UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Implementação Funcional de *Bulletproofs*

José Carlos Ferrão Pascoal

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa

Covilhã, setembro de 2019

Agradecimentos

Perto de terminar mais uma etapa na minha vida, chega altura de realizar esta dissertação, um último esforço para a conclusão do Mestrado em Engenharia Informática. A todos os que me acompanharam ao longo desta caminhada aqui vos deixo umas palavras de agradecimento.

Primeiramente, agradecer aos meus pais, Maria do Céu Pascoal e Arlindo Pascoal, porque sem a sua ajuda, dedicação e apoio nada disto seria possível. Ao meu irmão Luís Pascoal porque sempre me apoiou e ajudou naquilo que pode. A toda a minha Família, sobretudo aos que estiveram sempre presentes, pois também eles me motivaram de alguma maneira para que este percurso terminasse da melhor forma possível. Um agradecimento especial à minha prima Nanci de Sá, pela paciência demonstrada ao rever este documento e corrigir algum dos seus erros.

Quero agradecer também ao meu orientador, Professor Doutor Simão de Melo Sousa, por me ter dado a oportunidade de trabalhar com ele nesta dissertação e pela ajuda prestada ao longo desta. E a todos os meus colegas do laboratório Release (RELiablE And SEcure Computation), local onde realizei esta dissertação, por me terem oferecido um bom ambiente de trabalho, prestando algum apoio e ajuda quando necessário.

Por último quero agradecer a todos os meu amigos e colegas. Aos que sempre me acompanharam na Covilhã, "Sanaitsabes", dando-me o apoio necessário para chegar até aqui. E aos meus amigos do Sabugal, "La Família", aqueles que nunca me abandonaram, sendo a sua amizade verdadeira. Vocês sempre me apoiaram e fizeram-me esquecer alguns dos problemas que foram surgindo neste percurso, ajudando a ultrapassa-los através dos momentos que partilhámos juntos.

Resumo

É devido á tecnologia de *blockchain* apresentada pela *Bitcoin* que é possível criar moedas digitais geridas pelos seus utilizadores e não por uma entidade central (como bancos ou governos). As criptomoedas oferecem mais uma forma de pagamento, as quais se destacam por não existir a necessidade de confiar em instituições mas sim no sistema, e em mais algumas características, como o pseudo-anonimato.

Para evitar que as moedas sejam gastas mais do que uma vez em várias transações (double spending), as transações têm de ser anunciadas publicamente para que todos as possam verificar e concordar apenas com um único histórico de transações. O sistema tradicional bancário alcança um nível de privacidade ao limitar o acesso às trasações. Um participante apenas tem acesso à suas transações enquanto que o banco tem acesso a todas. Como todas as transações são publicas, a privacidade numa *blockchain* depende apenas da propriedade do pseudo-anonimato. O público apenas consegue observar que um endereço enviou moedas a outro, não revelando informações sobre qualquer entidade. Mas caso sejam transacionadas moedas, a entidade que as vai receber necessita de anunciar um dos seus endereços. Desta forma, o remetente da transacção consegue estabelecer uma ligação entre o endereço anunciado e a entidade, e a partir daí pode descobrir outros endereços da entidade e quantas moedas possuí nos endereços descobertos.

Através de provas de conhecimento nulo é possível alcançar uma maior privacidade nas criptomoedas. É o caso das *Bulletproofs*, as quais permitem realizar de uma forma mais eficiente *confidential transactions*. Numa *confidential transaction*, a quantidade transferida é "escondida" através de *Pedersen Commitments*, os quais mantêm a propriedade algébrica da soma. Assim é possível verificar que a quantidade de moedas como *input* e *output* de uma transação estão equilibradas. É também necessário verificar que um *Pedersen Commitments* de um valor está em um determinado intervalo para prevenir valores negativos ou *overflows*. Para tal é necessário uma *range proof*. As *Bulletproofs* permitem *range proofs* mais eficientes, as quais apenas crescem logaritmicamente com o tamanho do intervalo e o número de *outputs*. Então é preferível ter múltiplos *outputs* numa única *range proof*. Esta característica permite o CoinJoin (múltiplos outputs de várias transações em apenas uma transação), melhorando a privacidade.

Palavras-chave

Blockchain, Criptomoedas, Pseudo-anonimato, Privacidade, Provas de Conhecimento Nulo, *Bulletproofs*, *Confidential Transactions*, *Pedersen Commitments*, *Range Proofs*, *CoinJoin*

Abstract

It's because of the blockchain technology presented by Bitcoin that it's possible to create digital currencies (cryptocurrencies) managed by its users and not by a central entity (such as banks or governments). These type of coins gives another form of payment, which stands out because there is no need to rely on institutions but only in the system, and some other features, such as pseudo-anonymity.

To prevent the currencies from being spent more than once on multiple transactions (double spending), the transactions must be publicly announced, in this way, everyone can verify and agree on only one transaction history. The traditional banking system achieves a level of privacy by limiting access to transactions. A participant only has access to their transactions while the bank has access to all of them. Since all transactions are public, privacy in a blockchain depends only on the property of pseudo-anonymity. The public can only observe that one address sent currencies to another, not revealing information about any entity. But in a transaction, the recipient needs to announce one of their addresses. In this way, the sender of the transaction can establish a connection between the announced address and the entity, and from there can discover other entity addresses and how many currencies it has in the discovered addresses.

Through zero-knowledge proofs, it's possible to achieve greater privacy in cryptocurrencies. The Bulletproofs are zero-knowledge proofs which allow efficient confidential transactions. In a confidential transaction, the amount transferred is "hidden" through Pedersen Commitments, which maintain the algebraic property of the sum. So it's possible to verify that the number of currencies as input and output of a transaction are balanced. But it's also necessary to verify that a Pedersen Commitments of one value is in a certain range to prevent negative values or overflows. For this, it requires a range proof. Bulletproofs enable more efficient range proofs, which only grow logarithmically with the range size and the number of outputs. So it's better to have multiple outputs in a single range proof. This feature allows CoinJoin (multiple outputs of multiple transactions in one transaction), improving privacy.

Keywords

Blockchain, Cryptocurrencies, Pseudo-anonymity, Privacy, Zero-knowledge Proofs, Bulletproofs, Confidential Transactions, Pedersen Commitments, Range Proofs, CoinJoin

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

CPA Chosen-plaintext atack

CPU Central Processing Unit

CRHF Collision-resistant hash functions

DL Discret log

DoS Denial-of-service

ECC Eliptic Curve Criptography

GB Gigabyte

GHz Gigahertz

HDD hard disk drive

kB Kilobyte

KOE Knowledge-of-exponent

MPC Multi-Party Computation

NIZK Non-Interactive Zero-Knowledge Proofs

NP Non-Deterministic Polynomial time

PoS Proof of Stake

PoW Proof of Work

RAM Random Access Memory

TB Terabyte

UBI Universidade da Beira Interior

UTXOs Unspent Transaction Outputs

wNAF w-ary non-adjacent form

Zk-Snarks Zero-knowledge succinct non-interactive argument of knowledge

Zk-Starks Zero-knowledge scalable and transparent argument of knowledge

ZKPK Zero-Knowledge Proof of Knowledge

ZKP Zero-Knowledge Proofs

Capítulo 1

Introdução

Este documento surge no âmbito de um projeto para obtenção do grau de mestre em Engenharia Informática na Universidade da Beira Interior (UBI). Esta Dissertação assenta sobre o tema das *blockchains*, mais propriamente na forma como é possível alcançar privacidade nas transações presentes numa *blockchain*, protegendo desta forma os seus utilizadores ao não revelar qualquer informação sensível a outros que não estão diretamente ligados à transação. A secção que se segue apresenta o contexto e a motivação sobre o tema, enquanto que as secções 1.2 e 1.3 retratam a definição do problema, objetivos e a solução proposta. Por último, a secção 1.4 descreve a estrutura desta dissertação.

1.1 Contexto e Motivação

Nem sempre existiu dinheiro e as pessoas antes da sua existência trocavam os recursos de que necessitavam. Por exemplo, se alguém possuía bastante carne e necessitava de fruta, teria de procurar alguém com bastante fruta que estivesse disposto a trocar as suas frutas por carne. Mas nem sempre estas trocas eram fáceis de se realizar e para trocar um recurso por outro era necessário que os desejos de ambas as partes fossem iguais, e mesmo que se verificasse essa coincidência, acrescia ainda a dificuldade relacionada com as quantidades e os termos de troca desejados. Assim, poderia não ser do interesse da pessoa que trocava fruta, trocar por carne mas sim por cereais. Então o que tinha carne e desejava fruta, tinha de procurar alguém que quisesse trocar carne por cereais e só depois é que poderia obter a fruta.

Com a evolução da sociedade, foi possível arranjar formas de se facilitarem estas trocas e surge então a moeda, que devido às suas características, serve como elemento intermediário e compensador do processo de troca. A partir deste momento as trocas passam a ter mais um componente e a ser realizadas em duas fases: primeiro vende-se para se obter moedas e depois empregam-se estas para a aquisição dos recursos desejados. Mais tarde surgem as notas de forma a facilitar o transporte de grandes quantidades de moedas aos comerciantes. Desta forma uma nota poderia valer 100 vezes mais do que uma determinada moeda. O processo de criação de moedas e notas tornou-se cada vez mais rigoroso para dificultar a contrafação de moedas e notas, de forma a que apenas as entidades competentes as possam emitir, sendo emitidas e controlados pelo governo de um país. Os bancos surgem em consequência do aparecimento da moeda, e o seu aparecimento criou a necessidade de instituições que as guardassem e emprestassem.

Então o dinheiro não é mais do que um meio de pagamento inventado pela humanidade para facilitar a obtenção de bens essenciais. O método mais comum do uso do dinheiro é através de notas ou moedas, ou seja dinheiro físico, o qual pode ser guardado num banco, para mais tarde poder ser usado para realizar pagamentos. Mas com o avanço tecnológico existiu a necessidade de transformar o dinheiro físico em digital, e a forma de pagamento mais utilizada hoje em

dia é o electrónico através do cartão de crédito ou através de aplicativos nos *smartphones* ou aplicações *web*, as quais têm acesso aos nossos cartões ou dados bancários. Sendo assim, nos dias que correm a forma mais avançada de dinheiro são os cartões de débito/crédito, e este continua a ser emitido e controlado por instituições como governos ou bancos.

Se antes de 2008 alguém afirmar-se que seria possível criar uma moeda que não fosse gerida por nenhuma instituição, mas sim por qualquer utilizador da sua comunidade, ninguém acreditaria e acharia absurdo, já que muitos dos seus utilizadores não seriam de confiança e tentariam usar o sistema a seu favor, ao gastar, por exemplo, o mesmo dinheiro mais do que uma vez. Mas nesse mesmo ano foi escrito um *paper* [Nak08] publicado pelo pseudónimo Satoshi Nakamato, no qual era descrito como seria possível alcançar tal proeza, sendo que em 2009 surge a primeira moeda totalmente digital e totalmente gerida pelos seus utilizadores, que foi apelidada de *Bitcoin*. Esta é nos dias de hoje um grande fenómeno de popularidade e uma só moeda pode valer milhares de dólares, dependendo do seu valor de mercado. É com base nos seus princípios que surgem novas moedas e estas passam a ser conhecidas como criptomoedas, devido à sua dependência em métodos criptográficos, os quais assentam sobre conceitos de manutenção de registos. À tecnologia que suporta a criação destas moedas dá-se o nome de *blockchain*.

Uma *blockchain* [DJY18] consiste num *ledger* (livro de contas), onde são registadas as transações dos vários utilizadores da rede. Estas são agrupadas em blocos, que podem ser vistos como páginas do livro de contas. Cada bloco está criptograficamente ligado ao bloco anterior, tornando o sistema resistente a alterações, ou seja, depois de um bloco ser validado e publicado, a sua alteração fica mais difícil à medida que se adicionam novos blocos depois deste. O *ledger* é distribuído pelos vários utilizadores da rede (nodos) através de uma rede *peer-to-peer*. Os nodos comunicam uns com os outros para partilharem alterações no *ledger*, novos blocos, ou transações por validar. Uma *blockchain* não é gerida por nenhuma entidade central, como instituições financeiras ou governos, mas sim gerida por qualquer um de seus utilizadores que queiram participar na sua manutenção. Por isso é que se diz que são descentralizadas. Para manter a rede os nodos verificam as transações que recebem e as que são válidas, agrupam-as num bloco. No final, o nodo envia o seu bloco para a rede e este é aceite segundo a política de consenso utilizada na *blockchain*. Os nodos são incentivados a participar e a manter um bom comportamento, pois caso sejam eles a publicar o novo bloco e este seja aceite pelos outros utilizadores, é lhes atribuída uma recompensa. A política de consenso define como é atribuída esta recompensa e como um conjunto de nodos pode alcançar um consenso em caso de conflito, uma vez que numa rede com vários nodos a tentar publicar blocos ao mesmo tempo implica que haja conflitos e que existam várias versões da *blockchain* em determinados momentos. A versão correta desta é escolhida segundo esta política.

As criptomoedas oferecem mais uma forma de pagamento seguro, as quais se destacam por não existir a necessidade de confiar em instituições mas sim no sistema, e em mais algumas características, como o pseudo-anonimato. Ou seja, olhando para uma transação numa *blockchain* é impossível ligá-la diretamente a uma pessoa, mas se alguma transação for realizada essa ligação pode ser feita. Por este motivo surge um problema: nas transações bancárias apenas o nosso banco as pode ver. O mesmo não se passa numa *blockchain*, já que os utilizadores têm de verificar se a transação é válida, verificando se a pessoa tem direito à utilização das moedas e se estas não foram já gastas numa outra transação passada. Logo todos têm acesso aos valores transacionados e aos endereços dos participantes da transação.

1.2 Definição do Problema e Objetivos

Como descrito no *paper* da *Bitcoin* [Nak08], a solução para verificar transações e impedir o *double-spending* — problema em que as mesmas moedas são gastas mais do que uma vez — sem recorrer a entidades confiáveis (como por exemplo, entidades bancárias) as transações têm de ser anunciadas publicamente para que todos os participantes possam verificar e concordar apenas com um único histórico de transações, com a ordem com que estas foram recebidas.

Deste modo todas transações são públicas, podendo ser vistas por qualquer pessoa que participa na rede. Existem campos da transacção ligados ao anonimato e outros á confidencialidade. A *Bitcoin* é pseudo-anónima e não possui confidencialidade, isto é, não é totalmente anónima uma vez que não é possível estabelecer uma ligação directa entre endereços e uma entidade real, mas é possível estabelecer uma ligação caso sejam transacionadas moedas e a entidade que as vai receber anuncie um dos seus endereços. Sabendo um dos endereços, consegue-se fazer uma ligação direta entre um endereço e a entidade, e a partir deste pode-se descobrir outros endereços utilizados pela entidade, dado que que uma transação poderá possuir *inputs* de diferentes endereços. Assim, percorrendo o histórico de tansações é possível saber quantas moedas uma entidade possui nos endereços descobertos, já que os valores transferidos encontram-se em texto limpo, não existindo qualquer tipo de confidencialidade. A nível de negócios a não confidencialidade nas transações implica alguns problemas, como por exemplo no pagamento de salários de uma empresa aos seus colaboradores (todos os salários são públicos), ou no pagamento a fornecedores. Uma empresa rival poderá saber quanto uma empresa paga aos seus fornecedores, o que é prejudicial para a competitividade entre empresas e para o negócio. A falta de privacidade pode também ser aproveitada para possíveis atacantes definirem alvos mais valiosos, na tentativa que o seu ataque seja bem sucedido e consigam extorquir o maior número de moedas possível. [Bü18]

Esta dissertação tem como objetivo mostrar as várias formas que permitem esconder os endereços e os valores transacionados e quais os métodos que permitem que uma transação possa continuar a ser verificada. Estes métodos só são possíveis graças a um protocolo criptográfico, *Zero-Knowledge Proofs (ZKP)* — provas de conhecimento nulo — nas quais é possível uma parte provar a outra que uma afirmação é verdadeira, sem revelar qualquer informação além da sua veracidade. As *Bulletproofs* são um desses métodos, e com elas é possível alcançar uma forma rápida de verificar uma transação onde os valores transacionados estão "escondidos". Sendo assim, pretende-se realizar uma implementação das *Bulletproofs* e perceber como elas se distinguem dos outros métodos, dando a entender os seus pontos fortes e fracos em relação a estes.

1.3 Abordagem Adotada para a Resolução do Problema

A escolha da abordagem para solucionar o problema mencionado começou por um estudo de como se pode obter privacidade nas transações numa *blockchain*. De seguida, aprofundou-se o estudo das *Bulletproofs* de modo a que fosse possível entender como estas podem fornecer privacidade nas transações, de forma a se prosseguir com a sua implementação. Foi escolhida a linguagem de programação funcional *Ocaml*, pois esta dissertação pretende dar mais uma forma de implementação de privacidade à *blockchain Tezos* (também escrita em *Ocaml*), uma vez que esta já tem planos para implementar privacidade nas transações através de um método chamado *Zero-knowledge succinct non-interactive argument of knowledge (Zk-Snarks)*, o qual vai ser descrito no capítulo 3.

1.4 Organização da Dissertação

De modo a refletir o trabalho realizado, esta dissertação encontra-se estruturada em seis capítulos, os quais são resumidamente descritos a seguir:

- Primeiro capítulo – **Introdução** – introduz o enquadramento e a motivação sobre o trabalho descrito nesta dissertação, bem como a exposição do problema, os objetivos principais e a solução proposta.
- Segundo capítulo – **Visão geral sobre a Tecnologia de *Blockchain*** – aborda o tema das *blockchains*, dando uma revisão geral sobre a tecnologia e seus conceitos.
- Terceiro capítulo – **Privacidade em Transações na *Blockchain*** – o qual se concentra sobre os meios que permitem alcançar privacidade em transações suportadas pela *blockchain*.
- Quarto capítulo – **Implementação de *Range Proofs* com *Bulletproofs*** – apresenta todos os detalhes de como se pode implementar *range proofs* através das *Bulletproofs*, apresentando alguns excertos de código na linguagem de programação *Ocaml*.
- Quinto capítulo – **Resultados e Testes** – apresenta os resultados da implementação e os teste realizados posteriormente, de forma a comprovar que a implementação das *range proofs* em *Ocaml* funcionam como foram desenhadas.
- O sexto Capítulo – **Conclusões e Trabalho Futuro** – apresenta as principais conclusões a retirar ao longo do desenvolvimento desta dissertação, e possíveis propostas de futuras melhorias.
- Apêndice A – **Exemplo de uma *Range Proof*** – contêm um simples exemplo de uma *range proof*.
- Apêndice B – **Algoritmos em *Ocaml***– contêm alguns excertos de códigos pertencentes ao quarto capítulo, os quais eram demasiado longos para constar no capítulo.

Capítulo 2

Visão Geral sobre a Tecnologia de *Blockchain*

2.1 Introdução

A *Bitcoin*, lançada em 2009, foi a primeira de muitas criptomoedas a utilizar a tecnologia de *blockchain* para manter um registo de todas as transações realizadas. Mas algumas das ideias da tecnologia descritas pelo pseudónimo Satoshi Nakamato no *paper* da *Bitcoin* [Nak08], surgiram nos anos 80 e 90. Em 1991 uma cadeia de informação assinada foi usada como *ledger* electrónico para assinar digitalmente, para que nenhum documento assinado fosse mostrado caso sofresse alterações. Em 1998 Leslie Lamport publicou o *paper "The PartTime Parliament"* [Lam98], onde descreve um modelo de consenso para que se chegue a um acordo sobre um resultado numa comunidade de computadores, onde muitos deles podem não ser confiáveis.

Mas para além do uso em criptomoedas, a tecnologia *blockchain* permite que código possa ser publicada nela e executado pelos vários nodos da rede (*smart contracts*). É uma tecnologia que está na "moda" e a tendência é de a implementar em todo o tipo de soluções. Por isso é necessário perceber esta tecnologia pois não é a solução para todos os problemas. Este capítulo tem como objectivo dar uma visão geral sobre a tecnologia, apresentando a sua caracterização, cada um dos seus componentes, quais as suas principais políticas de consenso e como *forks* na *blockchain* podem ser caracterizados.

2.2 Caracterização das *Blockchain*

As *blockchains* podem ser caracterizadas em termos do seu modelo de permissão: *permisionless* ou *permissioned*.

Uma *blockchain permisionless* (sem permissão) está aberta a toda a gente, não sendo necessário pedir autorização para participar na rede. Como qualquer um pode ler e escrever no *ledger*, não existe confiança entre os utilizadores porque muitos deles podem ter um comportamento malicioso. Então para evitar que o sistema seja corrompido, as *blockchains permissionless* utilizam modelos de consenso que requerem que os seus utilizadores façam algum trabalho de manutenção caso queiram publicar blocos, atribuindo uma recompensa a quem mantém um bom comportamento ao publicar blocos válidos.

Por contraste, o acesso a uma *blockchain permissioned* (com permissão) é limitado e requer uma autorização que é dada por uma entidade responsável pela *blockchain*. Então numa *blockchain* deste tipo pode-se restringir a leitura ou não do *ledger*. O mesmo se passa com a escrita ao poder-se restringir as submissões de transações apenas a utilizadores registados, ou permitir a todos. Isto leva a que os utilizadores necessitem de confiar na entidade que a gere, já que ela controla quem pode publicar blocos. Como a autorização requer que um utilizador se identifique, alguém que não tenha um comportamento honesto, pode ver a sua autorização revogada.

Isto implica que haja uma maior confiança entre os utilizadores e que os modelos de consenso requeiram um menor gasto de recursos computacionais. Este tipo de *blockchains* são normalmente utilizadas por organizações que querem manter uma maior controlo e segurança sobre a sua *blockchain*.

2.3 Componentes de uma *Blockchain*

Uma *blockchain* é um conjunto de várias componentes. Logo para a sua compreensão, deve-se examinar cada um dos seus componentes. Elas utilizam mecanismos de computação e primitivas criptográficas combinadas com conceitos de manutenção de registos.

2.3.1 Funções de *Hash* Criptográficas

Uma função de *hash* mapeia um valor de entrada para um único valor de saída. Para além disso, uma função de *hash* criptográfica tem as seguintes propriedades:

- **Resistentes à descoberta do valor de entrada:** Dado um valor *hash* h, a probabilidade de descobrir um valor de entrada m, tal que $H(m) = h$, é negligenciável.
- **Resistentes à descoberta de um segundo valor de entrada:** Dado um valor de entrada m_1, a probabilidade de descobrir um segundo valor de entrada m_2 com o mesmo valor de *hash*, tal que $H(m_1) = H(m_2)$, é negligenciável.
- **Resistentes a colisões:** A probabilidade de encontrar quaisquer dois valores de entrada m_1 e m_2 com o mesmo valor *hash*, tal que $H(m_1) = H(m_2)$, é negligenciável.

Através destas propriedades, as funções de *hash* criptográficas garantem que não houve alterações nos dados de entrada, pois qualquer alteração iria resultar num valor *hash* completamente diferente. Uma das funções mais usada nas *blockchains* é o SHA256 (Secure Hash Algoritm), esta debita 256 bits como valor *hash* para qualquer valor de entrada. Isto quer dizer que há $2^{256} \approx 10^{77} \approx 115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936$ possibilidades de valores *hash*.

2.3.2 *Nonce*

Nonce é um número aleatório, geralmente de grandes dimensões, que é usado apenas uma vez. É o único mecanismo que permite que os mesmos dados possam ser usados para produzir valores de *hash* diferentes. Para isso basta concatenar os dados com o *nonce*, e calcular o valor de *hash*.

$$H(m) \neq H(m||nonce) \quad \textbf{(2.1)}$$

2.3.3 Criptografia de chave pública

Ao contrário da criptografia simétrica, em que apenas se utiliza uma chave para cifrar e decifrar mensagens, a criptografia da chave assimétrica ou criptografia de chave pública, utiliza um par de chaves: uma privada e outra pública. Embora as chaves estejam matematicamente relacionadas, é difícil de computar a chave privada apenas sabendo a chave pública em tempo útil. A chave pública serve para cifrar e a privada para decifrar. Enquanto que em métodos de

assinatura a privada serve para assinar, a pública serve para verificar a assinatura.

Na tecnologia *blockchain* fornece um mecanismo para verificar a integridade e autenticidade de uma transação. O remetente da transação terá de provar que possui a chave privada para a chave publica facultada. Isto garante que ele tem acesso às moedas que pretende gastar. Na figura 2.1 pode-se visualizar como as transações são assinadas. A transação de B para C, possui uma assinatura do conteúdo da transação, a qual incluí o *hash* da transação anterior e possui também a chave pública de B. Qualquer nodo na rede pode verificar esta transação ao averiguar se a chave pública de B corresponde ao endereço de B na transação anterior e se a assinatura é válida ao usar a chave pública fornecida. Um endereço é o valor de *hash* de uma chave pública, sendo estes usados em vez das chaves públicas porque são mais curtos e permitem não vazar qualquer informação sobre a chave pública do remetente da transação. Numa *blockchain permissionless* é permitido aos utilizadores gerar os pares de chaves que desejarem e, portanto, os endereços que desejarem, permitindo uma maior grau de pseudo-anonimato.

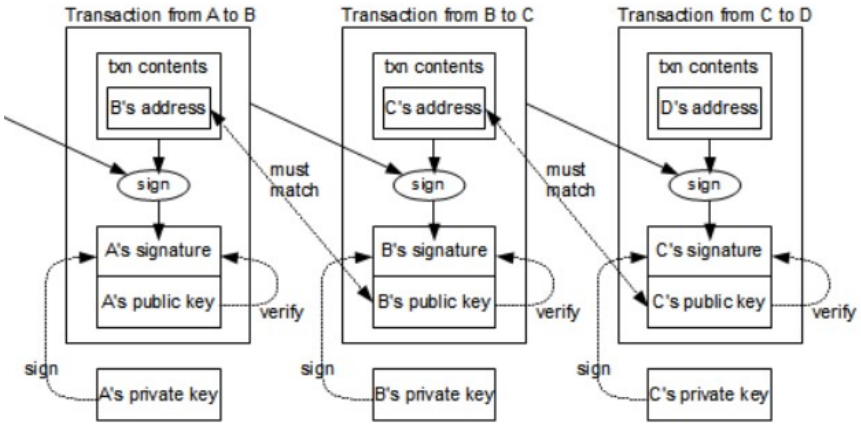

Figura 2.1: Como as transações são assinadas. [Shi14]

As chaves são normalmente guardadas em *software* seguro, designado por *wallet* (carteira). Uma carteira guarda não só as chaves dos utilizadores, como também os seus endereços. Mas para uma maior segurança as chaves privadas devem ser guardadas em *hardware* seguro, uma vez que, se um utilizador perder uma das suas chave privadas fica sem acesso às moedas associadas a esta. Ou pior ainda, caso um atacante consiga uma chave privada de um utilizador, poderá apoderar-se dessas moedas ao realizar uma transação para um dos seus endereços. Uma vez que possui a chave privada, é capaz de produzir uma assinatura válida. Logo a transação será válida para a rede e depois de publicada não poderá ser desfeita.

2.3.4 Transações

Uma transação representa uma interação entre partes, o que nas criptomoedas representa a transferência de moedas entre utilizadores da rede. Então uma transação é uma maneira de registar atividades. As transações podem não ser só usadas para transferir moedas, como podem ser usadas também para publicar dados na *blockchain*. Nos *smart contracts*, transações podem ser usadas para enviar dados, processar dados e guardar resultados na *blockchain*.

Geralmente os dados presentes numa transação são o endereço do receptor ou outro identificador, a chave pública do remetente, a assinatura digital da transação, a quantidade de moedas a

transferir como *inputs* e a quantidade de moedas que cada endereço irá receber como *outputs*. Cada *input* é uma referência a um evento passado onde o remetente da transação recebeu as moedas como *output* numa transação anterior. Cada *output* é composto pela quantidade de moedas juntamente com o endereço que as irá receber. Isto significa que um *input* pode ser dividido em mais do que um *output*. Na figura 2.2 pode-se visualizar como é realizada uma transação em criptomoedas. Na transação C, são gasto 0.008 moedas, recebidas nas transações A e B, e o receptor com o endereço presente no campo out_1 recebe 0.003 moedas, enquanto que o endereço presente no out_2 recebe 0.004 moedas. Normalmente é associado uma taxa na transação (*fee*) para distribuir por quem publicar o bloco onde constam as transações; neste exemplo é de 0.001 moedas. Para uma transação ser válida, a soma dos *inputs* deve ser igual á soma dos *outputs* mais a taxa da transação. Se o valor das moedas como *input* é maior do que o requerido para a transferência o troco é devolvido a um dos endereços do remetente. Para impedir que se gastem as mesmas moedas mais do que uma vez (*double spending*) as transações são publicadas na *blockchain* e qualquer nodo as pode verificar. *Unspent Transaction Outputs (UTXOs)* são todos os *outputs* recebidos em transações que podem ser gastos numa nova transação como *inputs*. No exemplo os UTXOs são out_1 da transação A e os dois *outputs* da transação C.

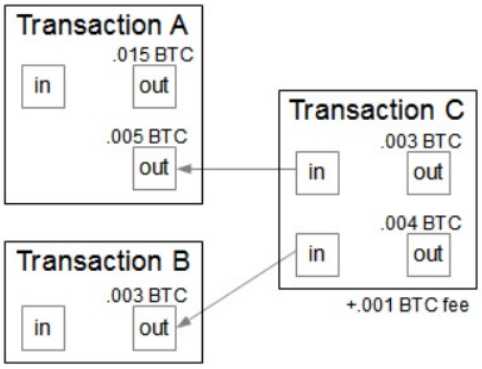

Figura 2.2: Exemplo de uma transações em criptomoedas. [Shi14]

2.3.5 Blocos

Uma *blockchain* pode ser vista por um conjunto de blocos, e um bloco como um conjunto de transações. Os utilizadores enviam as suas transações através de um *software*, e estas são propagadas pela rede mas não são logo publicadas na *blockchain*. Normalmente um nodo espera até que tenha um conjunto de transações antes de as poder publicar num bloco. Cada bloco contem um cabeçalho e uma lista de transacções válidas. O cabeçalho contém o *hash* do bloco anterior, o *hash* da representação do bloco (normalmente gera-se uma *merkle tree* com todas as transações do bloco e guarda-se apenas a sua *root* como *hash*), o *nonce*[1] que ajudou a resolver o desafio do modelo de consenso *Proof of Work (PoW)* e um *timestamp* (data e hora). Como se pode visualizar na figura 2.3 os blocos estão ligados uns aos outros, e uma mudança num bloco faz com que o seu *hash* mude, assim como os blocos a seguir a ele.

Devido à possibilidade de blocos serem reescritos, uma transação só é confirmada quando vários blocos tenham sido adicionados depois do bloco que contém a transação. Então, com a adição de novos blocos, diminui a probabilidade de um dos blocos anteriores voltar a ser reescrito.

[1]Poderá estar presente ou não, ou usado para outro propósito.

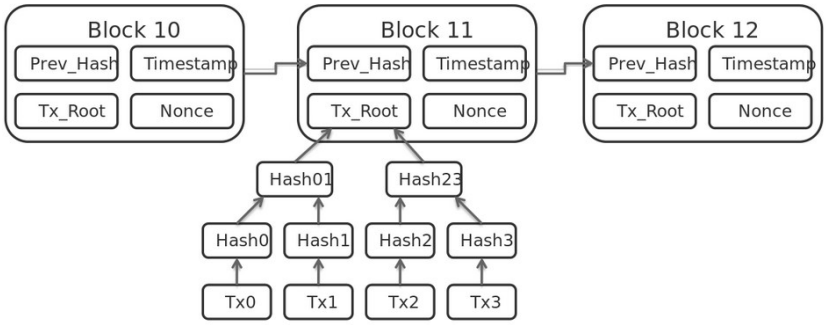

Figura 2.3: Cadeia de Blocos (*Blockchain*). [Nak08]

2.4 Modelos de Consenso

Depois de se perceber um pouco mais sobre cada um dos componentes de uma *blockchain*, surgem várias questões: como é publicado um bloco; Numa rede com imensos nodos a publicar blocos ao mesmo tempo, como resolver casos de conflitos e quem será o nodo a publicar o bloco. Estas questões são respondidas pelos vários modelos de consenso existentes. Estes terão sempre que contar com potenciais utilizadores maliciosos.

O primeiro bloco da rede chama-se *genesis*, representa o estado inicial da *blockchain* e já existe durante o começo da rede. Os utilizadores quando entram têm de concordar com o estado inicial da *blockchain*, e como todos os blocos são adicionados depois do bloco *genesis*, estes podem verificar a integridade da *blockchain* e concordar qualquer que seja o estado desta.

Em caso de conflitos, poderão existir duas ou mais versões válidas da *blockchain*. A rede resolve o conflito ao chegar a um consenso, que consiste na maior parte da rede concordar com um dos seus estados. Normalmente espera-se até à publicação do próximo bloco e a cadeia mais longa será a correta e adotada, porque é a que possui uma quantidade maior de trabalho realizado.

2.4.1 *Proof of Work* (*PoW*)

Numa prova de trabalho um utilizador publicará um bloco se conseguir resolver um desafio, o que exige um elevado gasto de recursos computacionais. Resolver este desafio é bastante trabalhoso, mas pode ser facilmente verificado através da sua solução. O desafio requerido consiste em calcular o valor *hash* do cabeçalho do bloco e que este seja menor do que um determinado valor alvo. Para isso deve-se realizar várias tentativas, mudando apenas o *nonce* do cabeçalho até que o valor de *hash* vá ao encontro com o requerido. Na *Bitcoin* e em outras criptomoedas este processo é chamado de *mining*, e os nodos que tentam resolver o desafio para publicar o bloco são conhecidos como *miners*. O valor alvo pode ser visto como a quantidade de zeros que um valor *hash* possui no seu início. A dificuldade do desafio pode ser ajustada e aumenta com o número de zeros. Na *Bitcoin* a dificuldade é ajustada para que um bloco seja publicado a cada 10 minutos e se a dificuldade aumentar, é devido ao aumento do poder computacional ou ao aumento do número de nodos publicadores. O aumento da dificuldade implica um acréscimo significativo no tempo de resolução do desafio, assim como, no consumo de energia. A esperança de uma recompensa que ultrapasse os gastos em energia, é o que motiva os nodos a resolver este desafio. Para não perderem a recompensa, os nodos são incentivados a ter um

bom comportamento, publicando apenas blocos válidos. Normalmente tendem a trabalhar em conjunto, distribuindo o desafio em partes iguais, diminuindo a carga computacional para cada um deles e repartindo a recompensa.

Uma vez resolvido o desafio por um nodo, este envia o bloco para outros nodos, que por sua vez verificam se o bloco é apenas composto por transações válidas e se o valor *hash* do cabeçalho vai ao encontro dos requisitos. Se assim for, é porque de facto o nodo resolveu o desafio e apresentou a solução correta e só então acrescenta o bloco á sua cópia da *blockchain* e reenvia o bloco para rede.

O uso deste desafio, computacionalmente difícil de resolver, ajuda a combater os "*Sybil Attack*" — um ataque informático, onde um atacante cria vários nodos para ganhar influência e exercer controlo sobre um sistema. No modelo *PoW*, o facto de um atacante possuir muitos nodos, não significa que ele ganhe controlo sobre a publicação de blocos. Para tal um atacante deverá possuir um poder computacional muito superior ao dos restantes nodos da rede, o que implica um gasto absurdo em *hardware*.

2.4.2 *Proof of Stake* (*PoS*)

Neste modelo de consenso o que interessa é a quantidade de moedas que um utilizador tem na sua *wallet* e não a quantidade de *hardware*. Quanto mais moedas tiver, maior será a sua aposta (*stake*) e, consequentemente, maior será a sua vontade no sucesso do sistema e menor será a vontade de subvertê-lo. No *PoS* (Prova de aposta/participação) os *miners* de um bloco são chamados de *forgers*, e o processo de publicação de blocos é chamado de *forging*. Uma vez libertadas, as moedas deixam de estar disponíveis para gastar. Ao contrário do *PoW* em que havia um gasto absurdo de recursos computacionais, tempo e de energia, no *PoS* não há necessidade destes gastos. Por isso as recompensas são recebidas apenas sobre a forma de taxas cobradas em cada transação do bloco e não com moedas adicionais como no *PoW*. Assim, as moedas são distribuídas pelos utilizadores em vez de serem geradas ao longo do tempo.

Como o modelo utiliza a quantidade de moedas de um utilizador em *stake* como factor de determinação de publicação de blocos, a probabilidade de um utilizador publicar um bloco está relacionada com o rácio da sua *stake* com as restantes *stakes* da rede. Tendo em consideração que os utilizadores mais ricos têm mais chances de publicar um bloco, algumas versões de *PoS* apresentam outras formas de beneficiar também os utilizadores com menos moedas. As mais comuns são:

- ***Randomised Block Selection***: Os *forgers* são escolhidos com base na combinação do menor valor de *hash* do bloco e das *stakes* mais altas.
- ***Coin age system or Coin age Proof of Stake (PoS)***: Neste modelo as moedas tem como propriedade a sua idade (*coin age*). Para selecionar os *forgers*, o modelo tem em conta o número de dias que as moedas em *stake* não foram usadas e a sua quantidade. Assim que o nodo publica o bloco, a idade das moedas em *stake* volta a zero e é necessário esperar um certo período de tempo antes de as poder voltar a apostar. Este modelo permite que utilizadores com mais moedas publiquem mais blocos, mas não dominarão o sistema, pois é preciso um tempo de espera antes de puder apostar as suas moedas novamente.

- ***Delegate-systems***: Neste modelo os utilizadores votam em outros nodos para que estes se tornem *delegates* (delegados). Quanto maior a *stake*, maior o peso do voto e, os mais votados, tornam-se *delegates*. Um *delegate* é responsável por validar transações e publicar blocos. Podem também ser responsáveis pela governação da *blockchain* e propor possíveis alterações a esta. As alterações serão aprovadas caso a maioria vote a favor da alteração. Os utilizadores podem também votar contra um nodo para que deixe de o ser. Por isso os *delegates* são incentivados a manter um bom comportamento.

No modelo de *PoS* em caso de conflitos, existe o problema do "*nothing at stake*", se existirem várias versões da *blockchain*. Um utilizador poderá actuar em todas elas, para aumentar a probabilidade de ser recompensado. O problema fará com que existam várias versões da *blockchain* ao longo do tempo, sem que seja reconhecida apenas uma delas como a correta. No modelo *PoW*, o problema não existe porque o utilizador teria de dividir o seu poder de computação para atuar em todas elas e arriscaria a nunca ser recompensado.

Relativamente ao modelo *PoW*, o *PoS* diminui o consumo energético da comunidade da *blockchain* (por exemplo, na comunidade da *Bitcoin*, a fatura eléctrica já é semelhante à de alguns países como a Dinamarca [DJY18]). Como os utilizadores investem em moedas da *blockchain* e não em *hardware* que pode ser re-utilizado noutras, existe uma maior lealdade para com a *blockchain*. Em termos de segurança, um atacante para assumir a publicação de blocos terá de possuir 51% de todas as moedas da rede e, tal como no modelo *PoW*, em que um atacante tem de possuir 51% de todo o poder de processamento da rede, implica custos muito elevados.

2.5 *Forks*

Um *fork* ocorre quando não existe um consenso unânime entre os nodos da rede e passam a existir duas ou mais versões da *blockchain*. A figura 2.4 pode-se visualizar que a partir de um bloco a rede divergiu e passaram a existir duas versões da *blockchain*. Este *fork* pode ser temporário e ocorre quase todos dias numa *blockchain* quando dois nodos publicam um bloco praticamente ao mesmo tempo, sendo que uma parte recebe um dos blocos primeiro e a outra parte recebe o outro primeiro. A rede chega a um consenso quando o próximo bloco é publicado e a maioria dos nodos escolhe a cadeia mais longa como correta. Para além de *forks* temporários existem mais duas categorias de *forks*: *soft fork* e *hard fork*. Podem ocorrer devido a mudanças no código da *blockchain* para resolver eventuais *bugs*, para adicionar novas funcionalidades, para resolver eventuais problemas de segurança, ou para a comunidade reverter transações realizadas num período de tempo, por desconfiar que foram feitas de forma ilícita e permitiram o roubo ou falsificação de moedas.

Um *soft fork* ocorre quando uma alteração no *software* da *blockchain* continua a ser compatível com a versão anterior. Os blocos publicados segundo as novas regras continuam a ser aceites pelos nodos que ainda não atualizaram o seu *software*. As novas regras só serão seguidas se a maioria da rede atualizar a sua versão. Um simples exemplo pode ser a redução do tamanho dos blocos: nodos que não atualizaram o seu *software* continuarão a considerar os blocos válidos pois esta mudança não viola as suas regras, mas os seus blocos serão rejeitados por nodos que atualizaram para a nova versão.

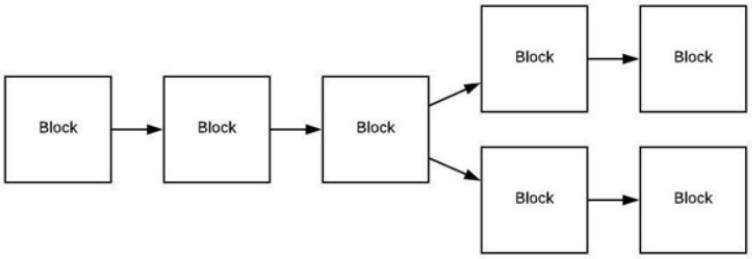

Figura 2.4: Exemplo de um *fork* em uma *blockchain*.

Já um *hard fork*, ocorre quando uma alteração no *software* da *blockchain* não é compatível com a versão anterior. Os blocos publicados segundo as novas regras não são aceites pelos nodos que ainda não atualizaram o seu *software*. Normalmente ocorrem quando se alteram as regras do modelo de consenso. Por exemplo, quando uma *blockchain* que usa *PoW* como modelo de consenso decide migrar para o modelo *PoS*, se a maioria dos nodos atualizarem não à qualquer problema. Mas se não for o caso, pode ocorrer uma divisão na comunidade da *blockchain* e passam a existir duas *blockchains* diferentes. A comunidade *Bitcoin* já se dividiu várias vezes devido a *hard forks*: no primeiro, a comunidade dividiu-se entre a *Bitcoin* e a nova *Bitcoin Cash*; no segundo, a nova comunidade da *Bitcoin Cash* voltou a dividir-se em *Bitcoin Cash* e *Bitcoin SV*. A *Ethereum* realizou um *hard fork* para resolver uma falha num *smart contact* chamado "*decentralized autonomous organization*" (DAO), que permitiu o roubou de 50 milhões de *Ethers* (moeda da *Ethereum*) [Lei17]. O *update* resolveu não só o erro, como permitiu que os lesados recuperassem as moedas que lhes foram roubados. Mas nem todos aceitaram a nova versão e a comunidade dividiu-se e, a versão mais antiga da *Ethereum*, mais conhecida como *Ethereum Classic*, continua a existir.

É necessário referir que nem sempre alterações ao *software* da *blockchain* é mau. Uma plataforma de qualidade deve sempre evoluir e ser constantemente atualizada. Para isso, é absolutamente necessário que uma *blockchain* passe por *forks* constantes, sejam eles *soft* ou *hard*. Deve-se é tentar evitar ao máximo que os *hard forks* criem divisões na comunidade. Os efeitos de uma divisão são imprevisíveis, podendo ditar uma perda acentuada no valor de mercado da moeda. Esta situação pode ser vista, por muitos, como uma oportunidade de investimento o que pode voltar a subir rapidamente o seu valor no mercado ou não. A *Bitcoin Cash* começou por valer cerca de 300$ e em 4 meses passou a valer 3000$, cerca de 10 vezes mais. O valor da *Bitcoin* no mercado também cresceu após a divisão, mas o mesmo não aconteceu na *Ethereum*.

2.6 Conclusão

O capítulo abordou os principais conceitos para a compreensão da tecnologia *blockchain* e como se pode utilizá-la para criar moedas digitais que não são geridas por nenhuma entidade. O principal objectivo era mostrar como se transacionam moedas, já que o próximo capítulo se foca em como se pode chegar à privacidade nas transações. Mas o futuro será das *blockchains* que permitem não só a criação da sua própria moeda mas também a execução de *smart contracts*.

Capítulo 3

Privacidade em Transações na *Blockchain*

3.1 Introdução

Existem várias técnicas para melhorar a privacidade das *blockchains* em que se trasanciona moedas de endereços para endereços. É o caso do *CoinJoin*, método em que se junta vários pagamentos de diferentes carteiras numa única transação, dificultando a entidades fora da transação determinar qual o remetente que pagou a destinatário ou a destinatários. Mas mesmo assim continua a ser possível rastrear as moedas transferidas.

Surge então a necessidade do aparecimento do termo *confidencial transactions* (transações confidenciais) proposto por Adam Back num fórum *Bitcoin*, *BitcoinTalk*, com o título "*bitcoins with homomorphic value*" [Bac13], e mais tarde introduzido por Greg Maxwell [Max16]. O capítulo abordará este tipo de transações e os métodos criptográficos por detrás destas. Neste contexto irão ser descritos o ZKP, os *Pedersen commitments* ou *homomorphic encryption* e as várias técnicas existentes que permitem continuar a verificar este tipo de transações, como a *Zk-Snarks*, a *Zero-knowledge scalable and transparent argument of knowledge (Zk-Starks)* e, a proposta de implementação desta dissertação, as *Bulletproofs*.

3.1.1 *Zero-Knowledge Proofs* (*ZKP*)

ZKP, ou provas de conhecimento-nulo, é uma poderoso protocolo criptográfico que permite assegurar a veracidade de uma afirmação sem revelar nenhuma informação sobre os seus dados, além da sua veracidade. Por exemplo, para quando se quer provar que se sabe os valores de entrada para qual uma função debita determinado valor, sem revelar os valores de entrada.

Estes sistemas de provas, quando interativas, são constituídos por duas entidades: o *prover* — entidade que quer provar que sabe um segredo para o qual uma afirmação é verdadeira (mais conhecido na gíria da criptografia como **P**eggy), e o *verifier* — entidade que verifica a veracidade da afirmação (mais conhecido por **V**ictor). Durante a prova são trocadas várias mensagens. A primeira é enviada pela Peggy, na qual consta a sua afirmação. O Victor não sabe o segredo da Peggy e logo não pode desde já aceitar a sua veracidade. No entanto, pode construir vários desafios para testar a veracidade da afirmação, sendo que no final, o Victor só aceita a afirmação se a Peggy responder corretamente a todos os desafios propostos. Para isso esta tem de conhecer o segredo e a afirmação tem de ser obrigatoriamente verdadeira.

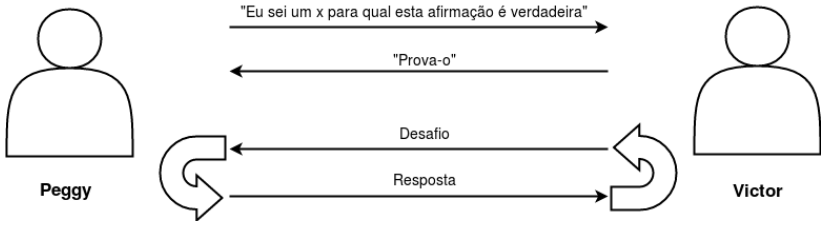

Figura 3.1: Esquema de um sistema de prova de conhecimento nulo iterativa.

O sistema de prova referido na figura 3.1, para um conjunto S, é um jogo entre duas entidades: a Peggy que possui recursos computacionais ilimitados, na qual não se pode confiar, e o Victor que possui recursos limitados, executando em tempo polinomial. O sistema satisfaz as seguintes propriedades [Gre17]:

- ***Completeness* ou Completude:** Se $x \in S$, então o Victor aceita todas as respostas depois de interagir com a Peggy sobre o valor comum x.
- ***Soundness* ou Corretude:** Para um dado polinómio p, este assegura que para todo o $x \notin S$ e para todas as possíveis respostas da Peggy aos desafios, P^*, Victor rejeita-as com probabilidade de pelo menos $\frac{1}{p(|x|)}$ depois de interagir com P^* sobre o valor comum x.

Por outras palavras, a propriedade *completeness* garante-nos que se a afirmação for verdadeira e se o Victor seguir corretamente o protocolo, uma Peggy honesta consegue convencer o Victor sobre este facto com grande probabilidade. Já a propriedade *Soundness*, se a afirmação for falsa, uma Peggy batoteira só consegue convencer um Victor honesto acerca da veracidade da afirmação com uma probabilidade negligenciável, ou seja, com muito pouca probabilidade. Estas provas só são de conhecimento-nulo se respeitarem a seguinte propriedade:

- ***Zero-knowledge* ou Conhecimento-nulo:** Se a afirmação for verdadeira nenhum Victor batoteiro descobrirá nada para além da sua veracidade.

3.1.2 Simples exemplos de ZKP

3.1.2.1 *The Ali Baba cave*

Um exemplo bastante simples é a gruta de Ali Baba (a gruta mágica) publicado pela primeira vez no *paper* "*How to Explain Zero-Knowledge Protocols to Your Children*" [JJ+90]. Até aos dias de hoje tem sido adaptado e aproveitado para efeitos de ensino inicial deste tema.

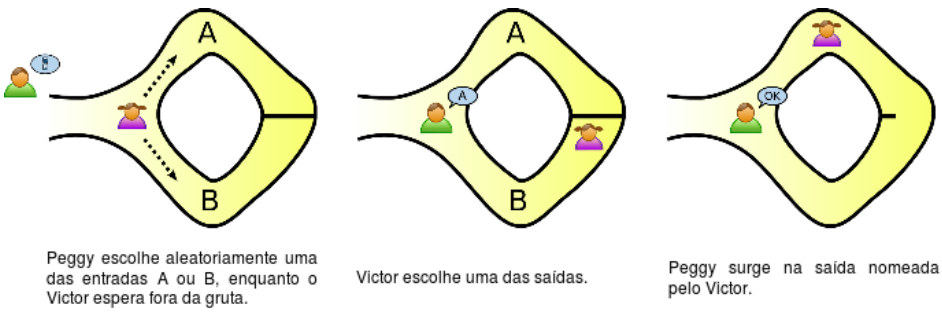

Figura 3.2: *The Ali Baba cave*. [Won14]

Como é demonstrado na figura 3.2, a gruta mágica tem uma bifurcação o que lhe dá forma de um anel, com uma entrada do lado esquerdo e uma porta mágica do outro, a qual só abre com

uma palavra secreta. Assim, se a Peggy escolher um dos caminhos só percorrerá a gruta e sairá na ponta oposta do caminho que escolheu, se souber a palavra secreta que abre a porta.

A Peggy diz conhecer a palavra secreta com a qual a porta mágica abre e por sua vez o Victor quer que esta prove que conhece mesmo. Mas a Peggy não quer que o Victor fique a saber o segredo. Então os dois combinam uma maneira que convença o Victor que a Peggy conhece de facto o segredo, sem que este seja revelado.

A Peggy entra na gruta e escolhe um dos caminhos, A ou B, enquanto que o Victor fica à espera fora da gruta, pois não lhe é permitido ver qual o caminho escolhido. Quando a Peggy estiver perto da porta mágica, o Victor entra na gruta e escolhe aleatoriamente o caminho por onde a Peggy terá que sair. Se o Victor escolher o caminho que a Peggy escolheu, esta só terá de voltar atrás. Mas se escolher o caminho contrário, a Peggy terá de abrir a porta mágica.

Supondo que a Peggy não sabe a palavra secreta, esta apenas consegue sair pela entrada do caminho que escolheu. Como o Victor escolhe uma das saídas aleatoriamente, ela apenas tem 50% de probabilidade de ser bem sucedida. Se a Peggy for posta à prova várias vezes as suas hipóteses de sucesso iram diminuir significativamente, tornando-se uma probabilidade negligenciável. O Victor acabará sempre por perceber se a Peggy sabe de facto a palavra secreta.

E porque simplesmente o Victor não diz à Peggy para escolher um dos caminhos e sair pelo o outro? Deste modo, a Peggy teria de usar o seu conhecimento do segredo logo na primeira interação e no final o Victor saberia se esta sabe de facto o segredo. O problema é que assim o Victor poderia seguir a Peggy e tentar escutar (*eavesdrop*) o segredo. Ao aleatorizar o caminho escolhido pela Peggy é reduzida a probabilidade de *eavesdrop* do Victor.

3.1.2.2 *Interactive Sudoku ZKP*

Um exemplo real e um pouco mais complexo é o jogo do Sudoku: como provar que se possui uma solução de um *puzzle* sem a revelar [GNPR07]. O jogo consiste numa tabela $n \times n$, onde $n = k^2$, dividida em sub-tabelas $k \times k$. Algumas das células já se encontram preenchidas com números de $[1, .., n]$. O objectivo é preencher as restantes células com números do mesmo intervalo de forma a que um número apareça apenas uma vez por linha, coluna e sub-tabela. Sudoku é um problema *Non-Deterministic Polynomial time (NP)*, em português tempo polinomial não determinístico, pois pode levar algum tempo a resolver. No entanto vendo uma solução é fácil de a verificar, e como é referido no *paper* [GMW91] existe uma ZKP para todas as linguagens em NP. Assim, pode ser visto como um problema de coloração de grafos.

	9			8		4		
		2		4	1			5
3							6	
	1							
7	6			2			1	9
							8	
	2							8
5			2	9		3		
		4		5			2	

Figura 3.3: Puzzle Sudoku 9x9.

1	9	7	6	8	5	4	3	2
6	8	2	3	4	1	7	9	5
3	4	5	9	7	2	8	6	1
4	1	8	5	6	9	2	7	3
7	6	3	8	2	4	5	1	9
2	5	9	7	1	3	6	8	4
9	2	6	4	3	7	1	5	8
5	7	1	2	9	8	3	4	6
8	3	4	1	5	6	9	2	7

Figura 3.4: Solução para *puzzle*.

O Victor não consegue resolver um *puzzle* Sudoku 9x9 (figura 3.3) e afirma que este não tem solução. Por sua vez a Peggy diz ter uma solução para o jogo (figura 3.4). O Victor quer que esta prove que há uma solução, mas não quer que esta a revele, pois quer resolver o jogo sem ajudas.

Para não revelar qualquer informação sobre a solução, a Peggy escolhe uma permutação aleatória σ dos números $[1, ..., 9]$, na permutação da figura 3.5, $\sigma(1) = 2$, $\sigma(2) = 8$, $\sigma(3) = 6$, $\sigma(4) = 5$, $\sigma(5) = 4$, $\sigma(6) = 9$, $\sigma(7) = 1$, $\sigma(8) = 7$, $\sigma(9) = 3$. Depois de preencher a tabela com a permutação, ela esconde o conteúdo de cada célula do *puzzle* e apresenta-o ao Victor. Na verdade a Peggy pode esconder os valores de uma forma criptográfica ao gerar um *nonce* aleatório para cada célula e achar o valor *hash* do conteúdo da célula juntamente com o *nonce*.

2	3	1	9	7	4	5	6	8
9	7	8	6	5	2	1	3	4
6	5	4	3	1	8	7	9	2
5	2	7	4	9	3	8	1	6
1	9	6	7	8	5	4	2	3
8	4	3	1	2	6	9	7	5
3	8	9	5	6	1	2	4	7
4	1	2	8	3	7	6	5	9
7	6	5	2	4	9	3	8	1

Figura 3.5: *Puzzle* preenchido com a permutação.

O Victor pode pedir à Peggy para revelar uma das linhas, uma das colunas, uma das sub-tabelas ou ver a permutação do puzzle inicial. Ou seja, o Victor pode fazer uma de 28 escolhas. Se o Victor escolher a terceira linha a Peggy irá revelar a figura 3.6. O Victor aceita a linha se todos os dígitos aparecerem apenas uma única vez nela. O mesmo se passaria se ele escolhe-se umas das colunas, sub-tabelas ou outra linha. Caso escolhe-se ver a permutação do puzzle inicial seria-lhe revelado a figura 3.7, e aceita-a se esta for mesmo uma permutação do puzzle inicial. De notar que o Victor não consegue aprender algo sobre a solução, pois todas as possíveis permutações são escolhidas com igual probabilidade. E cada vez que o Victor pedir à Peggy para revelar uma das suas escolhas uma nova permutação é escolhida aleatoriamente.

6	5	4	3	1	8	7	9	2

Figura 3.6: Terceira linha da permutação.

	3			7		5		
		8		5	2			4
6							9	
	2							
1	9			8			2	3
							7	
	8							7
4			8	3		6		
		5		4			8	

Figura 3.7: Permutação do *puzzle* original.

Agora fica a questão, será que o Victor pode acreditar que há mesmo uma solução para o *puzzle*? Imaginando que a Peggy é batoteira, e que na realidade ela não tinha a solução para o *puzzle*, se o Victor fizer as sua escolhas aleatoriamente, a probabilidade de este apanhar a Peggy é de $\frac{1}{28}$. Logo a probabilidade de ela não ser apanhada é bastante alta, $\frac{27}{28}$. Repetindo o desafio várias vezes, digamos 150 vezes, as hipóteses de não ser apanhada diminuem significativamente para uma probabilidade negligenciável, $\frac{27}{28}^{150} < 0.004$. O Victor acabará sempre por perceber se a Peggy possui uma solução válida para o *puzzle*.

3.1.3 *Proofs of Knowledge*

Uma prova de conhecimento é algo mais do que provar que algo é verdadeiro e depende realmente do que o *Prover* sabe. Por exemplo, pode-se provar que um número N é um número composto mesmo não sabendo a sua fatorização — apenas se prova um facto. Numa prova de conhecimento o *Prover* mostra que conhece a fatorização de N.

3.1.4 Σ-*protocols*

Até agora vimos apenas provas em que duas pessoas tem necessariamente de interagir presencialmente, sendo apenas exemplos ilustrativos para compreensão deste tipo de protocolos. Na realidade a interação é entre dois computadores, muitas vezes referidos como *Turing machines* no mundo das ciências da computação, devido ao facto de Alan Tuting ter sido o "pai" da computação. As propriedades definidas anteriormente são mais difícil de demonstrar e, em problemas com grande complexidade, executar o protocolo várias vezes pode levar a um grande custo computacional. Assim surgem os Σ-*protocols* [Dam10].

Seja R uma relação binária, i.e., R é um sub-conjunto de $\{0,1\}^* \times \{0,1\}^*$, onde a única restrição é se $(x, w) \in R$, então o comprimento de w é no máximo $p(|x|)$ para um dado polinómio $p()$. Para um par $(x, w) \in R$, x pode ser visto como uma instância de um problema computacional e w como a sua solução, muitas vezes referido como a *witness* (testemunha) para x. A letra grega Σ representa a forma como decorre o fluxo do protocolo. Uma interação entre a Peggy e o Victor sobre um valor comum x e, w como valor privado da Peggy, tal que $(x, w) \in R$:

1. Peggy envia uma mensagem a.
2. Victor envia uma *string* t-bit aleatória e.
3. Peggy envia uma resposta z, e Victor aceita-a ou rejeita-a com base nos dados que tem ao seu dispor (x, a, e, z).

Então um protocolo P é um Σ-*protocol* para a relação R se possuir o fluxo referido em cima e as seguintes propriedades:

- ***Completeness*:** Referida já anteriormente.
- ***Special Soundness*:** Também conhecida como a extração do conhecimento. Para todo $(x, y) \in R$, existe um algoritmo em tempo polinomial E, que permite extrair w de duas comunicações válidas sobre o valor x, $(a, e, z), (a, e', z')$ com $e \neq e'$.
- ***Honest verifier zero-knowledge*:** Existe um simulador em tempo polinomial M, o qual sobre o valor x e um e aleatório é capaz de produzir uma comunicação válida (a, e, z) com a mesma distribuição de probabilidade que uma comunicação honesta entre a Peggy e o Victor.

A segunda propriedade assegura não só a propriedade *soundness*, já referida anteriormente, como também a prova do conhecimento que a Peggy sabe de facto w. A propriedade *zero-knowledge* diz que para qualquer *verifier* existe um simulador que gera uma comunicação com a mesma distribuição que uma comunicação entre a *Peggy* e o *Victor*. Σ-*protocols* não são *zero-knowledge*, porque não se consegue construir um simulador para qualquer *verifier*, já que um *verifier* malicioso pode realizar escolhas imprevisíveis de e. Para simular é necessário

adivinhar e antecipadamente. Mas dado o espaço de amostragem exponencial e, as hipóteses de acontecer são negligenciáveis. Resumindo, a propriedade *honest verifier zero-knowledge* consiste num *verifier* honesto que segue exatamente o protocolo e não tem comportamento malicioso. Contudo os Σ-*protocols* são uma boa ferramenta para se conseguir eficientemente *zero-knowledge* [Lin11]. Para isso o *verifier* envia primeiro um *commitment* para o desafio e antes de o *prover* enviar a mensagem a. Um *commitment* de e é um compromisso para e sem revelar e, para mais tarde, quando o *verifier* enviar e se possa verificar que de facto o *commitment* enviado era um *commitment* de e.

3.1.4.1 Schnorr's *Protocol*

Um dos primeiros Σ-*protocols* foi proposto por Claus Schorr [Sch91]. É uma prova de conhecimento nulo que se baseia no problema do logaritmo discreto. Este protocolo é muito usado como protocolo de identificação em que se prova que um utilizador é quem diz ser, permitindo a este autenticar-se num servidor. Este esquema é mais seguro que o típico método *username-password*, mas também tem utilizações no contexto da computação segura, e.g., provar que se conhece um valor cifrado.

O protocolo é definido sobre um grupo cíclico[1] G de ordem q, com o gerador g. Seja $R = \{(x, w) : x = g^w\}$ a relação binária, o objectivo do protocolo é provar que $(x, w) \in R$, onde apenas conhecendo x é computacionalmente difícil calcular w. A Peggy interage com o Victor com intuito de provar que conhece o logaritmo discreto de x, como é demonstrado na figura 3.8:

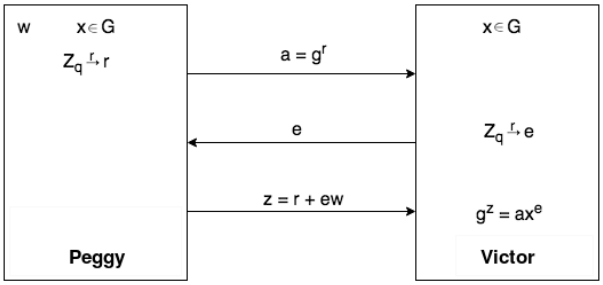

Figura 3.8: Schnorr's *Protocol*.

1. Peggy escolhe aleatoriamente um número $r \in \mathbb{Z}q$ e envia $a = g^r$ ao Victor.
2. Victor escolhe aleatoriamente um desafio $e \in \mathbb{Z}q$ e envia-o à Peggy.
3. Peggy responde ao desafio com a resposta $z = r + ew$ e envia-a ao Victor. O Victor só a aceita se $g^z = ax^e$.

De notar que no inicio do protocolo ambos possuem os valores públicos g e q. Para lém destes, a Peggy possui o par (x, w) o qual diz pertencer a R, enquanto que o Victor apenas possui x.

Apenas falta demonstrar as propriedades referidas anteriormente para mostrar que o protocolo é um Σ-*protocols*:

[1] Um grupo, é um grupo cíclico se este poder ser gerado por um único elemento do grupo. Supondo que G é um grupo cíclico gerado por $g \in G$ de ordem p, então todos os elementos de G podem ser representados por g^k com $k \in \mathbb{Z}p$.

- ***Completeness*:** $\forall r, e : g^z = g^{r+ew} = g^r(g^w)^e = ax^e$
- ***Special soundness*:** É necessário construir um extrator que calcule w quebrando o problema do logaritmo discreto. Por outras palavras, reduz-se a propriedade *soudness* do protocolo ao problema do logaritmo discreto com a ajuda do extrator E. O extrator interage com a Peggy como sendo o Victor da seguinte forma:

  1. A Peggy envia $a = g^r$.
  2. O Extrator escolhe aleatoriamente um $e \in \mathbb{Z}q$ e envia o desafio.
  3. A Peggy responde ao desafio enviando $z = r + ew$.
  4. O Extrator recebe o desafio e rebobina a Peggy para o passo 2 e escolhe um novo desafio $e' \in \mathbb{Z}q$ para a Peggy.
  5. A Peggy responde ao novo desafio enviando $z' = r + e'w$.
  6. O Extrator extrai w ao calcular $(z - z')/(e - e')$.
     $(z - z')/(e - e') = ((r + ew) - (r + e'w))/(e - e') = (e - e')/(e - e').w = w$

  De notar que isto é apenas uma simulação. Na realidade a Peggy escolhe um novo r aleatoriamente para cada execução do protocolo. A utilização do mesmo r para pelo menos duas execuções diferentes do protocolo, implica uma grave vulnerabilidade, já que um atacante poderia facilmente calcular w. Numa boa implementação do protocolo, r é escolhido aleatoriamente de um espaço amostral extremamente grande. Por isso a probabilidade de um $r \in \mathbb{Z}q$ ser escolhido mais do que uma vez é negligenciável.

- ***Honest Verifier Zero Knowledge*:** O objectivo é construir um simulador que crie uma interação, na qual o *prover* tenta provar que conhece w para x, mas na realidade não conhece. O simulador assume que o *verifier* é honesto e irá escolher os seus desafios aleatoriamente e não com base nos valores fornecidos pelo *Prover*. O Simulador funciona da seguinte forma:

  1. O Simulador envia um $a = g^r$.
  2. O Victor envia um desafio $e \in \mathbb{Z}q$.
  3. O Simulador rebobina o Victor, escolhe aleatoriamente um $z \in \mathbb{Z}q$, calcula $a' = g^z/x^e$, e envia a' como primeira mensagem.
  4. O Victor envia de novo o desafio e.
  5. O simulador responde ao desafio enviando z.

  A comunicação (a', e, z) é de facto válida e aceite pelo Victor. Contudo o Simulador não conhecia w. Isto prova que, se for possível rebobinar o *verifier*, é possível enganá-lo, fazendo com que ele acredite que alguém possui o valor secreto. Como não é possível distinguir uma simulação de uma execução real, o protocolo é *zero knowledge* contra um *verifier* honesto.

3.2 *Non-Interactive Zero-Knowledge Proofs* em transações na Blockchain

Se as transações numa *blockchain* são tornadas públicas para que todos as possam verificar, alguns dos seus campos com dados mais sensíveis são expostos, como os endereços e a quantidade de moedas transferidas. Assim, é necessário arranjar uma solução para esconder esses campos. A forma mais óbvia de o fazer seria cifrar os campos, mas isto tornava impraticável o ato de verificação. A solução é mesmo usar ZKP, visto ser um protocolo que prova que algo está correto sem revelar nada além da sua veracidade. Até agora as provas referidas são interativas, onde duas entidades interagem entre si, trocando mensagens a fim de provar que algo está de facto correto. No fim da prova só o *verifier* pode dizer algo acerca da veracidade da prova, sendo que entidades fora da interação nada podem dizer acerca da veracidade desta, pois o *prover* e o *verifier* podem ter conspirado entre si. Se as transações são verificadas por mais do que uma entidade e estas são verificadas em qualquer altura, mesmo por entidades que tenham estado *offline* no momento da transação, as provas não podem ser interativas. Numa prova de conhecimento nulo não-interativa (*Non-Interactive Zero-Knowledge Proofs (NIZK)*), é possível publicar a prova juntamente com a transação para que seja verificada a sua veracidade e a validade da transacção em qualquer altura.

Uma forma eficiente de tornar uma prova interativa em uma não-interativa é utilizar a heurística de Fiat-Shamir. A ideia deste método é eliminar a interação nos protocolos *public coin*[2], tornando os protocolos anteriores, onde há troca de três mensagem, num simples esquema de assinatura. David Pointcheval e Jacques Stern provaram que o método é seguro contra Chosen-plaintext atack (CPA) (ataques com texto limpo escolhido) sob o modelo *random oracle*[3]. O protocolo referido anteriormente pode ser transformado numa prova não interativa ao utilizar uma função de *hash* criptográfica para obter os desafios, da seguinte forma:

1. A Peggy diz conhecer w para o qual $x = g^w$, e quer prová-lo.
2. A Peggy escolhe aleatoriamente um número $r \in \mathbb{Z}q$ e calcula $a = g^r$.
3. A Peggy obtém o desafio ao calcular $e = H(g, x, a)$, onde $H()$ representa uma função de *hash*.
4. A Peggy calcula $z = r + ew$, e publica a prova ao publicar o par $\pi = (a, z)$.
5. Qualquer entidade pode verificar a prova π, ao verificar se $g^z = ax^e$, onde $e = H(g, x, a)$.

Numa *blockchain* onde um bloco pode conter várias transações, as provas de conhecimento nulo devem ser o mais curtas possíveis para não aumentar em demasia o tamanho da *blockchain* e para diminuir o tempo que leva a computar e verificar a prova. Isto sem comprometer a segurança da *blockchain*. Graças à tecnologia de *blockchain* estar no centro das atenções e a necessidade de haver uma melhor privacidade nelas, a investigação para construir provas NIZK mais eficientes aumentou. Da pesquisa realizada foram escolhidas três técnicas, sendo que duas delas já são utilizadas em *blockchains*.

[2]Public coin significa que as escolhas aleatórias do *verifier* são tornadas públicas.

[3]O modelo *random oracle* é usado para analisar a segurança de um esquema que precisa de uma função de *hash*, substituindo a função *hash* por um *random oracle*. Uma função de *hash* não é um *random oracle* porque para problemas diferentes, o mesmo valor de entrada, a função debita o mesmo valor.

3.2.1 *Zero-knowledge succinct non-interactive argument of knowledge* (Zk-Snarks)

Zk-Snarks foi descrito pela primeira vez no *paper* [BCCT12]. Nele é descrito que através de *extractable collision-resistant hash fuctions*[4] é possível construir um protocolo *SNARK*. As siglas do acrónimo Zk-Snarks significam:

- ***Zero-Knowledge*:** O *verifier* não descobre nada da afirmação além da sua veracidade.
- ***Succint*:** Mesmo para afirmações de grandes dimensões, as provas são curtas e de tamanho fixo, por volta de 288 bytes, e podem ser facilmente verificadas em poucos milissegundos.
- ***Non-interactive*:** O *Prover* publica a prova e qualquer *verifier* a pode verificar, sem que haja alguma interação entre eles.
- ***Argument*:** A propriedade *soundness* apenas é assegurada se o *Prover* tem poder computacional limitado. Logo Zk-Snarks são considerados argumentos e não provas. Isto significa que um *prover* desonesto tem uma chance bastante baixa de conseguir enganar um *verifier*. Mas caso tenha poder computacional ilimitado pode criar provas falsas que são consideradas válidas pelo *verifier*.
- ***Knowledge*:** A prova não pode ser construída sem se conhecer a testemunha (*witness*) para a qual a afirmação é válida.

Zk-Snarks pode ser visto como três algoritmos:

1. O gerador de chaves G, recebe como valor de entrada um valor secreto λ, e debita duas chaves: uma para a prova pk e outra para a sua verificação vk.

   $G(\lambda) -> (pk, vk)$

2. O *Prover* P, recebe como parâmetros de entrada, a chave da prova gerada pelo algoritmo anterior, a afirmação e o segredo. No final retorna a prova.

   $P(pk, x, w) -> prf$

3. O *Verifier* V, recebe como parâmetros de entrada a prova e a afirmação, e retorna *TRUE* caso a prova esteja correta ou *FALSE* caso contrário.

   $V(prf, x) -> bool$

De notar que o valor secreto λ que é usado no gerador de chaves, deve permanecer desconhecido. Se alguém tiver acesso a estes valores secretos, chamados parâmetros de aleatoriedade, pode criar provas falsas que são consideradas válidas mesmo que o *prover* não conheça a testemunha w. A este esquema inicial de geração de chaves dá-se o nome de *trusted setup*, porque as chaves devem ser geradas por uma entidade de confiança. Esta deve ter um comportamento honesto, nunca revelando os parâmetros secretos. Esta característica é uma das maiores desvantagens do *Zk-Snarks*.

A *Zcash* foi uma das primeiras criptomoedas a implementar o *Zk-Snarks* de forma eficiente e segura [Zca] [SCG+14]. O uso do *Zk-Snarks* na Zcash tem como propósito a criação e validação

[4] Uma função hash é *extractable* se para um dado adversário A, a única maneira de retirar elementos da imagem da função de *hash* é conhecer a pré-imagem correspondente. E é *collision-resistance* se for difícil encontrar dois valores de entrada com o mesmo valor *hash*.

de *shielded transaction*. Nestas transacções os campos mais sensíveis, como os endereços e os valores envolvidos, não são revelados e, mesmo assim, continua a ser possível verificar a validade da transação. O remetente de uma *shielded transaction* cria uma prova, na qual mostra com grande probabilidade que a soma dos *inputs* não é menor que a soma dos *outputs* e que possuí as chaves privadas que lhe dá a autoridade de gastar as moedas como *inputs*. Estas chaves são referidas como *private spending keys*, estando ligadas criptograficamente a uma assinatura para impedir que a transação seja alterada por utilizadores que não conheçam as chaves. Para além disso, a prova mostra também que os *inputs* ainda não foram gastos.

Para reduzir as chances de um atacante obter os parâmetros privados, a *Zcash* implementou o *trusted setup* de uma forma descentralizada. Isto significa que os parâmetros públicos usados na construção e verificação da prova não são gerados apenas por uma entidade, mas sim por um conjunto de nodos da rede, através de um protocolo *Multi-Party Computation (MPC)* — esta etapa é conhecida como *The Cerenomy* [Wil16]. Cada um dos nodos usa uma parte do valor secreto para gerar uma parte dos parâmetros públicos. No final, os parâmetros públicos são a junção de cada uma das partes e cada nodo deve eliminar o valor secreto que usou. Ao distribuir o parâmetro secreto — designado de *Toxic Waste* na *Zcash* — pelos vários nodos, dificulta a ação de um possível atacante na obtenção deste, já que se pelo menos um dos nodos é honesto, este elimina o valor secreto que usou. Deixa então de ser possível recuperar a totalidade do parâmetro secreto. Mas mesmo que isto não seja possível, não significa que ninguém consiga subverter o sistema. Alguém que o consiga subverter, como os valores transacionados deixam de estar visíveis, o atacante pode criar moedas do nada, criando inflação. Inflações não detectáveis põem em risco o valor de mercado de uma criptomoeda. Mas mesmo o medo de as ter pode ser prejudicial, já que alguém podia criar o pânico dizendo que o sistema foi subvertido.

3.2.2 *Zero-knowledge scalable and transparent argument of knowledge* (Zk-Starks)

Zk-Starks é uma tecnologia ainda recente, mas já é vista por muitos como um possível substituto do *Zk-Snarks*. Surgiu em 2018 no *paper* [BSBHR18], publicado por Eli Ben-Sasson e outros co-autores. A maior vantagem em relação ao *Zk-Snarks* é o facto de não necessitar de um *trusted setup*. Daí a sigla "T" do acrónimo ser de *transparent*, em português transparente. O "S" do acrónimo é de *Scalable*, uma vez que, uma das principais preocupações do *Zk-Starks* é a escalabilidade dos sistemas que o usam. Por exemplo, numa *blockchain* é importante que as provas tenham um tamanho reduzido e sejam rápidas de verificar e a *Zk-Starks* permite construir provas e verificá-las de forma rápida, mantendo a segurança dos sistemas.

Em comparação com a Zk-Snarks, uma das vantagens já referidas, é o facto de não necessitar de um *trusted setup*. Assim deixa de haver a necessidade de confiança entre os participantes. Na *Zcash* era necessário confiar em pelo menos um dos participantes e, embora o *setup* seja descentralizado, não é suficientemente comparado com o número de nodos da rede, o que causa os problemas típicos de sistema pouco descentralizados. Outras das vantagens é que a sua suposição criptográfica depende de funções *hash* criptográficas resistentes a colisões e, por isso, as *Zk-Starks* são consideradas resistentes a ataques de computadores quânticos. O mesmo não acontece no *Zk-Snarks* que usa criptografia em curvas elípticas que dependem do problema do logaritmo discreto, o qual dizem ser susceptível aos avanços no poder de computação que os

computadores quânticos podem vir a oferecer.

A maior limitação do *Zk-Starks* é o tamanho das provas, o que faz com que sejam demasiado grandes para serem usadas em *blockchains*. Mas a investigação sobre a tecnologia continuará e certamente que os programadores arranjaram maneiras de reduzir o tamanho da prova. Isto tornará o *Zk-Starks* ainda mais atraente do que já é. O próprio Eli Ben-Sasson, a cabeça por detrás do *Zk-Starks*, já fundou a sua própria empresa, *StarkWare Industries*, que tem como foco a sua investigação e aplicação em todos os tipos de *blockchains*. [BS17]

3.2.3 *Bulletproofs*

Bulletproofs são NIZK curtas, e tal como o *Zk-Starks* não necessitam de um *trusted setup*. Foram divulgadas pela primeira vez em 2017 no *paper* [BBB+18], sendo Benedikt Bünz o seu principal autor. Ao contrários das técnicas anteriores que foram implementadas de forma a permitir *shielded transactions*, o autor das *Bullletproofs* focou-se em garantir eficientes *confidential transactions*. Neste tipo de transacções apenas os valores transacionados são escondidos e todas as transações contêm uma prova de que esta é válida. O *paper* conta também com nomes como Greg Maxwell, um dos nomes por de trás das *confidential transactions* [Max16], Dan Boneh, um dos grandes nomes da criptografia, Jonathan Bootle, o autor das técnicas *Bootle et al*.[5] [BCC+16], as quais serviram como base na construção das *Bulletproofs*, entre outros.

As *confidential transactions* necessitam de uma *range proof*, prova de intervalo, nas quais é possível provar que um número cifrado está entre um dado intervalo, e.g. $w \in [0, 2^{64})$, sem revelar nada acerca do número. As *Bulletproofs* comparativamente às técnicas vistas anteriormente permitem realizar de uma forma mais eficiente e segura *range proofs*. Não precisam de um *trusted setup* como no *Zk-Snarks* e as provas não são tão grandes como no *Zk-Starks*. Em comparação com as *range proofs* anteriores, que eram realizadas através de Σ-*Protocols* com a heurística de *Fiat-Shamir*, as *Bulletproofs* permitem que as provas apenas cresçam logaritmicamente consoante o tamanho do circuito, em vez de linearmente. Através de um protocolo *MPC*, é possível tirar partido do crescimento logarítmico e agregar várias provas em apenas uma, ou seja, é possível juntar vários valores e provar que todos estão num dado intervalo em apenas uma prova. A figura 3.9 permite verificar o crescimento de uma *range proof* em *Bulletproof* com 10 provas agregadas, em relação às *range proofs* anteriores e ao *Zk-Snarks*. Deste modo, as *Bulletproofs* permitem juntar várias transações em apenas uma (protocolo *CoinJoin*) de uma forma eficiente e passam não só a oferecer confidencialidade, como também anonimato, melhorando a privacidade nas transações. Benedikt Bünz refere que se a *Bitcoin* usa-se *confidential transactions*, onde na altura existiam 50 milhões de *UTXOs* de 22 milhões de transações, usando uma representação de 52 bits (que cobria os valores de 1 até 21 milhões de *bitcoins*), resultaria em 160 Gigabyte (GB) de dados de *ranges proofs*, caso fossem usadas as *ranges proofs* anteriores. Ao usar as *Bulletproofs* a redução no tamanho seria de factor 10, passando apenas para 17 GB [BBB+18].

[5] Bootle et al. permitem ZKP curtas, que apenas crescem logaritmicamente consoante o tamanho da afirmação em vez de linearmente, para circuitos aritméticos arbitrários.

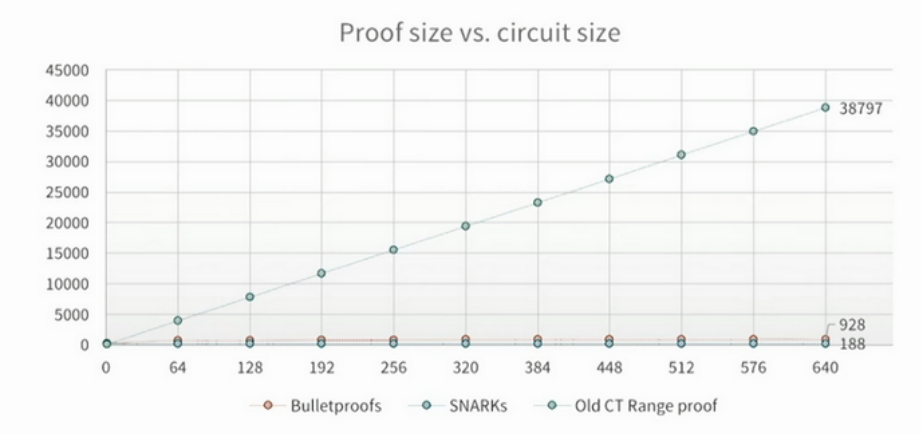

Figura 3.9: Tamanho da prova vs tamanho do circuito para uma agregação de 10 provas [Bü18].

A criptomoeda *Monero* foi uma das primeiras a implementar e a usar *Bulletproofs* nas suas *confidential transactions*. Anteriormente, a *Monero* usava as antigas *range proofs*, que cresciam linearmente com o número dos *outputs* e o número de bits do intervalo, e caso uma transação contivesse mais do que um *output* era necessário mais do que uma *range proof*. Uma simples transação com dois *outputs*, tinha um tamanho de 13.2 Kilobyte (kB). Com as *Bulletproofs*, uma transação com duas *range proofs*, passaria a ter apenas 2.5 kB, um decréscimo significativo, que poderia ser ainda maior, caso se usa-se uma única *range proof* para os dois *outputs*. Uma *range proof* com vários *ouputs* pode causar ataques de *Denial-of-service (DoS)*, porque como o processo de verificação aumenta linearmente com o número de *outputs*, um atacante poderia realizar uma transação de poucos valores mas com vários *outputs* que iria exigir elevados recursos computacionais para a verificar. A *Monero* teve o cuidado de primeiro implementar uma *range proof* para cada *output* e, só depois, as *range proofs* para vários *outputs*, pois é necessário aumentar as *fees* de uma transação consoante o número de *outputs*. [Noe17]

3.2.4 *Zk-Snarks* vs *Zk-Starks* vs *Bulletproofs*

A tabela 3.1 mostra as comparações dos diferentes sistemas de prova em termos da complexidade dos seus algoritmos, tamanho de prova e segurança. Os dados foram obtidos após cruzamento de diferentes referências e o único dado que não bate certo é a complexidade do algoritmo *Prover* para as *Bulletproofs*. Segundo Benedikt Bünz, a complexidade é linear, ou seja, $\mathcal{O}(N)$. Uma possível explicação para tal, poderá ser devido à comparação de Benedikt ser em relação ás *range proofs* e não às provas com circuitos aritméticos. As provas *Zk-Snarks* e *Zk-Starks* são construídas com circuitos aritméticos e, por esta razão, não é trivial construir *range proofs* nestes sistemas. As *Bulletproofs* também podem ser construídas com circuitos aritméticos.

Proof System	Σ-*Protocols*	*Zk-Snarks*	*Zk-Starks*	*Bulletproofs*
Proof Size	$\mathcal{O}(N)$	$\sim \mathcal{O}(1)$	$\mathcal{O}(\log^2(N))$	$\mathcal{O}(log(N))$
Prover	$\mathcal{O}(N)$	$\mathcal{O}(N \cdot \log(N))$	$\mathcal{O}(N \cdot polylog(N))$	$\mathcal{O}(N \cdot \log(N))$
Verifier	$\mathcal{O}(N)$	$\sim \mathcal{O}(1)$	$\mathcal{O}(polylog(N))$	$\mathcal{O}(N \cdot \log(N))$
Trusted Setup	Não	Sim	Não	Não
Post-quantum secure	Não	Não	Sim	Não
Crypto assumptions	DL[1]	KOE[2]	CRHF[3]	DL

[1] Discret log [2] Knowledge-of-exponent [3] Collision-resistant hash functions

Tabela 3.1: Comparação dos diferentes tipos de sistemas de prova. [Glu18] [MBKM19] [Bü18]

Segundo Zooko Wilcox's, o fundador da *Zcash*, se as *Bulletproofs* ou o *Zk-Starks* fossem implementados na sua *blockchain*, os resultados seriam os mostrados na figura 3.10. Ao comparar o *Zk-Snarks* com *Zk-Starks*, este último é mais seguro, os tempos de computação da prova e verificação são semelhantes e aceitáveis para *blockchains* e o único contra é o tamanho de prova, o que torna impraticável o seu uso em *blockchains*. Em relação aos tempos apresentados para as *Bulletproofs*, pensamos que os valores verificados são demasiado elevados, porque o fundador da *Zcash* terá estimado o uso das *Bulleproofs* em *shielded transactions* com circuitos aritméticos. Não é uma comparação justa, porque as *Bulletproofs* foram desenhadas para permitir *range proofs* mais eficientes e, por esta razão, devem ser usadas em *confidential transactions*. O sistema de prova *Aurora* é uma evolução do *Zk-Snarks*. Não necessita de um *trusted setup* e presume-se ser seguro contra computação quântica. Mas tal como o *Zk-Starks*, o tamanho das suas provas são demasiado elevadas [BSCR+18].

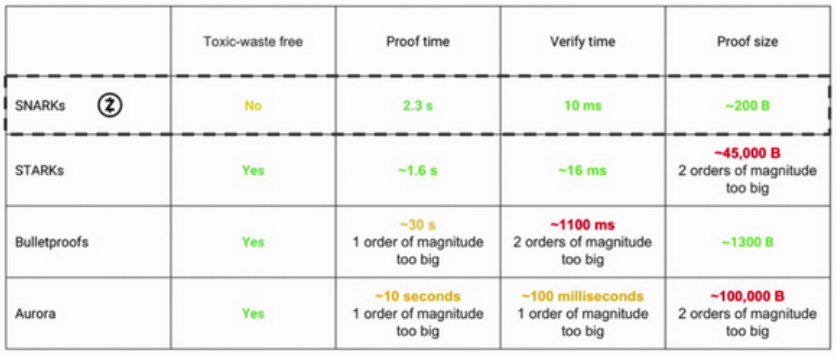

Figura 3.10: Comparação dos diferentes tipos de sistemas de prova, em termos de tempo de execução e tamanho de prova. [Wil18]

3.3 *Confidential Transactions*

Uma *confidential transaction* consiste em cifrar os valores transacionados através de *homomorphic encryption*, desta forma é possível manter algumas propriedades algébricas, como a soma, e realizar cálculos sobre os valores cifrados. A ferramenta básica para isso é um *Pedersen commitment*. Um *commitment* permite manter os dados secretos, mas obriga a ficar comprometido com eles para que não seja possível alterá-los mais tarde. Um simples exemplo é escrever um número no papel e guardá-lo numa caixa. Desta maneira ninguém consegue adivinhar o número que está dentro dela, mas se quiser-se mostrar esse número a alguém, esta apenas se pode abrir para o número que foi escrito.

$$Commitment = SHA\_256(blinding\_factor||dados) \quad (3.1)$$

O *blinding factor* é um número aleatório usado apenas para impedir que alguém tente adivinhar os dados. Por exemplo, se os dados fossem números de um intervalo curto, era fácil adivinhar e comparar o palpite com o *commitment*. Desta forma, se alguém revelar um *commitment*, ninguém consegue determinar os dados para os quais essa pessoa se comprometeu, mas mais tarde, se decidir revelar os dados e o *blinding factor*, qualquer um pode verificar que de facto se comprometeu com esses dados. *Pedersen commitments* sobre curvas elípticas têm a seguinte forma:

$$commit(v, r) = vG + rH \quad (3.2)$$

Onde G e H são pontos da curva, r é o *blinding factor* e v é o valor com o qual alguém se está a comprometer. De notar que v e r são valores secretos, e apenas conhecendo o *commitment*, G e H, é difícil descobrir v. O ponto H deve ser escolhido de maneira a que ninguém conheça o logaritmo discreto de H com respeito a G, ou seja, ninguém deve conhecer o x, para o qual $xG = H$. Assim, H deve ser escolhido com a ajuda de uma função *hash* criptográfica:

$$H = to\_point(SHA\_256(ENCODE(G))) \quad (3.3)$$

Os *Pedersen commitments* mantêm as propriedades algébricas da soma. Assim a soma de um conjunto de *commitments* é igual a um *commitment* com a soma dos valores e a soma dos *blinding factors*:

$$\begin{aligned} C(v_1, r_1) + C(v_2, r_2) &= v_1G + r_1H + v_2G + r_2 \\ &= (v_1 + v_2)G + (r_1 + r_2H) \quad (3.4) \\ &= C(v_1 + v_2, r_1 + r_2) \end{aligned}$$

$$C(v_1, r_1) - C(v_1, r_1) = 0 \quad (3.5)$$

Se $v_n = \{5, 3, 2\}$ e $r_n = \{20, 15, 5\}$ então:

$$C(v_1, r_1) - C(v_2, r_2) - C(v_3, r_3) = 0 \quad (3.6)$$

A segurança do esquema depende do problema do logaritmo discreto e da escolha do *blinding factor*, que deve ser aleatória.

Através dos *Pedersen commitments* é possível substituir os valores transacionados por um *commitment*. A comunidade, mesmo não sabendo a quantidade dos valores transacionados, pode verificar a transação ao conferir se a soma dos *commitments* como *output* é igual à soma dos *commitments* como *inputs*. Isto significa que, a soma dos valores como *outputs* é igual à soma

dos valores como *inputs*. De notar que a soma dos *bliding factors* como *outputs* deve também ser igual à soma dos *bliding factors* como *inputs*. Por exemplo, uma transacção com um 1 *input* e 2 outputs, pode ser verificada da seguinte maneira:

$$C(in_1, r_1) - (C(out_1, r_2) + C(out_2, r_3) + fee * H) = 0 \quad \textbf{(3.7)}$$

Onde $in_1 = out_1 + out_2 + fee$ e $r_1 = r_2 + r_3$. A taxa de transação, *fee*, é o único valor explicito na transação.

Apesar de já ser possível verificar o balanço entre os *inputs* e os *outputs*, o esquema é inseguro. Os *Pedersen commitments* assentam sobre grupos cíclicos. Isto significa que o grupo tem uma ordem e que a soma de dois valores é $mod\ P$. Uma adição de grandes números pode causar um *overflow* e comportar-se como valores negativos. Isto significa que as operações de adição podem conter valores negativos para dar zero. Por exemplo, poderíamos criar uma transação válida com os valores $\{2, 3, 5\}$ como *inputs*, e $\{-10, 5, 15\}$ como *outputs*:

$$(2 + 3 + 5) - (-10 + 5 + 15) = 0 \quad \textbf{(3.8)}$$

A equação é verdadeira mas um dos valores de *output* é negativo e, a soma dos outros é maior que a soma dos valores de *input*. Desta forma criou-se dinheiro do nada e ninguém se irá aperceber, pois é desprezado o valor negativo e os valores deixam de estar visíveis. A transação pode ser interpretada como alguém que gastou 10 moedas e o seu receptor recebeu uma moeda de -10, que descarta, mais uma de 5 e outra de 15. Ou seja, se a transação for feita entre endereços da mesma carteira, alguém com 10 moedas conseguiu falsificar 10 moedas, ficando agora com 20 moedas para gastar. Este problema gera inflação que ninguém poderá prever, podendo causar a perca de valor de mercado da moeda.

Para resolver o problema, podia-se enviar os valores e os *blinding factors*. Assim, a comunidade já os podia verificar mas, desta forma, voltava a perder-se a privacidade nas transações. Em vez disso, é necessário provar que o valor com o qual alguém se comprometeu (*commited value*) está dentro de um intervalo, e.g. $[0, 2^{64})$, mas sem revelar mais nada acerca do valor. Para isso é necessário uma *NIZK*, mais conhecida como *range proof*, ou em português prova de intervalo. Através de um *commitment* de um valor, é possível provar que este está dentro de um intervalo sem revelar nada acerca do valor. Então, para verificar uma transação, é necessário verificar se a soma dos *commitments* como *inputs* é igual à soma dos *commitments* como *outputs* e, caso, contenha mais do que um *output*, é preciso fazer uma *range proof* para cada um deles.

3.4 Conclusão

Todas as pessoas têm direito à sua privacidade, seja nas suas conversas na internet, como nas redes sociais e até mesmo nas suas transações numa *blockchain*. Mas tudo o que é positivo tem os seus aspectos negativos: a privacidade vai incentivar ainda mais os criminosos a usarem as criptomoedas. A *Bitcoin* é um dos meios de pagamento mais utilizados na *dark web* e a sua privacidade ainda é fraca. Esta pode ser usada para lavagens de dinheiro, como forma de pagamento de actos ilícitos, e o pior de tudo, pode ser usada para financiar organizações terroristas. As

técnicas para obtenção de privacidade devem ser implementadas mas devem continuar a haver formas de controlar o uso das criptomoedas como forma de financiamento para fins ilícitos.

Qualquer *blockchain* que implemente estas técnicas para obtenção de privacidade, perde a capacidade de deteção de possíveis *bugs*, os quais podem permitir o roubo de carteiras ou falsificação de criptomoedas. Como os dados mais sensíveis de uma transação deixam de estar expostos, torna impossível a detetação destes *bugs* através das transações, a menos que alguém o descubra e proponha uma alteração no código. O risco de ter inflações indetetáveis é um risco para qualquer ciptomoeda, pois pode levar a perdas significativas no seu valor de mercado.

O capítulo abordou as principais técnicas para a obtenção de privacidade, mas existem muitas mais e irão continuar a surgir novas, já que existe um grande interesse de investigação nesta área. O objectivo da investigação é melhorar as técnicas já existentes e com isso aumentar a escalabilidade das *blockchains*, ao fornecer formas mais rápidas de verificar transações e diminuir o seu tamanho, mantendo a privacidade e segurança dos seus utilizadores. A Visa consegue processar milhares de transações por segundo, em comparação com as *blockchains* mais conhecidas onde os valores, em média, ficam abaixo das centenas, o que ainda é uma diferença considerável.

Capítulo 4

Implementação de *Range Proofs* com *Bulletproofs*

4.1 Introdução

Como foi referido anteriormente, uma *range proof* permite verificar se um *commitment* representa um valor dentro de um intervalo específico, sem revelar nada acerca do valor. Por exemplo, uma *range proof* pode ser usada para validar se a idade de alguém está entre os 28 e 52 anos, sem revelar a idade exata da pessoa. As *confidential transactions*, onde os valores transacionados são escondidos, possibilitam uma forma de verificar se os valores não são negativos e não excedem um determinado valor, e.g. $[0, 2^{64})$. O capítulo abordará como se podem construir *range proofs* através das *Bulletproofs*, quais as etapas e equações envolvidas no processo, e como se pode verificar uma prova. Este capítulo foi elaborada através da consulta do *paper* das *Bulletproofs* [BBB$^+$18] e da documentação de uma implementação escrita em *Rust* [Cha18].

Foi escolhida a linguagem de programação funcional *Ocaml*, pois esta implementação tem em vista a sua integração numa plataforma *blockchain* escrita também em *Ocaml*, mais conhecida como *Tezos*.

4.2 Notação

- Seja $\mathbb{G}$ um grupo cíclico de ordem p, onde p é um número primo, e $\mathbb{Z}_p$ o anel de inteiros módulo p.
- Seja $\mathbb{G}^n$ e $\mathbb{Z}_p^n$ espaços vetoriais de dimensão n sobre $\mathbb{G}$ e $\mathbb{Z}_p$ respetivamente.
- Seja $\mathbb{Z}_p^*$ igual a $\mathbb{Z}_p \backslash \{0\}$.
- Escalares em $\mathbb{Z}_p$ são denotados pelas letras minúsculas $a, b, c \in \mathbb{Z}_p$.
- Geradores de $\mathbb{G}$ são denotados pelas letras maiúsculas $G, H, P, Q \in \mathbb{G}$.
- Os desafios são denotados pelas letras $x, y, z, w \in \mathbb{Z}_p^*$ e $x \xleftarrow{\$} \mathbb{Z}_p^*$ denota a amostragem uniforme de um elemento de $\mathbb{Z}_p^*$.
- Vectores são denotados a negrito, e.g, $\mathbf{a} \in \mathbb{Z}_p^n$ com os elementos $a_1, ..., a_n \in \mathbb{Z}_p$ ou $\mathbf{G} \in \mathbb{G}^n$ com os elementos $G_1, ..., G_n \in \mathbb{G}$.
- A multiplicação de um escalar $c \in \mathbb{Z}_p$ por um vetor $\mathbf{a} \in \mathbb{Z}_p^n$ produz um vector $\mathbf{b} = c \cdot \mathbf{a}$ onde $b_i = c \cdot a_i \in \mathbb{Z}_p$.
- O produto interno de dois vetores é denotado por $\langle \_, \_ \rangle$. De notar que $\langle a, b \rangle = \sum_{i=0}^{n} a_i \cdot b_i \in \mathbb{Z}_p$, enquanto $\langle a, G \rangle = \sum_{i=0}^{n} a_i \cdot G_i \in \mathbb{G}$.
- A multiplicação de elementos de dois vetores é denotada por $\mathbf{a} \circ \mathbf{b} = (a_1 \cdot b_1, ..., a_n \cdot b_n) \in \mathbb{Z}_p^n$.

- Vetores de polinómios são denotados por $\mathbf{p}(x) = \sum_{i=0}^{d} \mathbf{p}_i \cdot x^i \in \mathbb{Z}_p^n [x]$ e o produto interno entre dois vetores de polinómios $\mathbf{l}(x), \mathbf{r}(x)$ por:

$$\langle \mathbf{l}(x), \mathbf{r}(x) \rangle = \sum_{i=0}^{d} \sum_{j=0}^{i} \langle \mathbf{l}_i, \mathbf{r}_j \rangle \cdot x^{i+j} \in \mathbb{Z}_p^n [x] \quad (4.1)$$

- Vetores com todos os elementos a 0 são denotados por **0** e com todos a 1 por **1**.
- Para um escalar $\mathbf{y} \in \mathbb{Z}_p^*$, y^n denota um vector com as primeiras n potências de y, i.e.:

$$\mathbf{y}^n = (1, y, y^2, ..., y^{n-1}) \in (\mathbb{Z}_p^*)^n \quad (4.2)$$

- Para vectores com tamanho par, $2k$, $\mathbf{a}_{lo}$ e $\mathbf{a}_{hi}$ denotam a primeira e a segunda metade do vector **a** respetivamente, i.e.:

$$\mathbf{a}_{lo} = (a_0, ..., a_{k-1}), \qquad \mathbf{a}_{hi} = (a_k, ..., a_{2k-1}) \quad (4.3)$$

- *Pedersen commitments* são denotados por:

$$Com(v) = Com(v, \tilde{v}) = v \cdot B + \tilde{v} \cdot \tilde{B} \quad (4.4)$$

  onde B e $\tilde{B}$ são geradores usados para o valor e para o *blinding factor*. *Blinding factors* são denotados por letras com um til.

- *Pedersen commitments* para vetores são denotados por:

$$Com(\mathbf{a}_L, \mathbf{a}_R) = Com(\mathbf{a}_L, \mathbf{a}_R, \tilde{a}) = \langle \mathbf{a}_L, \mathbf{G} \rangle + \langle \mathbf{a}_R, \mathbf{H} \rangle + \tilde{a} \cdot \tilde{B} \quad (4.5)$$

  onde **G** e **H** são vetores de geradores.

- Concatenação de dois vetores é denotada por $\mathbf{a}_L || \mathbf{a}_R$.

4.3 Gerador de números pseudo-aleatórios

Números aleatórios usados em computadores comuns não são verdadeiramente "aleatórios", sendo por isso chamados de números pseudo-aleatórios. O algoritmo deve gerar uma sequência de números que devem ser aproximadamente independentes uns dos outros. Como o número máximo de bits que um inteiro em *Ocaml* pode possuir é 64 bits, o algoritmo utiliza uma biblioteca *Ocaml* chamada de *Zarith*, a qual permite operações aritméticas e lógicas sobre inteiros de grandes dimensões (*bignum*). A biblioteca *Zarith* não contém nenhuma função para gerar números pseudo-aleatórios, então implementou-se um gerador, o qual pode ser visualizado no *listing* 4.1. Através de uma função auxiliar recursiva é gerado um bit em cada interação para construir uma *string* com o número de bits do valor recebido como argumento na função principal (*bound*). A *string* em binário gerada é então transformada no tipo *Zarith* através de um *cast*.

```
1 let rec random_big_int bound =
2   let rec aux i acc =
3     if i = 0 then
```

```
      acc
    else
      let b = Random.int 2 in
      aux (i-1) (acc ^ string_of_int b)
  in
  let () = Random.self_init () in
  let size =
    String.length (Z.format "%b" bound)
  in
  let num = Z.of_string (aux size "0b") in
  match num < bound with
  | true -> num
  | false -> random_big_int bound
```

Listing 4.1: Algoritmo do gerador de números pseudo-aleatórios.

4.4 *Eliptic Curve Criptography (ECC)*

Como os *Petersen commitments* e as *range proofs* assentam sobre o problema do logaritmo discreto, a implementação usa *ECC* (criptografia sobre curvas elípticas). Para tal usou-se uma biblioteca *Ocaml* com o nome *ECC-OCaml*, disponibilizada através do *github* [zn13]. No *listing* 4.2 pode-se visualizar o tipo dos ponto da curva e duas variáveis instanciadas com pontos da curva, sendo um deles o ponto que representa o infinito e, o outro, o ponto com as coordenadas $x = 9$ e $y = 5$.

```
type point = Infinity | Point of Z.t * Z.t
let p1 = Infinity
let p2 = Point(Z.of_int(9),Z.of_int(5))
```

Listing 4.2: Tipo point e duas variáveis do tipo point.

Foram realizados vários testes à biblioteca e descobriu-se um *bug* na função da soma de dois pontos. O caso onde dois pontos são iguais deve estar primeiro que o caso onde apenas a coordenada x dos dois pontos é igual. No *listing* 4.3 pode-se visualizar a ordem correta dos casos.

```
let add_point (r1 : point) (r2 : point) =
    ....
    match r1, r2 with
    | _, Infinity -> r1
    | Infinity, _ -> r2
    | r1, r2 when r1 = r2 -> double_point r1
    | Point (x1, _), Point (x2, _) when x1 = x2 -> Infinity
    | Point (x1, y1), Point (x2, y2) ->
        ...
```

Listing 4.3: Casos da função da soma de dois pontos.

Foram realizadas mais algumas alterações à biblioteca e criou-se um *pull request*, para que o seu autor e *mantainers*, tivessem em consideração algumas das alterações, sobretudo o *bug*

na adição de pontos. O módulo passou a ser um *functor*, o qual permite que se receba como parâmetro um módulo com as especificações de uma curva. Assim, um programador que queira utilizar a biblioteca pode recorrer à curva que desejar e apenas tem de conhecer as suas especificações. O *functor* apenas permite curvas com equações do tipo *Weierstrass*. Foi adicionado também um novo ficheiro, *Secp256k1.ml*, onde consta um módulo com a curva *Secp256k1* de tamanho 256 bits. Para a multiplicação de um ponto P por um escalar k, a biblioteca utiliza o algoritmo *Montgomery ladder*. Mas como as *range proofs* fazem bastante uso da multiplicação de pontos, foi implementado o algoritmo *wNAF* para uma melhor performance. Este algoritmo utiliza pontos pré-computados $\{1, 3, 5, ..., 2^{w-1} - 1\}P$, onde P é o ponto a ser multiplicado e w o tamanho da janela (*windows size*), e.g., para $w = 4$ é necessário pré-computar os pontos $\{1, 3, 5, 7\}P$. Primeiro utiliza-se o algoritmo da figura 4.1 para se obter a representação wNAF de um escalar k, e.g., para $w = 4$ e $k = 10$, a representação *wNAF* de k é $\{5, 0\}$. De seguida utiliza-se a representação para se obter a multiplicação do ponto pelo escalar ($k \cdot P$) através do algoritmo da figura 4.2. Para a representação anterior, a multiplicação é obtida apenas com um *double point*, i.e., $2 \cdot 5P$. O objetivo do algoritmo é reduzir o número de somas, porque esta é mais dispendiosa que um *double point*.

Input: k
Output: width-w NAF of k
A. $i = 0$.
B. while k ≥ 1 do
  if k is odd
    $k_i = u$, where $u \equiv k (mod 2^w)$
    $k = k - k_i$
  else $k_i = 0$.
C. $k = k/2$
D. $i = i + 1$
Return $\{k_{i-1}, k_{i-2}, \cdots, k_1, k_0\}$

Figura 4.1: Algoritmo para computar o *wNAF* de um escalar [Kod14].

Input: w, width-w NAF of k, P
Output: Q = kP
A. Compute $P_i = iP$ for $i = 1, 3, \cdots, 2^w - 1$.
B. $Q = \phi$.
C. for i = t-1 downto 0 do
  $Q = 2Q$.
  if $k_i \neq 0$ then
    if $k_i > 0$ then $Q = Q + P_{k_i}$.
    else $Q = Q - P_{k_i}$.
Return Q.

Figura 4.2: Algoritmo da multiplicação de pontos *wNAF* [Kod14].

4.5 *Rangeproofs*

O *prover* deseja convencer um *verifier* de que um determinado valor v está dentro de um intervalo, sem revelar nenhuma informação sobre v:

$$v \in [0, 2^n) \quad \textbf{(4.6)}$$

O valor v pode ser representado pelo produto interno do seu vetor de bits com o vetor das potências de dois:

$$\begin{aligned} v &= \langle \mathbf{a}, \mathbf{2}^n \rangle \\ &= a_0 \cdot 2^0 + ... + a_{n-1} \cdot 2^{n-1} \end{aligned} \quad \textbf{(4.7)}$$

onde **a** denota o vector de bits de v.

Se o vector **a** é apenas constituído por $\{0, 1\}$ então a seguinte equação tem de ser válida:

$$\mathbf{a} \circ (\mathbf{a} - \mathbf{1}) = \mathbf{0} \quad (4.8)$$

A multiplicação dos dois vetores resulta num vetor de zeros apenas se o vetor $\mathbf{a}$ for constituído por bits. Se $\mathbf{a}_L = \mathbf{a}$ e $\mathbf{a}_R = \mathbf{a} - \mathbf{1}$ obtêm-se as seguintes equações:

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \quad (4.9)$$
$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0} \quad (4.10)$$
$$(\mathbf{a}_L - \mathbf{1}) - \mathbf{a}_R = \mathbf{0} \quad (4.11)$$

As equações 4.10 e 4.11 podem ser transformadas num produto interno. Se $\mathbf{b} = \mathbf{0}$ então $\langle \mathbf{b}, \mathbf{y}^n \rangle = 0$ para qualquer y, as equações em cima podem ser reescritas da seguinte forma:

$$\langle \mathbf{a}_L, \mathbf{2}^n \rangle = v \quad (4.12)$$
$$\langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle = 0 \quad (4.13)$$
$$\langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle = 0 \quad (4.14)$$

onde y representa um desafio escolhido pelo *verifier*.

As três equações podem agora ser combinadas com o desafio z escolhido pelo *verifier*:

$$z^2 v = z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L - \mathbf{1} - \mathbf{a}_R, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \quad (4.15)$$

O próximo passo é combinar os termos da equação num único produto interno, de maneira a que os termos $\mathbf{a}_L$ apareçam apenas do lado esquerdo e os termos $\mathbf{a}_R$ apareçam apenas no lado direito.

$$\begin{aligned} z^2 v &= z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle - z \langle \mathbf{a}_R, \mathbf{y}^n \rangle - z \langle \mathbf{1}, \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\ z^2 v + z \langle \mathbf{1}, \mathbf{y}^n \rangle &= z^2 \langle \mathbf{a}_L, \mathbf{2}^n \rangle + z \langle \mathbf{a}_L, \mathbf{y}^n \rangle - z \langle \mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\ \shortparallel \qquad &= \langle \mathbf{a}_L, z^2 \mathbf{2}^n \rangle + \langle \mathbf{a}_L, z\mathbf{y}^n \rangle + \langle -z\mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle \mathbf{a}_L, \mathbf{a}_R \circ \mathbf{y}^n \rangle \\ \shortparallel \qquad &= \langle \mathbf{a}_L, z^2 \mathbf{2}^n + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle + \langle -z\mathbf{1}, \mathbf{a}_R \circ \mathbf{y}^n \rangle \end{aligned} \quad (4.16)$$

Para combinar os termos no lado direito adiciona-se $\langle -z\mathbf{1}, z^2 \mathbf{2}^n + z\mathbf{y}^n \rangle$ aos dois lados:

$$\begin{aligned} z^2 v + z \langle \mathbf{1}, \mathbf{y}^n \rangle - \langle z\mathbf{1}, z^2 \mathbf{2}^n + z\mathbf{y}^n \rangle &= \langle \mathbf{a}_L, z^2 \mathbf{2}^n + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \\ &\quad + \langle -z\mathbf{1}, z^2 \mathbf{2}^n + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \\ z^2 v + (z - z^2) \langle \mathbf{1}, \mathbf{y}^n \rangle - z^3 \langle \mathbf{1}, \mathbf{2}^n \rangle &= \langle \mathbf{a}_L - z\mathbf{1}, z^2 \mathbf{2}^n + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n \rangle \end{aligned} \quad (4.17)$$

Se $\delta(y, z)$ representar todos os termos não secretos fora do produto interno, finalmente obtém-se:

$$z^2 v + \delta(y,z) = \left\langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^2\mathbf{2}^n \right\rangle \quad (4.18)$$

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

O Prover ainda não pode enviar o produto interno ao *verifier* porque este revela informação sobre v. O próximo passo é construir vetores de polinómios, os quais são construídos através dos vetores do produto interno anterior e dos vetores de *blinding factors* ($\mathbf{s}_L, \mathbf{s}_R$) gerados aleatoriamente:

$$\mathbf{s}_L, \mathbf{s}_R \xleftarrow{\$} \mathbb{Z}_p^n \quad (4.19)$$

$$\mathbf{l}(x) = \mathbf{l}_0 + \mathbf{l}_1 x = (\mathbf{a}_L + \mathbf{s}_L x) - z\mathbf{1} \in \mathbb{Z}_p^n[x] \quad (4.20)$$
$$\mathbf{r}(x) = \mathbf{r}_0 + \mathbf{r}_1 x = \mathbf{y}^n \circ ((\mathbf{a}_R + \mathbf{s}_R x) + z\mathbf{1}) + z^2\mathbf{2}^n \in \mathbb{Z}_p^n[x] \quad (4.21)$$

onde os termos de grau 0 dos polinómios são $\mathbf{l}_0 = \mathbf{a}_L - z\mathbf{1}$ e $\mathbf{r}_0 = z^2\mathbf{2}^n + z\mathbf{y}^n + \mathbf{a}_R \circ \mathbf{y}^n$ e os termos de grau 1 são $\mathbf{l}_1 = \mathbf{s}_L$ e $\mathbf{r}_1 = \mathbf{s}_R$.

O produto interno pode agora ser representado através dos vetores de polinómios, o qual resulta num polinómio de grau dois:

$$t(x) = \langle \mathbf{l}(x), \mathbf{r}(x) \rangle = t_0 + t_1 x + t_2 x^2 \quad (4.22)$$

onde os coeficientes do polinómio $t(x)$ podem ser calculados através do método de Karatsuba's:

$$t_0 = \langle \mathbf{l}_0, \mathbf{r}_0 \rangle = \left\langle \mathbf{a}_L - z\mathbf{1}, \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}) + z^2\mathbf{2}^n \right\rangle \quad (4.23)$$
$$t_2 = \langle \mathbf{l}_1, \mathbf{r}_1 \rangle \quad (4.24)$$
$$t_1 = \langle \mathbf{l}_0 + \mathbf{l}_1, \mathbf{r}_0 + \mathbf{r}_1 \rangle - t_0 - t_2 \quad (4.25)$$

O *prover* tem de convencer o *verifier* que t_0 é igual a $z^2 + \delta(y,z)$ e que $t(x)$ é o polinómio correto. Para provar que t_0 está correto, o *prover* calcula um *commitment* para $t(x)$ e convence o *verifier* que é um *commitment* para $t(x)$ ao avaliar o polinómio com um desafio x. O *prover* já calculou um *commitmet* para v ($V = Com(v)$) e então apenas necessita de calcular um *commitment* para t_1 ($T_1 = Com(t_1)$) e para t_2 ($T_2 = Com(t_2)$), os quais se podem relacionar através das seguintes equações:

$$
\begin{array}{rcrcrcrcr}
t(x)B & = & z^2 vB & + & \delta(y,z)B & + & xt_1B & + & x^2t_1B \\
+ & & + & & + & & + & & + \\
\tilde{t}(x)\tilde{B} & = & z^2\tilde{v}\tilde{B} & + & 0\tilde{B} & + & x\tilde{t_1}\tilde{B} & + & x^2\tilde{t}_2\tilde{B} \\
\shortparallel & & \shortparallel & & \shortparallel & & \shortparallel & & \shortparallel \\
 & = & z^2V & + & \delta(y,z)B & + & xT_1 & + & x^2T_2
\end{array}
$$

De notar que a soma de cada coluna é um *commitment* para a variável da primeira linha e a variável da segunda linha como seu *blinding factor*. A soma de todas as colunas é um *commitment* para $t(x)$, com $\tilde{t}(x)$ como seu *synthetic blinding factor*[1]. O *prover* envia $t(x)$ e $\tilde{t}(x)$ ao *verifier*, de modo a convencê-lo que $t(x) = z^2v + \delta(y,z) + t_1x + t_2x^2$ e para tal é só verificar a igualdade da última linha:

$$t(x)B + \tilde{t}(x)B = z^2V + \delta(y,z)B + xT_1 + x^2T_2 \quad \textbf{(4.26)}$$

Por último, para provar que $t(x)$ está correto, é necessário provar que os vetores de polinómios $\mathbf{l}(x)$ e $\mathbf{r}(x)$ foram construídos corretamente e que $t(x) = \langle \mathbf{l}(x), \mathbf{r}(x) \rangle$. Para tal, relaciona-se $\mathbf{l}(x)$ e $\mathbf{r}(x)$ com os *commitments* para $\mathbf{a}_L, \mathbf{a}_R, \mathbf{s}_L$ e $\mathbf{s}_R$. Uma vez que $\mathbf{r}(x) = \mathbf{r}_0 + \mathbf{r}_1 x = \mathbf{y}^n \circ ((\mathbf{a}_R + \mathbf{s}_R x) + z\mathbf{1}) + z^2\mathbf{2}^n$ era necessário os *commitments* para $\mathbf{y}^n \circ \mathbf{a}_R$ e $\mathbf{y}^n \circ \mathbf{s}_R$. Mas os *commitments* são construídos antes de o *prover* receber o desafio y. Então o *verifier* necessita de converter os *commitments* $Com(\mathbf{a}_L, \mathbf{a}_R)$ e $Com(\mathbf{s}_L, \mathbf{s}_R)$ para $Com(\mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R)$ e $Com(\mathbf{s}_L, \mathbf{y}^n \circ \mathbf{s}_R)$:

$$Com(\mathbf{a}_L, \mathbf{y}^n \circ \mathbf{a}_R) = \langle \mathbf{a}_L, \mathbf{G} \rangle + \left\langle \mathbf{y}^n \circ \mathbf{a}_R, \mathbf{H}' \right\rangle + \tilde{a}\tilde{B} \quad \textbf{(4.27)}$$

$$Com(\mathbf{s}_L, \mathbf{y}^n \circ \mathbf{s}_R) = \langle \mathbf{s}_L, \mathbf{G} \rangle + \left\langle \mathbf{y}^n \circ \mathbf{s}_R, \mathbf{H}' \right\rangle + \tilde{s}\tilde{B} \quad \textbf{(4.28)}$$

onde $\mathbf{H}' = \mathbf{y}^{-n} \circ \mathbf{H}$.

O *verifier* pode agora relacionar os *commitmets* do *prover*: $A = Com(\mathbf{a}_L, \mathbf{a}_R)$ e $S = Com(\mathbf{s}_L, \mathbf{s}_R)$ a $\mathbf{l}(x)$ e $\mathbf{r}(x)$ através das seguintes equações:

$$
\begin{array}{rcrcrcr}
\langle \mathbf{l}(x), \mathbf{G} \rangle & = & \langle \mathbf{a}_L, \mathbf{G} \rangle & + & x \langle \mathbf{s}_L, \mathbf{G} \rangle & + & \langle -z\mathbf{1}, \mathbf{G} \rangle \\
+ & & + & & + & & + \\
\left\langle \mathbf{r}(x), \mathbf{H}' \right\rangle & = & \langle \mathbf{a}_R, \mathbf{H} \rangle & + & x \langle \mathbf{s}_R, \mathbf{H} \rangle & + & \left\langle z\mathbf{y}^n + z^2\mathbf{2}^n, \mathbf{H}' \right\rangle \\
+ & & + & & + & & \\
\tilde{e}\tilde{B} & = & \tilde{a}\tilde{B} & + & x\tilde{s}\tilde{B} & & \\
\shortparallel & & \shortparallel & & \shortparallel & & \shortparallel \\
 & = & A & + & xS & + & \left\langle z\mathbf{y}^n + z^2\mathbf{2}^n, \mathbf{H}' \right\rangle - \langle z\mathbf{1}, \mathbf{G} \rangle
\end{array}
$$

Como no esquema de equações anterior, a soma de cada coluna é um *Pedersen Commitments* para vetores, com a parte esquerda e direita do vetor na primeira e segunda linha e os *blinding factors* na terceira. A soma de todas as colunas é um *Pedersen commitment* para os vetores

[1] *synthetic* porque é calculado através de *blinding factors* de outros *commitments*

$\mathbf{l}(x)$ e $\mathbf{r}(x)$ com o *synthetic blinding factor* $\tilde{e}$. O *prover* envia o $\tilde{e}$ ao *verifier* e este usa a última linha para calcular:

$$\begin{aligned} P &= -\tilde{e}\tilde{B} + A + xS + \left\langle z\mathbf{y}^n + z^2\mathbf{2}^n, \mathbf{H}' \right\rangle - \langle z\mathbf{1}, \mathbf{G} \rangle \\ &= -\tilde{e}\tilde{B} + A + xS + \left\langle z\mathbf{1} + z^2\mathbf{y}^{-n} \circ \mathbf{2}^n, \mathbf{H}' \right\rangle - \langle z\mathbf{1}, \mathbf{G} \rangle \end{aligned} \tag{4.29}$$

Falta apenas provar que $P = \langle \mathbf{l}(x), \mathbf{G} \rangle + \left\langle \mathbf{r}(x), \mathbf{H}' \right\rangle$ e que $t(x) = \langle \mathbf{l}(x), \mathbf{r}(x) \rangle$. Para tal o *prover* pode simplesmente enviar $\mathbf{l}(x)$ e $\mathbf{r}(x)$, e o *verifier* pode verificar diretamente. Mas isto resultaria na troca de $2n$ escalares entre o *prover* e o *verifier*. Em vez disso, usa-se o protocolo *inner-product argument* para reduzir a complexidade da comunicação e, por sua vez, o tamanho da prova de $\mathcal{O}(n)$ para $\mathcal{O}(log(n))$. O protocolo recebe como *input* $t(x)$ e P, e permite construir uma prova de conhecimento para a seguinte relação:

$$P = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle \wedge c = \langle \mathbf{a}, \mathbf{b} \rangle \tag{4.30}$$

onde $\mathbf{a}$ e $\mathbf{b}$ passam a denotar os vetores $\mathbf{l}(x)$ e $\mathbf{r}(x)$ e c o seu produto interno, i.e., passa a denotar $t(x)$.

Pode-se combinar as duas equações em apenas uma ao usar um desafio $w \xleftarrow{\$} \mathbb{Z}_p^*$ da seguinte forma:

$$P + cwB = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle + \langle \mathbf{a}, \mathbf{b} \rangle wB \tag{4.31}$$

Se $P' = P + cwB$ e $Q = wB$ a equação anterior pode ser reescrita da seguinte forma:

$$P' = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle + \langle \mathbf{a}, \mathbf{b} \rangle Q \tag{4.32}$$

A equação é útil uma vez que permite comprimir cada vetor ao meio e usá-la da mesma forma. Ao realizar esta compressão $\lg n$ vezes, vai acabar-se com uma equação onde ambos os vetores contêm apenas um elemento e pode-se simplesmente transmiti-los para verificar a equação final diretamente. Os vetores podem ser comprimidos com a adição da primeira metade do vetor com a segunda metade através do desafio μ_k, onde $k = \lg x$, da seguinte forma:

$$\mathbf{a}^{(k-1)} = \mathbf{a}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{a}_{hi} \tag{4.33}$$
$$\mathbf{b}^{(k-1)} = \mathbf{b}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{b}_{hi} \tag{4.34}$$
$$\mathbf{G}^{(k-1)} = \mathbf{G}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{G}_{hi} \tag{4.35}$$
$$\mathbf{H}^{(k-1)} = \mathbf{H}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{H}_{hi} \tag{4.36}$$

As potências de μ_k são escolhidas de maneira a simplificar os produtos internos de interesse, da seguinte forma:

Seja $P_k = P'$, pode-se usar a mesma equação para P_k para definir P_{k-1}, mas com os vetores comprimidos:

$$\begin{aligned} P_{k-1} &= \left\langle \mathbf{a}^{(k-1)}, \mathbf{G}^{(k-1)} \right\rangle + \left\langle \mathbf{b}^{(k-1)}, \mathbf{H}^{(k-1)} \right\rangle + \left\langle \mathbf{a}^{(k-1)}, \mathbf{b}^{(k-1)} \right\rangle \cdot Q \\ &= \left\langle \mathbf{a}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{a}_{hi}, \mathbf{G}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{G}_{hi} \right\rangle + \\ &\quad \left\langle \mathbf{b}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{b}_{hi}, \mathbf{H}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{H}_{hi} \right\rangle + \\ &\quad \left\langle \mathbf{a}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{a}_{hi}, \mathbf{b}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{b}_{hi} \right\rangle \cdot Q \end{aligned} \tag{4.37}$$

Simplificando em produtos internos mais simples:

$$P_{k-1} = \langle \mathbf{a}_{lo}, \mathbf{G}_{lo} \rangle + \langle \mathbf{a}_{hi}, \mathbf{G}_{hi} \rangle \quad + \mu_k^2 \langle \mathbf{a}_{lo}, \mathbf{G}_{hi} \rangle + \mu_k^{-2} \langle \mathbf{a}_{hi}, \mathbf{G}_{lo} \rangle + \tag{4.38}$$

$$\langle \mathbf{b}_{lo}, \mathbf{H}_{lo} \rangle + \langle \mathbf{b}_{hi}, \mathbf{H}_{hi} \rangle \quad + \mu_k^2 \langle \mathbf{b}_{hi}, \mathbf{H}_{lo} \rangle + \mu_k^{-2} \langle \mathbf{b}_{lo}, \mathbf{H}_{hi} \rangle + \tag{4.39}$$

$$\langle \mathbf{a}_{lo}, \mathbf{b}_{lo} \rangle + \langle \mathbf{a}_{hi}, \mathbf{b}_{hi} \rangle \cdot Q \; + (\mu_k^2 \langle \mathbf{a}_{lo}, \mathbf{b}_{hi} \rangle + \mu_k^{-2} \langle \mathbf{a}_{hi}, \mathbf{b}_{lo} \rangle) \cdot Q \tag{4.40}$$

Na equação em cima, as duas colunas mais à esquerda representam a definição de P_k. Agrupando todos os termos com μ_k^2 e todos com μ_k^{-2}, a equação anterior pode ser representada da seguinte forma:

$$P_{k-1} = P_k + \mu_k^2 \cdot L_k + \mu_k^{-2} \cdot R_k \tag{4.41}$$

$$L_k = \langle \mathbf{a}_{lo}, \mathbf{G}_{hi} \rangle + \langle \mathbf{b}_{hi}, \mathbf{H}_{lo} \rangle + \langle \mathbf{a}_{lo}, \mathbf{b}_{hi} \rangle \cdot Q \tag{4.42}$$

$$R_k = \langle \mathbf{a}_{hi}, \mathbf{G}_{lo} \rangle + \langle \mathbf{b}_{lo}, \mathbf{H}_{hi} \rangle + \langle \mathbf{a}_{hi}, \mathbf{b}_{lo} \rangle \cdot Q \tag{4.43}$$

Se o *prover* se comprometer primeiro com L_k e R_k antes de μ_k ser escolhido aleatoriamente, então a afirmação com os vetores comprimidos é verdadeira, sendo que a afirmação sobre os vetores não comprimidos também é verdadeira com grande probabilidade. Pode-se continuar a comprimir agora para a afirmação P_{k-2} com um novo desafio u_{k-1}, da mesma forma que foi usado o desafio u_k. Continua-se a comprimir até que os vetores contenham apenas um elemento (a_0, b_0, G_0, H_0) e P_0 pode ser calculado das seguintes formas:

$$P_0 = a_0 G_0 + b_0 H_0 + a_0 b_0 Q \tag{4.44}$$

$$P_0 = P_k + \sum_{i=1}^{k} (u_i^2 \cdot L_i + u_i^{-2} \cdot R_i) \tag{4.45}$$

Rescrevendo as definições de $P_k = P' = P + cwB$ e $Q = wB$, resulta na seguinte equação:

$$P + cwB = a_0 G_0 + b_0 H_0 + a_0 b_0 wB - \sum_{i=1}^{k} (u_i^2 \cdot L_i + u_i^{-2} \cdot R_i) \tag{4.46}$$

O protocolo tem $\lg n$ rondas de compressão onde o *prover* envia (L_i, R_i) em cada ronda $i = k...1$ e possui uma ronda adicional onde o *prover* envia (a_0, b_0) para que o *verifier* possa verificar a equação.

Estas são todas as equações por detrás de uma *range proof*. De seguida seram abordados os algoritmos por detrás da construção de uma prova e da sua verificação.

4.5.1 Algoritmo do *prover*

Uma *range proof* agregada, é uma *Zero-Knowledge Proof of Knowledge (ZKPK)* cujo o objetivo é mostrar que m valores estão dentro de um intervalo:

$$ZKPK\{v_0, .., v_{m-1} \in [0, 2^n)\} \quad (4.47)$$

onde m e n são ambos potências de dois. Se $m = 1$ a *range proof* é apenas para um único valor, enquanto se $m > 1$ a prova é para vários valores.

Como as provas são não-interativas, todos os desafios são gerados através da heurística de Fiat-Shamir e não escolhidos aleatoriamente pelo *verifier*. O algoritmo começa por calcular *commitments* para cada valor v_i, para os vetores de bits de cada valor $\mathbf{a}_{L_i}$ e $\mathbf{a}_{R_i} = \mathbf{a}_{L_i - 1}$ e para os vetores de *blinding factors* $\mathbf{s}_{L_i}$ e $\mathbf{s}_{R_i}$:

$$V_i = Com(v_i) = v_i \cdot B + \tilde{v}_i \cdot \tilde{B} \quad (4.48)$$

$$A_i = Com(\mathbf{a}_{L_i}, \mathbf{a}_{R_i}) = \langle \mathbf{a}_{L_i}, \mathbf{G} \rangle + \langle \mathbf{a}_{R_i}, \mathbf{H} \rangle + \tilde{a} \cdot \tilde{B} \quad (4.49)$$

$$S_i = Com(\mathbf{s}_{L_i}, \mathbf{s}_{R_i}) = \langle \mathbf{s}_{L_i}, \mathbf{G} \rangle + \langle \mathbf{s}_{R_i}, \mathbf{H} \rangle + \tilde{s} \cdot \tilde{B} \quad (4.50)$$

onde $\tilde{v}_i, \tilde{a}_i, \tilde{s}_i$ são escolhidos aleatoriamente de $\mathbb{Z}_p^*$ e $\mathbf{s}_L, \mathbf{s}_R$ de $(Z_p^*)^n$.

No *listing* 4.4 constam as funções que permitem calcular os *commitments*. A função "multiscalar_mul" utilizada no cálculo dos *commitments* permite multiplicar vários escalares com os respetivos pontos e somar todos os pontos resultantes da multiplicação.

```
(* Commitment of secret value v *)
let com_v =
  multiscalar_mul [value; blinding]
    [gens.b; gens.b_blinding]

(*Commitment of aL and aR*)
let compute_com_A v a_blinding gens =
  let rec aux i acc g h =
    let bit = Z.( (v asr i ) land one) in
    match g,h with
    | [],[] -> acc
    | hg::tg,hh::th ->
      if bit = Z.zero then
        aux (i+1) (add_point (inverse_point hh ) acc ) tg th
      else
        aux (i+1) (add_point hg acc ) tg th
    | _ ->
      raise (Invalid_argument "The two lists have different lengths")
  in
  add_point (aux 0 Infinity gens.vec_G gens.vec_H)
    (multiply_point gens.b_blinding a_blinding)
```

```
22
23 (* Commitment of sL and SR *)
24 let compute_com_S sL sR s_blinding gens =
25   multiscalar_mul ([s_blinding] @ sL @ sR)
26     ([gens.b_blinding] @ gens.vec_G @ gens.vec_H)
```

Listing 4.4: Funções para calcular os *commitments* para v, a_L, a_R, s_L, s_R.

De seguida é necessário somar todos os *commitments* A_i e S_i da seguinte forma:

$$A = \sum_{i=0}^{m-1} A_i \quad (4.51)$$

$$S = \sum_{i=0}^{m-1} S_i \quad (4.52)$$

O *prover* obtém os desafio $y, z \in \mathbb{Z}_p$ ao concatenar cada um dos *commitments* V_i com A e S e utiliza a concatenação como *input* para a função de *hash*. Como são dois desafios em apenas um passo, para obter o desafio z, calcula-se novamente o valor *hash* do desafio y.

$$y = SHA\_256(V_0||...||V_{m-1}||A||S) \quad (4.53)$$

$$z = SHA\_256(y) \quad (4.54)$$

No *listing* 4.5 pode-se visualizar o código da obtenção dos desafios y, z. A função "return_challenge" faz uso de uma biblioteca *Ocaml* com a função *hash* SHA_256.

```
1  (* Concatenation of all commitment of v, sent for each party *)
2  let s = List.fold_left (fun acc var ->
3    acc ^ point_to_string var.com_v) "" bit_com
4
5  (* Sum of all commitments A_i and S_i*)
6  let aA = List.fold_left (fun acc var ->
7    add_point acc var.com_a) Infinity bit_com
8
9  let sS = List.fold_left (fun acc var ->
10   add_point acc var.com_s) Infinity bit_com
11
12 (* Obtain bitchallenges y and z through Fiat_Shamir heuristic *)
13 let y =
14   let s = s ^ point_to_string aA ^ point_to_string sS in
15     return_challenge s
16
17 let z = return_challenge (Z.to_string y)
```

Listing 4.5: Como se podem obter os desafios y, z.

Com os desafios y, z, o *prover* pode construir os vetores de polinómios para cada um dos valores:

$$\mathbf{l}_i(x) = \mathbf{l}_{0_i} + \mathbf{l}_{1_i} x \tag{4.55}$$
$$\mathbf{r}_i(x) = \mathbf{r}_{0_i} + \mathbf{r}_{1_i} x \tag{4.56}$$
$$\mathbf{l}_{0_i} = \mathbf{a}_{L_i} - z\mathbf{1} \tag{4.57}$$
$$\mathbf{l}_{1_i} = \mathbf{s}_{L_i} \tag{4.58}$$
$$\mathbf{r}_{0_i} = \mathbf{y}^{n \cdot m}_{[i \cdot n:(i+1) \cdot n]} \circ (\mathbf{a}_{R_i} + z\mathbf{1}) + z^2 z^i \mathbf{2}^n \tag{4.59}$$
$$\mathbf{r}_{1_i} = \mathbf{y}^{n \cdot m}_{[i \cdot n:(i+1) \cdot n]} \circ \mathbf{s}_{R_i} \tag{4.60}$$

No *listing* 4.6 pode-se visualizar o tipo dos vetores de polinómios e o código para a sua construção:

```
type vec_poly1 = {
  mutable vec_poly1_0 : Z.t list;
  mutable vec_poly1_1 : Z.t list;
}

let compute_l_r_poly_vecs bit_com bit_chal l_p_vec r_p_vec =
  (* Contruction of the polynomial vectors l and r *)
  let offset_y = Z.pow bit_chal.y (bit_com.i * bit_chal.n) in
  let offset_z = Z.pow bit_chal.z bit_com.i in
  let zz = Z.(bit_chal.z * bit_chal.z) in
  let exp_y = ref offset_y in
  let exp_2 = ref Z.one in
  let i = ref 0 in
  List.iter2 (fun s_Li s_Ri ->
    let aL_i = Z.( (bit_com.v asr !i ) land one) in
    let aR_i = Z.(aL_i - Z.one) in
    l_p_vec.vec_poly1_0 <- l_p_vec.vec_poly1_0 @ [Z.(aL_i - bit_chal.z)];
    l_p_vec.vec_poly1_1 <- l_p_vec.vec_poly1_1 @ [s_Li];
    r_p_vec.vec_poly1_0 <- r_p_vec.vec_poly1_0 @ [Z.(!exp_y *
      (aR_i + bit_chal.z) + zz * offset_z * !exp_2)];
    r_p_vec.vec_poly1_1 <- r_p_vec.vec_poly1_1 @ [Z.(!exp_y * s_Ri)];
    exp_y := Z.(!exp_y * bit_chal.y);
    exp_2 := Z.(!exp_2 * ~$2);
    i := !i + 1;
  ) bit_com.s_L bit_com.s_R
```

Listing 4.6: Tipo dos vetores de polinómios e função para a sua contrução.

O produto interno dos vetores de polinómios produz um polinómio de grau 2 e o *prover* tem de calcular os seus coeficientes. Para isso usa o método de *Karatsuba*:

$$t_i(x) = \langle \mathbf{l}_i(x), \mathbf{r}_i(x) \rangle = t_{0_i} + t_{1_i} x + t_{2_i}^2 \quad (4.61)$$
$$t_{0_i} = \langle \mathbf{l}_{0_i}, \mathbf{r}_{0_i} \rangle \quad (4.62)$$
$$t_{2_i} = \langle \mathbf{l}_{1_i}, \mathbf{r}_{1_i} \rangle \quad (4.63)$$
$$t_{1_i} = \langle \mathbf{l}_{0_i} + \mathbf{l}_{1_i}, \mathbf{r}_{0_i} + \mathbf{r}_{1_i} \rangle - t_{0_i} - t_{2_i} \quad (4.64)$$

No *listing* 4.7 pode se visualizar o tipo dos polinómios de grau 2 e a função para o cálculo dos seus coeficientes:

```
type poly2 = {
  poly2_0 : Z.t;
  poly2_1 : Z.t;
  poly2_2 : Z.t;
}

let compute_t_poly l_p_vec r_p_vec =
  (* The inner_product of the above vector polynomials, using
    the Karatsuba's method *)
  let t0 = inner_product l_p_vec.vec_poly1_0 r_p_vec.vec_poly1_0 in
  let t2 = inner_product l_p_vec.vec_poly1_1 r_p_vec.vec_poly1_1 in
  let l0_plus_l1 = add_list l_p_vec.vec_poly1_0 l_p_vec.vec_poly1_1 in
  let r0_plus_r1 = add_list r_p_vec.vec_poly1_0 r_p_vec.vec_poly1_1 in
  let t1 = Z.((inner_product l0_plus_l1 r0_plus_r1) - t0 - t2) in
  {poly2_0 = t0 ; poly2_1 = t1; poly2_2 = t2 }
```

Listing 4.7: Tipo dos polinómios de grau 2 e função para calcular os seus coeficientes.

Depois de calcular os coeficientes de todos os polinómios de grau 2, o *prover* calcula *commitments* para cada t_{1_i} e t_{2_i}:

$$T_{1_i} = Com(t_{1_i}, \tilde{t}_{1_i}) = t_{1_i} \cdot B + \tilde{t}_{1_i} \cdot \tilde{B} \quad (4.65)$$
$$T_{2_i} = Com(t_{2_i}, \tilde{t}_{2_i}) = t_{2_i} \cdot B + \tilde{t}_{2_i} \cdot \tilde{B} \quad (4.66)$$

E calcula $T1$ e $T2$ ao somar cada um dos *commitments* calculados anteriormente. Obtém o desafio x ao concatenar T_1 e T_2, e usa esta como *input* para a função *hash*, tal como obteve o desafio y, z.:

$$T_1 = \sum_{j=0}^{m-1} T_{1_i} \quad (4.67)$$
$$T_2 = \sum_{j=0}^{m-1} T_{2_i} \quad (4.68)$$
$$x = SHA\_256(T_1 || T_2) \quad (4.69)$$

Após obter o desafio x, o *prover* usa o desafio para avaliar os polinómios $l_i(x), r_i(x), t_i(x)$:

$$\mathbf{l}_i = \mathbf{l}_{0_i} + \mathbf{l}_{1_i} x \quad (4.70)$$

$$\mathbf{r}_i = \mathbf{r}_{0_i} + \mathbf{r}_{1_i} x \quad (4.71)$$

$$t_i = t_{0_i} + t_{1_i} x + t_{2_i} x^2 \quad (4.72)$$

No *listing* 4.8 pode-se visualizar as funções que permitem avaliar os polinómios no ponto x.

```
let eval_vec_poly1 p1 x =
  List.map2 (fun a b ->  Z.(a + b * x) )
    p1.vec_poly1_0 p1.vec_poly1_1

let eval_poly2 p2 x =
  Z.(p2.poly2_0 + x * (p2.poly2_1 + x * p2.poly2_2))

(* evaluate the polynomial vectors l(x) and r(x) at the point x *)
let l_vec_x =
  eval_vec_poly1 poly_comm.l_poly poly_chal.x
let r_vec_x =
  eval_vec_poly1 poly_comm.r_poly poly_chal.x

(* evaluate the polynomial t(x) at the point x *)
let t_x =
  eval_poly2 poly_comm.t_poly poly_chal.x
```

Listing 4.8: Funções para avaliar os polinómios no ponto x.

De seguida o *prover* calcula os *synthetic blinding factors*:

$$\tilde{t}_i = z^2 + \tilde{t}_1 x + \tilde{t}_1 x^2 \quad (4.73)$$

$$\tilde{e}_i = \tilde{a}_i + \tilde{s}_i x \quad (4.74)$$

Para obter o desafio w, o *prover* soma cada um dos valores $t_i(x), \tilde{t}_i(x), \tilde{e}_i$ e obtém o desafio da mesma forma que obteve os outros desafios:

$$t = \sum_{i=0}^{m-1} t_i \quad (4.75)$$

$$\tilde{t} = \sum_{i=0}^{m-1} \tilde{t}_i \quad (4.76)$$

$$\tilde{e} = \sum_{i=0}^{m-1} \tilde{e}_i \quad (4.77)$$

$$w = SHA\_256(t||\tilde{t}||\tilde{e}) \quad (4.78)$$

Apenas falta executar o protocolo *inner product argument*. Para tal, o *prover* faz a concatenação de cada um dos vetores $\mathbf{l}_i$ e $\mathbf{r}_i$:

$$\mathbf{l} = \mathbf{l}_0||...||\mathbf{l}_{m-1} \quad (4.79)$$
$$\mathbf{r} = \mathbf{r}_0||...||\mathbf{r}_{m-1} \quad (4.80)$$

E executa o protocolo:

$$PK\{(\mathbf{G}, \mathbf{H} \in \mathbb{G}^{n \cdot m}; P', Q \in \mathbb{G}; \mathbf{a}; \mathbf{b} \in \mathbb{Z}_p^{n \cdot m}) : P' = \langle \mathbf{a}, \mathbf{G} \rangle + \langle \mathbf{b}, \mathbf{H} \rangle + \langle \mathbf{a}, \mathbf{b} \rangle Q\} \quad (4.81)$$

onde $\mathbf{a} = \mathbf{l}, \mathbf{b} = \mathbf{r}, \mathbf{Q} = wB$ e $\mathbf{H}' = y^{-m \cdot m}$.

O protocolo contém $k = \lg n \cdot m$ rondas, cada uma com o índice $i = k, .., 1$. O *prover* começa por calcular:

$$L_i = \langle \mathbf{a}_{lo}, \mathbf{G}_{hi} \rangle + \langle \mathbf{b}_{hi}, \mathbf{H}_{lo} \rangle + \langle \mathbf{a}_{lo}, \mathbf{b}_{hi} \rangle \cdot Q \quad (4.82)$$
$$R_i = \langle \mathbf{a}_{hi}, \mathbf{G}_{lo} \rangle + \langle \mathbf{b}_{lo}, \mathbf{H}_{hi} \rangle + \langle \mathbf{a}_{hi}, \mathbf{b}_{lo} \rangle \cdot Q \quad (4.83)$$

e usa a concatenação de L_i e R_i para obter o desafio μ_i:

$$\mu_i = SHA\_256(L_i||R_i) \quad (4.84)$$

De seguida o *prover* usa o desafio μ_i para comprimir os vetores:

$$\mathbf{a} = \mathbf{a}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{a}_{hi} \quad (4.85)$$
$$\mathbf{b} = \mathbf{b}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{b}_{hi} \quad (4.86)$$
$$\mathbf{G} = \mathbf{G}_{lo} \cdot \mu_k^{-1} + \mu_k \cdot \mathbf{G}_{hi} \quad (4.87)$$
$$\mathbf{H} = \mathbf{H}_{lo} \cdot \mu_k + \mu_k^{-1} \cdot \mathbf{H}_{hi} \quad (4.88)$$

e usa os vetores comprimidos na próxima ronda. O *prover* continua a comprimir os vetores até à ronda $i = 1$, onde os vetores possuem apenas um elemento. O resultado do protocolo são $2k$ pontos e 2 escalares, onde $k = \log_2(n \cdot m)$:

$$\{L_k, R_k, ..., L_1, R_1, a_0, b_0\} \quad (4.89)$$

No *listing* do apêndice B.1 pode-se visualizar a função para executar o protocolo *inner product argument*.

O *prover* publica a prova e esta consite em $m + 9 + 2k$ valores:

$$\{V_0, ..., V_{m-1}, A, S, T_1, T_2, t, \tilde{t}, \tilde{e}, L_k, R_k, ..., L_1, R_1, a_0, b_0\} \quad (4.90)$$

4.5.2 Algoritmo do *Verifier*

O *verifier* recebe como input a prova, ou seja, os $m + 9 + 2k$ valores, onde $k = \log_2(n \cdot m)$:

$$\{V_0, ..., V_{m-1}, A, S, T_1, T_2, t, \tilde{t}, \tilde{e}, L_k, R_k, ..., L_1, R_1, a_0, b_0\} \quad \textbf{(4.91)}$$

O primeiro passo é calcular os desafios através da heurística de *Fiat-Shamir*:

1. Obtém os desafios $y, z \in \mathbb{Z}_p^*$:

$$y = SHA\_256(V_0||...||V_{m-1}||A||S) \quad \textbf{(4.92)}$$
$$z = SHA\_256(y) \quad \textbf{(4.93)}$$

2. Obtém o desafio $x \in \mathbb{Z}_p^*$:

$$x = SHA\_256(T_1||T_2) \quad \textbf{(4.94)}$$

3. Obtém o desafio $w \in \mathbb{Z}_p^*$:

$$w = SHA\_256(t||\tilde{t}||\tilde{e}) \quad \textbf{(4.95)}$$

O próximo passo é executar o protocolo *inner product argument* para obter os desafios $\mu_k, ..., \mu 1 \in \mathbb{Z}_p^*$. O *verifier* obtém os desafios ao calcular:

$$\mu_k = SHA\_256(L_k||R_k) \quad \textbf{(4.96)}$$
$$\mu_{k-1} = SHA\_256(L_{k-1}||R_{k-1}) \quad \textbf{(4.97)}$$
$$... \quad \textbf{(4.98)}$$
$$\mu_1 = SHA\_256(L_1||R_1) \quad \textbf{(4.99)}$$

Para além dos desafios, o *verifier* necessita também dos vetores com apenas um elemento G_0 e H_0. Em vez de os comprimir passo a passo como o *prover*, o *verifier* pode-os calcular diretamente através dos desafios $\mu_k, ..., \mu_1 \in \mathbb{Z}_p^*$:

Seja $\mathbf{G}^i$ o vetor comprimido de $\mathbf{G}$ na ronda i, e G_j o elemento na posição j do vetor inicial $\mathbf{G} = (G_0, ..., G_{n-1})$. A próxima compressão do vetor $\mathbf{G}^i$ é:

$$\mathbf{G}^{i-1} = \mathbf{G}_{lo}^i \cdot \mu_i^{-1} + \mu_i \cdot \mathbf{G}_{hi}^i \quad \textbf{(4.100)}$$

Então o coeficiente de G_j no vetor com apenas um elemento G_0 é:

$$s_j = \mu_k^{b(j,k)} \cdot \mu_{k-1}^{b(j,k-1)} ... \mu_1^{b(j,1)} \quad \textbf{(4.101)}$$

onde $b(j, i)$ é igual a -1 ou 1, consoante a posição de G_j, ou seja, se este aparece na primeira metade ou na segunda de $\mathbf{G}^i$. Uma vez que, G_j aparece na posição (j mod 2^i) de $\mathbf{G}^i$, $b(j, i)$ pode ser representado através do seguinte sistema:

$$\begin{cases} -1 & \text{se } (j \bmod 2^i) < 2^{i-1} \\ 1 & \text{se } (j \bmod 2^i) \geqslant 2^{i-1} \end{cases} \quad (4.102)$$

ou:

$$\begin{cases} -1 & \text{se o bit (i-1) de } j = 0 \\ 1 & \text{se o bit (i-1) de } j = 0 \end{cases} \quad (4.103)$$

Então $G_0 = \langle \mathbf{s}, \mathbf{G} \rangle$, onde $\mathbf{s} = (s_0, ..., s_{n-1})$. Como H_0 é calculado de forma similar, apenas $\mathbf{H}_{lo}$ e $\mathbf{H}_{hi}$ estão numa ordem reversa. $H_0 = \langle 1/\mathbf{s}, \mathbf{G} \rangle$, onde $1/\mathbf{s}$ representa o vetor $\mathbf{s}$ na ordem reversa.

O protocolo fornece ao *verifier* uma lista com os seguintes escalares:

$$\{\mu_1^2, ..., \mu_k^2, \mu_1^{-2}, ..., \mu_k^{-2}, ..., s_0, ..., s_{n-1}\} \quad (4.104)$$

No *listing* do apêndice B.1 pode-se visualizar a função para executar o protocolo *inner product argument* para a obtenção dos escalares em cima.

Depois de obter todos os desafios, o objetivo do *verifier* é verificar as duas seguintes equações:

1. Verifica o coeficiente de grau zero do polinómio $t(x)$:

$$\begin{aligned} t(x)B + \tilde{t}(x)\tilde{B} &= \sum_{i=0}^{m-1} z^2 z^i V_i + \delta(y,z)B + xT1 + x^2 T_2 \\ 0 &= \sum_{i=0}^{m-1} z^2 z^i V_i + \delta(y,z)B + xT1 + x^2 T_2 - t(x)B - \tilde{t}(x)\tilde{B} \end{aligned} \quad (4.105)$$

   com $\delta(y,z) = (z - z^2) \cdot \langle \mathbf{1}, \mathbf{y}^{n \cdot m} \rangle - \sum_{i=0}^{m-1} z^{i+3} \cdot \langle \mathbf{1}, \mathbf{2}^{n \cdot m} \rangle$.

2. Verifica se o produto interno dos vetores $\mathbf{l}(x)$ e $\mathbf{r}(x)$ é igual a $t(x)$:

$$\begin{aligned} P + t(x)wB &= \langle a_0 \cdot \mathbf{s}, \mathbf{G} \rangle + \left\langle \mathbf{y}^{-n \cdot m} \circ (b_0 / \mathbf{s}), \mathbf{H} \right\rangle + a_0 b_0 wB - \sum_{i=0}^{k} L_i \mu_i^2 + \mu_i^{-2} R_i \\ 0 &= P + t(x)wB - \langle a_0 \cdot \mathbf{s}, \mathbf{G} \rangle - \left\langle \mathbf{y}^{-n \cdot m} \circ (b_0 / \mathbf{s}), \mathbf{H} \right\rangle - a_0 b_0 wB + \sum_{i=0}^{k} L_i \mu_i^2 + \mu_i^{-2} R_i \end{aligned} \quad (4.106)$$

   onde P é igual:

$$\begin{aligned} P &= -\tilde{e}\tilde{B} + A + xS - z \langle \mathbf{1}, \mathbf{G} \rangle + z \left\langle \mathbf{y}^{n \cdot m}, \mathbf{H}' \right\rangle + \sum_{i=0}^{m-1} \left\langle z^{i+2} \cdot \mathbf{2}^n, \mathbf{H}'_{[i \cdot n:(i+1) \cdot n]} \right\rangle \\ &= -\tilde{e}\tilde{B} + A + xS - z \langle \mathbf{1}, \mathbf{G} \rangle + z \langle \mathbf{1}, \mathbf{H} \rangle + \sum_{i=0}^{m-1} \left\langle z^{i+2} \cdot \mathbf{2}^n \circ \mathbf{y}^{-n \cdot m}, \mathbf{H}_{[i \cdot n:(i+1) \cdot n]} \right\rangle \\ &= -\tilde{e}\tilde{B} + A + xS - z \langle \mathbf{1}, \mathbf{G} \rangle + z \langle \mathbf{1}, \mathbf{H} \rangle + \left\langle \mathbf{y}^{-n \cdot m} \circ z^2 (z^0 \mathbf{2}^n || z^1 \mathbf{2}^n || ... || z^{m-1} \mathbf{2}^n), \mathbf{H} \right\rangle \end{aligned} \quad (4.107)$$

O *verifier* pode juntar as duas equações ao realizar a sua adição e multiplicando a primeira por um escalar escolhido aleatoriamente $c \xleftarrow{\$} \mathbb{Z}_p^*$. Assim pode realizar a verificação apenas com uma multiplicação multi-escalar, ao agrupar todos os escalares e pontos:

$$
\begin{aligned}
0 = & \ 1 \cdot A \\
& + x \cdot S \\
& + cz^2 \cdot V_0 + cz^3 \cdot V_1 + ... + cz^{m+1} \cdot V_{m-1} \\
& + cx \cdot T_1 \\
& + cx^2 \cdot T_2 \\
& + (w(t(x) - a_0 b_0) + c(\delta(y,z) - t(x))) \cdot B \\
& + (-\tilde{e} - c\tilde{t}(x)) \cdot \tilde{B} \\
& + \langle -z\mathbf{1} - a_0 \mathbf{s}, \mathbf{G} \rangle \\
& + \left\langle z\mathbf{1} + \mathbf{y}^{-n \cdot m} \circ z^2 (z^0 \mathbf{2}^n || z^1 \mathbf{2}^n || ... || z^{m-1} \mathbf{2}^n) - b_0 / \mathbf{s}, \mathbf{H} \right\rangle \\
& + \left\langle [\mu_1^2, ..., \mu_k^2], [L_1, ..., L_k] \right\rangle \\
& + \left\langle [\mu_1^{-2}, ..., \mu_k^{-2}], [R_1, ..., R_k] \right\rangle
\end{aligned}
\tag{4.108}
$$

Todo o código Ocaml para a verificação de uma *rangeproof* pode ser visualizado no *listing* do apêndice B.2.

4.6 Conclusão

Neste capítulo foi abordado todas as equações e etapas que permitem a construção de uma *range proof* e sua verificação. No apêndice A pode-se visualizar uma simples *rangeproof* com o intervalo $[0, 2^4)$ para o valor 5, na qual é utilizada o grupo cíclico $\mathbb{Z}_{13}^*$. A implementação *Ocaml* encontra-se num repositório no *gitlab* [PdS19] e qualquer programador a pode utilizar. Para tal basta seguir os passos da sua instalação e respectivos exemplos para a sua utilização.

Capítulo 5

Resultados e Testes

5.1 Introdução

Todo o *software* necessita de ser testado antes de ser publicado para que se encontrem eventuais *bugs* introduzidos durante a fase de implementação. Testes de carga também são importantes para se perceber o quão eficiente se encontra o nosso código em comparação com outras implementações ou com eventuais expectativas. Neste capítulo irão constar todos os testes realizados à implementação para se descortinarem eventuais *bugs*, bem como os resultados dos testes de carga realizados à implementação. Estes serão comparados com a implementação já publicada em Rust.

5.2 Testes

5.2.1 Verificação dos *inputs*

Sempre que os *inputs* das funções de computação e verificação de uma *range proof* não cumpram os requisitos, deve ser lançada uma excepção com um aviso ao utilizador, para que este conheça os motivos de não ter conseguido computar ou verificar a prova. Os requisitos dos *inputs* são os seguintes:

- O número de valores que se quer provar que pertencem ao intervalo deve ser igual ao número de *blindings factors*. A excepção lançada pode ser visualizada na figura 5.1.

```
Fatal error: exception Bulletproofs.Utils.Error("Wrong number of blinding factors!")
```

Figura 5.1: Excepção lançada quando o utilizador erra no número de *blinding factors* para cada valor.

- O tamanho do circuito da *rangeproof* (n) deve ser uma potência de dois, assim como o número de valores que se quer provar que pertencem ao intervalo (m). O valor máximo do circuito é de 64 bits, i.e., os valores que este pode tomar são $8, 16, 32$ ou 64. Sempre que não são cumpridos os requisitos, as excepções lançadas podem ser visualizadas nas figuras 5.2 e 5.3, consoante o erro do utilizador for relativamente ao tamanho do circuito ou ao número de valores;

```
Fatal error: exception Bulletproofs.Utils.Error("Invalid bit size!")
```

Figura 5.2: Excepção lançada quando o utilizador erra o tamanho do circuito.

```
Fatal error: exception Bulletproofs.Utils.Error("Invalid proof aggregation!")
```

Figura 5.3: Excepção lançada quando o utilizador erra o número de valores.

- O número de geradores do vetor **G** deve ser igual ao número de geradores do vetor **H**, e o seu número deve ser igual ou maior que o tamanho do circuito (n). A excepção lançada pode ser visualizada na figura 5.4.

```
Fatal error: exception Invalid_argument("The two lists have different lengths")
```

Figura 5.4: Exceção lançada quando o utilizador erra no número de geradores de **G**, **H**.

Depois de serem realizados vários testes concluiu-se que as *range proofs* não são computadas ou verificadas caso um dos *inputs* não cumprisse os requisitos.

Sempre que pelo menos um dos valores que se quer provar que pertencem ao intervalo, não pertença, a verificação da *range proof* deve falhar. Assim como deve falhar, quando o tamanho do circuito na construção de uma *range proof* tem um determinado valor e um valor diferente na sua verificação. Se a verificação da *range proof* falhar é porque os cálculos da sua verificação não dá o ponto do infinito da curva. No *listing* 5.1 pode-se visualizar uma prova agregada de dois valores, uma vez que o tamanho da prova é igual a $n = 2^8$ (os valores devem estar entre 0 e 255) e como a função de verificação retorna *true* caso o resultado final seja igual ao ponto infinito ou *false* caso contrário, o *assert* na linha 18 irá falhar. Isto porque um dos valores não pertence ao intervalo a execução é abortada e é lançada a exceção da figura 5.5.

```
1  ...
2  let values = [Z.of_int 5; Z.of_int 256]
3
4  let blindings = rand_numbers_list 2 BulletproofGens.get_n
5
6  let t = Sys.time()
7
8  let (proof, values_commitments) =
9    Rangeproof.prove_multiple values blindings 8
10
11 let () =
12   let time = truncate ((Sys.time() -. t) *. 1000.) in
13   printf "Proving time: %dms\n" time
14
15 let t = Sys.time()
16
17 let () =
18   assert(Rangeproof.verify_multiple proof values_commitments 8);
19
20   let time = truncate ((Sys.time() -. t) *. 1000.) in
21   printf "Verification time: %dms\n" time
```

Listing 5.1: Código para uma prova agregada de dois valores e com tamanho de prova igual a $n = 2^8$.

```
Proving time: 308ms
Fatal error: exception Assert failure("test2.ml", 55, 2)
```

Figura 5.5: Resultado da execução do código presente no *listing* 5.1.

Depois de serem realizados vários testes, concluiu-se que sempre que as *range proofs* contiverem valores fora do intervalo, a sua verificação falha.

5.3 Testes de carga

Foram realizados ao todo cinco testes de carga a *rangeproofs* agregadas, até ao valor máximo de 64 provas, cada uma com tamanho igual a 64 bits. Em cada um dos testes é apresentado o tempo em milissegundos que demora a computar e a verificar uma prova. No fim é apresentada a média dos cinco testes e é construído um gráfico com esses valores para mostrar qual é o custo da agregação de provas. Todos os testes de carga foram realizados num computador Asus, com disco *HDD* de 1TB, dos quais apenas 100GB dedicados ao sistema operativo baseado em Linux, i7-6700HQ CPU de 2.60GHz, e 16GB de RAM. Os testes foram realizados não só a esta implementação em Ocaml mas também à implementação em Rust, e os resultados são os seguintes:

- ***Rangeproofs* em Ocaml**

	Número de provas agregadas						
	1	2	4	8	16	32	64
Teste 1	912	1863	3744	7536	15483	32584	72964
Teste 2	924	1863	3744	7580	15503	32636	73264
Teste 3	916	1855	3719	7544	15467	32604	72872
Teste 4	924	1863	3759	7584	15544	32775	73164
Teste 5	972	1884	3739	7571	15472	32688	72708
Média	929.6	1865.6	3741	7563	15493.8	32657.4	72994.4

Tabela 5.1: Tempos para a computação de uma *rangeproof* em Ocaml.

	Número de provas agregadas						
	1	2	4	8	16	32	64
Teste 1	208	399	799	1676	3816	9951	28584
Teste 2	204	399	799	1675	3824	9996	28895
Teste 3	204	399	799	1668	3820	9927	28551
Teste 4	204	403	804	1676	3804	9963	28747
Teste 5	211	403	796	1676	3819	10019	28512
Média	206.2	400.6	799.4	1674.2	3816.6	9971.2	28657.8

Tabela 5.2: Tempos para a verificação de uma *rangeproof* em Ocaml.

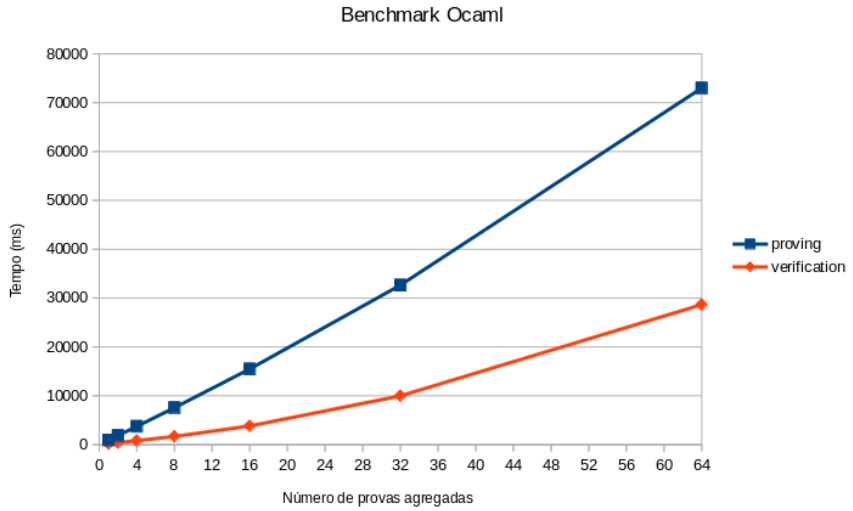

Figura 5.6: Gráfico de linhas para o custo médio de computação e verificação de uma *rangeproof* consoante a número de provas agregadas em Ocaml.

- ***Rangeproofs* em Rust**

	Número de provas agregadas						
	1	2	4	8	16	32	64
Teste 1	220	430	847	1687	3364	6707	13427
Teste 2	221	432	849	1692	3525	6781	13478
Teste 3	220	429	847	1678	3427	6764	13606
Teste 4	220	431	846	1679	3352	6721	14107
Teste 5	221	430	846	1684	3352	6799	13596
Média	220.4	430.4	847	1684	3404	6754.4	13642.8

Tabela 5.3: Tempos para a computação de uma *rangeproof* em Rust.

	Número de provas agregadas						
	1	2	4	8	16	32	64
Teste 1	30	53	104	194	378	783	1518
Teste 2	30	54	102	207	380	768	1524
Teste 3	30	54	101	194	412	758	1550
Teste 4	30	54	102	194	381	761	1571
Teste 5	31	54	102	194	381	759	1632
Média	30.2	54	102.2	196.6	386.4	765.8	1559

Tabela 5.4: Tempos para a verificação de uma *rangeproof* em Rust.

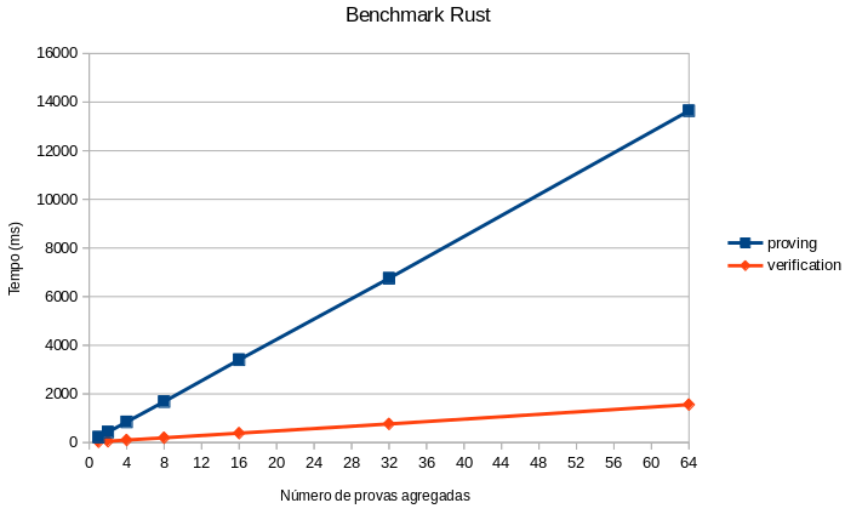

Figura 5.7: Gráfico de linhas para o custo médio de computação e verificação de uma *rangeproof* consoante a número de provas agregadas em Rust.

5.4 Conclusão

Da análise dos dados das tabelas e dos gráficos, pode-se concluir que em ambas as soluções a complexidade dos algoritmos para a computação e verificação de uma prova cresce de forma linear ($\mathcal{O}(N)$) consoante o número de provas agregadas. Também se pode concluir que a versão implementada em Rust é mais eficiente que a nossa versão em Ocaml. Uma justificação plausível para tal, é o facto da linguagem Rust ser de mais baixo nível e logo mais eficiente, e devido ao facto de a nossa implementação não estar a utilizar uma biblioteca de curvas elípticas muito eficiente. Como as *range proofs* fazem bastante uso da multiplicação de pontos, acabam por pecar na sua eficiência. Em relação ao tempo total, na implementação Rust 89% do tempo é para computar a prova e 11% para verificá-la, enquanto que na nossa implementação em Ocaml 80% do tempo é para computar a prova e 20% para verificá-la. Apesar de os testes de carga não serem os mais favoráveis, as *rangeproofs* passaram em todos os testes e estão prontas a ser utilizadas em qualquer sistema.

Capítulo 6

Conclusão

6.1 Conclusões Principais

Esta dissertação de mestrado teve como objectivo principal implementar *range proofs* em Ocaml através das *Bulletproofs*, tendo em vista a sua integração em transações realizadas na *Blockchain* da Tezos. A sua implementação foi concluída com sucesso. Contudo ainda lhe falta alguns ajustes para melhorar a sua eficiência. No entanto, é possível realizar a sua integração nas transações da Tezos, com o objetivo de esconder a quantidade de moedas da Tezos (XTZ) envolvidas numa transação com vários *outputs*.

O documento permitiu descortinar o tema das *Blockchain* que nos dias de hoje é bastante popular devido ao que a moeda digital Bitcoin veio a oferecer, e cada vez mais é um fenómeno de popularidade devido à extrema utilidade e diversificação que a mesma começa a apresentar. O futuro das *blockchains* passa não só pelo seu uso em criptmoedas mas também em *smart-contracts*, as quais permitem a escrita de contratos (como contratos de arredamento de casas) e que estes sejam guardados nela em formato digital com o intuito de que os contratos sejam respeitados e pagos nas devidas datas através da sua moeda digital.

Como tudo o que está na *blockchain* é público com o objetivo de que todos os seus participantes possam verificar e concordar com apenas um único histórico de transações. Concluí-se que é necessário que haja privacidade nelas sem comprometer a segurança do sistema. As *Bulletproofs* em forma de *rangeproofs* é apenas um dos vários métodos para se obter privacidade em transações nas *blockchains*. Contudo não há apenas um método que se destaque porque todos têm prós e contras comparados uns com os outros. Devido às *blockchains* serem um fenómeno de popularidade, as investigações para se obterem novos métodos mais seguros e sobretudo mais eficientes para a privacidade em transações têm aumentado, e o seu objetivo é de que eles possam vir a fornecer uma forma bastante rápida de verificar as transações. Para além disso, pretende-se que estas sejam cada vez menores para aumentar a escalabilidade da *blockchain* e permitir que um maior número de transações possam ser processadas por segundo. Com a implementação da privacidade nas transações, deve continuar a haver formas de controlar transações entre utilizadores para fins ilícitos, que deve ser realizada por entidades competentes para tal.

A parte da implementação envolveu a compreensão de todas etapas de computação de uma *rangeproof* e da sua verificação, bem como o processo que permite a agregação de provas (o *inner product argument*). Para tal compreensão, contribuiu bastante o *paper* das *Bulletproofs* e a documentação da implementação em Rust. Como as *Bulletproofs* assentam sobre o problema do logarítmico discreto, a implementação necessitava de uma biblioteca de curvas elípticas. Optou-se por utilizar uma já implementada em vez de uma nova. Tal decisão acabou por não ter sido uma boa escolha, devido a não possuir a devida eficiência quando vários pontos são multiplicados e somados, e devido aos ajustes que foi necessário realizar nela.

6.2 Trabalho Futuro

Todo o software deve ser mantido para melhorar e otimizar o software já desenvolvido, bem como corrigir eventuais bugs. Para tal é proposto algum trabalho que deve ser realizado no futuro:

- Procurar uma biblioteca de curvas elípticas mais eficiente e realizar as mudanças necessárias para que esta se adapte ao código já escrito;
- Fazer a integração desta biblioteca "*Bulletproofs-Ocaml*" numa *blockchain*. Para a sua integração na *Tezos*, pode desde já ser utilizada para realizar *confidental transactions* com a finalidade de "esconder" os vários *outputs* de uma transação;
- Realizar as devidas mudanças á bibloteca para que esta permita protocolos como o *CoinJoin* (vários *outputs* de várias transações em apenas uma) para que caso uma das partes haja de forma maliciosa se saiba qual delas foi.

Bibliografia

[Bac13] Adam Back. bitcoins with homomorphic value, 2013. Available from: https://bitcointalk.org/index.php?topic=305791.0. 13

[BBB+18] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, May 2018. 23, 29

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 327–357, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 23

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, New York, NY, USA, 2012. ACM. Available from: http://doi.acm.org/10.1145/2090236.2090263. 21

[BS17] Eli Ben-Sasson. Scalable, transparent and post-quantum secure computational integrity, with applications to crypto-currencies. Ethereum Foundation Developers Conference, Nov 1-4 2017, Cancún, México., 2017. Available from: https://www.youtube.com/watch?v=bhUXa0C8Ybc. 23

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018. 22

[BSCR+18] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. In *EUROCRYPT*, 2018. 25

[Bü18] Benedikt Bünz. Bulletproofs. BPASE '18, 2018. Available from: https://www.youtube.com/watch?v=gZjDKgR4dw8&feature=youtu.be. xi, xiii, 3, 24, 25

[Cha18] Inc. Chain. bulletproofs [online]. 2018. Available from: https://doc-internal.dalek.rs/bulletproofs/notes/index.html. 29

[Dam10] Ivan Damgard. On Sigma protocols, 2010. Available from: http://www.cs.au.dk/{~}ivan/Sigma.pdf. 17

[DJY18] Nik Roby Karen Scarfone Dylan J. Yaga, Peter M. Mell. Blockchain technology overview. NIST Pubs, 2018. https://doi.org/10.6028/NIST.IR.8202. 2, 11

[Glu18] Alex Gluchowski. Awesome zero knowledge proofs (zkp), 2018. Available from: https://github.com/matter-labs/awesome-zero-knowledge-proofs. xiii, 25

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991. Available from: http://doi.acm.org/10.1145/116825.116852. 15

[GNPR07] Ronen Gradwohl, Moni Naor, Benny Pinkas, and Guy N Rothblum. Cryptographic and Physical Zero-Knowledge Proof Systems for Solutions of Sudoku Puzzles. In Pierluigi Crescenzi, Giuseppe Prencipe, and Geppino Pucci, editors, *Fun with Algorithms*, pages 166–182, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 15

[Gre17] Matthew Green. Zero knowledge proofs: An illustrated primer, part 2 [online]. 2017. Available from: https://blog.cryptographyengineering.com/2017/01/21/zero-knowledge-proofs-an-illustrated-primer-part-2/. 14

[JJ+90] Jean-Jacques et al. How to explain zero-knowledge protocols to your children, 1990. Available from: http://www.cs.wisc.edu/~mkowalcz/628.pdf. 14

[Kod14] R. K. Kodali. An efficient scalar multiplication algorithm for ecc in wsns. In *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 229–233, July 2014. xi, 32

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. Available from: http://doi.acm.org/10.1145/279227.279229. 5

[Lei17] Matthew Leising. The ether thief [online]. 2017. Available from: https://www.bloomberg.com/features/2017-the-ether-thief/?source=post_page. 12

[Lin11] Yehuda Lindell. Sigma protocols and zero knowledge. Winter School on Secure Computation and Efficiency. Department of Computer Science. Bar-Ilan University., 2011. Available from: https://www.youtube.com/watch?v=nwsmG3S9wIc. 18

[Max16] Greg Maxwell. Confidential transactions, 2016. Available from: https://people.xiph.org/~greg/confidential_values.txt. 13, 23

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Report 2019/099, 2019. https://eprint.iacr.org/2019/099. xiii, 25

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Available from: http://www.bitcoin.org/bitcoin.pdf. xi, 2, 3, 5, 9

[Noe17] Sarang Noether. Monero compatible bulletproofs, 2017. Available from: https://web.getmonero.org/2017/12/07/Monero-Compatible-Bulletproofs.html. 24

[PdS19] José Pascoal and Simão Melo de Sousa. Bulletproofs-ocaml [online]. 2019. Available from: https://gitlab.com/releaselab/bulletproofs-ocaml. 46

[SCG+14] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, May 2014. 21

[Sch91] Claus Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 01 1991. 18

[Shi14] Ken Shirriff's. Bitcoins the hard way: Using the raw bitcoin protocol [online]. 2014. Available from: http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html. xi, 7, 8

[Wil16] Zooko Wilcox. The design of the ceremony, 2016. Available from: `https://z.cash/blog/the-design-of-the-ceremony/`. 22

[Wil18] Zooko Wilcox. Privacy for everyone. Just a few days after Zcash's much anticipated Sapling network upgrade activation, Zooko will share his newest perspectives on the mission to enable privacy for everyone. The various developments in new zero-knowledge proof systems and the applications in public policy and regulatory needs are establishing a new era for privacy technology., 2018. Available from: `https://slideslive.com/38911617/privacy-for-everyone`. xi, 25

[Won14] David Wong. Schnorr's signature and non-interactive protocols [online]. 2014. Available from: `https://cryptologie.net/article/193/schnorrs-signature-and-non-interactive-protocols/`. xi, 14

[Zca] Zcash. What are zk-snarks? Available from: `https://z.cash/technology/zksnarks/`. 21

[zn13] zoep and nickgian. Ecc-ocaml [online]. 2013. Available from: `https://github.com/nickgian/ECC-OCaml`. 31

Apêndice A

Exemplo de uma *Rangeproof*

5∈[0,2^n)						
n	4					
Z13	1	2	3	4	5	6
	7	8	9	10	11	12
G	2	6	7	11		
a	1	0	1	0		
2^4	1	2	4	8		
v	5					
aL	1	0	1	0		
aR	0	-1	0	-1		
aL∘aR	0	0	0	0		
aL-1-aR	0	0	0	0		
G	7	11	2	6		
H	11	2	7	6		
B	2					
$\dot{B}$	7					
$\tilde{v}$	4					
$\tilde{\tilde{a}}$	3					
$\dot{\tilde{s}}$	9					
sL	5	3	1	10		
sR	7	3	5	8		

Bit Commitment	
V	12
A	9
S	8

Bit Challenge						
y	2					
z	3					
y^n	1	2	4	8		
y^-n	1	7	10	5		
z^2v	6					
δ(y,z)	12					
aL-z1		11	10	11	10	
y^n ∘ (aR+z1) + $z^2$2^n		12	9	9	10	
z^2v+δ(y,z)	5	TRUE	5	<aL-z1,y^n ∘ (aR+z1) + $z^2$2^n>		
l0	aL-z1					
r0	y^n ∘ (aR+z1) + $z^2$2^n					
t0	5					
l1=sL	5	3	1	10		

r1=y^n ○ sR	7	6	7	12
t2=<l1,r1>	180			
l0+l1	3	0	12	7
r0+r1	6	2	3	9
t1=<l0+l1,r0+r1>-t0-t2		10		
$\dot{t}1$	4			
$\dot{t}2$	6			
T1	9			
T2	12			

Poly Challenge				
x	6			
t(x)=t0+t1x+t2x²		6		
l(x)=l0+l1(x)	2	2	4	5
r(x)=r0+r1(x)	2	6	12	4
$\dot{t}(x)= z^2 \tilde{v} + x \dot{t}1 + x^2 \dot{t}2$		3		
$\tilde{e}(x)=\tilde{a} + x\dot{s}$	5			

Proving t0 is correct

t0= <l0,r0> = < aL-z1 , y^n ○ (aR+z1) + z²2^n > =z²v+δ(y,z)

t(x)*B =	z²vB + δ(y,z)B + x t1B x² t2B		12
+	+	+	
$\dot{t}(x)\dot{B}$ =	$z^2 \tilde{v} \dot{B} + 0\dot{B} + x \dot{t}1\dot{B} + x^2 \dot{t}2\dot{B}$		8
=	=	=	
7	z²V + δ(y,z)B + xT1 + x²T2		7
	=		
TRUE	7		

Proving l(x), r(x) are correct

H'	11	1	5	4
zy^n + z²2^n	12	11	9	5
<l(x),G> =	<aL,G> + x<sL,G> + < -z1,G>			9
+	+		+	
<r(x),H'> =	<aR,H> + x<sR,H> + <zy^n + z²x^n, H'>			0
+	+		+	
$\tilde{e}(x)\dot{B}$	=			9
=	A + xS + <zy^n + z²2^n, H'> - <z1,G>		=	
5	=			5
TRUE	5			

Inner Product Proof		
w	8	
Q	3	
P = -ẽ(x)Ḃ+ A + xS + <zy^n + z²2^n, H'> - <z1,G>		9
P' = P + t(x)Q		1
K=2		
L2	1	
R2	10	
u2	5	
u2^-1	8	
u2^-2	12	
l1	3	11
r1	11	3
G1	1	1
H'1	4	11
k=1		
L1	3	
R1	1	
u1	2	
u1^-1	7	
u1^-2	10	
l0	5	
r0	5	
G0	9	
H0	7	
L1*u1²+L2*u2²+P'+R2*1/u2²+R1*1/u1²		12
l0G0 + r0G0 + l0r0Q		12
		TRUE

Final Verification				
	1/u2*1/u1	1/u2*u1	u2*1/u1	u2*u1
s	4	3	9	10
c	12			

A	9
+ xS	9
+ cz^2V	9
+ cx.T1	11
+ cx^2.T2	10
+ (w.(t(x)-l0r0)+c.(δ(y,z) -t(x))).B	9
+ (-$\tilde{e}$(x)-c$\dot{t}$(x)).$\dot{B}$	12
+ < -z1 – l0s, G>	7
+ <z1 + y^-n ○ ($z^2$2^n – b/s), H>	4
+ < [$u^2$1,$u^2$2], [L1,L2] >	11
+ < [1/$u^2$1,1/$u^2$2], [R1,R2] >	0
	0

Apêndice B

Algoritmos em *Ocaml*

B.1 Protocolo *Inner Product Argument*

```
type inner_product_proof = {
  lL_vec : point list;
  rR_vec : point list;
  a0 : Z.t;
  b0 : Z.t
}

let create_ipp_proof qQ pow_y_inv vec_G vec_H l_vec r_vec =
  let rec aux p_size i a b g h pow_y_inv l_vec r_vec =
    if i = 1 then
    {
        lL_vec = List.rev l_vec;
        rR_vec = List.rev r_vec;
        a0 = List.hd a;
        b0 = List.hd b;
    }
    else if i = p_size then
    begin
      let (a_l, a_r) = split a (i/2) in
      let (b_l, b_r) = split b (i/2) in
      let (g_l, g_r) = split g (i/2) in
      let (h_l, h_r) = split h (i/2) in
      let (p_y_inv_l, p_y_inv_r) =
        split pow_y_inv (i/2)
      in

      let c_l = inner_product a_l b_r in
      let c_r = inner_product a_r b_l in

      let l = multiscalar_mul
        (a_l @ (List.map2 (fun a b -> Z.(a * b)) b_r p_y_inv_l)
          @ [c_l]) (g_r @ h_l @ [qQ])
      in

      let r = multiscalar_mul
        (a_r @ (List.map2 (fun a b -> Z.(a * b)) b_l p_y_inv_r)
          @ [c_r]) (g_l @ h_r @ [qQ])
      in
```

```
39
40      let u =
41        let s = point_to_string l ^ point_to_string r in
42          return_challenge s
43      in
44
45      let u_inv = inverse u get_n in
46
47      let a_ = List.map2 (fun a b -> Z.(a * u + u_inv * b))
48        a_l a_r
49      in
50      let b_  = List.map2 (fun a b -> Z.(a * u_inv + u * b))
51        b_l b_r
52      in
53
54      let g_ = List.map2 (fun a b ->
55        multiscalar_mul [u_inv; u] [a; b])
56          g_l g_r
57      in
58
59      let h_ =
60        List.map2 (fun a b -> add_point a b )
61          (List.map2 (fun a b ->
62            multiscalar_mul [Z.(a * u)] [b]) p_y_inv_l h_l)
63          (List.map2 (fun a b->
64            multiscalar_mul [Z.(a * u_inv)] [b]) p_y_inv_r h_r)
65      in
66
67      aux p_size (i/2) a_ b_ g_ h_ pow_y_inv (l :: l_vec) (r :: r_vec)
68    end
69    else
70      begin
71        let (a_l, a_r) = split a (i/2) in
72        let (b_l, b_r) = split b (i/2) in
73        let (g_l, g_r) = split g (i/2) in
74        let (h_l, h_r) = split h (i/2) in
75        let c_l = inner_product a_l b_r in
76        let c_r = inner_product a_r b_l in
77
78        let l = multiscalar_mul
79          (a_l @ b_r @ [c_l]) (g_r @ h_l @ [qQ])
80        in
81
82        let r = multiscalar_mul
83          (a_r @ b_l @ [c_r]) (g_l @ h_r @ [qQ])
84        in
85
```

```
        let u =
          let s = point_to_string l ^ point_to_string r in
          return_challenge s
        in

        let u_inv = inverse u get_n in

        let a_ = List.map2 (fun a b -> Z.(a * u + u_inv * b))
          a_l a_r
        in
        let b_  = List.map2 (fun a b -> Z.(a * u_inv + u * b))
          b_l b_r
        in

        let g_ = List.map2 (fun a b ->
          multiscalar_mul [u_inv; u] [a; b]) g_l g_r
        in
        let h_ = List.map2 (fun a b ->
          multiscalar_mul [u; u_inv] [a; b]) h_l h_r
        in

        aux p_size (i/2) a_ b_ g_ h_ pow_y_inv (l :: l_vec) (r :: r_vec)
      end
  in
  let proof_size = List.length vec_G in

  (* All vector must have the same lenght *)
  assert(List.length vec_H = proof_size);
  assert(List.length l_vec = proof_size);
  assert(List.length r_vec = proof_size);
  (* And the length must be a power of two *)
  assert(is_power_of_2 proof_size);

  aux proof_size proof_size l_vec r_vec
    vec_G vec_H pow_y_inv [] []
```

Listing B.1: Função do protocolo *inner product argument*, para o *prover* obter os valores $\{L_k, R_k, ..., L_1, R_1, a_0, b_0\}$.

```
(* Computes three vectors of verification scalars (u^2, u^-2 and s)
  for combined multiscalar multiplication *)
let verification_scalars n ipp_proof =
  (* Computes the challenges u and return the lists with
    u^2, u^-2 values and the value of multiplication of all u^-1 *)
  let rec aux_u acc acc_inv mul_allinv =
    function
    | [], [] ->
      List.rev acc, List.rev acc_inv, mul_allinv
```

```
    | h_l::t_l, h_r::t_r ->
      let u =
        let s = point_to_string h_l ^ point_to_string h_r in
        return_challenge s
      in

      let u_inv = inverse u get_n in

      aux_u (Z.(u * u) :: acc) (Z.(u_inv * u_inv) :: acc_inv)
        Z.(mul_allinv * u_inv) (t_l,t_r)
    | _ ->
      raise (Invalid_argument "The two lists have different lengths")
  in

  (* Compute the s values inductively *)
  let rec aux_s i log2_n u_sq acc =
    if i = n then
      Array.to_list acc
    else
      let log2_i = truncate (log (float i)/. log 2.) in
      let k = 1 lsl log2_i in

      (* The challenges are stored in creation order as [u_k,...,u_1],
        so u[log2(i)+1] = is indexed by (log2_n-1) - log2_i *)
      let u_log2_i_sq = u_sq.((log2_n - 1) - log2_i) in
      acc.(i) <- Z.mul acc.(i - k) u_log2_i_sq;

      aux_s (i + 1) log2_n u_sq acc
  in

  let log2_n = List.length ipp_proof.lL_vec in
  let u_sq, u_inv_sq, mul_allinv =
      aux_u [] [] Z.one (ipp_proof.lL_vec, ipp_proof.rR_vec)
  in

  let s_array = Array.make n Z.one in
  s_array.(0) <- mul_allinv;
  let s = aux_s 1 log2_n (Array.of_list u_sq) s_array in

  u_sq, u_inv_sq, s
```

Listing B.2: Função do protocolo *inner product argument*, para o *verifier* obter os escalares $\{\mu_1^2, ..., \mu_k^2, \mu_1^{-2}, ..., \mu_k^{-2}, ..., s_0, ..., s_{n-1}\}$.

B.2 Algoritmo da Verificação

```
(* Verifies an aggregated rangeproof for the given value commitments. *)
let verify_multiple
    (proof : range_proof) (value_commitments : point list)  (n : int) =

  if not (n = 8 || n = 16 || n = 32 || n = 64) then
    raise (Error "Invalid bit size!");
  if List.length get_vec_G = List.length get_vec_H
  && List.length get_vec_G < n then
    raise (Error "Invalid generators length!");

  let m = List.length value_commitments in

  let gens = {
    b = get_b;
    b_blinding = get_b_blinding;
    vec_G = get_vec_G_n n;
    vec_H = get_vec_H_n n;
    order = get_n
  } in

  let s = List.fold_left (fun acc com_v ->
      acc ^ point_to_string com_v) "" value_commitments
  in

  (* Obtain bitchallenges y and z through -FiatShamir heuristic *)
  let y =
    let s =
      s ^ point_to_string proof.aA_ ^ point_to_string proof.sS_
    in
    return_challenge s
  in
  let z = return_challenge (Z.to_string y) in
  let zz = Z.(z * z) in
  let minus_z = Z.(-z) in

  let x =
    let s =
      point_to_string proof.tT1_ ^ point_to_string proof.tT2_
    in
    return_challenge s
  in

  let w =
    let s = Z.to_string proof.t_x_ ^
            Z.to_string proof.t_x_blinding_ ^
```

```
            Z.to_string proof.e_blinding_
    in
    return_challenge s
  in

  let c = random_big_int gens.order in

  let u_sq, u_inv_sq, s =
    Ipp_proof.verification_scalars (n * m) proof.ipp_proof
  in

  let a0 = proof.ipp_proof.a0 in
  let b0 = proof.ipp_proof.b0 in

  let g_scalar =
    List.map (fun s_i ->
        Z.(minus_z - a0 * s_i )) s
  in

  let h_scalar =
    let rec aux i y_inv exp_y_inv exp_z exp_2 acc = function
      | [] -> List.rev acc
      | h::t when i = n - 1 ->
        aux 0 y_inv Z.(y_inv * exp_y_inv) Z.(exp_z * z) Z.one
          (Z.(z + exp_y_inv * (zz * exp_z * exp_2 - b0 * h))
           :: acc) t
      | h::t ->
        aux (i + 1) y_inv Z.(y_inv * exp_y_inv) exp_z Z.(~$2 * exp_2)
          (Z.(z + exp_y_inv * (zz * exp_z * exp_2 - b0 * h))
           :: acc) t
    in
    aux 0 (inverse y gens.order) Z.one Z.one Z.one [] (List.rev s)
  in

  let value_commitment_scalar =
    let rec aux m z_exp acc =
      if m = 0 then
        List.rev acc
      else
        aux (m - 1) Z.(z * z_exp) (Z.(c * zz * z_exp) :: acc)
    in
    aux m Z.one []
  in

  let basepoint_scalar =
    Z.(w * (proof.t_x_ - a0 * b0) + c * ((delta n m y z) - proof.t_x_))
  in
```

```
93
94  let check = multiscalar_mul
95      ([Z.one;
96        x;
97        Z.(c * x);
98        Z.(c * x * x);
99        basepoint_scalar;
100       Z.(- proof.e_blinding_ - c * proof.t_x_blinding_)]
101     @ g_scalar
102     @ h_scalar
103     @ u_sq
104     @ u_inv_sq
105     @ value_commitment_scalar)
106
107     ([proof.aA_;
108       proof.sS_;
109       proof.tT1_;
110       proof.tT2_;
111       gens.b;
112       gens.b_blinding]
113     @ (concat_m_times gens.vec_G m)
114     @ (concat_m_times gens.vec_H m)
115     @ proof.ipp_proof.lL_vec
116     @ proof.ipp_proof.rR_vec
117     @ value_commitments)
118 in
119
120 if check = Infinity then
121   true
122 else
123   false
```

Listing B.3: Algoritmo da verificação de uma *range proof*.