Universidade de Brasília

Faculdade de Tecnologia

Departamento de Engenharia Elétrica

Protocolo de Swap Atômico Entre Mints de Cashu

Hugo Szerwinski

TRABALHO DE CONCLUSÃO DE CURSO

ENGENHARIA DE REDES DE COMUNICAÇÃO

Brasília

2025

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

Protocolo de Swap Atômico Entre Mints de Cashu

Hugo Szerwinski

Trabalho de Conclusão de Curso apresentado como requisito para a obtenção do título de Engenheiro de Redes de Comunicação.

Orientador: Professor Doutor José Edil Guimarães de Medeiros

Brasília
2025

FICHA CATALOGRÁFICA

Szerwinski, Hugo.

Protocolo de Swap Atômico Entre Mints de Cashu / Hugo Szerwinski; orientador José Edil Guimarães de Medeiros. -- Brasília, 2025.

103 p.

Trabalho de Conclusão de Curso (Engenharia de Redes de Comunicação) -- Universidade de Brasília, 2025.

1. Swap. 2. Cashu. 3. ECash. 4. Schnorr. 5. Multisig. I. Medeiros, José Edil Guimarães de, orient. II. Título.

Universidade de Brasília
Faculdade de Tecnologia
Departamento de Engenharia Elétrica

Protocolo de Swap Atômico Entre Mints de Cashu

Hugo Szerwinski

Trabalho de Conclusão de Curso apresentado como requisito para a obtenção do título de Engenheiro de Redes de Comunicação.

Brasília, 14 de julho de 2025:

Professor Doutor José Edil Guimarães de Medeiros,
UnB/FT/ENE
Orientador

Professor Doutor João José Costa Gondim,
UnB/FT/ENE
Examinador interno

Engenheiro Eletricista Breno Rodrigues Brito,
UnB/FT/ENE
Examinador externo

Dedico essa conquista a meus pais, Ana Carolina e Leandro, cujo amor e apoio incondicionais foram a base de minha jornada acadêmica;

A mim mesmo, pela resiliência e comprometimento incansáveis;

A todos os entusiastas de tecnologia financeira e mecanismos que empoderam o indivíduo em oposição ao controle estatal.

Agradecimento

Por oportuno, expresso minha profunda gratidão ao estimado Professor Doutor José Edil Guimarães de Medeiros, personagem imprescindível na condução deste Trabalho de Conclusão de Curso. A ele, que ofereceu suporte inestimável em todas as etapas, dedico o meu obrigado, obrigado, muito obrigado!

04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e
63656c6c6f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420
666f722062616e6b73

.

O problema fundamental com a moeda convencional é toda a confiança necessária para fazê-la funcionar. O banco central deve ser confiável para não desvalorizar a moeda, mas a história das moedas fiduciárias é cheia de quebras dessa confiança (Satoshi Nakamoto)

Resumo

Este trabalho propõe um Protocolo de Swap Atômico, destinado a permitir trocas seguras e confiáveis de tokens entre usuários de mints no sistema Cashu, uma versão moderna do ecash *Chaumian* que combina o anonimato oferecido pelas assinaturas cegas com a infraestrutura criptográfica robusta do Bitcoin. A proposta surge devido às limitações de interoperabilidade no Cashu, causadas pela dependência de chaves criptográficas únicas para cada mint, e apresenta uma solução baseada em Adaptor Signatures (derivação do esquema Schnorr) e contratos P2PK multisig. O protocolo garante a atomicidade das transações através da linearidade nas assinaturas ($s = sa + t$) e compromissos criptográficos, assegurando que ambas as partes cumpram suas obrigações ou nenhuma o faça. O processo envolve a geração de Adaptor Signatures, cálculo de Pontos e Nonces adaptadores e validação pelas mints, eliminando conversões intermediárias custosas. Os resultados confirmam a viabilidade de trocas eficientes e privadas, com proteção contra ações mal-intencionadas e recuperação de ativos via locktime em caso de interrupções. Esse protocolo melhora a interoperabilidade do ecash, ampliando seu uso em situações de baixa, ou nenhuma confiança.

Palavras-chave: Swap; Cashu; eCash; Schnorr; Multisig.

Abstract

This document proposes an Atomic Swap Protocol, designed to enable secure and reliable token exchanges between users of different mints within the Cashu system, a modern iteration of "Chaumian" ecash that integrates the anonymity provided by blind signatures with the robust cryptographic infrastructure of Bitcoin. The proposal arises from the interoperability limitations in Cashu, stemming from the reliance on unique cryptographic keys for each mint, and presents a solution based on Adaptor Signatures (a derivative of the Schnorr scheme) and P2PK multisig contracts. The methodology ensures transaction atomicity by leveraging the linearity of signatures ($s = sa + t$), guaranteeing that both parties fulfill their obligations or neither does. The process includes the generation of Adaptor Signatures, the calculation of Adaptor Points and Nonces, and validation through mints, eliminating costly intermediate conversions. The results confirm the feasibility of efficient and private exchanges, offering protection against malicious actions and asset recovery via locktime in case of interruptions. This protocol enhances ecash interoperability, expanding its applicability in scenarios of low or no trust.

Keywords: Swap; Cashu; eCash; Schnorr; Multisig.

Lista de figuras

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas — Entidade responsável por normas técnicas no Brasil.
BDHKE	Blind Diffie-Hellman Key Exchange — Protocolo criptográfico para troca de chaves cegas, utilizado no protocolo Cashu para preservar privacidade.
BIP-340	Bitcoin Improvement Proposal 340 — Definição de assinaturas Schnorr na curva secp256k1 no Bitcoin, base para mecanismos do protocolo Cashu.
Bolt 11	Payment Encoding - Lightning Network Specification — Formato de fatura utilizado na Lightning Network para codificar pagamentos, especificado no BOLT 11, permitindo transações off-chain seguras e eficientes.
BTC	Bitcoin — Criptomoeda principal baseada na blockchain, usada como exemplo em swaps atômicos e conversões no Cashu.
DLEQ	Discrete Logarithm Equality — Prova de igualdade de logaritmos discretos, empregada para validar assinaturas no protocolo Cashu.
eCash	Electronic Cash — Sistema de moeda digital anônima, base histórica do protocolo Cashu inspirado em Chaum.
ECC	Elliptic Curve Cryptography — Criptografia baseada em curvas elípticas, fundamento de segurança em assinaturas como Schnorr e BDHKE.
HTLC	Hashed Timelock Contract — Contrato inteligente com limite de tempo para transações em blockchains, base para swaps atômicos no Bitcoin e Cashu.
LTC	Litecoin — Criptomoeda derivada do Bitcoin, utilizada como exemplo em swaps atômicos.
NUT-04	Network Upgrade Technical Document 04 — Especificação do processo de mintagem no protocolo Cashu.
NUT-05	Network Upgrade Technical Document 05 — Especificação do processo de melt no protocolo Cashu.
NUT-07	Network Upgrade Technical Document 07, especificação de consulta de estado de token no protocolo Cashu
NUT-10	Network Upgrade Technical Document 10, padronização de contratos P2PK e HTLC no protocolo Cashu
NUT-11	Network Upgrade Technical Document 11, especificação de provas P2PK no protocolo Cashu
NUT-14	Network Upgrade Technical Document 14 — Especificação de HTLCs no protocolo Cashu.
P2PK	Pay-to-Public-Key — Formato de script do Bitcoin que bloqueia fundos para uma chave pública, adaptado no protocolo Cashu.

P2SH	Pay-to-Script-Hash — Contrato que bloqueia fundos em um script hash, utilizado em HTLCs no Bitcoin.
TCC	Trabalho de Conclusão de Curso — Projeto final para obtenção do título de Engenheiro de Redes de Comunicação na UnB.
UnB	Universidade de Brasília — Instituição onde o trabalho foi desenvolvido.
ZKP	Zero-Knowledge Proof — Prova de conhecimento zero, explorada como mecanismo de segurança no protocolo Cashu.

Sumário

1 Introdução

Os sistemas de pagamento digitais evoluíram significativamente com a popularização da internet, substituindo gradualmente métodos tradicionais como o escambo, metais preciosos, cédulas bancárias e cartões de crédito por soluções mais avançadas, como *criptomoedas*.

Este contexto possibilitou o surgimento de tecnologias disruptivas, como o *ecash*, um sistema de dinheiro eletrônico baseado em *assinaturas cegas* proposto por David Chaum em 1982, cujo objetivo é garantir às transações digitais o anonimato proporcionado pelo dinheiro físico (Chaum, 1982). Preocupado com a falta de privacidade nas infraestruturas bancárias tradicionais, Chaum destacava que as instituições financeiras acessavam informações sensíveis, tais como domicílio, identidade, valores, finalidades e datas acerca dos envolvidos nas transações. Esses dados possibilitavam a construção de perfis detalhados abarcando padrões de consumo, localizações, renda estimada e associações dos usuários, comprometendo sua privacidade.

Apesar de sua inovação, o *ecash* original, implementado pela DigiCash (empresa fundada por David Chaum), não alcançou ampla adoção devido a alguns fatores, como: o estágio inicial de desenvolvimento da internet na década de 1980, a concorrência com métodos de pagamento mais acessíveis, como cartões de crédito, e também em face da resistência cultural e conflitos com instituições bancárias. Em desvirtude destes obstáculos, a DigiCash declarou falência em 1998.

Uma década depois, em 2008, o *white paper* do Bitcoin (Nakamoto, 2008) foi publicado em um fórum de cripto entusiastas (https://www.metzdowd.com/pipermail/cryptography/2008-October/014810.html), introduzindo um sistema de dinheiro eletrônico descentralizado, ponto-a-ponto, que resolve o problema do *gasto duplo* – a possibilidade de um ativo digital ser gasto mais de uma vez – sem intermediários.

Desde então, a adesão global ao Bitcoin reacendeu o interesse por moedas digitais, inspirando soluções modernas como o Cashu (Team, 2023a), um protocolo *open-source* *"Chaumian ecash"* que combina a privacidade do *ecash* original com a resistência à censura do Bitcoin. Diferentemente do Bitcoin, entretanto, o Cashu depende de emissoras centralizadas, denominadas *mints*, as entidades responsáveis por executarem um cliente completo do protocolo.

No Cashu, tokens de *ecash* são emitidos pelas *mints* em troca de ativos, como Bitcoin, por meio de um processo chamado *mintagem*, especificado na NUT-04 (Team, 2023a). Este processo pode ser descrito de forma simplificada, em algumas etapas, utilizando-se personagens fictícios intitulados Alice e Bob:

1. Alice solicita à sua *mint* uma cotação para converter bitcoins em tokens da emissora.

2. A *mint* responde com a fatura Bolt 11 (BOLT, 2023), uma *invoice* da rede Lightning Network (Poon; Dryja, 2016), solução de segunda camada do Bitcoin.

3. Alice paga a fatura usando sua carteira Lightning.

4. A *mint* retorna uma estrutura de dados (observada na Figura 1.1 abaixo) contendo um identificador para a chave utilizada na assinatura, acompanhado do *amount* (quantia em *satoshis* que o token representa), bem como a *assinatura cega* ($C\_$)[1] sobre ele e uma prova criptográfica ($DLEQ$)[1] que permite à Alice confirmar sua autenticidade sem que a mint revele informações adicionais.

5. De posse do retorno da *mint*, Alice constrói agora o objeto *JSON* que compõe o *token* propriamente dito, evidenciado na Figura 1.2. Ela realiza o descegamento da assinatura cega ($C\_$) e agrega ao objeto o campo *secret* (contrato que especifica as condições de gasto do token) - estes processos serão discutidos em detalhes nas seções posteriores.

6. Alice pode usar os tokens emitidos para realizar pagamentos ou enviá-los a outros usuários, incluindo aqueles de *mints* distintas, mas sem garantia de reciprocidade em eventuais trocas.


```
{
    "signatures": [
        {
            "id": "000b4c3d8b0e7397",
            "amount": 1000,
            "C_": "0229e33b12fc3a4585ed8cb0df6ee799377d986d03b66132a408f12836c529b97f",
            "dleq": {
                "e": "6664edcb624fdc4d0b25bfbf918419169aa33a6030dc75e949146a4ae786b0c9",
                "s": "0dde7b1f7daae889ff8d5c263fee9791335c26ec0258badf489d65823881e1ad"
            }
        }
    ]
}
```


Figura 1.1 – Retorno obtido à solicitação de *mintagem* do token

Figura 1.2 – Estrutura de dados do token

Cada *mint* de *Cashu* trabalha com pares de chaves criptográficas únicas, geradas na curva secp256k1 (Core, 2023), da mesma forma utilizada no padrão do Bitcoin. A chave privada, mantida em segredo pela *mint*, é usada para assinar os tokens durante a *mintagem*, enquanto a chave pública permite sua validação no gasto. Assim, um token emitido pela Mint A só pode ser validado e gasto através dela, impossibilitando que a Mint B processe diretamente tokens de outra emissora.

Observa-se, portanto, que apesar de uma *mint* não aceitar e nem operar com tokens emitidos por outra, é possível a um usuário enviar tokens obtidos em sua *mint* (através do processo descrito acima) para outro usuário que opera em uma *mint* distinta, como Alice enviando tokens da Mint A para Bob, usuário da Mint B e vice-versa. Bob poderia dirigir-se à mint de Alice com o token A que ela o enviou e trocá-lo por *bitcoin*, ou até usá-lo para fazer um pagamento a outro usuário, Carol, que gostaria de receber tokens da Mint A.

Entretanto, apesar dos cenários apresentados, não há um mecanismo nativo que ofereça a possibilidade de usuários realizarem uma troca de forma atômica – um *swap* em que ambos entreguem os tokens prometidos ou nenhum o faça. A atomicidade em trocas faz-se necessária em situações de pouca (ou nenhuma) confiança, como por exemplo em uma plataforma para trocas *peer-to-peer*, em que dois participantes desejam obter tokens um do outro, mas não têm confiança entre si. Sem essa garantia, e com desconfiança entre as partes e na plataforma utilizada para a troca, em eventual operação a ser realizada, Alice não tem confiança de que Bob enviará os tokens equivalentes da Mint B em contrapartida, e vice-versa, criando um desafio para permutas confiáveis. **Diante do exposto, fica nítida a existência de um problema de interoperabilidade entre *mints* de *Cashu* que impossibilita usuários do protocolo de realizarem trocas atômicas e seguras entre si, sem terceiros de confiança.**

[1] As *assinaturas cegas* - NUT-00, (Team, 2023a) - permitem que a *mint* assine tokens sem conhecer seu conteúdo, garantindo privacidade. A prova DLEQ (Discrete Logarithm Equality) - NUT-12, (Team, 2023a) - valida a autenticidade do token sem expor informações adicionais.

Alternativamente, os usuários interessados na troca poderiam converter seus tokens em *bitcoin* com suas respectivas *mints* (incorrendo em taxas de *peg-out* e *mintagem*), e realizarem a troca através de contratos HTLC (*hashed timelock contracts*) (Community, 2015) - mecanismo em que um dos participantes precisa revelar um segredo para realizar seu gasto, oportunizando que a outra parte veja este segredo e gaste sua contrapartida, um processo custoso, complexo e susceptível a erros, desmotivando os envolvidos e forçando-os a abandonar o *ecash* para recorrer à mecanismos e tecnologias externas para possibilitar a operação. A ausência de trocas atômicas confiáveis compromete a usabilidade do Cashu, limitando sua flexibilidade em cenários de confiança mínima.

MOTIVAÇÃO

A inspiração para o presente trabalho foi pautada no protocolo **Granola**, apresentado inicialmente por Breno Brito e Luis Schwab durante o *hackathon* SatsHack 2024, cuja proposta era possibilitar swaps atômicos entre usuários de mints no Cashu mediante o uso de HTLCs (Hash Time-Locked Contracts), conforme detalhado no repositório https://github.com/GranolaCash. Este trabalho partiu da análise e implementação do projeto referenciado, explorando sua viabilidade. Contudo, a abordagem evoluiu para incorporar *Adaptor Signatures*, uma variação das assinaturas Schnorr introduzida por Andrew Poelstra em *Scriptless Scripts* (Poelstra *et al.*, 2018), preservando o propósito da solução inicial, mas utilizando conceitos criptográficos distintos para cumprir com tal objetivo.

1.1 Objetivos do trabalho

O presente trabalho tem como objetivo o desenvolvimento de um *Protocolo de Swap Atômico* que permita trocas de tokens seguras e indivisíveis entre usuários de *mints* distintas no protocolo Cashu, superando a atual limitação de interoperabilidade que impede a realização destes tipos de transações por uma via direta e confiável. A proposta visa integrar um mecanismo complementar ao framework existente, eliminando a dependência de conversões intermediárias de tokens e assegurando que as trocas sejam concluídas apenas se ambas as partes cumprirem com suas obrigações. Para esta finalidade, foram utilizadas assinaturas *Schnorr* (Proposal, 2020), *Adaptor Signatures* (Poelstra *et al.*, 2018) e condições P2PK *multisig* (NUT-11) (Team, 2023a), observando as técnicas criptográficas padrão.

No intuito de se alcançar este objetivo central, o trabalho estabelece uma série de metas específicas que orientam o desenvolvimento do protocolo e garantem sua viabilidade técnica

e prática, envolvendo para tal a necessidade de:

- **Especificar um protocolo de troca atômica**: Projetar um mecanismo que assegure a troca de tokens entre usuários de *mints* diferentes de forma simultânea e segura, garantindo que, se uma parte falhar em cumprir sua obrigação, a outra não sofra perdas; a atomicidade é essencial para mitigar riscos em transações sem confiança mútua, um cenário comum no uso do Cashu.

- **Implementar assinaturas adaptadoras**: Integrar *Adaptor Signatures*, baseadas no esquema Schnorr, para vincular as transações entre os participantes, permitindo que o segredo de uma parte seja revelado apenas após a conclusão bem-sucedida da contrapartida pela outra. Essa técnica elimina a necessidade de terceiros de confiança.

- **Estender os mecanismos de Prova de Conhecimento Zero**: Ampliar as técnicas de validação criptográfica, incluindo o compartilhamento de variáveis adicionais, além da geração de uma prova *Schnorr* para demonstrar a procedência de certas informações, reforçando a segurança e a confiança na troca.

- **Desenvolver a prova de conceito**: Implementar uma demonstração prática do protocolo de swap atômico em um ambiente controlado, validando sua funcionalidade e viabilidade técnica para trocas seguras entre usuários de *mints* distintas no Cashu.

Estes objetivos refletem uma abordagem estruturada para a proposta de intervenção ao problema central, combinando fundamentação técnica com a preservação dos mecanismos já existentes no Cashu. Ao resolver a limitação de interoperabilidade, o trabalho busca não apenas facilitar trocas entre usuários de *mints* distintas, mas também pavimentar o caminho para uma adoção mais ampla do protocolo-base (Cashu), posicionando-o como uma alternativa viável a sistemas de pagamento tradicionais e outras criptomoedas. A validação desses objetivos será demonstrada por meio da implementação e teste do protocolo de *swap* atômico proposto, cujos resultados serão detalhados nas seções subsequentes, contribuindo para o avanço da pesquisa em tecnologias de privacidade e interoperabilidade no contexto do Bitcoin e soluções de fronteira.

1.2 Escopo definido

O escopo definido durante a elaboração do estudo em tela restringe seu foco no desenvolvimento e validação de um *Protocolo de Swap Atômico* para o sistema Cashu. A pesquisa concentra-se, portanto, exclusivamente na estruturação técnica e prática do protocolo, in-

cluindo sua lógica criptográfica, implementação em um ambiente controlado e demonstração de viabilidade, com ênfase na garantia de atomicidade e privacidade nas transações. A demonstração da prova de conceito do protocolo está presente no Capítulo 4, fazendo-se a ressalva, entretando, que não houve preocupação com a implementação segura das primitivas de criptografia, como testes de ataques ou baterias de segurança, que são típicos em estudos avançados. Para assegurar a clareza e a viabilidade do estudo, foram estabelecidas delimitações explícitas que excluem aspectos periféricos, permitindo um enfoque preciso na solução proposta. Neste sentido, e com o intuito de manter o escopo gerenciável e alinhado aos objetivos do trabalho, os seguintes elementos foram intencionalmente excluídos da análise:

- **Camada de comunicação**: Este estudo não aborda os detalhes da camada de comunicação utilizada para a troca de informações entre os participantes do protocolo. Aspectos como: protocolos de transporte da camada OSI, TCP ou UDP, e os mecanismos de tráfego, como NOSTR (Team, 2023c) ou APIs, não são explorados. O trabalho pressupõe que a comunicação entre os usuários e as *mints* ocorre de forma funcional, sem detalhar os meios técnicos pelos quais os dados são transmitidos.

- **Infraestrutura de rede**: Questões relacionadas à infraestrutura de rede, como escalabilidade em termos de largura de banda, latência, topologia ou tolerância a falhas, estão fora do escopo. A proposta assume um ambiente de rede ideal, onde a conectividade é garantida, permitindo que o foco permaneça na lógica do *swap* atômico e suas propriedades criptográficas.

- **Tecnologias subjacentes**: Embora o Cashu seja construído sobre a infraestrutura do Bitcoin e utilize a rede Lightning para operações como conversão de saldo, este trabalho não detalha o funcionamento interno dessas tecnologias, exceto se diretamente relevantes e indispensáveis para o entendimento da solução como um todo. Por exemplo, a conversão de tokens em Bitcoin (*peg-out*) é mencionada como um processo existente, mas sua implementação técnica ou otimização não é objeto do presente estudo.

- **Segurança operacional das *mints***: A pesquisa não aborda aspectos de segurança operacional das *mints*, como proteção contra ataques cibernéticos, falhas de hardware ou comportamento malicioso das próprias *mints* (ex.: recusa em validar tokens válidos). Assume-se que as *mints* operam de forma honesta, mas curiosa (*honest-but-curious*), seguindo as especificações do Cashu.

- **Testes em larga escala**: A validação do protocolo será realizada por meio de demons-

trações práticas e análises em um ambiente controlado, sem incluir testes em larga escala ou simulações de cargas de trabalho reais. O objetivo é comprovar a corretude e a viabilidade do protocolo, e não sua performance em cenários de produção com múltiplos usuários ou alto volume de transações.

- **Integração com outras tecnologias**: O trabalho não considera a integração do protocolo com outras tecnologias ou plataformas além do Cashu, como sistemas de pagamento alternativos ou blockchains alternativas. A solução é projetada especificamente para o ecossistema Cashu, com foco nas funcionalidades definidas pelas especificações NUT-07, NUT-10 e NUT-11 (Team, 2023a).

Estas delimitações garantem que o trabalho permaneça focado na inovação técnica do *Protocolo de Swap Atômico*, contribuindo para a resolução do problema de interoperabilidade no Cashu sem se desviar para aspectos secundários. Ao restringir o escopo à estruturação e exemplificação técnica e prática da solução proposta, o estudo mantém a profundidade necessária para oferecer uma contribuição significativa ao campo das tecnologias de pagamento digital privativas, alinhando-se aos objetivos estabelecidos e às limitações do Trabalho de Conclusão de Curso.

2 Revisão Bibliográfica

O presente capítulo oferece uma análise abrangente das tecnologias que sustentam o desenvolvimento do protocolo de *Swap Atômico* proposto, permeando desde conceitos como Bitcoin (Nakamoto, 2008) e Lightning Network (Poon; Dryja, 2016), até o funcionamento do Cashu (Team, 2023a), uma solução moderna de *ecash* (Chaum, 1982). Exploraremos também os pilares criptográficos que subsidiam a operação da solução apresentada, como o Diffie-Hellman (Diffie; Hellman, 1976), Blind Signatures (Chaum, 1982), Blind Diffie-Hellman Key Exchange (BDHKE) (Wagner, 1996), Schnorr (Proposal, 2020), Adaptor Signatures (Poelstra *et al.*, 2018), contratos criptográficos, e mecanismos multisig (Team, 2023a), que são cruciais para garantir segurança, privacidade e atomicidade. Por fim, examinaremos mecanismos de troca em criptomoedas e casos de uso práticos (Community, 2015), preparando o terreno para uma compreensão profunda do protocolo aqui apresentado.

2.1 Bitcoin e Lightning Network

Adiante, detalharemos os fundamentos das tecnologias mencionadas, bem como os conceitos criptográficos e de segurança que viabilizam a construção de um mecanismo robusto e privado para trocas atômicas.

2.1.1 Bitcoin: Fundamentos e Operação

O Bitcoin, introduzido por Satoshi Nakamoto em 2008 (Nakamoto, 2008), é o protocolo que deu origem à primeira criptomoeda descentralizada operando sob arquitetura *peer-to-peer*, que registra transações em um livro-razão público, conhecido como *blockchain*. A segurança do Bitcoin é garantida por técnicas criptográficas, como assinaturas digitais baseadas na curva elíptica secp256k1, e pelo mecanismo de consenso de prova de trabalho (*proof-of-work*). Nesse sistema, mineradores competem para resolver problemas matemáticos complexos, validando transações e adicionando blocos à *blockchain*, o que previne gastos duplos e assegura a integridade da rede.

Cada transação no Bitcoin é composta por entradas e saídas, assinadas com chaves privadas para autenticar a transferência de fundos. A *blockchain* mantém um registro imutável dessas transações, garantindo transparência e auditabilidade. No contexto do Cashu, o Bitcoin serve como o ativo subjacente para a criação de tokens *ecash*, onde os usuários trocam bitcoins por esses tokens que o representam indiretamente, mantendo uma paridade

de 1:1 com o valor especificado (geralmente em *satoshis*) durante a emissão.

2.1.2 Lightning Network: Escalabilidade e Eficiência

A Lightning Network, proposta por Poon e Dryja em 2016 (Poon; Dryja, 2016), é uma solução de segunda camada que opera sobre a *blockchain* do Bitcoin, projetada para aumentar a escalabilidade e reduzir custos de transação. Ela utiliza canais de pagamento bidirecionais, permitindo que transações sejam realizadas fora da cadeia (*off-chain*), com apenas os saldos finais registrados na *blockchain*. Esses canais são estabelecidos por meio de transações on-chain, que alocam bitcoins para uso off-chain.

As transações na Lightning Network são rápidas e de baixo custo, pois não exigem validação imediata por mineradores. A segurança é mantida por contratos criptográficos, como Hashed Timelock Contracts (HTLCs), que garantem que os fundos só sejam liberados sob condições específicas, como a revelação de um segredo criptográfico ou a expiração de um prazo (*timelock*). No Cashu, a Lightning Network também possui papel essencial no processo de mintagem dos tokens, funcionando como o canal pelo qual os usuários trocam bitcoins — o ativo colateral — por tokens *ecash*. Essa troca ocorre quando os usuários pagam faturas Bolt 11 geradas pelo sistema, convertendo os bitcoins em tokens que representam seu valor.

Observação 2.1. A fatura BOLT-11 é um formato padronizado utilizado na Lightning Network para facilitar pagamentos rápidos e de baixo custo entre usuários. Ela é codificada como uma string no formato *bech32* (Community, 2017), contendo informações essenciais como o valor da transação, um identificador único, um prazo de expiração, e uma assinatura criptográfica do nó receptor (neste caso, uma *mint* do Cashu). No Cashu, as faturas BOLT-11 são geradas por meio do endpoint `POST /v1/mint/quote/bolt11` (NUT-23), permitindo que usuários convertam bitcoins em tokens *ecash* ao pagá-las, com o processo detalhado na NUT-04 (Team, 2023a).

2.1.3 Relevância para o Protocolo de Swap Atômico

O Bitcoin atua como o ativo colateral que sustenta a emissão de tokens no Cashu, enquanto a Lightning Network fornece o canal para os pagamentos em Bitcoin que viabilizam essa mintagem. A compreensão desses fundamentos é essencial para entender como o protocolo de *swap* atômico proposto aproveita essas tecnologias para possibilitar trocas seguras e eficientes entre usuários de *mints* distintas, superando limitações de interoperabilidade no ecossistema.

2.2 Contexto e Funcionamento do Cashu

Esta seção apresenta o contexto histórico e os mecanismos operacionais do Cashu, fornecendo os insumos necessários para compreender sua integração ao protocolo de *swap* atômico proposto.

2.2.1 O que é Cashu

Cashu é um protocolo de *ecash*[5] open-source que surgiu como uma releitura do conceito original introduzido por David Chaum em 1983 (Chaum, 1982), com a proposta de utilizar assinaturas cegas para permitir transações anônimas.

Desenvolvido pelo programador *Open-Source* que se identifica pelo pseudômino **Calle**, o Cashu adapta a ideia original de David Chaum ao ecossistema do Bitcoin (utilizando como base as assinaturas cegas), destacando-se como uma solução de código aberto profundamente integrada à Lightning Network. Essa integração o posiciona como uma alternativa de confiança mínima (*trust-minimized*) aos sistemas de pagamento tradicionais.

Além disso, o Cashu mantém o anonimato durante retiradas de fundos, permitindo que usuários financiem, transacionem e retirem bitcoins sem revelar identidade ou histórico. Contudo, esse nível de privacidade implica em um *trade-off* de controle custodial, pois os usuários dependem da integridade da *mint*, responsabilidade esta que pode ser mitigada pela escolha de *mints* com boa reputação ou pela limitação do valor depositado.

2.2.2 Funcionamento Principal

O Cashu baseia-se em uma variante moderna das assinaturas cegas, conhecida como Blind Diffie-Hellman Key Exchange (BDHKE), proposta por David Wagner como uma evolução do método original de Chaum. O "Chaumian blinding", modelo original apresentado em 1983, utiliza um esquema genérico de cegamento onde o usuário modifica uma mensagem com uma função comutativa antes de enviá-la ao assinante, que a assina sem conhecer o conteúdo original, permitindo ao usuário remover o cegamento após a assinatura para obter uma versão válida, conforme apresentado abaixo:

[5] *eCash* (dinheiro eletrônico) refere-se a uma representação digital de valor emitida por um sistema de pagamento que utiliza técnicas criptográficas para garantir anonimato nas transações, inspirado no modelo de notas e moedas físicas. O *token de ecash*, portanto, é puramente uma estrutura de dados (objeto) que contém informações necessárias e suficientes para vinculá-lo criptograficamente a uma prova de posse/conhecimento. No contexto do Cashu, os tokens emitidos pelas *mints* funcionam como dinheiro eletrônico anônimo, lastreado em bitcoin.

Resumo das Fórmulas de Chaum (Chaum, 1982):

- $c(x)$: Função de cegamento aplicada pelo usuário ao valor x.
- $s'(c(x))$: Assinatura cega gerada pelo assinante.
- $c'(s'(c(x))) = s'(x)$: Descegamento pelo usuário, resultando na assinatura válida.
- Verificação: $s(s'(x)) = x$ e $r(s(s'(x)))$ confirma a validade, onde r é o predicado de redundância.

Este processo é introduzido no documento original - *Blind Signatures for Untraceable Payments* - (Chaum, 1982) a partir de uma analogia com urnas eleitorais, em que Chaum propõe um mecanismo pelo qual o eleitor preenche seu voto em uma cédula, insere-a em um envelope forrado com papel carbono, e empacota este último dentro de outro envelope com seu endereço de retorno, enviando-o ao validador. O responsável pela apuração assina o envelope externo (que contém o endereço do remetente) sem abrí-lo, transferindo a assinatura ao envelope interno através do papel carbono, e o devolve ao eleitor, que verifica a assinatura, extrai o voto assinado e o envia de volta anonimamente. Assim, o responsável pela contagem dos votos valida as cédulas sem identificar os eleitores, refletindo o princípio de cegamento que assegura privacidade nas transações de pagamento.

Já na implementação de Wagner (Wagner, 1996), esse conceito foi adaptado para curvas elípticas, especificamente na secp256k1 (Core, 2023), usada no Bitcoin, integrando operações de multiplicação escalar que envolvem o problema do logarítmo discreto. Essa adaptação assegura que a *mint* assine pontos cegos sem acessar os segredos subjacentes, tornando o protocolo mais eficiente, compatível com o ecossistema Bitcoin e livre de patentes associadas ao método original de Chaum. Abaixo, detalham-se as etapas principais, com definições das variáveis e uma explicação resumida do processo:

Definição 2.1. A função hash_to_curve mapeia um valor arbitrário (como um segredo) para um ponto válido na curva elíptica secp256k1, utilizada no Bitcoin e no Cashu. Ela realiza dois hashes criptográficos em sequência (SHA-256), sendo o primeiro entre o segredo e um fator para se evitar colisões (chamado de *domain_separator*), e o segundo entre o resultado do primeiro e um contador iterativo iniciado em 0. O resultado é avaliado até encontrar um ponto que satisfaça as propriedades da curva (incrementando-se o contador).

- **Cegamento pelo Usuário**: O usuário (ex.: Alice) seleciona um valor secreto x (um segredo aleatório de 32 bytes) e aplica a função hash_to_curve(x), que mapeia x para

um ponto Y na curva elíptica secp256k1. Em seguida, gera um nonce aleatório r (um escalar aleatório na curva) e computa $B\_ = Y + r \cdot G$, onde G é o ponto gerador da curva secp256k1. O valor $B\_$, representando o ponto cego que oculta Y do assinante, é enviado em conjunto ao *id*, que indica o catálogo de chaves públicas da *mint*, em conjunto ao *amount*, especificando a chave pelo valor do token, em uma estrutura de dados *JSON*, como observado na Figura 2.1.


```
{
    "amount": 1000,
    "id": "000b4c3d8b0e7397",
    "B_": "03da1346b55d553dc49059714180e61fefcff93ead3ee667f3755f97bb2dd98c01"
},
```


Figura 2.1 – Estrutura de dados inicial enviada à mint (*output*)

- **Assinatura Cega pela Mint**: A *mint*, que possui uma chave privada k_x específica para o *amount* solicitado, multiplica $B\_$ por k_x, resultando em $C\_ = k_x \cdot B\_$. Este valor $C\_$ é a assinatura cega retornada ao usuário, em conjunto à prova de conhecimento zero do logaritmo discreto - DLEQ), de acordo com a Figura 2.2.


```
{
    "signatures": [
        {
            "id": "000b4c3d8b0e7397",
            "amount": 1000,
            "C_": "0229e33b12fc3a4585ed8cb0df6ee799377d986d03b66132a408f12836c529b97f",
            "dleq": {
                "e": "6664edcb624fdc4d0b25bfbf918419169aa33a6030dc75e949146a4ae786b0c9",
                "s": "0dde7b1f7daae889ff8d5c263fee9791335c26ec0258badf489d65823881e1ad"
            }
        }
    ]
}
```


Figura 2.2 – Estrutura de dados retornada pela mint

- **Descegamento pelo Usuário**: Ao receber $C\_$ da *mint*, o descegamento é realizado pelo usuário através da equação $C = C\_ - r \cdot P$, sendo P_x a chave pública da *mint* derivada de k_x. Esse valor C é a assinatura válida da *mint* sobre Y, agora desassociada do nonce r. A carteira de Alice então monta a estrutura de dados final do token propriamente dito (após a validação DLEQ na etapa seguinte), evidenciada na Figura 2.3.

- **Verificação com Prova DLEQ**: Para garantir a autenticidade, o usuário realiza uma prova de igualdade de logaritmos discretos (DLEQ) composta por um desafio e e uma

resposta s. A verificação é iniciada construindo-se $R_1 = s \cdot G - e \cdot P_x$ e $R_2 = s \cdot B\_ - e \cdot C\_$. Posteriormente, Alice calcula um hash $e' = sha256(G||P_x||B\_||C\_||R1||R2)$ (Team, 2023b). Por fim, ela compara o resultado obtido no hash realizado e', com o valor e recebido da *mint*. Se a igualdade for confirmada, Alice validou que $B\_$ foi corretamente assinado pela *mint*, sem que Alice precisasse revelar x, ou a *mint* revelar k_x.


```
[
    {
        "amount": 8,
        "id": "000b4c3d8b0e7397",
        "secret": "[\"P2PK\", {\"nonce\": \"41514113b070f38b6bf299c2748028...",
        "C": "02e71e893e924e6b9d683b68d045a5c073d9bc960cb4c683a9c56bec85c94ec157"
    }
]
```


Figura 2.3 – Estrutura de dados do token

A construção apresentada acima, seguindo a implementação de *Wagner*, especifica o processo de geração de tokens (NUT-04 - Minting Tokens) assegurando que a *mint* assine Y sem conhecer seu conteúdo original, preservando a privacidade do usuário, enquanto a prova DLEQ permite verificar a integridade da assinatura de forma eficiente.

A construção apresentada acima, baseada na implementação do protocolo BDHKE de Wagner (Wagner, 1996), define o processo de geração de tokens conforme especificado na NUT-04 (Minting Tokens), assegurando que a *mint* assine Y sem conhecer seu conteúdo original, preservando a privacidade do usuário ao passo em que a prova DLEQ valida a integridade da assinatura de forma eficiente.

A Figura 2.4 abaixo ilustra o fluxo de geração e troca de tokens no Cashu, dividido em oito etapas principais. Inicialmente - primeira etapa, o usuário gera um segredo e computa um ponto cego ($B\_$), enviando-o à *mint*, que retorna uma assinatura cega ($C\_$) - segunda etapa. O usuário então descega o valor e valida o token emitido com uma prova DLEQ - terceira etapa. Após a validação, o token é transferido a outro usuário (Bob), que pode utilizá-lo ou passá-lo adiante - quarta etapa. Bob então gera um novo segredo e computa outro ponto cego ($B\_$) que só ele conhece - quinta etapa - enviando posteriormente à mint o token recebido de Alice (como input) e este novo segredo (como output) - sexta etapa. A mint responde à Bob com uma nova assinatura cega - sétima etapa, e ele realiza o descegamento, obtendo um novo token válido e invalidando o anterior de Alice - oitava etapa. O diagrama destaca a interação entre os usuários e a *mint*, oferecendo uma visão resumida e estruturada do caso de uso mais comum no Cashu.

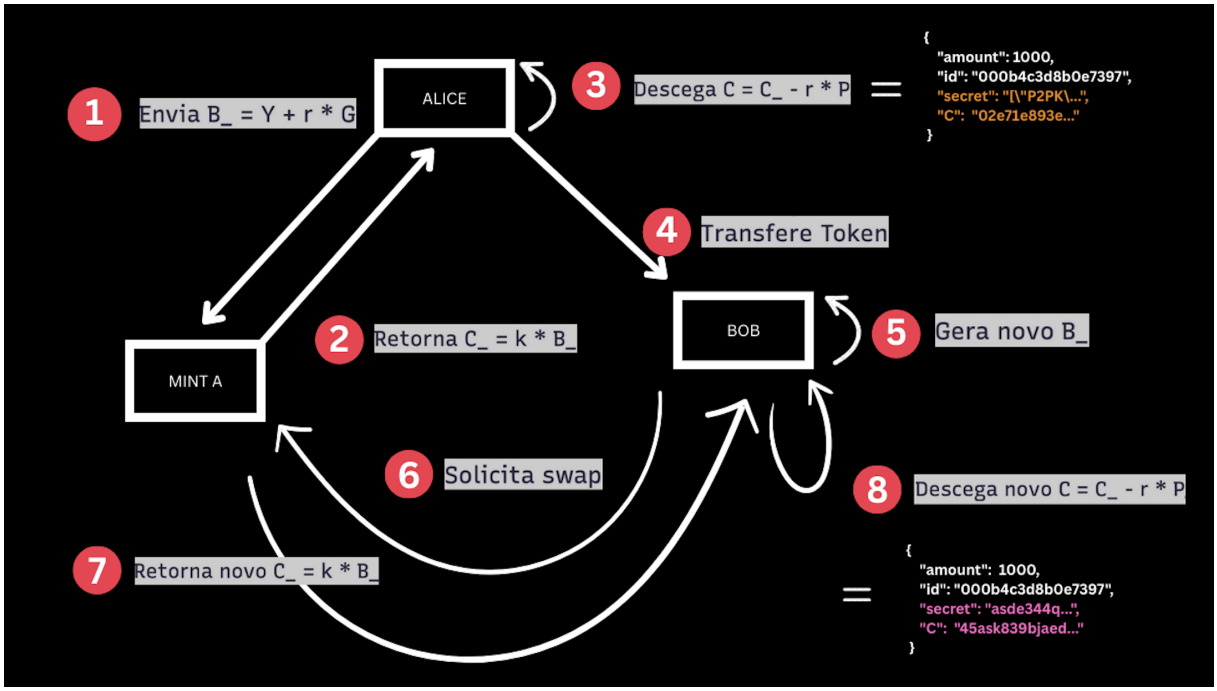

Figura 2.4 – Diagrama de fluxo - Mintagem (Alice) | Envio à Bob | Swap (Bob)

Avançaremos agora sobre o funcionamento geral das principais funcionalidades do Cashu, mais especificamente sobre a definição dos processos de geração dos *tokens*, gastos, trocas, verificação de estado, entre outros, que por sua vez são regidos por especificações chamadas Network Upgrade Transactions (NUTs), responsáveis por definir as operações criptográficas e operacionais do protocolo. As definições relevantes para o contexto apresentado, são:

- **Princípios Criptográficos (NUT-00)**: A NUT-00 estabelece a função *hash_to_curve*, que mapeia um *secret* (valor privado) a um ponto Y na curva elíptica secp256k1. Essa operação é crucial para a privacidade e integridade dos tokens, garantindo que o *secret* seja representado de forma única e segura. A NUT-00 também fundamenta o uso do BDHKE, adaptando o cegamento de Chaum para curvas elípticas (Wagner), essencial para o processo de assinatura cega.

- **Mintagem (geração) de Tokens (NUT-04)**: O usuário realiza um pagamento em Bitcoin via fatura da Lightning Network (NUT-04 (Team, 2023a)) para uma *mint*. Após confirmação do pagamento, o usuário gera o ponto cego $B\_ = Y + r \cdot G$, onde $Y = \text{hash\_to\_curve}(secret)$ é derivado de um *secret* privado, e r é um fator de cegamento aleatório escolhido pelo usuário. A *mint* assina $B\_$ com sua chave privada k_x, gerando $C\_ = k_x \cdot B\_$. O usuário descega $C\_$ calculando $C = C\_ - r \cdot P_x$, onde $P_x = k_x \cdot G$ é a chave pública da *mint*, obtendo a assinatura final.

- **Operação de Swap (NUT-03)**: A NUT-03 regula o *swap*, uma forma de gasto que permite trocas seguras e privadas de tokens entre o usuário e a *mint* que os emitiu, sendo utilizada majoritariamente para rebalanciamento de saldo (ex: o usuário possui um token de 200 sats, mas precisa de 150 sats para realizar um pagamento, então recorre à *mint* para substituir a unidade original por outras duas, uma de 50 sats e outra de 150 sats) ou para invalidação de um *token* recebido em pagamento de outro usuário (ex: Alice recebeu de Lana 21 mil sats em pagamento de um produto que vendeu - ou seja, recebeu o *Proof* formado pela estrutura de dados contendo: *amount*, *id*, *secret* e *C*. Entretanto, Lana ainda pode gastar esse token, bastando apresentá-lo à *mint*, e por essa razão, Alice realiza um *swap* na mint para trocar este token de 21 mil sats por outro igual, mas que Lana desconhece, invalidando o anterior).

- **Operação de Melt (NUT-05)**: A NUT-05 gerencia o *melt*, mecanismo que permite ao usuário reaver o colateral, trocando tokens *ecash* por *bitcoin*. Esse procedimento é iniciado pelo usuário gerando uma fatura Bolt-11 da Lightning Network no valor pretendido de resgate, e posteriormente informando-a à *mint* em conjunto aos dados do token que está sendo entregue em contrapartida. A *mint* valida os dados do token entregue, e então realiza o pagamento da fatura, que irá creditar o respectivo saldo em *bitcoin* na conta do usuário.

- **Verificação de Estado (NUT-07)**: A NUT-07 habilita a consulta do estado do token via *Y*, retornando *SPENT*, *PENDING* ou *UNSPENT*. Essa ferramente é particularmente útil para o protocolo de *swap* atômico pois se fará necessária a validação de disponibilidade do *token* durante a operação de troca.

- **P2PK (NUT-11)**: A NUT-11 define o mecanismo Pay-to-Public-Key (P2PK), uma condição de gasto que prevê a necessidade da apresentação protegendo de uma ou mais assinaturas válidas para que se concretize o gasto, aumentando a segurança em transações específicas e permitindo estabelecer diretrizes customizdas sob as quais um *token* pode ser gasto.

- **HTLC (NUT-14)**: A NUT-14, embora não utilizada no protocolo implementado, introduz o suporte a Hashed Timelock Contracts (HTLCs) no Cashu, definindo uma condição de gasto baseada em um segredo criptográfico (pré-imagem) e um prazo (*timelock*). Essa condição, uma extensão do formato P2PK (NUT-11), requer que o usuário forneça a pré-imagem correspondente ao hash armazenado no campo `Secret.data` e uma assinatura válida do campo `Secret.tags.pubkeys` para liberar o token, ou uma assinatura do campo `Secret.tags.refund` após o *locktime*.

2.2.3 Relevância para o Protocolo de Swap Atômico

A estrutura do Cashu, fundamentada em assinaturas cegas via BDHKE e tokens lastreados em Bitcoin, é a base para o protocolo de *swap* atômico proposto. A compreensão das etapas de mintagem, gasto (incluindo *swap*), validação de estado e *redeem* do *bitcoin* (através do *melt*), são fundamentais para que se compreenda a implementação de trocas seguras e atômicas entre usuários de *mints* distintas, superando o desafio de interoperabilidade no ecossistema.

2.3 Fundamentos Criptográficos

Esta seção apresenta os fundamentos criptográficos que sustentam o protocolo Cashu e o *swap* atômico proposto, com ênfase em curvas elípticas, assinaturas Schnorr e Adaptadoras, e técnicas como Diffie-Hellman, assinaturas cegas, BDHKE e DLEQ.

2.3.1 Criptografia de Curvas Elípticas

A criptografia de curvas elípticas - ECC - (Coates; Welsh, 2017) é uma abordagem amplamente utilizada nos algoritmos modernos, mais especificamente no Bitcoin, e, por via de consequência, também no Cashu. Oferecendo alta segurança com chaves de tamanho reduzido em comparação com sistemas tradicionais, como RSA, uma curva elíptica sobre um campo finito é definida pela equação:

$$y^2 = x^3 + ax + b \mod p$$

onde p é um número primo e a e b são constantes que satisfazem a condição $4a^3 + 27b^2 \neq 0 \mod p$. No contexto do Bitcoin e do Cashu, a curva *secp256k1* é adotada, com $a = 0$, $b = 7$ e $p = 2^{256} - 2^{32} - 977$. As operações principais incluem a **adição de pontos** e a **multiplicação escalar** ($k \cdot P$, onde P é um ponto da curva e k é um escalar), usadas para gerar chaves públicas e assinaturas digitais. A segurança da ECC reside na dificuldade do problema do logaritmo discreto em curvas elípticas (ECDLP), que é computacionalmente intratável para curvas bem projetadas.

2.3.2 Schnorr e Adaptor Signatures

As **assinaturas Schnorr** são um esquema de assinatura digital eficiente e seguro, implementado no Bitcoin por meio do BIP-340 (Proposal, 2020). Dada uma mensagem m, uma chave privada d e uma chave pública $P = d \cdot G$ (onde G é o ponto gerador da curva secp256k1), o processo de assinatura *Sign(sk, m)*, com *sk* sendo a chave privada do assinante

e m a mensagem é:

- Inicia-se por gerar um fator de aleatoriedade randômico a via HNG, RNG ou qualquer outro método para geração de aleatoriedade (entropia) preferível (um array de 32 bytes que será utilizado posteriormente).
- Calcula-se a chave pública customizada de 32 bytes[6] $P = d' \cdot G$, em que d' é o inteiro correspondente ao hexadecimal de 32 bytes da secret key, sk. Vale ressaltar ainda que d' deve ser menor que n[7] e diferente de 0.
- Assumimos um valor d, que será igual a d' caso P tenha paridade par. Se P tiver paridade ímpar, consideramos que $d = n - d'$.
- Realizamos o cálculo matemático para obtermos t com uso do operador xor (também conhecido como *ou exclusivo*: $bytes(d)\ XOR\ \text{hash}_{\text{BIP0340/aux}}\ (a)$. As operações de hash conforme a utilizada aqui são chamadas de hashes tagueados, que na especificação técnica seguem o formato SHA256(SHA256("tag")||(SHA256("tag") ||data), com a *tag* sendo o texto *BIP0340/aux* utilizado aqui nesta etapa e $data$ o input do hash, como (a) utilizado na presente explicação.
- Obtém-se $rand = \text{hash}_{\text{BIP0340/nonce}}\ (\ t||\ \text{bytes}(P)||m)$
- Assumimos $k' = int(rand) mod n$, que irá falhar se $k' = 0$
- Definimos $R = k' \cdot G$, que irá falhar se $k' = 0$
- Assumimos um valor k, que será igual a k' caso R tenha paridade par. Se P tiver paridade ímpar, consideramos que $k = n - k'$.
- Calculamos o desafio $e = \text{int}(\text{hash}_{\text{BIP0340/challenge}}(\text{bytes}(R)||\text{bytes}(P)||m) \mod n)$.
- Por fim, a assinatura é dada por $s = (r + e \cdot d) \mod n$, onde s é um escalar que combina o nonce e a contribuição da chave privada ajustada pelo desafio.
- A verificação é realizada confirmando que $s \cdot G = R + e \cdot P$, onde:
  - R é reconstruído como $R = \text{lift}_x(x(R))$, onde a função lift_x deriva o ponto completo (x, y) na curva elíptica secp256k1 a partir da coordenada x de R, calculando y como a raiz quadrada par da equação $y^2 = x^3 + 7$ (módulo $p = 2^{256} - 2^{32} - 977$), garantindo que a coordenada y seja par e consistente com o formato comprimido da BIP 340.
  - O desafio e é recalculado usando a mesma função tagueada, e a equação é validada para assegurar que a assinatura é consistente com a chave pública e a mensagem.

A função de hash tagueada é um mecanismo crítico para a segurança das assinaturas Schnorr, pois previne reutilização acidental de nonces ou colisões em diferentes contextos. Isso protege contra ataques relacionados a chaves derivadas de maneira não segura, como no BIP32, e assegura que cada assinatura seja vinculada exclusivamente ao contexto de uso.

As **Adaptor Signatures**, por sua vez, constituem uma extensão do esquema de assinaturas Schnorr, projetadas para criar assinaturas condicionais que dependem de um segredo criptográfico t. Em um grupo de ordem prima gerado por G - ponto gerador da curva *secp256k1*, com H sendo uma função *hash SHA256* tagueada, então uma assinatura Schnorr padrão para uma mensagem m e chave pública P é o par (R, s), satisfazendo a equação $s \cdot G = R + H(R\|P\|m) \cdot P$, onde $R = r \cdot G$ é um nonce e $s = r + H(R\|P\|m) \cdot k$ (com k a chave privada). Uma Adaptor Signature, em contrapartida, introduz um ponto adicional $T = t \cdot G$ na assinatura, resultando em um par $(R_{adaptor}, s_a)$, que obedece a equação $s_a \cdot G = R + H(R_{adaptor}\|P\|m) \cdot P$, onde $R_{adaptor} = R + T$.

Esse mecanismo cria uma assinatura inválida $(R_{adaptor}, s_a)$ por si só, pois T introduz uma dependência que só pode ser resolvida com o conhecimento de t, o segredo. A chave está na relação entre uma assinatura Schnorr (R, s) e sua contraparte adaptadora $(R_{adaptor}, s_a)$: ao subtrair as equações s e s_a, obtém-se $(s_a - s) \cdot G = T$, permitindo calcular o segredo t como o logaritmo discreto de T em relação a G (ou seja, $t = s_a - s$, ajustado pelos termos de hash). Assim, uma vez que (R, s) é revelado, t pode ser extraído utilizando-se $(R_{adaptor}, s_a)$, transformando-a em uma assinatura válida $(R_{adaptor}, s_a + t)$. Essa propriedade torna as Adaptor Signatures úteis para cenários onde a validade de uma assinatura depende de uma ação coordenada, como em trocas condicionais, sem exigir o uso de ferramentas externas ou de terceiros.

---

[6] Tradicionalmente, chaves públicas em sistemas como ECDSA (Burt, 2020) no Bitcoin são representadas no formato comprimido de 33 bytes, incluindo: (1) 1 byte inicial que indica a paridade da coordenada y (0x02 para y par, 0x03 para y ímpar) e (2) 32 bytes da coordenada x do ponto P. O formato X-only (32 bytes) na BIP-340 adota apenas os 32 bytes da coordenada x (chamada "X-only public key"), descartando o byte de paridade, pois a verificação Schnorr assume implicitamente que y é par (via $\text{lift}_x(x)$), conforme página 5 ("Implicit Y coordinates"). Para reutilizar chaves ECDSA existentes (ex.: de BIP32), a BIP-340 instrui remover o primeiro byte da codificação comprimida de 33 bytes, resultando em 32 bytes, o que é seguro pois a paridade é inferida na verificação. Detalhe técnico: se P tem y ímpar, a chave pode ser ajustada negando d (usando $n - d'$), mas o formato X-only simplifica armazenamento e compatibilidade.

[7] Na curva secp256k1, n representa a ordem do subgrupo gerado por G, um primo da ordem de aproximadamente $1.157920892373162 \times 10^{77}$, usado para modularizar os escalares nas assinaturas Schnorr.

2.3.3 Diffie-Hellman, Blind Signatures, BDHKE e DLEQ

O protocolo **Diffie-Hellman** (Diffie; Hellman, 1976) permite que duas partes estabeleçam uma chave secreta compartilhada por meio de um canal público. Em sua forma clássica, usa exponenciação modular ($(g^b)^a \mod p = (g^a)^b \mod p$), mas em curvas elípticas é adaptado para $(b \cdot G) \cdot a = (a \cdot G) \cdot b$.

Já em **Blind Signatures**, introduzidas por Chaum (Chaum, 1982), torna-se possível que um usuário obtenha uma assinatura de uma mensagem sem revelar seu conteúdo ao assinante, garantindo privacidade. No Cashu, isso é implementado por meio do **Blind Diffie-Hellman Key Exchange (BDHKE)**, uma variante que combina cegamento com Diffie-Hellman em curvas elípticas.

O **DLEQ (Discrete Logarithm Equality)** (**??**) é uma prova de conhecimento zero que verifica se duas assinaturas foram geradas com a mesma chave privada, sendo essencial para validar assinaturas cegas no Cashu sem comprometer a privacidade.

2.4 Contratos e Mecanismos Multisig

Esta seção explora os fundamentos dos contratos criptográficos e mecanismos multisig no contexto do Cashu, destacando sua relevância para o protocolo proposto. Inicialmente, apresenta-se uma definição geral de contratos, seguida por uma análise detalhada dos tipos de contratos suportados pelo Cashu e dos mecanismos multisig, conectando esses conceitos à implementação do *swap* atômico.

2.4.1 Definição de Contratos

Um contrato, no contexto de criptomoedas, é um conjunto de regras criptográficas que define as condições sob as quais fundos ou tokens podem ser gastos. Essas regras são implementadas como estruturas de dados ou scripts que especificam requisitos, como assinaturas válidas, segredos criptográficos ou condições temporais. No protocolo Cashu, os contratos podem ser aplicados aos tokens *ecash*, garantindo que o gasto ocorra apenas sob condições específicas, preservando a segurança e a privacidade das transações. Diferentemente de contratos inteligentes complexos em blockchains como Ethereum, os contratos do Cashu são leves (não touring complete), operando sem uma blockchain (*off-chain*) e otimizados para eficiência e anonimato.

2.4.2 Tipos de Contratos no Cashu

O Cashu suporta dois tipos principais de contratos, conforme especificado nas Network Upgrade Transactions (NUTs) (Team, 2023a): Pay-to-Public-Key (P2PK) e Hashed Timelock Contract (HTLC). Esses contratos são definidos na estrutura de dados do *Proof*, especificamente no campo *secret*, que contém as condições necessárias para satisfazer as exigências de gasto. Nas Figuras 2.5 e 2.6 que serão apresentadas adiante, observaremos as estruturas no formato JSON para ambas as variações de scripts apresentadas: P2PK e HTLC, respectivamente.


```
{
    [
        "P2PK",
        {
            "amount": 8,
            "nonce": "03e6552e6bcb0b82b706028ce4553654d2911baa49a51783c21486eaa6bdf693",
            "data": "02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447",
            "tags": [
                ["sigflag", "SIG_INPUTS"],
                ["n_sigs", "2"], ["locktime", "1750610181"],
                ["refund", "02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447"],
                ["pubkeys", "02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5"]
            ]
        }
    ]
}
```


Figura 2.5 – Estrutura de dados de um contrato P2PK


```
[
  "HTLC",
  {
    "nonce": "da62796403af76c80cd6ce9153ed3746",
    "data": "023192200a0cfd3867e48eb63b03ff599c7e46c8f4e41146b2d281173ca6c50c54",
    "tags": [
      [
        "pubkeys",
        "02698c4e2b5f9534cd0687d87513c759790cf829aa5739184a3e3735471fbda904"
      ],
      ["locktime", "1689418329"],
      [
        "refund",
        "033281c37677ea273eb7183b783067f5244933ef78d8c3f15b1a77cb246099c26e"
      ]
    ]
  }
]
```


Figura 2.6 – Estrutura de dados de um contrato HTLC

2.4.2.1 Pay-to-Public-Key (P2PK)

O contrato P2PK, descrito na NUT-11 (Team, 2023a), exige uma ou mais assinaturas digitais válidas correspondentes a uma ou mais chaves públicas específicas para gastar um token. O processo é:

- O *Proof* inclui um *secret* com um campo *data* que especifica uma chave pública e, opcionalmente, *tags* com condições adicionais, como o número de assinaturas necessárias (*n_sigs*). Adicionalmente, mais chaves de trancamento podem ser exigidas no campo *pubKeys*.
- Para gastar, o usuário fornece as assinaturas no campo *witness*, verificado pela *mint* contra as chaves usando a curva secp256k1.
- Adicionalmente, no campo *pubKeys* pode ser especificada também uma chave que tem o privilégio de gastar isoladamente o token após decorrido o tempo em *locktime*. Esse caso é particularmente útil para cenários de reversão mediante algum acontecimento, ou a ausência dele, como o que usamos em nossa arquitetura.

2.4.2.2 Hashed Timelock Contract (HTLC)

O contrato HTLC, por sua vez, combina uma condição de hash criptográfico com um limite de tempo (*timelock*). Ele exige que o possível destinatário, e detentor da chave pública especificada na tag *pubkeys*, apresente uma assinatura válida correspondente, acompanhada ainda da pré-imagem que, aplicada à função de *hash SHA-256*, resulte no mesmo valor apontado no campo *data*. Somente após o decurso do tempo de expiração, o proprietário da chave presente em *refund* pode gastá-lo, sem necessidade de apresentar a pré-imagem do *hash*. Este processo envolve algumas variáveis, como:

- *data*, que especifica o hash do segredo, e *locktime* (unix timestamp - quantidade de segundos decorridos desde 01 de janeiro de 1970).
- O campo *witness*, que deve conter a pré-imagem que satisfaz $H(\text{pré-imagem}) = \text{hash}$, onde H é SHA-256.
- *Timelock*, que ao expirar, uma chave pública de *refund* (especificada nos *tags*) pode reclamar o token.

2.4.2.3 Outros Contratos

De acordo com as especificações do Cashu (Team, 2023a) confirmamos que os tipos P2PK e HTLC são os únicos contratos suportados. Tokens do tipo *anyone-can-spend*, que não

possuem proteção contratual, não são considerados contratos, pois não impõem condições de gastos, permitindo que qualquer um que detenha o *Proof* do *token* emitido pela *mint* possa gastá-lo.

2.4.3 Mecanismos Multisig

Os mecanismos **multisig** (multi-assinatura) exigem que múltiplas assinaturas com chaves privadas distintas sejam fornecidas para autorizar uma transação, distribuindo a confiança entre as partes. No Cashu, o multisig é implementado em contratos P2PK por meio do *tag n_sigs*, que especifica o número de assinaturas necessárias de um conjunto de chaves públicas listadas no *tag pubkeys*. Por exemplo:

- Um contrato P2PK com $n\_sigs = 2$ e duas chaves públicas exige assinaturas de ambas as chaves privadas correspondentes.
- Já um contrato contrato P2PK com $n\_sigs = 5$ e três chaves públicas exige assinaturas uma combinação de 3 dessas chaves, quaisquer que sejam elas.
- As assinaturas são fornecidas no campo *witness* do *Proof*, verificadas pela *mint* durante o gasto.

O multisig é crucial para o presente protocolo de *swap* atômico, pois permite que Alice e Bob criem tokens com exigência de consentimento mútuo, garantindo interações sem confiança.

2.5 Mecanismos de Troca (Swap) em Criptomoedas

Esta seção examina os mecanismos de troca em criptomoedas, essenciais para a interoperabilidade entre blockchains, como no caso do protocolo Cashu, onde propusemos um *swap* atômico para trocas de tokens entre *mints* sem intermediários. Esses conceitos são viáveis em blockchains como o Bitcoin, servindo de base para o protocolo proposto, que visa trocas seguras e privadas.

2.5.1 Definição de Swaps

Um *swap* em criptomoedas é a troca direta de ativos digitais entre partes, na mesma ou em blockchains distintas, usando técnicas criptográficas para garantir segurança e atomicidade, ou seja, que a troca seja concluída integralmente ou não ocorra.

2.5.2 Atomic Swaps no Bitcoin com HTLC

Os *atomic swaps* no Bitcoin, baseados em Hashed Timelock Contracts (HTLCs) conforme BIP-199 (Community, 2015), permitem trocas sem confiança, como entre Bitcoin (BTC) e Litecoin (LTC). Um exemplo envolve Alice (que possui 1 BTC) e Bob (que possui 2.5 LTC).

O diagrama abaixo resume o fluxo do *swap* entre Alice (1 BTC) e Bob (2.5 LTC) nas testnets, mostrando a criação de HTLCs com o mesmo hash h, o gasto inicial de Alice revelando s, e o gasto de Bob, com reembolsos possíveis se os timelocks expirarem. Esse mecanismo inspira o *Protocolo de Swap Atômico* proposto para o Cashu, adaptando os princípios de HTLC para trocas de tokens.

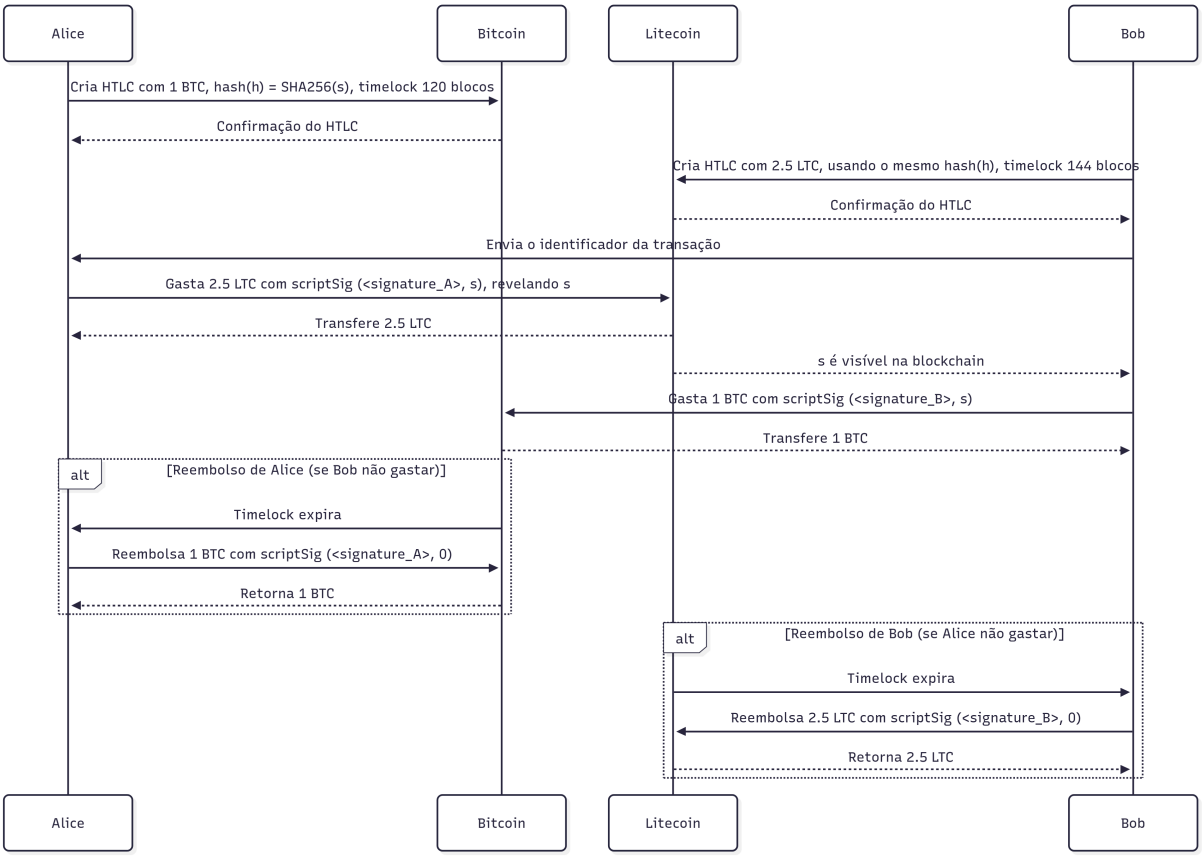

Figura 2.7 – Diagrama de Fluxo do Swap BTC/LTC

2.5.2.1 Processo Resumido

- **Negociação**: Alice e Bob acordam a troca, definindo um segredo s (ex.: "swapsecret123") e seu hash $h = \text{SHA256}(s)$, com timelocks ajustados (ex.: 10 min para Bitcoin, 2.5 min

para Litecoin).

- **HTLC de Alice**: Alice bloqueia 1 BTC em um P2SH com script que exige *s* e a assinatura de Bob, ou permite reembolso após o timelock com sua assinatura.
- **HTLC de Bob**: Bob bloqueia 2.5 LTC com script similar, exigindo *s* e a assinatura de Alice, ou reembolso após seu timelock.
- **Gasto por Alice**: Alice gasta os 2.5 LTC de Bob, revelando *s* publicamente.
- **Gasto por Bob**: Bob usa *s* para gastar os 1 BTC de Alice antes do timelock, completando a troca.

3 O Protocolo de Swap Atômico

Esta seção apresenta o Protocolo de *Swap* Atômico desenvolvido para o Cashu, detalhando sua operacionalização com base em assinaturas adaptadoras e mecanismos criptográficos acessórios. O objetivo é permitir trocas seguras e privadas de tokens entre usuários de *mints* distintas, garantindo atomicidade — ou seja, ambas as partes completam a troca ou nenhuma delas o faz — sem intermediários. A abordagem combina *Schnorr Signatures*, *Adaptor Signatures*, contratos P2PK (Multisig e de assinatura única), Provas de Conhecimento Zero - ZKP, e a infraestrutura da Lightning Network, mantendo a privacidade inerente ao Cashu por meio de Assinaturas Cegas e Blind Diffie-Hellman Key Exchange (BDHKE). Os códigos-fonte que implementam a construção do protocolo estão disponíveis publicamente nos repositórios https://github.com/szerwinski/Cashu-Alice, contendo os scripts relacionados às ações de Alice, e https://github.com/szerwinski/Cashu-Bob, dedicado às operações de Bob. Esses recursos permitem a replicação e verificação do protocolo, promovendo transparência e colaboração na comunidade de desenvolvimento do Cashu.

3.1 Adaptor Signatures e Funcionamento do Protocolo

3.1.1 Fundamentos das Assinaturas Adaptadoras

As *Adaptor Signatures*, uma extensão das assinaturas Schnorr descritas na BIP-340 (Proposal, 2020), permitem vincular um segredo criptográfico a uma assinatura digital, garantindo que sua validação revele este segredo. De acordo com a definição já apresentada, essa assinatura é formada por um par $(R_{\text{adaptor}}, s_a)$, onde $R_{\text{adaptor}} = R + T$, com $R = r \cdot G$ (nonce público do assinante), $T = t \cdot G$ (ponto público associado ao segredo t que foi escondido inicialmente), e $s_a = r + e \cdot k$, sendo r o nonce privado do assinante, d sua chave privada, e o desafio criptográfico construído a partir da especificação presente na BIP-340 evidenciada na seção 2.3.2, e, por fim, s_a como o escalar resultante desta operação.

Lembremo-nos, ainda, que o par $(R_{\text{adaptor}}, s_a)$ apresentado acima não constitui uma assinatura válida sobre a mensagem original, mas sim parte intermediária que proporciona o compartilhamento de um segredo sem confiança, de forma que, quando revelado, ambas as partes consigam atingir seus objetivos. A assinatura completa, portanto, é formada por (R_{adaptor}, s), em que $s = s_a + t$, é calculado por um dos participantes quando obtém a parcela faltante, e reciprocamente do outro lado, quando o primeiro realiza seu gasto, permitindo a atomicidade no *swap*.

Cada uma das etapas envolvendo o correto funcionamento da solução desenvolvida são apresentados, de forma numerada, na Figura 3.1 abaixo, e o detalhamento que as evolve é explicado em seguida na seção 3.1.2, identificados de igual maneira.

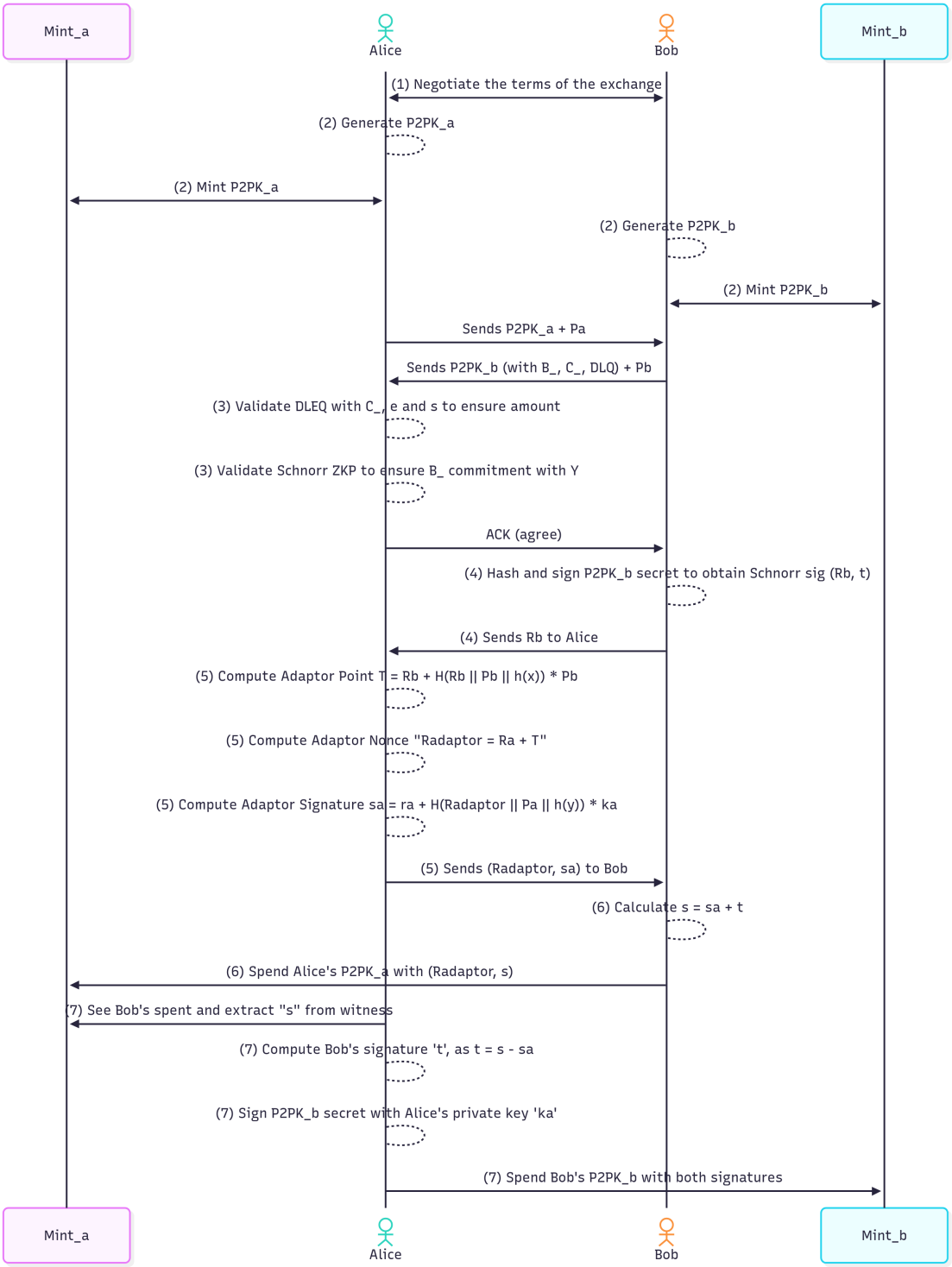

Figura 3.1 – Diagrama de Fluxo do Protocolo de Swap Atômico

3.1.2 Funcionamento do Protocolo

O protocolo de *swap* atômico no Cashu é projetado para que Alice e Bob troquem tokens de diferentes *mints* (ex.: 1000 satoshis da Mint A por 1000 satoshis da Mint B). A seguir, detalha-se o processo técnico, assumindo que ambas as *mints* operam como nós da Lightning Network.

1. **Negociação Inicial**:

   - Alice e Bob estabelecem as diretrizes da transação (ex.: 1 - serão trocados 1000 satoshis entre eles; 2 - Bob realizará o gasto primeiro; 3 - o *timelock*, tempo de expiração caso Alice não contribua, será de 24 horas após o aceite dos termos).

     Observação 3.1. O fator variável que implica em quem irá gastar primeiro o token é extremamente relevante para o caso em tela, pois, como apresentado no cenário de exemplo, Alice trava seu token recém-emitido em um contrato P2PK simples, que exige apenas sua assinatura para realizar o gasto. Já no caso de Bob, o token gerado por ele é travado em um P2PK *Multisig*, que exige tanto sua assinatura quanto a de Alice, especificando também um prazo de expiração que permite a Bob reaver seu token caso Alice não colabore. Este mecanismo evita que Bob, ao se apropriar do token de Alice, recupere o token que ele mesmo gerou, lesando a outra parte. Caso a prioridade de um dos participantes se invertesse, os contratos também deveriam ser gerados de forma contrária.

2. **Criação dos Tokens e Contratos**:

   - Alice solicita uma cotação para gerar 1.000 satoshis em *tokens* de *ecash* na *Mint A* através do *endpoint* "POST https://mint.host:3338/v1/mint/quote/bolt11", informando como entrada o *amount* e a unidade de conversão *unit*, de acordo com a estrutura apresentada abaixo:

     ```
     {
          "amount": 1000,
          "unit":  "sat"
     }
     ```

   - O retorno da mint para a solicitação acima contém informações necessárias para que Alice confirme os dados e realize o pagamento, sendo algumas delas: *quote*

- um identificador único da cotação; *request* - a *url* de pagamento para a fatura *Bolt11*; *amount* - a quantia a ser obtida; *unit* - a unidade de conversão (em nosso caso, "sat", acrônimo para satoshis); *expiry* - validade da cotação. A disposição dos dados obtidos em resposta é apresentada a seguir:

```
{
    "quote": "113MTeKDDGsb2IG3...",
    "request": "lnbc600n1p5xrzfu9qypqqqdq...",
    "amount": 1000,
    "unit": "sat",
    "state": "UNPAID",
    "expiry": 1751226188,
    "pubkey": null,
    "paid": false
}
```

- Alice envia o pagamento de 1000 satoshis via Lightning Network, utilizando a *request* recebida no passo anterior.
- Enquanto o pagamento da fatura não é confirmado, Alice prepara os dados para a criação do token, iniciando pela construção do ponto $Y = hash\_to\_curve(secret_a)$, sendo $secret_a$ os bytes serializados do contrato P2PK que ela definiu para estabelecer as condições de gasto do seu *token*, apresentado abaixo, em que o *nonce* corresponde a um fator de unicidade do token, *data* é a chave pública de Alice, e nas *tags* é especificada uma obrigação de assinatura contra essa chave pública no ato do gasto:

```
[
    "P2PK",
    {
        "nonce": "41514113b070f38b6bf299...",
        "data": "02c776bf1be8e58e9b900...\",
        "tags": [
            ["sigflag", "SIG_INPUTS"]
        ]
    }
]
```

- Posteriormente, com Y calculado, ela deriva $B\_ = Y + r \cdot G$, sendo r um *nonce* aleatório obtido a partir da curva *secp256k1*.
- Com os dados calculados acima em mãos, ela solicita a criação dos *tokens* através do *endpoint* "POST https://mint.host:3338/v1/mint/bolt11", fornecendo a estrutura a seguir na solicitação:

```
{
    "quote": "113MTeKDDGsb2IG3...",
```

```
    "outputs": [
        {
            "amount": 1000,
            "id": "000b4c3d8b0e7397",
            "B_": "02bc581a5731bfc17e267d7..."
        }
    ]
}
```

- A mint responde à ela com um registro contendo $(amount, id, C\_, DLEQ)$, que correspondem, respectivamente, ao valor do token, id do keyset que contém a chave utilizada pela mint para assinar cegamente a transação, assinatura cega da mint (onde $C\_ = k_x \cdot B\_$, com $B\_ = Y + r \cdot G$ — sendo r o nonce privado gerado por Alice, $Y = hash\_to\_curve(secret_a)$, e k_x a chave privada da Mint A para o amount especificado) e a prova de conhecimento zero para que seja conferida a validade desta assinatura. Os dados mencionados são apresentados de acordo com a seguinte estrutura:

```
{
    "id": "000b4c3d8b0e7397",
    "amount": 1000,
    "C_": "0200c648ecf13fe818...",
    "dleq": {
        "e": "ab73fdf4f051fb87ca4...",
        "s": "d403a51095960b9..."
    }
}
```

Definição 3.1. No Cashu, as mints trabalham com dicionários formados por pares de chaves públicas (abertas à consulta) e privadas (confidenciais e armazenadas secretamente). Os mesmos são chamados de *keysets*, que são nada mais nada menos que estruturas de dados com listas relacionando cada chave a um *amount* especifico. Dessa forma, o usuário deve especificar qual dos "acervos"de chaves da mint ele deseja que a mesma utilize para realizar a assinatura, ou seja, em qual *keyset* está a chave privada associada ao amount do token mintado.

Definição 3.2. A DLEQ no Cashu é uma prova de conhecimento zero que verifica, sem a mint revelar sua chave privada k, se a assinatura $C\_ = k \cdot B\_$ foi gerada com a *privateKey* k correspondente à chave pública $P = k \cdot G$. O cliente envia $B\_ = Y + r \cdot G$, e a *mint* retorna $C\_$ e a prova DLEQ com e = ab73fdf4f051fb87ca4... e s = d403a51095960b9....

Funcionamento: Mint escolhe w (um nonce aleatório), calcula $R_1 = w \cdot G$,

$R_2 = w \cdot B\_$ e compartilha $e = \text{Hash}(G, P, B\_, C\_, R_1, R_2)$ e $s = w + e \cdot k \mod n$, retorna (e, s). O cliente calcula $R'_1 = s \cdot G - e \cdot P$, $R'_2 = s \cdot B\_ - e \cdot C\_$, verificando se $e = \text{Hash}(G, P, B\_, C\_, R'_1, R'_2)$. Caso seja válido, confirma que $C\_$ é uma assinatura autêntica de $B\_$ com k.

- Por fim, Alice realiza o processo de descegamento da assinatura $C\_$ obtida, através da operação $C = C\_ - r \cdot Px$, em que Px é a chave pública da mint, e constrói a estrutura de dados final do *Proof*, formada pelo conjunto $(amount, id, secret, C)$, sendo os dois últimos correspondentes a condição de gasto estabelecida no contrato e assinatura descegada, respectivamente.
- Bob realiza as mesmas etapas com a sua Mint B, ressalvando, porém, a diferença existente no contrato P2PK estabelecido por ele (Multisig) e Alice (simples). A estrutura de dados do contrato P2PK Multisig de Bob se apresenta da seguinte forma, em que o *nonce* representa o fator de unicidade, *data* representa a chave pública de Bob, *sigflag* especifica que devem ser apresentadas assinaturas correspondentes a todas as chaves de travamento, *n_sigs* define que o contrato precisa de 2 assinaturas válidas (Alice e Bob) para ser gasto antes do *locktime*, que por sua vez apresenta o marco temporal a partir do qual o token pode ser gasto apresentando-se apenas a assinatura válida para a chave em *refund* - chave de Bob, e, por fim, em *pubKeys* é especificada a chave pública de Alice:


```
[
    "P2PK",
    {
        "nonce": "ca1068e138f0fcc823...",
        "data": "02c15e12abf164078fd11...",
        "tags": [
            ["sigflag", "SIG_INPUTS"],
            ["n_sigs", "2"],
            ["locktime", "1751933852"],
            ["refund", "02c15e12abf16407..."],
            ["pubkeys", "02c776bf1be8e58e9b..."]
        ]
    }
]",
```


- Ambos compartilham entre si os *Proofs* dos *tokens* mintados em seus respectivos provedores de serviço, bem como as chaves públicas efêmeras que geraram para esta transação (chaves efêmeras referen-se àquelas geradas única e exclusivamente para uma transação, e não são reutilizadas por questões de segurança). Alice envia uma estrutura de dados contendo: *amount*, *id*, *secret* e *C*, enquanto Bob compartilha essa mesma estrutura acrescida de $B\_$, $C\_$ e a prova DLEQ do

seu token, composta por e e s, dados agregados pela necessidade de conferir segurança à Alice no sentido de confirmar o *amount* do token. Não obstante à prova de conhecimento zero que a própria mint de Bob o retornou após a criação do token (e foi transmitida para Alice junto a $B\_$ e $C\_$), ele cria também uma *ZKP (Zero Knowledge Proof)* baseada em *Schnorr* para demonstrar à Alice que o $B\_$ informado foi gerado a partir da equação $B\_ = Y + r \cdot G$, para algum r escolhido por Bob; nesse caso Alice já conhece o Y, pois o calcula a partir da função *hash_to_curve* aplicada ao secret do token. A motivação por trás dessas informações adicionais, bem como sua fundamentação teórica, serão explicadas em mais detalhes adiante. As estruturas de dados dos objetos *JSON* que os participantes trocam entre si podem ser observadas na Figura 3.1.

token a, emitido por Alice


```
{
    "amount": 1000,
    "id": "000b4c3d8b0e7397",
    "secret": "[\"P2PK\", {\"amount\": 8, \"nonce\": \"18d25291c4dfe01cb...\",
      \"data\": \"02c776bf1be8e58e9b...\", \"tags\": [[\"sigflag\", \"SIG_INPUTS\"]]}]",
    "C": "03b93da6142789871b45d9bb042bda5309e5946a95abc4b2f194ed6f2705cfebee"
},
```


token b, emitido por Bob


```
{
    "amount": 1000,
    "id": "00afe2bda7e0855b",
    "secret": "[\"P2PK\", {\"amount\": 8, \"nonce\": \"03e6552e6bcb0b...\", \"data\":
      \"02c15e12abf1640...\", \"tags\": [[\"sigflag\", \"SIG_INPUTS\"], [\"n_sigs\",
      \"2\"], [\"locktime\", \"1750610181\"], [\"refund\", \"02c15e12abf16407...\"],
      [\"pubkeys\", \"02c776bf1be8e58e...\"]]}]",
    "B_": "0386d0d7690c284bdbad41b3029069e2ad00dec85236913ed5345c14fc9c9ab5c5",
    "C": "03063d354dc805157d7b20b153442132815f4c535a819860a2ed3b7e145d7ea36d",
    "C_": "028c7b059f39150d5604e8d97dfc1e5f14f2e95aa1f8b0ed6d3442f226029f23a2",
    "dleq": {
        "e": "05d6d8b5e5c2b8eb96cb00e94b1afff902f01d9df0c41200c47f76569a420f17",
        "s": "802f33a1a373c5ff37798ff012f3f196b0a49a546172ad2a3d60e99cdb55e40c"
    }
}
```


Figura 3.2 – Estruturas de dados dos tokens emitidos por Alice e Bob

Observação 3.2. Ressaltamos ainda que, conforme já apresentado anteriormente, a diferença principal entre os dados dos *tokens* reside no campo *secret*, onde é especificado o contrato de travamento dos fundos (Alice - P2PK com uma assinatura; Bob - P2PK Multisig com 2 assinaturas e condição de refund).

3. **Prevenção de Alice contra possível Falsificação por Bob**:

   Até o presente momento, Alice e Bob concordaram acerca das condições sob as quais a troca seria realizada, mintaram seus tokens e compartilharam os *Proofs* um com o outro, sendo que Bob agregou algumas informações a mais nos dados transmitidos à Alice, $B\_$, $C\_$ e a prova DLEQ do seu token.

   No protocolo de *swap* atômico proposto, Bob deve compartilhar com Alice $B\_$, $C\_$ e a prova DLEQ (e = ab73fdf4f051fb87ca4..., s = d403a51095960b9...) do token mintado, além de uma prova de conhecimento zero (ZKP) **Schnorr** para que Alice possa validar duas condições principais, sendo elas: 1 - o *amount* (ex.: 1000 satoshis) do *Proof* recebido de Bob; 2 - a validade da derivação de $B\_$ a partir de Y, conhecido por Alice.

   Essa troca visa garantir que Alice receba um token legítimo. Entretanto, Bob pode ter motivação para forjar dados, como criar um token fictício para lucrar sem custos reais ou enganar Alice com um valor inexistente. Os dois cenários em que Bob poderia tentar surrupiá-la serão abordados a seguir, bem como as precauções adotadas para não permitir que o referido aconteça.

   Motivação e tentativa de falsificação para burlar o *amount*:

   A única maneira que Alice tem para verificar a validade do amount recebido no *Proof* de Bob frente ao que fora acordado inicialmente é conferindo a DLEQ, pois esta demonstra criptograficamente que $C\_$ foi gerado a partir de $B\_$ com k_x - a chave privada da mint - sem revelar k_x. A referida validação é necessária e suficiente para conferir o *amount*, pois k_x é uma chave privada da mint exclusiva para essa quantia (a mint assina cada token com uma chave atrelada ao seu valor), e a comprovação matemática que valida a procedência de $C\_$ frente a k_x se mostra competente em cumprir essa demonstração.

   Neste sentido, Bob poderia tentar falsificar $C\_$ e a DLEQ para inventar um token cuja verificação matemática fosse procedente, mas não corresponderia à uma unidade legítima gerada pela Mint B.

   Por exemplo, ao invés de usar $C\_ = k_x \cdot B\_$ (onde k_x é a chave privada da *mint* e $P_x = k_x \cdot G$ é a chave pública), ele poderia gerar $C'' = k' \cdot B\_$ com uma chave k' que ele controla, e criar uma nova DLEQ. O protocolo DLEQ funciona assim:

- A *mint* originalmente escolhe um nonce w, calcula $R_1 = w \cdot G$, $R_2 = w \cdot B\_$, e define $e = \text{Hash}(G, P_x, B\_, C\_, R_1, R_2)$, com $s = w + e \cdot k_x \mod n$ (onde n é a ordem da curva secp256k1)
- Bob, ao tentar fabricar dados para forjar a validação de igualdade do logaritmo discreto, usaria C'', recalculando $e' = \text{Hash}(G, P_x, B\_, C'', R_1, R_2)$ e $s' = w + e' \cdot k' \mod n$, enviando (e', s') a Alice.

Validação e Falha da Falsificação

Alice validaria a DLEQ forjada calculando:

$$R_1' = s' \cdot G - e' \cdot P_x$$

$$R_2' = s' \cdot B\_ - e' \cdot C''$$

e verifica se $e' = \text{Hash}(G, P_x, B\_, C'', R_1', R_2')$. Substituindo $s' = w + e' \cdot k'$:

$$R_1' = (w + e' \cdot k') \cdot G - e' \cdot P_x = w \cdot G + e' \cdot k' \cdot G - e' \cdot k_x \cdot G$$

Para R_1' ser igual a R_1, temos que $R_1' = R_1 = w \cdot G$ (o compromisso original), deve-se ter:

$$w \cdot G = w \cdot G + e' \cdot (k' - k_x) \cdot G$$

Isso implica $e' \cdot (k' - k_x) = 0$. Como $e' \neq 0$ (hash não-trivial), segue que $k' = k_x$, o que Bob não pode satisfazer, pois k_x é secreto. Assim, $R_1' \neq R_1$, e a verificação falha, expondo a falsificação.

Validação de $B\_$ com ZKP Schnorr

Para $B\_ = Y + r \cdot G$, Bob fornece uma ZKP:

escolhe um fator aleatório k, calcula $R = k \cdot G$, $e = \text{Hash}(R, Y, B\_)$, e $s = k + e \cdot r$. Alice verifica:

$$s \cdot G = R + e \cdot (B\_ - Y)$$

Substituindo $s = k + e \cdot r$:

$$s \cdot G = (k + e \cdot r) \cdot G = k \cdot G + e \cdot r \cdot G = R + e \cdot (B\_ - Y)$$

Como $B\_ - Y = r \cdot G$ (conforme a definição de cegamento), a equação $s \cdot G = R + e \cdot (B\_ - Y)$ se mantém válida. A verificação do hash $e = H(R \| Y \| B\_)$ assegura a integridade da prova. Bob não pode falsificar a relação $B\_ = Y + r \cdot G$ usando um Y ou r inválido sem

que a ZKP detecte a inconsistência, e a prova garante a validade de $B\_$ sem revelar o segredo r. Essa abordagem foi escolhida como precaução, pois os riscos de Bob compartilhar r não foram investigados.

Por fim, concluímos que a DLEQ impede Bob de forjar $C\_$ sem k_x, e a ZKP valida $B\_$ sem expor r, garantindo que Alice receba um token legítimo com a segurança de um *amount* correto e o vínculo de $B\_$ com Y atestado. A segurança reside na dificuldade do problema de logaritmo discreto.

4. **Geração da *Schnorr Signature* de Bob**:

   - Dando continuidade no processo transacional após demonstradas possíveis fragilidades que foram contornadas com a agregação de mais informações, e com Alice tendo validado algebricamente as demonstrações e de acordo em seguir, Bob agora gera uma assinatura *Schnorr* padrão para seu token:

     - Calcula $t = rb + H(Rb||Pb||h(x)) \cdot kb$, sendo:
     - **rb**: nonce privado de Bob (32 bytes obtidos na curva *secp256k1*).
     - **Rb**: nonce público, calculado como $Rb = rb \cdot G$.
     - **Pb**: sua *pubKey*.
     - **kb**: sua *privateKey*.
     - **x**: os bytes serializados do contrato P2PK *Multisig* criado por ele, também chamado de *secret*.
     - **h(x)**: *hash sha256* do secret.
     - A assinatura final obtida é formada pelo par (Rb, t), sendo o primeiro um ponto sobre a curva elíptica, e o segundo um escalar pertencente ao campo finito. Esta é uma assinatura válida para que seja gasto o token de Bob, mas por ser uma unidade travada por meio do contrato P2PK *multisig*, deve ser apresentada em conjunto com a assinatura de Alice. Por esta razão, ele não pode enviar o par (Rb, t) à ela, apresentando portanto apenas Rb.
   - O ponto público Rb é compartilhado com Alice.

5. **Geração da *Adaptor Signature* por Alice**:

   - Após receber o *nonce* público de Bob (Rb), Alice calcula o *Adaptor Point* $T = Rb + H(Rb||Pb||h(x)) \cdot Pb$. Repare que, a essa altura do campeonato, Alice possui as informações necessárias para realizar a operação: o nonce público foi enviado à ela por Bob, assim como sua *pubKey*; o secret também é conhecido, pois está

presente no *Proof* do token de Bob.

- Em segundo momento, e de posse do valor T, é computado agora o *Adaptor nonce*, identificado como $Radaptor$. Esta variável é calculada a partir da operação $Radaptor = Ra + T$, em que Ra refere-se ao nonce público de Alice, obtido pela multiplicação de seu nonce privado pelo ponto gerador da curva, $Ra = ra \cdot G$.
- Por fim, Alice calcula a *Adaptor Signature*, obtida por meio da equação $sa = ra + H(Radaptor||Pa||h(y)) \cdot ka$. Interessante reparar que, por definição, se esta equação fosse utilizada como fundamento para se construir uma assinatura *Schnorr* padrão, o resultado seria o par $(Radaptor, sa)$. É perceptível que essa construção não satisfaz a condição de validade para se obter uma assinatura válida, e podemos observar esse fenômeno ao tentar validá-la:

$$
\begin{aligned}
(R_{\text{adaptor}}, s_a) \longrightarrow s &= s_a + t \\
s_a &= r_a + H(R_{\text{adaptor}}||P_a||h(y)) \cdot k_a \\
s_a \cdot G &= r_a \cdot G + H(R_{\text{adaptor}}||P_a||h(y)) \cdot P_a \\
&\neq R_{\text{adaptor}} + H(R_{\text{adaptor}}||P_a||h(y)) \cdot P_a \\
&\text{pois } R_{\text{adaptor}} = R_a + T \text{ e } s_a \text{ não inclui } t.
\end{aligned}
$$

- O fator de interesse, e que constrói o compromisso de Alice com Bob para promover a atomicidade da troca, reside, portanto, na assinatura *Schnorr* modificada que chamamos de *Adaptor Signature*. Nela, escondemos dentro do nonce público do hash um segredo T somado a Ra, gerando $Radaptor$. Destarte, por si só a assinatura não é válida. Para construir a assinatura final de Alice, Bob precisa, portanto, conhecer s, que pode ser calculado quando Alice compartilhar com ele o par $(Radaptor, sa)$, pois $s = sa + t$. De posse desses valores, Bob formará o par $(Radaptor, s)$, que será uma assinatura funcional de Alice para o *hash do secret* y, como veremos adiante.

6. **Gasto do token de Alice por Bob**:

- Bob recebe o par $(Radaptor, sa)$ de Alice e, como apresentado anteriormente, consegue realizar a operação $(s = s_a + t)$, pois já conhece t, que ele mesmo gerou, e na etapa anterior recebeu sa.
- Formando o par $(Radaptor, s)$, Bob possui agora uma assinatura válida de Alice sobre o secret do token que ela mintou. Podemos confirmar essa informação com a evidência abaixo:

$$
\begin{aligned}
(R_{\text{adaptor}}, s_a) \rightarrow s &= s_a + t \\
s_a &= r_a + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot k_a \\
s \cdot G &= (s_a + t) \cdot G \\
&= [r_a \cdot G + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot P_a] + t \cdot G \\
&= R_a + T + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot P_a \\
&= R_{\text{adaptor}} + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot P_a
\end{aligned}
$$

- Por fim, Bob agora utiliza a NUT-03 do Cashu para realizar um *swap*, fornecendo como entrada o token de Alice, com sua respectiva assinatura obtida acima, e como saída uma nova *Proof* que ele mesmo criou. Desta forma, ele invalida (gasta) o token emitido inicialmente por Alice e recebe um novo token do qual ela não tem informações, no valor pretendido.
- Para realizar esse processo, é utilizado o *endpoint* "POST https://mint.host:3338/v1/swap". A resposta da mint segue o mesmo padrão apresentado anteriormente para quando ambos mintaram seus tokens. Já para realizar o swap, deve ser fornecida como entrada a estrutura abaixo (repare que no input passamos a assinatura de Alice que Bob calculou, para atender à condição de gasto do P2PK):

```
{
  "inputs": \\ Token mintado por Alice
    [
      {
        "amount": 1000,
        "id": "000b4c3d8b0e7397",
        "secret": "407915bc212be61a7...",
        "C": "02bc9097997d81afb2cc...",
        "witness": {
            "signatures": ["283nanlaju88na..."]
        }
      }
    ],
  "outputs": \\ Nova estrutura construída por Bob
    [
      {
        "amount": 1000,
        "id": "novo id do keyset escolhido por Bob",
        "B_": "02634a2c2b34bec9..." \\ Novo B_ calculado por Bob
      }
    ],
}
```

7. **Gasto do token de Bob por Alice**:

- Durante a operação, Alice deve monitorar sua Mint A para consultar o status do token que emitiu. Isso é realizado por meio do uso da NUT-07, através da

url "POST https://mint.host:3338/v1/checkstate". Tal método pode ser chamado fornecendo-se o parâmetro Y como entrada, fator gerado por Alice inicialmente. A mint devolve como resposta 3 variáveis: Y - o próprio valor consultado; *STATE*: um enumerador que possui três condições possíveis (SPENT, UNSPENT e PENDING); *witness*: assinatura(s) apresenta(s) no gasto, se for o caso.

- Inicialmente, antes da transação acontecer, a consulta apresentada acima retornaria para Alice o status $UNSPENT$ com o atributo *witness* vazio. Uma vez que Bob realize a operação de *swap* descrita no passo anterior, o retorno será com status $SPENT$ e no *witness* estará presente a assinatura que ele usou para gastá-lo, s.
- Com s em mãos, Alice agora pode calcular t a partir da fórmula $t = s - sa$. De posse desse valor, que é a assinatura de Bob sobre o token que ele mintou inicialmente, Alice agora tem condições de obter sua parte.
- Da mesma forma que apresentamos para Bob, ela constrói a estrutura de dados com o *input* e *output*, fornecendo no *witness* da entrada não só a assinatura t de Bob, mas também a sua própria assinatura sobre o hash aplicado ao secret do $Proof b$.
- A troca atômica é concluída e ambas as partes são agora detentoras de seus respectivos tokens.

8. **Reembolso e Falhas**:

   - Se Alice não informar a Bob o par $(Radaptor, sa)$ por alguma razão, Bob pode reembolsar seu token com a Mint B após o decurso do prazo acordado entre eles inicialmente, atendendo à condição de *refund* apresentada no token que ele mintou. A troca não acontece, mas nenhuma das partes é lesada.
   - Se Bob receber o par e mesmo assim não realizar o spending com a operação de *swap*, Alice ainda tem a propriedade sobre seu token, e Bob pode reaver o seu depois de locktime.
   - Como discutido, qualquer que seja o caso, nenhuma das partes pode ser prejudicada em face de um mal comportamento ou malícia da outra, preservando a atomicidade e segurança da transação.

4 Evidências - Demonstração e Validação de Funcionamento do Código

A presente seção tem como objetivo demonstrar e validar o funcionamento do código implementado para o protocolo de *swap* atômico entre Alice e Bob, utilizando os conceitos teóricos discutidos nas seções anteriores. Através de uma série de scripts Python, acompanhados de logs de execução otimizados, esta seção ilustra cada etapa do processo, desde a geração de tokens até a conclusão do *swap*.

4.1 Mintagem por Alice

Alice inicia o processo gerando um token de 1000 sats, utilizando o script inicial `gen_p2pk_outputs.py`. Este script é responsável por criar o ponto Y a partir do secret P2PK, calcular o ponto cego $B\_$ com um fator de cegamento r, e interagir com a mint para obter os proofs correspondentes. O processo começa com um contrato P2PK hardcoded, estruturado como um dicionário Python serializado em bytes, contendo um nonce aleatório, a chave pública de Alice, e tags como `sigflag` para indicar a intenção de entrada, servindo como base para a geração do ponto Y. O fluxo de mintagem, apresentado na Figura 4.1 abaixo, ilustra cada uma destas etapas, cuja codificação será evidenciada em mais detalhes em seguida.

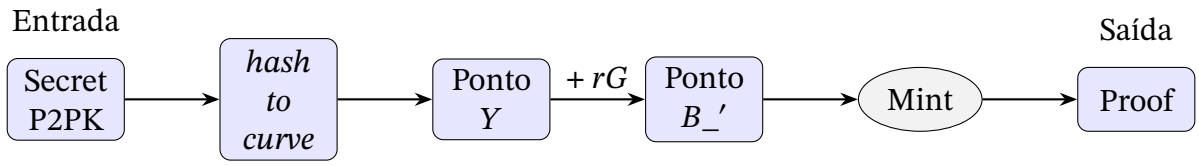

Figura 4.1 – Fluxo de mintagem de tokens por Alice.

Trecho de Código: Geração de Y e $B\_$

```python
# Importa dependencias

import hashlib
import secrets
import requests
import json
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Define variáveis auxiliares
```

```
outputs = [] # Array de outputs
secrets_list = [] # Array de secrets
r_scalars = [] # Array de nonces

# Definição da função hash_to_curve para converter o secret em um ponto Y
def hash_to_curve(x: bytes, counter=0) -> secp256k1.PublicKey:
    msg_to_hash = hashlib.sha256(DOMAIN_SEPARATOR + x).digest()
    while counter < 2**16:
        _hash = hashlib.sha256(msg_to_hash + counter.to_bytes(4,
            "little")).digest()
        try:
            Y = secp256k1.PublicKey(b"\x02" + _hash, raw=True)
            return Y
        except Exception:
            counter += 1
    raise ValueError("No valid point found after 2**16 iterations.")

# Gerar o secret no formato P2PK

nonce = secrets.token_bytes(32).hex()
p2pk_secret = [
    "P2PK",
    {
        "nonce": nonce,
        "data": alice_public_key,
        "tags": [["sigflag", "SIG_INPUTS"]]
    }
]
secret_str = json.dumps(p2pk_secret)  # Serializa como string JSON
secret_bytes = secret_str.encode('utf-8')  # Converte para bytes

secrets_list.append(secret_bytes) # Adiciona na lista

# Geração de Y usando hash_to_curve
Y = hash_to_curve(secret_bytes)

# Geração de r (fator de cegamento aleatório) e cálculo de B_ = Y + r * G
r = secp256k1.PrivateKey()
# r é um escalar aleatório usado para cegamento
r_scalar = int.from_bytes(r.private_key, 'big')
r_scalars.append(r_scalar) # Adiciona na lista

B_ = Y + r.pubkey  # B_ é o ponto cego, somando Y com r * G
B_serialized = B_.serialize().hex()

# Adiciona a estrutura de dados ao array do output
outputs.append({
    "amount": amount,
    "id": keyset_id,
    "B_": B_serialized
})

# Salva os outputs, secrets e r_scalars em um arquivo
```

```
output_data = {
    "outputs": outputs,
    "secrets": [s.hex() for s in secrets_list],  # Armazena como hex
    "r_scalars": r_scalars,
    "Ys": Ys, # Pontos Y
}
with open("p2pk_outputs_data.json", "w") as f:
    json.dump(output_data, f, indent=4)

# Fazer cotação
response = requests.post(f"{MINT_URL}/v1/mint/quote/bolt11", json={"unit":
    "sat", "amount": sum(amounts)})
quote_data = response.json()
quote_id = quote_data["quote"]

# Enviar os outputs para a mint
mint_response = requests.post(f"{MINT_URL}/v1/mint/bolt11", json={"quote":
    quote_id, "outputs": outputs})
proofs = mint_response.json()

# Salvar os proofs retornados pela mint
with open("p2pk_proofs_response.json", "w") as f:
    json.dump(proofs, f, indent=4)
```

Explicação do Trecho: Este código inicia pela definição da função *hash_to_curve*, que converte o secret (contrato P2PK) em um ponto Y na curva Secp256k1 por meio de um hash iterativo em *loop*. Posteriormente é formatada a estrutura do contrato P2PK de Alice (sem multisig) contendo: *nonce* - um segredo aleatório de 32 bytes, *data* - chave pública de Alice e *tags* - parâmetros adicionais do contrato, que, no caso em tela, exige assinatura em todos os inputs.

Em seguida, realiza-se a operação de hash (sha256) entre o $DOMAIN\_SEPARATOR$, que é uma *string* conhecida e pré-determinada no protocolo Cashu para evitar colisões: "Secp256k1_HashToCurve_Cashu_"e o secret (contrato P2PK elaborado por Alice), ambos serializados em bytes previamente. Na sequência, a mesma operação de hash é realizada, agora entre o resultado da etapa anterior e o *counter*, inteiro iniciado em 0 e incrementado um a um, até que se obtenha um ponto válido sobre a curva ao adicionar o byte $0x02$ de paridade par.

Após essa etapa, um fator de cegamento aleatório r é criado (32 bytes), e seu componente público $(r \cdot G)$ é somado a Y para obter o ponto cego $B\_$, que será enviado à mint para cegamento. Os dados referentes ao secret, nonce (r_scalar), Y e a estrutura de dados do output $(amount, id, B\_)$ são salvos em um arquivo localmente, intitulado `p2pk_outputs_data.json`.

Alice realiza então uma consulta solicitando a cotação para o *amount* desejado e constrói a estrutura de dados para realizar a mintagem do token (contendo o amount, id do keyset designado e o ponto $B\_$). Após receber o retorno da mint, salva o *Proof* recebido em outro arquivo local: `p2pk_proofs_response.json`

Logs obtidos: Exibição de *secret*, Y, r, $B\_$, e confirmação de salvamento

```
secret (P2PK):

["P2PK", {"nonce": "41514113b070f38b6bf299c2748028db...", "data":
    "02c776bf1be8e58e9b900ecb9bd414fe...", "tags": [["sigflag",
    "SIG_INPUTS"]]}]

Ponto Y encontrado com counter: 0

Y: 0225c264b827cb4ddfd401fc728f3243761bf095c5588e722499ce9c97e0be30c6
r_scalar: 266287847413095451096838372763309080810103955O...
r.pubkey: 02931932990d87f2e29a0962e68d1709d512d21254cb27b...
Adição de pontos bem-sucedida
B_: 03b3aec3cb2514b4f901773da7d752fc68b583f12de7d0e7df5f8d30e68dd44366

Outputs e dados salvos em p2pk_outputs_data.json

Quote ID: -oTIiDH_mZ9Q9u5DIyvmRcbdnwkhowxERbcdUDc4
Proofs retornados pela mint salvos em p2pk_proofs_response.json

Resultado:

[{'amount': 1000, 'id': '000b4c3d8b0e7397', 'B_':
    '03da1346b55d553dc49059714180e61fefcff93ead3ee667f3755f97bb2dd98c01'}]
```

Explicação do Log: O log mostra a geração do ponto Y a partir do secret com um contador inicial de 0, o fator de cegamento r, e o cálculo do ponto cego $B\_$. A interação com a mint via requisição POST é concluída com sucesso, e a resposta da mint (proof com $C\_$, DLEQ, id e amount) é salva no arquivo `p2pk_proofs_response.json`, confirmando a mintagem do token de 1000 sats.

A Figura 4.2 abaixo ilustra demonstra o resultado obtido através da solicitação feita à mint para geração do token. Pode-se observar a presença do *id* do *keyset* que contém a chave privada utilizada pela mint na assinatura cega, bem como o *amount*, assinatura cega $C\_$ e a prova de conhecimento zero, *DLEQ*. Alice armazena os dados recebidos para utilizá-los na etapa de descegamento a seguir.

```
{
    "signatures": [
        {
            "id": "000b4c3d8b0e7397",
            "amount": 1000,
            "C_": "0229e33b12fc3a4585ed8cb0df6ee799377d986d03b66132a408f12836c529b97f",
            "dleq": {
                "e": "6664edcb624fdc4d0b25bfbf918419169aa33a6030dc75e949146a4ae786b0c9",
                "s": "0dde7b1f7daae889ff8d5c263fee9791335c26ec0258badf489d65823881e1ad"
            }
        }
    ]
}
```


Figura 4.2 – Resposta obtida da mint na geração do token

4.2 Descegamento e Validação por Alice

Após receber os proofs da mint salvos em `p2pk_proofs_response.json`, Alice utiliza o script `unblind_p2pk_outputs.py` para descegar o ponto $C\_$, validar a prova DLEQ (garantindo que o token de 1000 sats foi corretamente emitido) e gerar a estrutura de dados final do token. O processo inicia com os dados previamente armazenados no arquivo `p2pk_outputs_data.json`, que contêm o secret, fator de cegamento r e os *outputs* que Alice enviou para a mint durante a geração dos tokens na etapa anterior. O fluxo de descegamento e validação é apresentado na Figura 4.3 abaixo.

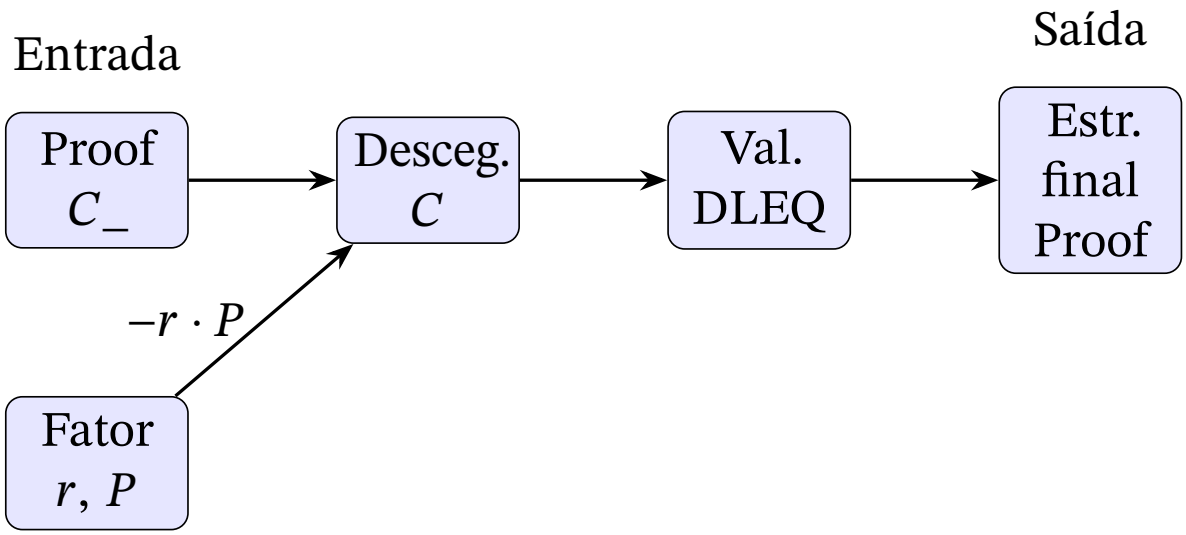

Figura 4.3 – Fluxo de descegamento e validação por Alice.

Trecho de Código: Descegamento de $C\_$ e Validação DLEQ

```python
# Importa dependencias
import hashlib
import json
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Funções para validação DLEQ
def hash_e(P: secp256k1.PublicKey, G: secp256k1.PublicKey, B_: secp256k1.PublicKey, C_: secp256k1.PublicKey, R1: secp256k1.PublicKey, R2: secp256k1.PublicKey) -> bytes:

    """Calcula o hash e = H(P, G, B_, C_, R1, R2) para validação DLEQ."""
    e_input = P.serialize() + G.serialize() + B_.serialize() + C_.serialize() + R1.serialize() + R2.serialize()

    return hashlib.sha256(e_input).digest()

def validate_unblinded_proof(secret: bytes, r: secp256k1.PrivateKey, C: secp256k1.PublicKey, e: bytes, s: bytes, P: secp256k1.PublicKey) -> bool:
    "Valida a prova DLEQ para um proof descegado."

    # Gera Y a partir do secret
    Y = hash_to_curve(secret)

    # Reconstrói C_ e B_ para validação
    C_ = C + P.mult(r)
    B_ = Y + r.pubkey

    # Ponto base G da curva - Padrão Bitcoin
    G_x = bytes.fromhex("79be667ef9dcbbac55...")

    # Ponto gerador G da curva Secp256k1
    G = secp256k1.PublicKey(b"\x02" + G_x, raw=True)

    # Calcula R1 e R2
    R1 = secp256k1.PublicKey(bytes.fromhex(s)) - P.mult(secp256k1.PrivateKey(bytes.fromhex(e)))
    R2 = B_.mult(secp256k1.PrivateKey(bytes.fromhex(s))) - C_.mult(secp256k1.PrivateKey(bytes.fromhex(e)))

    # Computa o hash e_computed com P, G, B_, C_, R1, R2
    e_computed = hash_e(P, G, B_, C_, R1, R2)

    return e == e_computed

# Carrega os proofs da mint salvos localmente
with open("p2pk_proofs_response.json", "r") as f:
    proofs_data = json.load(f)
proofs_list = proofs_data["signatures"]

# Carrega secret e r_scalar salvos anteriormente
```

```
with open("p2pk_outputs_data.json", "r") as f:
    output_data = json.load(f)

secrets = [bytes.fromhex(s) for s in output_data["secrets"]]
r_scalars = output_data["r_scalars"]

# Descegamento dos proofs
for i, proof in enumerate(proofs_list):
    amount = proof["amount"]
    C_ = proof["C_"]
    r_scalar = r_scalars[i]
    secret_bytes = secrets[i]

    # Chave pública fixa para 1000 sats - Mint A (Alice)
    P_hex = "037a1fc79515736aa955efd758e12e10f9..."
    P = secp256k1.PublicKey(bytes.fromhex(P_hex), raw=True)

    C_pubkey = secp256k1.PublicKey(bytes.fromhex(C_), raw=True)
    r = secp256k1.PrivateKey(r_scalar.to_bytes(32, 'big'))
    C = C_pubkey - P.mult(r)  # Descegamento: C = C_' - r * P

    # Validação DLEQ
    dleq = proof.get("dleq")
    if dleq:
        e = bytes.fromhex(dleq["e"])
        s = bytes.fromhex(dleq["s"])
        valid = validate_unblinded_proof(secret_bytes, r, C, e, s, P)
        if valid:
            unblinded_proof = {
                    "amount": amount,
                    "id": proof["id"],
                    "secret": secret_bytes.decode('utf-8'),
                    "C": C.serialize().hex()
                }
            unblinded_proofs.append(unblinded_proof)

# Salvar os proofs descegados
    with open("p2pk_unblinded_proofs.json", "w") as f:
        json.dump(unblinded_proofs, f, indent=4)
```

Explicação do Trecho: Este código inicia pelo carregamento dos proofs salvos em `p2pk_proofs_response.json` e dos dados de cegamento (`secrets`, `r_scalars`) presentes no arquivo `p2pk_outputs_data.json`, gerados anteriormente por Alice. O descegamento é realizado subtraindo do ponto cego $C\_$, o produto do fator de cegamento r_scalar pela chave pública da mint P (fixa para `1000 sats`), resultando em C, a assinatura descegada da mint sobre o token gerado. Em seguida, é feita a validação DLEQ ao invocar a função `validate_unblinded_proof` para verificar a assinatura da mint, comparando o parâmetro e recebido, com o recalculado por Alice a partir de G - ponto gerador da curva elíptica, P -

chave pública da mint para o *amount* de 1000 sats, $B\_$ - ponto Y cego, $C\_$ - assinatura cega da mint, R_1 - nonce calculado por Alice para validação, sendo $R_1 = s \cdot G - e \cdot P$, e R_2 - obtido através da expressão $R_2 = s \cdot B\_ - e \cdot C\_$. Por fim, a estrutura de dados do *Proof* descegado é salva no arquivo auxiliar `p2pk_unblinded_proofs.json`

Logs obtidos: Exibição de $C\_$, r, C, e confirmação de salvamento

```
Descegando proof para 1000 sats:

Chave pública (P): 029d0e1eefde673...
C_: 02076d8061e5bf4...
r_scalar: 889934819221537...
secret (bytes): 5b225032504b222c20...
C após descegamento: 0340bce0fd5f7de7254802ba6868bc79...

Proof descegado salvo em p2pk_unblinded_proofs.json
```

Explicação do Log: O log exibe o descegamento do ponto $C\_$ em C, usando o fator *r_scalar* e a chave pública fixa da mint, confirmando o sucesso do processo. A validação DLEQ verifica a autenticidade do proof com os parâmetros e e s salvos, e o proof descegado é salvo localmente em `p2pk_unblinded_proofs.json`.

Explicação da Estrutura de Dados Final: A estrutura final do token consiste de um objeto JSON contendo um único objeto com os seguintes campos: *amount* (1000 sats), indicando o valor do token; **id** ("000b4c3d8b0e7397"), identificador único do keyset usado pela mint; *secret* (o contrato P2PK original serializado); e "C"("02e71e893e924e6b9d683b68d..."), o ponto descegado que representa a assinatura sobre o token válido. Na Figura 4.4 apresentada a seguir, pode-se verificar a formatação dos dados que compõem, portanto, o token que Alice guarda consigo, que será utilizado na troca com Bob posteriormente.

Figura 4.4 – Estrutura de dados final do proof descegado.

4.3 Mintagem e Descegamento por Bob

Bob realiza as etapas de mintagem e descegamento em sua mint de forma análoga ao processo observado para Alice, utilizando para tanto os scripts `gen_p2pk_outputs.py` e `unblind_p2pk_outputs.py`, mas adaptados para seu contrato P2PK multisig. Diferentemente de Alice, que utiliza um P2PK simples, o contrato de Bob inclui campos adicionais como *n_sigs*, *locktime*, *refund* e *pubkeys*, permitindo condições mais complexas, como múltiplas assinaturas e expiração. Após descegar e validar o token, Bob gera uma prova Schnorr (Zero-Knowledge Proof) para demonstrar que $B\_ = Y + r \cdot G$, salvando-a em `schnorr_proof_bob.json` para que Alice possa recuperá-la posteriormente no processo de *swap*. O fluxo completo é ilustrado na Figura 4.5 abaixo.

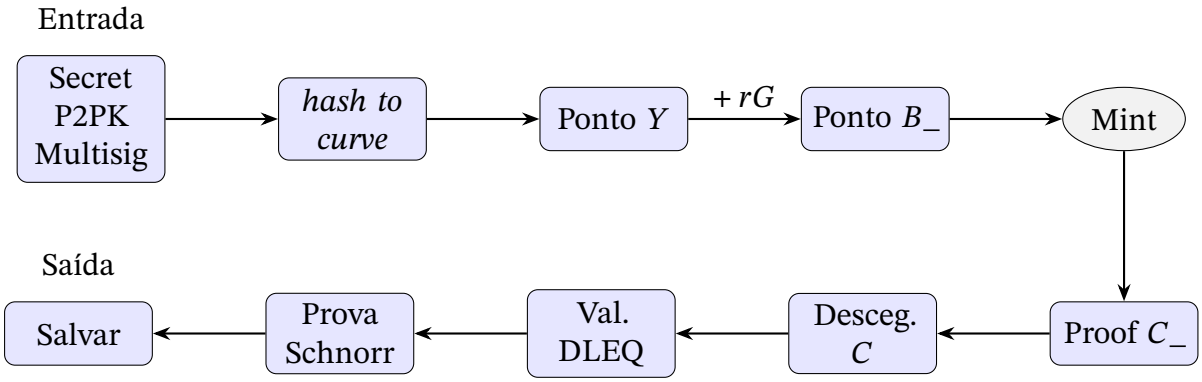

Figura 4.5 – Fluxo de mintagem e descegamento de tokens por Bob.

Trecho de Código: Mintagem, Descegamento e Geração da Prova Schnorr

```
# Importa dependencias
import hashlib
import secrets
import requests
import json
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Função para gerar a prova Schnorr (ZKP)
def generate_schnorr_proof(Y: secp256k1.PublicKey, B_prime:
    secp256k1.PublicKey, r: secp256k1.PrivateKey) -> dict:
    """Gera a prova Schnorr (ZKP) para demonstrar que B' = Y + r * G."""
    # Gera um nonce aleatório k
    k = secp256k1.PrivateKey()
    R = k.pubkey  # R = k * G

    # Calcula o desafio e = H(R || Y || B')
    e_input = R.serialize() + Y.serialize() + B_prime.serialize()
    e = hashlib.sha256(e_input).digest()  # e como bytes
```

```
    e_scalar = int.from_bytes(e, "big") % CURVE_ORDER

    # Converte r.private_key de bytes para inteiro
    r_scalar = int.from_bytes(r.private_key, "big") % CURVE_ORDER

    # Calcula s = (k + e * r) mod n, evitando overflow
    k_scalar = int.from_bytes(k.private_key, "big") % CURVE_ORDER
    temp_product = (e_scalar * r_scalar) % CURVE_ORDER  # Módulo na
        multiplicação
    s = (k_scalar + temp_product) % CURVE_ORDER

    # Garante que s seja um inteiro válido dentro da ordem da curva
    if s < 0:
        s += CURVE_ORDER  # Ajuste para caso de underflow

    # Retorna a prova
    proof = {
        "R": R.serialize().hex(),
        "e": e.hex(),
        "s": hex(s)[2:].zfill(64),  # Converte para hex com 64 caracteres
        "B_prime": B_prime.serialize().hex(),
        "Y": Y.serialize().hex()
    }
    return proof

# Mintagem: Geração do secret P2PK multisig
nonce = secrets.token_bytes(32).hex()
bob_public_key = "02c15e12abf164078fd114c72b679ce..."
p2pk_secret = [
    "P2PK",
    {
        "nonce": nonce,
        "data": bob_public_key,
        "tags": [["sigflag", "SIG_INPUTS"], ["n_sigs", "2"], ["locktime",
            "1751933852"],
             ["refund", "02c15e12abf164078fd114c72b679ce..."],
             ["pubkeys", "02c776bf1be8e58e9b900ecb9bd414fe..."]]
    }
]
secret_str = json.dumps(p2pk_secret)
secret_bytes = secret_str.encode('utf-8')

# Geração de Y e B_
Y = hash_to_curve(secret_bytes)
r = secp256k1.PrivateKey()
B_ = Y + r.pubkey

# Interação com a mint
response = requests.post(f"{MINT_URL}/v1/mint/quote/bolt11", json={"unit":
    "sat", "amount": 1000})

quote_data = response.json()
quote_id = quote_data["quote"]
```

```
mint_response = requests.post(f"{MINT_URL}/v1/mint/bolt11", json={"quote": quote_id, "outputs": [{"amount": 1000, "id": "000b4c3d8b0e7397", "B_": B_.serialize().hex()}]})

proofs = mint_response.json()

# Salvar os dados
output_data = {"outputs": [{"amount": 1000, "id": "000b4c3d8b0e7397", "B_": B_.serialize().hex()}], "secrets": [secret_bytes.hex()], "r_scalars": [int.from_bytes(r.private_key, 'big')]}

with open("p2pk_outputs_data_bob.json", "w") as f:
    json.dump(output_data, f, indent=4)
with open("p2pk_proofs_response_bob.json", "w") as f:
    json.dump(proofs, f, indent=4)

# Descegamento e validação
with open("p2pk_proofs_response_bob.json", "r") as f:
    proofs_data = json.load(f)
proofs_list = proofs_data["signatures"]

with open("p2pk_outputs_data_bob.json", "r") as f:
    output_data = json.load(f)
secrets = [bytes.fromhex(s) for s in output_data["secrets"]]
r_scalars = output_data["r_scalars"]

unblinded_proofs = []
for i, proof in enumerate(proofs_list):
    amount = proof["amount"]
    C_ = proof["C_"]
    r_scalar = r_scalars[i]
    secret_bytes = secrets[i]

    # Chave pública da Mint B (Bob)
    P_hex = "029d0e1eefde673b104cddb01c4bc8d1db77c0efc..."
    P = secp256k1.PublicKey(bytes.fromhex(P_hex), raw=True)
    C_pubkey = secp256k1.PublicKey(bytes.fromhex(C_), raw=True)
    r = secp256k1.PrivateKey(r_scalar.to_bytes(32, 'big'))
    C = C_pubkey - P.mult(r)  # Descegamento

    dleq = proof.get("dleq")
    if dleq:
        e = bytes.fromhex(dleq["e"])
        s = bytes.fromhex(dleq["s"])

        # A mesma função definida para Alice (omitida neste trecho)
        valid = validate_unblinded_proof(secret_bytes, r, C, e, s, P)
        if valid:
            unblinded_proofs.append({
                "amount": amount,
                "id": proof["id"],
                "secret": secret_bytes.decode('utf-8'),
                "C": C.serialize().hex()
```

```
            })

# Salvar os proofs descegados
with open("p2pk_unblinded_proofs_bob.json", "w") as f:
    json.dump(unblinded_proofs, f, indent=4)

# Geração da prova Schnorr
first_proof = output_data["outputs"][0]
first_r = secp256k1.PrivateKey(r_scalars[0].to_bytes(32, 'big'))
secret_bytes = bytes.fromhex(secrets[0])
Y = hash_to_curve(secret_bytes)
B_prime = secp256k1.PublicKey(bytes.fromhex(first_proof["B_"]), raw=True)
schnorr_proof = generate_schnorr_proof(Y, B_prime, first_r)

with open("schnorr_proof_bob.json", "w") as f:
    json.dump(schnorr_proof, f, indent=4)
```

Explicação do Trecho: Bob inicia gerando um contrato P2PK multisig com campos adicionais (*n_sigs* - quantidade de assinaturas a serem apresentadas no gasto do token, antes do tempo de expiração, *locktime* - *unix_timestamp* em segundos, a partir do qual o token pode ser gasto apenas com a apresentação da assinatura correspondente à chave de *refund*, data - *pubKey* de Bob e *pubKeys* - pubKey de Alice), serializado como JSON e convertido em bytes. O ponto Y é calculado via `hash_to_curve`, e $B\_$ é gerado somando $r \cdot G$. Após interação com a mint, os dados são salvos em `p2pk_outputs_data_bob.json` e `p2pk_proofs_response_bob.json`. No descegamento, C é obtido subtraindo $r \cdot P$ de $C\_$, seguido pela validação DLEQ. Por fim, a prova Schnorr é gerada para provar $B\_ = Y + r \cdot G$, essencial para a confiança de Alice no *swap*, e salva em `schnorr_proof_bob.json`. Lembremos que essa prova adicional foi adicionada apenas do lado de Bob, para que Alice tenha garantia sobre o valor $B\_$ ter sido computado a partir do Y que ela espera, caso contrário o token recebido poderia contar com um *nonce* dentro do contrato que fora manipulado, por exemplo. Essa validação vincula criptograficamente o *secret* - contrato - à prova de conhecimento zero disponibilizada. Essa estratégia foi combinada ao compartilhamento da *DLEQ* que a mint de Bob o enviou com Alice, para preservar sua segurança durante a transação.

Logs obtidos:

```
secret (P2PK):

["P2PK", {"nonce": "...", "data": " 02 c15e12abf164078fd114c72b679ce...",
    "tags": [["sigflag", "SIG_INPUTS"], ["n_sigs", "2"], ...]}]

Y: 023c1a32c2a718a4bd563e7820482369f7855af93fdbf6cb5b7a80908d36394450
B_: 027f343dd36f9aa75bc5bf9a9532dce91ee561c7ad5c3a1a08f7906f776bd77fe1
```

```
C_: 02076d8061e5bf4...
C após descegamento: 0340bce0fd5f7de7254802ba6868bc79...
Proof descegado salvo em p2pk_unblinded_proofs_bob.json
Prova Schnorr salva em schnorr_proof_bob.json
```

Explicação do Log: O log mostra a geração de Y e $B\_$, a interação com a mint, o descegamento de $C\_$ em C, e a validação DLEQ. A prova Schnorr é gerada e salva, confirmando a conclusão do processo de Bob para geração do token conforme acordado com Alice inicialmente, e atendendo às condições do protocolo de *swap* atômico proposto.

Prova Schnorr: A Figura 4.6 abaixo apresenta a estrutura de dados da prova Schnorr gerada por Bob, salva em `schnorr_proof_bob.json`, destinada a ser validada por Alice para confirmar a relação $B\_ = Y + r \cdot G$. Essa prova inclui os seguintes campos: R - o nonce público gerado a partir da multiplicação do segredo aleatório k, criado por Bob, pelo ponto gerador da curva através da equação $R = k \cdot G$; e - o desafio calculado como hash $e = sha256(R \| Y \| B\_)$; s - a resposta que combina o segredo aleatório, k, somado ao produto do desafio e pelo nonce secreto r, sendo $s = k + e \cdot r$; $B\_$ - o ponto cego enviado à mint; e Y - o ponto derivado do secret. Estes dados permitem a Alice verificar a integridade do token de Bob antes do processo de *swap*, assegurando que o ponto cego $B\_$ foi corretamente construído a partir de Y e r.


```
{
    "R": "0312dc1430704d3f5b9ecb143cdd430f4d8d3008a1249fea29eac77679377daade",
    "e": "e109c1d58a49a20b79b87a7370b0f90f32a7e8c4d8ad8e7a2bbf785249aa50b2",
    "s": "cbdfe8559baecdfe463bc30ed9b61f25b6b3aafb8b01f51f79864e4c8530eada",
    "B_prime": "027f343dd36f9aa75bc5bf9a9532dce91ee561c7ad5c3a1a08f7906f776bd77fe1",
    "Y": "023c1a32c2a718a4bd563e7820482369f7855af93fdbf6cb5b7a80908d36394450"
}
```


Figura 4.6 – Estrutura da prova Schnorr gerada por Bob.

4.4 Validações DLEQ e Schnorr por Alice

Antes de prosseguir com troca de tokens com Bob, Alice realiza duas validações essenciais:

- 1 - Conferência da prova DLEQ;
- 2 - Verificação da prova Schnorr.

A validação da DLEQ assegura que o token foi assinado pela mint de Bob com a chave pública correta para o amount acordado, enquanto a prova Schnorr confirma que o ponto

cego $B\_$ foi gerado a partir do ponto Y esperado por Alice, impedindo alterações no secret por Bob e garantindo que, ao consultar o *status* do *token* através do endpoint especificado na NUT-07, o retorno da *mint* seja reflexo da situação real daquela unidade. O fluxo dessas validações é ilustrado na Figura 4.7.

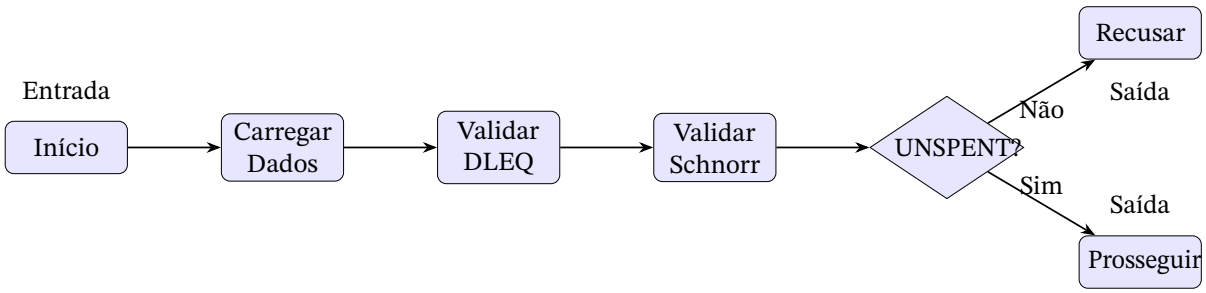

Figura 4.7 – Fluxo de validações por Alice.

4.4.1 Trecho de Código

Abaixo está o código utilizado por Alice para realizar as validações, com comentários explicativos:

```
import json
import hashlib
import secp256k1
from secp256k1_ext import *

# Ordem da curva secp256k1
CURVE_ORDER =
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

# Função para calcular o hash e na validação DLEQ
def hash_e(*publickeys):
    """Calcula o hash e = H(R1, R2, A, C_) para validação DLEQ."""
    e_ = ""
    for p in publickeys:
        _p = p.serialize(compressed=False).hex()
        e_ += str(_p)
    e = hashlib.sha256(e_.encode("utf-8")).digest()
    return e

# Função principal de validação DLEQ
def validate_dleq_alice():

    # Carrega o arquivo de outputs e as proofs de Bob

    with open("p2pk_outputs_data_bob.json", "r") as f:
        output_data = json.load(f)
    with open("p2pk_proofs_response_bob.json", "r") as f:
        proofs_data = json.load(f)
```

```
28
29      B_prime =
            secp256k1.PublicKey(bytes.fromhex(output_data["outputs"][0]["B_"]),
            raw=True)
30
31      C_prime =
            secp256k1.PublicKey(bytes.fromhex(proofs_data["signatures"][0]
32          ["C_"]), raw=True)
33
34      e = bytes.fromhex(proofs_data["signatures"]
35      [0]["dleq"]["e"])
36
37      s = bytes.fromhex(proofs_data["signatures"]
38      [0]["dleq"]["s"])
39
40      # Pubkey da mint para o amount acordado - 1000 sats
41      P = secp256k1.PublicKey(bytes.fromhex("029d0e1eefde67..."), raw=True)
42
43      is_valid = alice_verify_dleq(B_prime, C_prime, e, s, P)
44      return is_valid
45
46  # Validação da prova DLEQ
47  def alice_verify_dleq(B_, C_, e, s, P):
48      """Verifica a equação DLEQ: e == hash_e(R1, R2, P, C_)."""
49
50      # Ponto base G da curva - Padrão Bitcoin
51      G_x = bytes.fromhex("79be667ef9dcbbac55...")
52
53      # Ponto gerador G da curva Secp256k1
54      G = secp256k1.PublicKey(b"\x02" + G_x, raw=True)
55
56      # Calcula R1 e R2
57      R1 = s.pubkey - P.mult(e)
58      R2 = B_.mult(s) - C_.mult(e)
59
60      # Computa o hash e_computed com P, G, B_, C_, R1, R2
61      e_computed = hash_e(P, G, B_, C_, R1, R2)
62      return e == e_computed
63
64  # Validação da prova Schnorr
65  def validate_schnorr_proof():
66      with open("schnorr_proof_bob.json", "r") as f:
67          schnorr_proof = json.load(f)
68
69      R = secp256k1.PublicKey(bytes.fromhex(schnorr_proof["R"]), raw=True)
70      e_scalar = int(schnorr_proof["e"], 16) % CURVE_ORDER
71      e_key = secp256k1.PrivateKey(e_scalar.to_bytes(32, 'big'))
72      s_scalar = int(schnorr_proof["s"], 16)
73      s_key = secp256k1.PrivateKey(s_scalar.to_bytes(32, 'big'))
74
75      B_prime = secp256k1.PublicKey(bytes.fromhex(schnorr_proof["B_prime"]),
            raw=True)
76      Y = secp256k1.PublicKey(bytes.fromhex(schnorr_proof["Y"]), raw=True)
```

```
77      G_x = bytes.fromhex("79be667ef9dcbbac55a06295ce870...")
78      G = secp256k1.PublicKey(b"\x02" + G_x, raw=True)
79
80      left_side = G.mult(s_key)
81      B_minus_Y = B_prime - Y
82      right_side = R + B_minus_Y.mult(e_key)
83
84      e_recomputed_input = R.serialize() + Y.serialize() +
            B_prime.serialize()
85      e_recomputed = hashlib.sha256(e_recomputed_input).digest()
86      e_recomputed_scalar = int.from_bytes(e_recomputed, "big") % CURVE_ORDER
87
88      is_valid = (left_side.serialize() == right_side.serialize() and
            e_scalar == e_recomputed_scalar)
89      return is_valid
90
91  if __name__ == "__main__":
92      dleq_valid = validate_dleq_alice()
93      schnorr_valid = validate_schnorr_proof()
94      print(f"Validação DLEQ: {dleq_valid}")
95      print(f"Validação Schnorr: {schnorr_valid}")
```

4.4.2 Resultados

Os logs gerados ao executar o código indicam o sucesso das validações:

```
1  Validação DLEQ: True
2  Validação Schnorr: True
```

4.4.3 Explicação

Alice inicia carregando os dados fornecidos por Bob, incluindo $B\_$, $C\_$, e os parâmetros da prova DLEQ (e e s). Na validação DLEQ, ela verifica se $e = \text{hash}_e(P\|G\|B\_\|C\_\|R_1\|R_2)$, onde $R1 = s \cdot G - e \cdot P$ e $R2 = s \cdot B\_ - e \cdot C\_$. Isso garante que o token foi assinado pela mint com a chave pública P correspondente ao amount acordado, refutando a possibilidade de Bob fabricar $C\_$, e ou s.

Na validação Schnorr, Alice confirma que $s \cdot G = R + e \cdot (B\_ - Y)$, recalculando e como $\text{hash}(R\|Y\|B\_)$. Essa etapa assegura que $B\_ = Y + r \cdot G$, onde Y representa o secret esperado. Se Bob alterasse o secret, a equação não se manteria, impedindo que Alice fosse enganada. Ambas as validações são fundamentais para a segurança do swap atômico, garantindo que o token seja autêntico e que o contrato permaneça íntegro.

4.5 Geração da Assinatura Schnorr por Bob

Bob gera uma assinatura Schnorr válida para o token de 1000 sats que mintou, utilizando o script `gen_schnorr_sig.py`. Essa assinatura, composta pelo par (R_b, t), serve para provar à mint que ele possui a chave privada correspondente à *pubKey* especificada no contrato do token, sem revelá-la. Já R_b (a coordenada x do nonce público) é compartilhado com Alice para as etapas subsequentes. O resultado é salvo em `schnorr_signature_bob.json`, e o processo completo é ilustrado na Figura 4.8 abaixo.

Figura 4.8 – Fluxo de geração da assinatura Schnorr por Bob.

Trecho de Código: Geração da Assinatura Schnorr

```
# Importa dependencias
import hashlib
import json
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Configurações
BOB_PRIVATE_KEY =
    "ff477dcb0152412435d6813b5ffb64d9af1a60a6c7a5cc04dd3a9c8cf9085b12"
BOB_PUBLIC_KEY =
    "02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447"
ALICE_PUBLIC_KEY =
    "02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5"
CURVE_ORDER =
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAaedce6af48a03bbfd25e8cd0364141

# Verificar consistencia da chave de Bob
kb = secp256k1.PrivateKey(bytes.fromhex(BOB_PRIVATE_KEY))
if kb.pubkey.serialize().hex() != BOB_PUBLIC_KEY:
    raise ValueError("A chave privada de Bob não corresponde a chave
        pública fornecida!")

# Passo 1: Carregar o proof descegado
with open("p2pk_unblinded_proofs_bob.json", "r") as f:
    proofs = json.load(f)

# Pegar o proof de 1000 sats
proof = next(p for p in proofs if p["amount"] == 1000)
x = proof["secret"]  # O secret P2PK (string JSON)

# Verificar se o secret é multisig com Bob e Alice
secret_json = json.loads(x)
```

```
tags = secret_json[1]["tags"]
if not (
    secret_json[1]["data"] == BOB_PUBLIC_KEY and
    ["n_sigs", "2"] in tags and
    ["pubkeys", ALICE_PUBLIC_KEY] in tags
):
    raise ValueError("Secret não é um P2PK multisig com as chaves de Bob e
        Alice! Detalhes: data={}, tags={}".format(
        secret_json[1]["data"], tags))

# Passo 2: Gerar a assinatura Schnorr t usando schnorr_sign
Pb = secp256k1.PublicKey(bytes.fromhex(BOB_PUBLIC_KEY), raw=True)
x_hash = hashlib.sha256(x.encode('utf-8')).digest()  # H(x)
try:
    t = kb.schnorr_sign(x_hash, None, raw=True)
    t_hex = t.hex()

    # Extrair R_b (coordenada x) dos primeiros 32 bytes da assinatura
    Rb_x = t[:32]  # Primeiros 32 bytes
    # Reconstruir ponto com prefixo 02
    Rb = secp256k1.PublicKey(b"\x02" + Rb_x, raw=True)
    print(f"Nonce público (Rb): {Rb_x.hex()}")

    # Validar a assinatura
    is_valid = Pb.schnorr_verify(x_hash, t, None, raw=True)
    print(f"Assinatura válida: {is_valid}")
    if not is_valid:
        raise ValueError("Assinatura Schnorr de Bob inválida.")
except Exception as e:
    print(f"Erro ao gerar ou verificar a assinatura Schnorr: {e}")
    raise

# Passo 3: Salvar a assinatura e Rb
signature = {
    "Rb": Rb_x.hex(),  # Apenas coordenada x
    "t": t_hex
}
with open("schnorr_signature_bob.json", "w") as f:
    json.dump(signature, f, indent=4)
print("Assinatura salva em schnorr_signature_bob.json")
```

Explicação do Trecho: O código inicia verificando a consistência entre a chave privada e pública de Bob, garantindo que os dados criptográficos estejam alinhados. Em seguida, carrega o proof descegado de 1000 sats a partir de `p2pk_unblinded_proofs_bob.json`, extraindo o secret P2PK multisig, que é validado para conter as chaves de Bob e Alice, além das tags multisig (*n_sigs* e *pubkeys*). O secret é hasheado com SHA-256 para gerar $H(x)$, e Bob utiliza sua chave privada para assinar esse hash, produzindo a assinatura Schnorr, que é representada no formato crú como uma string hexadecimal de 64 bytes. A coordenada x do nonce público R_b é extraída dos primeiros 32 bytes da string retornada pela biblioteca,

e então reconstruída como um ponto público; a assinatura é validada para assegurar sua corretude. Por fim, a estrutura contendo R_b e t, observada abaixo na Figura é salva em `schnorr_signature_bob.json`, sendo R_b o *nonce* público que Alice irá utilizar na próxima etapa do protocolo.


```
{
    "Rb": "23aa0933ebf43d50d7e2644b0fcbc99ee75c557f3f4cbad19f09ef1d526d5b0e",
    "t": "(32bytes)...8c4ad6ed0d5244ed926f93fb91132d8812058d985d0c3e272962d51a1e7601a"
}
```


Figura 4.9 – Estrutura da assinatura Schnorr gerada por Bob.

Logs obtidos: Exibição do processo de geração da assinatura Schnorr

```
Secret a ser assinado (x):

["P2PK", {"nonce": "ca1068e138f0fcc823e8769830fb07af48ca49ce...", "data":
    "02c15e12abf164078fd114c72b679ce...", "tags": [["sigflag",
    "SIG_INPUTS"], ["n_sigs", "2"], ["pubkeys",
    "02c776bf1be8e58e9b900ecb9bd414fe..."]]}]

Assinatura (t): 23aa0933ebf43d50d7e2644b0fcbc99ee75c55...(+32bytes)

Nonce público (Rb): 23aa0933ebf43d50d7e2644b0fcbc99ee75c55

Assinatura válida: True
Assinatura salva em schnorr_signature_bob.json
```

Explicação do Log: O log exibe o secret multisig utilizado na assinatura, o resutado t gerado, o nonce público R_b extraído dele, e a validação bem-sucedida da assinatura. A confirmação do salvamento em `schnorr_signature_bob.json` indica que o processo foi concluído, com R_b pronto para ser compartilhado com Alice.

4.6 Geração da *Adaptor Signature* por Alice

Nessa etapa, Alice gera uma assinatura adaptadora como parte do protocolo de swap atômico, utilizando o script `gen_adaptor_sig.py`. Esse processo envolve a criação dos seguintes atributos:

- *Adaptor point*, definido como $T = R_b + H(R_b \| P_b \| hash(x)) \cdot P_b$, em que $H(...)$ é uma função de *hash tagueada* - tag $"BIP0340/challenge"$ - seguindo a especificação da BIP340 no Bitcoin, R_b é o ponto público que Bob extraiu de sua assinatura *Schnorr*

na etapa anterior, P_b sua *pubKey*, x o *secret* do token mintado por ele e $h(x)$ um hash SHA256 padrão sobre este último valor.

- *Adaptor nonce*, computado através da equação $R_{adaptor} = R_a + T$, sendo R_a o *nonce* público obtido por Alice ao multiplicar sua componente secreta ra, um segredo aleatório de 32 bytes, pelo ponto gerador: $Ra = ra \cdot G$.
- *Adaptor signature*, a assinatura adaptadora formada por $sa = ra + H(R_{adaptor} \| Pa \| h(y)) \cdot k_a$, sendo P_a a chave pública de Alice, k_a sua chave privada e $h(y)$ o mesmo hash SHA256 padrão utilizado na construção do ponto T, mas que, alternativamente, recebe como *input* o *secret* do token mintado por Alice. O par que constitui a assinatura mencionada é formado por $R_{adaptor}$, sa, e é salvo em `adaptor_signature_alice.json`. O fluxo detalhado do processo é ilustrado na Figura 4.10 abaixo.

Observação 4.1. A assinatura adaptadora permite que Alice forneça a Bob o par que a compõe - $(R_{adaptor}, s_a)$, vinculando criptograficamente o *nonce* adaptador $R_{adaptor}$, que por sua vez contém o ponto T calculado a partir de R_b, recebido de Bob. Apesar de não compor uma assinatura válida sobre o token de Alice da forma como ela compartilhou, esse mecanismo permite a Bob extrair a componente escalar s_a para computar $s = s_a + t$, sendo esta sim uma assinatura válida de Alice. Entretanto, para gastar o token, Bob deve revelar este último valor apresentado à mint, e Alice o recupera e calcula $t = s - s_a$, a assinatura de Bob sobre seu token. Dessa maneira, e fazendo uso da *Adaptor Signatura*, Alice pôde revelar seu valor a Bob, "amarrando"criptograficamente sua exposição à revelação da parte que faltava à Alice. Essa construção é o que garante a atomicidade do *swap* no protocolo.

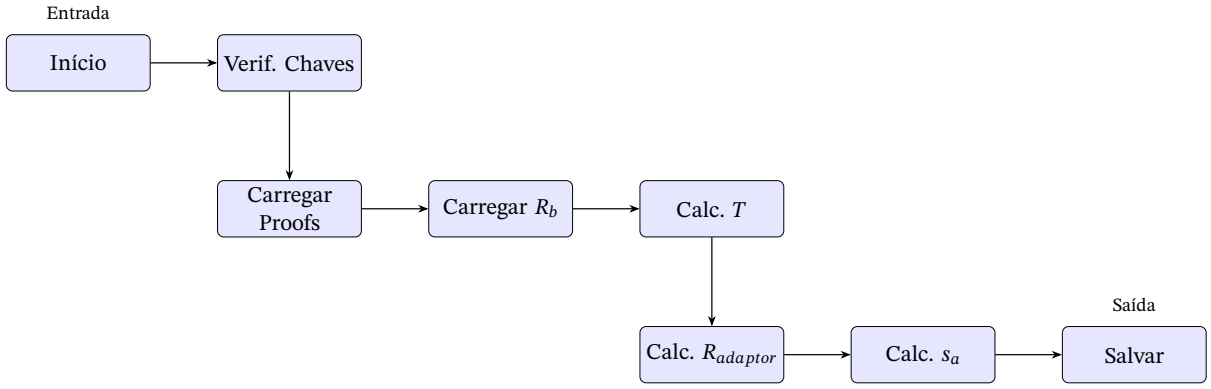

Figura 4.10 – Fluxo de geração da assinatura adaptadora por Alice.

Trecho de Código: Geração da Assinatura Adaptadora

```
# Importa dependencias
import hashlib
import json
import secrets
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Configurações
ALICE_PRIVATE_KEY =
    "ee3375f2c778e0d0e69a8fd27679120cc447d5cf023a4cccf4e5acb0d70e939b"
ALICE_PUBLIC_KEY =
    "02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5"
BOB_PUBLIC_KEY =
    "02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447"
CURVE_ORDER =
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAaedce6af48a03bbfd25e8cd0364141

def tagged_hash(tag: str, data: bytes) -> bytes:
    """Calcula um hash etiquetado conforme BIP-340: SHA256(SHA256(tag) ||
       SHA256(tag) || data)."""
    tag_hash = hashlib.sha256(tag.encode('utf-8')).digest()
    return hashlib.sha256(tag_hash + tag_hash + data).digest()

def has_even_y(point: secp256k1.PublicKey) -> bool:
    """Verifica se a coordenada y do ponto é par, conforme BIP-340."""
    serialized = point.serialize()
    return serialized[0] == 0x02  # 0x02 indica y par, 0x03 indica y ímpar

# Verificar consistencia da chave de Alice
ka = secp256k1.PrivateKey(bytes.fromhex(ALICE_PRIVATE_KEY))
if ka.pubkey.serialize().hex() != ALICE_PUBLIC_KEY:
    raise ValueError("A chave privada de Alice não corresponde a chave
       pública fornecida!")

ka_prime = int.from_bytes(ka.private_key, 'big')

if ka_prime == 0 or ka_prime >= CURVE_ORDER:
    raise ValueError("Chave privada inválida: ka' = 0 ou ka' >= n")

Pa = ka.pubkey  # Deriva P = ka * G
if not has_even_y(Pa):
    ka_prime = (CURVE_ORDER - ka_prime) % CURVE_ORDER
    ka = secp256k1.PrivateKey(ka_prime.to_bytes(32, 'big'))
    Pa = ka.pubkey
    print("Ajustando ka para garantir has_even_y(P)")

# Carregar o proof de Alice (Mint A, 1000 sats)
with open("p2pk_unblinded_proofs.json", "r") as f:
    proofs = json.load(f)
proof = proofs[0]
```

```
y = proof["secret"]  # O secret P2PK de Alice

# Verificar se o secret contém a chave pública correta
secret_json = json.loads(y)
if secret_json[1]["data"] != ALICE_PUBLIC_KEY:
    raise ValueError(f"Chave pública no secret de Alice não corresponde: {secret_json[1]['data']} (esperado: {ALICE_PUBLIC_KEY})")

# Carregar o proof de Bob (Mint B, 1000 sats) para x
with open("p2pk_unblinded_proofs_bob.json", "r") as f:
    proofs_bob = json.load(f)
proof_b = proofs_bob[0]
x = proof_b["secret"]  # O secret P2PK de Bob
print(f"Secret de Bob (x): {x}")

# Carregar Rb de Bob
with open("schnorr_signature_bob.json", "r") as f:
    bob_signature = json.load(f)
Rb_x = bytes.fromhex(bob_signature["Rb"])

# Reconstruir ponto com prefixo 02
Rb = secp256k1.PublicKey(b"\x02" + Rb_x, raw=True)

# Calcular o adaptor point T = Rb + H(Rb || Pb || H(x)) * Pb
Pb = secp256k1.PublicKey(bytes.fromhex(BOB_PUBLIC_KEY), raw=True)

# Calcular H(x)
x_hash = hashlib.sha256(x.encode('utf-8')).digest()

# Calcular H(Rb || Pb || H(x)) com tagged hash BIP0340/challenge
hash_input = Rb_x + Pb.serialize()[1:] + x_hash
h = tagged_hash("BIP0340/challenge", hash_input)
h_int = int.from_bytes(h, 'big') % CURVE_ORDER

hPb = Pb.mult(secp256k1.PrivateKey(h_int.to_bytes(32, 'big')))
T = Rb + hPb
print(f"Adaptor point (T): {T.serialize()[1:].hex()}")

# Calcular o nonce adaptador Radaptor = Ra + T
# Gerar dados aleatórios auxiliares a
a = secrets.token_bytes(32)
t = bytes(a_int ^ b_int for a_int, b_int in zip(ka.private_key, tagged_hash("BIP0340/aux", a)))

# Calcular H(y)
y_hash = hashlib.sha256(y.encode('utf-8')).digest()

# Calcular rand = hash_BIP0340/nonce(t || bytes(Pa) || H(y))
hash_input = t + Pa.serialize()[1:] + y_hash
rand = tagged_hash("BIP0340/nonce", hash_input)
ra_prime = int.from_bytes(rand, 'big') % CURVE_ORDER

if ra_prime == 0:
```

```
96      raise ValueError("Nonce inválido: ra_prime = 0")
97
98  # Calcular Ra = ra_prime * G
99  Ra = secp256k1.PrivateKey(ra_prime.to_bytes(32, 'big')).pubkey
100
101 # Calcular Radaptor = Ra + T
102 Radaptor = Ra + T
103
104 # Ajustar ra para garantir has_even_y(Radaptor)
105 if not has_even_y(Radaptor):  # Verifica se y é par
106     ra_prime = (CURVE_ORDER - ra_prime) % CURVE_ORDER
107     Ra = secp256k1.PrivateKey(ra_prime.to_bytes(32, 'big')).pubkey
108     Radaptor = Ra + T
109
110 print(f"Nonce adaptador (Radaptor): {Radaptor.serialize()[1:].hex()}")
111
112 # Calcular a adaptor signature sa = ra + H(Radaptor || Pa || H(y)) * ka
113
114 # Calcular H(Radaptor || Pa || H(y)) com tagged hash BIP0340/challenge
115 hash_input = Radaptor.serialize()[1:] + Pa.serialize()[1:] + y_hash
116 e = tagged_hash("BIP0340/challenge", hash_input)
117 e_int = int.from_bytes(e, 'big') % CURVE_ORDER
118
119 ka_int = int.from_bytes(ka.private_key, 'big')
120 sa = (ra_prime + e_int * ka_int) % CURVE_ORDER
121 sa_bytes = sa.to_bytes(32, 'big')
122 print(f"Adaptor signature (sa): {sa_bytes.hex()}")
123
124 # Salvar a assinatura
125 signature = {
126     "Radaptor": Radaptor.serialize()[1:].hex(),
127     "sa": sa_bytes.hex()
128 }
129 with open("adaptor_signature_alice.json", "w") as f:
130     json.dump(signature, f, indent=4)
131 print("Assinatura salva em adaptor_signature_alice.json")
```

Explicação do Trecho: O código apresentado inicia verificando a consistência entre as chaves privada e pública de Alice, ajustando ka' para garantir que P_a tenha a coordenada y par, conforme o padrão BIP-340. Alice carrega seu proof descegado de 1000 sats e o de Bob, confirmando que o secret de Bob contém um contrato P2PK multisig válido com sua chave pública e a de Bob. O nonce público R_b fornecido por Bob é carregado e usado para calcular o adaptor point $T = R_b + H(R_b||P_b||H(x)) \cdot P_b$, onde H é um hash etiquetado conforme BIP-340. Em seguida, Alice gera um nonce adaptador $R_{adaptor} = R_a + T$, ajustando R_a para garantir paridade par, e calcula a adaptor signature $s_a = r_a + H(R_{adaptor}||P_a||H(y)) \cdot k_a$. Por fim, a estrutura contendo $R_{adaptor}$ e s_a é salva em `adaptor_signature_alice.json`, permitindo a próxima etapa do swap.

Logs obtidos: Exibição do processo de geração da assinatura adaptadora

```
Secret a ser assinado (y):

["P2PK", {"nonce": "e8f9a2b3c4d5e6f7a8b9c0d1e2f3a4b5...", "data":
    "02c776bf1be8e58e9b900ecb9bd414fe...", "tags": [["sigflag",
    "SIG_INPUTS"]]}]

Secret de Bob (x):

["P2PK", {"nonce": "ca1068e138f0fcc823e8769830fb07af...", "data":
    "02c15e12abf164078fd114c72b679ce...", "tags": [["sigflag",
    "SIG_INPUTS"], ["n_sigs", "2"], ["pubkeys",
    "02c776bf1be8e58e9b900ecb9bd414fe..."]]}]

Rb carregado de Bob:
    23aa0933ebf43d50d7e2644b0fcbc99ee75c557f3f4cbad19f09ef1d526d5b0e
Adaptor point (T):
    03a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a1b2
Nonce adaptador (Radaptor):
    02d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4
Adaptor signature (sa):
    1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3

Assinatura salva em adaptor_signature_alice.json
```

Explicação do Log: O log exibe os secrets de Alice e Bob, a validação do R_b fornecido por Bob, o cálculo do adaptor point T, o nonce adaptador $R_{adaptor}$, e a assinatura adaptadora s_a. A confirmação do salvamento em `adaptor_signature_alice.json` indica o sucesso do processo, preparando a assinatura para o próximo passo do swap.

Explicação da Estrutura da Assinatura Adaptadora: A Figura 4.11 abaixo apresenta a estrutura de dados da assinatura adaptadora gerada por Alice, que será usada por Bob para validar o compromisso de Alice no swap. Essa estrutura inclui $R_{adaptor}$, um ponto sobre a curva elíptica, e s_a, o escalar correspondente. Esses dados permitem que Bob verifique o compromisso de Alice, enquanto a assinatura que Alice precisa permanece oculta até a troca ser concluída por Bob.

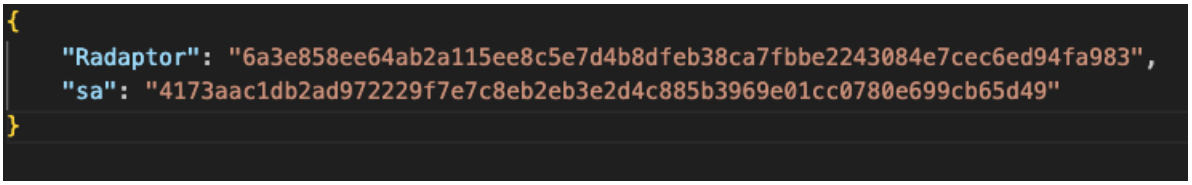

Figura 4.11 – Estrutura da assinatura adaptadora gerada por Alice.

4.7 Cálculo da Assinatura de Alice por Bob

Bob calcula a assinatura válida s de Alice sobre o token de 1000 sats que ela mintou utilizando o script `compute_spend_sig.py`. Esse processo envolve combinar a componente escalar da assinatura adaptadora, s_a, fornecida por Alice, com o escalar da assinatura Schnorr t gerada por Bob, validando-a contra o secret de Alice, e salvando o resultado em `schnorr_signature_alice_by_bob.json`. A assinatura s permite a Bob gastar o token de Alice na Mint A, e o fluxo detalhado do cálculo é ilustrado na Figura 4.12 abaixo.

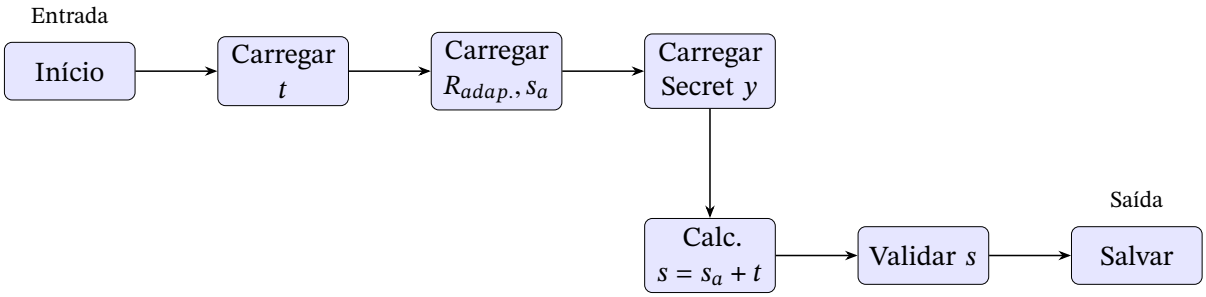

Figura 4.12 – Fluxo de cálculo da assinatura válida por Bob.

Trecho de Código: Cálculo da Assinatura Válida

```
# Importa dependencias
import hashlib
import json
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Configurações
BOB_PRIVATE_KEY =
    "ff477dcb0152412435d6813b5ffb64d9af1a60a6c7a5cc04dd3a9c8cf9085b12"
BOB_PUBLIC_KEY =
    "02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447"
ALICE_PUBLIC_KEY =
    "02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5"
CURVE_ORDER =
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAaedce6af48a03bbfd25e8cd0364141

# Carregar t de Bob
with open("schnorr_signature_bob.json", "r") as f:
    bob_signature = json.load(f)
t = bytes.fromhex(bob_signature["t"])

# Extrair s_b (últimos 32 bytes de t)
s_b = t[32:]  # Ultimos 32 bytes
print(f"s_b (escalar de t): {s_b.hex()}")

# Carregar (Radaptor, sa) de Alice
with open("adaptor_signature_alice.json", "r") as f:
    alice_signature = json.load(f)
```

```
Radaptor = bytes.fromhex(alice_signature["Radaptor"])
sa = bytes.fromhex(alice_signature["sa"])
print(f"Radaptor (Alice): {Radaptor.hex()}")
print(f"sa (Alice): {sa.hex()}")

# Carregar o secret do proof de Alice (Mint A, 1000 sats)
with open("p2pk_unblinded_proofs.json", "r") as f:
    proofs = json.load(f)
proof = proofs[0]
y = proof["secret"]  # O secret P2PK de Alice
print(f"Secret do proof de Alice (y): {y}")

# Calcular s = sa + t
sa_int = int.from_bytes(sa, 'big') % CURVE_ORDER
s_b_int = int.from_bytes(s_b, 'big') % CURVE_ORDER
s = (sa_int + s_b_int) % CURVE_ORDER
s_bytes = s.to_bytes(32, 'big')
print(f"Assinatura válida de Alice (s): {s_bytes.hex()}")

# Validar a assinatura s
Pa = secp256k1.PublicKey(bytes.fromhex(ALICE_PUBLIC_KEY), raw=True)
message = hashlib.sha256(y.encode('utf-8')).digest()
try:
    is_valid = Pa.schnorr_verify(message, Radaptor + s_bytes, None,
        raw=True)
    print(f"Assinatura s válida: {is_valid}")
    if not is_valid:
        raise ValueError("Assinatura Schnorr s inválida.")
except Exception as e:
    print(f"Erro ao verificar a assinatura s: {e}")
    raise

# Salvar a assinatura Schnorr
signature = {
    "Radaptor": Radaptor.hex(),
    "s": s_bytes.hex()
}
with open("schnorr_signature_alice_by_bob.json", "w") as f:
    json.dump(signature, f, indent=4)
print("Assinatura salva em schnorr_signature_alice_by_bob.json")
```

Explicação do Trecho: A implementação apresentada acima inicia carregando o escalar da assinatura Schnorr de Bob, t, salvo no arquivo `schnorr_signature_bob.json`. Em seguida, carrega o par $(R_{adaptor}, s_a)$ fornecido por Alice em `adaptor_signature_alice.json` e o secret y do proof descegado em `p2pk_unblinded_proofs.json`. Bob calcula a assinatura $s = s_a + s_b \mod n$, validando-a contra o secret de Alice por meio da chave pública P_a. A validação Schnorr verifica a equação $s \cdot G = r_{adaptor} \cdot G + H(R_{adaptor}||P_a||H(y)) \cdot k_a \cdot G$ **[Validação Schnorr]**, que é demonstrada passo a passo:

- **Definição da Equação**: A assinatura final s deve satisfazer a equação **[Validação Schnorr]**, destacada acima, para que componha uma prova válida de Alice sobre o token que ela criou e seja aceita pela mint durante o gasto por Bob. Nesse contexto, s é a soma dos compromissos de Alice (s_a) e Bob (t), r_a é o nonce secreto de Alice, e o desafio $H(...)$ é calculado com base no nonce adaptador R_a, a chave pública P_a, e o hash do secret y.

- **Lado Esquerdo** $(s \cdot G)$: O valor s é calculado como $s = s_a + t \mod n$, onde s_a representa o compromisso adaptador de Alice, definido como $s_a = r_a + H(R_{adaptor}||P_a||h(y)) \cdot k_a \mod n$, com r_a sendo o nonce aleatório de Alice, $H(R_{adaptor}||P_a||H(y))$ o desafio calculado com o nonce adaptador, a chave pública $P_a = k_a \cdot G$, k_a a chave privada de Alice e o hash do secret, $h(y)$. O termo t é a assinatura de Bob, dada por $t = r_b + H(R_b||P_b||h(x)) \cdot k_b \mod n$, onde r_b é o nonce privado de Bob, $H(R_b||P_b||h(x))$ o desafio baseado no nonce R_b, a chave pública $P_b = k_b \cdot G$, k_b a chave privada de Bob, e o hash do secret de Bob $h(x)$. Multiplicando s por G, obtemos um ponto na curva elíptica, expresso como $s \cdot G = (s_a + t) \cdot G = [(r_a + H(R_{adaptor}||P_a||h(y)) \cdot k_a] \cdot G + (t \cdot G)$ $= R_a + T + H(R_{adaptor}||P_a||h(y)) \cdot P_a$, que deve corresponder ao lado direito.

- **Lado Direito** $(r_{adaptor} + H(R_{adaptor}||P_a||h(y)) \cdot k_a) \cdot G$: O lado direito começa com $r_{adaptor} \cdot G$, que é o nonce adaptador $R_{adaptor}$ de Alice, definido por $R_{adaptor} = R_a + T$, onde $T = R_b + H(R_b||P_b||H(x)) \cdot P_b$ é o adaptor point calculado por Alice com o nonce R_b de Bob e a chave pública P_b. O segundo termo, $H(R_{adaptor}||P_a||H(y)) \cdot k_a \cdot G$, é o compromisso adaptador de Alice, onde $H(R_{adaptor}||P_a||H(y))$ é o desafio etiquetado conforme BIP-340, calculado com o nonce adaptador $R_{adaptor}$, a chave pública P_a, e o hash do secret $h(y)$, multiplicado pela chave privada k_a e pelo ponto gerador G, resultando em $H(R_{adaptor}||P_a||H(y)) \cdot P_a$. Assim, o lado direito se expande como $r_{adaptor} \cdot G + H(R_{adaptor}||P_a||H(y)) \cdot k_a \cdot G = R_{adaptor} + H(R_{adaptor}||P_a||H(y)) \cdot P_a$, mas como $R_{adaptor} = R_a + T$, então a equação anterior se consolida em $R_a + T + H(R_{adaptor}||P_a||h(y)) \cdot P_a$.

- **Conclusão da Validação**: A função `schnorr_verify`, utilizada no código por Bob (linha 48), valida, internamente, a assinatura s, formada pelo par $(R_{adaptor}, s)$, verificando os dois lados da equação **[Validação Schnorr]** apresentada anteriormente. Esse mecanismo assegura à Bob que, para a mensagem (hash do secret y de Alice) e assinatura $(R_{adaptor}, s)$ fornecidas, a solicitação de gasto será aceita pela *mint* de Alice quando Bob realizar a operação. A estrutura final, contendo s e $R_{adaptor}$, é armazenada em `schnorr_signature_alice_by_bob.json` para uso posterior.

Logs obtidos: Exibição do cálculo da assinatura válida

```
t (Bob):
    23aa0933ebf43d50d7e2644b0fcbc99ee75c557f3f4cbad19f09ef1d526d5b0e(+32
    bytes)
Rb (Bob): 23aa0933ebf43d50d7e2644b0fcbc99ee75c557f3f4cbad19f09ef1d526d5b0e
s_b (escalar de t):
    18c4ad6ed0d5244ed926f93fb91132d8812058d985d0c3e272962d51a1e7601a

Radaptor (Alice):
    02d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4
sa (Alice):
    1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3
Secret do proof de Alice (y):

["P2PK", {"nonce": "e8f9a2b3c4d5e6f7a8b9c0d1e2f3a4b5...", "data":
    "02c776bf1be8e58e9b900ecb9bd414fe...", "tags": [["sigflag",
    "SIG_INPUTS"]]}]

Assinatura válida de Alice (s):
    3c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7
Assinatura s válida: True
Assinatura salva em schnorr_signature_alice_by_bob.json
```

Explicação do Log: O log exibe os componentes t e R_b da assinatura de Bob, o escalar s_b, os dados adaptadores ($R_{adaptor}, s_a$) e o secret de Alice. O cálculo de s e sua validação bem-sucedida são confirmados. A Figura 4.13 abaixo apresenta a estrutura de dados da assinatura válida calculada por Bob, que permite a ele gastar o token de Alice. Essa estrutura inclui $R_{adaptor}$, o ponto público da assinatura, e s, o componente escalar.

Figura 4.13 – Estrutura da assinatura válida calculada por Bob.

4.8 Gasto do Token de Alice por Bob

Já de posse da assinatura de Alice sobre o token P2PK que ela gerou inicialmente, Bob deve agora utilizá-la ($s = sa + t$) para interagir com a Mint A e solicitar o *swap*. A motivação para essa operação reside no fato do token original pertencer à Alice, ou seja, ela pode gastá-lo a qualquer momento apresentando uma assinatura válida, e por esta razão Bob não deve permanecer com ele. Como ambos conhecem uma assinatura que permite

gastar o token, sendo s conhecido por Bob, e $s\_$ uma assinatura alternativa, que Alice pode gerar a qualquer momento por ser proprietária da chave privada k_a, então ambos podem ir até a *Mint* A e realizar um *swap* ou *melt* dessa unidade. Alice não fará essa operação, embora tenha poderes para tal, pois seu objetivo é que Bob realize o gasto, revelando a assinatura que fora utilizada, permitindo que ela compute o segredo t de Bob e gaste o token dele posteriormente, através de $t = s - sa$. Dessa maneira, e para invalidar na mint de Alice a unidade inicial, Bob constrói um novo *output*, no mesmo valor de 1000 sats, e obedecendo as etapas já vistas neste trabalho em seções anteriores, quais sejam: gerar um segredo aleatório de 32 bytes, aplicar a função *hash_to_curve* fornecendo este segredo como entrada, computar $B\_ = Y + r \cdot G$, em que r é um nonce secreto, e transmitir esses dados para a mint (como input) na operação de *swap*, em conjunto ao proof de Alice assinado (como output), na intenção de gerar uma nova unidade proprietária. Assim, Bob obtém um novo token sob seu controle, do qual Alice não tem acesso nem pode gastar. O fluxo é ilustrado na Figura 4.14.

Observação 4.2. A assinatura alternativa $s\_$ mencionada acima refere-se à construção $s\_ = r\_ + H(R\_||P_a||h(y)) \cdot k_a$, ou seja, uma assinatura **válida** que Alice pode gerar a quaquer momento sobre o token que ela mintou. Essa assinatura alternativa não deve ser confundida com $s = sa + t$, que também é uma assinatura válida que permite gastar o token. Esta última, é formada a partir de um compromisso criptográfico que envolve uma contribuição de ambas as partes, ou seja, apesar de existir mais de uma assinatura que satisfaça à condição de validação *Schnorr* para gastar o token travado no contrato P2PK vinculado à *pubKey* de Alice, a assinatura de interesse dela é especificamente s, pois ao conhecê-la, ela pode calcular a assinatura t de Bob sobre o seu token, que é o objeto de interesse em questão, pois eventual assinatura $s\_$ que Alice venha a construir, não possui essa propriedade, e consequentemente não revelará a informação desejada por ela.

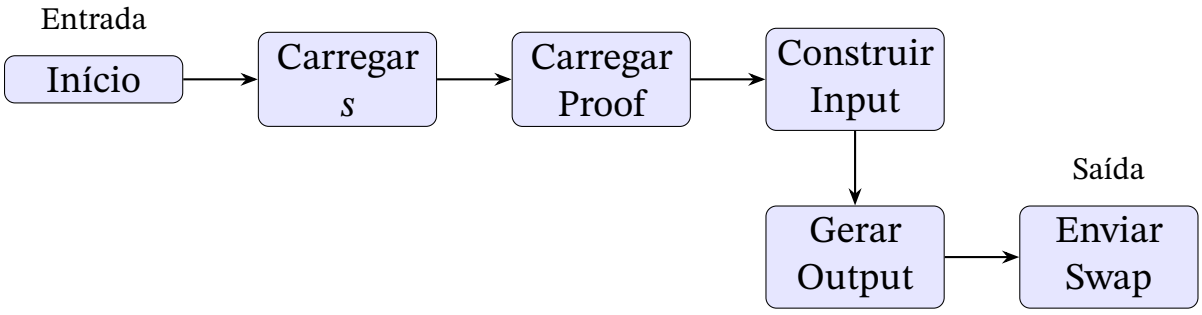

Figura 4.14 – Fluxo de gasto do token de Alice por Bob.

Trecho de Código: Gasto do Token de Alice por Bob

```
# Importa dependencias
import hashlib
import json
import secrets
import requests
import secp256k1
from secp256k1_ext import *  # Importa o monkeypatching

# Configurações
MINT_URL = "http://localhost:3338"  # Mint A
KEYSET_ID = "000b4c3d8b0e7397"  # Keyset da Mint A
ALICE_PUBLIC_KEY =
    "02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5"
DOMAIN_SEPARATOR = b"Secp256k1_HashToCurve_Cashu_"

# Passo 1: Carregar a assinatura de Alice
with open("schnorr_signature_alice_by_bob.json", "r") as f:
    alice_signature = json.load(f)
Radaptor = alice_signature["Radaptor"]
s = alice_signature["s"]
schnorr_signature = Radaptor + s  # Concatenar para 64 bytes
print(f"Assinatura Schnorr de Alice (Radaptor || s): {schnorr_signature}")

# Passo 2: Carregar o proof de Alice (Mint A, 1000 sats)
with open("p2pk_unblinded_proofs.json", "r") as f:
    proofs = json.load(f)
proof = proofs[0]
print(f"Proof de Alice: {proof}")

# Verificar se o secret contém a chave pública correta
secret_json = json.loads(proof["secret"])
if secret_json[1]["data"] != ALICE_PUBLIC_KEY:
    raise ValueError(f"Chave pública no secret de Alice não corresponde")

# Passo 3: Validar a assinatura
Pa = secp256k1.PublicKey(bytes.fromhex(ALICE_PUBLIC_KEY), raw=True)
message = hashlib.sha256(proof["secret"].encode('utf-8')).digest()
try:
    is_valid = Pa.schnorr_verify(message,
       bytes.fromhex(schnorr_signature), None, raw=True)
    if not is_valid:
        raise ValueError("Assinatura Schnorr de Alice inválida.")
except Exception as e:
    print(f"Erro ao verificar a assinatura Schnorr: {e}")
    raise

# Passo 4: Preparar o proof com witness
witness = {
    "signatures": [schnorr_signature]
}
```

```
print(f"Witness gerado: {json.dumps(witness, indent=2)}")
proof_with_witness = {
    "amount": proof["amount"],
    "id": proof["id"],
    "secret": proof["secret"],
    "C": proof["C"],
    "witness": json.dumps(witness)
}

# Passo 5: Gerar novos outputs (2 tokens de 500 sats)
outputs = []
new_amounts = [500,500]
secrets_list = []
r_scalars = []

def hash_to_curve(x: bytes, counter=0) -> secp256k1.PublicKey:
    """Converte um segredo em um ponto Y na curva usando hash_to_curve, conforme Cashu."""
    msg_to_hash = hashlib.sha256(DOMAIN_SEPARATOR + x).digest()
    while counter < 2**16:
        _hash = hashlib.sha256(msg_to_hash + counter.to_bytes(4, "little")).digest()
        try:
            Y = secp256k1.PublicKey(b"\x02" + _hash, raw=True)
            print(f"Ponto Y encontrado com counter: {counter}, prefix: 02")
            return Y
        except Exception:
            counter += 1
    raise ValueError("No valid point found after 2**16 iterations.")

for amount in new_amounts:
    x = secrets.token_bytes(32)
    secrets_list.append(x)
    print(f"Secret para novo output ({amount} sats): {x.hex()}")

    Y = hash_to_curve(x)
    print(f"Y: {Y.serialize().hex()}")

    r = secp256k1.PrivateKey()
    r_scalar = int.from_bytes(r.private_key, 'big')
    r_scalars.append(r_scalar)
    print(f"r_scalar: {r_scalar}")
    print(f"r.pubkey: {r.pubkey.serialize().hex()}")

    B_ = Y + r.pubkey
    print("Adição de pontos bem-sucedida")

    B_serialized = B_.serialize().hex()
    print(f"B_: {B_serialized}")

    outputs.append({
        "amount": amount,
        "id": KEYSET_ID,
```

```
        "B_": B_serialized
    })

# Passo 6: Salvar os dados dos novos outputs
swap_data = {
    "outputs": outputs,
    "secrets": [s.hex() for s in secrets_list],
    "r_scalars": r_scalars
}
with open("swap_outputs_data_bob.json", "w") as f:
    json.dump(swap_data, f, indent=4)
print("Novos outputs, secrets e r_scalars salvos em
   swap_outputs_data_bob.json")

# Passo 7: Enviar o pedido de swap
payload = {
    "inputs": [proof_with_witness],
    "outputs": outputs
}
headers = {"Content-Type": "application/json"}
print(f"Payload enviado: {json.dumps(payload, indent=2)}")
try:
    response = requests.post(f"{MINT_URL}/v1/swap", json=payload,
       headers=headers)
    print(f"Resposta da mint: {response.status_code}")
    print(f"Conteúdo da resposta: {response.text}")

    if response.status_code == 200:
        swap_response = response.json()
        with open("swap_response_bob.json", "w") as f:
            json.dump(swap_response, f, indent=4)
        print("Swap bem-sucedido! Resposta salva em
           swap_response_bob.json")
    else:
        print("Swap falhou. Verifique o proof ou a mint.")
except requests.exceptions.RequestException as e:
    print(f"Erro na requisição de swap: {e}")
```

Explicação do Trecho: Bob inicia carregando a assinatura Schnorr s e $R_{adaptor}$ de Alice em `schnorr_signature_alice_by_bob.json`, concatenando-os para formar a assinatura completa. O proof de Alice de 1000 sats é carregado de `p2pk_unblinded_proofs.json`, e sua assinatura é validada contra o hash do secret y como mensagem, usando a chave pública P_a. O proof é enriquecido com um witness contendo a assinatura Schnorr, formando o input do swap. Bob gera novos outputs de 500 sats cada, criando secrets aleatórios, pontos Y via `hash_to_curve`, e pontos cegos $B\_$ com fatores de cegamento r, salvando-os em `swap_outputs_data_bob.json`. O swap é motivado pela transferência de propriedade: o token de Alice, originalmente dela, é invalidado pela mint com a assinatura s apresentada, gerando novos tokens sob controle exclusivo de Bob, inacessíveis a Alice. O pedido é

enviado à mint via POST, e a resposta é salva em `swap_response_bob.json` se bem-sucedido.

Logs obtidos: Exibição do processo de gasto do token

```
Assinatura Schnorr de Alice (Radaptor || s):
    02d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8b9c0d1...

Proof de Alice:

{'amount': 8, 'id': '000b4c3d8b0e7397', 'secret': '["P2PK", {"nonce":
    "e8f9a2b3c4d5e6f7a8b9c0d1e2f3a4b5...", "data":
    "02c776bf1be8e58e9b900ecb9bd414fe...", "tags": [["sigflag",
    "SIG_INPUTS"]]}]', 'C': '02e71e893e924e6b9d683b68d045a...'}

Assinatura válida: True

Witness gerado: {
  "signatures": ["02d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7..."]
}

Secret para novo output (500 sats):
    a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a1b2
Y: 0225c264b827cb4ddfd401fc728f3243761bf095c5588e722499ce9c97e0be30c6
r_scalar: 266287847413095451096838372763309080810103955O...
B_: 03b3aec3cb2514b4f901773da7d752fc68b583f12de7d0e7df5f8d30e68dd44366

Secret para novo output (500 sats):
    b2c3d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a1b2c3
Y: 0340bce0fd5f7de7254802ba6868bc79a9afe56bc7e83aabb2a0cf9241f0de504d
r_scalar: 915725372251253308777909624511544578989557422926667759765...
B_: 027f343dd36f9aa75bc5bf9a9532dce91ee561c7ad5c3a1a08f7906f776bd77fe1

Novos outputs, secrets e r_scalars salvos em swap_outputs_data_bob.json

Payload enviado: {
  "inputs": [{
    "amount": 1000,
    "id": "000b4c3d8b0e7397",
    "secret": "[\"P2PK\", {\"nonce\":
       \"e8f9a2b3c4d5e6f7a8b9c0d1e2f3a4b5...\", \"data\":
       \"02c776bf1be8e58e9b900ecb9bd414fe...\", \"tags\": [[\"sigflag\",
       \"SIG_INPUTS\"]]}]",
    "C": "02e71e893e924e6b9d683b68d045a5c073d9bc960cb4c683...",
    "witness": "{\"signatures\": [\"02d3e4f5a6b7c8d9e0f1a2b3c4d5...\"]}"
  }],

  "outputs": [{
    "amount": 500,
    "id": "000b4c3d8b0e7397",
    "B_":
       "03b3aec3cb2514b4f901773da7d752fc68b583f12de7d0e7df5f8d30e68dd44366"
```

```
38    }, {
39      "amount":500,
40      "id": "000b4c3d8b0e7397",
41      "B_":
            "027f343dd36f9aa75bc5bf9a9532dce91ee561c7ad5c3a1a08f7906f776bd77fe1"
42    }]
43  }
44  Resposta da mint: 200
45  Conteúdo da resposta: {"status": "success", "new_proofs": [...]}
46  Swap bem-sucedido! Resposta salva em swap_response_bob.json
```

Explicação do Log: O log inicia exibindo a assinatura Schnorr de Alice ($R_{adaptor}||s$), uma string de 64 bytes hexadecimal que combina o nonce adaptador (um ponto sobre a curva elíptica) e o componente escalar s. Seguindo, observamos o proof de Alice, detalhando o token de 1000 sats com "amount", "id", "secret"(contrato P2PK com nonce, data e tags), e "C"(ponto descegado). A validação da assinatura retorna "True", indicando que ($R_{adaptor}||s$) é uma assinatura *Schnorr* válida. O witness, que acompanha o *input*, inclui a assinatura formada por ($R_{adaptor}||s$). Para os novos outputs, dois secrets aleatórios são gerados (um para cada 500 sats), convertidos em pontos Y via `hash_to_curve` e cegados com fatores r para formar $B\_$, sendo salvos em `swap_outputs_data_bob.json`. O payload enviado à mint contém o input com o proof e witness, e os outputs com os novos $B\_$, totalizando 1000 sats. A resposta da mint com status 200 e novos proofs confirma o sucesso do swap, salvo em `swap_response_bob.json`.

4.9 Monitoramento e Cálculo por Alice

Alice utiliza a funcionalidade especificada na NUT-07 do Cashu para monitorar o estado do token através do ponto Y, um processo crucial no protocolo de swap atômico, pois garante que ela tenha transparência e visibilidade sobre o gasto por parte de Bob, podendo extrair a informação desejada quando detectá-lo. O script `check_token_state.py` implementa essa funcionalidade, consultando o estado do token na mint e, quando identificado como "SPENT", captura a assinatura Schnorr usada por Bob, qual seja $s = sa + t$, que inclui o escalar s, necessário para Alice realizar a operação que a permitirá descobrir o valor do segredo t, a assinatura de Bob, calculado utilizando o script `compute_t.py`. Em caso de não colaboração por parte de Bob, ela nunca identifica o *spending*, e recupera seu token quando desejado. Vale relembrar também que, uma vez identificado o gasto por Bob, Alice possui um prazo seguro para extrair s, computar t e gastar o token de Bob - prazo especificado em *locktime*, no contrato P2PK travando o token de Bob; decorrido este tempo, somente ele pode reaver o token, pois sua chave pública é a única especificada no campo *refund*. Essa nuance já fora apresentada no detalhamento técnico do protocolo, e por essa razão, ao estabelecerem

as condições sob as quais a troca irá ocorrer, foi acordado que o gasto por Bob deve acontecer antes de um período X, e caso não aconteça, Alice recupera seu token; isso evita que Bob faça o gasto em um instante tal que Alice não tenha tempo suficiente de: consultar a mint via NUt-07, computar t e gastar na Mint B. A Figura 4.15 abaixo ilustra as etapas referenciadas.

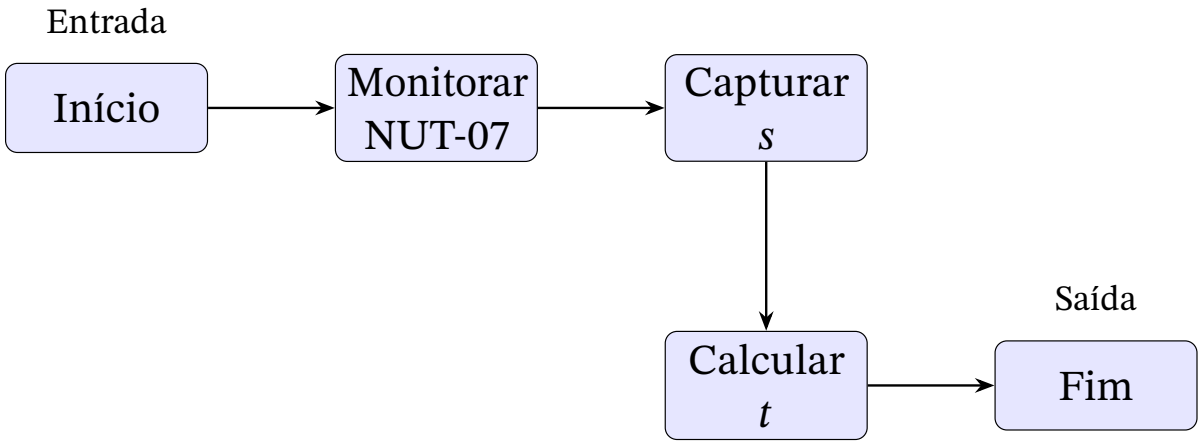

Figura 4.15 – Fluxo de monitoramento e cálculo por Alice.

Trecho de Código: Monitoramento via NUT-07

```
# Configurações
MINT_URL = "http://localhost:3338"
P2PK_OUTPUTS_FILE =
    "/Users/szerwinski/dev/Bitcoin/TCC/nutshell/p2pk_outputs_data.json"
WITNESS_OUTPUT_FILE =
    "/Users/szerwinski/dev/Bitcoin/TCC/nutshell/witness_signature.json"

def load_y_for_amount(amount: int) -> str:
    """Carrega o ponto Y correspondente ao token com o valor
       especificado."""
    with open(P2PK_OUTPUTS_FILE, "r") as f:
        data = json.load(f)
    outputs = data["outputs"]
    Ys = data["Ys"]
    for i, output in enumerate(outputs):
        if output["amount"] == amount:
            return Ys[i]
    raise ValueError(f"Nenhum token encontrado com valor {amount} sats")

def check_token_state(y: str) -> dict:
    """Consulta o estado do token na mint usando NUT-07."""
    endpoint = f"{MINT_URL}/v1/checkstate"
    headers = {"Content-Type": "application/json"}
    payload = {"Ys": [y]}
    try:
        response = requests.post(endpoint, json=payload, headers=headers)
        response.raise_for_status()
```

```
25          return response.json()
26      except requests.exceptions.RequestException as e:
27          print(f"Erro na requisição: {e}")
28          return None
29
30  def save_witness_signature(y: str, signature: str):
31      """Salva a assinatura s (últimos 32 bytes do witness) em um arquivo
            JSON."""
32      if len(signature) != 128:
33          raise ValueError(f"Assinatura inválida: esperado 64 bytes,
                recebido {len(signature)//2}")
34      s_hex = signature[64:]
35      witness_data = {"Y": y, "s": s_hex}
36      with open(WITNESS_OUTPUT_FILE, "w") as f:
37          json.dump(witness_data, f, indent=4)
38      print(f"Assinatura s salva em: {WITNESS_OUTPUT_FILE}")
39
40  def main():
41      try:
42          y = load_y_for_amount(8)
43          print(f"Ponto Y para token de 8 sats: {y}")
44      except ValueError as e:
45          print(f"Erro: {e}")
46          return
47      response = check_token_state(y)
48      if not response:
49          print("Falha ao consultar o estado do token.")
50          return
51      states = response.get("states", [])
52      if not states:
53          print("Nenhum estado retornado pela academia.")
54          return
55      for state in states:
56          y_returned = state.get("Y")
57          token_state = state.get("state")
58          witness = state.get("witness")
59          print(f"\nEstado do token (Y: {y_returned}):")
60          print(f"Estado: {token_state}")
61          if witness:
62              try:
63                  witness_data = json.loads(witness)
64                  signatures = witness_data.get("signatures", [])
65                  if signatures:
66                      print(f"Assinatura Schnorr (witness): {signatures[0]}")
67                      if token_state == "SPENT":
68                          save_witness_signature(y_returned, signatures[0])
69                  else:
70                      print("Nenhuma assinatura encontrada no witness.")
71              except json.JSONDecodeError:
72                  print(f"Witness inválido: {witness}")
73          else:
74              print("Nenhum witness retornado.")
75
```

```
76 if __name__ == "__main__":
77     main()
```

Explicação do Monitoramento via NUT-07: O código `check_token_state.py` implementa o monitoramento do estado de um token usando o protocolo NUT-07. As configurações definem a URL da mint (`MINT_URL`), o arquivo com os outputs (`P2PK_OUTPUTS_FILE`), e o arquivo de saída para a assinatura (`WITNESS_OUTPUT_FILE`). A função `load_y_for_amount` carrega o ponto Y correspondente ao token de 1000 sats a partir de `p2pk_outputs_data.json`, iterando sobre os outputs e retornando o Y associado ao valor especificado, ou levantando um erro se não encontrado. A função `check_token_state` envia uma requisição POST ao endpoint `/v1/checkstate` com o payload contendo Y, retornando o estado do token ("UNSPENT", "SPENT", ou "PENDING") ou `None` em caso de erro. Por fim, na função definida em `save_witness_signature`, são extraídos os últimos 32 bytes da assinatura Schnorr (escalar s) de um witness de 64 bytes, salvando-os em `witness_signature.json` se o token estiver "SPENT". No escopo principal, função `main`, o programa tenta carregar Y, consultar o estado, e processar a resposta, exibindo o estado e a assinatura, salvando s apenas se o token foi gasto, com tratamento de exceções para erros de arquivo ou JSON.

Trecho de Código: Cálculo de t

```
1  # Configurações
2  CURVE_ORDER =
       0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
3  WITNESS_FILE =
       "/Users/szerwinski/dev/Bitcoin/TCC/nutshell/witness_signature.json"
4  ADAPTOR_SIGNATURE_FILE =
       "/Users/szerwinski/dev/Bitcoin/TCC/nutshell/adaptor_signature_alice.json"
5  OUTPUT_FILE =
       "/Users/szerwinski/dev/Bitcoin/TCC/nutshell/bob_signature_t.json"
6
7  def load_witness_signature() -> str:
8      """Carrega o escalar s do witness_signature.json."""
9      with open(WITNESS_FILE, "r") as f:
10         data = json.load(f)
11     s_hex = data.get("s")
12     if not s_hex or len(s_hex) != 64:
13         raise ValueError(f"Escalar s inválido: {s_hex}")
14     return s_hex
15
16 def load_adaptor_signature() -> str:
17     """Carrega o escalar sa de adaptor_signature_alice.json."""
18     with open(ADAPTOR_SIGNATURE_FILE, "r") as f:
19         data = json.load(f)
20     sa_hex = data.get("sa")
21     if not sa_hex or len(sa_hex) != 64:
```

```
22          raise ValueError(f"Escalar sa inválido: {sa_hex}")
23      return sa_hex
24
25  def compute_t(s_hex: str, sa_hex: str) -> bytes:
26      """Calcula t = s - sa mod n."""
27      s_int = int.from_bytes(bytes.fromhex(s_hex), 'big') % CURVE_ORDER
28      sa_int = int.from_bytes(bytes.fromhex(sa_hex), 'big') % CURVE_ORDER
29      t_int = (s_int - sa_int) % CURVE_ORDER
30      t_bytes = t_int.to_bytes(32, 'big')
31      return t_bytes
32
33  def save_t(t_hex: str):
34      """Salva o escalar t em um arquivo JSON."""
35      output_data = {"t": t_hex}
36      with open(OUTPUT_FILE, "w") as f:
37          json.dump(output_data, f, indent=4)
38      print(f"Escalar t salvo em: {OUTPUT_FILE}")
39
40  def main():
41      try:
42          s_hex = load_witness_signature()
43          print(f"Escalar s (do witness): {s_hex}")
44          sa_hex = load_adaptor_signature()
45          print(f"Escalar sa (adaptor): {sa_hex}")
46      except (FileNotFoundError, ValueError) as e:
47          print(f"Erro ao carregar dados: {e}")
48          return
49      try:
50          t_bytes = compute_t(s_hex, sa_hex)
51          t_hex = t_bytes.hex()
52          print(f"Escalar t calculado: {t_hex}")
53      except ValueError as e:
54          print(f"Erro ao calcular t: {e}")
55          return
56      save_t(t_hex)
57
58  if __name__ == "__main__":
59      main()
```

Explicação do Cálculo de t: O código `compute_t.py` calcula o escalar t necessário para o protocolo de swap atômico. As configurações definem a ordem da curva Secp256k1 (`CURVE_ORDER`), os arquivos de entrada (`WITNESS_FILE` para s, `ADAPTOR_SIGNATURE_FILE` para s_a), e o arquivo de saída (`OUTPUT_FILE` para t). A função `load_witness_signature` carrega o escalar s de `witness_signature.json`, validando seu tamanho (32 bytes em hex), enquanto `load_adaptor_signature` faz o mesmo para s_a de `adaptor_signature_alice.json`. A função `compute_t` realiza o cálculo $t = (s - s_a) \mod n$, convertendo os valores hexadecimais em inteiros, subtraindo-os modularmente, e retornando t em bytes. A função `save_t` salva t em `bob_signature_t.json` como JSON. Na função `main`, o programa tenta carregar

s e s_a, calcular t, e salvá-lo, com tratamento de exceções para erros de arquivo ou valores inválidos.

4.10 Gasto do Token de Bob por Alice

Alice utiliza o script `spend_bob_token.py` para gastar o token de 1000 sats de Bob, protegido por um contrato P2PK multisig que exige assinaturas de ambos (Alice e Bob). Alice carrega o proof de Bob, valida o secret multisig, recupera a assinatura t de Bob, gera sua própria assinatura com k_a sobre o secret do token para incluir no witness junto à t e realiza o *swap*. O fluxo é ilustrado na Figura 4.16.

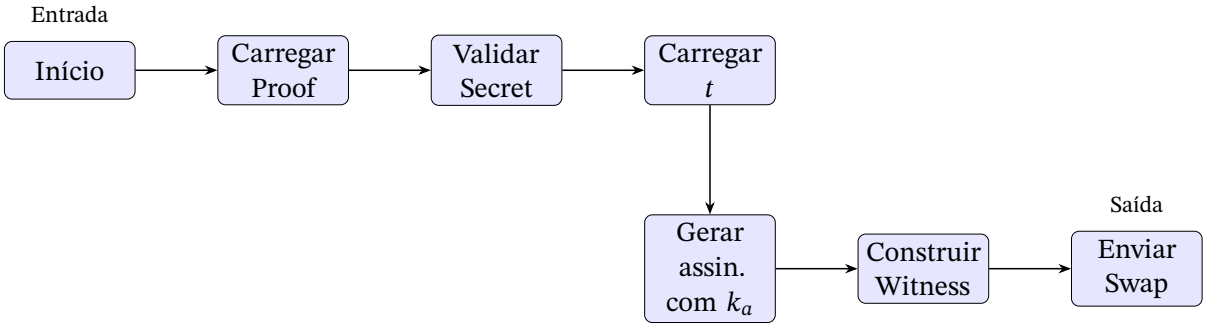

Figura 4.16 – Fluxo de gasto do token de Bob por Alice.

Trecho de Código: Gasto do Token de Bob

```
# Configurações

MINT_B_URL = "http://localhost:3339"  # URL da Mint B
BOB_PROOFS_FILE = "/Users/../nutshell_bob/p2pk_unblinded_proofs_bob.json"
BOB_SIGNATURE_FILE = "/Users/.../nutshell_bob/schnorr_signature_bob.json"
T_FILE = "/Users/.../bob_signature_t.json"
OUTPUT_PROOFS_FILE = "/Users/.../nutshell_bob/new_proofs_alice.json"

ALICE_PRIVATE_KEY =
    "ee3375f2c778e0d0e69a8fd27679120cc447d5cf023a4cccf4e5acb0d70e939b"
ALICE_PUBLIC_KEY =
    "02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5"
BOB_PUBLIC_KEY =
    "02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447"

CURVE_ORDER =
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAaedce6af48a03bbfd25e8cd0364141
DOMAIN_SEPARATOR = b"Secp256k1_HashToCurve_Cashu_"


def tagged_hash(tag: str, data: bytes) -> bytes:
    """Calcula um hash etiquetado conforme BIP-340: SHA256(SHA256(tag) ||
        SHA256(tag) || data)."""
```

```
    tag_hash = hashlib.sha256(tag.encode('utf-8')).digest()
    return hashlib.sha256(tag_hash + tag_hash + data).digest()

def has_even_y(point: secp256k1.PublicKey) -> bool:
    """Verifica se a coordenada y do ponto é par, conforme BIP-340."""
    serialized = point.serialize()
    return serialized[0] == 0x02

def hash_to_curve(x: bytes, counter=0) -> secp256k1.PublicKey:
    """Converte um segredo em um ponto Y na curva usando hash_to_curve,
       conforme Cashu."""
    msg_to_hash = hashlib.sha256(DOMAIN_SEPARATOR + x).digest()
    while counter < 2**16:
        _hash = hashlib.sha256(msg_to_hash + counter.to_bytes(4,
           "little")).digest()
        try:
            Y = secp256k1.PublicKey(b"\x02" + _hash, raw=True)
            print(f"Ponto Y encontrado com counter: {counter}, prefix: 02")
            return Y
        except Exception:
            counter += 1
    raise ValueError("No valid point found after 2**16 iterations.")

def load_bob_proof(amount: int) -> dict:
    """Carrega o Proof de Bob para o valor especificado."""
    with open(BOB_PROOFS_FILE, "r") as f:
        proofs = json.load(f)
    for proof in proofs:
        if proof["amount"] == amount:
            return proof
    raise ValueError(f"Nenhum Proof encontrado para {amount} sats")

def load_bob_signature() -> tuple[str, str]:
    """Carrega Rb e t de schnorr_signature_bob.json."""
    with open(BOB_SIGNATURE_FILE, "r") as f:
        data = json.load(f)
    Rb_hex = data.get("Rb")
    t_hex = data.get("t")
    if not Rb_hex or not t_hex:
        raise ValueError("Rb ou t inválidos em schnorr_signature_bob.json")
    return Rb_hex, t_hex

def load_t() -> str:
    """Carrega o escalar t de bob_signature_t.json."""
    with open(T_FILE, "r") as f:
        data = json.load(f)
    t_hex = data.get("t")
    if not t_hex or len(t_hex) != 64:
        raise ValueError(f"Escalar t inválido: {t_hex}")
    return t_hex

def generate_alice_signature(secret: str, private_key:
   secp256k1.PrivateKey) -> bytes:
```

```
69      """Gera uma assinatura Schnorr de Alice para o secret."""
70      message = hashlib.sha256(secret.encode('utf-8')).digest()
71      signature = private_key.schnorr_sign(message, raw=True, bip340tag=None)
72      return signature
73
74  def create_witness(signatures: list[bytes]) -> str:
75      """Cria o witness com as assinaturas fornecidas."""
76      witness = {"signatures": [sig.hex() for sig in signatures]}
77      return json.dumps(witness)
78
79  def generate_new_output(amount: int, keyset_id: str) -> tuple[dict, bytes]:
80      """Gera um novo output."""
81
82      secret = secrets.token_bytes(32).hex()
83      print(f"Secret para novo output): {secret.hex()}")
84
85      Y = hash_to_curve(secret)
86      print(f"Novo Y: {Y.serialize().hex()}")
87
88      r = secp256k1.PrivateKey()
89      print(f"Novo r.pubkey: {r.pubkey.serialize().hex()}")
90
91      B_ = Y + r.pubkey
92      B_serialized = B_.serialize().hex()
93      print(f"Novo B_: {B_serialized}")
94
95      output = {"amount": amount, "id": keyset_id, "B_": B_serialized}
96      return output, secret_bytes
97
98  def swap_token(proof: dict, input_witness: str) -> dict:
99      """Envia a requisição de swap para a Mint B."""
100     endpoint = f"{MINT_B_URL}/v1/swap"
101     headers = {"Content-Type": "application/json"}
102     new_output, new_secret = generate_new_output(proof["amount"],
            proof["id"])
103     payload = {
104         "inputs": [{"amount": proof["amount"], "id": proof["id"],
                "secret": proof["secret"], "C": proof["C"], "witness":
                input_witness}],
105         "outputs": [new_output]
106     }
107     print(f"Payload enviado: {json.dumps(payload, indent=2)}")
108     try:
109         response = requests.post(endpoint, json=payload, headers=headers)
110         response.raise_for_status()
111         return response.json()
112     except requests.exceptions.RequestException as e:
113         print(f"Erro na requisição de swap: {e}")
114         return None
115
116 def validate_signature(pubkey: str, message: bytes, signature: bytes) ->
        bool:
117     """Valida uma assinatura Schnorr para depuração."""
```

```
118     try:
119         pub = secp256k1.PublicKey(bytes.fromhex(pubkey), raw=True)
120         return pub.schnorr_verify(message, signature, bip340tag=None)
121     except Exception as e:
122         print(f"Erro na validação da assinatura: {e}")
123         return False
124 
125 def main():
126     try:
127         proof = load_bob_proof(1000)
128         print(f"Proof de Bob carregado: {proof['secret']}")
129     except ValueError as e:
130         print(f"Erro: {e}")
131         return
132     secret_json = json.loads(proof["secret"])
133     if not (secret_json[1]["data"] == BOB_PUBLIC_KEY and ["pubkeys",
            ALICE_PUBLIC_KEY] in secret_json[1]["tags"]):
134         print(f"Secret não é multisig com Bob e Alice: {proof['secret']}")
135         return
136     if not ["sigflag", "SIG_INPUTS"] in secret_json[1]["tags"]:
137         print(f"Secret não contém sigflag SIG_INPUTS: {proof['secret']}")
138         return
139     try:
140         Rb_hex, t_hex = load_bob_signature()
141         print(f"Rb de Bob: {Rb_hex}")
142     except ValueError as e:
143         print(f"Erro ao carregar assinatura de Bob: {e}")
144         return
145     try:
146         t_calculated = load_t()
147         print(f"t calculado: {t_calculated}")
148         if t_calculated != t_hex[64:]:
149             print(f"AVISO: t calculado ({t_calculated}) não corresponde a
                    s_b ({t_hex[64:]})")
150     except ValueError as e:
151         print(f"Erro ao carregar t: {e}")
152         return
153     try:
154         ka = secp256k1.PrivateKey(bytes.fromhex(ALICE_PRIVATE_KEY))
155         alice_input_signature = generate_alice_signature(proof["secret"],
                ka)
156         print(f"Assinatura de Alice (input):
                {alice_input_signature.hex()}")
157         message = hashlib.sha256(proof["secret"].encode('utf-8')).digest()
158         is_valid = validate_signature(ALICE_PUBLIC_KEY, message,
                alice_input_signature)
159         print(f"Assinatura de Alice válida: {is_valid}")
160     except Exception as e:
161         print(f"Erro ao gerar assinatura de Alice: {e}")
162         return
163     try:
164         bob_input_signature = bytes.fromhex(Rb_hex + t_calculated)
165         is_valid = validate_signature(BOB_PUBLIC_KEY, message,
```

```
            bob_input_signature)
166         print(f"Assinatura de Bob válida: {is_valid}")
167     except Exception as e:
168         print(f"Erro ao validar assinatura de Bob: {e}")
169         return
170     input_witness = create_witness([alice_input_signature,
            bob_input_signature])
171     print(f"Witness (input): {input_witness}")
172     response = swap_token(proof, input_witness)
173     if response:
174         print("Swap bem-sucedido!")
175         with open(OUTPUT_PROOFS_FILE, "w") as f:
176             json.dump(response, f, indent=4)
177         print(f"Novos proofs salvos em: {OUTPUT_PROOFS_FILE}")
178     else:
179         print("Falha no swap.")
180
181 if __name__ == "__main__":
182     main()
```

Explicação do Trecho: Alice carrega o proof de Bob de 1000 sats, armazenado no arquivo `p2pk_unblinded_proofs_bob.json`, validando o secret multisig que exige as chaves públicas de ambos, além da tag "SIG_INPUTS". Ela obtém t de `bob_signature_t.json` e Rb de `schnorr_signature_bob.json`, comparando t com a parte escalar de Bob (s_b). Alice gera sua assinatura (s_a) com k_a, sua *privateKey* sobre o hash do secret, validando-a contra P_a, sua *pubKey*. A assinatura de Bob ($Rb||t$) é validada contra P_b. Ambas as assinaturas são combinadas no witness para o input do swap. Um novo output de 1000 sats é gerado com um secret aleatório (não protegido por contrato algum), $B\_$ e o *id* do *keyset* desejado, enviado à Mint B no payload. O swap é concluído com a resposta salva em `new_proofs_alice.json` se bem-sucedido.

Logs obtidos: Exibição do processo de gasto do token

```
1 Proof de Bob carregado:
2
3 ["P2PK", {"nonce": "ca1068e138f0fcc823e8769830fb07af48ca49ce...", "data":
      "02c15e12abf164078fd114c72b679ce...", "tags": [["sigflag",
      "SIG_INPUTS"], ["n_sigs", "2"], ["pubkeys",
      "02c776bf1be8e58e9b900ecb9bd414fe..."], ["refund",
      "02c15e12abf164078fd114c72b679ce..."]]}]
4
5 Rb de Bob: 23aa0933ebf43d50d7e2644b0fcbc99ee75c557f3f4cbad19f09ef1d526d5b0e
6
7 t calculado:
      18c4ad6ed0d5244ed926f93fb91132d8812058d985d0c3e272962d51a1e7601a
8
```

```
Assinatura de Alice (input):
    1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d7e8f9a0b1c2d3e4f5a6b7c8d9e0f1a2b3
Assinatura de Alice válida: True
Assinatura de Bob válida: True
Witness (input): {"signatures": ["1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6...",
    "23aa0933ebf43d50d7e2644b0fcbc99e..."]}

Payload enviado: {
  "inputs": [{
    "amount": 1000,
    "id": "000b4c3d8b0e7397",
    "secret": "[\"P2PK\", {\"nonce\":
       \"ca1068e138f0fcc823e8769830fb07af48ca49ce...\", \"data\":
       \"02c15e12abf164078fd114c72b679ce...\", \"tags\": [[\"sigflag\",
       \"SIG_INPUTS\"], [\"n_sigs\", \"2\"], [\"pubkeys\",
       \"02c776bf1be8e58e9b900ecb9bd414fe...\"]]}]",
    "C": "02e71e893e924e6b9d683b68d0...",
    "witness": "{\"signatures\": [\"1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d...\",
       \"23aa0933ebf43d50d7e2644b0fcbc99ee75c...\"]}"
  }],
  "outputs": [{
    "amount": 1000,
    "id": "000b4c3d8b0e7397",
    "B_": "03b3aec3cb2514b4f901773da7d7..."
  }]
}

Swap bem-sucedido!
Novos proofs salvos em:
    /Users/szerwinski/dev/Bitcoin/TCC/nutshell_bob/new_proofs_alice.json
```

Explicação do Log: O log começa exibindo o proof de Bob carregado através do arquivo `p2pk_unblinded_proofs_bob.json`, confirmando o secret multisig com as chaves de Bob e Alice. O nonce Rb e o escalar t de Bob são carregados de `schnorr_signature_bob.json`, com t validado contra o valor calculado. A assinatura de Alice sobre o secret é gerada e validada como "True"contra P_a, enquanto a assinatura de Bob ($Rb||t$) também é validada como "True"contra P_b. O witness combina ambas as assinaturas, pronto para o input. Um novo output de 1000 sats é gerado como um token *anyone can spend* - qualquer um pode gastar se tiver o proof, ou seja, não está protegido por contrato algum. Os dados são agregados e transmitidos, com posterior retorno da Mint B com status de sucesso com o novo proof de Alice salvo em `new_proofs_alice.json`. Com este último passo concluído, a operação de *swap atômico* é finalizada com sucesso, e ambos os participantes detém agora os tokens de interesse (Bob o de Alice e vice-versa), seguindo as condições especificadas na primeira etapa do presente protocolo.

5 Conclusão

5.1 Síntese do Problema Abordado

Este trabalho abordou o desafio de viabilizar trocas atômicas de tokens ecash entre usuários de instâncias distintas do protocolo Cashu, como Alice e Bob, que possuem token A (emitido pela Mint A) e token B (emitido pela Mint B), respectivamente. O problema central foi explorado a partir da falta de interoperabilidade nativa entre mints, devido ao relacionamento instrínseco e indivisível que cada uma delas tem com suas respectivas chaves públicas e privadas, agravado ainda pela inexistência de uma ferramente ou mecanismo orquestrador que possibilitasse o intercâmbio de informações e gastos cruzados.

5.2 Síntese do Protocolo Proposto

A proposta de intervenção aqui apresentada consiste, portanto, de um *Protocolo de Swap Atômico* projetado para viabilizar trocas seguras de tokens entre usuários de *mints* distintas no sistema Cashu. A solução garante esta característica por meio de uma estrutura matemática que vincula criptograficamente as ações de Alice e Bob, eliminando a necessidade de intermediários. A abordagem explora a linearidade das assinaturas e os princípios de validação na curva secp256k1.

O processo começa com Bob gerando uma assinatura *Schnorr* padrão para o *secret* de seu *proof P2PK multisig*. Ele seleciona um nonce secreto r_b, calcula o nonce público $R_b = r_b \cdot G$ (onde G é o ponto gerador da curva secp256k1), e gera a assinatura $t = r_b + H(R_b \| P_b \| h(x)) \cdot k_b \mod n$, onde P_b é a chave pública de Bob, $h(x)$ é o hash do *secret*, H é uma função de hash (como SHA-256), e k_b é sua chave privada. Bob compartilha R_b com Alice.

Em seguida, Alice calcula o ponto adaptador $T = R_b + H(R_b \| P_b \| h(x)) \cdot P_b$ e gera o nonce adaptador $R_{\text{adaptor}} = R_a + T$, onde $R_a = r_a \cdot G$ é seu nonce público. A assinatura adaptadora $s_a = r_a + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot k_a \mod n$ é computada, e o par $(R_{\text{adaptor}}, s_a)$ é enviado a Bob. Nota-se que s_a não é uma assinatura válida por si só para o *proof P2PK* de Alice[3], mas, quando combinada com t, forma a assinatura válida $s = s_a + t$.

A validação da atomicidade é confirmada matematicamente. A equação $s \cdot G = (s_a + t) \cdot G$ deve corresponder a uma assinatura válida para o *proof P2PK* de Alice. Expandindo:

$$s \cdot G = (s_a + t) \cdot G = s_a \cdot G + t \cdot G$$

Substituindo $s_a = r_a + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot k_a$ e $t = r_b + H(R_b \| P_b \| h(x)) \cdot k_b$:

$$s \cdot G = [r_a + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot k_a] \cdot G + [r_b + H(R_b \| P_b \| h(x)) \cdot k_b] \cdot G$$

Como $R_{\text{adaptor}} = R_a + T$ e $T = R_b + H(R_b \| P_b \| h(x)) \cdot P_b$, a validação se reduz a:

$$s \cdot G = R_{\text{adaptor}} + H(R_{\text{adaptor}} \| P_a \| h(y)) \cdot P_a$$

Essa igualdade é verificada pela *mint* de Alice ao gastar o token, permitindo que Bob revele s. Alice, ao observar o gasto via NUT-07, e realizando uma consulta na *mint* através do ponto Y obtido pela função *hash_to_curve*, extrai $t = s - s_a$ do *witness* da transação e usa-o para gastar o token de Bob na *mint* dele, assegurando a atomicidade.

5.2.1 Desafios encontrados

A implementação revelou, durante seu desenvolvimento, alguns gargalos na arquitetura do protocolo que poderiam afetar sua robustez; as mesmas serão apresentadas aqui em caráter elucidativo para: 1 - evidenciar uma escolha de design no Cashu que nos intrigou inicialmente; 2 - contornar uma brecha de segurança que impediria Alice de validar o amount do $Proof$ recebido de Bob; 3 - resolver uma fragilidade que permitira a Bob alterar um parâmetro na estrutura de dados do token fornecido à Alice, impossibilitando seu gasto.

O primeiro ponto identificado, embora não tendo se apresentado como intercorrência para o desenvolvimento do presente trabalho, é curioso do ponto de vista técnico e está presente na NUT-07, que permite consultar o estado de um token a partir de um ponto público Y derivado do *secret* via *hash-to-curve*[4] (NUT-00).

---

[3] A assinatura adaptadora s_a não é válida isoladamente para o *proof P2PK* de Alice, mas, ao ser combinada com t, a assinatura de Bob, forma $s = s_a + t$, que satisfaz as verificações criptográficas do protocolo conforme a BIP-340.

[4] A funcao *hash-to-curve* no Cashu deriva do Blind Diffie-Hellman Key Exchange (BDHKE), uma adaptacao do Diffie-Hellman Key Exchange (DHKE) clássico para curvas elipticas como secp256k1. No DHKE, partes (X e Y) usam um número primo p e outro número gerador g para calcular $g^a \mod p$ e $g^b \mod p$, obtendo uma chave compartilhada. O BDHKE permite que Alice receba uma assinatura cega da mint sem expor o token, usando um domain separator e counter. Para x (ex.: *meu_segredo*), concatena-se com *DOMAIN SEPARATOR*, aplica-se SHA-256, adiciona-se *COUNTER*, aplica-se SHA-256 novamente e gera-se um ponto. Se inválido, incrementa-se o *COUNTER* ate encontrar Y.

Para essa consulta, a *mint* pode retornar os status *SPENT* (token gasto), *PENDING* (relacionado mais intimamente a um pagamento *on-flight* durante o processo de *melt* , NUT-05), ou, por fim, *UNSPENT*.

O estado *UNSPENT* é o que nos chamou atenção, pois a *mint* aplica a função *hash-to-curve* aos *secrets* armazenados de tokens previamente gastos e os compara com o Y fornecido; se não houver correspondência, retorna *UNSPENT*, mesmo que Y seja inválido. Essa falta de validação pode ser uma escolha de eficiência, evitando cálculos adicionais para verificar a conformidade de Y ou, alternativamente, uma forma de se evitar ataques de força bruta ao utilizar a *mint* como um oráculo para atacantes.

Se a *mint* validasse Y e retornasse *UNSPENT* apenas para pontos válidos e não gastos, um atacante poderia:

- Gerar múltiplos $Y's$ a partir de *secrets* aleatórios usando *hash-to-curve*.
- Consultar a *mint* repetidamente para identificar quais Y correspondem a tokens válidos não gastos.

No entanto, a viabilidade desse ataque é limitada:

- A função *hash-to-curve* é unidirecional, com saída em um espaço de 2^{256} pontos na curva secp256k1, tornando a busca exaustiva e computacionalmente inviável.
- O uso de um *domain separator* e *counter* na função *hash-to-curve* adiciona complexidade, dificultando a precomputação de tabelas.
- Em implementações reais, a *mint* estabelece limites de taxa ou medidas de segurança para prevenir abusos, reduzindo a eficácia de consultas em loop.

Portanto, e de acordo com a função *hash-to-curve* especificada na NUT-00, que mapeia um *secret* para um ponto na curva secp256k1, utilizando SHA-256 com um *domain separator* (no caso do Cashu: *Secp256k1_HashToCurve_Cashu_*) e um *counter*, a saída está em um espaço de aproximadamente 2^{256} pontos, tornando a geração de colisões ou a inversão da função computacionalmente inviável (ataque de pré-imagem). Além disso, a *mint* pode implementar limites de taxa ou outras medidas de segurança proprietárias, como bloqueio de consultas excessivas, para mitigar abusos.

Em segundo momento, identificamos uma brecha de segurança que ocorre no *handshake* inicial, quando Alice e Bob trocam as estruturas de dados (mais especificamente os Proofs)

dos tokens que foram mintados, como observado na Figura 5.1 abaixo.


```
{
    "amount": 8,
    "id": "00afe2bda7e0855b",
    "secret": "[\"P2PK\", {\"amount\": 8, \"nonce\":
        \"03e6552e6bcb0b82b706028ce4553654d2911baa49a51783c21486eaa6bdf693\", \"data\":
        \"02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447\",
        \"tags\": [[\"sigflag\", \"SIG_INPUTS\"], [\"n_sigs\", \"2\"], [\"locktime\",
        \"1750610181\"], [\"refund\",
        \"02c15e12abf164078fd114c72b679ce186f0338de7086c559422297a76b432f447\"],
        [\"pubkeys\",
        \"02c776bf1be8e58e9b900ecb9bd414fe48fe99bad2cb76754d39c2c3c7b519a2e5\"]]}]",
    "C": "03063d354dc805157d7b20b153442132815f4c535a819860a2ed3b7e145d7ea36d"
}
```


Figura 5.1 – Estrutura de dados do token mintado por Bob

Considerando a hipótese de Bob, mal intencionado, alterar o *amount* (e.g., de 8 sats para 1 satoshi), Alice não consegue validar a procedência do Proof recebido e seu comprometimento com o valor acordado apenas utilizando a NUT-07, pois Y não carrega um *commitment* criptográfico associado ao valor do token. Durante o *swap*, Bob gasta o token de Alice com s, e consequentemente já realiza a validação desejada, pois para realizar a troca ele deve fornecer como input a proof fornecida por Alice assinada, e como output a estrutura de dados que ele mesmo gerou, no valor esperado. Caso não haja validade no swap, a operação falha e a troca não irá ocorrer.

Por um outro lado, se Bob enviou uma proof à Alice com o *amount* adulterado, Alice só detectaria a operação maliciosa ao tentar gastá-lo, ou seja, não haveria como validar a procedência do proof recebido de Bob antes de se comprometer com a operação. Essa foi uma brecha identificada que se demonstrou como grave potencial de adulterações.

Uma solução teórica seria permitir que, fazendo uso da NUT-07, a *mint* validasse Y perante C (assinatura descegada), verificando $C = k \cdot Y$ com sua chave privada k, que é única para o amount apresentado, mas isso demandaria uma atualização no Cashu, o que, além de exigir a revelação de C para a mint antes do gasto, estaria fora do escopo apresentado para a presente discussão.

Para contornar a situação apresentada, fizemos uso de uma estratégia que necessitou agregar mais dados do que o previsto inicialmente. Mais especificamente, Bob compartilha algumas informações adicionais com Alice ($B\_, C\_$) e a $DLEQ$ recebida de sua *mint*, no intuito de que Alice se convença sobre a procedência do que fora recebido de Bob, em conjunto ao que irá consultar na *mint* para prosseguir com a execução.

O processo envolve o compartilhamento (de Bob para Alice), dos seguintes fatores:

- $B\_$: ponto cego na curva elíptica, calculado como $B\_ = rb \cdot G + Y$, onde rb é um fator de cegamento aleatório e secreto gerado por Bob, e $Y = \text{hash\_to\_curve}(secret)$ é o ponto derivado do *secret* no token por ele mintado.
- $C\_$: assinatura cega fornecida pela *mint*, calculada como $C\_ = k_x \cdot B\_$, onde k_x é a chave privada da *mint* associada ao *amount* específico do token.
- DLEQ: prova de logaritmo discreto, composta pelos valores e e s, que permite verificar a relação entre $C\_$, $B\_$, e a chave pública $P_x = k_x \cdot G$ da *mint* para o *amount* declarado.

Alice, por sua vez, realiza as seguintes etapas de validação do DLEQ. Essa prova garante que a assinatura cega $C\_$, fornecida por Bob, foi gerada corretamente pela *mint* para o *amount* estabelecido entre as duas partes envolvidas na transação.

- Alice obtém P_x, a chave pública da *mint* associada ao *amount* esperado.
- Calcula $R_1 = s \cdot G - e \cdot P_x$.
- Calcula $R_2 = s \cdot B\_ - e \cdot C\_$.
- Verifica se $e = H(G \| P_x \| B\_ \| C\_ \| R_1 \| R_2)$.
- Se a verificação passar, Alice confirma que $C\_ = k_x \cdot B\_$ e $P_x = k_x \cdot G$, vinculando o token ao *amount* correto.

Por último, identificamos uma vulnerabilidade adicional relacionada à integridade dos dados fornecidos por Bob. Se ele mantivesse $B\_$, $C\_$ e a prova DLEQ conforme recebidos da *mint*, mas alterasse o *nonce* no campo *secret* do token, $B\_ = r \cdot G + Y$ deixaria de estar associado ao Y original, pois o novo Y derivado do *secret* modificado não corresponderia ao ponto cego inicial.

Como Alice não conhece o fator de cegamento r usado por Bob, ela não pode validar diretamente essa relação, e caso utilize a NUT-07 para consultar o estado do token, e como a mint não verifica Y diretamente, mas tão somente a existência de um *secret* associado em seu banco de dados, Alice receberia um retorno UNSPENT como falsa evidência de disponibilidade. Nesse cenário, ao tentar gastar o token, o *secret* alterado não corresponderia a uma entidade válida, comprometendo a transação. Bob então esperaria pelo *timeout* estabelecido no contrato para reaver seu capital, prejudicando Alice.

Para mitigar essa ameaça, determinamos que Bob forneça uma prova de conhecimento

zero, semelhante à DLEQ da *mint*, demonstrando que $B\_ = Y + r \cdot G$ sem revelar r. Essa prova garante a conectividade criptográfica entre Y e $B\_$, assegurando a integridade do token sem expor o fator de cegamento.

Por decisão de design, optou-se por não compartilhar o fator de cegamento r com Alice, uma vez que tal prática foi considerada potencialmente insegura, embora os fatores técnicos subjacentes dessa escolha não tenham sido investigados neste trabalho.

Destarte, as dificuldades identificadas foram contornadas com estratégias que reforçam a validação do *amount* e a integridade dos dados intercambiados, garantindo que o protocolo de *swap* atômico seja robusto contra manipulações por Bob. A introdução de provas adicionais, como a DLEQ e a prova de conhecimento zero, proporcionam à Alice segurança plena na execução do *swap*, embora a dependência de dados complementares aumente a complexidade do protocolo. Esses ajustes, aliados à escolha consciente de não expor r, equilibram segurança e eficiência, pavimentando o caminho para melhorias futuras.

5.3 Perspectivas e Contribuições Futuras

Durante o desenvolvimento do nosso protocolo de *swap* atômico, encontramos algumas limitações envolvendo o código-fonte do protocolo Cashu, mais precisamente na especificação NUT-07 e no comprometimento criptográfico de Y com $B\_$. Estes obstáculos estão relacionados à ausência de mecanismos diretos para consultar o (*amount*) de um token ou validar a procedência do ponto Y fornecido. Apesar disso, conseguimos contornar estas dificuldades com soluções colaborativas que envolvem o compartilhamento de dados adicionais entre os participantes, garantindo a segurança e a integridade das transações. A seguir, sugerimos otimizações futuras para o Cashu que poderiam simplificar implementações como a nossa, e embora o *workaround* apresentado tenha se mostrado eficaz para contornar a situação em tela, evidenciamos os pontos adiante elencados como investigações posteriores que se mostram pertinentes.

5.3.1 Consulta do *Amount* via *C*

Uma possível melhoria no protocolo Cashu seria estender a consulta NUT-07 para permitir a validação do *amount* associado a um token. Propomos que o usuário forneça, além do ponto público Y, a assinatura descegada C. A *mint* poderia verificar a relação $C = k \cdot Y$, onde k é a chave privada da *mint* ligada ao *amount*. Essa abordagem confirmaria o valor do token de forma segura, aproveitando elementos já existentes no processo de mintagem, sem expor dados sensíveis. Tal funcionalidade reduziria a dependência de dados extras

em implementações como a nossa. Ressaltamos, entretanto, que não foi realizada uma investigação sobre os potenciais riscos envolvendo a exposição de C de maneira precoce, ficando portanto a disussão envolvendo esta matéria para outro momento.

5.3.2 Validação da Procedência de Y

Outro aspecto observado é a falta de um mecanismo que proporcione a validação direta de procedência do ponto Y. Atualmente, a *mint* retorna (também via NUT-07) o estado do token com base em Y, mas não verifica explicitamente sua legitimidade na curva elíptica ou sua associação com o secret do token. Em nosso protocolo, contornamos esta situação com provas criptográficas adicionais de conhecimento zero, mas uma otimização futura no Cashu poderia incluir a validação explícita de Y. Isso tornaria o processo mais robusto e facilitaria o trabalho em cenários como o nosso.

5.3.3 Contribuições Futuras

As soluções desenvolvidas e sugestões apresentadas visam propor uma nova NUT para o Cashu, integrando o Protocolo de *Swap* Atômico ao seu protocolo base para aprimorar interoperabilidade e privacidade. Essa integração fortalece a capacidade de trocas seguras entre *mints* distintas, alinhando-se aos princípios de confiança mínima do Cashu e promovendo sua evolução como uma solução de pagamento digital robusta e escalável.

Mensagem Final

Esperamos que as perspectivas e contribuições aqui delineadas inspirem novas pesquisas e desenvolvimentos no campo dos sistemas de *ecash* e protocolos de *swap*. Que o esforço seja concentrado no avanço científico e na inovação tecnológica, promovendo soluções mais seguras e eficientes que entreguem cada vez mais poder aos indivíduos e menos ao Estado.

Referências

BOLT, L. **BOLT 11: Payment Encoding - Lightning Network Specification**. [*S.l.*], 2023. Disponível em: https://github.com/lightning/bolts/blob/master/11-payment-encoding.md. Citado na p. 15.

BURT, P. N. **ECDSA Certification - Elliptic Curve Digital Signature Algorithm**. [*S.l.*], 2020. Disponível em: https://www.cs.miami.edu/home/burt/learning/Csc609.142/ecdsa-cert.pdf. Citado na p. 31.

CHAUM, D. **Blind Signatures for Untraceable Payments**. [*S.l.*], 1982. Disponível em: https://chaum.com/wp-content/uploads/2022/01/Chaum-blind-signatures.pdf. Citado nas pp. 14, 21, 23, 24 e 32.

COATES, P. N.; WELSH, P. N. **Research Paper from REU 2017 - University of Chicago**. [*S.l.*], 2017. Disponível em: https://math.uchicago.edu/~may/REU2017/REUPapers/CoatesWelsh.pdf. Citado na p. 29.

COMMUNITY, B. **BIP-199: Hashed Time-Locked Contracts for Atomic Swaps**. [*S.l.*], 2015. Disponível em: https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki. Citado nas pp. 17, 21 e 36.

COMMUNITY, B. **BIP-173: Segregated Witness (Bech32) Address Format**. [*S.l.*], 2017. Disponível em: https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki. Citado na p. 22.

CORE, B. **secp256k1 - Elliptic Curve Cryptography Library**. [*S.l.*], 2023. Disponível em: https://github.com/bitcoin-core/secp256k1. Citado nas pp. 16 e 24.

DIFFIE, W.; HELLMAN, M. **New Directions in Cryptography**. [*S.l.*], 1976. v. 22, n. 6, 644-654 p. Citado nas pp. 21 e 32.

NAKAMOTO, S. **Bitcoin: A Peer-to-Peer Electronic Cash System**. [*S.l.*], 2008. Disponível em: https://bitcoin.org/bitcoin.pdf. Citado nas pp. 14 e 21.

POELSTRA, A. *et al.* **Adaptor Signatures and Atomic Swaps from Scriptless Scripts**. 2018. Disponível em: https://github.com/BlockstreamResearch/scriptless-scripts/blob/master/md/atomic-swap.md. Citado nas pp. 17 e 21.

POON, J.; DRYJA, T. **The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments**. [*S.l.*], 2016. Disponível em: https://lightning.network/lightning-network-paper.pdf. Citado nas pp. 15, 21 e 22.

PROPOSAL, B. I. **Schnorr Signatures for secp256k1**. [*S.l.*], 2020. Disponível em: https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki. Citado nas pp. 17, 21, 29 e 38.

TEAM, C. **Cashu Protocol Specifications**. [*S.l.*], 2023. Disponível em: https://cashubtc.github.io/nuts/. Citado nas pp. 14, 15, 16, 17, 20, 21, 22, 27, 33 e 34.

TEAM, H. **SHA-256 - Secure Hash Algorithm Specification**. [*S.l.*], 2023. Disponível em: https://helix.stormhub.org/papers/SHA-256.pdf. Citado na p. 26.

TEAM, N. **Nostr - Decentralized Social Network Protocol**. [*S.l.*], 2023. Disponível em: https://nostr.com/. Citado na p. 19.

WAGNER, D. **Blind Diffie-Hellman Key Exchange Scheme**. [*S.l.*], 1996. Disponível em: https://cypherpunks.venona.com/date/1996/03/msg01848.html. Citado nas pp. 21, 24 e 26.


UnB