commitments* e nulificadores de um bloco anterior na cadeia de blocos.
- N^{old} é o número de moedas velhas, sendo gastas.
- N^{new} é o número de moedas novas, sendo criadas.
- $\texttt{nf}_{1,\ldots,N^{\text{old}}}^{\text{old}}$ são os nulificadores de cada moeda gasta
- $\texttt{cm}_{1,\ldots,N^{\text{new}}}^{\text{new}}$ são os *commitments* de cada moeda nova
- `pk` é a chave pública da transação (ver seção 3.4.5)
- `randomSeed` é uma semente aleatória
- $h_{1,\ldots,N^{\text{old}}}$ são tags autentificadoras das chave públicas $a_{\text{pk},i}^{\text{new}}$ (ver 3.4.6)
- $\pi_{\texttt{ZKJoinSplit}}$ é uma prova zero-knowledge da transação `JoinSplit`
- $C_{1,\ldots,N^{\text{new}}}^{\text{enc}}$ são os texto cifrados para os recipientes (ver seção 3.4.7)

A prova $\pi_{\texttt{ZKJoinSplit}}$ é computada pelo remetente da transação usando um sistema de provas definido pelo protocolo. Esta prova não revela valores confidencias. Os únicos valores revelados são aqueles públicos presentes na *Description*, mas estes não têm informações sobre a origem, destino ou valor.

JoinSplit Zk-SNARKs Statement

Conforme visto em 3.2.4, um sistema zk-SNARKs é construído com base em um Circuito Aritmético para um problema NP. Assim, é definido um problema NP correspondente à satisfação das restrições criptográficas impostas em uma transação *JoinSplit*.

Uma instância deste problema x é definida como [Hop+22, p. 57]:

$$(\texttt{rt}, \texttt{nf}_{1,\ldots,N^{\text{old}}}, \texttt{cm}_{1,\ldots,N^{\text{new}}}, v_{\text{pub}}^{\text{old}}, v_{\text{pub}}^{\text{new}}, h_{\texttt{Sig}}, h_{1,\ldots,N^{\text{old}}})$$

Onde:

- `rt` é uma raiz Merkle
- N^{old} é o número de moedas antigas
- N^{new} é o número de moedas novas
- $\texttt{nf}_{1,\ldots,N^{\text{old}}}$ são os nulificadores das moedas antigas

- $v_{\mathsf{pub}}^{\mathsf{old}}$ valor público de moedas gastas advindo da *Pool* transparente
- $v_{\mathsf{pub}}^{\mathsf{new}}$ valor público de moedas novas depositas na *Pool* transparente
- $h_{\mathtt{Sig}}$ o hash sobre a assinatura digital da transação
- $h_{1,\ldots,N^{\mathsf{old}}}$ tags para autentificar as chaves públicas do recipiente, o novo dono do saldo transferido

Uma testemunha a de x é definida como [Hop+22, p. 57]:

$$(\mathtt{path}_{1,\ldots,N^{\mathsf{old}}}, \mathtt{pos}_{1,\ldots,N^{\mathsf{old}}}, \mathsf{n}^{\mathsf{old}}_{1,\ldots,N^{\mathsf{old}}}, a^{\mathsf{old}}_{\mathsf{sk},1,\ldots,N^{\mathsf{old}}}, \mathsf{n}^{\mathsf{new}}_{1,\ldots,N^{\mathsf{new}}}, \varphi, \mathtt{mp}_{1,\ldots,N^{\mathsf{old}}})$$

Onde:

- $(\mathtt{path}_i, \mathtt{pos}_i)$ especifica a posição na Árvore Merkle do *commitment* da i-ésima moeda sendo gasta
- $\mathsf{n}^{\mathsf{old}}_{1,\ldots,N^{\mathsf{old}}}$ são as *notas* das moedas antigas
- $\mathsf{n}^{\mathsf{new}}_{1,\ldots,N^{\mathsf{new}}}$ são as *notas* das moedas novas
- $a^{\mathsf{old}}_{\mathsf{sk},i}$ é a chave privada correspondente à i-ésima moeda gasta
- φ é uma semente privada aleatória [Hop+22, p. 140] utilizada para derivar o valor ρ da nota (ver seção 3.4.3)
- $\mathtt{mp}_i$ indica se é necessário verificar a posição do *commitment* da moeda na Árvore Merkle. É 0 se o valor da i-ésima moeda v_i^{old} for zero e 1 caso contrário[7]

A testemunha a é valida para x se as seguintes condições[8]são satisfeitas:

1. O saldo é preservado:

$$v_{\mathsf{pub}}^{\mathsf{old}} + \sum^{N^{\mathsf{old}}} v_i^{\mathsf{old}} = v_{\mathsf{pub}}^{\mathsf{new}} + \sum^{N^{\mathsf{new}}} v_i^{\mathsf{new}}$$

2. Para cada $i \in 1, \ldots, N^{\mathsf{old}}$:
   (a) Se $v_i^{\mathsf{old}} = 0$, então $\mathtt{mp}_i = 0$.
   (b) Se $v_i^{\mathsf{old}} \neq 0$, então $\mathtt{mp}_i = 1$.
   (c) Se $\mathtt{mp}_i = 1$, então $(\mathtt{path}_i, \mathtt{pos}_i)$ especifica uma posição correta do *commitment* $\mathtt{cm}_i$ na Árvore especificada por `rt`
   (d) `rt` é uma raiz de uma Árvore Merkle que existe
   (e) os nulificadores $\mathtt{nf}_i$ foram computados corretamente (ver seção 3.4.3):

[7] É possível gastar uma moeda que não existia (portanto, que não tem um *commitment* na Árvore Merkle) e cujo valor é zero

[8] A ordem listada não é a ordem exata na qual é feita verificação das exigência (isto não é mencionada na especificação oficial do protocolo), mas a enumeração foi meramente feita para facilitar a explicação de cada requisito.

(f) as chaves públicas $a^{\mathtt{old}}_{\mathtt{pk},i}$ foram computadas corretamente (ver seção 3.4.2):

(g) as tags de autentificação h_i foram computadas corretamente (ver seção 3.4.6):

3. Para cada $i \in 1, \ldots, N^{\mathtt{new}}$:

   (a) os valores $\rho_i^{\mathtt{new}}$ foram computados corretamente (ver seção 3.4.3)

   (b) os *commiments* $\mathtt{cm}_i^{\mathtt{new}}$ foram computados corretamente (ver seção 3.4.3)

Dessa forma:

- O primeiro requisito exige que o saldo seja mantido, impedindo criação arbitrária de dinheiro no sistema.
- Os requisitos 2a-2d exigem que a moeda sendo gasta é uma moeda que existe no registro de transações (e portanto seu *commitment* está em uma Árvore Merkle).
- Os requisitos 2e e 2f garantem a integridade do remetente, visto que apenas ele poderia calcular estes valores usando sua chave privada.
- O requisito 2g garante a autenticidade do remetente.
- O requisito 3a garante que o recipiente poderá derivar os nulificadores para gastar seus fundos.
- O requisito 3b garante que as moedas dos novos fundos têm *commitments* válidos para ser gastos.

O problema NP consiste em, dado um x, encontrar a que satisfaça estas restrições. O usuário do Zcash, detentor das chaves privadas das moedas, consegue computar a e e providenciar uma prova $\pi_{\mathtt{ZKJoinSplit}}$ de seu conhecimento sobre a usando um sistema de provas zk-SNARKs especificado pelas regras do protocolo.

3.4.9 JoinSplit zk-SNARKs proofs

O sistema de provas usado no Protocolo `Sprout` era um zk-SNARKs projetado para uma arquitetura RISC de von Neumann. Este sistema consistia de dois componentes: (1) um subsistema de provas criptográficas para verificar a satisfabilidade de circuitos aritméticos e (2) um gerador de circuitos que transforma programas desta arquitetura em circuitos aritméticos para que possam ser verificados [Ben+13][9].

Este sistema deixou de ser usado no Zcash devido à vulnerabilidades que poderiam ser exploradas para gerar moedas infinitas [Hop+22, p. 108]. Diante disso, Zcash recebeu uma atualização de protocolo chamada *Sapling*, a qual passou a utilizar um novo sistema de provas chamado `Groth16` [Hop+22, p. 33]. Após a atualização de protocolo chamada `Canopy`, o Protocolo Blindado *Sprout* foi depreciado, não sendo mais possível depositar valor na *pool Sprout* [Hop19].

Apesar de ser um protocolo depreciado, é relevante para o trabalho abordar *Sprout* por ser uma implementação concreta do `ZeroCash` e percursor de `Sapling` e `Orchard`, os quais são melhorias deste sistema.

[9] A implementação deste sistema complexo foge o escopo do trabalho, mas pode ser encontrada em [Ben+13].

3.5 Zcash Sapling

Sapling é uma atualização do protocolo da rede Zcash ativada em 2018 e foi a primeira atualização *hard fork*[10]. Esta atualização introduz várias melhorias em termos de privacidade, funcionalidade novas e eficiência em requisitos computacionais.

Sapling introduz mudanças, mas também é semelhante ao Sprout em vários aspectos, usando os mesmos esquemas e conceitos abstratos de como os componentes criptográficos se encaixam para compor o sistema, porém com implementações mais sofisticadas.

Portanto, o foco desse capítulo será nas diferenças conceituais entre Sapling e Sprout, nas melhorias trazidas e nessas novas funcionalidades.

3.5.1 Endereços

O protocolo Sprout utiliza endereços baseados em criptografia assimétrica para especificar o destinatário dos pagamentos. Sapling expande este conceito, adicionando novas funcionalidades e componentes do endereço.

No Sapling, é possível criar múltiplos endereços, chamados *endereços diversificados*. Cada endereço diversificado $\mathtt{pk}_d$ é derivado a partir de um *diversificador d* [11]e de uma chave `ivk`. Apesar de serem endereços completamente diferentes, seus fundos podem ser gastos pela mesma chave secreta [Hop+22, p. 35] e pagamentos recebidos nestes endereços podem ser revelados pelas mesma chave de visualização.

Uma das vantagens deste esquema é evitar o armazenamento e gerenciamento seguro de várias chaves secretas. Outra vantagem é a redução do custo computacional feito pelo recipiente, que deve escanear transações públicas e tentar decriptar o texto cifrado para averiguar se o saldo da transação lhe pertence (ver 3.4.7). Com uma única chave, o recipiente pode efetuar este procedimento apenas uma vez, do contrário teria que repetir o procedimento para cada chave distinta.

Um endereço Sapling é constituído dos seguintes componentes:

- **chave secreta de gasto - `sk`** : chave principal que permite enviar e receber moedas. Dela são derivados todos os outros componentes do endereço. Deve ser obtida gerando-se aleatoriamente uma sequência de 256 bits.
- **chave privada expandida - `(ask, nsk, ovk)`**: tupla de chaves derivadas diretamente de `sk`. Confere todos os privilégios associados aos endereços do usuário.
- **chave privada de autorização - `ask`**: gera as assinaturas digitais autentificando o usuário dono das chaves.

[10] *hardfork* neste contexto significa atualização com mudanças não compatíveis com o protocolo anterior, bifurcando a rede atual em duas redes: (1) rede com nós cujo cliente implementa o *fork* e (2) rede original com nós rodando clientes antigos

[11] a especificação do protocolo recomenda user valores aleatórios para o diversificador para proporcionar anonimato do usuário, mas nada impede o uso de valores arbitrários e repetidos.

- **chave privada nulificadora - `nsk`**: computa os nulificadores necessários para gastar as moedas.
- **chave de visualização de pagamentos enviados - `ovk`**: apenas com esta chave é possível identificar pagamentos enviados para outros, bem como todos os detalhes destes pagamentos.
- **chave pública de prova de autorização - `ak`**: chave pública correspondente à chave secreta `ask`.
- **chave pública nulificadora - `nk`**: chave pública correspondente à chave secreta `nsk`.
- **chave de visualização completa - `fvk`**: tupla de chaves `fvk = (ak,nk,ovk)` que permite visualizar os detalhes de pagamentos recebidos e enviados, bem como gerar endereços diversificados.
- **chave de visualização de pagamentos recebidos - `ivk`**: permite visualizar os detalhes de pagamentos recebidos, bem como gerar endereços diversificados. É computada a partir de `(ak, nk)`

3.5.2 Transferências

No Sprout, temos a transferência `JoinSplit`, a qual inclui uma *Description* com os dados públicos para que todos os nós verifiquem a legitimidade da transação e os dados encriptados para que o recipiente possa reconhecer e gastar os fundos recebidos.

No Sapling, estes dois aspectos são separados em duas transferências: *Spend Transfer* e *Output Transfer*.

Spend Transfer

Uma *Spend Transfer* é uma transferência de valor que gasta uma nota $n^{\mathtt{old}}$. Associada à ela está uma *Spend Description*: informações públicas usados para verificar que o remetente é dono legitimo dos fundos de origem e para criar as moedas de destino usando *commitments* [Hop+22, p. 55]. Uma destas informações é uma assinatura digital da transação sob a chave `ask`.

Output Transfer

Uma *Output Transfer* é uma transferência de valor que cria uma nota $n^{\mathtt{new}}$. Associada à ela está uma *Output Description*: informações encriptadas destinados ao recipiente e necessárias para reivindicar o saldo proveniente do pagamento. Esta descrição não gera moedas novas, apenas transmite os dados necessários para que o recipiente saiba que recebeu as moedas, o qual ele recupera com sua chave de visualização.

3.5.3 Encriptação

Diferente de Sprout, cada *Output description* é encriptada por uma nova chave pública efêmera. Esta chave pública e sua chave privada correspondente são derivadas a partir da

chave pública do endereço diversificado pk_d do recipiente e a partir da *chave de visualização de pagamentos enviados* ovk do remetente. Assim, ambos o remetente e o recipiente conseguem decriptar o texto cifrado.

3.5.4 Sistemas de Provas

Cada transferência *Spend Transfer* e *Output Transfer* possui sua própria prova de zero-conhecimento e sistema zk-SNARKs. Estes sistemas são baseados em um problema NP, *Spend Statement* e *Output Statement* respectivamente, cada qual consiste em satisfazer um conjunto de restrições sobre as transferências para garantir a legitimidade das mesmas [Hop+22, pp. 58–59].

O sistema de provas zk-SNARKs usado no Sapling é denotado por *Groth16* em referência aos seus autores [Hop+22, p. 109]. Este sistema corrigi as vulnerabilidades presentes no sistema anteriormente usado no Sprout. Ademais, este sistema permite provas zero-knowledge menores em tamanho e com menor tempo de verificação comparado com o sistema anterior do Sprout. Assim, diminuindo-se os requisitos computacionais e de armazenamento de disco para os nós integrantes da rede Zcash.

Além de provas zero-knowledge, o Sapling utiliza *Pedersen commitments* para garantir a preservação de saldo, que é feito separadamente ao sistema zk-SNARKs [Hop+22, p. 18]. Assim, além dos *commitments* de nota, cada nota possui associada a si *commitments* de valor que são usados de forma parecida com Monero (ver 2.4.2). Estes *commitments* são referenciados nas descrições de notas *Spend Description* e *Output Description* e a os valores são somados para averiguar a balança.

3.5.5 Outros componentes

Sapling também utiliza diferentes componentes criptográficos. Aqui vale ressaltar alguns deles:

Curva JubJub: curva elíptica projetada para que possa ser implementada de modo eficiente em circuitos zk-SNARKs.

RedDSA: Esquema de assinatura digital baseado na assinatura Schnorr e com suporte opcional para chaves re-aleatorizáveis [Hop+22, p. 90].

RedJubJub: adaptação de *RedDSA* para a curva *JubJub*.

Homomorphic Pedersen commitments: adaptação de *Pedersen commitments* para a curva *JubJub* e usados nos *commitments* de valor.

BLS12-391: outra curva elíptica, usada no sistema *Groth16*.

Estes componentes criptográficos são mais sofisticados do que aqueles usados no Sprout e contribuem para fazer Sapling um protocolo mais eficiente, robusto e seguro.

3.6 Zcash Orchard

Orchard é o terceiro e mais novo Protocolo Blindado de Zcash. Foi ativado na sexta atualização de Zcash chamada NU5 em Maio de 2022 [Hop+22, p. 118].

3.6.1 Motivação

A motivação para a projeção e ativação do Orchard se dá por duas razões principais [Hop+21]:

- **Eficiência e escalabilidade**: apesar de Sapling ter melhorias significantes de eficiência comparado com Sprout, estes dois protocolos não são compatíveis com técnicas modernas de escalabilidade devido à limitações técnicas. Uma destas técnicas é Prova de Zero-Conhecimento Recursiva, que requer ciclo de curvas elípticas e não pode ser usado em Sprout porque não faz uso de EC e nem em Sapling porque sua EC não suporta uma implementação eficiente de ciclo de curva.
- **Configuração de Confiança**: ambos Sprout e Sapling usam um sistema de provas zero-knowledge (Groth16) quer requer a geração de Parâmetros de Sistema para os circuitos aritméticos. Estes parâmetros são gerados usando Computação Multipartidária e cada membro deve deletar permanente dados usados na geração, para segurança do sistema. Neste método, se houver pelo menos um participante não comprometido a estrutura resultante ficará seguro. Entretanto, caso haja coluio, o sistema de provas é comprometido e vulnerável à produção arbitrária de moedas no sistema.

O Orchard visa implementar um Protocolo Blindado construído em cima de ciclo de curva e um sistemas de provas que não requer Configuração de Confiança.

3.6.2 Diferenças com Sapling

O Orchard tem as seguintes diferenças em relação ao Sapling [Hop+21]:

- **Sistema de Provas**: Orchard usa o sistema de provas *Halo2* em vez de *Groth16*.
- **Hash para Ponto**: Orchard usa o algoritmo *SWU simplificado* para definir uma função de Hashing Para Ponto Na Curva, em vez do mecanismo do Sapling baseado no hashing BLAKE2.
- **Curvas Elípticas**: Orchard usa as curvas *Pallas* (no protocolo) e *Vesta* (no circuito aritmético) em vez das curvas *JubJub* e *BLS12-381*.
- ***Commitments***: Orchard usa *Sinsemilla commitments* e *Sinsemilla hashing* em vez de *Pedersen commitments* e *Pedersen hashing*.
- **Assinaturas**: Orchard usa *RedPallas* (*RedDSA* adaptado para a curva *Pallas*) em vez de *RedJubJub* usada no Sapling
- **Transferências**: Orchard possui um único tipo de transferência chamada *Action Transfer* e sua descrição *Action Description*, em vez de duas transferências e descrições como no sapling: *Spend Transfer / Description* e *Output Transfer / Description*

- **Notas e nulificadores**: Orchard usa notas e nulificadores com uma estrutura de dados diferente no que se refere aos tipos de dados, ordem e significado dos parâmetros, e codificação.
- **Endereços e chaves**:
  - Chave Privada Nulificadora `nsk` é removida
  - Chave Pública Nulificadora `nk` é derivada de `sk` em vez de `nsk`
  - Chave de Visualização de Pagamentos Enviados `ovk` é derivada a partir de `fvk` em vez de `sk`
  - Chave de Visualização de Pagamentos Recebidos `ivk` é derivada usando `Sinsemilla commitment` em vez de *BLAKE2*
  - Chave de Aleatorização de Visualização de Pagamentos Recebidos `rivk` é introduzida em `fvk` como elemento fonte de aleatoriedade

3.6.3 Semelhanças com Sapling

O Orchard possui vários aspectos em comum com Sapling, valendo ressaltar os seguintes:

- **endereço**: ambos protocolos permitem o uso de múltiplos endereços diversificados derivados a partir das mesmas chaves secretas.
- **chaves**: ambos protocolos possuem chaves criptográficas com funcionalidade específicas, tais como gerar endereços, detectar pagamentos recebidos, e reconhecer pagamentos passados enviados.
- **nulificadores**: ambos protocolos usam nulificadores para prevenir o gasto duplo e para garantir a anonimato de quem está enviando as moedas.
- **assinaturas**: ambos protocolos implementam esquemas de assinaturas simétricas e assimétricas para autentificar ambos o remetente e o destinatário dos fundos da transação.
- **zk-SNARKs**: ambos protocolos usam sistemas de provas zk-SNARKs. construídos em circuitos aritméticos de problemas NP.
- **notas**: ambos protocolos usam *notas* como estrutura de dados para armazenar e transmitir informações das transferências.
- **memo**: ambos protocolos permitem a adição de 512 bytes dados arbitrários em uma transação que o remetente pode usar para comunicar uma anotação ao remetente.
- **commitments de valor**: ambos protocolos usam *commitments* de valor para provar balanço de saldo.
- **Árvore Merkle**: ambos protocolos usam árvores de Merkle para permitir buscas eficientes de *commiments* de nota e de valor.

Capítulo 4

Comparação de Monero e Zcash

Nesta comparação, não será analisado Zcash Sprout, visto que se trata de um protocolo legado depreciado e os protocolos atuais do Zcash são mais eficientes.

4.1 Funcionalidade

Em termos de funcionalidades relacionadas à privacidade, Monero e Zcash são parecidos:

Funcionalidade	Monero	ZCash Sapling	Zcash Orchard
Ocultação da origem	Sim	Sim	Sim
Ocultação do destino	Sim	Sim	Sim
Ocultação da quantia	Sim	Sim	Sim
Chave de visualização	Sim	Sim	Sim
Visualização *incoming*	Não	Sim	Sim
Visualização *outgoing*	Não	Sim	Sim
Privacidade obrigatória	Sim	Não	Não

Tabela 4.1: *Funcionalidades de privacidade em Monero e Zcash*

Ambos Monero e Zcash tem mecanismos para ocultar a origem de uma transação, o destino, e a quantia. Estes são as três características relevantes para a privacidade do usuário quando abordados do ponto de vista de um observador externo tentando extrair informações dos dados públicos.

Também deve se levar em conta a privacidade dos usuários para com conhecidos de confiança. Um usuário pode desejar delegar o gerenciamento de suas finanças para um profissional financeiro, mas sem lhe dar total autorização dos fundos. Igualmente, uma empresa pode contratar uma firma para administrar pagamentos de determinado setor, e para isso deseja revelar alguns dados mas não todos. Neste sentido, chaves de visualização têm um papel importante na privacidade, pois permitem a revelação seletiva de dados das transações monetárias.

Ambos Monero e Zcash têm suporte para *chave de visualização*, que são chaves que

permitem visualizar pagamentos enviados e recebidos, mas não têm autorização para gastar saldos (ver seções 2.2, e 3.5.1).

Zcash Orchard expande este conceito para a inclusão de uma *chave de visualização incoming*, a qual permite ver os detalhes somente de pagamentos recebidos, e uma *chave de visualização outgoing*, a qual permite ver detalhes somente de pagamentos enviados; juntas elas formam a *chave de visualização completa* (ver seções 3.5.1).

4.2 Tecnologias

Apesar de terem funcionalidades equivalentes, Monero e Zcash empregam diferentes tecnologias na implementação destas funcionalidades:

	Monero	ZCash Sapling	Zcash Orchard
Endereço de pagamento	Stealth Addresses	Diversified Addresses	Diversified Addresses
Ocultação do destino	Stealth Addresses	Encriptação de dados	Encriptação de dados
Ocultação da quantia	Pedersen commitments	Encriptação de dados	Encriptação de dados
Ocultação da origem	Ring Signatures	Groth16 zk-SNARKs	Halo2 zk-SNARKs
Preservação de balança	Pedersen commitments	Pedersen commitments	Sinsemilla commitments
Curva Elíptica	Twisted Edwards Ed25519	JubJub e BLS12-381	Pallas e Vesta
Família de hashing	SHA-3	SHA-2, BLAKE2 e BLAKE2b	SHA-2, BLAKE2, BLAKE2b

Tabela 4.2: *Tecnologias criptográficas em Monero e Zcash*

Nos campos *Ocultação da quantia* e *Ocultação do destino*, o valor *Encriptação de dados* refere-se ao esquema de Zcash que encripta estes valores de forma que somente o recipiente pode decriptar os dados usando sua chave privada (ver seções 3.4.7 e 3.5.3).

Ademais, Zcash usa sistemas zk-SNARKs não apenas para ocultar a origem, mas para provar a legitimidade dos valores ocultos pelo texto cifrado e seu vínculo com o recipiente do pagamento. Assim, em Zcash as provas criptográficas relacionadas à privacidade são feitas solenemente usando zk-SNARKs.

Já no Monero, as provas criptográficas relacionadas à privacidade não utilizam provas de zero-knowledge e fazem uso da combinação de três tecnologias principais: Stealth Addresses, Ring Signatures, e Pedersen commitments junto com *Bullet proofs*.

A escolha da tecnologia usada no sistema tem implicações diretas no nível de anonimato e privacidade dos usuários, nos requisitos de poder computacional e espaço de disco dos nós da rede e, consequentemente, na descentralização e sustentabilidade de rede do sistema como um todo.

4.3 Anonimato

Seja *Anonimity Set* definido como um conjunto de membros candidatos para identificar uma entidade. Esta entidade, que pode ser um indivíduo ou organização de indivíduos, visa manter seu anonimato e para isso utiliza um *Anonimity Set* com n membros (incluindo si mesmo).

Se os membros deste conjunto forem idênticos na perspectiva de um terceiro, então cada membro é igualmente provável de ser a entidade, portanto a probabilidade de a

entidade ser identificada é $1/n$. A entidade possui então *negação plausível*: quando acusada, ela pode alegar não ser um dos membros do *Anonimity Set* com probabilidade $(n-1)/n$. Quando maior e mais uniforme o *Anonimity Set*, maior anonimato é obtido pelos seus membros.

No Monero, o RingCT usa um tamanho de anel fixo igual à 16. Então, seu *Anonimity Set* é 16: um observador terceiro tem probabilidade $1/16 \approx 6.7\%$ de identificar o remetente verdadeiro de uma transação.

Conforme o saldo de uma transação é transferido por meio de outras transações, rastrear a origem dos saldos se torna inviáveis: a probabilidade de identificar o dono original de um saldo após m transações consecutivas é $(1/n)^m = (1/16)^m$. Isto pois trata-se de eventos independentes para um observador externo devido às propriedades criptográficas que ofuscam os vínculos entre transações.

Dessa forma, o *Anonimity Set* de uma sequência de transações do Monero aumenta exponencialmente. Um usuário do Monero pode então efetuar múltiplas transações consecutivas transferindo as moedas para si mesmo, sendo que a cada transferência envolve endereços de uso único e novos *Anonimity Sets*. Após este processo, o usuário terá seu saldo em um novo endereço de sua posse, mas o vínculo com o endereço original será irrisório devido ao número alto de candidatos possíveis para a origem dos fundos.

No Zcash, usa-se provas zero-knowledge para provar que a moeda sendo gasta pertence à um conjunto de moedas não gastas. Este conjunto de moedas não gastas consiste de todas as moedas não gastas de todos os participantes da rede. Portanto, o *Anonimity Set* do Zcash são as moedas de todos os usuários. Este conjunto é muito maior do que o do Monero, visto que há milhares de usuários no Zcash.

De fato, dados públicos analisados [Com22] pela Electric Coin Company, a empresa por trás de Zcash, mostram que no até o ano de 2022 o tamanho do *Anonimity Set* do Sapling é cerca de 1 milhão e do Orchard é 300 mil. Conforme mais transações são efetuadas, estes números vão aumentando.

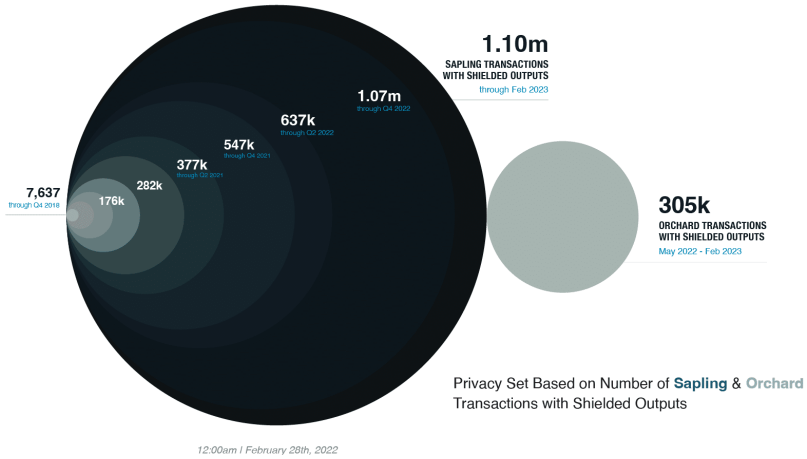

Figura 4.1: *Anonimity set em Zcash*
Créditos: Electric Coin Company

Estes dados são derivados do número de transações que usam o protocolo *Sapling* e o *Orchard*, pois as transações em si são públicas mas os dados dentro das transações são

encriptados e confidenciais.

Ressalta-se o *Anonimity Set* de Zcash inclui apenas os usuários dos Protocolos Blindados. Usuários que usam o Protocolo Transparente não contribuem para o anonimato de outros usuários, visto que não estão usando o protocolo com as funcionalidades de privacidade.

Um estudo de 2020 [Ye+20, p. 8] relata que dentre um período analisado de 30 dias as transações de Protocolos Blindados representavam apenas 0.09% do valor transacionado, que havia 5 vezes mais transações públicas do que transações parcialmente blindadas (transação da *Pool* transparent para uma *Pool* Blindada), e 13 vezes mais transações públicas do que transações completamente blindadas (transação feita dentro de um Protocolo Blindado). Sendo assim, o anonimato no Zcash é dezenas de vezes menor do que poderia ser caso houvesse somente Protocolos Blindados.

De fato, dados oficiais da Electric Coin Company, coletados a partir de dados públicos das transações, mostram que cerca de 1 milhão de moedas estão dentro de *Pools* Blindadas, enquanto o resto está na transparente. Atualmente o número de moedas no Zcash é cerca de 14 milhões [bit23], o que significa que menos de 10% do valor está sendo usado em *Pools* Blindadas.

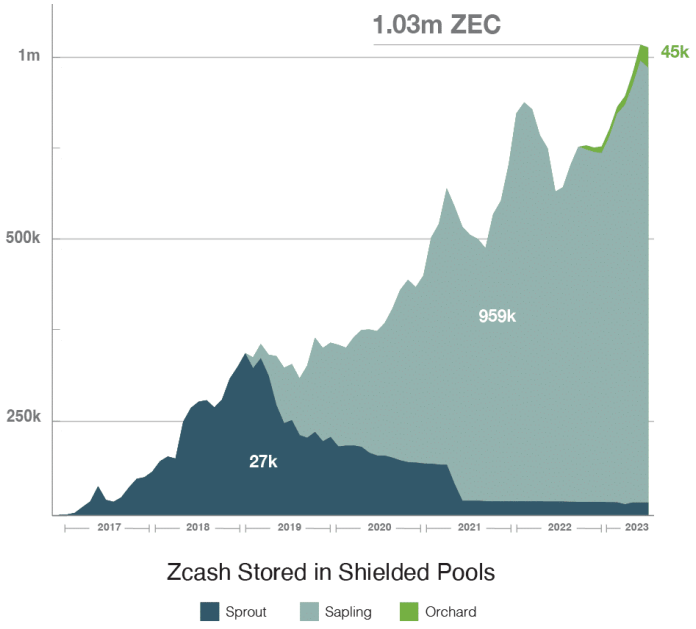

Figura 4.2: *Moedas em Pool Blindados*
Créditos: Electric Coin Company

Os dados do gráfico acima são de 2022 e vão mudar conforme usuários efetuam transações e o ecossistema de Zcash evolui. É possível saber quantas moedas há em cada *Pool* porque as transações entre *Pools diferentes* não são completamente privadas e revelam o valor transferido (ver 3.3).

Com relação ao número de transações usando Protocolos Blindados, dados oficiais da Electric Coin Company também confirmam que apenas uma fração de usuários utiliza os Protocolos Blindados [Com22]. No período de fevereiro de 2023, cerca de 12 mil transações eram totalmente blindadas, 23 mil parcialmente blindadas, e 130 mil totalmente públicas. Mas os dados também mostram que o número de transações blindadas vêm aumentando.

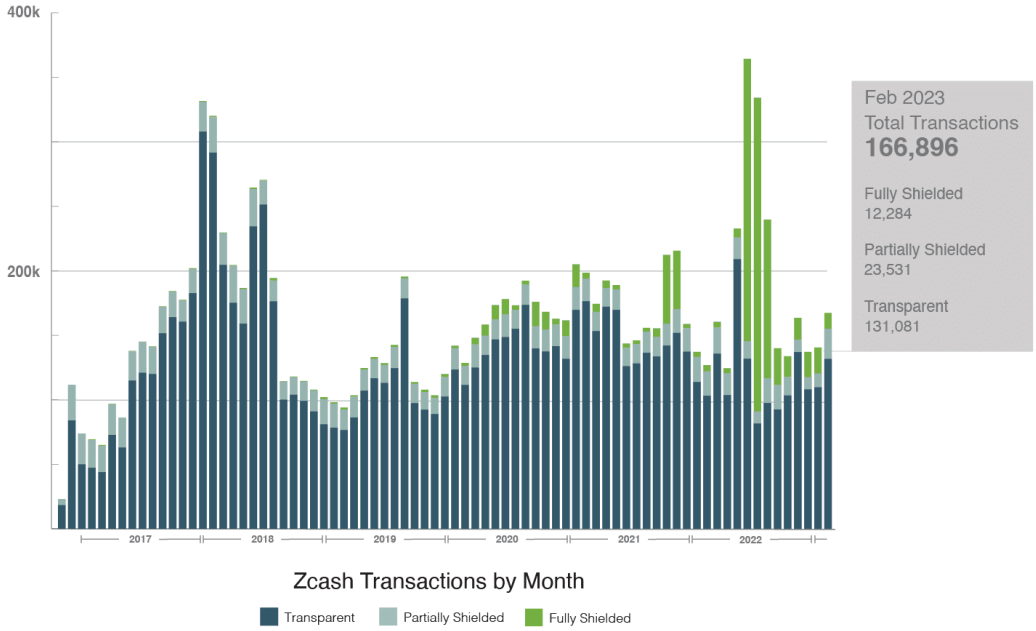

Figura 4.3: *Transações em Zcash por mês*
Créditos: Electric Coin Company

Ao dar a opção para os usuários de não permanecer anonimo, Zcash abre margem para redução do anonimato de usuários que querem ter mais privacidade, que é o que de fato aconteceu e não apenas uma hipótese. Uma possível razão para os usuários optarem pelo protocolo transparente é que é mais fácil conduzir transações neste protocolo do que nos protocolos Blindados, por exigir algumas etapas adicionais por parte do usuário.

A abordagem do Monero é diferente: todos usuários utilizam as funcionalidades de privacidade com os mesmos parâmetros. Dessa forma, Monero cria uma base de usuários uniforme, dificultando sua identificação por um observador externo.

Um estudo de 2020 [Ye+20] concluiu que Monero, com um tamanho de anel fixo e maior uniformidade da base de usuários, atingiu um nível de anonimato maior e heurísticas de rastreamento de usuários tem se tornado menos efetivas. Por outro lado, algumas heurísticas usadas no estudo conseguiriam identificar e agrupar usuários do Zcash por meio da análise dos dados públicos, sendo questionável as garantias de privacidade do sistema. O mesmo estudo afirma que tais problemas poderiam ser resolvidos se os usuários usassem os Protocolos Blindados.

Sendo assim, apesar do Zcash usar tecnologias que podem proporcionar um alto grau de anonimato, as decisões de design do sistema Zcash acabaram por reduzir substancialmente este anonimato. Por outro lado, Monero usa tecnologias que isoladamente proporcionam um menor grau de anonimato comparado com Zcash, mas que devido ao design do sistema, resultam em um elevado nível de anonimato.

4.4 Espaço de disco

No Monero, a transação *RCTTypeBulletproof2* tem os seguintes requerimentos de espaço de disco [NAK20, p. 54]:

- $(2(v+1)+3) \times m \times 32$ bytes: prova *RCTTypeBulletproof2*
- $(2 \times \lceil \log_2(64 \times p) \rceil + 9) \times 32$ bytes: prova *Bulletproof range proof*
- $m \times 32$ bytes: *imagens de chave*
- $p \times 32$ bytes: endereços de uso único de saída
- $p \times 32$ bytes: *commitments* de saída
- $p \times 8$ bytes: quantias codificadas
- 32 bytes: chave pública da transação
- 32 bytes: ID de pagamento opcional arbitrário
- 8 bytes: taxa de transação
- $(v+1) \times m \times 8$ bytes: *offsets* que localizam os membros do anel no registro de dados público
- 8 bytes: campo opcional `Unlock Time`, especifica data mínima a partir da qual as saídas podem ser gastas

Onde:

- v é o número de *decoys*: entradas estrangeiras usadas para ofuscar o verdadeiro remetente
- p é o número de saídas (endereços que receberão saldo)
- m é o número de entradas (endereços origem do saldo)
- *RCTTypeBulletproof2* é o tipo de transação usada atualmente no Monero, a qual faz uso de assinaturas CLSAG e *Pedersen commitments* (ver seção 2.5)
- *Bulletproof range proof* é uma prova que valores da quantia estão dentro de um intervalo de valores (ver seção2.6)
- No primeiro item:
  - O termo $v+1$ é o tamanho total do Anel: v corresponde aos membros não assinaram a transação, e +1 corresponde ao assinante
  - O termo $2(v+1)$ é o tamanho do membro do anel, que é um par $(K_{i,j}, C_{i,j} - C'_{\pi,j})$ de imagem de chave e *commitment para zero* (ver seção 2.4.6)
  - O termo +3 contabiliza a assinatura CLSAG σ, a imagem de chave $\hat{K}_{\pi,j}$ da entrada verdadeira, e um *pseudo commitment* $C'_{\pi,j}$.
  - O termo m é o número de entradas, sendo que para cada entrada teremos $2(v+1)+3$ dados de 32 bytes.
  - 32 é o número de bytes de cada dado
- No segundo item, os termos representam a complexidade de espaço do algoritmo da prova *Bulleproof range proof* presente na transação

- Os fatores m e p nos demais itens refletem o fato de que para cada entrada e saída estão associados *commitments*, imagens de chave, e quantias, cada qual ocupando 32 bytes.

Desde a atualização de protocolo *Fluorine Fermi* [sel22] o valor de v é 15. Daí, efetuando os cálculos (ver apêndice B), têm-se a seguinte tabela com o tamanho de uma transação para alguns valores sensatos de m e p:

m	p	transação em *bytes*
1	1	2104
1	2	2240
2	2	3520
3	3	4936
5	3	7496
4	8	6640
8	4	11408
8	8	11760

Tabela 4.3: *Tamanhos estimados de transações em Monero*

Já no Zcash Sapling, o tamanho máximo de uma transação é 2MB, e seu tamanho mínimo depende do número de *Spend Descriptions* e *Output Descriptions*, com mínimo de 352 e 948 bytes cada, respectivamente [Hop+22, p. 52].

A razão de uma *Output Description* ter um tamanho maior que a *Spend Description* é devido ao texto cifrado com dados destinados ao recipiente. A codificação destes dados, em texto puro, contém 564 bytes, sendo 512 bytes do campo *memo* [Hop+22, p. 110].

Por fim, a assinatura usada em uma transação Sapling, *bindingSigSapling*, contém 64 bytes [Hop+22, p. 119]. Outros dados dentro de uma transação Sapling variam em tamanho e alguns são opcionais[1], por isso seu tamanho também pode variar, assim como as do Monero.

Portanto, é possível estimar o tamanho de uma transação Sapling levando em conta os principais fatores contribuintes para seu tamanho: número m de *Spend Descriptions*, número n de *Output Descriptions*, e 64 bytes da assinatura. Baseado nestes valores, constrói-se a seguinte tabela estimando um valor aproximado para a transação resultante (detalhes dos cálculos no apêndice B):

[1] Na verdade, a assinatura *bindingSigSapling* é "opcional"caso haja zero descrições do tipo *Spend* e *Output*, mas uma transação assim é vazia e não efetua nenhuma transferência. Qualquer transação que efetue alguma transferência terá pelo menos uma descrição.

m	n	transação em *bytes*
1	1	4144
1	2	4144
2	2	4496
3	3	4848
5	3	5552
4	8	9056
8	4	6608
8	8	10464

Tabela 4.4: *Tamanhos estimados de transações em Zcash Sapling*

Conforme especificado na codificação de uma transação Zcash v4 (usada no Sapling) [Hop+22, p. 119], demais campos da transação não ocupariam mais que 50 bytes, por isto eles não foram levado em contas nos cálculos visto que são menos que 5% por do tamanho de uma transação com 1 descrição *Spend* e 1 descrição *Output*.

No caso do Zcash *Orchard*, usa-se uma transferência chamada *Action Transfer* para as mesmas finalidades de *Spend Transfer* e *Output Transfer* no *Sapling*. Assim, têm-se apenas uma descrição *Action Description*, a qual tem 820 bytes [Hop+22, p. 120]. Há também uma assinatura `vSpendAuthSigsOrchard` de 64 bytes para cada *Action Description*, e uma assinatura para a transação toda `bindingSigOrchard`.

Com base nisso, pode-se comparar o tamanho, em bytes, das transações em Monero e Zcash para parâmetros similares (ver apêndice B):

m	n	Monero	Zcash Sapling	Zcash Orchard
1	1	2104	1364	948
1	2	2240	2312	1832
2	2	3520	2664	1832
3	3	4936	3964	2716
5	3	7496	4668	4484
8	4	11408	6672	7136
4	8	6640	9056	7136
8	8	11760	10464	7136

Tabela 4.5: *Tamanhos estimados de transações em Monero e Zcash*

Percebe-se que Monero e ZCash Sapling tem um transações com tamanhos próximos para os mesmos parâmetros e que ZCash Orchard é, no geral, mais eficiente em termos de tamanho que ambos.

Pode se notar que o tamanho da transação no Monero é mais afetado pelo parâmetro m, que representa o número de entradas, devido à necessidade de fornecer uma assinatura em anel e especificar os membros do anel. Por outro lado, a descrição da saída em Monero envolve apenas especificar o endereço único do recipiente.

No caso do Zcash Sapling, o parâmetro n, que representa o número de saídas, afeta mais o tamanho da transação porque cada *Output Description* tem o texto cifrado de parâmetros confidenciais, que tem mais de 500 bytes.

Por fim, no caso do Zcash Orchard, uma *Action Transfer* desempenha o papel equivalente de *Spend Transfer* e *Output Transfer*. Portanto, é razoável supor, apesar de não ser explicitamente declarado no protocolo, que o equivalente de 5 *Spend Transfers* e 5 *Output Transfers* são 5 *Action Transfers*, e o equivalente de 5 *Spend Transfers* e 1 *Output Transfers* também são 5 *Action Transfers*. Logo, o que importa para os cálculos é o maior valor entre m e n, $\max(m, n)$.

O tamanho das transações tem um impacto direto na descentralização e robustez do sistema, visto que estas transações são mantidos em registros de dados distribuídos sincronizados entre os participantes da rede. Se os requisitos de espaço de disco forem muito altos, então os participantes da rede serão constituídos de poucas entidades capazes de custear máquinas potentes e com alta capacidade de armazenamento. Atualmente, este não é o caso para Zcash e Monero, cujas listas de blocos de transação são aproximadamente 260GB [blo23] e 140GB-150GB respectivamente [Mon].

Entretanto, existe um risco potencial para Zcash: em um período de 1 ano e 4 meses, o banco de dados de Zcash aumentou de 30GB para cerca de 260GB [blo23]. Tudo indica que se trata de um ataque de *spam* na rede [Dub22]. Os desenvolvedores do Zcash alegaram que o *spam* não burla as regras do sistema e que, no limite, o sistema foi projetado para aguentar transações de até 2MB e até 2GB por dia de adição de dados [Zca22b]. É questionável se o sistema poderia manter-se descentralizado e sustentável caso operasse no limite, visto que os requisitos de espaço poderiam rapidamente alcançar TeraBytes.

Com base nestas informações, vê-se que Monero e Zcash tem transações com tamanho da mesma ordem de grandeza. A principal diferença é que Zcash tem regras menos restritas quanto ao tamanho máximo de um bloco de transações, e por isto foi alvo de ataques de *spam*, causando um crescimento substancial dos requisitos de armazenamento de disco para os nós da rede. Monero tem um tamanho de bloco dinâmico, autoajustável com base nas transações passadas (ver bibliografia [NAK20, p. 63]) e esta abordagem têm funcionado até agora, conforme vê-se um crescimento linear do espaço exigido para armazenar os blocos.

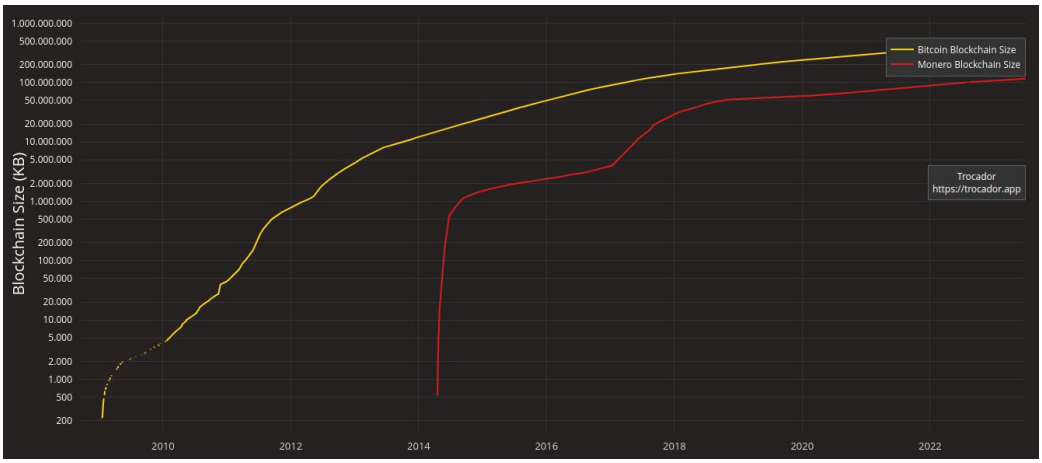

Figura 4.4: *Espaço exigido para armazenar o histórico de transações de Monero e Bitcoin*
Créditos: https://moneroj.net/blockchainsize/

4.5 Tempo de processamento

Transações em Zcash requerem um tempo de processamento computacional maior do que transações em outras moedas criptográficas por causa das provas de zero conhecimento. O tempo para computar a prova de zero conhecimento no Zcash Sprout era cerca de 37 segundos e no Sapling este tempo é 2 segundos [Pet18].

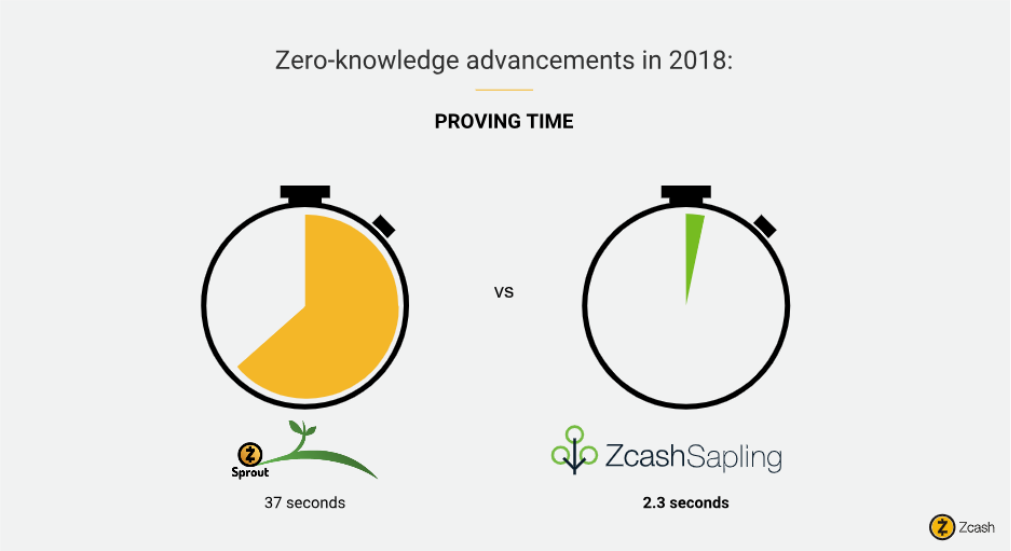

Figura 4.5: *Tempo para computar a prova zk-SNARKs em Zcash*
Créditos: https://electriccoin.co/blog/reducing-shielded-proving-time-in-sapling/

Não há dados oficiais sobre o tempo médio necessário para computar provas no Orchard, mas este deve ser menor que do Sapling, tendo em vista que Orchard é designado para ser um protocolo mais eficiente [Hop+21].

Por outro lado, o tempo de verificação das provas zero-knowledge é da ordem de milissegundos [Zca22a]. É por isso mesmo, por se tratar de uma prova de rápida verificação, que se usa o termo *sucinto* em *zero-knowledge Succint Non-Interactive Arguments of Knowledge*.

É importante ressaltar que o tempo de geração da prova de zero-knowledge não tem efeito na latência da rede, pois a computação da prova é feita apenas pelo usuário que está efetuando a transferência. Os nós da rede realizam apenas a verificação das provas zero-knowledge, que é da ordem de milissegundos, e não a sua geração. Sendo assim, um tempo de alguns segundos para gerar a prova zk-SNARKs pode ter efeito na experiência do usuário, mas não na rede em si. Portanto, o custo computacional da prova de zero-knowledge não compromete a estabilidade da rede: este custo é repassado para o usuário final somente, o qual gastará este poder de processamento computacional somente quando for enviar transações.

Já no Monero, o tempo médio para a construção e verificação de uma transação `RCTTypeBulletproof2` é da ordem de milissegundos (ver apêndice C). Para transações com poucas entradas e saídas, este tempo pode chegar a menos de 1 milissegundo. Já para uma transação com dezenas de entradas e saídas, este valor pode chegar a algumas

centenas de milissegundos.

De toda forma, ambos Monero e Zcash tem tempo de verificação de transação da ordem de milissegundos. Este tempo é um fator determinante na estabilidade da rede, visto que os nós devem entrar em sincronia e verificar as provas criptográficas de cada transação. É fundamental que estas provas não demandem alto custo computacional, pois do contrário pode comprometer a sincronia dos nós da rede de modo que apenas máquinas potentes farão parte da rede, o que pode resultar na centralização de nós em empresas e instituições.

Capítulo 5

Conclusão

Monero e Zcash são dois sistemas criptográficos construídos com base no modelo do Bitcoin, com diversas adaptações e melhorias para prover privacidade aos seus usuários. Como visto na seção 4.1, ambas moedas utilizam tecnologias que permitem ocultar a quantia, origem e destino de uma transação. Assim sendo, têm bastante semelhança no que tange a funcionalidades de privacidade.

Conforme discutido na seção 4.2, a diferença está nas tecnologias empregadas: Zcash utiliza provas zero-knowledge, permitindo um Anonymity Set grande, e oculta completamente o destino por meio de encriptação dos dados. Monero utiliza RingCT com número de anel fixo para prover uma uniformidade das transações e dificultar a distinção de seus usuários, e usa endereços de pagamento únicos para o destinatário.

Ambos os sistemas são seguros e o fato de estarem funcionando por anos é testemunha de sua credibilidade: Monero está em operação desde 2014 e Zcash desde 2016. A manutenção da privacidade e segurança destes sistemas não é fixa e dependerá da constante evolução dos mesmos, adaptando-se aos ataques e corrigindo-se as vulnerabilidades que surgirem. Por isto, ambos estão em continua melhoria, tendo se desenvolvido no decorrer de anos e implementado novas versões de protocolos mais robustas, seguras e eficientes.

Portanto, é possível que daqui alguns anos as tecnologias abordadas neste trabalho não estejam mais sendo usadas e tenham sido substituídas por outras mais eficazes. Mas isto apenas reflete uma característica da criptografia: adaptação contínua em um ambiente adversarial dinâmico.

Apêndice A

Computação de hash em CPU

Para estimar um limite inferior do número de hashes que um computador moderno pode computar, foi utilizado um programa em Rust adaptado de uma discussão no fórum *BitcoinTalk* (ver https://bitcointalk.org/index.php?topic=5471654.0)

Eis o arquivo TOML:

Programa A.1 Arquivo TOML

```
1    [package]
2    name = "hashesdemo"
3    version = "0.1.0"
4    edition = "2018"
5    [[bin]]
6    name = "main"
7    path = "main.rs"
8
9    [dependencies]
10   sha2 = "0.10.8"
11   rayon = "1.8.0"
```

Eis o arquivo principal:

Programa A.2 Computação de sha256 hashes em Rust

```
1   extern crate rayon;
2   extern crate sha2;
3
4   use rayon::prelude::*;
5   use sha2::{Sha256, Digest};
6   use std::time::Instant;
7
8   const NUM_HASHES: usize = 10000000;
9   const NUM_THREADS: usize = 2;
10
11  // expensive function that computes double sha256 hashes
12  fn compute_hashes() {
13     for i in 0..NUM_HASHES {
14        let first_digest = Sha256::digest(i.to_ne_bytes());
15        let _second_digest = Sha256::digest(first_digest);
16     }
17  }
18
19  fn main() {
20     // library that provides thread support
21     rayon::ThreadPoolBuilder::new()
22        .num_threads(NUM_THREADS)
23        .build_global()
24        .unwrap();
25
26     // start timer
27     let start = Instant::now();
28
29     // each thread will run `compute_hashes` once
30     (0..NUM_THREADS).into_par_iter().for_each(|_| compute_hashes());
31
32     // stop timer
33     let duration = start.elapsed();
34
35     // Each thread runs `compute_hashes`, which computes 2*NUM_HASHES sha256 hashes
36     let total_hashes = 2 * NUM_HASHES * NUM_THREADS;
37
38     // hashes per second
39     let hashes_per_second = total_hashes as f64 / duration.as_secs_f64();
40
41     // output results
42     println!("Total hashes: {:.0} hashes", total_hashes);
43     println!("Time taken: {:?} seconds", duration);
44     println!("Hash rate: {:.0} hashes per second", hashes_per_second);
45  }
```

Os resultados foram os seguintes:

```
$ cargo run --release 2>/dev/null
Total hashes: 40000000 hashes
Time taken: 6.41430237s seconds
Hash rate: 6236064 hashes per second
```

O programa mencionado computou mais de 6 milhões de hashes por segundo em uma máquina Linux, com arquitetura x86_64, e processador Intel Core i5 2.60Hz.

Alguns usuários do fórum apontaram que o resultado deste código pode ser menor do que deveria por usar um código não otimizado, e que ferramentas como *hashcat*, com código otimizado para o kernel, poderiam providenciar um *benchmark* mais acurado.

De toda forma, o código fornecido é capaz de evidenciar que um CPU moderno consegue computar milhões de hashes *sha256* por segundo. Isto é suficiente para os propósitos da discussão na seção 1.7.

Apêndice B

Tamanho das transações

O seguinte programa em Python foi usado para estimar um limite inferior para o tamanho das transações em Monero e Zcash (ver seção 4.4):

Programa B.1 Estimação do tamanho de certas transações em Monero e Zcash

```
#!/usr/bin/python3
import math

def tx_size_monero(m=1, p=1, v=15):
   size = 0
   size += (2*(v+1)+3)*m*32
   size += (2 * math.ceil(math.log2(64*p)) + 9) * 32
   size += (m+p+p)*32
   size += p*8
   size += 32+32+8+8
   size += (v+1)*m*8
   return size

def tx_size_sapling(m=1, n=1):
   return 64 + 352 * m + 948 * n

def tx_size_orchard(n=1):
   return 64 + (64+820) * n

tx_params = [ [1,1], [1,2], [2,2], [3,3], [5,3], [4,8], [8,4], [8,8], ]

print("m\tn\tmonero\tsapling\torchard")
for (m, n) in tx_params:
   size_monero = tx_size_monero(m, n )
   size_sapling = tx_size_sapling(m, n)
   size_orchard = tx_size_orchard(max(m, n))
   print(f"{m}\t{n}\t{size_monero}\t{size_sapling}\t{size_orchard}")
```

O programa acima produz a seguinte saída:

```
m  n  monero  sapling  orchard
1  1  2104    1364     948
1  2  2240    2312     1832
2  2  3520    2664     1832
3  3  4936    3964     2716
5  3  7496    4668     4484
4  8  6640    9056     7136
8  4  11408   6672     7136
8  8  11760   10464    7136
```

Apêndice C

Tempo de transação em Monero

O tempo para construção e verificação de transações do Monero pode ser medido pelos próprios testes de desempenho incluídos no repositório oficial do Monero https://github.com/monero-project/monero.

Para tanto, primeiro é preciso compilar o código fonte com `Make`, conforme as instruções de plataforma providas no `README` do repositório.

Após isto, o código compilado estará no diretório `build`, e o teste relevante é `tests/performance_tests/performance_test`. A localização exata do arquivo vai depender da plataforma na qual o código foi compilado. Se o código for compilado no Linux na versão master, então a localização do arquivo será `build/Linux/master/releases/tests/performance_tests/performance_test`.

Os testes foram executados em uma máquina Linux arquitetura x86_64, com 4 CPUs Intel Core i5 2.60Hz e 24 GB de RAM. Os resultados foram salvos em um arquivo, para eventual consulta.

```
$ cd tests/performance_tests && ./performance_test > test.log
```

O teste relevante para o tempo gasto para construir uma transação é `test_construct_tx`, cujo resultados foram os seguintes:

```
test_construct_tx<1, 1, false> (200 calls) - OK: 600 us/call
test_construct_tx<1, 2, false> (200 calls) - OK: 685 us/call
test_construct_tx<1, 10, false> (25 calls) - OK: 1320 us/call
test_construct_tx<1, 100, false> (5 calls) - OK: 9400 us/call
test_construct_tx<1, 1000, false> (5 calls) - OK: 92200 us/call
test_construct_tx<2, 1, false> (200 calls) - OK: 825 us/call
test_construct_tx<2, 2, false> (200 calls) - OK: 925 us/call
test_construct_tx<2, 10, false> (25 calls) - OK: 1600 us/call
test_construct_tx<2, 100, false> (5 calls) - OK: 9800 us/call
test_construct_tx<10, 1, false> (25 calls) - OK: 2680 us/call
test_construct_tx<10, 2, false> (25 calls) - OK: 2760 us/call
test_construct_tx<10, 10, false> (25 calls) - OK: 3480 us/call
test_construct_tx<10, 100, false> (5 calls) - OK: 11800 us/call
test_construct_tx<100, 1, false> (5 calls) - OK: 23600 us/call
test_construct_tx<100, 2, false> (5 calls) - OK: 23800 us/call
test_construct_tx<100, 10, false> (5 calls) - OK: 24400 us/call
```

```
test_construct_tx<100, 100, false> (5 calls) - OK: 33000 us/call
test_construct_tx<2, 1, true> (10 calls) - OK: 17300 us/call
test_construct_tx<2, 2, true> (10 calls) - OK: 32200 us/call
test_construct_tx<2, 10, true> (5 calls) - OK: 168 ms/call
test_construct_tx<10, 1, true> (5 calls) - OK: 20800 us/call
test_construct_tx<10, 2, true> (5 calls) - OK: 42400 us/call
test_construct_tx<10, 10, true> (5 calls) - OK: 166 ms/call
test_construct_tx<100, 1, true> (5 calls) - OK: 60000 us/call
test_construct_tx<100, 2, true> (5 calls) - OK: 77000 us/call
test_construct_tx<100, 10, true> (5 calls) - OK: 227 ms/call
```

Como pode se ver, a maioria das transações foram construídas em poucos milissegundos, sendo que a transação que levou mais tempo tinha 100 entradas e 10 saídas e gastou 227 milissegundos. Transações mais simples, com 2 entradas e saídas, levaram menos de 1 milissegundo.

Os testes relevantes para medir o tempo de verificação de uma transação são `test_sig_clsag` e `test_bulletproof_plus`, cujo resultados foram os seguintes:

```
test_sig_clsag<4, 2, 2> (1000 calls) - OK: 2501 us/call
test_sig_clsag<8, 2, 2> (1000 calls) - OK: 4755 us/call
test_sig_clsag<16, 2, 2> (1000 calls) - OK: 9346 us/call
test_sig_clsag<32, 2, 2> (1000 calls) - OK: 18606 us/call
test_sig_clsag<64, 2, 2> (1000 calls) - OK: 37752 us/call
test_sig_clsag<128, 2, 2> (1000 calls) - OK: 78572 us/call
test_sig_clsag<256, 2, 2> (1000 calls) - OK: 170 ms/call
test_bulletproof_plus<true, 1> (100 calls) - OK: 4020 us/call
test_bulletproof_plus<false, 1> (20 calls) - OK: 23400 us/call
test_bulletproof_plus<true, 2> (50 calls) - OK: 10540 us/call
test_bulletproof_plus<false, 2> (10 calls) - OK: 74600 us/call
test_bulletproof_plus<true, 15> (10 calls) - OK: 50100 us/call
test_bulletproof_plus<false, 15> (2 calls) - OK: 388 ms/call
```

Pode se ver que o tempo de verificação também é da ordem de milissegundos, sendo apenas alguns milissegundos para transações com parâmetros menores e 200 a 400 milissegundos para transações com parâmetros altos.

Referências

[AB09] Sanjeev Arora e Boaz Barak. *Computational complexity: a modern approach*. 1ª ed. Cambridge University Press, 2009 (ver p. 41).

[Ant17] Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain*. "O'Reilly Media, Inc.", 2017 (ver pp. 13, 16).

[Bac02] Adam Back. *HashCash: a denial of service counter-measure*. 2002. URL: http://www.hashcash.org/papers/hashcash.pdf (ver p. 9).

[Ben+13] Eli Ben-Sasson et al. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Cryptology ePrint Archive, Paper 2013/879. https://eprint.iacr.org/2013/879. 2013. URL: https://eprint.iacr.org/2013/879 (ver p. 63).

[Ber+11] Daniel J. Bernstein et al. "High-speed high-security signatures". Em: (set. de 2011) (ver p. 56).

[bit23] bitinfocharts. *InfoChar Zcash*. 2023. URL: https://bitinfocharts.com/zcash/ (acesso em 10/10/2023) (ver p. 72).

[blo23] blockchair. *Zcash blockchain size chart*. 2023. URL: https://blockchair.com/zcash/charts/blockchain-size (ver p. 77).

[Com22] Electric Coin Company. *Zcash Metrics*. 2022. URL: https://electriccoin.co/zcash-metrics/ (acesso em 17/10/2023) (ver pp. 71, 72).

[Dub22] Parth Dubey. *Zcash hit spam attack*. Out. de 2022. URL: https://www.ibtimes.com/privacy-focused-blockchain-zcash-hit-spam-attack-3621403 (ver p. 77).

[flu17] fluffypony. *Monero 0.11.0.0 "Helium Hydra"Released*. Set. de 2017. URL: https://www.getmonero.org/2017/09/07/monero-0.11.0.0-released.html (acesso em 29/08/2023) (ver p. 29).

[flu18] fluffypony. *Monero 0.13.0 "Beryllium Bullet"Release*. Out. de 2018. URL: https://www.getmonero.org/2018/10/11/monero-0.13.0-released.html (acesso em 29/08/2023) (ver p. 29).

[Hai+09] Iftach Haitner et al. *Statistically Hiding Commitments And Statistical Zero-Knowledge Arguments From Any One-Way Function*. 2009. URL: https://scholar.harvard.edu/sites/scholar.harvard.edu/files/salil/files/hidingcommitments.pdf (ver p. 40).

[Hop+21] Daira Emma Hopwood et al. *Orchard Shielded Protocol*. Fev. de 2021. URL: https://zips.z.cash/zip-0224 (acesso em 06/10/2023) (ver pp. 67, 78).

[Hop+22] Daira Hopwood et al. *Zcash Protocol Specification*. Set. de 2022 (ver pp. 52–67, 75, 76).

[Hop19] Daira Emma Hopwood. *Disabling Addition of New Value to the Sprout Chain Value Pool*. Mar. de 2019. URL: https://zips.z.cash/zip-0211 (acesso em 21/09/2023) (ver p. 63).

[Lab] Monero Research Labs. *Monero Research Labs*. URL: https://www.getmonero.org/resources/research-lab/ (ver p. 35).

[LSP82] Leslie Lamport, Robert Shostak e Marshall Pease. "The Byzantine Generals Problem". Em: *ACM Transactions on Programming Languages and Systems* (jun. de 1982), pp. 382–401. URL: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/ (ver pp. 7, 8).

[Mie+13] Ian Miers et al. "Zerocoin: Anonymous distributed e-cash from bitcoin". Em: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 397–411. URL: https://zerocoin.org/media/pdf/ZerocoinOakland.pdf (ver p. 36).

[Mon] MoneroDocs. *Monero Technical Specs*. URL: https://monerodocs.org/technical-specs/ (acesso em 18/10/2023) (ver p. 77).

[Nak08] Satoshi Nakamoto. *Bitcoin: A Peer-To-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (ver pp. 1, 5, 7).

[NAK20] Sarang Noether, Kurt M. Alonso e Koe. *Zero to Monero*. 2ª ed. 2020. URL: https://www.getmonero.org/library/Zero-to-Monero-2-0-0.pdf (ver pp. 19, 20, 22, 24, 25, 27, 29–31, 33–35, 73, 77).

[Pet18] Paige Peterson. *Reducing Shielded Proving Time in Sapling*. Dez. de 2018. URL: https://electriccoin.co/blog/reducing-shielded-proving-time-in-sapling/ (ver p. 78).

[QSM99] Jean-Jacques Quisquater, Xavier Serret-Avila e Henri Massias. *Design of a Secure Timestamping Service with Minimal Trust Requirements*. 1999 (ver p. 12).

[Sab13] Nicolas van Saberhagen. *CryptoNote V2.0*. 2013. URL: https://cryptonote.org/whitepaper.pdf (ver pp. 14, 17, 19, 28).

[Sas+14] Eli Ben Sasson et al. "Zerocash: Decentralized anonymous payments from bitcoin". Em: *2014 IEEE symposium on security and privacy*. IEEE. 2014, pp. 459–474 (ver pp. 38, 39, 42–44, 49).

[sel20] selsta. *Monero 0.17.0.0 "Oxygen Orion"released*. Set. de 2020. URL: https://www.getmonero.org/2020/09/17/monero-0.17-released.html (acesso em 30/08/2023) (ver p. 24).

[sel22] selsta. *Monero 0.18.0.0 'Fluorine Fermi' released*. Jul. de 2022. URL: https://www.getmonero.org/2022/07/19/monero-0.18.0.0-released.html (acesso em 29/08/2023) (ver pp. 28, 75).

[Ser18] SerHack. *Mastering Monero: the future of private transactions*. 1ª ed. 2018. URL: https://masteringmonero.com/book/Mastering%20Monero%20First%20Edition%20by%20SerHack%20and%20Monero%20Community.pdf (ver pp. 14–16, 35).

[SP18] Douglas R. Stinson e Maura B. Paterson. *Cryptography: Theory and Practice*. 4ª ed. 2018 (ver pp. 3, 30).

[tha04] thankful_for_today. *Bitmonero: a new coin based on Cryptonote technology*. 2004. URL: https://bitcointalk.org/index.php?topic=563821.0 (acesso em 20/05/2023) (ver p. 14).

[Ye+20] Claire Ye et al. *Alt-Coin Traceability*. Cryptology ePrint Archive, Paper 2020/593. https://eprint.iacr.org/2020/593. 2020. URL: https://eprint.iacr.org/2020/593 (ver pp. 72, 73).

[Zca22a] Zcash. *What are zk-SNARKs*. 2022. URL: https://z.cash/learn/what-are-zk-snarks/ (ver p. 78).

[Zca22b] ZcashCommunity. *Zcash blockchain size risks*. 2022. URL: https://forum.zcashcommunity.com/t/zcash-blockchain-size-risks/43126 (ver p. 77).