Capítulo 1

Blockchain, Contratos Inteligentes, Sistemas Web: Teoria e Prática

Jauberth Abijaude, Henrique Serra, Levy Santiago,
Péricles Sobreira, Fabíola Greve

Abstract

Blockchain is a disruptive technology that offers a digital trust network for carrying out transactions between peers. Smart contracts are codes hosted on the blockchain and establish contractual clauses to be followed. In this mini-course, we present the fundamental concepts, mechanisms, and platforms used by these technologies, to develop in their participants the needs for implementing required and distributed Web applications (DApps). For this, we will demonstrate, through practical exercises, a model for implementing smart contracts interacting with Web systems, in addition to discussing the advances, opportunities, and challenges in research related to this area of knowledge.

Resumo

A blockchain é uma tecnologia disruptiva que oferece uma rede de confiança digital para a realização de transações entre pares. Os contratos inteligentes são códigos hospedados na blockchain e estabelecem cláusulas contratuais a serem seguidas. Neste minicurso, apresentamos os conceitos fundamentais, os mecanismos e as plataformas utilizadas por estas tecnologias, de forma a desenvolver em seus participantes as competências necessárias à implementação de aplicações Web confiáveis e distribuídas (DApps). Para isto, demonstraremos, através de exercícios práticos, um modelo de implementação de contratos inteligentes interagindo com sistemas Web, além de discutirmos os avanços, as oportunidades e os desafios em pesquisas relacionadas a esta área do conhecimento.

1.1. Introdução

Este capítulo tem como objetivo desenvolver competências na criação de sistemas web integrados com blockchain e contratos inteligentes, com foco na plataforma *Ethereum*, para formar estudantes da área de Sistemas de Informação, Análise e Desenvolvimento

de Sistemas, Engenharia de Software, Ciência da Computação e afins, principalmente pela existência de uma crescente demanda de profissionais, aliado ao fato destes assuntos não serem abordados nos currículos universitários atuais.

Será apresentada a Blockchain (BC), os Contratos Inteligentes (CI) e sua integração com sistemas Web. A abordagem é dividida em seis seções: a primeira e segunda, teóricas, abordam histórico, conceitos fundamentais e apresentação da plataforma *Ethereum*. A terceira, apresenta a linguagem de programação *Solidity* e as características fundamentais para o desenvolvimento de CIs na plataforma *Ethereum*, introduzindo a primeira prática do curso. A quarta seção, teórica e prática, explora a integração com sistemas web e o desenvolvimento de uma aplicação distribuída (DApp). A quinta, discute como um professor pode abordar este assunto em uma sala de aula, relatando a experiência dos autores. Por fim, a última seção apresenta os desafios e perspectivas na área.

1.1.1. Histórico e Conceitos Fundamentais

Nick Szabo, em 1997, apresenta os CIs como uma combinação de protocolos com interfaces de usuário para formalizar e proteger relacionamentos em redes de computadores [Szabo 1997]. Esta foi a primeira vez que este termo foi usado, porém a ideia permaneceu adormecida por não encontrar um sistema computacional que oferecesse os requisitos mínimos necessários para a sua implementação.

Uma década depois, em agosto de 2008, Satoshi Nakamoto descreve uma habilidosa combinação de técnicas para dar suporte à criptomoeda *bitcoin* [Nakamoto 2008]. Este protocolo descreve a rede (BC) e suas propriedades; em janeiro de 2009, ela entrou em operação. Esta BC original incorpora uma máquina de estados simplificada, com transações voltadas para a moeda *bitcoin*. Ela é composta por uma combinação de técnicas robustas provenientes da computação distribuída confiável (tolerância a falhas bizantinas, sistemas P2P), criptografia (chave assimétrica, funções *hash*, desafios criptográficos) e teoria dos jogos (mecanismos de incentivos) [Greve e outros 2018].

Vitalik Buterin, após observar e acompanhar as discussões da BC *Bitcoin*, em 2013, propôs uma nova plataforma - o *Ethereum*, que entrou em operação em julho de 2015 [Buterin e outros. 2014]. Esta nova proposta de BC estabeleceu uma nova criptomoeda, o *ether*, e retomou a ideia de Nick Szabo sobre os contratos inteligentes, permitindo transacionar a criptomoeda e códigos de programação mais avançados. Tais códigos são os CIs que passam a ser implementados na rede BC e permitem a execução de uma máquina de *Turing* completa, possibilitando as chamadas Aplicações Descentralizadas ou DApps [Antonopoulos e Wood 2018].

Atualmente, existem diversas plataformas de blockchain com forte investimento da indústria para desenvolvimento de aplicações robustas e descentralizadas em vários segmentos. Novas plataformas de BC como *Hyperledger Fabric* [Androulaki e outros 2018], *Nano* [LeMahieu 2018], *Iota* [Popov 2018], e diversas outras surgiram, cada uma com suas características particulares.

1.1.2. Propriedades da Blockchain

A BC possui propriedades que cooperam no desenvolvimento das DApps e permitem que estas as herdem, por conseguinte. Dentre elas pode-se destacar [Greve e outros 2018]:

- Descentralização: Sistemas e aplicações que usam a BC não precisam de uma entidade central para coordenar as ações, as tarefas são executas de forma distribuída;
- Disponibilidade e integridade: Os dados e as transações são replicados para todos os participantes da BC, mantendo o sistema seguro e consistente;
- Transparência e auditabilidade: A cadeia de blocos que registra as transações é pública e pode ser auditada e verificada;
- Imutabilidade e Irrefutabilidade: os registros são imutáveis e a correção só pode ser feita a partir de novos registros. O uso de recursos criptográficos garante que os lançamentos não podem ser refutados;
- Privacidade e Anonimidade: As transações são anônimas, com base nos endereços dos usuários. Os servidores armazenam apenas fragmentos criptografados dos dados do usuário;
- Desintermediação: A BC consegue eliminar terceiros em suas transações, atuando como um conector de sistemas de forma confiável e segura.
- Cooperação e incentivos: Uso do modelo de teoria dos jogos como forma de incentivo.

1.1.3. Componentes de uma Rede Blockchain

As redes BC podem ser públicas ou privadas [Greve e outros 2018]. A primeira é também conhecida como não permissionada ou de acesso aberto. A segunda é comumente chamada de permissionada ou federada. Nas BC públicas, o acesso é anônimo, não há nenhum controle sobre a entrada e saída de nós na rede e eles não possuem confiança mútua. Como exemplos destas redes, tem-se a *Bitcoin* e a *Ethereum*. Já nas BC privadas ou federadas, os nós são conhecidos e precisam ser autenticados. São redes voltadas normalmente para ambientes corporativos, onde cada participante tem um papel definido. A BC *Hyperledger Fabric* é um exemplo de BC privada.

Os componentes essenciais de uma BC pública ou privada são [Greve e outros 2018] [Antonopoulos e Wood 2018]:

- Uma rede ponto a ponto (P2P) conectando participantes e propagando transações e blocos de transações verificadas, com base em um protocolo padronizado;
- Mensagens, na forma de transações, representando transições de estado;
- Um conjunto de regras de consenso, definindo o que constitui uma transação e o que contribui para uma transição de estado válida;
- Uma máquina de estado que processa transações de acordo com as regras de consenso;
- Uma cadeia de blocos protegidos criptograficamente que atua como um diário de todas as transições de estado verificadas e aceitas;

- Um algoritmo de consenso que descentraliza o controle sobre a blockchain, forçando os participantes a cooperarem na aplicação das regras de consenso;

Em adição a esses componentes, as BC públicas apresentam, a partir de elementos de teoria dos jogos:

- Um esquema de incentivos teoricamente sólido para proteger economicamente a máquina de estados em um ambiente aberto, como por exemplo, os custos do algoritmo de consenso *Proof of Work*[Nakamoto 2008], em conjunto com custos de recompensas em bloco.

A grande totalidade destes componentes geralmente é combinada em um único cliente de software. Por exemplo, no Bitcoin, a implementação de referência é desenvolvida pelo projeto de código aberto *Bitcoin Core* e implementada como o cliente bitcoind. No Ethereum, em vez de uma implementação de referência, há uma especificação de referência, uma descrição matemática do sistema no *yellow paper* [Buterin e outros. 2014]. Existem vários clientes, que são construídos de acordo com a especificação de referência, dentre eles, o mais popular é o *Geth* [Go-ethereum 2013].

1.2. Plataforma Ethereum e Contratos Inteligentes

Nesta seção, discute-se a teoria da BC *Ethereum* e dos CIs. Os principais componentes da BC *Ethereum*, as carteiras, a estrutura das transações, redes disponíveis e outros detalhes serão discutidos.

1.2.1. Plataforma *Ethereum*

O acesso a plataforma *Ethereum* pode ser classificado sob dois aspectos: (a) Acesso para os desenvolvedores e (b) Acesso para os usuários. Em cada um deles há ferramentas e métodos diferentes.

Em (a), os desenvolvedores, comumente, usam a `web3`. Ela é uma coleção de bibliotecas que permitem a interação com um nó *Ethereum*, local ou remoto, usando HTTP, IPC ou *WebSocket* [Web3js 2016], com APIs(*Application Programming Interface*) disponíveis para Java, Android e Javascript.

Em (b), os usuários conectam-se à rede *Ethereum* usando um cliente remoto (um aplicativo de software que implementa a especificação *Ethereum* e se comunica pela rede ponto a ponto com outros clientes *Ethereum*). Estes clientes remotos oferecem um subconjunto da funcionalidade de um cliente completo. Eles não armazenam a blockchain *Ethereum* completa, são mais rápidos de configurar e requerem menos armazenamento de dados.

Geralmente, os clientes remotos permitem: (1) Gerenciar chaves privadas e endereços *Ethereum* em uma carteira; (2) Criar, assinar e transmitir transações; (3) Interagir com contratos inteligentes, usando a carga útil de dados; (4) Navegar e interagir com DApps; (5) Oferecer links para serviços externos, como exploradores de blocos; (6) Converter unidades de *ether* e recuperar taxas de câmbio de fontes externas; (7) Injetar uma instância `web3` no navegador web como um objeto JavaScript; (8) Usar uma instância

`web3` fornecida/injetada no navegador por outro cliente; e/ou (9) Acessar os serviços RPC em um nó *Ethereum* local ou remoto.

As carteiras móveis (*wallets*) são clientes remotos, já que os smartphones não têm recursos adequados para executar um cliente *Ethereum* completo. Os mais populares são *Jaxx* [Jaxx 2018], *Status* [Status 2019] , *Trust Wallet* [Trust 2019] e *Coinbase* [Coinbase 2018].

Os usuários podem também usar navegadores, onde as carteiras estão disponíveis como plugins ou extensões do Chrome ou Firefox, por exemplo. Estes clientes remotos são executados no navegador. Os mais populares são *Metamask* [Metamask 2018], *Jaxx*, *MyEtherWallet* [Myetherwallet 2019], *Nifty* e MyCrypto [MyCrypto 2019].

Usando uma destas carteiras é possível acessar a plataforma *Ethereum*, constituída da rede principal e da rede de testes. Na rede principal são transacionados *ethers* reais que precisam ser comprados e possuem valor financeiro. Na rede de testes, transacionam-se *ethers* fictícios, sem valor financeiro, adquiridos gratuitamente através de geradores de *ethers* na internet.

O *ether* é subdividido em unidades menores. A menor unidade possível é denominada *wei*. Um *ether* equivale a 1 quintilhão de *weis* ($1*10^{18}$ ou 1.000.000.000.000.000.000). Uma tabela de conversão e diversas informações sobre *gas* e *ethers* podem ser encontradas em [GasStation 2017].

1.2.1.1. Rede Principal e de Testes

A rede principal, endereçável na porta TCP 30303 trabalha com *ethers* que precisam ser comprados com dólares e as transações sofrem consequências reais.

As redes de teste trabalham com *ethers* que não possuem valor real e que podem ser adquiridos em geradores de *ethers* na Internet sem custos financeiros. A rede *Ropsten* é uma rede de teste pública de blockchain. A rede de teste *Kovan* é uma rede de teste pública que usa o protocolo de consenso Aura com prova de autoridade (Esta é uma rede permissionada). A rede de teste *Rinkeby* utiliza o protocolo de consenso *"Clique"* com prova de autoridade (Esta é uma rede permissionada). A opção *localhost* 8545 conecta-se a um nó em execução no mesmo computador que o navegador. A opção *Custom RPC* permite que conexão a qualquer nó com uma interface de Chamada de Procedimento Remoto (RPC) compatível com *Geth*. O nó pode fazer parte de qualquer blockchain pública ou privada.

Este minicurso usa a rede de testes *Rinkeby* e o plugin *Metamask*. Para abastecer a carteira do metamask com *ethers* sem valor comercial na rede *Rinkeby*, usa-se, por exemplo, o site `https://faucet.rinkeby.io/`.

O *metamask* é um gateway para aplicações da plataforma *Ethereum*. Ele fornece acesso a todas as redes da plataforma com uma única conta, permitindo que se gerencie as carteiras de todas as redes *Ethereum*. A Figura 1.1 ilustra isto. O *metamask* pode ser instalado nos navegadores Chrome ou Firefox sob forma de extensão. Ao instalar, você receberá 12 palavras mnemônicas, e deve guardá-las sob o maior sigilo, pois são a

única forma de recuperar a sua conta ou fazer transações usando a `web3`. Estas palavras devem ser informadas na ordem em que são apresentadas, na criação da conta, quando for necessário. O procedimento para instalação do *Metamask* pode ser acessado na página do curso [Abijaude e outros 2020].

O *Metamask* pode criar outras contas de acesso às redes *Ethereum*. Isto quer dizer que com as mesmas palavras mnemônicas e com a mesma instância instalada no navegador, o usuário pode ter vários endereços de contas. Isto é muito importante, principalmente no momento em que formos usar o CI e a DApp, descritas no decorrer do texto.

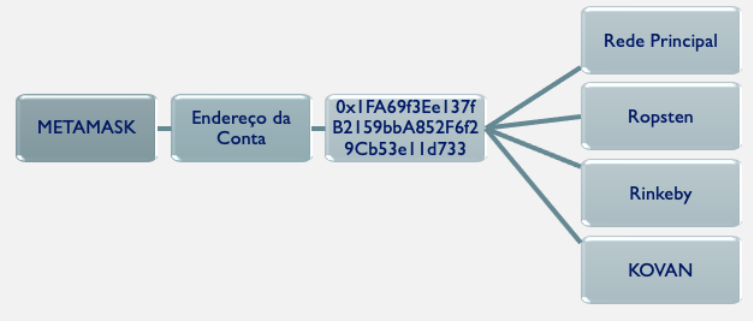

Figura 1.1. Conta na carteira metamask para acesso as redes *Ethereum*.

1.2.1.2. Transações

As transações *Ethereum* são mensagens de rede que incluem os campos ***nonce*** (número de sequência, emitido pela origem, usado para evitar a reprodução da mensagem), ***recipient*** (endereço para onde será enviada uma quantidade de *ether*), **value** (quantidade de *ether* a ser enviada), ***gasPrice*** (valor a ser pago por unidade de *gas* para a transação ser processada), ***startGas/gasLimit*** (quantidade de *gas* que esta transação pode consumir), ***data*** (A carga útil de dados binários de comprimento variável) e "**r**", "**v**"e "**s**"(variáveis criptográficas).

A função do *gasPrice* e *gasLimit* em uma transação precisa ficar bem clara. O *gas* é como se fosse o combustível do *Ethereum*. O *gas* não é *ether*, é uma moeda virtual separada com sua própria taxa de câmbio em relação ao *ether*. O *Ethereum* utiliza o *gas* separado do *ether* como forma de isolar a cotação da moeda *ether* no mundo real do valor das transações na rede, pelos quais o *gas* paga (computação, memória e armazenamento).

O campo *gasPrice* em uma transação permite que o emissor da transação defina o preço que está disposto a pagar para adquirir o *gas*. O preço é medido em *wei* por unidades de *gas*. O campo *limitGas* indica qual o limite de *gas* a ser consumido pela transação, ou seja, o número máximo de unidades de *gas* que o emissor da transação está disposto a comprar para concluir a transação.

Uma transação simples, de transferência de *ether* de uma conta para outra, custa 21.000 unidades de *gas*. O valor a ser pago em *ether* é encontrado multiplicando-se 21.000 x *gasPrice*. As carteiras geralmente possuem um valor médio cobrado na rede

para que uma transação seja confirmada dentro de um tempo aceitável (no momento da escrita deste texto estava em torno de 12,3 *gwei*). Neste caso, uma transferência de *ether* custaria 21.000 x 12,3 = 266.700 *gwei*,o que equivale a 0.0002667 *ether*.

As carteiras podem ajustar o *gasPrice* nas transações originadas para obter uma confirmação mais rápida das transações. Quanto maior o *gasPrice*, mais rápido a transação provavelmente será confirmada. Por outro lado, as transações de baixa prioridade podem ter um preço reduzido, resultando em uma confirmação mais lenta. O valor mínimo que *gasPrice* pode ser definido é zero, o que significa uma transação sem taxas. Durante os períodos de demanda por espaço em um bloco, essas transações podem ser preteridas.

1.2.1.3. Consenso

O *Ethereum* usa o modelo de consenso do *Bitcoin* (*PoW - Proof of Work*). No entanto, há planos em um futuro próximo de mudar para um sistema de votação ponderada (*PoS - Proof of Stake*), cujo codinome é *Casper*. As regras de consenso da *Ethereum* são definidas em [Wood e outros. 2014]. O algoritmo PoW usado pelo *Ethereum* é o *Ethash* [Wiki 2017].

As transições de estado do *Ethereum* são processadas pela Máquina Virtual *Ethereum* EVM (do inglês *Ethereum Virtual Machine*), baseada em uma pilha que executa bytecodes gerados pela compilação dos CIs.

O estado de *Ethereum* é armazenado localmente em cada nó como um banco de dados (geralmente o *LevelDB* do Google), que contém as transações e o estado do sistema em uma estrutura de dados em *hash* serializada chamada de *Merkle Patricia Tree*.

Estes nós clientes são um aplicativo de software que implementa a especificação *Ethereum*, comunicando-se pela rede ponto a ponto com outros clientes *Ethereum*. Há diferentes implementações, mas interoperáveis, e dentre as mais comuns, temos: *Geth* (escrito em *Go*), *Parity* (escrito em *Rust*), *Cpp-ethereum* (escrito em C++), *Pyethereum* (escrito em Python), *Mantis* (escrito em Scala) e *Harmony* (escrito em Java).

1.2.2. Contratos Inteligentes

Os CIs, segundo Nick Szabo [Szabo 1997], representam "um conjunto de promessas, especificado em formato digital, incluindo protocolos nos quais as partes cumprem estas promessas". Este conceito evoluiu, especialmente após a introdução de plataformas blockchain descentralizadas.

Recentemente, Antonopoulos reformou este conceito para se referir a programas de computador imutáveis, que são executados de forma determinística, no contexto de uma EVM, como parte do protocolo de rede *Ethereum* - ou seja, no computador mundial *Ethereum* descentralizado [Antonopoulos e Wood 2018].

Portanto, os CIs são simplesmente programas de computador. A palavra **contrato** não tem significado legal neste contexto. Eles são imutáveis, por que uma vez implementado em uma rede *Ethereum*, o código não pode ser alterado nem substituído. A única forma de se modificar o seu conteúdo é implementando um novo contrato, o qual terá um

novo endereço.

Assim como os softwares, os contratos são determinísticos, pois o resultado de sua execução é sempre o mesmo para todos os que o executam, conservando-se o contexto no momento da execução. Os CI estão em constante evolução e operam com um contexto muito limitado, por enquanto. No caso dos CIs para a rede *Ethereum*, existem diversas versões do compilador (*Solc*), com mudanças significativas entre elas. De modo geral, os CIs acessam seu próprio estado, o contexto da transação que os chamou e algumas informações sobre os blocos mais recentes.

As linguagens de programação de alto nível atualmente suportadas para construção dos contratos inteligentes são *LLL*, *Serpent*, *Vyper*, *Bambu* e *Solidity*. Esta última é a mais popular, suportada pela plataforma *Ethereum* e utilizada neste minicurso. Após escritos, os contratos precisam ser compilados para depois serem implementados em uma rede *Ethereum*. O resultado do processo de compilação são os *bytecodes* e a *Application Binary Interface* (ABI), conforme ilustrado na Figura 1.2.

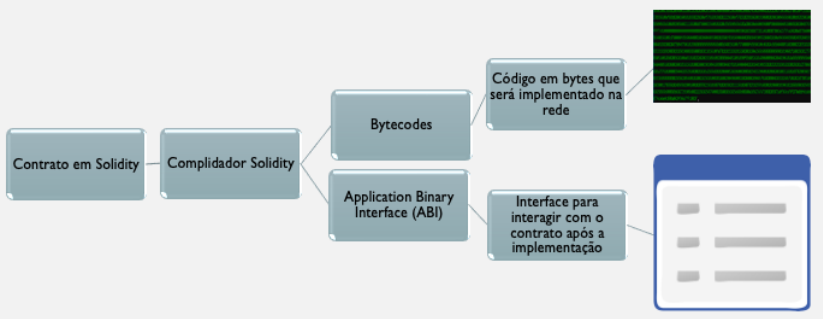

Figura 1.2. O contrato, após compilado, gera duas saídas - Os *bytecodes* e a ABI.

Os *bytecodes*, de baixo nível, são implementados na plataforma *Ethereum* usando uma transação de criação de contrato enviada para um endereço especial de criação de contratos. Cada contrato, portanto, possui um endereço *Ethereum*, que é derivado da transação de criação do contrato em função da conta e do nonce de origem. Ele é proprietário de seu próprio endereço, o qual pode ser usado, em uma transação, para receber *ethers*, por exemplo, de uma outra conta contrato ou de uma conta *Ethereum* cliente.

Para tanto é necessário acessar uma função do contrato, por intermédio da ABI. Somente através desta interface é que se pode ter acesso às funções do contrato e executar as rotinas previamente programadas.

É importante acrescentar que os contratos somente executam funções se forem chamados por uma transação. Os contratos nunca podem chamar a si próprios ou atuarem em *background*, mas podem chamar outros contratos em cadeia. As transações são atômicas, e caso a execução ocorra sem erros até o final, toda a transação é registrada.

As transações envolvendo os CIs possuem algumas particularidades em relação às transações envolvendo apenas as carteiras. Uma destas particularidades é que a transação envolvendo CIs contém dois campos: valor e dados. Estes valores podem alternadamente serem preenchidos ou não. Quando o endereço de destino de uma transação for relativa a um contrato, a EVM executará o contrato e tentará chamar a função nomeada na carga

útil de dados de sua transação. Se não houver dados, o EVM irá chamar uma função de *fallback* e, se esta função envolver pagamentos, irá executá-la para determinar o que fazer a seguir.

Outra particularidade é que as contas que representam CI possuem diferenças em relação às contas que representam apenas uma carteira eletrônica. Enquanto nestas, uma única conta pode acessar as diversas redes Ethereum, nas contas de CI isto não é possível. Um conta que representa um CI só pode acessar a rede na qual ela foi implementada. Para que este contrato possa ser implementado em outra rede *Ethereum* é necessário implementar uma nova instância do contrato nesta nova rede, com outro pagamento das taxas da transação.

Os CIs podem ser gerados basicamente de duas formas. Usando editores online como *Studio Ethereum* [StudioEthereum 2019], *Ethfidlle* [Ethfiddle 2017] e o *Remix* [Remix 2015], ou através de qualquer editor de texto, após configurar adequadamente um ambiente local para desenvolvimento, o qual será discutido na seção 1.3.

O *Remix* é um ambiente online configurado para programar, compilar e implementar CIs. Além de um editor de texto integrado, ele possui diversas versões de compiladores prontos para usar. Nele, existem 3 modos de se implementar os contratos: (a) Através de uma máquina virtual JavaScript, implementada no navegador; (b) usando a `web3` injetada pelo *Metamask*; ou (c) fornecendo um endereço para conexão de um provedor `web3`.

A figura 1.3 ilustra o ambiente de desenvolvimento do *Remix*. O contrato em tela é um exemplo didático e pode ser encontrado na página do minicurso. Este contrato é compilado na versão 7.4 do `solc` (compilador do *solidity*). Na linha 1 é informado o tipo de licença para o contrato. Em seguida, na linha 2 informa-se qual a versão do `solc` será usada para compilar o contrato.

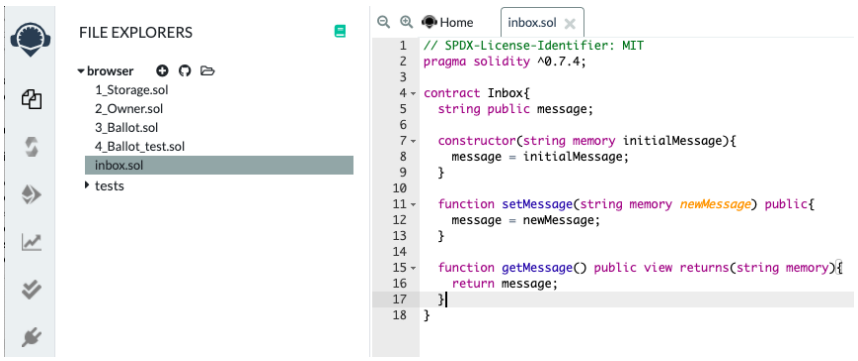

Figura 1.3. Exemplo didático de CI utilizando o editor on-line Remix.

Entre as linhas 4 e 18 está o CI propriamente dito. A linha 4 define o nome do contrato e a linha 5 declara a variável que será utilizada. As linhas 7 e 8 declaram o construtor do contrato, que será executado uma única vez, recebendo como parâmetro uma mensagem inicial.

Por fim, temos duas funções definidas no contrato. A função `setMessage`, na linha 11, quando invocada, recebe como parâmetro uma mensagem nova e atualiza a variável do contrato. A função `getMessage`, na linha 15, retorna o valor armazenado

na variável `message`. Observe que, enquanto a primeira mensagem modifica o valor de uma variável do contrato, a segunda apenas lê o seu conteúdo.

Isto implica que a função `setMessage`, quando for invocada, vai gerar custos para executar a transação e será necessário desembolsar *ethers* da carteira para que a transação seja completada. Já a função `getMessage` não tem custo nenhum para ser executada.

A função `getMessage` está presente neste contrato apenas como exemplo didático. Todas as variáveis declaradas no contrato, automaticamente, terão uma função `get` associada no momento da compilação.

1.3. Desenvolvimento de Contratos Inteligentes

O desenvolvimento de CI em *Solidity* pode ser feito no *Remix*, como mostrado na seção 1.2.2 ou usando um ambiente local, onde se tenha mais liberdade para escolher ferramentas que possam tornar o trabalho mais produtivo. Esta seção apresenta a linguagem *Solidity*, explica como configurar um ambiente de desenvolvimento e logo em seguida, apresenta um CI que será usado com uma DApp.

Ainda não há um Ambiente de Desenvolvimento Integrado (IDE, do inglês *Integrated Development Environment*) ou um ambiente amigável para o desenvolvimento dos contratos e de DApps. A solução encontrada pelos autores e já testada em cursos na Universidade Estadual de Santa Cruz (UESC), Universidade Estadual do Sudoeste da Bahia (UESB) e Universidade Federal da Bahia (UFBA) será descrita abaixo. Toda a documentação, *scripts* e códigos estão disponíveis na página do curso [Abijaude e outros 2020].

1.3.1. A Linguagem Solidity

O primeiro exemplo de um CI foi muito simples e sem explorar recursos da linguagem *Solidity*. Evidentemente que este não é um minicurso de *Solidity*, pois esta é uma linguagem poderosa e em constante evolução. No entanto, serão apresentados pontos da linguagem, como tipos de variáveis, métodos e funções com o objetivo de fornecer uma base suficiente para que os alunos possam explorar sozinhos novos conhecimentos. Os tipos básicos de dados do *Solidity* são listados na tabela 1.1.

Quando criamos uma transação e a enviamos para a rede, algumas informações podem ser encapsuladas na mensagem. Estas opções são pré-definidas pelo *Solidity* e agrupadas em 3 categorias: `msg, block e tx`.

`msg` - O objeto `msg` é uma chamada de transação originada de um cliente *Ethereum* ou de um contrato. Ela contém uma série de atributos úteis:

- `msg.sender`: Representa o endereço que iniciou a chamada de contrato
- `msg.value`: O valor de *ether* enviado com esta chamada (em *wei*).
- `msg.gas`: A quantidade de *gas* restante no suprimento de *gas* desse ambiente de execução. Isso foi descontinuado no *Solidity* v0.4.21 e substituído pela função `gasleft()`.
- `msg.data`: A carga útil de dados desta chamada no contrato.

Tabela 1.1. Tabela com os tipos de dados disponíveis no *Solidity*

Tipo	Descrição
`int`	Inteiros positivos ou negativos (`int`) declarados em incrementos de 8 bits (`int8`, `int16`, ... `int256`).
`uint`	Inteiros positivos declarados em incrementos de 8 bits (`uint8`, `uint16`... `uint256`).
`bool`	Valor lógico, verdadeiro ou falso, com operadores lógicos ! (não), && (e), || (ou), == (igual) e! = (diferente).
`fixed/ ufixed`	Números de ponto fixo, declarados com `(u)fixedMxN` em que M é o tamanho em bits (incrementos de 8 até 256) e N é o número de decimais após o ponto (até 18); por exemplo, `ufixed32x2`.
`address`	Usado para armazenar endereços *Ethereum* de 20 bytes. O objeto de endereço tem muitas funções membro úteis, como `balance` (retorna o saldo da conta) e `transfer` (transfere *ether* para uma conta).
`byte array (fixed)`	Matrizes de bytes de tamanho fixo, declaradas com bytes.
`byte array (dynamic)`	Matrizes de bytes de tamanho variável, declaradas com bytes ou `string`.
`enum`	Tipo definido pelo usuário para enumerar valores discretos `enum name {rotulo1, rotulo2...}`.
`array`	Um `array` de qualquer tipo, fixo ou dinâmico.
`struc`	`Conteiners` de dados definidos pelo usuário para agrupar variáveis `struct Car { String year; int color; }`.
`Mapping`	Tabelas de pesquisa de *hash* para pares chave => `mapping (key_type=>value_type)`.

- `msg.sig`: Os primeiros quatro bytes da carga de dados, que é o seletor de função.

`block` - O objeto de bloco contém informações sobre o bloco atual:

- `block.blockhash(blockNumber)`: O *hash* de um bloco específico. Em desuso e substituído pela função `blockhash()` no *Solidity* v0.4.22.
- `block.coinbase`: O endereço do destinatário das taxas do bloco atual e da recompensa do bloco.
- `block.difficulty`: A dificuldade (prova de trabalho) do bloco atual.
- `block.gaslimit`: A quantidade máxima de *gas* que pode ser gasta em todas as transações incluídas no bloco atual.
- `block.number`: O número do bloco atual.
- `block.timestamp`: O carimbo de data/hora colocado no bloco atual pelo minerador.

`tx` - O objeto *tx* fornece um meio de acessar informações relacionadas à transação:

- `tx.gasprice`: o preço do *gas* na transação de chamada.
- `tx.origin`: O endereço da conta *Ethereum* de origem para esta transação. Esta é uma operação considerada insegura!

O *Solidity* oferece uma facilidade de programação para manipular os dados relativos aos endereços passados como entrada. Eles facilitam a escrita dos contratos e são apresentados na forma de atributos e métodos.Os principais estão listados abaixo:

- `address.balance`: O saldo do endereço, em *wei*. Por exemplo, o saldo do contrato atual é `address(this).balance`.
- `address.transfer(quantidade)`: Transfere o valor (em *wei*) para este endereço, lançando uma exceção para qualquer erro.
- `address.send(quantidade)`: Semelhante ao `transfer`. Ao invés de lançar uma exceção, ele retorna falso em caso de erro.
- `address.call(payload)`: pode construir uma chamada de mensagem arbitrária com uma carga de dados. Retorna falso em caso de erro. Mas o destinatário pode (acidentalmente ou maliciosamente) esgotar todo o seu *gas*, fazendo com que seu contrato seja interrompido com uma exceção.

Dentro de um contrato, é necessário definir funções que podem ser chamadas por uma transação originada em uma carteira *Ethereum* ou em outro contrato. A sintaxe usada para declarar uma função no *Solidity* é a seguinte:

`function FunctionName ([parâmetros]) public|private|internal|external [pure|constant|view|payable] [modifier] [return (tipos de retorno)]`, onde:

`FunctionName` é o nome usado para chamar a função em uma transação de uma carteira *Ethereum*, de outro contrato ou de dentro do mesmo contrato. Uma função pode ser definida sem um nome. Neste caso, é a função de *fallback*, que é chamada quando nenhuma outra função é nomeada. A função de *fallback* não pode ter argumentos ou retornos.

Os parâmetros vêm após o nome, especificado os argumentos que devem ser passados para a função, com seus nomes e tipos.

O próximo atributo especifica a visibilidade da função. O padrão são funções públicas que podem ser chamadas por outros contratos, transações de carteiras *Ethereum*, ou de dentro do contrato. As funções com atributo `external` são como funções públicas, exceto que não podem ser chamadas de dentro do contrato, a menos que explicitamente prefixadas com a palavra-chave this.

As funções com atributo `internal` são acessíveis apenas de dentro do contrato ou por contratos derivados de outro contrato. Elas não podem ser chamadas por outro

contrato ou transações de carteiras *Ethereum*. As funções com o atributo `private` são como funções `internal`, mas não podem ser chamadas por contratos derivados.

Lembre-se de que os termos `internal` e `private` são um tanto enganosos. Qualquer função ou dado dentro de um contrato está sempre visível na blockchain pública, o que significa que qualquer pessoa pode ver o código ou os dados. As palavras-chave descritas aqui afetam apenas como e quando uma função pode ser chamada.

O segundo conjunto de palavras-chave `(pure, constant, view, payable)` afetam o comportamento da função:

Uma função `constant` ou `view` promete não modificar nenhum estado. Os termos possuem o mesmo objetivo e o primeiro será descontinuado em uma versão futura.

Uma função `pure` é aquela que não lê nem grava nenhuma variável no armazenamento. Ele só pode operar em argumentos e retornar dados, sem referência a nenhum dado armazenado.

Uma função `payable` é aquela que pode aceitar pagamentos recebidos. Funções não declaradas como `payable` rejeitarão pagamentos recebidos.

Existem 3 tipos especiais de funções que deve-se ficar atento: construtoras, auto-destruição e modificadoras.

As funções construtoras são executadas apenas uma vez, durante a criação do contrato e possuem a palavra-chave `constructor()`. As funções de auto-destruição possuem a palavra-chave `destroy()` e são utilizadas, como o próprio nome diz, para destruir o contrato implementado. As funções modificadoras são aplicadas adicionando-se o nome do `modifier` na declaração da função. São usados para criar condições que se aplicam a muitas situações em um contrato, e para isto, basta acrescentar o seu nome na declaração de uma função.

1.3.2. Construindo um Contrato Inteligente

Será apresentado agora um novo e mais sofisticado CI, explorando mais as funcionalidades da linguagem Solidity. Este CI simula uma loteria, de forma que os participantes apostam e depois, realiza-se um sorteio ente eles. O ganhador receberá o saldo do contrato em sua conta *Ethereum*.

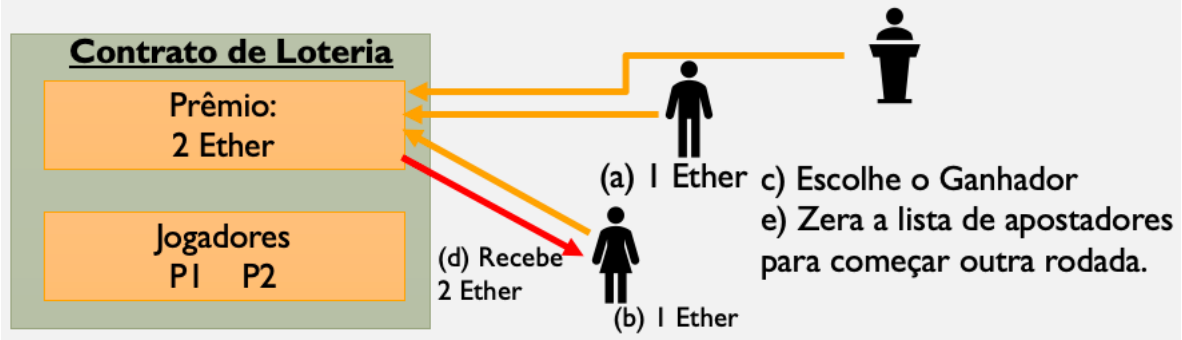

Figura 1.4. Lógica do CI que simula um sorteio entre apostadores.

A Figura 1.4 mostra como funciona a lógica a ser implementada no contrato. Em (a) um jogador aposta 1 ether, enviando-o para o contrato. Em (b), outro jogador aposta mais 1 ether. O gerente, representado pelo endereço da conta que implementou o contrato na rede *Ethereum*, decide fazer o sorteio em (c). Isto executa uma função que faz uma escolha aleatória e conclui que o segundo jogador ganhou, por exemplo, transferindo para ele o saldo de 2 *ethers*, em (d). Terminado o processo, o gerente zera a lista de apostadores e inicia outra rodada em (e).

Este contrato necessita de 5 funções: (`jogar`, `random`, `sorteio`, `verificaGerente`, `getJogadores`) e as variáveis `gerente` e `jogadores`.

O código do contrato é exibido na Figura 1.5. A linha 1 indica sob qual licença de uso está o contrato. A linha 2 informa a versão do compilador. É obrigatório que um contrato inicie com estas informações. A linha 4 define o nome do CI (`Loteria`). Este bloco do contrato começa na linha 4 e vai até a linha 36.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.7.4;
3  contract Loteria{
4      address public gerente;
5      // cria a variável para receber o endereço do gerente
6      address payable[] public jogadores;
7      // cria o array para reeber o endereço dos jogadores com
8      //capacidade de pagamento
9      constructor(){
10         gerente = msg.sender;
11         //atribui o endereço do gerente à variável
12     }
13     function jogar() public payable{
14         require(msg.value > 0.1 ether);
15         jogadores.push(msg.sender);
16         //adiciona no array o endereço do jogador
17     }
18     function random() private view returns (uint){
19         return uint(keccak256(abi.encodePacked(block.difficulty,
20          block.timestamp, jogadores)));
21         // keccak256 gera um hash
22     }
23     function sorteio() public verificaGerente{
24         uint indice = random() % jogadores.length;
25         //usamos o operador modulo (%) para sortear um indice
26         jogadores[indice].transfer(address(this).balance);
27         jogadores = new address payable[](0);
28     }
29     modifier verificaGerente(){
30         require(msg.sender == gerente);
31         _;
32     }
33     function getJogadores() public view returns(address payable[] memory){
34         return jogadores;
35     }
36 }
```

Figura 1.5. Código-fonte do CI `Loteria.sol`, que simula a aposta e sorteio entre jogadores.

As linhas 4 e 6 definem as variáveis a serem utilizadas no contrato. A variável `gerente` é do tipo `address` e de escopo público. Ela armazena o endereço da conta

que implementa o contrato. A variável `jogadores` é um `array` de escopo público, do tipo `payable`, para receber os endereços das contas *Ethereum* que irão realizar apostas, inclusive a conta do próprio administrador. Como estes endereços vão transacionar valores, eles precisam ser do tipo `payable`.

Entre as linhas 9 e 12, define-se um construtor, o qual é executado apenas uma vez durante o ciclo de vida do contrato. Este construtor captura o endereço da conta *Ethereum* que implementou o contrato e armazena-o na variável `gerente`.

A função `jogar()`, entre as linhas 13 e 17, realiza duas tarefas. A primeira é restringir as apostas, permitindo apenas aquelas com valor superior a 0.1 *ether*. A instrução `require` executa esta tarefa. Se o resultado for falso, a execução da função é abandonada e uma exceção é gerada. Se for verdadeiro, como é uma função `payable`, ela transfere o valor para a conta contrato e continua a execução na linha seguinte. Começa então a segunda tarefa: adicionar na matriz de jogadores o endereço da carteira *Ethereum* que fez a aposta.

A função `random()`, entre as linhas 18 e 22, possui escopo privado e não modifica dados no contrato, retornando um inteiro, que representa um *hash* dos parâmetros repassados. Este valor é obtido através da função `keccak256`, que recebe como parâmetros a dificuldade do bloco atual (`block.difficulty`), o tempo em que este bloco foi gerado (`block.timestamp`) e a matriz `jogadores` e retorna um inteiro.

A função `sorteio()`, linhas 23 a 28, é uma função pública que usa a função modificadora `verificaGerente()`. Isto implica que, antes de esta função executar o seu próprio conteúdo, o fluxo do programa é desviado para a função modificadora `verificaGerente()`. Na linha 29 temos a declaração desta função. Na linha 30, compara-se o remetente da mensagem com o endereço da variável `gerente`. Se for verdadeiro, a execução segue para a próxima linha. O símbolo `"_"` quer dizer que o fluxo do programa retornará para a primeira linha da função que invocou a função modificadora. Caso o resultado da expressão `require`, na função modificadora, fosse falso, seria gerada uma exceção e o fluxo não retornaria para a função `sorteio()`.

Considerando que os requisitos da função modificadora foram atendidos, finalmente a função `sorteio()` será executada. A linha 24 declara a variável `indice`, do tipo inteiro, que calcula o módulo do valor gerado pela função `random()` pelo tamanho da matriz de jogadores. Isto garante que o resultado será sempre um número compreendido entre zero e o tamanho desta matriz. A linha 26 localiza o índice na matriz de jogadores e transfere o saldo da conta contrato para este endereço. Na linha 27, a matriz de jogadores é zerada para que seja feita uma nova rodada de apostas.

A última função, `getJogadores()`, é do tipo pública, não modifica dados do contrato, e retorna os endereços da matriz de jogadores. Observe que esta função não é utilizada na lógica do jogo. Ela será usada na DApp desenvolvida na seção 1.4.

1.3.3. Configuração do Ambiente

Esta seção vai detalhar como configurar um ambiente de desenvolvimento de CIs no computador. Existe um guia completo na página do minicurso com informações detalhadas sobre este processo. Lá estão três roteiros práticos para desenvolvimento de CIs que serão

executados na apresentação do minicurso. O projeto que será usado como referência para configuração do ambiente é o Loteria.

O primeiro passo é ter o Node.js e um editor de código de sua preferência instalados e configurados no computador. Após isto, pode-se instalar os pacotes, executar os *scripts* para compilar e implementar os contratos e preparar uma rotina de testes.

Após instalar o Node.js, deve-se criar uma pasta no computador para armazenar os arquivos do projeto. Em seguida, acesse a página do minicurso, escolha o projeto Loteria e siga as instruções. Ao final, a estrutura de diretório do projeto estará pronta e todos os arquivos disponíveis para uso.

Os seguintes pacotes adicionais para o Node.js precisam ser instalados: a) `solc`: compilador *Solidity*; b) `mocha`: *framework* para testar os contratos antes de implementá-los em uma rede BC; c) `web3`: coleção de bibliotecas que permite interagir com um nó *Ethereum* local ou remoto usando HTTP; d) `ganache-cli`: é uma BC pessoal para desenvolvimento rápido de aplicativos distribuídos *Ethereum* e *Corda* em um ambiente seguro e determinístico; e) `truffle-hdwallet-provider`: para realizar as assinaturas usando as palavras mnemônicas.

A instalação de todos estes pacotes pode ser feita facilmente executando, dentro da pasta do projeto, o comando `npm install --save`. Isto pode demorar um pouco, mas, quando o procedimento de instalação for encerrado, seu ambiente estará configurado com todos os arquivos necessários para escrever, compilar e implementar CIs.

Os autores fornecem dois *scripts*, escritos na linguagem JavaScript, que permitem compilar e implementar os CIs. Procure na pasta raiz do projeto por `compile.js` e `deploy.js`. Ambos precisam, para cada contrato, de pequenos ajustes.

O *script* `compile.js` é mostrado na figura 1.6. As linhas de 1 a 6 declaram variáveis com alguns requisitos necessários, como por exemplo o compilador `solc`. Observe que na linha 5 é informado o nome do contrato que se deseja compilar. Neste exemplo, o *script* está preparado para compilar o contrato `Loteria.sol`. O mesmo acontece nas linhas 12, 27 e 29. Estas são as modificações que precisam ser feitas quando este *script* for utilizado em outros contratos. A chamada para o compilador `solc` ocorre na linha 25, armazenando o resultado na variável `contratoCompilado`. É neste momento que o contrato é de fato compilado. A linha 27 exibe no console o resultado da compilação. Ela está aí apenas para fins didáticos, com o objetivo de exibir na tela a saída do compilador. Quando houver a necessidade de compilar um contrato para posterior implementação, a linha 27 deve ser comentada e retirado o `//` da linha 29.

O *script* `deploy.js`, exibido na figura 1.7, trabalha em conjunto com o *script* `compile.js`. Entre as linhas 1 e 4 são definidas variáveis que importam bibliotecas. A linha 5 envia como parâmetro informações que devem ser colocadas no arquivo `.env`, a ser criado no diretório raiz do projeto. Ele deve conter 2 linhas: `mnemonic = 'Suas palavras mnemônicas'` e `provider = 'endereço do site infura'`.

A linha 9 instancia o web3 e passa como referência o endereço da rede Infura para implementação do contrato. A linha 10 declara uma função assíncrona, cujo objetivo é enviar para a blockchain o pedido de implementação do contrato e aguardar

```
1  const path = require("path"); // linhas para indicar o caminho onde o arquivo será lido
2  const fs = require("fs"); // e garantir a compatibilidade de sistemas operacionais
3  const solc = require("solc");
4  // Pega o arquivo Inbox.sol e atribui a variável
5  const LoteriaPath = path.resolve(__dirname, "contracts", "Loteria.sol");
6  const source = fs.readFileSync(LoteriaPath, "utf8");
7  // * Mais informações sobre o input e output
8  // * https://docs.soliditylang.org/en/v0.7.4/using-the-compiler.html#output-description
9  var input = {
10   language: "Solidity",
11   sources: {
12     "Loteria.sol": {
13       content: source,
14     },
15     // Pode-se adicionar otros contratos, caso exista
16   },
17   settings: {
18     outputSelection: {
19       "*": {
20         "*": ["*"],
21       },
22     },
23   },
24 };
25 let contratoCompilado = JSON.parse(solc.compile(JSON.stringify(input)));
26 // Gera o log para investigação
27  console.log(contratoCompilado.contracts["Loteria.sol"].Loteria);
28 // Pedimos apenas o nosso contrato para exportação
29 //module.exports = contratoCompilado.contracts["Loteria.sol"].Loteria;
```

Figura 1.6. *Script* `compile.js` para compilação de contratos inteligentes no ambiente Node.js.

que o processo de mineração seja concluído. A linha 11 e 12 definem a conta onde será debitada as taxas de implementação do contrato. Das linhas 13 a 16 preparamos a chave privada para assinar a transação. Entre as linhas 18 e 20 declaramos a variável `contract` com informações sobre os bytecodes. As linhas de 21 a 24 definem a variável `transactionObject` que define o conteúdo da transação. Nesta parte, pode-se observar que define-se 4.000.000 de unidades de gas. Entre as linhas 26 e 28, assina-se a transação com a chave privada, e finalmente na linha 30 e 31 envia-se o a transação para a rede blockchain.

A execução do *script* fica paralisada. Enquanto isto, esta transação vai para uma piscina de transações da rede *Ethereum* e aguarda a sua vez para ser inserida em um bloco e publicada como válida. Somente depois deste processo é que o *script* prossegue a execução, imprimindo no console o endereço atribuído ao contrato (linha 33) e fechando a conexão com o provedor na linha 37. Anote o endereço do contrato, pois esta informação será necessária quando a DApp for construída na seção 1.4.

Existem dois sites muito importantes que hospedam aplicativos e que complementam as atividades a serem desenvolvidas com os CI. O primeiro (`www.infura.io`) permite o acesso à rede BC, criando um endereço nos servidores *Ethereum* para que se possa implementar nossos SCs. O segundo (`https://faucet.rinkeby.io`) é usado para

```
require("dotenv").config();
const HDWalletProvider = require("@truffle/hdwallet-provider");
const Web3 = require("web3");
const { abi, evm } = require("./compile");
const provider = new HDWalletProvider({
  mnemonic: { phrase: process.env.mnemonic },
  providerOrUrl:process.env.provider,
});
const web3 = new Web3(provider);
const deploy = async () => {
  const accounts = await web3.eth.getAccounts();
  const deploymentAccount = accounts[0];
  const privateKey = provider.wallets[
    accounts[0].toLowerCase()
  ].privateKey.toString("hex");
  console.log("Conta usada para o deploy ", accounts[0]);
  try {
    let contract = await new web3.eth.Contract(abi)
      .deploy({ data: evm.bytecode.object, arguments: [] })
      .encodeABI();
    let transactionObject = {
      gas: 4000000,
      data: contract,
      from: deploymentAccount,
    };
    let signedTransactionObject = await web3.eth.accounts.signTransaction(
      transactionObject,
      "0x" + privateKey
    );
    let result = await web3.eth.sendSignedTransaction(
      signedTransactionObject.rawTransaction
    );
    console.log("Contract deployed to", result.contractAddress);
  } catch (error) {
    console.log(error);
  }
  provider.engine.stop();
};
deploy();
```

Figura 1.7. *Script* `deploy.js` para implementação de contratos na rede *Rinkeby*.

a geração de *ethers* sem valor comercial, empregados nas redes de teste da plataforma *Ethereum*, conforme mencionado anteriormente.

O site *Infura* permite o acesso instantâneo às redes *Ethereum* através de *websockets* ou HTTPS. O serviço permite até 3 projetos gratuitos. O nó *Infura* cria a solicitação de acesso a rede desejada (principal ou teste) e retorna um link, o qual aponta para o endereço válido. Este link deve ser adicionado na segunda linha do arquivo `.env`. Neste momento, os alunos são convidados para acessarem a página do curso e executarem o tutorial da segunda prática.

1.3.4. Rotina de Testes para CI

Os CIs são programas especiais que gerenciam ativos digitais na BC. É difícil recuperar a perda se os usuários fizerem transações por meio de CIs com bugs, que não podem ser corrigidos diretamente. Portanto, é importante garantir a exatidão dos contratos inteligentes antes de implementá-los [Wu e outros 2019].

A referência [Wu e outros 2019] propõe uma estrutura sistemática para teste de mutação para contratos inteligentes no *Ethereum*. Quinze novos operadores de mutação foram projetados para os CIs, em termos de palavra-chave, variável/função global, unidade de variável e tratamento de erros.

A referência [Andesta e outros 2019] propõe um mecanismo de testes para CIs na linguagem *Solidity*, baseado em testes de mutação. Analisando uma lista abrangente de *bugs* conhecidos nos contratos inteligentes do *Solidity* foi possível catalogar 10 classes de operadores de mutação inspirados nas falhas reais.

Uma técnica automatizada, *SolAnalyser* [Akca e outros 2019], para detecção de vulnerabilidade em CIs desenvolvidos em *Solidity* usa análise estática e dinâmica. Ela oferece suporte à detecção automatizada de 8 tipos diferentes de vulnerabilidades que atualmente carecem de amplo suporte nas ferramentas existentes e podem ser facilmente estendidas para oferecer suporte a outros tipos.

Estas referências, no entanto, não apresentaram detalhes suficientes para que, em tempo hábil, fossem analisadas e incluídas nas práticas deste minicurso. Existem ainda muitos artigos versando sobre este tema nas bases de pesquisa.

Optou-se em usar o *framework* `mocha` [OpenJS 2017]. As razões que contribuíram para isto são a farta documentação disponível, a experiência dos autores com este *framework* e o amplo uso deste em projetos envolvendo Javascript e CIs na rede *Ethereum*.

O ciclo de testes do `mocha` é bastante simples e segue 4 passos: Início -> Implementação de um contrato -> Manipulação do Contrato -> Comparações. Uma vez realizado o primeiro e segundo passos, o terceiro e quarto se repetem tanto quanto forem os testes a serem realizados.

A figura 1.8 ilustra um fragmento de código do arquivo de testes. A primeira parte consiste em preparar o teste, importando os requisitos necessários e definindo as variáveis que serão usadas (veja as linhas de 1 a 7). Esta parte representa o primeiro passo (Início).

As linhas de 8 a 13 representam o segundo passo e implementam o contrato na rede `ganache`. Esta rede simula localmente uma rede *Ethereum*. A partir da linha 14 começam efetivamente a manipulação dos contratos e as comparações para testes, agrupados em blocos definidos pelo comando `describe`. Estes são o terceiro e quarto passos, respectivamente. Cada um destes blocos manipula e depois testa o contrato. No exemplo que compreende as linhas de 14 até 18 é feito um teste para verificar se o contrato foi implementado na rede *ganache*. Isto é verificado na linha 17, quando solicita-se que seja retornado o endereço do contrato implementado anteriormente. Caso haja um endereço associado ao contrato, então ele foi implementado com sucesso.

O arquivo `test/Loteria.test.js` contém todo o arquivo de teste. Ao abrí-

```
1   const assert = require("assert");
2   const ganache = require("ganache-cli");
3   const Web3 = require("web3");
4   const web3 = new Web3(ganache.provider());
5   const { abi, evm } = require("../compile");
6   let loteria;
7   let contas;
8   beforeEach(async () => {
9     contas = await web3.eth.getAccounts();
10    loteria = await new web3.eth.Contract(abi)
11      .deploy({ data: evm.bytecode.object })
12      .send({ from: contas[0], gas: "1000000" });
13  });
14  describe("Contrato Loteria", () => {
15    it("Deploy a contract", () => {
16      // console.log(inbox);
17      assert.ok(loteria.options.address);
18    });
```

Figura 1.8. Fragmento do arquivo de testes usado no contrato `Loteria.sol` com o *framework* `mocha`.

lo, observe que foram feitos testes para recuperar o endereço do contrato Loteria; permitir que uma conta seja adicionada ao sorteio; permitir que várias contas sejam adicionadas ao sorteio; verificar a quantidade mínima de ether a ser apostado; verificar se o gerente solicita que seja feito o sorteio; e o teste geral, envolvendo todos os passos do contrato `Loteria.sol`. Para executar os testes basta digitar `npm run test` na pasta do projeto e aguardar o resultado. Há um tutorial na página do curso para guiá-lo.

1.4. Desenvolvendo DApps

Uma DApp é um aplicativo que é parcialmente ou totalmente descentralizado e pode oferecer características únicas, exigindo conhecimentos em novas linguagens e em conceitos até então pouco explorados.

Aspectos como por exemplo o *back-end*, o *front-end*, o armazenamento de dados, a comunicação e até mesmo a questão de resolução de nomes podem ser centralizados ou descentralizados. Por exemplo, um *front-end* pode ser desenvolvido como um aplicativo da web executado em um servidor centralizado ou como um aplicativo móvel executado em seu celular ou *tablet*. O *back-end* e o armazenamento podem estar em servidores privados e bancos de dados proprietários, ou você pode usar um CI e o armazenamento P2P [Antonopoulos e Wood 2018].

Além disto, as DApps oferecem peculiaridades que os sistemas centralizados não conseguem, como por exemplo a resiliência, a transparência e a resistência a censura.

A interface do lado do cliente de uma DApp pode usar tecnologias da web padrão (HTML, CSS, JavaScript, etc.). Isso permite que um desenvolvedor da web tradicional use ferramentas, bibliotecas e estruturas familiares. As interações com a rede *Ethereum*, como assinatura de mensagens, envio de transações e gerenciamento de chaves, geralmente são conduzidas por meio do navegador web utilizando uma extensão como *Meta-*

mask.

O *front-end* é geralmente vinculado à rede *Ethereum* por meio da `web3.js`, uma biblioteca, empacotada com os recursos do próprio *front-end* e servida a um navegador por um servidor web. O *Metamask* injeta uma biblioteca `web3` no navegador, mas para que as DApps funcionem adequadamente, é necessário rejeitar esta biblioteca injetada pelo *Metamask* e injetar a biblioteca `web3.js`. A figura 1.9 ilustra isto. Ambas as bibliotecas conseguem acessar a rede *Rinkeby*, mas a `web3.js`, proveniente da aplicação, deve assumir o controle, conforme discutiremos na seção 1.4.1.

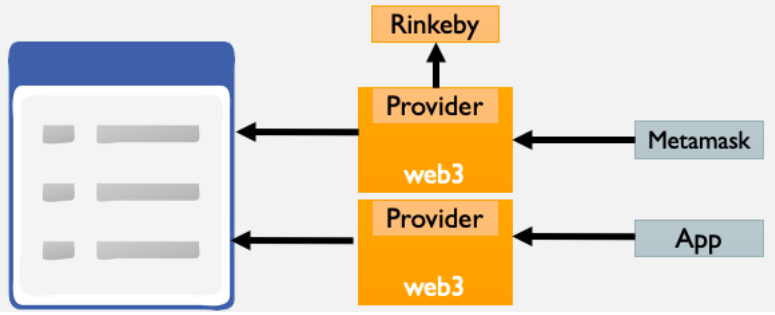

Figura 1.9. Biblioteca `web3` injetada pelo *metamask* x injetada pela aplicação.

1.4.1. Criando a DApp para interagir com o contrato `Loteria.sol`

Esta seção descreve os passos para criar uma DApp que interage com o contrato criado na seção 1.3.2. A página do curso possui o projeto `loteria_react` com todos os arquivos e configurações necessárias, inclusive o tutorial para a prática realizada ao final desta seção. Crie uma nova pasta e salve o projeto `loteria_react` da página do curso. Após isto siga o roteiro descrito para instalar as dependências do projeto.

Para que a DApp funcione, o primeiro passo é implementar o contrato na rede *Rinkeby* e guardar o endereço gerado pelo site *Infura*. Este passo já foi executado no laboratório anterior.

Na estrutura de diretório criada, existe a pasta `src` com 4 arquivos: `web3.js`, `index.js`, `loteria.js` e `App.js`. Os três primeiros preparam o ambiente para a execução da DApp. Todos possuem comentários explicativos e são de fácil entendimento. O arquivo `web3.js` realiza as configurações necessárias para injetar a `web3` na DApp que iremos criar. O arquivo `index.js` importa o `react`, o `react-dom` e o arquivo `app.js`, imprescindíveis para a construção da página web. O arquivo `loteria.js` possui a ABI gerada pela compilação do contrato `loteria.sol`.

O arquivo `app.js` é onde as coisas acontecem. Este arquivo é responsável por exibir uma tela no navegador, conforme a figura 1.10. Observe que há componentes dinâmicos, destacados no retângulo laranja. Estes componentes são capturados diretamente do CI e atualizados todas as vezes que se clica nos botões Jogar ou Sortear.

Ao clicar no botão Jogar, o usuário já deve ter preenchido a quantidade de *ether* a ser enviada com um valor superior a 0.1 *ether*. Caso não faça isto uma exceção será gerada e a aposta não será concluída. A extensão do *Metamask* precisa estar ativa no

Figura 1.10. Tela apresentada pela DApp com os comentários sobre os campos.

navegador para que uma conta seja registrada no momento da aposta.

Ao clicar no botão Sortear, a conta selecionada no *Metamask* será enviada ao contrato para verificar se o endereço coincide com a conta que implementou o contrato. Se a verificação for verdadeira, o sorteio irá acontecer e a conta sorteada receberá os *ethers* da conta contrato, aumentando o seu saldo. Para verificar isto, basta navegar pelas contas no *Metamask* e observar os saldos das contas.

O código do arquivo `App.js` é divido em 5 seções. A primeira seção importa o `react`, com o conjunto de bibliotecas para construção do site; o arquivo `web3.js`, cuja função é trabalhar com as requisições da rede com a BC; e o `loteria.js` com a ABI, necessária para acessar o CI.

A segunda seção é composta pela função assíncrona `carregarDados()`. As linhas de 18 e 20, exibidas na figura 1.11 acionam o contrato para executar as funções `gerente()` e `getJogadores()`, ambas no CI implementado na BC. A linha 22 solicita ao contrato que envie o saldo em *ethers* da conta contrato.

```
15  // Função assincrona que carrega os dados do contrato
16  const carregarDados = async () => {
17    // Pega a carteira do gerente do contrato
18    const _gerente = await loteria.methods.gerente().call();
19    // Pega a carteira dos jogadores
20    const _jogadores = await loteria.methods.getJogadores().call();
21    // Pega o valor total vinculado ao contrato
22    const _saldo = await web3.eth.getBalance(loteria.options.address);
```

Figura 1.11. Função assíncrona para acessar o CI na BC e carregar informações para uso na DApp.

A terceira seção do código possui a função assíncrona `apostar()` (veja figura 1.12). Esta função faz um tratamento de exceções devido à condição do valor mínimo para a aposta. A linha 42 configura uma mensagem para alertar ao usuário, no momento adequado, que ele precisa aguardar a validação da transação de aposta submetida ao CI, que por sua vez depende da publicação desta transação na cadeia de blocos da rede *Ethe-*

reum. A linha 44 usa o `web3` para capturar a conta selecionada no *Metamask*. A linha 48 invoca a função `jogar()` no CI, enviando a conta e o valor que farão parte da aposta. A linha 53 recarrega os dados da página, atualizando a mensagem, assim que a rede *Ethereum* validar a transação feita na linha 48. As linhas de 56 a 62 fazem o tratamento das exceções que podem ocorrer.

```
37  const apostar = async (event) => {
38    try {
39      // Evita que a página seja recarregada
40      event.preventDefault();
41      // Altera valor da mensagem exibida
42      setMensagem("Aguardando a  validação da transação...");
43      // Pega contas do metamask
44      const contas = await web3.eth.getAccounts();
45      // console.log(contas);
46
47      // Joga passando valor da conta principal e o valor de ether em wei
48      await loteria.methods.jogar().send({
49        from: contas[0],
50        value: web3.utils.toWei(value, "ether"),
51      });
52      // Recarrega dados da página
53      await carregarDados();
54      // Altera mensagem
55      setMensagem("Transação concluida!");
56    } catch (error) {
57      // Caso o usuário cancele a solicitação no metamask
58      if (error.code === 4001) {
59        setMensagem("Transação cancelada!");
60      } else {
61        // Caso algo esteja fora das políticas do contrato
62        setMensagem("Transação vai contra regras do contrato");
63      }
64    }
65  };
```

Figura 1.12. Função assíncrona para realizar as apostas. Esta função roda em um bloco *Try/Cath* para tratar as exceções geradas.

A quarta seção descreve a função assíncrona `sortear()` (veja figura 1.13). A linha 72, através da `web3`, captura a conta selecionada no *Metamask* para a variável `contas`. Isto permite a verificação da conta que implementou o contrato, condição exigida para a realização do sorteio. Esta etapa, então é realizada na linha 74, com o envio de uma transação para a rede *Ethereum*, invocando a função do CI `sorteio()`. Antes de executar esta função, o fluxo de execução do CI será redirecionado para a função modificadora `verificaGerente()`. Se a conta ativa no momento da execução for a mesma que implementou o contrato, a função do CI `sorteio()` segue a execução do código e sorteia um ganhador para a rodada. Após o processo de validação desta transação, a mensagem é exibida na tela da DApp para o usuário (linhas 78 e 80). O bloco de linhas entre 81 e 87 trata as possíveis exceções que podem acontecer.

A quinta e última seção, compreendida entre as linhas 92 e 118 exibe a tela da DApp. O código está ilustrado na figura 1.14. A linha 97 consulta o contrato, solicitando o valor da variável `saldo`. Este valor é obtido do CI `Loteria.sol` na linha 22. Esta informação é dinâmica e atualizada cada vez que a página é lida pelo navegador. A linha

```
67  const sortear = async () => {
68    try {
69      // Altera mensagem
70      setMensagem("Aguardando processamento...");
71      // Pega contas do metamask
72      const contas = await web3.eth.getAccounts();
73      // Solicita sorte e manda conta que está realizando o sorteio
74      await loteria.methods.sorteio().send({
75        from: contas[0],
76      });
77      // Recarrega dados da página
78      await carregarDados();
79      // Altera mensagem
80      setMensagem("Um vencedor foi escolhido!");
81    } catch (error) {
82      // Caso o usuário cancele a solicitação no metamask
83      if (error.code === 4001) {
84        setMensagem("Transação cancelada!");
85      } else {
86        // Caso algo esteja fora das políticas do contrato
87        setMensagem("Transação vai contra regras do contrato");
88      }
89    }
90  };
```

Figura 1.13. Função assíncrona para realizar o sorteio entre as contas que apostaram.

100 invoca a função local `apostar()` quando o botão Jogar do formulário (definido na linha 110) for acionado. E por fim, quando o botão Sorteio, definido na linha 114 for acionado, a função local `sortear()` será invocada.

```
92      <div>
93        <h2>Contrato de Loteria</h2>
94        <p>Este contrato é gerenciado por {gerente}</p>
95        <p>
96          Existem agora {jogadores.length} pessoas apostando para ganhar{" "}
97          {web3.utils.fromWei(saldo, "ether")} ether
98        </p>
99        <br />
100       <form onSubmit={apostar}>
101         <h4>Quanto deseja apostar?</h4>
102         <div>
103           <label>Quantidade de ether para ser enviado: </label>
104           <input
105             value={value}
106             // Altera o valor que está sendo apostado
107             onChange={(event) => setValue(event.target.value)}
108           />
109         </div>
110         <button>Jogar</button>
111       </form>
112       <hr />
113       <h4>Realizar sorteio? </h4>
114       <button onClick={sortear}> Sortear</button>
115       <hr />
116       {/* Mostra mensagem ao usuário */}
117       <h1>{mensagem}</h1>
118     </div>
119   );
120 };
```

Figura 1.14. Código para exibição da tela da DApp.

1.5. Como montar um curso de Desenvolvimento Web com Blockchain e Contratos Inteligentes

A programação de CIs não é uma tarefa trivial. Envolve conceitos, propriedades e conhecimentos que vão além da linguagem de programação *Solidity*. Existe uma escassez de material teórico e prático para o ensino de tecnologias emergentes como Blockchain, CIs e desenvolvimento de DApps.

Uma alternativa para isto são as plataformas MOOC (*Massive Open Online Course*, como Udemy, Coursera, edX. Algumas Universidades promovem cursos de extensão ou treinamentos para seus alunos [Rao e Dave 2019, Delmolino e outros 2016, Dettling 2018, Araujo e outros 2019].

Os autores deste minicurso elaboraram e ministraram um curso em três universidades na Bahia: A Universidade Estadual de Santa Cruz (UESC), a Universidade Estadual do Sudoeste da Bahia (UESB) e a Universidade Federal da Bahia UFBA).

Os cursos são compostos de três módulos. O primeiro, básico, serviu de base para a criação deste minicurso e aborda conceitos de BC, CIs e DApps, criando um contrato simples e uma DApp. O segundo, explora mais profundamente as funções da linguagem *Solidity*, construindo um contrato bem mais complexo, utilizando técnicas de engenharia de software, e criando uma DApp multipágina. O terceiro módulo, ainda em elaboração, prevê a integração com a Internet das Coisas, coletando dados dos sensores, armazenando-os em CIs e disparando ações quando determinadas situações forem alcançadas.

O hardware empregado para o desenvolvimento dos laboratórios é bastante simples, uma vez que não há plataformas que necessitem de muitos recursos computacionais. Nos 3 treinamentos ministrados havia computadores equipados com processadores que vão desde o Core 2 Duo com 4 Gb de RAM, até o Core i7 com 32 Mb de RAM. O espaço em disco também não é um fator limitante, uma vez que a maioria das máquinas possui espaço de armazenamento suficiente os experimentos realizados.

O conjunto de softwares necessários à realização de todas as atividades práticas do treinamento é composto por navegadores, extensões para navegadores, ferramentas, pacotes e aplicativos hospedados em sites. Os softwares são compatíveis com praticamente todos os sistemas operacionais, como por exemplo Windows, Linux, Unix e MacOs.

O curso básico foi ministrado com carga horária de 16h. Em 2 dias de aula, com 4 horas no período da manhã e 4 horas no período da tarde. No primeiro dia, durante o período da manhã foi abordada toda a parte teórica e conceitual da Blockchain com ênfase na plataforma Ethereum, além da apresentação do editor de contratos on-line *Remix*. Durante o período da tarde, configuramos as máquinas locais e realizamos rotinas de testes.

No segundo dia, durante o período da manhã apresentamos mais um pouco de teoria com foco na interação com as redes *Ethereum*. Aprendemos mais um pouco sobre a linguagem de programação *Solidity* e escrevemos, compilamos, testamos e implementamos um contrato na rede *Rinkeby*. No período da tarde desenvolvemos uma DApp que interagia com o contrato implementado.

É possível encontrar mais recursos na internet, como por exemplo:

a) *CryptoZombies*: uma plataforma online onde o intuito é ensinar sobre contratos inteligentes de forma interativa. O usuário desenvolve um jogo com foco em zumbis onde a logística é administrada por um contrato e com interação visual através de html, css e javascript. Disponível em `https://cryptozombies.io/pt/`.

b) *Ethernaut*: uma plataforma online que apresenta diversos tutoriais voltados para jogos. Ela foca puramente na criação de contratos, sem implementação visual. Alguns exemplos possibilitam a interação através da ferramenta do desenvolvedor do navegador. Disponível em `https://ethernaut.openzeppelin.com/`.

c) *Vyper Tutorials*: semelhante a ideia do CryptoZombies, esta plataforma propõe a criação de um jogo de *pokémon*. Atualmente está em fase de desenvolvimento, mas já é possível aprender como funciona a criação de contratos. Futuramente serão adicionados interações através de interface assim como *CryptoZombies*. Disponível em `https://vyper.fun/#/`.

d) *Ethereum Studio*: uma ferramenta para desenvolvedores que desejam aprender sobre como construir aplicações na rede *Ethereum*. Os modelos ensinam como escrever um contrato inteligente, implementá-lo e interagir com os CIs por meio de um aplicativo baseado na web. Disponível em `https://studio.ethereum.org/`.

1.6. Desafios e Perspectivas

Embora existam ferramentas que auxiliam no desenvolvimento e nos testes de CIs e da BC, ainda há desafios em aberto. Nesta seção, serão apresentados alguns destes desafios e possíveis propostas que estão em andamento para solucionar problemas encontrados e aperfeiçoar os métodos de desenvolvimento e análise.

Considerando o que foi explicado anteriormente, percebe-se que ainda existe um problema no que se refere à garantia da segurança no desenvolvimento de CIs. Como explica [Atzei e outros 2016], mesmo existindo ferramentas que auxiliam na verificação destas falhas, o uso de uma linguagem *Turing*-completa (como o *Solidity*) pode limitar o processo de verificação. A sugestão é a criação de linguagens não *Turing*-completas para este fim, sendo uma direção a ser seguida pela literatura. Partindo deste princípio, [Jansen e outros 2019] faz um estudo para avaliar se realmente os contratos da rede *Ethereum* utilizam todo o aparato de controle de fluxo oferecido pelo *Solidity*. O resultado deste estudo mostrou que uma pequena parte dos contratos analisados (35,3% de 53757 contratos) utilizam os mecanismos complexos de controle de fluxo oferecidos pela linguagem, e afirmam portanto, que o uso de linguagens não *Turing*-completas no contexto de sistemas baseados em blockchain faria todo o sentido.

Há uma grande quantidade de artigos buscando aprimorar a forma como os CIs são desenvolvidos, de forma a aumentar o nível de abstração para que, além de abstrair a estrutura de implementação, diminua a responsabilidade do desenvolvedor de ter que analisar o CI a fim de encontrar possíveis falhas. A referência [Frantz e Nowostawski 2016] utiliza uma estrutura de gramática institucional para representar CIs, de modo a permitir que tanto desenvolvedores quanto outras pessoas possam ser capazes de criar uma estrutura e, a partir desta, gerar automaticamente o código do CI. Uma ferramenta semelhante é proposta por [Mavridou e Laszka 2018], que utiliza estrutura de máquina de estados finita

para modelar e gerar os códigos dos CIs já aplicando métricas para evitar falhas de segurança. Como uma forma de abstrair ainda mais, [Qin e outros 2019] propõe transformar a linguagem natural em código de CI, para isso, utiliza uma gramática como um dicionário para ajudar na representação do CI em linguagem melhor entendível pelo humano.

A verificação formal é outro desafio que contribui para o estágio de análise de CIs. Com o objetivo de investigar se o CI está se comportando de acordo com a especificação correta, aplica-se uma prova matemática e constrói-se um modelo formal do CI para certificar que este comportamento é válido [Wang e outros 2019]. Alguns métodos de verificação formal são abordados em [Murray e Anisi 2019], mas observa-se que é uma área relativamente nova e ainda não existem padrões definidos. A referência [Bhargavan e outros 2016] apresenta uma forma de traduzir o código *Solidity* e os *bytecodes* da EVM em F*, uma linguagem funcional para verificação de CIs. É uma ferramenta que não suporta toda a sintaxe do *Solidity*, mas é capaz de encontrar algumas vulnerabilidades e erros de sintaxe. A referência [Amani e outros 2018], propõe descompilar os *bytecodes* para dividir o código em blocos e utilizar um provador lógico *Isabelle/HOL* para finalizar a verificação do CI.

Junto com as diversas plataformas blockchain existentes, variadas linguagens foram criadas para implementação de CIs. Em cada uma delas, aumenta-se a complexidade em desenvolver contratos para diferentes plataformas [Coblenz e outros 2019]. Pensando em uma solução para este e outros desafios apresentados, uma nova linguagem de programação de CIs que envolve propriedades que são comuns às linguagens de várias plataformas BC, além de oferecer verificação formal e análises de CIs de forma automática é proposta por [Coblenz e outros 2020].

Diante deste cenário que os autores apresentaram, espera-se que este material auxilie e encoraje os pesquisadores e alunos a compreenderem as relações entre a Blockchain, os Contrato Inteligentes e os Sistemas Web a fim de que possam ampliar suas pesquisas e trabalhos correlatos.

Referências

[Abijaude e outros 2020] Abijaude, J., Serra, H., Santiago, L., Sobreira, P., e Greve, F. (2020). Blockchain, contratos inteligentes sistemas web:teoria e prática. `https://github.com/lifuesc/minicurso-blockchain`. Acessado em 03/03/2021.

[Akca e outros 2019] Akca, S., Rajan, A., e Peng, C. (2019). Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489. IEEE.

[Amani e outros 2018] Amani, S., Bégel, M., Bortin, M., e Staples, M. (2018). Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 66–77.

[Andesta e outros 2019] Andesta, E., Faghih, F., e Fooladgar, M. (2019). Testing smart contracts gets smarter. *arXiv preprint arXiv:1912.04780*.

[Androulaki e outros 2018] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., e outros. (2018). Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM.

[Antonopoulos e Wood 2018] Antonopoulos, A. M. e Wood, G. (2018). *Mastering ethereum: building smart contracts and dapps*. O'reilly Media.

[Araujo e outros 2019] Araujo, P., Viana, W., Veras, N., Farias, E. J., e de Castro Filho, J. A. (2019). Exploring students perceptions and performance in flipped classroom designed with adaptive learning techniques: A study in distributed systems courses. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, volume 30, page 219.

[Atzei e outros 2016] Atzei, N., Bartoletti, M., e Cimoli, T. (2016). A survey of attacks on ethereum smart contracts. *IACR Cryptol. ePrint Arch.*, 2016:1007.

[Bhargavan e outros 2016] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., e outros. (2016). Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96.

[Buterin e outros. 2014] Buterin, V. e outros. (2014). A next-generation smart contract and decentralized application platform. *white paper*, 3(37).

[Coblenz e outros 2020] Coblenz, M., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B. A., Sunshine, J., e Aldrich, J. (2020). Obsidian: Typestate and assets for safer blockchain programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(3):1–82.

[Coblenz e outros 2019] Coblenz, M., Sunshine, J., Aldrich, J., e Myers, B. (2019). Smarter smart contract development tools. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 48–51. IEEE.

[Coinbase 2018] Coinbase (2018). Coinbase wallet. `https://wallet.coinbase.com/`. Acessado em 03/03/2021.

[Delmolino e outros 2016] Delmolino, K., Arnett, M., Kosba, A., Miller, A., e Shi, E. (2016). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International conference on financial cryptography and data security*, pages 79–94. Springer.

[Dettling 2018] Dettling, W. (2018). How to teach blockchain in a business school. In *Business Information Systems and Technology 4.0*, pages 213–225. Springer.

[Ethfiddle 2017] Ethfiddle (2017). Ethfiddle editor rinkeby. `https://ethfiddle.com/`. Acessado em 03/03/2021.

[Frantz e Nowostawski 2016] Frantz, C. K. e Nowostawski, M. (2016). From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 210–215. IEEE.

[GasStation 2017] GasStation (2017). Gas station. `https://ethgasstation.info/index.php`. Acessado em 03/03/2021.

[Go-ethereum 2013] Go-ethereum, T. (2013). Go ethereum - official go implementation of the ethereum protocol. `https://geth.ethereum.org/`. Acessado em 03/03/2021.

[Greve e outros 2018] Greve, F. G., Sampaio, L. S., Abijaude, J. A., Coutinho, A. C., Valcy, Í. V., e Queiroz, S. Q. (2018). Blockchain e a revolução do consenso sob demanda. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)-Minicursos*.

[Jansen e outros 2019] Jansen, M., Hdhili, F., Gouiaa, R., e Qasem, Z. (2019). Do smart contract languages need to be turing complete? In *International Congress on Blockchain and Applications*, pages 19–26. Springer.

[Jaxx 2018] Jaxx (2018). Jaxx safely manager ethereum. `https://jaxx.io/`. Acessado em 03/03/2021.

[LeMahieu 2018] LeMahieu, C. (2018). Nano: A feeless distributed cryptocurrency network. *Nano [Online resource]. URL: https://nano. org/en/whitepaper (date of access: 24.03. 2018)*.

[Mavridou e Laszka 2018] Mavridou, A. e Laszka, A. (2018). Tool demonstration: Fsolidm for designing secure ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 270–277. Springer.

[Metamask 2018] Metamask (2018). Metamask crypto wallet and gateway. `https://metamask.io/`. Acessado em 03/03/2021.

[Murray e Anisi 2019] Murray, Y. e Anisi, D. A. (2019). Survey of formal verification methods for smart contracts on blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6. IEEE.

[MyCrypto 2019] MyCrypto (2019). Mycrypto. `https://mycrypto.com/`. Acessado em 03/03/2021.

[Myetherwallet 2019] Myetherwallet (2019). Myetherwallet original wallet. `https://www.myetherwallet.com/`. Acessado em 03/03/2021.

[Nakamoto 2008] Nakamoto, S. (2008). A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 4. Acessado em 03/03/2021.

[OpenJS 2017] OpenJS (2017). Mocha test framework. `https://mochajs.org/`. Acessado em 03/03/2021.

[Popov 2018] Popov, S. (2018). The tangle, iota whitepaper. https://iota.org/IOTA_Whitepaper.pdf. Acessado em 03/03/2021.

[Qin e outros 2019] Qin, P., Guo, J., Shen, B., e Hu, Q. (2019). Towards self-automatable and unambiguous smart contracts: Machine natural language. In *International Conference on e-Business Engineering*, pages 479–491. Springer.

[Rao e Dave 2019] Rao, A. R. e Dave, R. (2019). Developing hands-on laboratory exercises for teaching stem students the internet-of-things, cloud computing and blockchain applications. In *2019 IEEE Integrated STEM Education Conference (ISEC)*, pages 191–198. IEEE.

[Remix 2015] Remix (2015). Remix ide. https://remix.ethereum.org/. Acessado em 03/03/2021.

[Status 2019] Status (2019). Status private, secure communication. https://status.im/. Acessado em 03/03/2021.

[StudioEthereum 2019] StudioEthereum (2019). Studio ethereum. https://studio.ethereum.org//. Acessado em 03/03/2021.

[Szabo 1997] Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday*.

[Trust 2019] Trust (2019). Trust wallet - secure crypto wallet. https://trustwallet.com/. Acessado em 03/03/2021.

[Wang e outros 2019] Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., e Wang, F.-Y. (2019). Blockchain-enabled smart contracts: architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277.

[Web3js 2016] Web3js (2016). Ethereum javascript api. https://web3js.readthedocs.io/en/v1.3.0/index.html. Acessado em 03/03/2021.

[Wiki 2017] Wiki, E. (2017). Ethash. *GitHub Ethereum Wiki. https://github.com/ethereum/wiki/wiki/Ethash*. Acessado em 03/03/2021.

[Wood e outros. 2014] Wood, G. e outros. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32.

[Wu e outros 2019] Wu, H., Wang, X., Xu, J., Zou, W., Zhang, L., e Chen, Z. (2019). Mutation testing for ethereum smart contract. *arXiv preprint arXiv:1908.03707*.