UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Bianca Cristina da Silva

Computação multiparte segura em *smart contracts*

Uberlândia, Brasil

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Bianca Cristina da Silva

Computação multiparte segura em *smart contracts*

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Ivan da Silva Sendin

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2021

Bianca Cristina da Silva

Computação multiparte segura em *smart contracts*

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Trabalho aprovado. Uberlândia, Brasil, 17 de junho de 2021:

Ivan da Silva Sendin
Orientador

Lásaro Jonas Camargos

Rodrigo Sanches Miani

Uberlândia, Brasil
2021

Resumo

Smart contracts representam novas possibilidades para desenvolvimento de *softwares* descentralizados, sendo plataformas financeiras e de comércio eletrônico exemplos de tais *softwares*. Entretanto, ainda que ofereçam soluções interessantes para aplicações, tópicos como a privacidade de dados são desafiadores para *smart contracts*, uma vez que sua natureza descentralizada permite que os dados sejam acessíveis a todos os nós da rede. Nessa pesquisa, apresentamos um protocolo capaz de prover confidencialidade dos dados utilizando *smart contracts* e *Secure multi-party computation*.

Palavras-chave: *Blockchain*, *Smart Contracts*, *Secure multi-party computation*, Ethereum.

Lista de ilustrações

Figura 1 – Redes descentralizadas . . . 9
Figura 2 – Funcionamento da Prova de Trabalho . . . 13
Figura 3 – Funcionamento da Prova de Participação . . . 13
Figura 4 – Grafo G . . . 17
Figura 5 – Grafo G colorido . . . 18
Figura 6 – Apenas informação de dois vértices de G são reveladas . . . 18
Figura 7 – Circuito Aritmético . . . 26
Figura 8 – Curva Elíptica . . . 30
Figura 9 – Grafo G' . . . 32
Figura 10 – Divisão do grafo G' entre Alice e Bob . . . 33
Figura 11 – Grafo G' colorido . . . 34
Figura 12 – Etapa de Ofuscamento . . . 38
Figura 13 – Execução do protocolo proposto . . . 40
Figura 14 – Tabela T inicial . . . 49
Figura 15 – Análise de Custos . . . 51

Lista de abreviaturas e siglas

CNDL	Confederação Nacional de Dirigentes Lojistas
CRS	Commom Reference String
DaaS	*Data as a Service*
KCA	Knowledge of Coefficient Assumption
PoS	Proof of Stake
PoW	Proof of Work
QAP	Quadratic Arithmetic Program
SMC	*Secure multi-party computation*
zk-SNARK	*Zero-knowledge succinct non interactive arguments of knowledge*
zk	Zero-knowledge
SPC	Serviço de Proteção ao Crédito

Sumário

1	**INTRODUÇÃO**	**8**
1.1	**Motivação**	**8**
1.2	**Objetivos**	**9**
2	**FUNDAMENTAÇÃO TEÓRICA**	**11**
2.1	**Ethereum**	**11**
2.1.1	Protocolos de consenso	12
2.1.2	*Smart contracts*	14
2.1.3	Vulnerabilidades	14
2.2	**zk-SNARK**	**17**
2.2.1	Definição	19
2.2.2	Criptografia	20
2.2.3	Ocultação homomórfica	20
2.2.4	Avaliação cega de polinômios	22
2.2.5	Conhecimento da suposição de coeficiente	23
2.2.6	Circuitos aritméticos	25
2.2.7	Programa aritmético quadrado	26
2.2.8	Protocolo Pinóquio	28
2.3	**Computação Multiparte Segura**	**31**
2.3.1	Definição	34
2.3.2	Aplicações	36
2.4	**Trabalhos relacionados**	**36**
3	**DESENVOLVIMENTO**	**38**
3.1	**Protocolo**	**38**
3.1.1	Ofuscamento	38
3.1.2	Execução	39
3.1.3	Execução Desonesta	40
3.2	**Implementação**	**41**
3.2.1	Contrato	42
3.2.2	Testes Unitários	45
3.3	**Aplicabilidade**	**48**
3.4	**Análise de Gás**	**50**
4	**CONCLUSÃO**	**52**

REFERÊNCIAS 53

1 Introdução

O presente capítulo apresenta uma introdução do assunto abordado na pesquisa. A seção 1.1 descreve a motivação relacionada ao tema. Já a seção 1.2 especifica os objetivos da pesquisa.

1.1 Motivação

O advento da Revolução Digital acarretou em profundas mudanças nos mais diversos setores: comunicação, consumo, jornalismo, entre outros. Um dos setores mais participativo nessa transformação é o financeiro, sobretudo, devido ao surgimento dos bancos digitais. Segundo dados obtidos pela Confederação Nacional de Dirigentes Lojistas (CNDL) e pelo Serviço de Proteção ao Crédito (SPC), aproximadamente 40% dos consumidores entrevistados são clientes de bancos digitais (BRASIL, 2020), algo que demonstra a popularização de serviços dessa natureza. Todavia, a maioria das informações trafegadas nesses serviços são concentradas em um grupo limitado de corporações, as quais, além de reter as informações, são responsáveis por tratá-las e garantir que estejam disponíveis para os clientes, ocasionando, assim, pontos únicos de falha.

Visando minimizar as consequências negativas de pontos únicos de falha, redes descentralizadas como Bitcoin e Ethereum despontaram. Nessas plataformas, as informações são distribuídas ao longo dos nós presentes na rede e difundida por meio de ações dos nós que as utilizam (DABEK et al., 2004). Na Ethereum, os nós são clientes que implementam a Ethereum e são responsáveis por validar transações de cada bloco da rede. A Figura 1 ilustra como os nós se organizam nessas plataformas, sendo que as arestas representam a estrutura de uma rede descentralizada.

Especificamente sobre a Ethereum, plataforma utilizada nessa pesquisa, houve a adoção do conceito de aplicações descentralizadas baseadas na *blockchain*. Para que isso seja alcançado, a Ethereum introduziu uma linguagem Turing completa por meio dos *smart contracts* (TIKHOMIROV, 2017) também denominados contratos inteligentes. Esses contratos são capazes de definir conjuntos de regras, os quais determinam o comportamento da aplicação na Ethereum, possibilitando a implementação de diversas aplicações com funcionalidades distintas. Uma característica interessante acerca desses contratos é que, diferente das aplicações difundidas em redes centralizadas, a execução dos *smart contracts* é controlada pela existência de um recurso denominado gás, sendo que a execução procede enquanto esse recurso está disponível.

Entretanto, ainda que seja uma tecnologia promissora, para garantir as proprie-

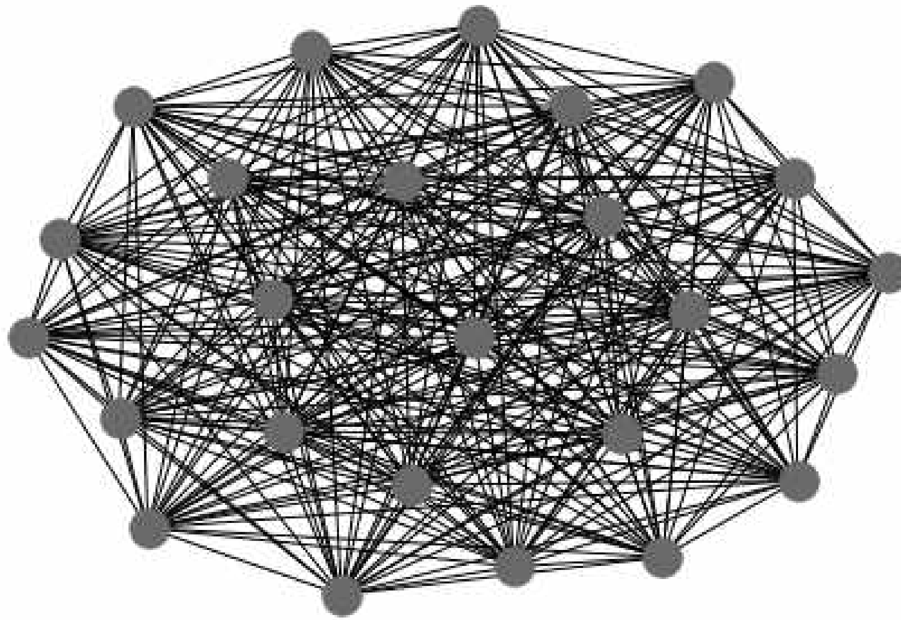

Figura 1 – Redes descentralizadas

dades da *blockchain*, os *smart contracts* herdam características desencorajadas em termos de privacidade de dados. Uma dessas características é o fato das transações realizadas na Ethereum serem processadas pelos nós da rede de forma descentralizada e, portanto, é necessário que essas operações sejam públicas (STEFFEN et al., 2019). Em virtude disso, não é possível armazenar informações sensíveis em *smart contracts* de forma segura, uma vez que esses dados são disponibilizados para todos os nós da rede.

1.2 Objetivos

Nesse contexto, o objetivo geral da presente pesquisa é aplicar técnicas que permitam a obtenção de privacidade de dados em redes descentralizadas como a Ethereum. Já os objetivos específicos do estudo são os seguintes:

1. Análise de uma técnica de provas de conhecimento zero denominada *Zero-knowledge succinct non interactive arguments of knowledge (zk-SNARK)*[1];
2. Análise de técnicas de *Secure multi-party computation (SMC)*;
3. Implementação de uma biblioteca utilizando *smart contracts* e SMC;
4. Análise do comportamento da biblioteca desenvolvida em um cenário real.

[1] Não há um padrão de tradução estabelecido para *zk-SNARK*. Portanto, nesse estudo, a sigla referente à definição em inglês será adotada.

Especificamente sobre o objetivo 4, o cenário real escolhido para analisar o comportamento da biblioteca é o seguinte: suponha que Alice e Bob desejam casar-se e a modalidade de união escolhida é a comunhão parcial de bens, ou seja, o casal optou pela separação dos bens anteriores ao casamento (PEREIRA, 1994) e, além disso, tais bens não devem ser revelados durante a união. Nesse contexto, a pesquisa almeja utilizar a biblioteca desenvolvida para permitir que a união seja realizada e passível de verificação, além de prover análises relacionadas ao custo necessário para a execução do *smart contract*.

Ademais, a principal justificativa para o estudo e desenvolvimento de uma biblioteca que conjure *smart contracts* e SMC é a promessa de combinar privacidade de dados e aplicações em redes descentralizadas. Além disso, tal biblioteca proporciona a vantagem de garantir a corretude dos cálculos já proporcionada pelos *smart contracts* e segurança a ser provida por técnicas como SMC e zk-SNARK, algo que ampliará as possibilidades em termos de aplicações descentralizadas.

2 Fundamentação Teórica

O presente capítulo apresenta a fundamentação teórica da pesquisa. A seção 2.1 discute características relacionadas à *blockchain* Ethereum. Já a seção 2.2 apresenta as definições matemáticas interessantes da técnica *zk-SNARK*. De maneira semelhante, a seção 2.3 discute definições necessárias para compreender o método SMC. Finalmente, a seção 2.4 apresenta estudos que interseccionam os objetivos da presente pesquisa.

2.1 Ethereum

A Ethereum é uma plataforma de *software* aberta que permite o desenvolvimento e *deploy* de aplicações descentralizadas baseadas na *blockchain*. O funcionamento dessas aplicações ocorre da seguinte forma:

(a) Desenvolvedor cria uma aplicação descentralizada e efetua o *deploy* na Ethereum;

(b) Após *deploy*, a aplicação estará disponível na *blockchain* para que usuários da rede usufruam dela via transações na Ethereum;

(c) Cada transação é validada, mantendo, assim, o estado da rede consistente.

No passo (b), para que o usuário consiga executar o contrato um recurso denominado gás é necessário. Na Ethereum, o gás atua como combustível para a execução dos *smart contracts* (GRECH et al., 2018), os quais são cobrados utilizando a moeda nativa da Ethereum denominada *ether*.

As principais vantagens associadas a Ethereum e suas aplicações estão relacionadas aos seguintes pontos (BUTERIN, 2016):

(i) **Autonomia.** Utilizando aplicações descentralizadas, elimina-se a necessidade de intermediários nas operações;

(ii) **Transparência.** Uma vez que a Ethereum é uma rede descentralizada, as transações precisam ser públicas a fim de que os nós da rede sejam capazes de validá-las. Logo, existe maior transparência no processo;

(iii) ***Backup.*** As transações são duplicadas diversas vezes, logo, existem cópias de segurança desses dados;

(iv) **Redução de custos.** Em virtude da ausência de intermediários, aplicações armazenadas na Ethereum tendem a custos de manutenção reduzidos;

(v) **Precisão.** A automatização de processos reduz a chance de ocorrência de erros devido a trabalhos manuais.

Para que a Ethereum funcione adequadamente, dois pontos são fundamentais: protocolos de consenso e *smart contracts*, os quais são abordados, respectivamente, nas seções 2.1.1 e 2.1.2.

2.1.1 Protocolos de consenso

Como mencionado anteriormente, as aplicações são armazenados na *blockchain*, a qual é uma estrutura que permite que criptomoedas registrem transações em dados descentralizados (LUU et al., 2016). Em virtude disso, para manter o estado da *blockchain* consistente, protocolos de consenso são utilizados.

Em sistemas distribuídos, protocolos de consenso são algoritmos que auxiliam a disseminação de informação na rede de modo que os nós participantes sejam capazes de concordar com o estado atual dessa rede (XIAO et al., 2019). Nesse sentido, especificamente para a *blockchain*, o papel dos protocolos de consenso é garantir que os nós aprovem o histórico de transações da cadeia, minimizando, assim, a influência de nós maliciosos (Sankar; Sindhu; Sethumadhavan, 2017). Para atingir tal objetivo, existem dois protocolos de consenso principais no contexto das criptomoedas:

(i) Prova de Trabalho ou *Proof of Work (PoW)*;

(ii) Prova de Participação ou *Proof of Stake (PoS)*.[1]

No protocolo (i), participantes denominados mineradores competem para resolver uma computação com uso intensivo de recursos (LIU; CAMP, 2006) produzido pela *blockchain* e aquele que resolver primeiro recebe recompensas em criptomoedas. A Figura 2 ilustra o procedimento.

Já no protocolo (ii), a permissão para validar o bloco é baseada na quantidade de criptomoedas que um participante possui. Para decidir qual participante validará um bloco, o protocolo usufrui de um algoritmo probabilístico baseado na quantidade de criptomoedas do participante de modo que uma maior quantidade de criptomoedas oferece maiores chances de receber permissão para validar um novo bloco (LEDGER, 2019a). Esse procedimento é demonstrado pela Figura 3.

Cada protocolo possui prós e contras. A vantagem do *PoW* é permitir que a *blockchain* funcione com consenso distribuído, seguro e sustentável, enquanto a desvantagem

[1] A tradução literal de *Proof of Stake* é Prova de Montante. Todavia, considerando o funcionamento do protocolo, Prova de Participação é uma tradução mais adequada. Além disso, assim como em outros momentos do estudo, a sigla em inglês será utilizada para referenciar tanto a Prova de Trabalho quanto Prova de Participação.

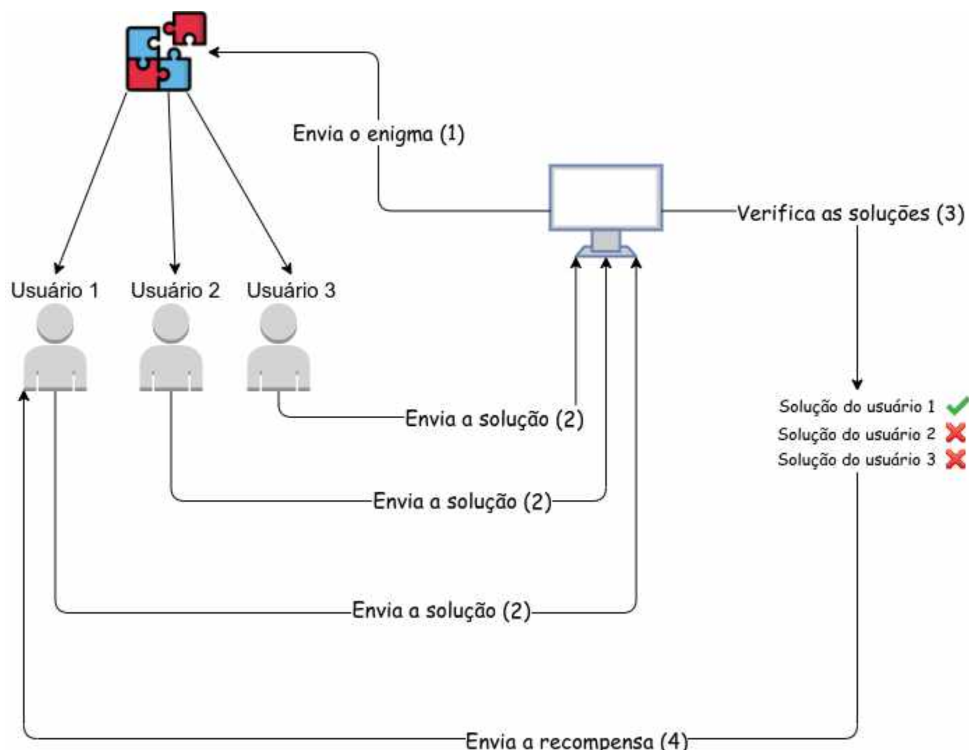

Figura 2 – Funcionamento da Prova de Trabalho

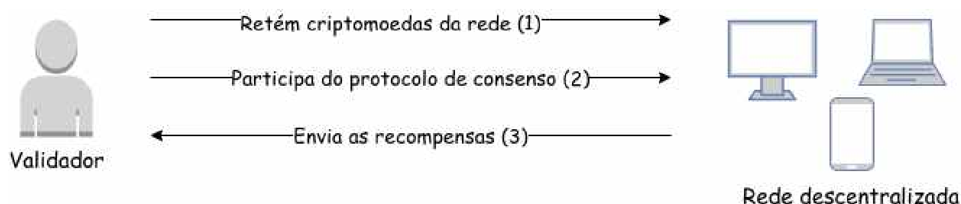

Figura 3 – Funcionamento da Prova de Participação

é o alto custo de *hardware* e energia para suportar os cálculos envolvidos na solução do enigma (LEDGER, 2019b). Já a vantagem do *PoS* é a menor necessidade *hardwares* específicos, enquanto a desvantagem está relacionada a pontos críticos de vulnerabilidade a ataques (LEDGER, 2019a). Ainda que o *PoS* possua quesitos de vulnerabilidade, em ambos os protocolos é necessário que o participante se comprometa com o sistema, uma vez que no *PoW* o participante se compromete via disponibilização de recursos de *hardware* enquanto no *PoS* o comprometimento é dado pela existência de criptomoedas do participante que incentivam a honestidade no processo.

Especificamente sobre a Ethereum, o protocolo adotado atualmente é a Prova de Trabalho, entretanto, existe um planejamento para que a plataforme migre para a Prova de Participação (ETHEREUM, 2021). Além disso, para esse estudo, a compreensão dos protocolos de consenso está relacionada às características adquiridas pelas aplicações que dependem desses protocolos, as quais serão abordadas em seções posteriores.

2.1.2 *Smart contracts*

Segundo (PINNA et al., 2019), um *smart contract*, denominado contrato inteligente em português, é um programa que implementa uma sequência de passos de acordo com determinadas cláusulas e regras, o qual possui três componentes fundamentais:

1. O código do programa;
2. O conjunto de mensagens que o programa pode receber;
3. O conjunto de métodos que resultam nas reações definidas na lógica do contrato.

Além disso, contratos inteligentes são imutáveis, armazenados na *blockchain* e identificados por um endereço. Dessa forma, como detalhado em (LUU et al., 2016), usuários podem invocar um contrato inteligente por meio do seguinte processo:

(a) Usuário envia transações para o endereço do contrato;

(b) Caso a *blockchain* aceite as transações, todos os outros usuários envolvidos na mineração executam o contrato utilizando o estado atual da *blockchain*;

(c) As transações são, então, carregadas como entradas;

(d) A rede aceita a saída e o estado seguinte do contrato por meio de um protocolo de consenso.

No passo (d), o próximo estado do contrato depende do protocolo de consenso entre todos os envolvidos.

A fim de permitir a implementação do processo supracitado, a Ethereum utiliza a linguagem Solidity, a qual é orientada a objetos e inspira-se em linguagem como Python, C++ e JavaScript. Dentre as principais *features* associadas a essa linguagem estão as variáveis de estado, modificadores de função, herança entre contratos, eventos e funções de *callback*. A documentação completa da versão mais recente da linguagem está disponível em (ETHEREUM, 2020).

2.1.3 Vulnerabilidades

Uma vez que transações, normalmente, lidam com quantidades relevantes de dinheiro, contratos inteligentes são alvos atrativos para usuários mal intencionados. Protocolos como os de consenso visam minimizar essas vulnerabilidades, entretanto os contratos ainda sofrem com problemas relacionados à segurança, sendo os principais:

Imutabilidade. A imutabilidade é uma característica intrínseca de contratos inteligentes, a qual garante que alterações aprovadas não podem ser modificadas após *deploy* na *blockchain* (RAMACHANDRAN; KANTARCIOGLU, 2017). Contudo, ainda que seja vantajosa em alguns pontos, a imutabilidade permite que usuários mal intencionados explorem brechas existentes em contratos já armazenados na *blockchain*, uma vez que para corrigir essas brechas seria necessário reverter a *blockchain*, algo que não é trivial (LUU et al., 2016). É importante ressaltar que, ainda que os contratos sejam, de fato, imutáveis, existem *design patterns* adotados pela comunidade tais como o *Mortal Pattern* que minimizam os efeitos dessa vulnerabilidade (WÖHRER; ZDUN, 2018);

Dificuldade na realização de testes. A falta de frameworks de teste conceituados para o Solidity como existem para outras linguagens (JUnit para Java, por exemplo) e a imutabilidade dos contratos após armazenamentos na *blockchain* dificultam a realização de testes nos contratos (DESTEFANIS et al., 2018). Devido a isso, contratos inteligentes estão suscetíveis a eventuais brechas em sua implementações;

Uso de gás limitado. A execução de contratos inteligentes na Ethereum é restringida pela quantidade de gás disponível (GRECH et al., 2018), em virtude disso quando funções de *fallback* dispendiosas são executadas, é provável que exceções relacionadas à falta de gás ocorram. Caso essas exceções não sejam tratadas de forma correta, é possível que usuários mal intencionados sejam capazes de armazenar *ether* incorretamente (JIANG; LIU; CHAN, 2018);

Dependência da ordem da transação. Ocorre quando assume-se um estado da *blockchain* que não corresponde ao atual. Logo, visto que a ordem de mineração das transações afeta qualquer outra transação, caso o estado assumido da *blockchain* seja incorreto, outras transações serão afetadas (LEE, 2017);

Tratamento de exceção instável. Refere-se ao modo instável de tratamento de exceções adotado pela linguagem Solidity devido a dependência na forma como contratos inteligentes interagem entre si. Nesse contexto, quando uma exceção ocorre, a transação é revertida completamente. Entretanto, caso algum dos contratos realize chamadas de baixo nível no endereço, a reversão da transação ocorrerá até o ponto da função chamada e depois desse ponto a exceção não será propagada. Desse modo, os contratos inteligentes chamados não serão alertados das exceções em tempo de execução (JIANG; LIU; CHAN, 2018). Além disso, haverá desperdício de gás para executar um contrato cujas ações serão descartadas (LEE, 2017);

Reentrância. Uma função é chamada de reentrante se execuções distintas dessa função com entradas diferentes não afetam-se entre si (WLOKA; SRIDHARAN; TIP, 2009). Na linguagem Solidity, algumas funções não foram desenvolvidas com o

propósito de serem reentrantes. Entretanto, usuários maliciosos invocam tais funções de forma reentrante, o que pode causar transferências múltiplas de *ether* de forma incorreta (Liu et al., 2018). Essa vulnerabilidade ocasionou um ataque conhecido como DAO, abordado em (ATZEI; BARTOLETTI; CIMOLI, 2017), o qual envolveu a perda de, aproximadamente, 60 milhões de dólares;

Dependência de *timestamp*. Ocorre quando um contrato utiliza o *timestamp* do bloco de forma condicional para executar operações ou como fonte para gerar números aleatórios. Essa dependência pode ser explorada por usuários mal intencionados que manipulam o *timestamp* do bloco (JIANG; LIU; CHAN, 2018);

Dependência de número de bloco. Funciona de forma semelhante a dependência de *timestamp* com ênfase na questão de ser usado para gerar números aleatórios;

Perda de *ether* em transações. Caso o endereço destino da transação não corresponda a nenhum usuário, o *ether* será perdido e não é possível recuperá-lo (LEE, 2017);

Chamadas do tipo *delegatecall*. Contratos inteligentes podem invocar funções de outros contratos por meio de *delegatecall*. Contudo, se o argumento de uma *delegatecall* do contrato for *msg.data*, esse contrato está vulnerável a ataques, visto que quando o argumento é do tipo *msg.data* um usuário malicioso pode fazer com que o contrato invoque qualquer função (JIANG; LIU; CHAN, 2018). Essa vulnerabilidade acarretou em um ataque a *Parity Wallet*, o qual foi abordado em (DESTEFANIS et al., 2018);

Congelamento de *ether*. Quando um contrato invoca outro por meio de *delegatecall*, existe uma relação de confiança entre os dois contratos, visto que um deles confiará no outro para enviar *ether*. Todavia, quando o contrato que manipula *ether* possui algum problema em suas operações, então não há maneiras de enviar *ether*, logo, o *ether* da transação será congelado. Essa vulnerabilidade também relaciona-se com o ataque a *Parity Wallet*, uma vez que os contratos que utilizavam a biblioteca por meio de *delegatecall* foram afetados e seus *ethers* foram congelados (JIANG; LIU; CHAN, 2018);

Privacidade. Visto que os contratos são imutáveis e públicos após *deploy* na *blockchain*, não é possível armazenar informações sensíveis nesses aplicações (CHENG et al., 2019). Além de informações sensíveis, dados relacionados ao saldo das contas (KOSBA et al., 2016) e configurações gerais também são públicos, algo que inviabiliza a proteção de dados dessa categoria na Ethereum.

2.2 zk-SNARK

As primeiras discussões acerca de provas de conhecimento zero surgiram na década de 80, sendo (GOLDWASSER; MICALI; RACKOFF, 1989) uma das pesquisas de maior destaque, com o objetivo de extinguir a quantidade de informações reveladas na tentativa de demonstrar se uma afirmação é verdadeira.

Tal discussão é útil em casos como a coloração de grafos. Nesse contexto, dado um grafo G e quaisquer vértices v,u de G, para todo o grafo G não há vértices v,u adjacentes que compartilham a mesma cor (BONDY; MURTY, 1976). As figuras 4 e 5 representam essa situação.

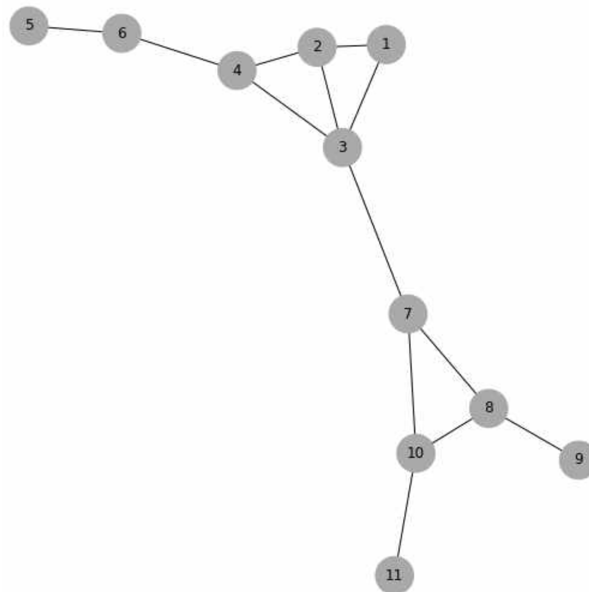

Figura 4 – Grafo G

Para demonstrar como é possível relacionar a coloração de grafos com provas de conhecimento zero, a seguinte situação é proposta: supondo que haja duas pessoas, Alice e Bob, e que Alice deseja provar para Bob que é capaz de colorir grafos de modo que vértices adjacentes não sejam coloridos com a mesma cor, assim como apresentado pela Figura 5. Existem mais de uma possibilidade para que isso seja feito, sendo que uma delas consiste em Alice apresentar todo o grafo colorido da Figura 5 para Bob e, assim, Bob pode conferir e atestar que Alice realmente consegue colorir grafos. Todavia, nessa solução, Alice revelaria informações além do fato de ser possível ou não colorir o grafo tais como o funcionamento do algoritmo. Nesse ponto, provas de conhecimento zero são relevantes, visto que permitem que Alice prove para Bob que o grafo foi colorido da forma correta sem, necessariamente, revelar o grafo colorido. Desse modo, para que Alice prove que consegue colorir grafos utilizando provas de conhecimento zero, o seguinte é proposto:

(a) Bob informa o grafo G a ser colorido (Figura 4)

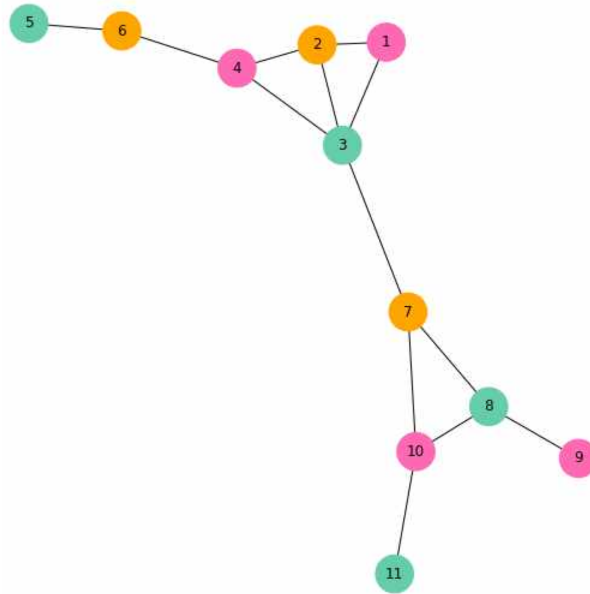

Figura 5 – Grafo G colorido

(b) Alice colore o grafo da forma correta (Figura 5)

(c) Alice permite que Bob escolha um vértice e revela o vértice escolhido e algum de seus adjacentes (Figura 6)

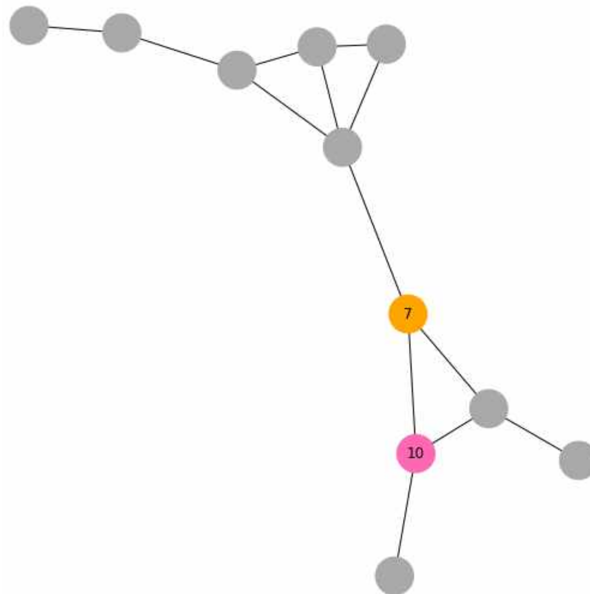

Figura 6 – Apenas informação de dois vértices de G são reveladas

Naturalmente, somente o passo (c) não é suficiente para convencer Bob. Em virtude disso, o procedimento anterior é repetido múltiplas vezes com colorações distintas do mesmo grafo e, conforme a solução apresentada por Alice esteja correta em todas

as verificações no passo (c), a probabilidade de que Alice esteja mentindo diminui até alcançar um ponto que essa probabilidade seja tão baixa que possa ser descartada.

Além disso, o procedimento descrito anteriormente é uma prova de conhecimento zero iterativa, o que significa que, ainda que Alice prove para Bob que consegue colorir grafos, caso exista outra pessoa a ser convencida por Alice, todo o procedimento seria refeito, pois, da forma como foi descrito, o procedimento não é transferível para uma terceira pessoa. Esse fato introduz a necessidade de desenvolver provas não iterativas a fim de que o provador possa demonstrar que a afirmação é verdadeira e o verificador seja capaz de conferir a prova por si.

2.2.1 Definição

Provas de conhecimento zero possuem três características fundamentais: completude, solidez e conhecimento zero. A completude está relacionada à capacidade que o provador possui de convencer o verificador caso a afirmação seja verdadeira. Já a solidez consiste em não haver possibilidade de um provador convencer o verificador de que uma afirmação falsa é verdadeira. Por último, o conhecimento zero é a propriedade que indica que um provador mal intencionado não consegue aprender nada além do fato de que a afirmação é verdadeira.

O método *zk-SNARK* consiste em uma especificação de provas de conhecimento zero com determinadas características adicionais, sendo elas:

(i) Sucintês: a prova é constante e de tamanho pequeno independente da quantidade de cálculos (PETKUS, 2019);

(ii) Não iteratividade: o provador apresenta os resultados dos cálculos e outras entidades envolvidas no protocolo são capazes de atestar a veracidade da informação fornecida por si (KOENS; RAMAEKERS; WIJK, 2017);

(iii) Conhecimento argumentado: é inviável que tanto o provador quanto o verificador consigam construir provas falsas e o protocolo *zk-SNARK* as considerem verdadeiras (PETKUS, 2019).

Dentre as características do *zk-SNARK*, é fundamental ressaltar a relação entre a sucintês e não iteratividade. Como em provas não iterativas cada entidade é capaz de verificar a afirmação fornecida, se o método não for sucinto, essa verificação pode ser lenta quando a informação é complexa. Todavia, no *zk-SNARK*, essa verificação independe se a informação é simples ou complexa, ou seja, o tempo de verificação não depende da complexidade da afirmação (BEN-SASSON et al., 2014).

2.2.2 Criptografia

Na maioria dos estudos sobre provas de conhecimento zero, técnicas algébricas são adotadas para explicar os conceitos do método. Sendo assim, uma simples operação matemática de adição entre duas entidades poderia usufruir de conceitos de conhecimento zero. Suponha que Alice e Bob substituíram os grafos por valores x, y de modo que Alice deseja provar para Bob que conhece x, y tais que $x + y = 11$. Uma forma fácil de resolver essa situação seria Alice enviar x, y e Bob realizar a operação para conferir se o resultado é válido. Entretanto, como essa solução revelaria os valores retidos por Alice, caso Bob fosse uma entidade mal intencionada, informação acerca de como o método realiza os cálculos seriam, de certa forma, expostas.

Em virtude disso, para garantir todas as propriedades relacionadas ao *zk-SNARK*, é interessante que os valores das operações utilizem suposições criptográficas de modo que seja possível realizar operações aritméticas nesses valores. Além disso, semelhante ao funcionamento de funções *hash*, deve ser difícil determinar o valor original de x à partir do valor computado de x (PETKUS, 2019).

A fim de analisar conceitos relacionados ao *zk-SNARK*, a plataforma Zcash foi escolhida, a qual possui um extenso artigo (ZCASH, 2017) que será utilizado como base nas seções 2.2.3 a 2.2.8 para explicar os conceitos matemáticos associados ao *zk-SNARK*.

2.2.3 Ocultação homomórfica

Uma ocultação homomórfica *E(x)* é uma função com as seguintes propriedades:

1. Para a maioria dos valores x escolhidos, dada *E(x)*, é difícil encontrar x;
2. Entradas diferentes resultam em saídas diferentes, comportamento semelhante ao de funções *hash*;
3. É possível aplicar operações aritméticas em ocultações homomórficas. Por exemplo, dado *E(x)* e *E(y)*, é possível e válido determinar *E(x + y)*.

A utilidade da ocultação homomórfica em provas de conhecimento zero pode ser demonstrada na situação descrita anteriormente na qual Alice para Bob que conhece os valores x, y. Assim:

(a) Alice envia *E(x)* e *E(y)* para Bob;

(b) Bob calcula *E(x + y)*;

(c) Bob calcula *E(11)* e verifica se *E(x + y)* = *E(11)*. Se os valores coincidirem, então Bob aceita a prova de Alice.

No processo acima, mesmo que os valores de x e y estejam, de certa forma, escondidos e a ocultação homomórfica seja utilizada, ainda há possibilidade de Bob aprender alguma informação sobre x e y, o que violaria os princípios de provas de conhecimento zero. Devido a isso, esse processo não é completamente uma prova de conhecimento zero e apenas demonstra, superficialmente, pontos interessantes da ocultação homomórfica que podem relacionar-se com provas de conhecimento zero. Nas próximas seções, esse detalhe será novamente abordado a fim de mostrar como a ocultação homomórfica é, de fato, utilizada em zk-SNARK.

Além disso, ao utilizar ocultação homomórfica, utiliza-se aritmética modular (mod n) para obter o resultado da função *E(x)* em um intervalo {0,...,n-1}. Mais especificamente, é possível utilizar um número primo p[2] para que o resultado de uma multiplicação, por exemplo, esteja no conjunto {1,...,p-1} por meio da realização da multiplicação de forma normal e utilizando a aritmética modular no resultado. Devido a isso, denomina-se $\mathbb{Z}_p^*$ o conjunto de elementos com essa operação, o qual possui as seguintes propriedades:

1. É um grupo cíclico, no qual existe algum elemento gerador g, $g \in \mathbb{Z}_p^*$, de modo que todos os elementos de $\mathbb{Z}_p^*$ podem ser expressos como g^α, $\alpha \in \{0, ..., p-2\}$, sendo $g^0 = 1$;
2. Considera-se que o problema do logaritmo discreto é difícil em $\mathbb{Z}_p^*$. Em virtude disso, quando p é suficientemente grande, dado $h \in \mathbb{Z}_p^*$, é difícil encontrar $\alpha \in \{0, ..., p-2\}$ de modo que $g^\alpha = h \pmod p$;
3. De acordo com as propriedades da potenciação, o produto de potências de mesma base resulta na manutenção da base e soma dos expoentes. Nesse caso, dado $a, b \in$ {0,...,p-2}, temos $g^a \cdot g^b = g^{a+b \pmod{p-1}}$.

Dessa forma, é possível reconstruir o processo entre Bob e Alice de modo a utilizar ocultação homomórfica assertivamente. Assim, assumindo que $x \in \mathbb{Z}_p^*$, $E(x) = g^x$ e $E(x)$ é uma ocultação homomórfica, temos:

(a) Alice envia $E(x)$ e $E(y)$ para Bob;

(b) Bob calcula $E(x+y)$ sendo $E(x+y) = g^{x+y \pmod{p-1}} = g^x \cdot g^y = E(x) \cdot E(y)$;

(c) Bob calcula *E(11)* e verifica se *E(x + y) = E(11)*. Se os valores coincidirem, então Bob aceita a prova de Alice.

[2] Se a escolha fosse por um número não primo, a definição da multiplicação poderia ser problemática. Isso ocorre devido ao fato de que o resultado da multiplicação seguida da aritmética modular poderia ser zero mesmo que os dois argumentos da multiplicação não fossem zero. Um exemplo disso é, dado $p = 4$, $2 \cdot 2 = 0 \pmod 4$.

2.2.4 Avaliação cega de polinômios

Antes de detalhar a avaliação cega de polinômios é necessário compreender como os polinômios e a ocultação homomórfica se relacionam. Considerando o que foi abordado na seção anterior, denota-se $\mathbb{F}_p$ cujos elementos são $\{0,...,p\text{-}1\}$ e operações tais como multiplicação e adição podem ser realizadas utilizando-se aritmética modular. Nesse contexto, dado o seguinte polinômio $P(X)$:

$$P(X) = \alpha_0 + \alpha_1 \cdot X + \alpha_2 \cdot X^2 + ... + \alpha_d \cdot X^d \tag{2.1}$$

onde $\alpha_0, ..., \alpha_d \in \mathbb{F}_p$.

É possível calcular *P(s)*, $s \in \mathbb{F}_p$, e obter a soma resultante:

$$P(s) = \alpha_0 + \alpha_1 \cdot s + \alpha_2 \cdot s^2 + ... + \alpha_d \cdot s^d \tag{2.2}$$

A relação entre ocultação homomórfica e o obtido em 2.1 e 2.2 é o fato de que, implicitamente, a ocultação homomórfica suporta combinações lineares dado o fato de que suporta operações aritmética como multiplicação e adição. Sendo assim, para um polinômio relativamente simples como $ax+by$, o seguinte resultado seria obtido ao utilizar ocultação homomórfica:

$$E(ax + by) = g^{ax+by} = g^{ax} \cdot g^{by} = (g^x)^a \cdot (g^y)^b = E(x)^a \cdot E(y)^b \tag{2.3}$$

Com essas definições, é possível realizar a avaliação cega de polinômios em conjunto com ocultação homomórfica. Para tal, a seguinte situação é proposta: Alice possui um polinômio P de grau d, Bob possui um ponto $s \in \mathbb{F}_p$ e Bob deseja aprender a ocultação homomórfica da avaliação de P no ponto s, ou seja, $E(P(s))$. Assim, temos:

(a) Bob envia $E(s^0), E(s^1), E(s^2), ..., E(s^d)$ para Alice;

(b) Alice calcula $E(P(s))$ a partir dos elementos do passo anterior e envia $E(P(s))$ para Bob.

A utilidade para esse processo em provas de conhecimento zero será discutida detalhadamente em seções posteriores. Contudo, resumidamente, é possível dizer que esse processo é útil em situações onde o verificador possui um polinômio e precisa checar se o verificador conhece esse polinômio e, ao obrigar o provador a avaliar P no ponto s, existe uma garantia de que o provador retornará um resultado inválido caso o polinômio que afirma ter não seja o correto.

2.2.5 Conhecimento da suposição de coeficiente

No processo anterior, no qual Alice e Bob utilizam ocultação homomórfica e avaliação cega de polinômios, não há garantias de que Alice realmente envia $E(P(s))$ e não um valor aleatório a fim de enganar o protocolo. Para que tal garantia exista, utiliza-se o *Knowledge of Coefficient Assumption (KCA)*, também conhecido como conhecimento da suposição de coeficiente. Antes de iniciar o teste baseado no *KCA*, supõe-se o seguinte:

1. Denota-se g o gerador de um grupo G, o qual $|G| = p$, ou seja, a quantidade de elementos no grupo G é p;
2. Para o grupo G e $\mathbb{F}_p$, o problema do logaritmo discreto é difícil;
3. No contexto atual, é vantajoso definir o grupo G usando adições em detrimento de multiplicações. Dessa forma, para $\alpha \in \mathbb{F}_p$, $\alpha \cdot g$ é o resultado de somar α cópias de g.

Além disso, é necessário definir α-par[3]. Para $\alpha \in \mathbb{F}_p^*$, um par (a, b) é α-par se $a, b \neq 0$ e $b = \alpha \cdot a$.

Assim, o teste de conhecimento de coeficiente ocorre da seguinte forma:

(a) Bob escolhe $\alpha \in \mathbb{F}_p^*$ aleatoriamente e $a \in G$;

(b) Bob calcula $b = \alpha \cdot a$ e envia o α-par (a, b) para Alice;

(c) Posteriormente, Alice envia um α-par (a', b') diferente para Bob;

(d) Bob aceita a resposta de Alice somente se (a', b') for α-par.

Uma vez que o problema do logaritmo discreto é difícil tanto para G quanto para $\mathbb{F}_p$, Alice não conhece α. Logo, uma das formas que produz uma resposta correta de Alice para Bob é Alice escolher $\gamma \in \mathbb{F}_p^*$ e retornar $(a', b') = (\gamma \cdot a, \gamma \cdot b)$. Assim, temos:

$$b' = \gamma \cdot b = \gamma \cdot \alpha \cdot a = \alpha \cdot (\gamma \cdot a) = \alpha \cdot a' \tag{2.4}$$

o que atesta o fato de (a', b') ser α-par e Alice conhecer o coeficiente γ tal que $a' = \gamma \cdot a$.

O conceito conhecimento de coeficientes adotado no teste é importante para a avaliação cega de polinômios de forma verificável, a qual será abordada em seções posteriores. Nesse contexto, duas pesquisas são importantes para compreender melhor como o conhecimento de coeficiente funciona: (KRAIEM et al., 2019) e (WU; STINSON,).

[3] Em (PETKUS, 2019), esse processo é chamado de deslocamento do valor encriptado.

Em (KRAIEM et al., 2019) o tema é abordado de forma matemática agregado a relação com grupos cíclicos. Já (WU; STINSON,) detalha o tema apresentando protocolos e algoritmos como exemplo.

No *KCA*, Bob envia um α-par (a, b) para Alice, a qual responde com um α-par (a', b') distinto. Todavia, caso Bob envie múltiplos α-par para Alice, é necessário modificar o *KCA* adotado para que Alice retorne uma resposta assertiva. Nesse caso, utiliza-se os múltiplos α-par enviados para gerar um único α-par como resposta. Sendo assim, pode-se definir uma interação entre Alice e Bob utilizando o *KCA* estendido da seguinte maneira:

(a) Bob escolhe $\alpha \in \mathbb{F}_p^*$ aleatoriamente e $a \in G$;

(b) Bob envia múltiplos α-par para Alice, os quais são da forma: $(a_1, b_1), ..., (a_d, b_d)$ para um mesmo α;

(c) Alice escolhe $c_1, c_2, ..., c_d \in \mathbb{F}_p$;

(d) Alice calcula um único α-par $(a', b') = (c_1 \cdot a_1 + ... + c_d \cdot a_d, c_1 \cdot b_1 + ... + c_d \cdot b_d)$;

(e) Alice envia o α-par calculado no passo anterior para Bob.

Em (c), desde que $\alpha \neq 0$, (a', b') é α-par pois:

$$\begin{aligned} b' &= c_1 \cdot b_1 + ... + c_d \cdot b_d \\ &= c_1 \cdot \alpha \cdot a_1 + ... + c_d \cdot \alpha \cdot a_d \\ &= \alpha(c_1 \cdot a_1 + ... + c_d \cdot a_d) \\ &= \alpha \cdot a' \end{aligned}$$

Formalmente, é possível definir o *KCA*, estendido da forma:

Definição 1 *Dado múltiplos α-par $(s^0 \cdot g, \alpha \cdot s^0 \cdot g), ..., (s^d \cdot g, s^d \cdot \alpha \cdot g)$, sendo $s^i \in \mathbb{F}_p$ e $\alpha \in \mathbb{F}_p^*$. Se Alice é capaz de calcular o α-par (a', b'), então, exceto com probabilidade insignificante, é possível afirmar que Alice conhece $c_0, ..., c_d \in \mathbb{F}_p$ tal que $a' = \sum_{i=0}^{d} c_i \cdot s^i \cdot g$.*

Definições e provas mais completas sobre o *KCA* estendido podem ser encontradas em (GROTH, 2010).

Na avaliação cega definida em seções anteriores, havia chances consideráveis de que Alice enviasse um valor aleatório e alegasse ser $E(P(s))$ de maneira mal intencionada. Logo, há necessidade de criar uma forma de verificar se o valor enviado por Alice é válido. Utilizando o *KCA* estendido, é possível atingir tal objetivo. Assim, para uma definição particular de $E(x)$ da forma $E(x) = x \cdot g$, temos:

(a) Bob escolhe, aleatoriamente, $\alpha \in \mathbb{F}_p^*$ e $\alpha \in G$;

(b) Bob envia $s^0 \cdot g, ..., s^d \cdot g$ (provenientes de $s^0, s^1, ..., s^d$)[4] e $\alpha \cdot s^0 \cdot g, ..., \alpha \cdot s^d \cdot g$ (provenientes de $\alpha \cdot s^0, \alpha \cdot s^1, ..., \alpha \cdot s^d$)[5] para Alice;

(c) Alice calcula $a = P(s) \cdot g$ e $b = \alpha \cdot P(s)$ utilizando as informações do passo anterior e envia o α-par (a, b) para Bob;

(d) Bob verifica se $b = \alpha \cdot a$ e, caso a igualdade seja mantida, aceita a resposta de Alice.

2.2.6 Circuitos aritméticos

Em seções anteriores, foram abordadas maneiras de lidar com polinômios em provas de conhecimento zero. Todavia, nem sempre uma informação é dada em forma de polinômio, logo, para usufruir dos conhecimentos detalhados anteriormente, é necessário um método que transforme a informação dada em polinômios.

O primeiro passo dessa transformação são os circuitos aritméticos. Supondo que Alice deseja provar para Bob que conhece $c_1, c_2, c_3 \in \mathbb{F}_p$ de modo que $(c_1 \cdot c_2) \cdot (c_1 + c_3) = 11$, temos que o circuito aritmético da informação a ser provada por Alice é representado pela Figura 2.2.6.

A construção do circuito aritmético da Figura 2.2.6 possui certas regras, tais como:

1. Quando uma conexão é direcionada para mais de uma porta, é considerado como uma única conexão (w_1 por exemplo);
2. Operações multiplicativas necessitam de duas conexões de entrada (PANKOVA, 2013);
3. Conexões que fluem de uma adição para uma multiplicação não são nomeadas, assim como a adição em si.

Para que o circuito seja considerado válido, é preciso que as saídas de cada multiplicação seja o produto das entradas correspondentes. Dessa forma, a informação a ser provada por Alice na forma de um circuito aritmético válido é $c_1, ..., c_5 \in \mathbb{F}_p$ de modo que $c_4 = c_1 \cdot c_2$, $c_5 = c_4 \cdot (c_1 + c_3)$ e c_5 deve ser o resultado esperado, ou seja, $c_5 = 11$. O próximo passo da transformação é reduzir essa informação a um polinômio utilizando uma técnica chamada *Quadratic Arithmetic Program (QAP)*, também conhecida como programa aritmético quadrado.

[4] Ressalta-se que $s^0 \cdot g, ..., s^d \cdot g = E(s^0), ..., E(s^d)$

[5] Ressalta-se que $\alpha \cdot s^0 \cdot g, ..., \alpha \cdot s^d \cdot g = E(\alpha \cdot s^0), ..., E(\alpha \cdot s^d)$

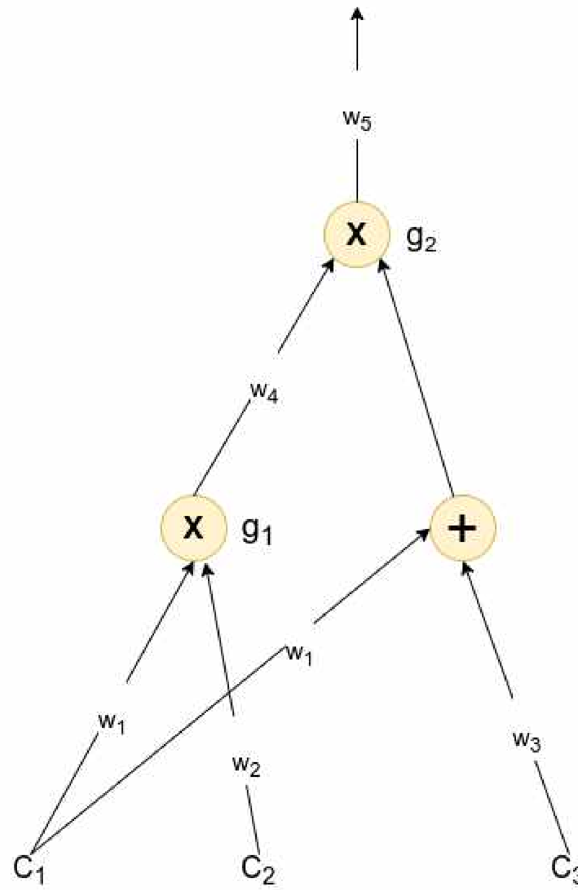

Figura 7 – Circuito Aritmético

2.2.7 Programa aritmético quadrado

Formalmente, QAP pode ser definido da seguinte forma (PANKOVA, 2013):

Definição 2 *Dado os inteiros m, n, k tal que $n-1 \geq k$, um programa aritmético quadrado sobre $\mathbb{F}_p$, denotado P(A,B,C), consiste em três matrizes $m \times n$ A,B,C sobre $\mathbb{F}_p$. Um vetor $x \in \mathbb{F}_p^k$ é aceito por P caso exista um vetor $w = (1, w_1, ..., w_{n-1})$ tal que $(w_1, ..., w_k) = x$ e $Aw \cdot Bw = Cw$.*

No contexto dessa pesquisa, A, B, C serão chamados de L, R, O e, respectivamente, nomeados como conexão polinomial à esquerda, direita e saída. Além disso, nos circuitos aritméticos, cada multiplicação é associada a um campo g_i distinto sendo $i \in \mathbb{F}_p$ e cada i é denominado ponto alvo. Na Figura 2.2.6, por exemplo, existem os campos g_1, g_2 e os pontos $\{1, 2\} \in \mathbb{F}_p$ são os pontos alvos.

Para determinar corretamente os polinômios que definem L, R e O considera-se polinômios que serão zerados nos pontos alvos, exceto nos pontos alvos que correspondem a multiplicação em questão. Seguindo o circuito aritmético da Figura 2.2.6, temos:

(a) Para g_1, w_1, w_2 e w_3 são as conexões polinomiais à esquerda, direita e saída respectivamente. Dessa forma, $L_1 = R_2 = O_4 = 2 - X$, visto que $2 - X$ é zero para o ponto alvo 2 e não zero para o ponto alvo 2;

(b) Para g_2, w_1 e w_3 são entradas à direita de g_2. Assim, $L_4 = R_1 = R_3 = O_5 = X - 1$, uma vez que $X - 1$ é zero para o ponto alvo 1 e não zero para o ponto alvo 2.

Assim, dado os valores $c_1, ..., c_5 \in \mathbb{F}_p$ do circuito aritmético válido de Alice, é possível obter L, R, O e P tal que (no caso de Alice, $m = 5$):

$$L = \sum_{i=1}^{m} c_i \cdot L_i \tag{2.5}$$

$$R = \sum_{i=1}^{m} c_i \cdot R_i \tag{2.6}$$

$$O = \sum_{i=1}^{m} c_i \cdot O_i \tag{2.7}$$

$$P = L \cdot R - O \tag{2.8}$$

Finalmente, para que o circuito seja considerado válido utilizando QAP, o valor de P é zero para todos os pontos alvos.

Verificando $P = 0$ para o ponto alvo $1 \in \mathbb{F}_p$, temos:

(a) Para as conexões polinomiais à esquerda, L, somente L_1 é diferente de zero no ponto alvo 1, especificamente, $L_1 = 1$. Logo, $L(1) = c_1 \cdot L_1 = c_1$;

(b) Para as conexões polinomiais à direita, R, somente R_2 é diferente de zero no ponto alvo 1, especificamente, $R_2 = 1$. Logo, $R(1) = c_2 \cdot R_2 = c_2$;

(c) Para as conexões polinomiais à saída, O, somente O_4 é diferente de zero no ponto alvo 1, especificamente, $O_4 = 1$. Logo, $O(1) = c_4 \cdot O_4 = c_4$;

(d) Pelos itens anteriores, $P(1) = c_1 \cdot c_2 - c_4 = 0$.

Já para o ponto alvo $2 \in \mathbb{F}_p$, temos:

(a) Para as conexões polinomiais à esquerda, L, somente L_4 é diferente de zero no ponto alvo 2, especificamente, $L_4 = 1$. Logo, $L(2) = c_4 \cdot L_4 = c_4$;

(b) Para as conexões polinomiais à direita, R_1 e R_3 são diferentes de zero no ponto alvo 2, especificamente, $R_1 = R_3 = 1$. Logo, $R(2) = c_1 \cdot R_1 + c_3 \cdot R_3 = c_1 + c_3$;

(c) Para as conexões polinomiais à saída, somente O_5 é diferente de zero no ponto alvo 2, especificamente, $O_5 = 1$. Logo, $O(2) = c_5 \cdot O_5 = c_5$;

(d) Pelos itens anteriores, $P(2) = c_4 \cdot (c_1 + c_3) - c_5 = 0$.

Visto que $P = 0$ para todos os pontos alvos, conclui-se que $c_1, ..., c_5 \in \mathbb{F}_p$ é uma atribuição válida para circuitos aritméticos se, e somente se, $P = 0$ para todos os pontos alvos.

Ademais, é possível definir QAP utilizando polinômios alvos. Na informação fornecida por Alice, o polinômio alvo T a ser definido é $T(X) = (X - 1) \cdot (X - 2)$ e, assim, T divide P se, e somente se, $c_1, ..., c_5 \in \mathbb{F}_p$ é uma atribuição válida para o circuito aritmético. Formalmente, para um polinômio P e um ponto $a \in \mathbb{F}_p$, $P(a) = 0$ se, e somente se, o polinômio $X - a$ divide P, ou seja, $P = (X - a) \cdot H$ para algum polinômio H.

Ao utilizar QAP, garante-se que a probabilidade de que Alice gere L, R, O, P por meio de uma atribuição inválida de $(c_1, ..., c_m)$ é pequena, visto que, de acordo com o Teorema Fundamental da Álgebra, um polinômio de grau d tem, no máximo, d soluções e, consequentemente, compartilha no máximo d pontos. Dessa forma, dado $s \in \mathbb{F}_p$ e $P(s) = H(s) \cdot T(s)$, se p for muito maior do que $2d$, a probabilidade de $P(s) = H(s) \cdot T(s)$ para um $s \in \mathbb{F}_p$ escolhido aleatoriamente é muito pequena.

Contudo, mesmo que seja garantido que Alice não consegue obter L, R, O, P por meio de uma atribuição inválida de $(c_1, ..., c_5)$, ainda é possível que Alice construa L, R, O, H arbitrariamente e $L \cdot R - O = T \cdot H$. Para impedir que isso ocorra, combinações lineares são utilizadas da seguinte maneira:

(a) Combina-se L, R, O em um único polinômio F tal que $F = L + X^{d+1} \cdot R + X^{2(d+1)} \cdot O$ [6];

(b) Combina-se os polinômios usando QAP de modo que $F_i = L_i + X^{d+1} \cdot R_i + X^{2(d+1)} \cdot O_i$ para $i \in \{1, ..., m\}$.

Em (a) e (b), considerando $F = \sum_{i=1}^{m} c_i \cdot F_i$ para $(c_1, ..., c_m)$, então temos 2.5, 2.6 e 2.7 para os mesmos $(c_1, ..., c_m)$. Portanto, caso F seja uma combinação linear dos F_i, então L, R, O foram construídos a partir de uma atribuição válida do circuito aritmético.

2.2.8 Protocolo Pinóquio

Nesse estudo, a fim de construir o protocolo *zk-SNARK* final, aspectos do protocolo de Pinóquio serão empregados. O protocolo Pinóquio define um sistema construído para verificar eficiência de cálculos gerais fundamentando-se somente em suposições criptográficas. Ao utilizar essa técnica, uma chave pública é criada na etapa que antecede a

[6] Os coeficientes foram divididos de maneira que L, R, O referenciassem coeficientes distintos. Assim, $1, X, ..., X^d$ são coeficientes de L, $X^{d+1}, ..., X^{2d+1}$ são de R e de $X^{2(d+1)}$ pertencem a O.

interação entre as entidades e, posteriormente, as partes envolvidas no protocolo são capazes de realizar cálculos a partir das entradas fornecidas e checarem a veracidade de tais cálculos (PARNO et al., 2016). Dada essa definição e o propósito de adequar os conceitos desse estudo com o de Pinóquio, dois pontos precisam ser adicionadas ao protocolo dessa pesquisa:

(i) Ocultação homomórfica que suporta adição e multiplicação;

(ii) Característica de não iteratividade.

No ponto (i), é necessário definir uma ocultação homomórfica que suporta tanto operações aditivas quanto multiplicativas, o que pode ser alcançado combinando curvas elípticas e emparelhamento de Tate. Essencialmente, dado um número primo p ($p > 3$), uma curva elíptica $\mathcal{C}$ é formada por um conjunto de pontos $(x, y) \in \mathbb{F}_p^2$ que seguem duas condições (GATHEN, 2015). A primeira baseia-se em dado $u, v \in \mathbb{F}_p$, o seguinte é atendido:

$$4u^3 + 27v^2 \neq 0 \tag{2.9}$$

Já a segunda consiste em obter tais pontos (x, y) por meio da seguinte equação:

$$Y^2 = X^3 + u \cdot X + v \tag{2.10}$$

A importância de tal definição para os fins dessa pesquisa é a possibilidade de gerar pontos a partir da interseção de retas em curvas elípticas. Para operações aditivas, dado $P, Q \in \mathcal{C}$, é possível obter R de modo que $P + Q + R = \mathcal{O}$, onde $\mathcal{O}$ é considerado o zero do grupo. A Figura 2.2.8 é a representação gráfica dessa operação.

Como 2.10 possui grau 3 em X e há dois pontos P e Q compartilhados com a reta, é garantido que também intersecta a reta em um terceiro ponto R e em nenhum outro. Além disso, uma vez que os pontos $(x, y) \in \mathbb{F}_p^2$, o grupo formado por pontos da curva $\mathcal{C}$ com coordenadas em $\mathbb{F}_p$ é denominado $\mathcal{C}(\mathbb{F}_p)$.

Todavia, ainda que isso seja suficiente para operações aditivas, etapas adicionais são necessárias quando as operações são multiplicativas. Uma dessas etapas é definir o grau de incorporação k de $\mathcal{C}$, o que pode ser feito assumindo-se que o número primo r ($r \neq p$) representa a quantidade de elementos em $\mathcal{C}(\mathbb{F}_p)$ e k é o menor inteiro possível[7] tal que $p^k - 1$ é divisível por r. Utilizando o grau de incorporação, é possível definir $\mathcal{C}(\mathbb{F}_{p^k})$, o qual consiste em um grupo formado por pontos da curva $\mathcal{C}$ com coordenadas em $\mathbb{F}_{p^k}$

[7] Se k não for um número muito pequeno, então o problema do logaritmo discreto é difícil em $\mathcal{C}(\mathbb{F}_p)$.

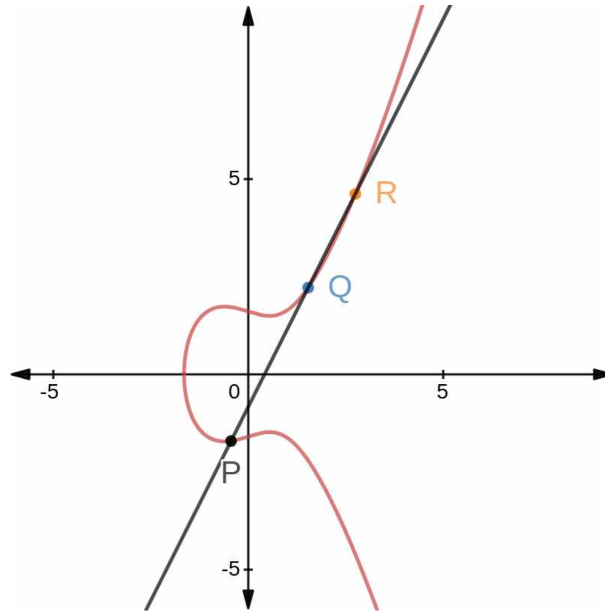

Figura 8 – Curva Elíptica

e, em virtude da definição de grau de incorporação, $\mathcal{C}(\mathbb{F}_p) \subset \mathcal{C}(\mathbb{F}_{p^k})$, ou seja, $\mathcal{C}(\mathbb{F}_p)$ é um subconjunto de $\mathcal{C}(\mathbb{F}_{p^k})$[8].

Após definir o grau de incorporação e, consequentemente, os subgrupos relacionados ao mesmo, utiliza-se o emparelhamento de Tate. Para fins desse estudo, o emparelhamento de Tate é usado como uma função de mapeamento de um par (j, h) proveniente de subgrupos de $\mathcal{C}(\mathbb{F}_{p^k})$ em grupo denominado G_T. Assim, assumindo que G_1 (equivalente a $\mathcal{C}(\mathbb{F}_p)$) e G_2 são subgrupos de $\mathcal{C}(\mathbb{F}_{p^k})$, dado $j \in G_1$ e $h \in G_2$, temos:

1. $Tate(j, h) = g \ (g \in G_T)$;
2. Dado $a, b \in \mathbb{F}_r$, $Tate(a \cdot j, b \cdot h) = g^{ab}$.

O emparelhamento de Tate não é o foco desse estudo. Contudo, é importante pontuar que para $a \in \mathbb{F}_p$, o polinômio $(X - a)^r$ possui uma raiz de ordem r em a e nenhuma outra raiz. Logo, dado $P \in G_1$ e uma função f_P da curva para $\mathbb{F}_p$, então existe uma raiz de ordem r em P e nenhuma outra raiz. Portanto, para P, Q define-se a função de emparelhamento de Tate $Tate(P, Q)$ como $f_P(Q)^{p^k - 1}/r$. Os estudos (GALBRAITH; HARRISON; SOLDERA, 2002) e (KOBAYASHI; AOKI; IMAI, 2006) abordam o emparelhamento de Tate com maiores detalhes assim como algoritmos relacionados ao mesmo.

Com essas definições, é possível obter uma ocultação homomórfica que suporta adição e multiplicação. É fundamental ressaltar que essa ocultação é classificada como

[8] É relevante ressaltar que, além de $\mathcal{C}(\mathbb{F}_p)$, $\mathcal{C}(\mathbb{F}_{p^k})$ contém outros $r - 1$ subgrupos de ordem r, sendo G_T um desses subgrupos, o qual está relacionado ao emparelhamento de Tate.

fraca, uma vez que dado $E(x) = x$, $E_1(x) = j \cdot x$ e $E_2(x) = h \cdot x$ seria possível determinar $E(xy)$ a partir de $E_1(x)$ e $E_2(y)$, mas não seria possível obter $E(xyz)$.

Já o ponto (ii) requer a inclusão de uma característica não atingida até o momento: não iteratividade. Isso significa que, no protocolo abordado nesse estudo, é necessário que em uma situação onde o provador afirma algo, outras entidades envolvidas no protocolo sejam capazes de verificar a afirmação por si (KOENS; RAMAEKERS; WIJK, 2017). As vantagens dessa característica é prover transferibilidade ao protocolo, uma vez que a confiabilidade do resultado não será restrita a apenas duas entidades (como Alice e Bob), mas sim a várias partes envolvidas. Para atingir tal objetivo, uma fase inicial denominada *setup* é realizada antes de iniciar qualquer protocolo de provas de conhecimento zero e, nessa etapa, é gerada uma sequência de caracteres obtidos aleatoriamente, a qual é conhecida como *Common Reference String (CRS)* ou sequência de referência comum (PETKUS, 2019).

Assim, baseando-se nos conceitos abordados anteriormente, na situação em que Alice deseja provar para Bob que conhece *x,y* tais que *x + y = 11*, é possível elaborar um protocolo *zk-SNARK* utilizando conceitos do protocolo de Pinóquio.

Primeiramente, é necessário definir a fase *setup*, a qual pode ser feita da seguinte maneira:

(a) Define-se, aleatoriamente, $\alpha \in \mathbb{F}_r^*$ e $s \in \mathbb{F}_r$;

(b) A *CRS* é da forma: $(E_1(s^0), E_1(s^1), ..., E_1(s^d), E_2(\alpha s^0), E_2(\alpha s), ..., E_2(\alpha s^d))$;

(c) A *CRS* é publicada.

Em seguida, temos o protocolo em si:

(A) Provador: Alice calcula $a = E_1(P(s))$ e $b = E_2(\alpha P(s))$ usufruindo da *CRS* e o suporte às combinações lineares da ocultação homomórfica;

(B) Verificador: Dado $x, y \in \mathbb{F}_r$ de modo que $a = E_1(x)$ e $b = E_2(y)$, Bob é capaz de calcular $E(\alpha x) = Tate(E_1(x), E_2(\alpha))$ e $E(y) = Tate(E_1(1), E_2(y))$;

(C) Se no passo anterior do verificador $E(\alpha x) = E(y)$, então $\alpha x = y$ e Bob confirma que Alice realmente retém a informação afirmada.

2.3 Computação Multiparte Segura

As primeiras aparições do conceito de computação segura foram nas pesquisas desenvolvidas pelo cientista Andrew Chi-Chih Yao por meio da apresentação de uma solução para o Problema dos Milionários (YAO, 1982). O problema dos milionários consiste

em, dado m milionários, determinar qual é o mais rico sem revelar a riqueza que cada um possui. Em (YAO, 1982), Yao propõe uma generalização do problema dos milionários por meio de uma função $f(x_1, x_2, ..., x_m)$, a qual recebe como parâmetro m inteiros e retorna o resultado da computação dessa função. Especificamente sobre o problema dos milionários, considerando $m = 2$, a função $f(x_1, x_2)$ seria responsável por definir se $x_1 > x_2$, determinando, assim, qual milionário retém maior riqueza.

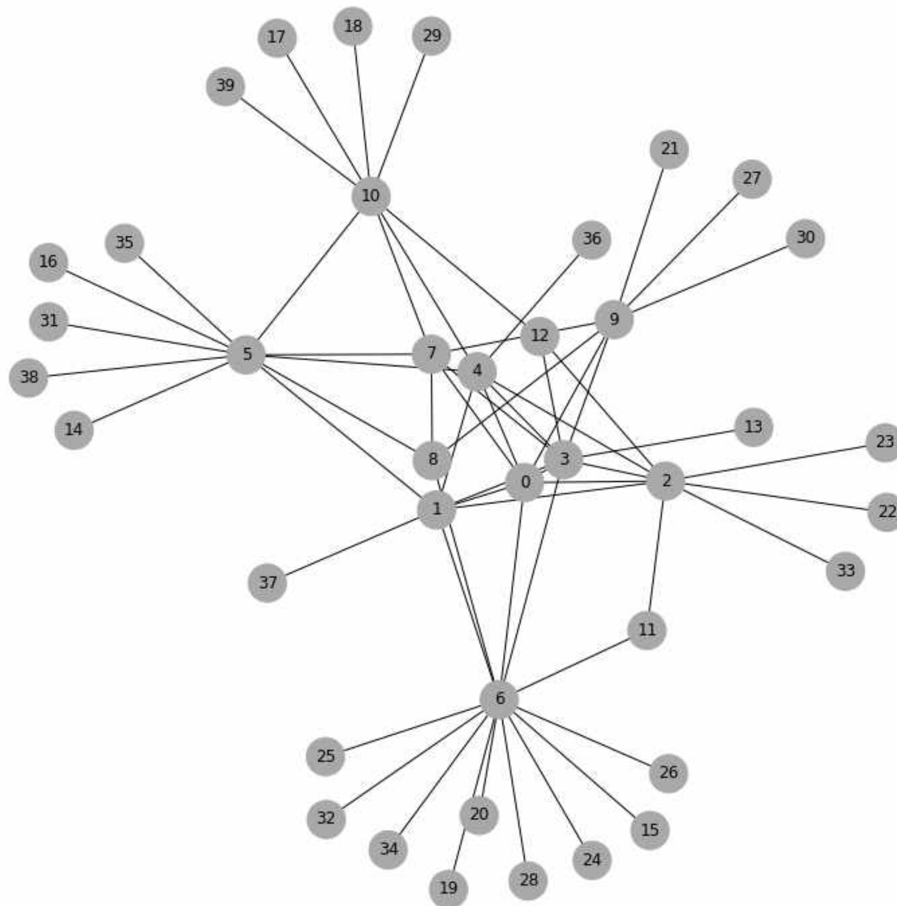

Figura 9 – Grafo G'

É possível expandir a discussão acerca de computação segura para os algoritmos de coloração de grafo mencionados na seção 2.2. Para tal, a seguinte situação é proposta: tanto Alice quanto Bob conhecem algoritmos capazes de colorir grafos e precisam colorir o grafo G' (Figura 9). Entretanto, o grafo G' possui vários nós e arestas, logo, seria necessário uma espera maior para concluir sua coloração. Devido a isso, Alice e Bob decidem unir-se e realizar a coloração de G' juntos sob a condição de que o funcionamento de seus respectivos algoritmos de coloração não fosse descoberto. Para tal, o seguinte processo foi utilizado:

(a) Alice e Bob definem a divisão do grafo G', sendo que os nós verdes serão coloridos

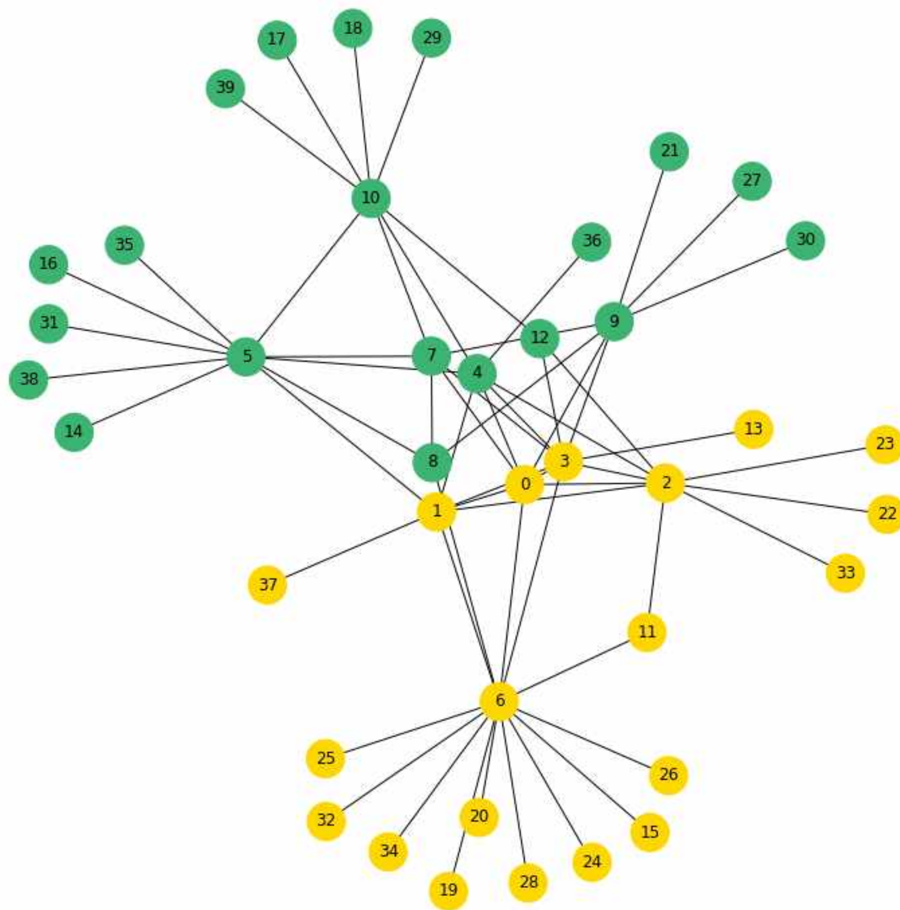

Figura 10 – Divisão do grafo G' entre Alice e Bob

por Alice e os amarelos por Bob (Figura 10);

(b) Alice e Bob colorem suas respectivas partes;

(c) Alice e Bob verificam o resultado (Figura 11) .

Ainda que o processo supracitado possibilite a coloração do grafo unificando as técnicas de Alice e Bob, é necessário definir como cada um demonstrará que coloriu seus nós corretamente. Nesse contexto, a computação segura poderia ser utilizada de modo que uma função f fosse definida para verificar se o grafo G' foi colorido corretamente. Uma possível função f seria a $f(x_1, x_2)$, na qual o resultado indicaria se o grafo foi colorido adequadamente e os parâmetros de entrada x_1, x_2 representariam, respectivamente, os nós coloridos por Alice e Bob. Além disso, é imprescindível aplicar técnicas relacionadas à computação segura que impedem a revelação dos dados de entrada de cada participante do processo, ou seja, é necessário proteger x_1, x_2 de modo que Alice e Bob não sejam capazes de descobrir aspectos relacionados ao algoritmo de coloração um do outro. Tais técnicas serão discutidas nas próximas seções.

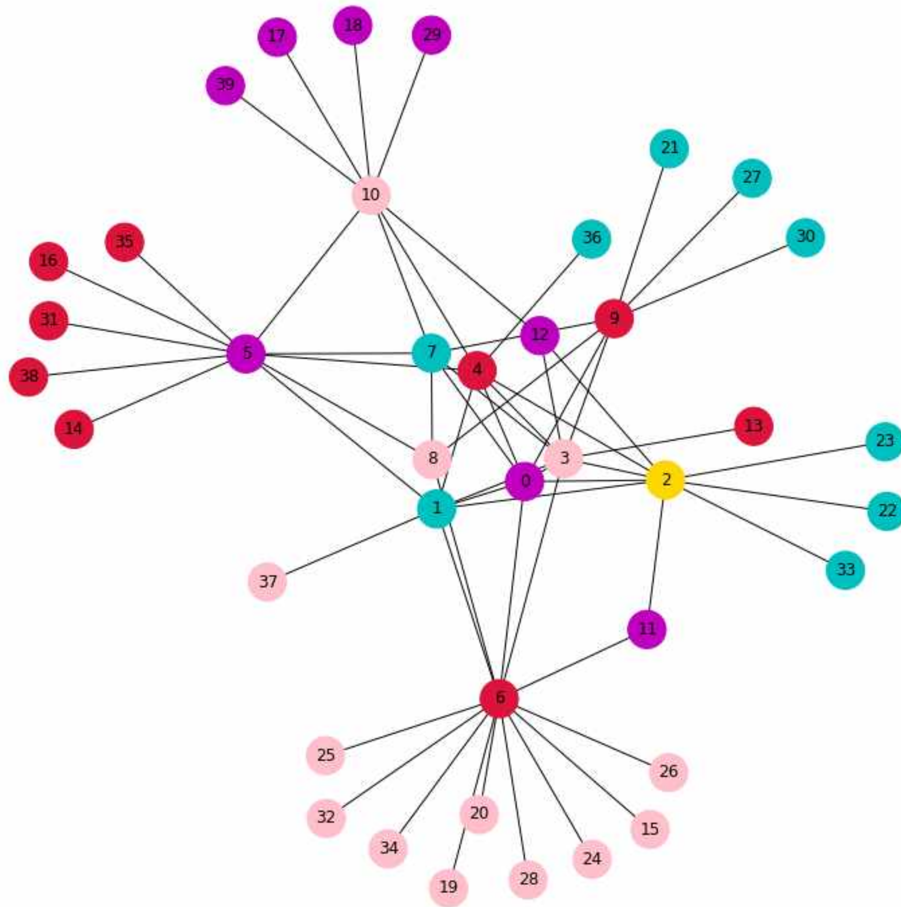

Figura 11 – Grafo G' colorido

2.3.1 Definição

É possível definir *Secure multi-party computation* como um método que permite que n participantes não confiáveis realizem a computação de uma função pré estabelecida de forma segura, sendo que cada entidade no processo informa dados de entrada. No contexto do SMC, segurança está relacionado a atingir o resultado esperado do cálculo da função e a capacidade de manter os dados de entrada de cada um dos participantes privados mesmo em casos onde alguma entidade aja de maneira mal intencionada (FITZI; HIRT; MAURER, 1998).

A fim de permitir que tal computação seja realizada entre as entidades não confiáveis, funções de sentido único são utilizadas. Inicialmente proposta em (DIFFIE; HELLMAN, 1976), uma função é caracterizada como de sentido único se é facilmente calculada, mas sua inversão é considerada difícil. Formalmente, sendo f_k uma família de funções de bits de string e $f_k : \{0,1\}^k \to \{0,1\}^k$, a família f_k será caracterizada como de sentido único se duas condições forem atendidas (FRANKLIN, 1994):

(a) Para toda $f_i \in f_k$, f_i é calculada em tempo polinomial determinístico relacionado a

k;

(b) Para todo algoritmo de tempo polinomial A e para todo c existe um k_c de modo que o seguinte seja atendido para todo $k > k_c$:

$$prob(f_k(A(f_k(x))) = f_k(x)|x \leftarrow \{0,1\}^k) < k^{-c} \tag{2.11}$$

Utilizando o conceito de funções de sentido único, em (YAO, 1982), Yao propõe um protocolo que soluciona o problema dos milionários. Dada as seguintes definições:

(i) Alice possui i milhões, Bob j milhões e $1 < i, j < 10$;

(ii) M é o conjunto de todos os inteiros não negativos de N-bits;

(iii) Q_N é o conjunto de todas as funções 1-1 de M para M;

(iv) E_a é a chave pública de Alice, a qual foi gerada por meio da escolha aleatória de uma função presente em Q_N;

(v) D_a é a função inversa de E_a.

Finalmente, temos o protocolo definido por Yao:

1. Bob seleciona, aleatoriamente, um inteiro de N-bits e calcula $k = E_a(x)$;

2. Bob envia $k - j + 1$ para Alice;

3. Alice calcula os valores $y_u = D_a(k - j + u)$ para $u = 1, 2, ..., 10$;

4. Alice gera, aleatoriamente, um primo p que possui $N/2$ bits e, posteriormente, calcula $z_u = y_u \pmod p$ para todos os u definidos previamente. Todos os z_u calculados devem diferenciar-se por pelo menos 2 unidades, caso contrário esse passo é repetido;

5. Alice envia p, $z_1, ..., z_10$ e $z_i + 1, ..., z_10 + 1$ para Bob, sendo que todos esses números foram previamente submetidos a $\pmod p$;

6. Descartando p, Bob verifica o número na posição j. Se esse número for igual a $x \pmod p$, então $i \geq j$. Caso contrário $i < j$.

Além das etapas supracitadas, Yao também detalha a demonstração do porquê o protocolo funciona no estudo (YAO, 1982). Posteriormente, em (YAO, 1986), Yao ainda acrescenta propriedades que adicionam equidade no protocolo, de modo a permitir que todos os n participantes do processo aprendam a saída da computação simultaneamente.

2.3.2 Aplicações

Conceitos relacionados a SMC podem ser utilizados para solucionar diversos problemas, tais como: Problema dos Milionários, Problema do Voto Digital, Problema do Pôquer Mental e dados como serviço.

O Problema dos Milionários, previamente discutido na presente pesquisa, consiste em determinar, dentre m milionários, qual detém a maior riqueza sem revelar a riqueza de cada um (YAO, 1982). Já o Problema do Voto Digital consiste em permitir que qualquer indivíduo interessado verifique as cédulas depositadas em uma dada eleição sem que o voto dos participantes seja revelado (CHAUM, 1981) (MICALI; ROGAWAY, 1991). Em contrapartida, o Problema da Moeda Lançada é definido pela possibilidade de que entidades consigam concordar com o resultado do lançamento de uma moeda ainda que uma das partes possa agir de maneira mal intencionada (BLUM, 1983) (MICALI; ROGAWAY, 1991). Finalmente, questões relacionadas a dados como serviço, do termo em inglês *Data as a Service (DaaS)*, consistem em permitir que dados sejam acessados como um recurso de algum serviço, sendo que é necessário proteger dados privados e ainda assim permitir o acesso (ARCHER et al., 2018).

2.4 Trabalhos relacionados

Trabalhos semelhantes já foram desenvolvidos com o objetivo de solucionar lacunas em termos de privacidade de dados relacionados aos objetos de estudo dessa pesquisa. Em relação a provas de conhecimento zero, as pesquisas (MIERS, 2019) e (BÜNZ et al., 2017) são interessantes. Já em termos de *blockchain*, as pesquisas (BENHAMOUDA; HALEVI; HALEVI, 2019) e (ZOU; LV; ZHAO, 2020) apresentam soluções viáveis. Finalmente, os estudos (UNTERWEGER et al., 2018), (BAUMANN et al., 2020) e (SÁNCHEZ, 2018) conjuram conceitos de *smart contracts* e obtenção de privacidade de dados.

Em (MIERS, 2019) a proposta é uma maneira de prover a geração de parâmetro do *zk-SNARK* de forma escalável e aplicável em aplicações do mundo real. Para tal, o autor apresenta um sistema baseado no *zk-SNARK* previamente desenvolvido por Jens Groth em (GROTH, 2016) com acréscimo de um processo amortizado para geração da *Common Reference String*.

Em (BÜNZ et al., 2017) um novo método baseado em *non-interactive zero-knowledge* é desenvolvido, o qual permite que as provas sejam sucintas e sem a necessidade de um *setup* confiável. Esse método basea-se na suposição do logaritmo discreto e não interatividade provida pela heurística Fiat-Shamir e é considerado ideal para provas de intervalo[9] e embaralhamento verificável.

[9] Embora a tradução literal seja provas de alcance, o presente estudo adota provas de intervalo para referenciar as denominadas *range proofs*.

Em relação a *blockchain*, a pesquisa (BENHAMOUDA; HALEVI; HALEVI, 2019) apresenta uma arquitetura baseada em *blockchain* é desenvolvida. Essa arquitetura é denominada *Hyperledger Fabric* e conjura conceitos relacionados a *blockchain* e SMC para apresentar uma solução que adicione suporte a dados privados em uma rede descentralizada. Para tal, a arquitetura suporta *smart contracts* por meio de *chaincodes* executados pelos nós da rede, sendo que esses *chaincode* são capazes de acessar a rede a fim de decidir se um dado deve ser armazenado.

Ainda sobre *blockchain*, o estudo (ZOU; LV; ZHAO, 2020) propõe solucionar problemas frequentemente encontrados em sistemas de medicina eletrônica também conhecidos como *eHealth*. Nesse estudo, um sistema baseado em *blockchain* é proposto de modo a solucionar questões relacionadas à segurança e disponibilidade de dados em redes descentralizadas específicas no setor da saúde. Para isso, os autores usufruem de funções *hash* baseadas e RepuCoin.

Especificamente sobre *smart contracts*, (UNTERWEGER et al., 2018) é um estudo analítico que apresenta as adversidades e conclusões relacionadas a implementação de *smart contracts* que objetivam prover privacidade de dados na Ethereum. Nessa pesquisa, os autores implementam um contrato utilizando primitivas criptográficas e suposições matemáticas na tentativa de prover privacidade de dados em uma rede descentralizada e, posteriormente, apresentam resultados analíticos relacionados à implementação.

Já em (BAUMANN et al., 2020) é apresentada a nova versão de um sistema denominado *zkay v0.2* que implementa privacidade de dados em *smart contracts*. Nesse sistema, existe suporte para criptografia assimétrica e híbrica, assim como suporte para novas *features* da linguagem adotada.

Finalmente em (SÁNCHEZ, 2018) é proposta uma maneira efetiva de solucionar ataques como o DAO e Gyges por meio de uma implementação que permite prover características de privacidade e verificabilidade aos *smart contracts*. Nessa implementação, o autor beneficia-se de conceitos relacionados ao SMC e provas de conhecimento zero para desempenhar funcionalidades tais como permitir a verificação de *smart contracts* e reuso de parâmetros necessários para a computação.

3 Desenvolvimento

O presente capítulo aborda detalhes acerca do desenvolvimento da biblioteca pretendida. A seção 3.1 descreve como o protocolo funciona e apresenta o comportamento da biblioteca nos cenários honestos e desonestos. Já a seção 3.2 apresenta a implementação do contrato em si. Finalmente, a seção 3.3 apresenta um cenário no qual a solução proposta na pesquisa é aplicado.

3.1 Protocolo

No capítulo 2 dois protocolos criptográficos foram apresentados. Ao longo do desenvolvimento da pesquisa, ambos os protocolos foram testados e optou-se pela utilização do SMC devido a facilidade de implementação e menores custos de gás associados à sua execução.

Nesse contexto, o protocolo apresentado utiliza conceitos de *Secure multi-party computation* baseado nas definições apresentadas em (CHAUM; DAMGÅRD; GRAAF, 1988), *smart contracts* e Ethereum. Para fins explicativos, o protocolo é dividido em duas etapas: ofuscamento e execução.

3.1.1 Ofuscamento

No protocolo, a etapa de ofuscamento é utilizada tanto por α quanto por β a fim de manter uma fração dos dados utilizados na computação privados. Nesse contexto, suponha que α deseja ofuscar uma dada tabela T, então os seguintes passos serão executados:

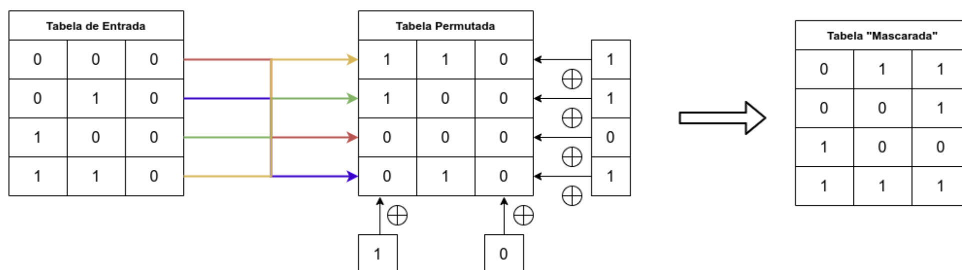

Figura 12 – Etapa de Ofuscamento

(a) Efetua-se uma permutação aleatória nas linhas de T;

(b) Em seguida, realiza-se a inversão de colunas, a qual consiste em aplicar a operação XOR utilizando duas colunas de T e dois bits b_1, b_3 escolhidos pelo participante α;

(c) Finalmente, realiza-se a criptografia da coluna de saída, a qual consiste em aplicar a operação XOR utilizando as entradas da coluna de saída e quatro bits d_1, d_2, d_3, d_4 escolhidos pelo participante α.

É importante ressaltar que a etapa de ofuscamento é realizada localmente por cada um dos participantes e como resultado gera uma tabela "mascarada", a qual será utilizada em etapas posteriores da computação. Além disso, uma vez que essa etapa não é executada pela biblioteca, técnicas para comprovar que a tabela resultante foi construída corretamente são necessárias.

3.1.2 Execução

Após a etapa de ofuscamento, ocorre a execução. Essa etapa é responsável por efetuar o *commit* das tabelas produzidas pela etapa de ofuscamento e, posteriormente, armazenar esses *commits* no *smart contract*. Nesse contexto, suponha que as entidades α e β proponham-se a efetuar a computação em conjunto usufruindo da biblioteca e α é responsável por iniciar o processo. Dessa forma, os seguintes passos serão adotados por α na etapa de execução:

(a) Localmente, na etapa de ofuscamento, α gera a tabela T' à partir da tabela T;

(b) Em seguida, α compromete-se com T' criando um *commit*. Esse *commit* consiste em um *hash* gerado pelo algoritmo *SHA-256* cujos argumentos para sua construção são os bits da inversão de coluna e criptografia da coluna de saída e uma *string* definida pela entidade α para ser utilizada como *nonce*;

(c) Após gerar o *commit*, α envia o *commit* e a tabela T' para o *smart contract*.

Assim que α finalizada suas etapas, β é capaz de continuar o processo. Para tal, β adota os seguintes passos:

(a) β recebe a tabela T' enviada por α por intermédio do *smart contract*. Localmente, assim como α, na etapa de ofuscamento, β gera a tabela T'' à partir da tabela T';

(b) Posteriormente, β compromete-se com T'' criando um *commit* de forma semelhante à α;

(c) Após gerar o *commit*, β também envia o *commit* e a tabela T'' para o *smart contract*.

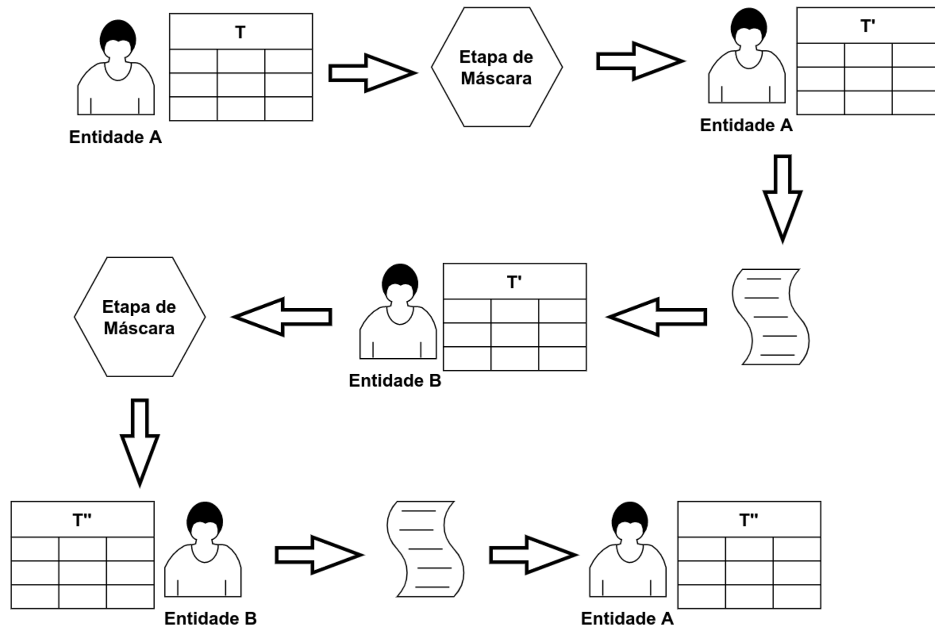

Figura 13 – Execução do protocolo proposto

Finalmente, após receber a tabela T'', o *smart contract* é capaz de calcular o resultado final da computação.

Entretanto, esse processo corresponde a execução honesta do protocolo e, dada a natureza descentralizada da plataforma Ethereum, não é possível garantir que os participantes agirão honestamente. Em virtude disso, a biblioteca possui mecanismos para permitir que os participantes envolvidos consigam comprovar que os dados foram gerados de forma correta.

3.1.3 Execução Desonesta

Durante a computação entre α e β, os seguintes cenários elencam possíveis execuções desonestas dos participantes:

(i) Suponha que a tabela T' resultante após etapa de ofuscamento seja construída utilizando b_1, b_3 como bits para inversão de colunas e d_1, d_2, d_3, d_4 como bits para criptografia da coluna de saída. Em um cenário honesto, o participante geraria um *commit* C' utilizando os valores mencionados anteriormente. Todavia, o participante decide utilizar b_4, b_5 como bits de inversão de colunas (sendo $b_1, b_3 \neq b_4, b_5$) na geração do *commit* C', ocasionando, assim, em uma execução desonesta;

(ii) De forma semelhante ao cenário apresentado em (i), o participante opta por utilizar d_5, d_6, d_7, d_8 como bits de criptografia da coluna de saída (sendo $d_5, d_6, d_7, d_8 \neq$

d_1, d_2, d_3, d_4) na geração do *commit* C', ocasionando, assim, em uma execução desonesta;

(iii) Suponha gere T' corretamente, informe os bits de inversão de coluna e criptografia da coluna de saída válidos e utilize a *string "meuNonce"* como *nonce* na geração do commit C'. Todavia, durante uma eventual verificação do processo, o participante decide informar o *nonce "outroNonce"*, acarretando, desse modo, em uma verificação desonesta;

(iv) De forma semelhante ao cenário apresentado em (iii), suponha que o participante utilize um *nonce*, b_1, b_3 como bits de inversão de coluna e $d_1, d_2, d_3 e d_4$ como bits de criptografia da coluna de saída para gerar o *commit* C'. Entretanto, durante uma eventual verificação do processo, o participante decide informar outros valores para os bits de inversão de coluna ou criptografia da coluna de saída, acarretando, assim, em uma verificação desonesta.

Os cenários (i) e (ii) podem ser prevenidos por meio da reconstrução das tabelas na etapa de ofuscamento. Assim, dado T antes do processo de ofuscamento, por meio dos bits de inversão de coluna e criptografia de coluna de saída informados, é possível construir uma nova tabela T' e verificar se essa tabela corresponde à tabela do *commit* gerado. Caso não corresponda, isso implica na detecção de uma execução desonesta.

Já em relação aos cenários (iii) e (iv), utiliza-se características intrínsecas de *hash* para mitigá-los. Isso é possível devido ao fato de que na geração de *hash* pequenas alterações nos argumentos acarretam em mudanças drásticas no resultado (MERKLE, 1990), ou seja, alterar um único *char* do *nonce* ou bit da inversão de colunas acarretaria em um *hash* distinto. Dessa forma, durante o processo de verificação, seria constatado que o *hash* informado não corresponde ao *commit* armazenado no *smart contract*, implicando, assim, na detecção de uma execução desonesta.

3.2 Implementação

O protocolo foi implementado utilizando Solidity, Python, Brownie e Ganache. O Brownie é um *framework* para desenvolvimento e teste de *smart contracts* baseado em Python (BROWNIE, 2020). Já o Ganache simula o comportamento da Ethereum, facilitando, assim, o desenvolvimento, *deploy* e testes de *smart contracts* (SUITE, 2020).

As próximas seções apresentam uma visão geral da implementação dos contratos, sendo o código completo disponibilizado em (SILVA, 2020).

3.2.1 Contrato

A principal estrutura da biblioteca é divida em dois contratos: *CommitHandler.sol* e *SMC.sol*. O contrato *CommitHandler* é uma biblioteca Solidity, a qual não é capaz de armazenar informações ou *ether*. Sua principal atribuição é definir funções básicas, as quais serão utilizadas no contrato *SMC*. Dentre as principais funcionalidades definidas nessa biblioteca, temos:

(a) Estrutura do *commit*

```
struct Commit {
    address owner;
    address previous;
    bytes32 commit;
    bool[3][4] truthTable;
}
```

Código 3.1 – Estrutura do *commit*

O *commit* consiste de quatro campos: *owner, previous, commit* e *truthTable* cujos significados são, respectivamente, endereço do dono do *commit*, endereço do dono do *commit* anterior, o *commit* gerado e a tabela referente à computação. É importante ressaltar que o segundo campo, o qual corresponde ao dono do primeiro *commit*, só é diferente do valor do campo *owner* caso seja o segundo *commit* do processo.

(b) Geração do *commit* inicial

```
function generate(bytes32 commit, bool[3][4] memory truthTable)
    public view returns(Commit memory) {
    address owner = msg.sender;
    address previousCommitOwner = msg.sender;
    return Commit(owner, previousCommitOwner, commit, truthTable);
}
```

Código 3.2 – Geração do *commit* inicial

Essa função é responsável por gerar o primeiro *commit* da computação. Para tal, recebe como parâmetro o *commit* e a tabela referente.

(c) Geração do *commit* final

```
function generate(bytes32 commit, address previousCommitOwner, bool
    [3][4] memory truthTable) public view returns(Commit memory) {
    address owner = msg.sender;
    return Commit(owner, previousCommitOwner, commit, truthTable);
}
```

Código 3.3 – Geração do *commit* final

Essa função é responsável por gerar o segundo *commit* da computação. Para tal, recebe como parâmetro do *commit*, o endereço do dono do primeiro *commit* e a tabela referente.

(d) Verificação de *commit*

```
1 function verify(bytes32 value, bytes memory nonce, bytes memory
     inversion_bits, bytes memory encryption_bits) public pure
     returns(bool) {
2      bytes32 generatedValue = sha256(abi.encodePacked(nonce,
          inversion_bits, encryption_bits));
3      return value == generatedValue;
4 }
```

Código 3.4 – Verificação do *commit*

Essa função é responsável por verificar se o *hash* do *commit* informado corresponde aos valores utilizados na permutação da tabela. Para tal, recebe como parâmetro o *commit* (*value*) a ser verificado, o *nonce*, os bits da inversão de colunas e os bits da criptografia da coluna de saída.

Já o contrato *SMC* é responsável por prover, de fato, as funcionalidades do protocolo. Nesse contexto, as principais funcionalidades desse contrato são as seguintes:

(a) Geração do *commit* inicial

```
1 function firstCommit(bytes32 commit, bool[3][4] memory truthTable)
     public returns (CommitHandler.Commit memory) {
2      CommitHandler.Commit memory commitGenerated = CommitHandler.
          generate(commit, truthTable);
3      commits[msg.sender] = commitGenerated;
4      emit StartComputation(msg.sender, commit);
5      return commitGenerated;
6 }
```

Código 3.5 – Geração do *commit* inicial

Essa função é responsável por gerar o primeiro *commit* da computação. Para tal, recebe como parâmetro o *commit* e a tabela correspondente.

(b) Geração do *commit* final

```
1 function secondCommit(address previousCommitOwner, bytes32 commit,
     bool[3][4] memory truthTable) public returns (CommitHandler.
     Commit memory) {
2      CommitHandler.Commit memory commitGenerated = CommitHandler.
          generate(commit, previousCommitOwner, truthTable);
3      commits[msg.sender] = commitGenerated;
```

```
4     emit FinishComputation(msg.sender, previousCommitOwner, commit)
         ;
5     return commitGenerated;
6 }
```

Código 3.6 – Geração do *commit* final

Essa função é responsável por gerar o segundo *commit* da computação. Para tal, recebe como parâmetro o endereço do dono do primeiro *commit*, o *commit* a ser criado e a tabela correspondente.

(c) Recuperação de um *commit* específico

```
1 function getCommit(address owner) public view returns (
     CommitHandler.Commit memory) {
2     return commits[owner];
3 }
```

Código 3.7 – Recuperação de um *commit* específico

Essa função é responsável por recuperar um *commit* armazenado no *smart contract*. Para tal, recebe o endereço do dono do *commit*.

(d) Verificação de *commit*

```
1 function verify(address owner, bytes memory nonce, bytes memory
     inversion_bits, bytes memory encryption_bits) public view
     returns (bool) {
2     bytes32 value = commits[owner].commit;
3     return CommitHandler.verify(value, nonce, inversion_bits,
         encryption_bits);
4 }
```

Código 3.8 – Verificação de *commit*

Essa função é responsável por verificar se um dado *commit* é válido. Para tal, recebe como parâmetro o endereço do dono do *commit* e os valores utilizados na etapa de ofuscamento da tabela: *nonce*, bits de inversão de coluna e bits de criptografia da coluna de saída.

(e) Computação final

```
1 function getValue(address owner, uint row, bool[2] memory
     first_commit_choices, bool[2] memory second_commit_choices)
     public view returns(bool) {
2     bool[3][4] memory truthTable = commits[owner].truthTable;
3     bool outputEntry = truthTable[row][2];
4     return outputEntry != first_commit_choices[0] !=
         first_commit_choices[1] != second_commit_choices[0] !=
         second_commit_choices[1];
```

```
5 }
```

Código 3.9 – Computação final

Essa função é responsável por retornar o valor da computação final de acordo com a lógica detalhada na seção 3.1.2. Para tal, recebe como parâmetro o endereço do dono do último *commit*, a linha selecionada pelos participantes, os bits revelados pelo participante α e os bits revelados pelo participante β.

3.2.2 Testes Unitários

De modo geral, testes unitários possibilitam a verificação do comportamento de uma unidade do sistema (ELBAUM et al., 2006). Nesse contexto, é importante testar as unidades *CommitHandler* e *SMC* a fim de determinar se suas funções comportam-se da maneira esperada tanto nos cenários honesto quanto desonestos. Sendo assim, usufruindo da compatibilidade entre as ferramentas Brownie e Pytest, os seguintes testes foram desenvolvidos para o contrato *CommitHandler*:

(a) Cenários Honestos

```
1 def test_verify(commit_handler, commit, nonce, inversion_bits,
    encryption_bits):
2     verify = commit_handler.verify(commit, nonce.encode(), bytes(
          inversion_bits), bytes(encryption_bits))
3     assert verify == True
4
5 def test_generate(commit_handler, A, commit, truthTable):
6     commit_generated = commit_handler.generate.call(commit,
          truthTable, {'from': A})
7     assert commit_generated[0] == A.address
8     assert commit_generated[1] == A.address
9     assert commit_generated[2] == commit
10    assert commit_generated[3] == truthTable
11
12 def test_generate_with_previous_owner(commit_handler, A, B, commit,
     truthTable):
13    commit_handler.generate(commit, truthTable, {'from': A})
14    commit_generated = commit_handler.generate.call(commit, A.
          address, truthTable, {'from': B})
15    assert commit_generated[0] == B.address
16    assert commit_generated[1] == A.address
17    assert commit_generated[2] == commit
18    assert commit_generated[3] == truthTable
```

Código 3.10 – Cenários Honestos do *CommitHandler*

Em relação os cenários honestos, três testes foram desenvolvidos para analisar o comportamento da geração e verificação de *commits*. O primeiro teste relaciona-se com a verificação de um *commit*, no qual dado um *commit* como parâmetro, o comportamento esperado da unidade testada é retornar *True*, uma vez que o *commit* informado é válido. O segundo e terceiro testes estão relacionados com a geração de *commits*, sendo o segundo acerca do comportamento da geração do primeiro *commit* e o terceiro sobre o comportamento da geração do segundo *commit*, em ambos o comportamento esperado é que os *assert* retornem *True* dado que os *commits* foram gerados da maneira correta.

(b) Cenários Desonestos

```
1  def test_verify_invalid_commit(commit_handler, invalid_commit,
      nonce, inversion_bits, encryption_bits):
2      verify = commit_handler.verify(invalid_commit, nonce.encode(),
          bytes(inversion_bits), bytes(encryption_bits))
3      assert verify == False
4
5  def test_verify_invalid_inversion_bits(commit_handler, commit,
      nonce, invalid_inversion_bits, encryption_bits):
6      verify = commit_handler.verify(commit, nonce.encode(), bytes(
          invalid_inversion_bits), bytes(encryption_bits))
7      assert verify == False
8
9  def test_verify_invalid_encryption_bits(commit_handler, commit,
      nonce, inversion_bits, invalid_encryption_bits):
10     verify = commit_handler.verify(commit, nonce.encode(), bytes(
          inversion_bits), bytes(invalid_encryption_bits))
11     assert verify == False
```

Código 3.11 – Cenários Desonestos do *CommitHandler*

Referente aos cenários desonestos, outros três testes foram desenvolvidos para analisar o comportamento da unidade *CommitHandler* quando um participante informa dados incoerentes. Nesse aspecto, o primeiro teste recebe um *commit* gerado de forma incorreta como parâmetro, já o segundo recebe bits de inversão de coluna inválidos e o terceiro recebe bits de criptografia da coluna de saída inválidos. Em ambos os testes o comportamento esperado é que a verificação retorne *False*, uma vez que os dados informados referem-se a comportamentos desonestos.

Já para o contrato *SMC*, temos:

(a) Cenários Honestos

```
1  def test_first_commit(contract, A, commit, truthTable):
```

```
2       first_commit = contract.firstCommit.call(commit, truthTable, {'
           from': A})
3       assert first_commit[0] == A.address
4       assert first_commit[1] == A.address
5       assert first_commit[2] == commit
6       assert first_commit[3] == truthTable
7
8   def test_second_commit(contract, A, B, commit, truthTable):
9       contract.firstCommit.call(commit, truthTable, {'from': A})
10      second_commit = contract.secondCommit.call(A.address, commit,
           truthTable, {'from': B})
11      assert second_commit[0] == B.address
12      assert second_commit[1] == A.address
13      assert second_commit[2] == commit
14      assert second_commit[3] == truthTable
15
16  def test_valid_verify_commit(A, contract, nonce, commit, truthTable
       , inversion_bits, encryption_bits):
17      contract.firstCommit(commit, truthTable, {'from': A})
18      is_valid_commit = contract.verify.call(A.address, nonce.encode
           (), bytes(inversion_bits), bytes(encryption_bits))
19      assert is_valid_commit == True
20
21  def test_get_commit(A, contract, commit, truthTable):
22      commit_generated = contract.firstCommit.call(commit, truthTable
           , {'from': A})
23      get_Commit = contract.getCommit.call(A.address)
24      assert get_Commit == commit_generated
```

Código 3.12 – Cenários Honestos do *SMC*

Em relação aos cenários honestos, quatro testes foram desenvolvidos para testar a unidade *SMC*. O primeiro e segundo testes relacionam-se, respectivamente, a geração do primeiro e segundo *commit* da computação. Já o terceiro teste refere-se a verificação de um *commit* e, finalmente, o quarto teste verifica o comportamento da função que recupera um *commit*. Para os cenários honestos, todos os testes retornam *True* para os *assert* dado que os valores informados são válidos.

(b) Cenário Desonesto

```
1   def test_invalid_verify_commit(A, contract, nonce, commit,
       truthTable, inversion_bits, encryption_bits):
2       contract.firstCommit(commit, truthTable, {'from': A})
3       is_valid_commit = contract.verify.call(A.address, nonce.encode
           (), bytes([False, True]), bytes(encryption_bits))
4       assert is_valid_commit == False
```

Código 3.13 – Cenário Desonesto do *SMC*

Referente aos cenários desonestos, apenas um teste foi desenvolvido para verificar o comportamento do contrato quando um *commit* inválido é informado na verificação. Nesse contexto, o resultado esperado do teste é retornar *False* para o *assert* dado que o *commit* informado é inválido.

3.3 Aplicabilidade

A fim de demonstrar os benefícios oferecidos pela solução proposta em um contexto real, o seguinte cenário é proposto: Alice e Bob almejam oficializar seu relacionamento por meio da união estável. Para tal, o casal opta pela modalidade união estável com comunhão parcial de bens, na qual os bens adquiridos antes do casamento são de posse individual de cada um e os bens adquiridos após o casamento são compartilhados entre o casal (PEREIRA, 1994).

Em um cenário comum, Alice e Bob optariam por assinar um contrato jurídico que validaria a união estável e garantiria os direitos para ambas as partes. Entretanto, o casal não deseja revelar quais bens possuíam antes do casamento e, em virtude disso, decidem utilizar o protocolo proposto nesta pesquisa para validar o casamento sem revelar seus bens. Em termos do protocolo, temos o seguinte contexto:

(i) A tabela inicial T' é arbitrária e concedida ao indivíduo responsável por iniciar o protocolo. Para o exemplo proposto, convenciona-se que Alice será essa responsável;

(ii) Os bens adquiridos antes da união estável são representados pelos bits de inversão de colunas (b_1, b_3) e bits de criptografia da coluna de saída (d_1, d_2, d_3, d_4). A relação semântica entre qual objeto um dado bit representa não é parte do escopo da pesquisa, dessa forma, para o exemplo proposto, assume-se que tanto Alice quanto Bob utilizarão o mesmo mapeamento semântico e que esse mapeamento é honesto;

(iii) A construção da tabela final T'' é dada pelo processo descrito nas seções 3.1.1 e 3.1.2, sendo que T'' representa o acordo final validado por Alice e Bob.

Sendo assim, para iniciar o protocolo, a tabela T arbitrária, representada pela Figura 14, é concedida à Alice. De posse de T, Alice executa as seguintes etapas:

(a) Alice escolhe os bits de inversão de coluna e de criptografia da coluna de saída que representarão seus bens;

(b) Utilizando os bits escolhidos anteriormente, Alice efetua o ofuscamento conforme descrito na seção 3.1.1 e obtém T' como resultado;

(c) Finalmente, Alice gera o *commit* de T' e envia para o *smart contract*.

T		
0	0	0
0	1	0
1	0	0
1	1	0

Figura 14 – Tabela T inicial

Assim que Alice sinaliza que encerrou suas etapas, Bob é capaz de continuar o processo. Dessa forma, o seguinte ocorre:

(a) Bob, de maneira semelhante à Alice, realiza o mapeamento semântico entre seus bens e o bits de inversão de coluna e criptografia da coluna de saída;

(b) Em seguida, Bob recebe T' por intermédio do *smart contract* e efetua o ofuscamento descrito na seção 3.1.1 a fim de obter T'';

(c) Posteriormente, Bob gera o *commit* de T'' e envia para o *smart contract*.

Ao final do processo, o *smart contract* é capaz de calcular o resultado final da computação. Além disso, caso necessário, tanto Alice quanto Bob são capazes de verificar cada etapa do processo por meio das verificações de cenários desonestos descritos na seção 3.1.3.

Em termos de implementação, o exemplo proposto é dado da seguinte forma:

```
1  from .base_permutation_handler import *
2  from brownie import *
3
4  ALICE = accounts[0]
5  BOB = accounts[1]
6  CommitHandler.deploy({'from': ALICE})
7  SMC.deploy({'from': ALICE})
8  contract = SMC[0]
9
10 def main():
11     T = [  [False,False,False],
12            [False,True,False],
13            [True,False,False],
14            [True,True,False] ]
15
16     # Etapa de Ofuscamento (Alice)
```

```
17      T_ALICE, commit_ALICE, nonce_ALICE, inversion_bits_ALICE,
            encryption_bits_ALICE = generate_transformed_TT(TT=T, nonceString
             = b'nonceALICE', firstColumn = 0, secondColumn = 2)
18      contract.firstCommit(commit_ALICE, T_ALICE, {'from': ALICE})
19
20      # Etapa de Ofuscamento (Bob)
21      T_from_contract = contract.getCommit.call(ALICE.address, {'from':
            BOB})[3]
22      T_BOB, commit_BOB, nonce_BOB, inversion_bits_BOB,
            encryption_bits_BOB = generate_transformed_TT(TT=T_from_contract,
             nonceString = b'nonceBOB', firstColumn = 1, secondColumn = 2)
23      contract.secondCommit(ALICE.address, commit_BOB, T_BOB, {'from': BOB
            })
24
25      # Final computation
26      row = 1
27      result = contract.getValue.call(BOB.address, row, [
            inversion_bits_ALICE[1], encryption_bits_ALICE[row]], [
            inversion_bits_BOB[1], encryption_bits_BOB[row]])
28      print("Final result: {r}".format(r=result))
```

Código 3.14 – Implementação do Exemplo do Casamento

Na implementação do 3.14, uma vez que o mapeamento semântico entre os bits e os bens não são do escopo da pesquisa, os bits são gerados de maneira aleatória tanto para Alice quanto para Bob e retornados por meio da função $generate_transformed_TT$.

Além disso, o casal é capaz de verificar o processo de construção das tabelas por meio da seguinte implementação:

```
1 # Verify
2 verifyALICE = contract.verify.call(ALICE.address, nonce_ALICE.encode(),
      bytes(inversion_bits_ALICE), bytes(encryption_bits_ALICE))
3 verifyBOB = contract.verify.call(BOB.address, nonce_BOB.encode(), bytes(
      inversion_bits_BOB), bytes(encryption_bits_BOB))
4 print("Commit from ALICE is valid: {v}".format(v=verifyALICE))
5 print("Commit from BOB is valid: {v}".format(v=verifyBOB))
```

Código 3.15 – Implementação da verificação da construção das tabelas

3.4 Análise de Gás

Dado que a viabilidade de executar um *smart contract* é diretamente proporcional à quantidade de gás necessária para sua execução, ou seja, quanto mais gás um contrato gasta menos viável é sua execução dependendo do cenário. Dessa forma, a análise de gás é fundamental para decidir sobre a utilização de um *smart contract*.

Definição 3 *Wei é a menor definição, também conhecida como unidade base, de um ether (ETHEREUM, 2021)*

De acordo com (ETHEREUM, 2021), 1 ether = 10^{18} *wei* e 1 ether = 10^{9} *Gwei*. Além disso, para fins de testes, foi convencionado que 1 gás correspondia a 59 *Gwei*, a qual é o custo médio da cotação atual.

A Tabela 15 mostra a relação entre uma dada ação utilizando a solução proposta e o custo necessário para efetuá-la. Para a construção dessa tabela, o ambiente utilizado é composto por um simulador de *blockchain* Ganache (versão 2.4.0), um *framework* de teste para *smart contracts* denominado Brownie (versão 1.14.3) e pela linguagem Solidity (versão 0.8.0). Foram consideradas as cotações do dólar e *ether* do dia 8 de maio de 2021. Além disso, o contrato utilizado como base para os valores da tabela é o mesmo discutido na seção 3.3.

Ação	Gás	Ether (ETH)	Real (BR)	Dólar (USD)
Deploy CommitHandler	420669	0.0248195	R$464,48	$88.70
Deploy SMC	857343	0.0505832	R$947,13	$180.87
Commit Alice	391740	0.0231127	R$432,80	$82.65
Commit Bob	412013	0.0243088	R$455,68	$87.02

Figura 15 – Análise de Custos

Analisando a Tabela 15 é perceptível que o maior custo está relacionado ao *deploy* do *smart contract*. Todavia, ainda que o *deploy* seja o custo mais elevado nas primeiras execuções do contrato, considerando que ele é realizado uma única vez, seu custo pode ser amortizado.

4 Conclusão

Nesse estudo, um protocolo para *Secure multi-party computation* utilizando *smart contracts*, cuja implementação pode ser conferida em (SILVA, 2020), foi desenvolvido de modo que seja possível realizar computações seguras entre duas entidades que não possuem uma relação de confiança entre si. Nesse sentido, uma computação segura consiste em utilizar dados privados de cada entidade durante o protocolo sem que esses dados precisem ser revelados ao outro participante.

Nesse contexto, a utilização dos *commits* armazenados no *smart contract* permite que as duas partes envolvidas no processo sejam capazes de manter dados privados e, ainda assim, participar de um protocolo seguro e verificável. Além disso, o protocolo permite que cada entidade participante realize parte do processamento das tabelas localmente, o que acarreta em economia de gás necessários para execução do contrato na *blockchain*.

Ainda que a solução proposta seja interessante, é importante ressaltar que a execução do *smart contract* com custo viável depende do custo do gás na cotação diária. Dessa forma, como o preço do gás é dinâmico, não é possível fixar um custo constante para a execução e, assim, os participantes do protocolo estão sujeitos a uma execução de preço dinâmico. Ademais, o contexto das aplicações que utilizam a solução desenvolvida nessa pesquisa interfere no custo final da execução: aplicações mais complexas tendem a utilizar mais gás e, consequentemente, geram maiores custos de execução.

De modo geral, por meio da aplicabilidade e resultados descritos, respectivamente, nas seções 3.3 e 3.4, é plausível concluir que o protocolo proposto é revelante, sobretudo, devido à possibilidade utilizar dados privados de cada entidade sem, necessariamente, revelá-los para efetivar a computação. Além disso, a decisão de utilizar SMC em detrimento do zk-SNARK proporcionou uma solução mais simples e que, ainda assim, atende os requisitos propostos. Finalmente, como trabalho futuro, pretende-se expandir o protocolo de modo a permitir a execução com mais de duas entidades e tabelas de tamanho arbitrário.

Referências

ARCHER, D. W. et al. From keys to databases—real-world applications of secure multi-party computation. *The Computer Journal*, Oxford University Press, v. 61, n. 12, p. 1749–1771, 2018. Citado na página 36.

ATZEI, N.; BARTOLETTI, M.; CIMOLI, T. A survey of attacks on ethereum smart contracts sok. In: *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Berlin, Heidelberg: Springer-Verlag, 2017. p. 164–186. ISBN 9783662544549. Disponível em: <https://doi.org/10.1007/978-3-662-54455-6_8>. Citado na página 16.

BAUMANN, N. et al. zkay v0. 2: Practical data privacy for smart contracts. *arXiv preprint arXiv:2009.01020*, 2020. Citado 2 vezes nas páginas 36 e 37.

BEN-SASSON, E. et al. Succinct non-interactive zero knowledge for a von neumann architecture. In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. USA: USENIX Association, 2014. (SEC'14), p. 781–796. ISBN 9781931971157. Citado na página 19.

BENHAMOUDA, F.; HALEVI, S.; HALEVI, T. Supporting private data on hyperledger fabric with secure multiparty computation. *IBM Journal of Research and Development*, IBM, v. 63, n. 2/3, p. 3–1, 2019. Citado 2 vezes nas páginas 36 e 37.

BLUM, M. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, ACM New York, NY, USA, v. 15, n. 1, p. 23–27, 1983. Citado na página 36.

BONDY, J. A.; MURTY, U. S. R. *Graph Theory with Applications*. New York: Elsevier, 1976. Citado na página 17.

BRASIL, C. 40% dos consumidores utilizaram cartão de crédito de alguma fintech nos últimos 12 meses, aponta pesquisa cndl/spc brasil. *CNDL*, Fevereiro 2020. Citado na página 8.

BROWNIE. *Brownie — Brownie 1.14.4 documentation*. [S.l.]: GitHub, 2020. <https://eth-brownie.readthedocs.io/en/stable/>. Citado na página 41.

BÜNZ, B. et al. Bulletproofs: Efficient range proofs for confidential transactions. *IACR Cryptology ePrint Archive*, v. 2017, p. 1066, 2017. Citado na página 36.

BUTERIN, V. What is ethereum? *Ethereum Official webpage. Available: http://www. ethdocs. org/en/latest/introduction/what-is-ethereum. html*, 2016. Citado na página 11.

CHAUM, D.; DAMGÅRD, I. B.; GRAAF, J. van de. Multiparty computations ensuring privacy of each party's input and correctness of the result. In: POMERANCE, C. (Ed.). *Advances in Cryptology — CRYPTO '87*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988. p. 87–119. ISBN 978-3-540-48184-3. Citado na página 38.

CHAUM, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, ACM New York, NY, USA, v. 24, n. 2, p. 84–90, 1981. Citado na página 36.

CHENG, R. et al. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. *2019 IEEE European Symposium on Security and Privacy (EuroSP)*, IEEE, Jun 2019. Disponível em: <http://dx.doi.org/10.1109/EuroSP.2019.00023>. Citado na página 16.

DABEK, F. et al. Vivaldi: A decentralized network coordinate system. *ACM SIGCOMM Computer Communication Review*, v. 34, 08 2004. Citado na página 8.

DESTEFANIS, G. et al. Smart contracts vulnerabilities: A call for blockchain software engineering? In: . [S.l.: s.n.], 2018. Citado 2 vezes nas páginas 15 e 16.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. *IEEE transactions on Information Theory*, IEEE, v. 22, n. 6, p. 644–654, 1976. Citado na página 34.

ELBAUM, S. et al. Carving differential unit test cases from system test cases. In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. [S.l.: s.n.], 2006. p. 253–264. Citado na página 45.

ETHEREUM. *Solidity*. 2020. GitHub Documentation. Disponível em: <https://solidity.readthedocs.io/>. Citado na página 14.

ETHEREUM. *PROOF-OF-STAKE (POS)*. [S.l.]: Ethereum, 2021. <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/>. Citado 2 vezes nas páginas 13 e 51.

FITZI, M.; HIRT, M.; MAURER, U. Trading correctness for privacy in unconditional multi-party computation. In: SPRINGER. *Annual International Cryptology Conference*. [S.l.], 1998. p. 121–136. Citado na página 34.

FRANKLIN, M. Complexity and security of distributed protocols. In: . [S.l.: s.n.], 1994. Citado na página 34.

GALBRAITH, S.; HARRISON, K.; SOLDERA, D. Implementing the tate pairing. In: . [S.l.: s.n.], 2002. v. 2369, p. 324–337. Citado na página 30.

GATHEN, J. v. z. *CryptoSchool*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2015. ISBN 3662484234. Citado na página 29.

GOLDWASSER, S.; MICALI, S.; RACKOFF, C. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, v. 18, n. 1, p. 186–208, 1989. Citado na página 17.

GRECH, N. et al. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 2, n. OOPSLA, out. 2018. Disponível em: <https://doi.org/10.1145/3276486>. Citado 2 vezes nas páginas 11 e 15.

GROTH, J. Short non-interactive zero-knowledge proofs. In: ABE, M. (Ed.). *Advances in Cryptology - ASIACRYPT 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 341–358. ISBN 978-3-642-17373-8. Citado na página 24.

GROTH, J. On the size of pairing-based non-interactive arguments. In: SPRINGER. *Annual international conference on the theory and applications of cryptographic techniques*. [S.l.], 2016. p. 305–326. Citado na página 36.

JIANG, B.; LIU, Y.; CHAN, W. K. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018. (ASE 2018), p. 259–269. ISBN 9781450359375. Disponível em: <https://doi.org/10.1145/3238147.3238177>. Citado 2 vezes nas páginas 15 e 16.

KOBAYASHI, T.; AOKI, K.; IMAI, H. Efficient algorithms for tate pairing. *IEICE Transactions*, v. 89-A, p. 134–143, 01 2006. Citado na página 30.

KOENS, T.; RAMAEKERS, C.; WIJK, C. van. Efficient Zero-Knowledge Range Proofs in Ethereum. 2017. Disponível em: <https://www.ingwb.com/media/2122048/zero-knowledge-range-proof-whitepaper.pdf>. Citado 2 vezes nas páginas 19 e 31.

KOSBA, A. et al. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE. *2016 IEEE symposium on security and privacy (SP)*. [S.l.], 2016. p. 839–858. Citado na página 16.

KRAIEM, F. et al. On the classification of knowledge-of-exponent assumptions in cyclic groups. *Interdisciplinary Information Sciences*, 09 2019. Citado 2 vezes nas páginas 23 e 24.

LEDGER. *What is Proof-of-Stake?* [S.l.]: Ledger, 2019. <https://www.ledger.com/academy/blockchain/what-is-proof-of-stake>. Citado 2 vezes nas páginas 12 e 13.

LEDGER. *What is Proof-of-Work*. [S.l.]: Ledger, 2019. <https://www.ledger.com/academy/blockchain/what-is-proof-of-work>. Citado na página 13.

LEE, J. H. Dappguard : Active monitoring and defense for solidity smart contracts. In: . [S.l.: s.n.], 2017. Citado 2 vezes nas páginas 15 e 16.

Liu, C. et al. Reguard: Finding reentrancy bugs in smart contracts. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. [S.l.: s.n.], 2018. p. 65–68. Citado na página 16.

LIU, D.; CAMP, L. J. Proof of work can work. In: CITESEER. *WEIS*. [S.l.], 2006. Citado na página 12.

LUU, L. et al. Making smart contracts smarter. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2016. (CCS '16), p. 254–269. ISBN 9781450341394. Disponível em: <https://doi.org/10.1145/2976749.2978309>. Citado 3 vezes nas páginas 12, 14 e 15.

MERKLE, R. C. A fast software one-way hash function. *Journal of Cryptology*, Springer, v. 3, n. 1, p. 43–58, 1990. Citado na página 41.

MICALI, S.; ROGAWAY, P. Secure computation. In: SPRINGER. *Annual International Cryptology Conference*. [S.l.], 1991. p. 392–404. Citado na página 36.

MIERS, I. Scalable multi-party computation for zk-snark parameters in the random beacon model. 2019. Citado na página 36.

PANKOVA, A. *Succinct Non-Interactive Arguments from Quadratic Arithmetic Programs*. 2013. Citado 2 vezes nas páginas 25 e 26.

PARNO, B. et al. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, v. 59, n. 2, p. 103–112, 2016. Disponível em: <http://dblp.uni-trier.de/db/journals/cacm/cacm59.html#ParnoHG016>. Citado na página 29.

PEREIRA, R. D. C. *Concubinato e união estável*. [S.l.]: Saraiva Educação SA, 1994. v. 8. Citado 2 vezes nas páginas 10 e 48.

PETKUS, M. Why and how zk-snark works. *CoRR*, abs/1906.07221, 2019. Disponível em: <http://arxiv.org/abs/1906.07221>. Citado 4 vezes nas páginas 19, 20, 23 e 31.

PINNA, A. et al. A massive analysis of ethereum smart contracts. empirical study and code metrics. *IEEE Access*, 06 2019. Citado na página 14.

RAMACHANDRAN, A.; KANTARCIOGLU, D. M. *Using Blockchain and smart contracts for secure data provenance management*. 2017. Citado na página 15.

SÁNCHEZ, D. C. Raziel: Private and verifiable smart contracts on blockchains. *arXiv preprint arXiv:1807.09484*, 2018. Citado 2 vezes nas páginas 36 e 37.

Sankar, L. S.; Sindhu, M.; Sethumadhavan, M. Survey of consensus protocols on blockchain applications. In: *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*. [S.l.: s.n.], 2017. p. 1–5. Citado na página 12.

SILVA, B. C. da. *SMC Contact*. [S.l.]: GitHub, 2020. <https://github.com/BiancaCristina/SMC-Contract>. Citado 2 vezes nas páginas 41 e 52.

STEFFEN, S. et al. Zkay: Specifying and enforcing data privacy in smart contracts. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2019. (CCS '19), p. 1759–1776. ISBN 9781450367479. Disponível em: <https://doi.org/10.1145/3319535.3363222>. Citado na página 9.

SUITE, T. *Ganache | Truffle Suite*. [S.l.]: Truffle Suite, 2020. <https://www.trufflesuite.com/docs/ganache/overview>. Citado na página 41.

TIKHOMIROV, S. Ethereum: State of knowledge and research perspectives. In: IMINE, A. et al. (Ed.). *Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France, October 23-25, 2017, Revised Selected Papers*. Springer, 2017. (Lecture Notes in Computer Science, v. 10723), p. 206–221. Disponível em: <https://doi.org/10.1007/978-3-319-75650-9_14>. Citado na página 8.

UNTERWEGER, A. et al. Lessons learned from implementing a privacy-preserving smart contract in ethereum. In: IEEE. *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. [S.l.], 2018. p. 1–5. Citado 2 vezes nas páginas 36 e 37.

WLOKA, J.; SRIDHARAN, M.; TIP, F. Refactoring for reentrancy. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2009. (ESEC/FSE '09), p. 173–182. ISBN 9781605580012. Disponível em: <https://doi.org/10.1145/1595696.1595723>. Citado na página 15.

WÖHRER, M.; ZDUN, U. Design patterns for smart contracts in the ethereum ecosystem. In: IEEE. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. [S.l.], 2018. p. 1513–1520. Citado na página 15.

WU, J.; STINSON, D. R. *An Efficient Identification Protocol and the Knowledge-of-Exponent Assumption*. Citado 2 vezes nas páginas 23 e 24.

XIAO, Y. et al. Distributed consensus protocols and algorithms. *Blockchain for Distributed Systems Security*, Wiley, v. 25, 2019. Citado na página 12.

YAO, A. C. Protocols for secure computations. In: IEEE. *23rd annual symposium on foundations of computer science (sfcs 1982)*. [S.l.], 1982. p. 160–164. Citado 4 vezes nas páginas 31, 32, 35 e 36.

YAO, A. C.-C. How to generate and exchange secrets. In: IEEE. *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. [S.l.], 1986. p. 162–167. Citado na página 35.

ZCASH. Zcash, 2017. Disponível em: <https://z.cash/technology/zksnarks/>. Citado na página 20.

ZOU, R.; LV, X.; ZHAO, J. Spchain: Blockchain-based medical data sharing and privacy-preserving ehealth system. *arXiv preprint arXiv:2009.09957*, 2020. Citado 2 vezes nas páginas 36 e 37.