UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Verificação formal para detecção de vulnerabilidades em contratos inteligentes

Gustavo Oliveira Dias

Qualificação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP SÃO CARLOS

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Gustavo Oliveira Dias

Verificação formal para detecção de vulnerabilidades em contratos inteligentes

Monografia apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, para o Exame de Qualificação, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional.

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
Maio de 2021

Gustavo Oliveira Dias

Formal verification for vulnerability detection in smart contracts

Monograph submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – as part of the qualifying exam requisites of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science.

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Adenilso da Silva Simão

USP – São Carlos
May 2021

RESUMO

DIAS, O. D. **Verificação formal para detecção de vulnerabilidades em contratos inteligentes**. 2021. 96 p. Monografia (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

A tecnologia blockchain, promovida pela criptomoeda Bitcoin, ficou conhecida pela sua capacidade de realizar o gerenciamento de posse descentralizado de criptomoedas por meio de um livro-razão distribuído. Com a introdução da blockchain Ethereum e da linguagem de programação Solidity, foi possível a implantação de contratos inteligentes, que são programas de computador que expressam cláusulas, condições e acordos estabelecidos entre as partes envolvidas. Uma vez implantados, esses contratos executam de forma autônoma, imutável e descentralizada, sem a necessidade de confiar em uma terceira parte reguladora. Desta forma, expandiram-se as aplicações da tecnologia blockchain, que passaram a abranger aplicações descentralizadas, Organizações Autônomas Descentralizadas, governança, finanças, cuidados médicos, entre outras áreas. Em tais aplicações, geralmente há movimentações e gerenciamento de grandes quantias monetárias feitas em Ether, a criptomoeda nativa da Ethereum. Devido a este fator, os contratos inteligentes foram alvos de diversos ataques, que causaram graves perdas financeiras. Nesses ataques, são exploradas vulnerabilidades encontradas no código dos contratos, visto que, em razão da imutabilidade da blockchain, uma vez implantado, um contrato não pode ter seu código atualizado. No primeiro desses ataques, uma aplicação de *crowdfunding* teve todos os seus fundos roubados por um invasor, que transferiu cerca de 150 milhões de dólares em Ether para sua conta, o que despertou a atenção da academia e da indústria para a segurança dos contratos inteligentes. Esta pesquisa tem o objetivo de propor uma estratégia para verificação formal de contratos inteligentes escritos em Solidity para detecção das vulnerabilidades de reentrância, *delegatecall injection* e contrato suicida, por meio da técnica de *model checking*. Para isso, os contratos devem ser convertidos em um modelo formal baseado em estados, e as propriedades de segurança devem ser especificadas, para então dar início ao processo de verificação. Caso alguma das propriedades sejam violadas, será fornecido um contra-exemplo indicando o caminho de execução até a violação, bem como a identificação da vulnerabilidade detectada. Além das vulnerabilidades que são alvos desta pesquisa, também será possível a verificação de propriedades relativas aos requisitos dos contratos. Para evitar trabalhos manuais e a necessidade de conhecimento especializado em métodos formais, a conversão do código deve ser realizada automaticamente, e, para verificação das propriedades, será fornecido um modelo para especificação em linguagem natural. Para validação da proposta, será aplicado um experimento sobre uma amostra de contratos com vulnerabilidades conhecidas para avaliar a eficácia da verificação e coletar dados de desempenho como tempo de execução, consumo de memória e processamento. Além disso, será elaborado ao menos um estudo de caso para

verificação de propriedades funcionais relacionadas aos requisitos do contrato. Por meio da proposta exposta neste trabalho, pretende-se contribuir para o aprimoramento da segurança dos contratos inteligentes, assim como prover um meio simplificado para utilização do *model checking*.

Palavras-chave: Blockchain, Livro-razão distribuído, Contrato inteligente, Verificação formal, *Model-checking*.

ABSTRACT

DIAS, O. D. **Formal verification for vulnerability detection in smart contracts**. 2021. 96 p. Monografia (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Blockchain technology, pioneered by the Bitcoin cryptocurrency, became known for its ability to perform decentralized cryptocurrency ownership management through a distributed ledger. With the introduction of the Ethereum blockchain and the Solidity programming language, it was possible to deploy smart contracts, computer programs that express clauses, conditions and agreements established among the parties involved. Once deployed, these contracts execute in an autonomous, immutable and decentralized way, without relying on a third regulatory party. Therefore, the applications of blockchain technology were expanded, which became to include decentralized applications, Decentralized Autonomous Organizations, governance, finance, health care, among other areas. In such applications, there are usually financial transactions and management of large amounts of Ether, the native cryptocurrency of Ethereum. Such a factor made smart contracts the target of several attacks, which caused severe financial losses. In these attacks, the attacker exploits vulnerabilities found in the contract code. Due to the immutability of the blockchain, once a contract is implemented, it cannot be updated. In the first of these attacks, an attacker stoles all the funds of a crowdfunding application, who transferred about $150 million in Ether to his account, which have attracted attention from academia and industry to the security of smart contracts. This research aims to propose a strategy for formal verification of smart contracts written in Solidity to detect vulnerabilities of reentrancy, delegatecall injection and suicide contract, through the technique of model checking. To this aim, the contracts must be converted into a formal state-based model, and security properties have to be specified, then the verification process can begin. If any of the properties are violated, a counter-example will be provided indicating the execution path until the violation and identifying the detected vulnerability. In addition to the vulnerabilities that are the targets of this research, it will also be possible to verify properties related to the requirements of the contracts. In order to avoid manual work and the need for specialized knowledge in formal methods, the conversion of the code must be carried out automatically, and to verify the properties, a model for specification in natural language will be provided. To validate the proposal, an experiment will be applied on a sample of contracts with known vulnerabilities to assess the effectiveness of the verification and collect performance data such as execution time, memory, and processing consumption. In addition, will be prepared at least one case study to verify functional properties related to the contract's requirements. Through the proposal exposed in this work, it is intended to contribute to the improvement of the security of smart contracts and provide a simple way to use model checking.

Keywords: Blockchain, Distributed Ledger, Smart contract, Formal verification, Model checking.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CC	conta de contrato
CI	contratos inteligentes
CPE	Conta de Propriedade Externa
CTL	*computational tree logic*
DLT	*Distributed Ledger Technology*
GFC	grafo de fluxo de controle
IA	inteligência artificial
LTL	*linear tree logic*
MEF	máquina de estados finito
MS	Mapeamento Sistemático
MVE	Máquina Virtual Ethereum
P2P	*peer-to-peer*
PBFT	*Practical Byzantine Fault Tolerance*
PoS	*Proof-of-Stake*
PoW	*Proof-of-Work*
RIE	representação intermediária estruturada
SMT-solver	Satisfiability Modulo Theories solvers

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

Blockchain é o nome dado à tecnologia subjacente utilizada em diversas plataformas de gerenciamento descentralizado de posse de bens digitais baseada em livro-razão distribuído (do inglês, *Distributed Ledger Technology* (DLT)) (KANNENGIESSER *et al.*, 2020). Essa tecnologia tem como principais características o armazenamento descentralizado e distribuído, a imutabilidade, a transparência e a dispensa da necessidade de confiança em uma terceira parte (FAN *et al.*, 2020; DINH *et al.*, 2018). O primeiro caso de êxito na aplicação da blockchain foi proposto por Nakamoto (2008), que apresentou a Bitcoin, uma criptomoeda gerada e gerenciada de forma distribuída e sem entidades centralizadoras (ZHANG; XUE; LIU, 2019). A geração e o gerenciamento de posse de unidades de Bitcoin são realizados por uma rede de nós conectados auto-gerenciáveis que trabalham para manter a integridade do sistema (DINH *et al.*, 2018).

A geração e gerenciamento de posse de criptomoedas é uma dentre diversas aplicações baseadas na DLT, ou seja, é apenas um fim para um meio (DRESCHER, 2017). Desde o seu surgimento, a blockchain tem passado por várias transformações, o que possibilitou sua aplicação em diversas áreas do conhecimento, como finanças, governo, Internet das Coisas, inteligência artificial (IA), saúde, entre outras (SWAN, 2015; MAESA; MORI, 2020; ZHU *et al.*, 2019; SALAH *et al.*, 2019; AGUIAR *et al.*, 2020).

Um fator crucial para impulsionar o avanço das DLTs foi a introdução dos contratos inteligentes (CI) (MAESA; MORI, 2020). A plataforma baseada em DLT, Ethereum, proposta por Buterin (2014), possibilitou a execução de CIs de forma descentralizada em uma rede ponto-a-ponto (do inglês, *peer-to-peer* (P2P)). Um CI consiste em um conjunto de cláusulas e condições, que são definidas entre as partes envolvidas e expressas por meio de uma linguagem de programação (ZHENG *et al.*, 2020). Depois de escrito, o contrato é implantado em uma blockchain e executado de forma autônoma, automática, e imutável (ZHENG *et al.*, 2020; KANNENGIESSER *et al.*, 2020).

Aplicações que executam sobre a plataforma Ethereum geralmente envolvem movimenta-

ções de grandes quantias de sua criptomoeda nativa, o Ether. Assim, essas aplicações tornaram-se alvos de diversos ataques que causaram transtornos e graves perdas financeiras (ATZEI; BARTOLETTI; CIMOLI, 2017; CHEN *et al.*, 2020a). O primeiro e um dos ataques mais conhecidos aconteceu em 2016 contra o *The DAO* (sigla para *Decentralized Autonomous Organization*), um projeto de *crowdfunding* que arrecadou cerca de 150 milhões de dólares (CHEN *et al.*, 2020a). Neste ataque, um contrato malicioso explorou uma falha no código e transferiu cerca de 3,6 milhões de Ether para sua conta, o equivalente a 50 milhões de dólares (CHEN *et al.*, 2020a; SIEGEL, 2020; ATZEI; BARTOLETTI; CIMOLI, 2017).

Grande parte dos ataques deve-se à exploração de vulnerabilidades encontradas nos CIs (CHEN *et al.*, 2020a; ATZEI; BARTOLETTI; CIMOLI, 2017; LIU; LIU, 2019). Devido à imutabilidade da blockchain, uma vez implantados, os CIs não podem ser alterados, e, portanto, não há como corrigir erros e vulnerabilidades contidos no código, o que ressalta a necessidade de identificá-los na fase de pré-implantação (VACCA *et al.*, 2020; DIKA; NOWOSTAWSKI, 2018). Os CIs são geralmente escritos em Solidity [1], uma linguagem de programação de alto nível, Turing-completa, e desenvolvida especialmente para escrever CIs para a plataforma Ethereum (VARELA-VACA; QUINTERO, 2021). Segundo Atzei, Bartoletti e Cimoli (2017) parte desses erros são ocasionados pelo desalinhamento que há entre a semântica da linguagem Solidity e a intuição dos desenvolvedores.

Há uma série de vulnerabilidades descritas na literatura que são encontradas em CIs implantados na Ethereum (ATZEI; BARTOLETTI; CIMOLI, 2017; CHEN *et al.*, 2020a; DIKA; NOWOSTAWSKI, 2018). No ataque cometido contra o *The DAO*, conhecido como *The DAO Attack*, foi explorada a vulnerabilidade de reentrância, que ocorre quando um contrato permite que uma função seja chamada recursivamente, mas o estado do contrato só é atualizado após a chamada. Na ocasião, uma função de saque do *The DAO* foi invocada sucessivas vezes até esgotar todos os fundos do contrato, pois a atualização da variável que limitaria o quantidade a ser sacada só ocorria após a chamada recursiva (ATZEI; BARTOLETTI; CIMOLI, 2017). Outros ataques que ganharam notoriedade foram os cometidos contra a carteira multi-assinatura *Parity Multsignature Wallet*, uma carteira de criptomoedas descentralizada na qual as transferências são realizadas mediante autorização de um grupo de usuários. Neste caso, um invasor se aproveitou de duas vulnerabilidades presentes no contrato: *delegatecall injection*; e contrato suicida. A primeira permitiu que o invasor atribuísse a si mesmo a posse do contrato, que então transferiu 31 milhões de dólares em Ether para sua conta. Na última, mesmo após uma tentativa de correção do código da aplicação, o invasor novamente obteve a posse indevida do contrato, e em seguida invocou uma função para destruição do contrato, resultando no bloqueio permanente de 280 milhões de dólares em Ether associados às carteiras cadastradas (CHEN *et al.*, 2020a; DESTEFANIS *et al.*, 2018; MANNING, 2018).

[1] Solidity documentation. <https://docs.soliditylang.org/en/develop/index.html>

1.1 Motivação

Motivadas pela existência de riscos à segurança das aplicações baseadas em CIs, diversas estratégias foram utilizadas no intuito de mitigar os riscos envolvidos, como exposto nos trabalhos de Liu e Liu (2019), Chen *et al.* (2020a), Sayeed, Marco-Gisbert e Caira (2020) e Singh *et al.* (2020). Segundo Dika e Nowostawski (2018), a forma mais efetiva para identificação de vulnerabilidades antes da implementação dos CIs é por meio da contratação de serviços de auditoria. Porém, tais serviços podem ser muito custosos para pequenas empresas e desenvolvedores individuais (DIKA; NOWOSTAWSKI, 2018). No estudo de Chen *et al.* (2020a), ressalta-se que as duas melhores formas de prevenir-se de vulnerabilidades consistem na escrita de contratos livres de erros por meio de boas práticas de programação, e, em seguida, na utilização de analisadores ou verificadores de código.

Na pesquisa de Almakhour *et al.* (2020), as abordagens para verificação de CIs são separadas em dois aspectos: (i) verificação formal para correção; (ii) e detecção de vulnerabilidades para garantia de segurança. A verificação formal para correção consiste na representação formal do programa por meio de métodos matemáticos, denominado o processo de modelagem do programa. Uma vez modelado, propriedades que representam a ocorrência de vulnerabilidades ou de erros lógicos são definidas, e então um processo de verificação é executado em busca de violações das propriedades (ALMAKHOUR *et al.*, 2020; SINGH *et al.*, 2020). A detecção de vulnerabilidades para garantia de segurança baseia-se na definição de padrões de vulnerabilidades conhecidas para que, por meio de ferramentas, se execute uma análise sobre o código para então detectá-las (ALMAKHOUR *et al.*, 2020).

Tanto a garantia de segurança de CIs quanto a própria tecnologia blockchain representam áreas de pesquisa relativamente novas e emergentes (CHEN *et al.*, 2020a; KANNENGIESSER *et al.*, 2020). Portanto, ainda não há uma abordagem ou ferramenta padronizadas para garantia de segurança dos CIs. Além disso, também há limitações nas abordagens existentes. As ferramentas para análise de código apresentam taxas consideráveis de falsos positivos e falsos negativos, e mostraram-se ineficientes para contratos complexos (KIM; LEE, 2020). As técnicas de verificação formal são baseadas em métodos formais, e costumam exigir conhecimento especializado para modelagem matemática dos contratos, além de limitações de tempo e memória (CHEN *et al.*, 2020a).

Desta forma, técnicas para verificação formal são aplicadas geralmente em sistemas críticos, em que falhas podem levar à graves prejuízos, como no caso dos CIs. Ademais, a verificação formal possui maior precisão, e também pode ser utilizada para detecção de vulnerabilidades, identificadas por meio da violação de propriedades predeterminadas (WANG; YANG; LI, 2020; NELATURU *et al.*, 2020; ALT; REITWIESSNER, 2018; WANG; ZHANG; SU, 2019). Um dos métodos formais mais utilizados é o *model checking*, no qual um sistema ou um *software* é representado como um modelo de transição de estados, e então é realizada uma pesquisa exaustiva sobre todo o espaço de estados do sistema para verificar se o modelo age de

acordo com propriedades predefinidas. Em caso de violação de uma propriedade, é fornecido um contra-exemplo com o caminho de execução realizado até a violação (CLARKE JR *et al.*, 2018; PELED, 2019).

Este trabalho é motivado pelos problemas relacionados a exploração de vulnerabilidades em CIs escritos em Solidity na blockchain Ethereum, assim como pelas limitações presentes nas abordagens existentes para mitigação destes problemas. Esta pesquisa tem o propósito de explorar este tema, tendo como base a seguinte questão de pesquisa:

"Como detectar as vulnerabilidades de reentrância, delegatecall injection e contrato suicida, em CIs escritos na linguagem Solidity na fase de pré-implementação?"

1.2 Objetivos gerais e específicos

Guiado pela questão de pesquisa, este trabalho tem como objetivo propor uma estratégia para verificação formal para aprimoramento de segurança aplicada na fase de pré-implementação de CIs escritos em Solidity para detecção das vulnerabilidades de reentrância, *delegatecall injection* e contrato suicida, por meio da técnica de *model checking*. Os objetivos específicos para se atingir o propósito deste trabalho são:

- Determinar o formalismo adequado para modelagem dos contratos e para representação das vulnerabilidades;
- Implementar o método de verificação;
- Definir as estratégias para validação da proposta.

1.3 Organização do trabalho

Este documento foi organizado da seguinte forma. No Capítulo 2 é apresentado o referencial teórico e os conceitos empregados neste trabalho. O Capítulo 3 expõe como a pesquisa foi conduzida e quais foram os métodos aplicados para levantar os trabalhos e responder às questões de pesquisa. O Capítulo 4 retrata os detalhes do método para verificação de CIs proposto, os trabalhos relacionados com a proposta desta pesquisa, e, por fim, o plano de trabalho elaborado.

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos relacionados com a tecnologia blockchain e a forma como cada um deles contribui para a manutenção de suas propriedades. Apesar de existirem diversas variações entre as blockchains, este capítulo tem como foco as plataformas Bitcoin e Ethereum, a primeira por ter sido a pioneira da tecnologia blockchain e a mais conhecida até os dias atuais, e a última por estar diretamente relacionada com este trabalho. Além de expor os aspectos estruturais e funcionais da Ethereum, este capítulo também descreve sobre CIs, vulnerabilidades existentes, e estratégias para mitigação dessas vulnerabilidades, fornecendo um conjunto de conceitos e informações necessárias para o entendimento da proposta deste trabalho.

Na Seção 2.1 são introduzidos os fundamentos da tecnologia blockchain. Na Seção 2.2 são abordados os conceitos e as particularidades inerentes à blockchain Ethereum. Na Seção 2.3 é discutido sobre o ciclo de execução dos CIs, vulnerabilidades presentes em CIs, e ataques ocorridos que exploraram essas vulnerabilidades. Enfim, algumas estratégias para verificação e detecção de vulnerabilidades são expostas Seção 2.4.

2.1 A tecnologia Blockchain

Com a ascensão das criptomoedas, a DTL tem ganhado visibilidade nos últimos anos. As DTLs são conhecidas também como blockchain, e provêm uma arquitetura descentralizada, que não necessita de confiança em uma entidade central (e.g., o banco central) para gerenciamento das transações, isto é, evita que uma terceira parte acesse as informações dos usuários (MONRAT; SCHELéN; ANDERSSON, 2019). Assim, essa tecnologia, juntamente com a implantação dos CIs, pode potencializar soluções em diversas áreas além da financeira (SWAN, 2015).

A primeira arquitetura blockchain, apresentada por Nakamoto (2008), surgiu com a criptomoeda Bitcoin, que permite aos usuários a realização de transações financeiras de forma pseudo-anônima na internet, sem necessitar de cadastro de uma agência intermediadora. Além

da Bitcoin, outras blockchains surgiram nos últimos anos, como a Ethereum, que possibilitou a implantação de CIs, que são programas de computador executados de forma automática, imutável e descentralizada (BUTERIN, 2014). Com os CIs observou-se uma expansão de novas áreas de aplicação das DTLs (MAESA; MORI, 2020). Deste modo, esta seção tem como objetivo apresentar os conceitos sobre blockchain Bitcoin e Ethereum, e expor brevemente alguns exemplos de aplicações. Os conceitos descritos a seguir, na Seção 2.1.1, abordam questões gerais sobre a blockchain, e têm como principal referência a Bitcoin. Na Seção 2.2 são tratados elementos específicos e exemplos de aplicação da blockchain Ethereum.

2.1.1 Estrutura e funcionamento da blockchain

Em empresas, utiliza-se um livro-razão para lançamento de registros contábeis para elaboração de relatórios financeiros (MARION, 1985). Na estrutura de dados de uma blockchain, este livro-razão fica disposto por meio uma cadeia de blocos interligados (i.e., uma lista encadeada) que armazenam todo o histórico de transações, como ilustrado na Figura 1. Cada participante da rede é responsável por manter uma versão atualizada do histórico de transações e preservar sua integridade (DRESCHER, 2017). Quando nova transação ocorre, informações como as contas envolvidas, a quantia transferida, a assinatura digital que autorizada a transação, e o horário da transação, são transmitidas entre todos os nós da rede. Os nós da rede, conhecidos como mineradores, coletam uma determinada quantidade de transações e as englobam em um componente chamado de bloco (DRESCHER, 2017).

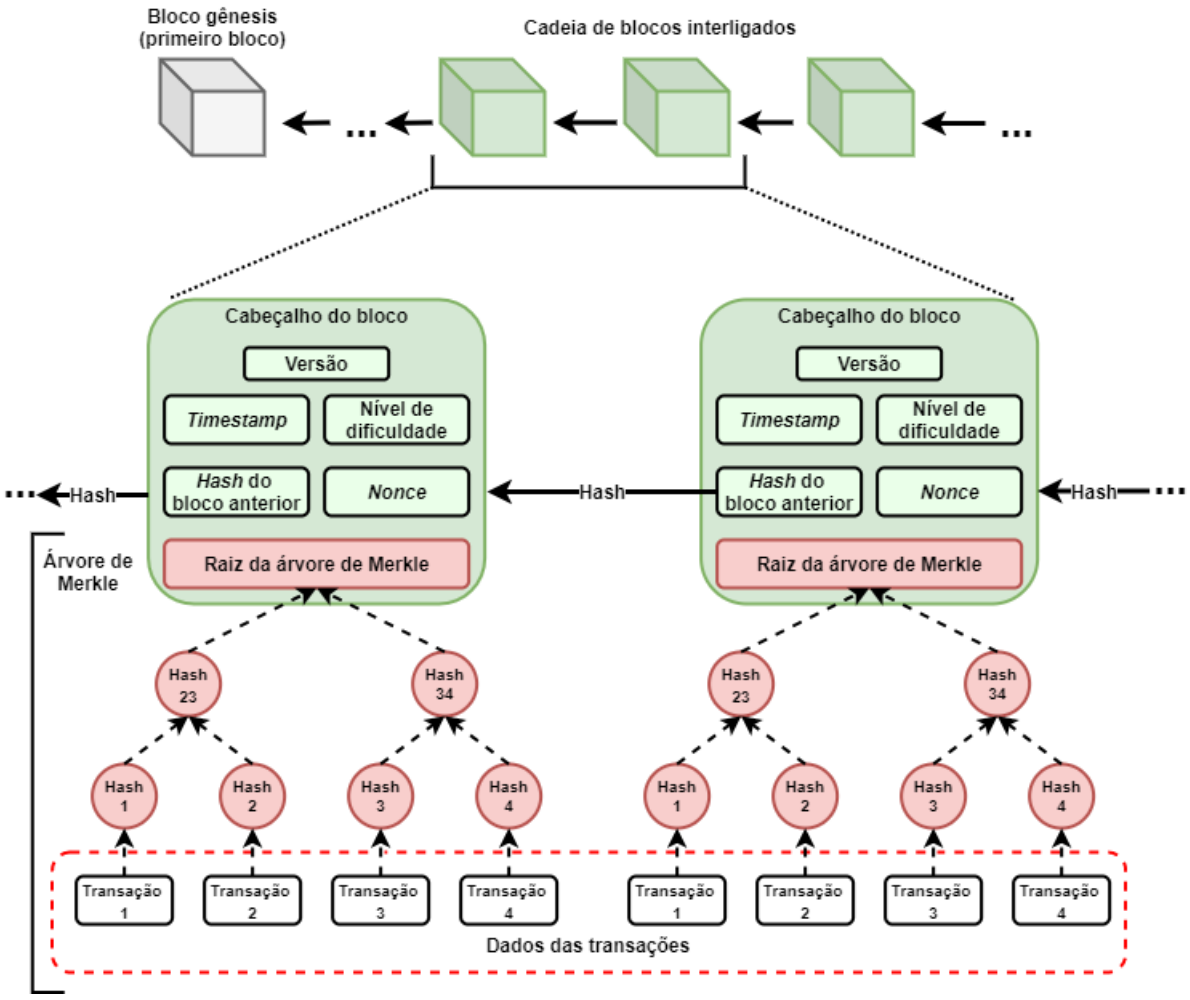
As transações englobadas em um bloco são organizadas na forma de uma árvore de Merkle. Nessa estrutura de dados do tipo árvore cada transação é alocada em um nó folha, e os demais nós armazenam referências do código *hash* gerado a partir dessas transações. Um código *hash* é gerado por meio de algum algoritmo criptográfico de geração de código *hash*, como o SHA-3 (DWORKIN, 2015) e o Keccak256 (BERTONI *et al.*, 2011). Esses algoritmos recebem e processam dados de entrada, e então geram um código formado por uma sequência de caracteres e dígitos, de forma que, para duas ou mais entradas distintas, a chance de um código *hash* igual ser gerado é extremamente baixa. Outro ponto importante é que, apenas com a posse de um código gerado, não há como se obter os dados de entrada do qual o código se originou.

Na Figura 1 é ilustrada a estrutura da blockchain como um conjunto de blocos interligados em que, a partir dos blocos, as informações das transações podem ser acessadas por meio da raiz da árvore de Merkle.

Para cada bloco é criado um cabeçalho, que passa a integrar o bloco. As informações presentes no cabeçalho podem variar de acordo com a rede blockchain. Na rede Bitcoin (NAKAMOTO, 2008), um cabeçalho é formado por cinco elementos:

- **Versão:** Número da versão do protocolo de regras de validação a ser seguido;

Figura 1 – Estrutura da blockchain



Fonte: Aguiar *et al.* (2020), Dinh *et al.* (2018).

- ***Hash* do bloco anterior:** Obtido a partir dos dados do cabeçalho do último bloco presente na blockchain no momento da construção do próximo bloco;
- **Raiz da árvore de Merkle:** Em um bloco, apenas a raiz da árvore de Merkle é armazenada.
- ***Timestamp*:** Horário atual referente ao momento em que o bloco está sendo criado. Esta informação é essencial para manter a ordenação dos blocos no histórico de transações mantido por cada nó da rede;
- **Nível de dificuldade:** Número que indica o nível de dificuldade do quebra-cabeça computacional que deve ser resolvido pelos mineradores na disputa pela criação do próximo bloco. Este item influencia diretamente no tempo e esforço computacional necessário para a criação de um bloco;
- ***nonce*:** Quando o *nonce* é incorporado ao cabeçalho do bloco, o código *hash* obtido a partir do cabeçalho deve ser iniciado por uma quantidade predefinida de zeros, que é indicada pelo nível de dificuldade.

Ao se criar um bloco, o minerador forma primeiro um bloco preliminar contendo os dados dos itens 1 ao 5. Para se obter o *nonce* é necessário realizar uma quantidade massiva de tentativas com o intuito de encontrar a sequência de caracteres e dígitos que satisfaça o nível de dificuldade predefinido. Essa tarefa é conhecidas como mineração, e gera uma disputa entre os nós da rede pela criação do próximo bloco. Nessa disputa, aqueles que possuem computadores mais robustos e com maior capacidade de processamento têm maiores chances de ganhar. A descoberta do *nonce* também é referida neste trabalho como um quebra-cabeça computacional ou quebra-cabeça de *hash* (DRESCHER, 2017; SWAN, 2015).

Assim que o *nonce* é adicionado ao bloco, este é então transmitido pela rede para que todos os nós possam acessá-lo e participar do processo de validação do bloco. Caso o bloco seja aceito no processo de validação, então cada nó adiciona o bloco válido à própria cópia da estrutura de dados blockchain e o minerador é recompensado pelo esforço empreendido (NAKAMOTO, 2008).

Como o *hash* gerado nas ramificações da árvore de Merkle depende diretamente do conteúdo das transações, qualquer alteração em uma transação invalida as referências de *hash* dos nós da ramificação da qual a transação pertence, inclusive o nó raiz. Com uma modificação no valor de *hash* da árvore de Merkle, o valor de *hash* do bloco também é alterado. Assim, alterar o valor de *hash* do bloco invalida a referência de *hash* que aponta para o cabeçalho do bloco modificado, invalidando, assim, toda a estrutura de dados (ANTONOPOULOS, 2014).

Desta forma, tentar fraudar dados de transação manipulados envolve uma série de operações custosas. Primeiro deve-se reescrever a árvore de Merkle à qual a transação manipulada pertence. Após isso, é necessário reescrever o cabeçalho do bloco a qual a raiz da árvore de Merkle reescrita pertence, o que requer a solução do quebra-cabeça de *hash* para obtenção de um novo *nonce*. Consequentemente, todos os cabeçalhos até o final da estrutura de dados da blockchain precisam ser reescritos, o que inclui encontrar o *nonce* de cada um. Este processo é propositalmente complexo e se faz necessário para manter os dados consistentes e íntegros. Isso atribui à tecnologia blockchain a propriedade de imutabilidade (ANTONOPOULOS, 2014; DRESCHER, 2017).

2.1.2 Processo de validação na blockchain

Um fator fundamental no êxito da Blockchain foi sua capacidade de garantir integridade e confiança em um ambiente de sistemas P2P puramente distribuídos, onde há um número ilimitado de nós conectados sem nenhum nível de confiança pré-estabelecidos entre estes (DRESCHER, 2017). Em uma rede de blocos com informações que podem ser produzidas por qualquer nó conectado, há o risco iminente de inserção de informações falsas e maliciosas. Para garantir a confiança de que os blocos na blockchain são legítimos, é necessário verificar a validade de um novo bloco antes deste ser inserido na rede. Para incentivar os nós a manterem a integridade das transações, são definidos mecanismos de incentivo, assim como formas de punição para

os nós que tentam inserir ou validar transações maliciosas. Em uma blockchain, as regras que regem esse protocolo são definidas por um algoritmo de consenso (SANKAR; SINDHU; SETHUMADHAVAN, 2017; ZHANG; LEE, 2020; BOURAGA, 2021).

Neste trabalho, o termo "protocolo de consenso" é usado para se referir de forma generalizada ao processo de tomada de decisão coletiva entre os nós para validação de novos blocos, um procedimento pertinente em qualquer rede blockchain. Contudo, em cada rede blockchain, esse protocolo pode ser composto por regras distintas. Cada conjunto específico de regras para estabelecimento de um protocolo de consenso é referido neste trabalho como um algoritmo de consenso, que pode apresentar diversas variações.

Um algoritmo de consenso é elaborado com o objetivo de garantir que todos os nós da rede concordem com o histórico da transações que compõem os blocos da rede, que será comum a todos, formando assim a rede blockchain (XIAO *et al.*, 2020). Desta forma, os nós são estimulados a participar do processo de validação. Além de proporcionar um ambiente participativo para criação e validação dos blocos, os algoritmos de consenso propõe formas de recompensar os nós honestos, isto é, aqueles que trabalham para manter a integridade da rede e não agem de forma maliciosa (SANKAR; SINDHU; SETHUMADHAVAN, 2017).

Cada blockchain pode utilizar uma variação de diferentes algoritmos de consenso. Dentre os principais algoritmos de consenso estão o *Proof-of-Work* (PoW), *Proof-of-Stake* (PoS) e *Practical Byzantine Fault Tolerance* (PBFT). Os próximos parágrafos descrevem os fundamentos desses protocolos de consenso.

O protocolo **PoW** tem entre seus principais mecanismos a competição entre os mineradores para resolução de um quebra-cabeça criptográfico para definir ou *nounce* do bloco. O nó que encontrar o *nounce* primeiro obtém o direito de validar o bloco, que é então criado, dissipado pela rede de nós para que todos os participantes possam verificar sua validade, e, por fim, adicionado à blockchain. O esforço computacional exigido para obtenção do *nounce* tem como consequência um alto consumo de energia. Para estimular a participação honesta dos nós no processo de mineração e compensar os custos financeiros envolvidos neste processo, algoritmos de PoW utilizados em blockchains como Bitcoin e Ethereum oferecem uma recompensa ao vencedor. Esta recompensa é feita por meio da obtenção da posse, por parte do minerador, de uma quantidade da moeda virtual utilizada como incentivo na rede, que pode posteriormente ser convertida em valor monetário (i.e., alguma moeda fiduciária) (NAKAMOTO, 2008; BUTERIN, 2014; DRESCHER, 2017). O alto esforço computacional e gasto energético despendido pelos mineradores agrega integridade aos blocos, pois não é vantajoso para um nó malicioso ter um alto gasto para resolução do *nouce* de um bloco com transações fraudadas e correr o risco iminente do bloco ser rejeitado no processo de validação. Por outro lado, um gasto computacional muito elevado pode restringir as condições de acesso dos usuários ao processo de mineração, além de aumentar o tempo para inclusão das transações, limitando questões práticas de implantação e uso de sistemas, como escalabilidade e performance (BOURAGA, 2021).

O algoritmo **PoS** foi proposto inicialmente por King e Nadal (2012) com o intuito de mitigar a dependência do alto consumo de energia e recursos computacionais do PoW (OA-POS;DWYER, 2014). No PoS, os nós que se candidatam para participar da criação dos blocos, chamados de validadores, investem uma quantia da criptomoeda vigente na blockchain. Esta quantia também é referida como valor de participação, e funciona como uma conta bloqueada com um saldo que representa o comprometimento do validador em manter a integridade da rede. Quanto maior o valor, maior a chance do validador ser selecionado para criar o próximo bloco. Enquanto no PoW a chance do minerador criar um bloco é proporcional ao seu poder computacional, no PoS a chance é proporcional ao valor de participação investido pelo validador (XIAO *et al.*, 2020; DINH *et al.*, 2018).

Baseado no trabalho de Castro e Liskov (1999), o **PBFT** é adotado na blockchain por meio de dois tipos de nós, o cliente e o servidor. O nó cliente envia um bloco aos nós servidores, e se o bloco for validado por um número suficiente de nós, então este é adicionado à blockchain. Este processo de validação das transações do PBFT consiste em cinco etapas: (i) o nó cliente envia o bloco proposto para os servidores; (ii) os servidores transmitem o bloco para outros servidores, que devem avaliar uma série de condições relacionadas à validade do bloco e chegar a um consenso sobre sua aceitação; (iii) Se o bloco for aceito, o segundo grupo de servidores enviam uma mensagem aos outros nós indicando que o bloco está pronto. Assim que esta mensagem é verificada e validada por um número suficiente de nós, estes entram em fase de "entrega"; (iv) após realizar a confirmação, cada nó transmite uma mensagem para a rede para atestar sua ação; e (v) o nó servidor que enviou o bloco recebe a resposta, seja o bloco validado ou não (BOURAGA, 2021; XIAO *et al.*, 2020; AHMED *et al.*, 2019; ZHANG; LEE, 2020).

Em dezembro de 2020 foi iniciada a primeira fase de implantação da *Ethereum* 2.0 [1], uma nova rede blockchain que utiliza o algoritmo de consenso PoS. Com isso, pretende-se aumentar a velocidade de validação das transações e integração dos blocos, expandindo a escalabilidade e otimizando a performance das aplicações.

O algoritmo PBFT é utilizado pela *Hyperledger Fabric* [2], uma plataforma blockchain privada desenvolvida pela Fundação Linux (ANDROULAKI *et al.*, 2018). Assim como o PoS, o PBFT também proporciona economia de energia para validação e integração das transações. Variações do PBFT também são empregadas nas blockchains Stellar (MAZIERES, 2015) e Ripple (SCHWARTZ; YOUNGS; BRITTO, 2014) (AHMED *et al.*, 2019; XIAO *et al.*, 2020; ZHANG; LEE, 2020).

2.1.3 Escolha do histórico de transações

O funcionamento da blockchain exige um ritmo de trabalho dos mineradores no qual, em algum momento, estes sempre estarão concentrados em alguma das seguintes tarefas: analisar

[1] <https://github.com/ethereum/eth2.0-specs>
[2] <https://www.hyperledger.org/use/fabric>

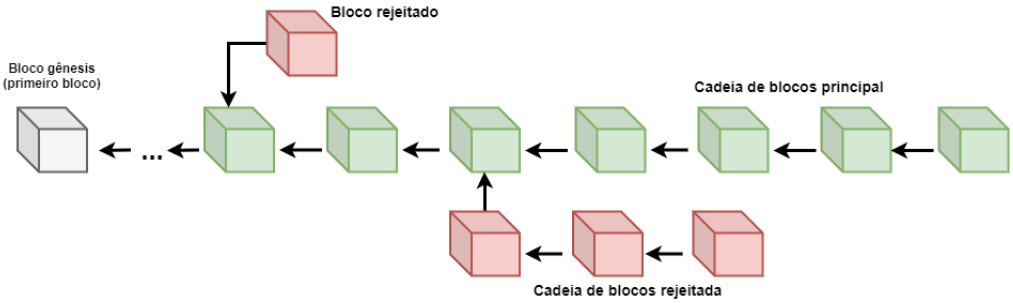
um novo bloco criado por algum nó da rede; ou se esforçar para criar o próximo bloco que, posteriormente, será analisado pelos demais nós (DRESCHER, 2017).

A capacidade de transmissão e entrega de novos blocos sofre grande influência da capacidade da entrega de mensagens de rede. Por consequência, vários nós podem terminar de construir um bloco em um pequeno intervalo de tempo. Esses blocos são transmitidos pela rede e coletados pelos nós em momentos distintos. Assim, os nós da rede não terão informações idênticas à sua disposição ao mesmo tempo (DRESCHER, 2017).

Quando um nó coleta em sua caixa de entrada mais de um bloco com o mesmo valor de referência do *hash* do bloco anterior, então uma ramificação, referida também como um *fork*, é criada, já que esses blocos possuem o mesmo bloco pai. Essas ramificações podem formar uma cadeia com diversos blocos. Dessa forma, a estrutura de dados da blockchain pode ser vista como uma árvore, porém, quando ocorre um *fork*, os nós da rede devem escolher apenas uma ramificação para compor a cadeia de blocos principal. Na Bitcoin é definido que, sempre que ocorrer um *fork*, deve-se escolher a cadeia mais longa, ou, em caso de empate, mantém-se aquela que foi recebida primeiro. (DRESCHER, 2017; SOMPOLINSKY; ZOHAR, 2015).

Em situações de decisão como essa, o protocolo da Bitcoin estabelece que a ramificação escolhida pela maioria dos nós é considerada como parte da cadeia de blocos principal, como ilustrado na Figura 2. Este procedimento é estabelecido pelo protocolo de consenso da blockchain e visa manter a integridade da blockchain por meio do consenso alcançado entre os nós participantes. Porém, deve-se considerar a premissa de que sempre haverá mais de 50% de participantes dispostos a agir de forma honesta (NAKAMOTO, 2008).

Figura 2 – Cadeia de blocos de uma blockchain



Fonte: Monrat, Schelén e Andersson (2019).

2.1.4 Criptografia e autorização de transações

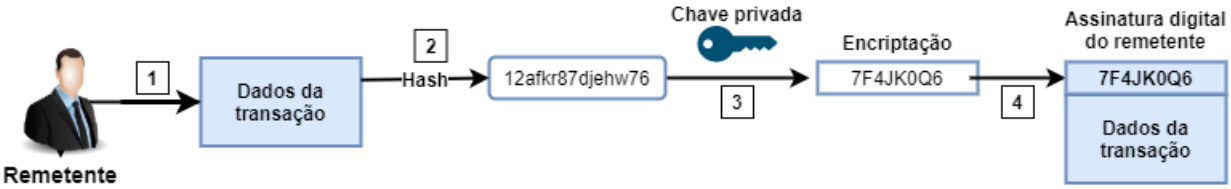
Para manter a segurança e integridade das transações, é essencial que apenas o proprietário legítimo de uma conta possa transferir o direito de propriedade ou de posse associado à sua conta (e.g., uma quantia de criptomoeda) para outra conta. Com o objetivo de garantir que somente o proprietário legítimo transfira a posse, é utilizada uma assinatura digital. Para isso, é aplicada a criptografia de curva elíptica (KOBLITZ, 1987), na qual são utilizadas técnicas

de *hash* e criptografia assimétrica por meio do par de chaves que cada nó detém, uma chave pública e outra privada. A chave privada fica disponível apenas para seu proprietário, e é utilizada para criptografar informações, transformando-as em um texto cifrado. Por se tratar de uma criptografia assimétrica, não há como se obter a informação original a partir do texto cifrado resultante. A única forma de descriptografar esse texto e obter novamente a informação original é utilizando a chave pública correspondente, que é única para cada proprietário e representa o identificador de sua conta. A chave pública é compartilhada com todos, assim, qualquer nó pode usá-la para se certificar de que a transação foi autorizada por quem cedeu a posse (DRESCHER, 2017; AHMED *et al.*, 2019).

A assinatura é utilizada em duas situações: (i) na assinatura de uma transação; (ii) e na verificação de uma transação (AHMED *et al.*, 2019). Na Figura 3 é ilustrado um exemplo em que o proprietário da conta que cede a posse realiza a assinatura da transação por meio dos passos a seguir:

1. Descreve a transação com todas a informações necessárias, exceto a assinatura;
2. Gera o valor de *hash* dos dados de transação;
3. Utiliza sua chave privada para gerar o valor de *hash* da transação a partir do valor gerado no passo 2. Esse processo é chamado de encriptação;
4. Adiciona o texto cifrado criado no item 3 à transação como sua assinatura digital.

Figura 3 – Processo de assinatura digital de uma transação



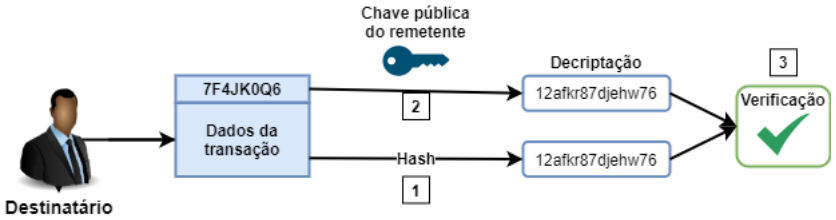
Fonte: Ahmed *et al.* (2019).

O processo de verificação de uma transação é ilustrado na Figura 4, no qual o nó verificador executa os seguintes passos:

1. Cria o valor de *hash* a partir dos dados da transação a ser verificada, com exceção da assinatura;
2. Utiliza a chave pública da conta que está cedendo a posse para descriptografar a assinatura digital da transação. Esse processo é chamado de decriptação;
3. Compara o valor do *hash* gerado no passo 1 com o valor obtido no passo 2. Se ambos forem idênticos, então indica que a transação foi autorizada pelo proprietário da chave

privada, que corresponde à chave pública que está cedendo a posse (i.e., o identificador da conta). Caso os valores não sejam idênticos, então conclui-se que o proprietário da chave privada não autorizou a transação, que é descartada.

Figura 4 – Processo de verificação da assinatura digital de uma transação



Fonte: Ahmed *et al.* (2019).

Um valor de *hash* criptográfico é único para cada transação. Analogamente, a associação entre uma chave pública e uma privada também é único. Essa característica faz com que as assinaturas digitais sejam apropriadas para servir como prova de que o proprietário da chave privada usada para criar a assinatura digital realmente concorda com o conteúdo da transação (DRESCHER, 2017).

2.2 Blockchain Ethereum

A Ethereum (BUTERIN, 2014) é uma das mais conhecidas implementações da tecnologia blockchain. Ela é definida como uma plataforma de computação distribuída composta por uma rede de computadores que operam de forma descentralizada, autônoma e democrática (WOOD, 2014). Embora também lide com geração e gerenciamento de posse de sua criptomoeda, o Ether, essa é apenas uma parte do que a plataforma é capaz de prover.

O funcionamento da Ethereum baseia-se na implantação de CIs, que são programas de computador que, uma vez implantados, executam automatica e obrigatoriamente de acordo a lógica definida em sua programação. Por meio desses programas é possível estabelecer um acordo entre duas ou mais partes envolvidas, que se comprometem a cumprir as regras estabelecidas expressas em código. Quanto compilado, um CI é convertido em um *bytecode*, uma representação de baixo nível utilizada para sua execução. Os contratos são executados de forma descentralizada por todos os participantes da rede por meio da Máquina Virtual Ethereum (MVE) (CHEN *et al.*, 2020).

Na Ethereum, as transações são disparadas por meio de mensagens, que podem conter instruções que causam a alteração na estado da blockchain (WOOD, 2014). Isso acontece, por exemplo, quando um nó executa uma função de um CI que altera o valor de algum atributo. Essas transações são coletadas pelos nós para formação dos blocos e são estruturadas em uma

trie, que é uma variação da árvore de Merkle feita especialmente para uso na Ethereum, e opera de forma semelhante na garantia da imutabilidade dos dados (WOOD, 2014).

A Ethereum foi elaborada por Buterin (2014) para ser um protocolo alternativo para criação de DApps. Na plataforma *State of The DApps* [3] há mais de 3800 DApps contabilizados, sendo que destes, pouco mais de 3 mil utilizam a Ethereum. Nas DApps, geralmente o *front-end* é implementado como uma aplicação *web*, enquanto que o *back-end* é implementado por um ou mais CIs (HEWA; YLIANTTILA; LIYANAGE, 2021). Os números envolvendo a plataforma ajudam a dimensionar o tamanho de sua popularidade. Em 2021 o valor de mercado da Ethereum superou 400 bilhões de dólares, sendo a segunda maior plataforma blockchain em valor de mercado, atrás apenas da Bitcoin, com cerca de 900 bilhões [4]. Além disso, de acordo com a plataforma Etherscan [5], há ao menos 2 milhões de CIs já executados.

Devido às tecnologias e técnicas que compõe a Ethereum, as aplicações que a utilizam dispõe de uma série de propriedades (BUTERIN, 2014; HEWA; YLIANTTILA; LIYANAGE, 2021), tais como:

- **Descentralização**: Eliminação da necessidade de confiança em uma terceira parte reguladora para execução da lógica do contrato;
- **Imutabilidade**: Uma vez executado, o código não pode ser alterado, assim como as transações resultantes da interação entre os contratos e os nós;
- **Persistência dos dados**: Uma vez inseridas na blockchain, as informações contidas em um bloco estarão sempre disponíveis;
- **Execução autônoma**: A execução de condições programadas e fluxo de eventos a serem realizados são disparados automaticamente conforme o sistema blockchain atinge um determinado estado, garantindo a autonomia da execução. O estado no qual uma ação é disparada é definido na programação do CI, em comum acordo com todas as partes envolvidas;
- **Acurácia**: Assim que o CI é executado, confia-se que as condições programadas serão cumpridas. A acurácia da execução do que foi programado é garantida por meio da transparência envolvida na execução autônoma, pois assim, vieses humanos e erros que podem acontecer em uma execução centralizada são evitados.

A seguir, na Seção 2.2.1, são abordados os tipos de contas que operam na plataforma Ethereum. Detalhes sobre as transações e troca de mensagens são tratados na Seção 2.2.2. Detalhes sobre a formação dos blocos e seu processo de validação são discutidos nas Seções 2.2.3

[3] <https://www.stateofthedapps.com/>
[4] Dados obtidos da CoinMarketCap, disponíveis em: <https://coinmarketcap.com/>.
[5] <https://etherscan.io/>

e 2.2.4, respectivamente. Alguns exemplos de aplicações baseadas em CIs que executam sobre a plataforma Ethereum são expostos na Seção 2.2.5.

2.2.1 Contas Ethereum

Diferente do modelo de representação de estados da Bitcoin, que é baseado no estado das moedas mineiradas, na Ethereum, o estado da blockchain é definido pelo estado das contas. O estado de todas as contas define o estado da blockchain, que é atualizado sempre que um novo bloco é adicionado. As contas são necessárias para que haja interação dos usuários com a blockchain por meio das transações (ETHEREUM COMMUNITY, 2018). Uma conta pode ser de dois tipos: Conta de Propriedade Externa (CPE); e conta de contrato (CC). Uma CPE é usada para armazenar os fundos do usuário em Wei, que é a menor sub-denominação de um Ether, sendo um Ether equivalente a 10^{18} Wei. As CPEs são associadas e controladas por uma chave privada, e são necessárias para que um cliente possa participar da rede. As CCs são controladas pelo código de um *bytecode* executável (CHEN *et al.*, 2020a).

O estado global da Ethereum é definido pelo estado de todas as contas. Internamente, o estado global é obtido por meio de um mapeamento entre os endereços das contas (identificadores de 20 bytes) e o estado de cada conta (WOOD, 2014). Ambas as contas possuem um estado dinâmico, definido por:

- ***nonce***: indica o número de transações iniciadas pelo proprietário da CPE correspondente, ou, no caso de uma CC, o número de contratos criados pela conta;
- ***balance***: saldo em Wei sob posse da CPE ou da CC;
- ***storageRoot***: Valor do *hash* da raiz da *trie*, a qual armazena o estado das variáveis do contrato associadas ao *bytecode* correspondente. Este atributo não é aplicável às CPEs;
- ***codeHash***: Valor do *hash* do código em *bytecode* da CC correspondente. Este atributo não é aplicável às CPEs.

As operações requisitadas em uma transação são executadas por meio da MVE, que pode seguramente verificar a identidade do remetente (i.e., uma CPE), pois, assim como na Bitcoin, as transações também são assinadas por meio da técnica de curva elíptica (ETHEREUM COMMUNITY, 2018).

2.2.2 Transações, mensagens e transição de estados

Na Ethereum, uma transação se refere a um pacote de dados criptograficamente assinado que armazena uma mensagem a ser enviada por uma CPE. Essa mensagem estabelece uma interação entre uma CPE e uma CC, ou outra CPE, e especifica alguma instrução a ser executada. Há dois tipos de transações: mensagens externas enviadas por uma CPE; e mensagens internas

enviadas por uma CC. Ambas as mensagens podem ser usadas para transferência de Ether, e criação e execução de CIs (WOOD, 2014). Em uma CPE pode-se enviar mensagens para outras CPEs ou para uma CC, basta criar uma mensagem, assinar digitalmente a transação e transmiti-lá para na rede. Sempre que uma CC recebe uma mensagem seu código é ativado. Uma mensagem enviada à uma CC tem o intuito de executar alguma função em seu código, e, se for o caso, fornecer os parâmetros necessários. Essa função pode executar alguma operação de leitura ou escrita em seu armazenamento interno (i.e., suas variáveis), ou até mesmo criar e executar outro CI. Embora um CI possa ser criado por uma CPE ou uma CC, uma CPE não pode ser criada por uma outra conta (BUTERIN, 2014; CHEN *et al.*, 2020a).

A execução de uma transação pode resultar em um certo custo computacional. Na Ethereum esse custo é calculado em *gas*, e assim, cada tipo de operação possui um determinado custo para ser executada, que varia de acordo com a quantidade de passos computacionais envolvidos, além de um custo fixo de 5 *gas* para cada *byte* dos dados da transação (WOOD, 2014). A utilização do *gas* como métrica é benéfica na medida que desvincula o custo computacional envolvido na execução das operações do custo do Wei, que possui valor monetário e está sujeito a variações de mercado. Assim, um cliente pode levar este último fator em consideração no momento de decidir o quanto está disposto a pagar em Wei por unidade *gas* utilizada na execução, que é especificado em uma transação pelo atributo *gasPrice*. Este item está diretamente relacionado com o valor pago como recompensa para o minerador que criar o bloco que contém esta transação, ou seja, quanto maior o *gasPrice*, maior é a recompensa para o minerador (WANG; ZHANG; SU, 2019).

Outro atributo fundamental em uma transação é o *gasLimit*, que define o valor máximo em *gas* que o remetente está disposto a pagar como taxa para o minerador que vencer a disputa pela criação do bloco no qual essa transação está inclusa. Este é um item essencial para evitar que estruturas de repetição consumam *gas* indefinidamente ou zerem o saldo do remetente, evitando assim maiores perdas (WOOD, 2014). Com base nisso, Chen *et al.* (2020) argumentam que a MVE pode ser considerada uma máquina quase Turing-completa. O termo "quase" refere-se ao fato de que a execução é limitada à quantidade de *gas* oferecida nas transações (CHEN *et al.*, 2020).

Na Ethereum, o estado global da blockchain é definido pelo estado das contas, seja uma CPE ou uma CC. Quando uma transação é executada, algum atributo de uma conta é alterado. Esse atributo pode ser o saldo em Ether após a realização de uma transferência, ou também o valor de uma variável de um CI, por exemplo. Desta forma, ao executar uma transação (TX), ocorre uma transição de um estado (S) para outro estado (S'), alterando assim o estado global da blockchain (BUTERIN, 2014).

2.2.3 Formação dos blocos

Conforme as transações são criadas por uma conta e transmitidas pela rede, estas são coletadas pelos mineradores e colocadas em uma *pool* de transações pendentes até serem escolhidas para constituir um bloco (WANG; ZHANG; SU, 2019). O cabeçalho de um bloco da Ethereum contém as seguintes informações:

- ***parentHash*:** Valor do *hash* do cabeçalho do último bloco pai, isto é, o antecessor do bloco atual;
- ***ommersHash*:** Valor do *hash* dos cabeçalhos dos blocos cujo antecessor são iguais ao antecessor do bloco atual. Essas blocos são chamados de *ommers*;
- ***beneficiary*:** O endereço do minerador deste bloco. Assim, o minerador é identificado e recebe as taxas de mineração coletadas;
- ***stateRoot*:** Valor do *hash* da raiz da *trie* que contém os estados das transações, após todas serem executadas e finalizadas;
- ***transactionsRoot*:** Valor do *hash* da raiz da *trie* que contém as transações que compõem o bloco;
- ***receiptsRoot*:** Valor do *hash* da raiz da *trie* que contém os recibos com as informações da execução de todas as transações listadas neste bloco;
- ***logsBloom*:** Contém um *Bloom filter*, uma estrutura de dados probabilística usada para testar se um dado elemento é membro de um conjunto. Neste caso, a estrutura é usada para armazenar informações dos *logs* de entrada dos destinatários de cada transação listada no bloco;
- ***difficult*:** Representa o nível de dificuldade para mineração do bloco. Este item é ajustado dinamicamente à cada novo bloco minerado com o objetivo de manter uma média de 15 segundos para o tempo de validação de cada bloco;
- ***number*:** Número de blocos antecessores a este na estrutura de dados blockchain, considerando o primeiro bloco, chamado de bloco gênesis, como bloco zero;
- ***gasLimit*:** Limite de gastos de *gas* por bloco;
- ***gasUsed*:** Soma de todo *gas* utilizado pelas transações deste bloco;
- ***timestamp*:** O horário do início deste bloco, definido a partir do padrão *Unix*;
- ***extraData*:** Dados extras relacionados a este bloco;
- ***mixHash*:** Valor do *hash* que, quando combinado com o *nonce*, prova que um esforço computacional suficiente foi empregado para a criação deste bloco;

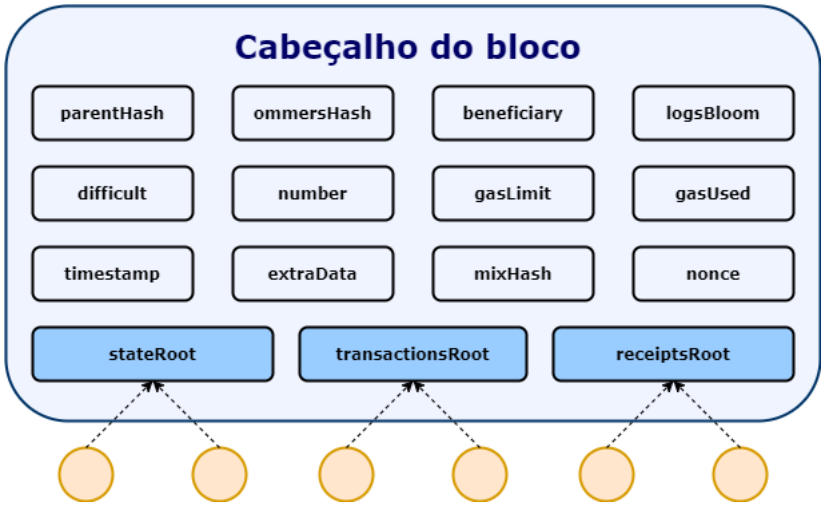
- ***nonce*:** Um valor que, quando combinado com o *mixHash* prova que um esforço computacional suficiente foi empregado para a criação deste bloco. É utilizado junto com o *mixHash* como parte do algoritmo de consenso PoW.

Ao se programar um CI, pode-se definir a emissão de *logs*, que são mensagens utilizadas para rastrear quando determinados eventos acontecem durante a execução do código. Esse evento pode ser, por exemplo, uma transferência bem sucedida entre duas contas, ou a criação de um contrato. Uma entrada de *log* contém o endereço da conta responsável pelo disparo da mensagem, tópicos que representam os eventos realizados pela transação, e demais dados associados a esses eventos [6]. O *logsBloom* é o componente do bloco em que esses *logs* são armazenados.

Durante a formação de um bloco, os mineradores tendem a selecionar as transações de forma a maximizar seus lucros. Consequentemente, as transações com o valor do *gasPrice* mais altos tendem a serem executadas primeiro. Além disso, enquanto que a *pool* pode armazenar muitas transações, o número de transações que podem ser incluídas em um bloco é restringida pelo limite máximo de gasto de *gas* por bloco (i.e., o *gasLimit* do bloco). Logo, não há como determinar com exatidão o tempo e a ordem na qual cada transação será executada (WANG; ZHANG; SU, 2019).

A estrutura do bloco da Ethereum é ilustrada na Figura 5. Observa-se que são utilizadas três estruturas em árvore para armazenamento das informações resultantes da execução das transações: *stateRoot*; *transactionsRoot*; e *receiptsRoot*. Desta forma, pode-se rastrear em detalhes o processo de execução de cada transação, o que agrega à Ethereum as propriedades de transparência, auditabilidade e persistência dos dados.

Figura 5 – Estrutura do cabeçalho de um bloco da Ethereum



Fonte: Wood (2014).

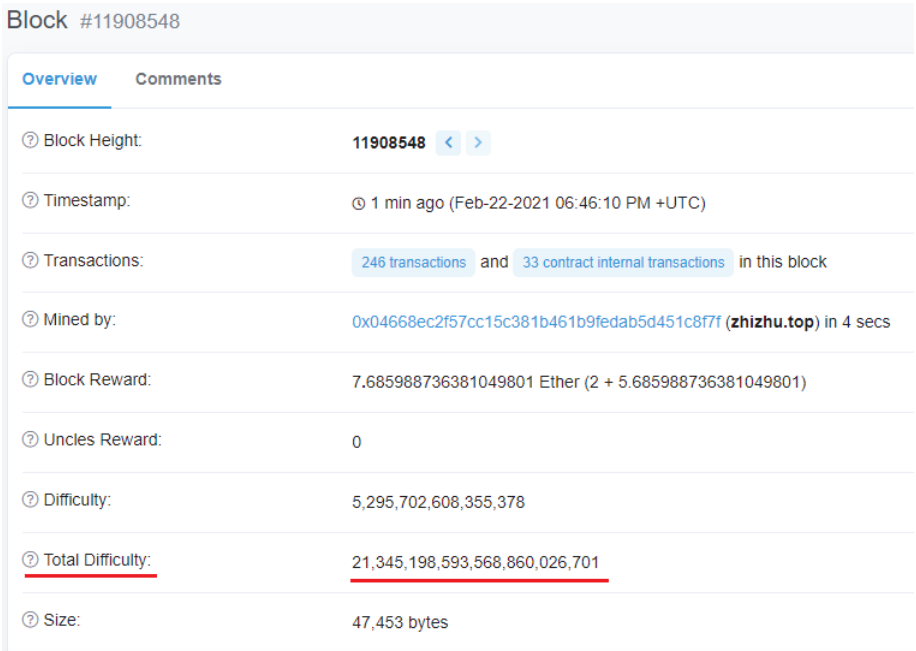
[6] Informação disponível em: <https://docs.soliditylang.org/en/v0.8.4/index.html>

2.2.4 Validação

Assim como na Bitcoin, na Ethereum os blocos também podem ser finalizados, transmitidos e recebidos em momentos distintos pelos mineradores, gerando assim um *fork* com versões distintas do histórico de transações. Na Ethereum é utiliza uma variação do protocolo GHOST (SOMPOLINSKY; ZOHAR, 2015) para selecionar como parte da rede principal a ramificação com a maior dificuldade de bloco acumulada, enquanto que as demais sub-redes continuam existindo, mas sem fazer parte da rede principal (BUTERIN, 2014).

Para cada ramificação, pode-se calcular a dificuldade acumulada, chamada também de "o caminho mais pesado", por meio do acesso às informações contidas no cabeçalho do último bloco adicionado. Como o cabeçalho contém a dificuldade de mineração do bloco, representada pelo campo *difficult*, basta somar recursivamente o valor da dificuldade de mineração de todos os blocos da rede, exceto o bloco gênesis (WOOD, 2014). Na plataforma Etherscan [7] são coletadas informações sobre todos os blocos e transações incluídos na blockchain Ethereum. Na Figura 6 pode-se observar que, entre outras informações, há o campo *Total Difficulty*, sublinhado em vermelho, seguido do valor da dificuldade acumulada na blockchain até o respectivo bloco.

Figura 6 – Dificuldade acumulada de um bloco na Ethereum



2.2.5 Aplicações

A tecnologia blockchain foi proposta inicialmente com o intuito de apoiar o desenvolvimento de criptomoedas como a Bitcoin. O êxito da Bitcoin chamou atenção tanto da academia quanto da indústria, e, posteriormente, outros tipos de criptomoedas e tecnologias baseadas na blockchain foram desenvolvidas. Como exposto por Swan (2015) e Maesa e Mori (2020),

[7] <https://etherscan.io/>

esses avanços são classificados como *Blockchain 1.0*, *2.0* e *3.0*. Blockchains aplicados ao para gerenciamento de posse de criptomoedas integram um conjunto de aplicações classificado como *Blockchain 1.0*.

Com a introdução dos CIs, impulsionados principalmente pela blockchain Ethereum, possibilitou-se a implementação dos DApps. Desta forma, viabilizou-se o uso de sistemas descentralizados projetados para automatizar aplicações financeiras baseadas em criptomoedas, como OADs e sistemas de *tokens*. Tais aplicações são baseadas na junção entre CIs e criptomoedas, e são definidas como *Blockchain 2.0* (MAESA; MORI, 2020). *Blockchain 3.0* é o estágio evolucionário no qual a tecnologia não se limita apenas à aplicações financeiras, mas também à áreas como cuidados médicos, ciências, inteligência artificial, internet das coisas, governança descentralizada, entre outras (MAESA; MORI, 2020).

No decorrer desta seção são abordadas algumas áreas de aplicação da tecnologia blockchain baseadas na utilização de CIs, e são citados alguns exemplos de sistemas relacionados com as aplicações tratadas. Como a plataforma Ethereum faz parte do foco e do escopo deste trabalho, os exemplos citados são de aplicações desenvolvidas sobre a Ethereum, apesar de existirem outras blockchains que executam aplicações semelhantes.

Sistemas de tokens

Um *token* é um ativo digital e programável gerenciado por um CI para ser utilizado em um DApp ou algum projeto específico. *Tokens* são similares às criptomoedas, porém, enquanto criptomoedas como Bitcoin e Ether possuem uma blockchain própria para sua mineração e gerenciamento, os *tokens* são criados sobre a estrutura de uma blockchain já existente (DI ANGELO; SALZER, 2020). *Tokens* são usados para representar o direito sobre algo, de forma que esse direito é representado como um artefato digital, um processo conhecido como tokenização. Quando um artefato é tokenizado, é possível fracionar seu valor para quem se interessa em obter a posse, assim como já acontece com as criptomoedas tradicionais. Desta forma, facilita-se a entrada de investidores (DI ANGELO; SALZER, 2020).

Um exemplo de aplicação de *tokens* são as *stable coins*, moedas digitais cujo valor é lastreado de acordo com alguma moeda fiduciária ou fundos de investimentos já existentes. Projetos como o Tether [8] e USD Coin [9] operam com as criptomoedas USDT e USDC, que são lastreadas pela cotação do dólar. Já o Pax Gold (CASCARILLA, 2019), opera por meio do PAXG, uma versão tokenizada do ouro físico.

A facilidade para programação e o estabelecimento de padrões para criação de tokens, como o padrão ERC-20 [10], foram fundamentais para o estabelecimento deste tipo de ativo, que

[8] Tether: Digital money for a digital age. <https://tether.to/>
[9] USDC: the world's leading digital dollar stablecoin. <https://www.circle.com/en/usdc>
[10] ERC-20 Token standard. <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>

abrange diversas aplicações. Na plataforma Etherscan [11] pode-se consultar uma lista de *tokens* criados, na qual, no momento da escrita deste trabalho, foram encontrados 362.745, considerando apenas aqueles escritos no padrão ERC-20.

Organizações Autônomas Descentralizadas

Uma OAD é uma organização desenvolvida por meio da tecnologia blockchain que pode ser gerida de forma autônoma, sem a necessidade de confiar em uma autoridade central ou estruturas hierárquicas (WANG *et al.*, 2019). Em uma OAD, todas as regras operacionais e de gerenciamento são programadas em um CI e gravadas em uma blockchain. Assim, protocolos de consenso e *tokens* são utilizados como incentivo para estimular a autonomia operacional e governamental das organizações. Por meio da implantação de uma OAD, espera-se abolir modelos de gerenciamento tradicionais baseados em hierarquia, além de reduzir os custos das organizações com comunicação, gerenciamento e colaboração (WANG *et al.*, 2019).

Em 2016, foi lançado o primeiro OAD, chamado de *The DAO* (sigla para *Decentralized Autonomous Organization*), o maior projeto de *crowdfunding* da época [12], que em pouco tempo arrecadou cerca de 150 milhões de dólares. Por meio do *The DAO*, propostas de investimento eram submetidas e os participantes compravam *tokens* que davam direito de participação na aprovação das propostas, assim como receber parte dos lucros gerados (WANG *et al.*, 2019). Após o *The DAO* outras OADs surgiram, como a *Aragon* [13] e a *Steemit* [14].

Cuidados médicos e serviços de saúde

Na área da saúde, a tecnologia blockchain pode oferecer uma infraestrutura adequada para integração de dados de prontuários médicos, e outros benefícios proporcionados pela integridade e imutabilidade dos dados. Uma proposta que utiliza CIs e a estrutura da Ethereum é o sistema MedRec (EKBLAW *et al.*, 2016), utilizado para gerenciamento de registros de prontuário eletrônicos. O MedRed permite que pacientes consultem suas informações de forma acessível e oferece uma estrutura modular que facilita a interoperabilidade com sistemas já existentes, além de gerenciar questões como autenticação, confidencialidade, contabilidade e compartilhamento de dados (EKBLAW *et al.*, 2016). Outros exemplos de aplicações da blockchain na área da saúde são relatados nos trabalhos de McGhin *et al.* (2019) e Aguiar *et al.* (2020).

2.2.6 Arquitetura em camadas

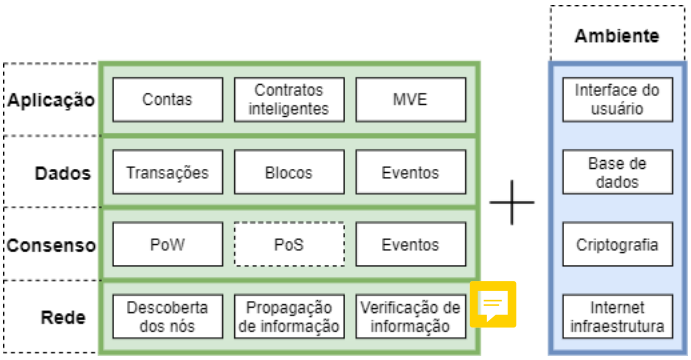
A blockchain Ethereum foi projetada sobre uma série de conceitos, protocolos e procedimentos, que dependem de recursos computacionais e tecnológicos para funcionar. Esse conjunto

[11] Etherscan. Token Traker. <https://etherscan.io/tokens>
[12] <https://bitcoinmagazine.com/business/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-
[13] Aragon: Next-level communities run on Aragon. <https://aragon.org/>
[14] Steemit. <https://steemit.com/>

de elementos compõem a arquitetura da Ethereum. Em seu trabalho, Chen *et al.* (2020a) dividem essa arquitetura em quatro camadas: aplicação; dados; consenso; e rede.

A arquitetura em camadas conta também com elementos presentes no ambiente, relacionados com recursos tecnológicos e infraestrutura, como exposto na Figura 7. Na camada de aplicação estão as contas, que podem ser CPEs ou CCs, os CIs, e a MVE, responsável pela execução do *bytecode* gerado na compilação do contrato. A camada de dados consiste nas informações geradas ao longo da execução dos contratos, como transações e *logs* de eventos, e no armazenamento destas, que são acessadas por meio dos blocos. Os mecanismos para validação dos blocos estão incluídos na camada de consenso, no qual um algoritmo de consenso e uma política de incentivos é utilizada para motivar os mineradores a agirem de forma honesta. A camada de rede é a responsável pela comunicação entre os participantes da rede, possibilitando a descoberta de novos nós e a propagação e verificação das informações, essencial para que cada nó possa manter seu histórico de transações atualizado (CHEN *et al.*, 2020a).

Figura 7 – Arquitetura da blockchain Ethereum e seu ambiente de execução



Fonte: Chen *et al.* (2020a).

Cada camada depende de componentes do ambiente de execução das aplicações, como uma interface *web* para interação dos usuários com as aplicações, uma base de dados para armazenamento dos dados da blockchain, mecanismos criptográficos para apoiar os protocolos de consenso, e o serviço de *Internet* que apoia as tarefas da camada de rede (CHEN *et al.*, 2020a).

Na Figura 7, nota-se que, na camada de consenso, o retângulo contendo o algoritmo PoS está com o contorno pontilhado. Isto deve-se ao fato da rede Ethereum 2.0, que opera com o algoritmo de consenso PoS, estar em fase inicial de implementação. Futuramente, de acordo com o planejamento do projeto, espera-se que a Ethereum seja englobada pela Ethereum 2.0.

Os contratos inteligentes, que integram a camada de aplicação, são discutidos adiante, na Seção 2.3. Além disso, também são apresentadas algumas vulnerabilidades presentes nas aplicações que executam sobre a Ethereum. Apesar de haverem diversas vulnerabilidades em todas as camadas (CHEN *et al.*, 2020a), na Seção 2.3.1 são expostas apenas vulnerabilidades e ataques relacionados com a camada de aplicação da Ethereum, pois são o foco desta pesquisa.

2.3 Contratos inteligentes

No trabalho de Szabo (1997) foi proposta pela primeira vez a ideia de um CI cujas cláusulas são escritas em programas de computador e executadas automaticamente sem a necessidade de confiar em uma terceira parte reguladora. Anos depois, por meio da tecnologia blockchain, os CIs puderam ser de fato implementados, impulsionados por plataformas como Ethereum, Hyperledger Fabric, Corda e Stellar (ZHENG *et al.*, 2020).

Para desenvolvimento de um CI, as cláusulas contratuais estabelecidas em comum acordo entre as partes envolvidas são expressas por meio de programas de computador executáveis. Esses programas são normalmente escritos em linguagens de programação de alto nível, como a linguagem Solidity [15]. Na plataforma Ethereum, independente da linguagem, os CIs são sempre convertidos em um *bytecode*, uma linguagem de baixo nível que é executada na MVE.

Contratos escritos na linguagem Solidity são similares à objetos. Cada contrato possui atributos e funções que podem ter seus controles de acesso definidos por modificadores. O controle lógico das condições estabelecidas pelas cláusulas podem ser definidos por meio de estruturas de controle, como *if*, *if-else*, *for*, etc. Um exemplo de contrato escrito na linguagem Solidity é exposto na Figura 8. Neste exemplo [16], a conta que implementa o contrato pode atribuir algum saldo para si ou para outras contas, e esses valores atribuídos podem ser transferidos para outras contas.

Figura 8 – Contrato escrito na linguagem Solidity para atribuição e transferência de saldo

```
pragma solidity ^0.5.9;

contract Coin {
    address public minter;
    mapping (address => uint) public balances;

    event Sent(address from, address to, uint amount);

    constructor() public{
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public  {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

Em seu trabalho, Zheng *et al.* (2020) descreveram a utilização dos CIs como um ciclo de vida que consiste em quatro fases: criação; implantação; execução; e conclusão. Cada fase é descrita como segue:

[15] <https://readthedocs.org/projects/solidity/>

[16] Exemplo obtido da documentação da linguagem Solidity, disponível em: <https://docs.soliditylang.org/en/v0.5.9/introduction-to-smart-contracts.html>

1. **Criação:** Essa primeira fase se inicia com a negociação entre as partes envolvidas para definição das obrigações, direitos e proibições que devem ser expressas no contrato. Em seguida, desenvolvedores e engenheiros de *software* descrevem esse acordo para alguma linguagem de programação para CIs, um processo para por etapas de projeto, implementação e validação. A criação de CIs é uma etapa interativa que pode envolver a participação de vários profissionais, como investidores, advogados e engenheiros de *software*;

2. **Implantação:** Consiste em implantar o contrato compilado na blockchain, que então não pode mais ser modificado. A implantação é feita por meio de plataformas como a Go Ethereum [17], que opera sobre a blockchain Ethereum. Nesta etapa, os envolvidos podem ter uma parcela de seus bens digitais bloqueados. Esse bem digital pode ser uma quantidade de Ether dada como garantia de uma transferência, por exemplo. Assim, as partes envolvidas são identificadas por meio de suas carteiras digitais;

3. **Execução:** Após a implantação, a execução do contrato é monitorada e avaliada. Conforme as condições estabelecidas são atingidas, operações e funções expressas no contrato são automaticamente executadas, o que gera um fluxo de transações que são executadas e validadas pelos mineradores;

4. **Conclusão:** Depois que um contrato é executado, o estado das contas envolvidas é atualizado. Logo, as transições e os dados de atualização dos estados são gravados na blockchain, as transferências entre as contas são concretizadas e os bens digitais das partes envolvidas são desbloqueados. Por fim, o ciclo de vida de um CI é concluído.

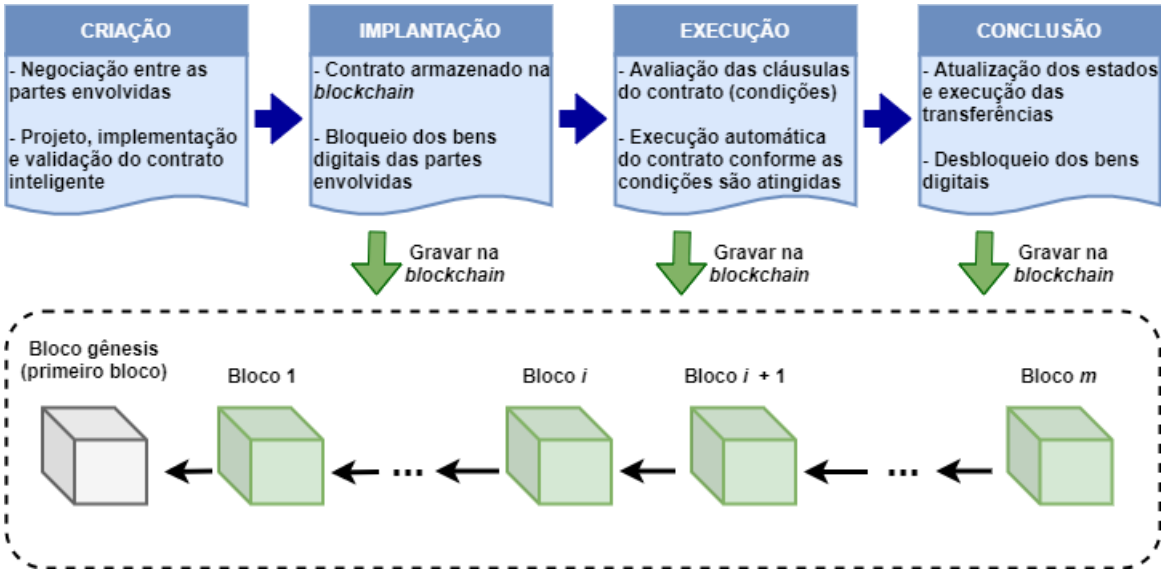
O ciclo de vida dos CIs é ilustrado na Figura 9. Nota-se que, durante as fases de implantação, execução e conclusão, uma série de transações são geradas, transmitidas, validadas e gravadas na blockchain, proporcionando a rastreabilidade e auditabilidade do contrato (ZHENG *et al.*, 2020).

Devido à imutabilidade da blockchain, um contrato implementado não pode mais ser alterado. Esta propriedade agrega integridade à tecnologia blockchain, mas também ressalta a importância da implementação de contratos livres de erros e de acordo com boas práticas, já que vulnerabilidades presentes nos contratos podem torná-los alvos de ataques.

A seguir, na Seção 2.3.1 são abordadas algumas das vulnerabilidades já encontradas em CIs e ataques que ocorreram por meio da exploração dessas vulnerabilidades.

[17] *Go Ethereum: Official Golang implementation of the Ethereum protocol*. <https://github.com/ethereum/go-ethereum>

Figura 9 – Ciclo de vida de um CI baseado em 4 fases: criação, implantação, execução e conclusão



Fonte: Zheng *et al.* (2020).

2.3.1 Vulnerabilidades e ataques

Aplicações desenvolvidas por meio de CIs, como os DApps e as OADs, costumam envolver transferências e gerenciamento de grandes quantidades de bens digitais, e isso tornou-os alvos de uma série de ataques que exploraram vulnerabilidades encontradas no código desses contratos (ATZEI; BARTOLETTI; CIMOLI, 2017; LIU; LIU, 2019; CHEN *et al.*, 2020a). O primeiro desses ataques ocorreu em 2016 sobre a OAD de *crowdfunding The DAO*, no caso conhecido como *The DAO Attack*. Na ocasião, um participante malicioso explorou uma falha no contrato e transferiu cerca de 3,6 milhões de Ether para sua conta, o equivalente a 50 milhões de dólares (SIEGEL, 2020). Este caso teve grande repercussão e chamou a atenção da academia e indústria, motivando estudos e estratégias para detecção e prevenção de vulnerabilidades em CIs (CHEN *et al.*, 2020a; LIU; LIU, 2019).

Há vários fatores que tornam a implementação de CIs propícios a erros. Segundo Atzei, Bartoletti e Cimoli (2017), parte desses erros são ocasionados pelo desalinhamento que há entre a semântica da linguagem Solidity e a intuição dos desenvolvedores. Apesar de alguns elementos em Solidity serem similares aos encontrados em outras linguagens, como funções, exceções e modificadores de acesso, estes não são implementados da mesma forma.

No decorrer desta seção são discutidas algumas vulnerabilidades conhecidas, assim como os ataques resultantes da exploração destas. Ao final, outras vulnerabilidades encontradas na literatura são listadas e brevemente descritas.

Reentrância

A reentrância foi a vulnerabilidade explorada contra a OAD *The DAO*, mencionada na Seção 2.2.5, no ataque conhecido como *The DAO Attack*. Essa vulnerabilidade ocorre quando o contrato de um receptor externo invoca novamente uma função do tipo *callback* de outro contrato antes que este termine de executar essa função. Quando um contrato vulnerável contém uma função *callback*, um contrato externo pode invocá-la sucessivas vezes até esgotar qualquer saldo contido no contrato (CHEN *et al.*, 2020a; SAYEED; MARCO-GISBERT; CAIRA, 2020).

Figura 10 – Exemplo simplificado do *The DAO Attack*

```
1  contract SimpleDAO {
2      mapping (address => uint) public credit;
3
4      function donate(address to) {
5          credit[to] += msg.value;
6      }
7      function withdraw(uint amount) {
8          if (credit[msg.sender] >= amount) {
9              msg.sender.call.value(amount)();
10             credit[msg.sender] -= amount;
11         }
12     }
13 }
```

```
1  contract Attacker {
2      SimpleDAO public dao = SimpleDAO(0x354...);
3      address owner;
4      function Attacker() {
5          owner = msg.sender;
6      }
7      function () {
8          dao.withdraw(dao.queryCredit(this));
9      }
10     function getJackpot() {
11         owner.send(this.balance);
12     }
13 }
```

Fonte: Atzei, Bartoletti e Cimoli (2017).

É apresentada na Figura 10 uma versão simplificada desenvolvida por Atzei, Bartoletti e Cimoli (2017) do contrato *The DAO* e de um contrato malicioso similar ao que executou o ataque [18] [19]. Neste exemplo, o contrato `Attacker` é capaz de explorar a vulnerabilidade e transferir todos os fundos do contrato `SimpleDAO` para a conta de seu criador. Após a implantação do contrato `Attacker`, o primeiro passo do ataque é a publicação do contrato (linha de código 2), na qual é criada uma instância do `SimpleDAO` com o endereço da CC do `Attacker`. Então, o usuário malicioso utiliza sua CPE para doar algum Ether para o contrato `Attacker` por meio de uma chamada à função `donate`, e logo após invoca a função do tipo *fallback* [20] (linha de código 7 do `Attacker`). Em seguida, a função *fallback* invoca a função `withdraw`, e o Ether é transferido para o `Attacker`. Da forma como é usada, a função `call` (linha 9 do `SimpleDAO`), que é uma função *callback*, tem como efeito uma nova invocação da função *fallback* do `Attacker`, que, maliciosamente, executa a função `withdraw` novamente. Como a execução da `withdraw` foi interrompida antes do valor `credit[msg.sender]` ser atualizado, a condição verificada na linha 8 tem êxito novamente. Consequentemente, o `SimpleDAO` realiza a transferência de Ether de novo, invoca outra vez a função *callback* e assim sucessivamente, até ocorrer um dos seguintes

[18] Esse código é desenvolvido na linguagem Solidity v0.4.2.

[19] Uma análise completa do *The DAO Attack* é fornecida em <https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>

[20] *Fallback* é uma função sem argumentos e sem retorno que é executada na invocação de um contrato quando nenhuma outra função corresponde ao identificador de função fornecido, ou quando não é fornecido nenhum dado.

eventos: (i) todo o *gas* é utilizado; ou (ii) a pilha de chamadas da MVE é totalmente preenchida; ou (iii) o saldo do `SimpleDAO` é zerado.

Esta vulnerabilidade poderia ter sido evitada se um dos seguintes procedimentos tivessem sido adotados (CONSENSYS DILIGENCE, 2021): (i) garantir que as variáveis do estado do contrato (e.g., `credit[msg.sender]`) são atualizadas antes de outro contrato ser invocado; (ii) introduzir uma trava *mutex* ao estado do contrato para assegurar que apenas o dono da trava pode alterar o estado; (iii) utilizar o método de transferência `transfer` para enviar Ether à outros contratos, pois este método possui um baixo limite de *gas* definido para sua execução.

Apesar dos danos do *The DAO Attack* terem sido revertidos, isso causou uma divisão entre os mineradores da Ethereum. A maior parte dos mineradores concordaram em reverter os danos por meio de um *hard fork*, um procedimento no qual é feita uma bifurcação na cadeia de blocos, que neste caso, foi referente ao momento anterior ao ataque. Após o *hard fork*, a cadeia de blocos principal da Ethereum continuou a partir do bloco 1920000 [21], enquanto que a outra cadeia foi continuada pelos mineradores que não concordaram com a decisão, e foi denominada como Ethereum Classic [22].

Delegatecall Injection

Para facilitar o reuso de código, a MVE dispõe do código de operação (do inglês, *opcode*) `delegatecall`, usado para inserir o *bytecode* de um contrato no *bytecode* de outro contrato, que irá executá-lo por meio de uma chamada. Quando isso ocorre, o contrato que é chamado pode alterar as variáveis de estado do contrato que o invocou. Essa característica torna este último contrato vulnerável à ação de contratos maliciosos que, quando chamados, podem causar alterações para obter benefícios e transferir *tokens* para sua conta (CHEN *et al.*, 2020a).

O primeiro ataque a explorar essa vulnerabilidade ocorreu contra a *Parity Multsignature Wallet* (ou apenas *Parity Wallet*), uma carteira multi-assinatura. Para se autorizar uma transação convencional de Ether, o remetente deve assinar a transação com sua chave privada. Na Ethereum, uma carteira multi-assinatura é um CI que requer múltiplas chaves privadas para desbloquear uma carteira e autorizar transferências. Em 2017, uma vulnerabilidade em uma chamada `delegatecall` foi explorada, e cerca de 31 milhões de dólares em Ether foi subtraído da *Parity Multsignature Wallet* (CHEN *et al.*, 2020a).

Uma versão simplificada do contrato explorado é exposta na Figura 11. A *Parity Multsignature Wallet* consiste em dois contratos. O primeiro, `WalletLibrary`, é utilizado como uma biblioteca, e implementa as principais funções da carteira. O segundo, `Wallet`, contém uma referência (i.e., `_walletLibrary`) dentro de uma função *fallback* que encaminha todas as chamadas de função não correspondidas para o contrato `WalletLibrary` por meio de uma chamada `delegatecall` (linha 7). No ataque ocorrido, o atacante assumiu a posse do contrato

[21] *Hard Fork Completed*. <https://blog.ethereum.org/2016/07/20/hard-fork-completed/>
[22] Ethereum Classic. <https://ethereumclassic.org/>

Figura 11 – Exemplo simplificado dos contratos da *Parity Multsignature Wallet*

```
1  contract Wallet {
2      address _walletLibrary = new WalletLibrary();
3      address owner;
4
5      function() payable {
6          if(msg.data.length > 0)
7              _walletLybrary.delegatecall(msg.data);
8      }
9  }
10
11 contract WalletLibrary {
12     function initWallet(address[] _owners, uint _required, uint _daylimit){
13         initDaylimit(_daylimit);
14         initMultiowned(_owners, _required);
15     }
16 }
```

Fonte: Chen *et al.* (2020a).

`Wallet` após enviar uma transação com o campo `msg.data` contendo `iniWallet()` como a função a ser chamada. Como essa função não existe no contrato `Wallet`, a função *fallback* foi invocada e a inicialização da carteira foi feita pela função `initWallet` em `WalletLibrary`, a qual substituiu os endereços originais de posse do contrato pelo endereço do atacante especificado em `msg.data`. Uma vez obtida a posse do contrato, o atacante transferiu 31 milhões de dólares em Ether para sua conta. Esta vulnerabilidade pode ser evitada se o contrato a ser compartilhado por meio de uma `delegatecall` (e.g., o contrato `WalletLibrary`) for declarado como uma biblioteca, que tem seu estado estático (*stateless*) e não como um contrato, que possui um estado dinâmico (*statefull*) (CHEN *et al.*, 2020a). Na linguagem Solidity, isto é feito usando a palavra-chave `library` (MANNING, 2018).

Contrato suicida

Um contrato pode ser "morto" pelo dono do contrato, ou alguma terceira parte confiável, por meio dos métodos `suicide` ou `selfdestruct`. Quanto isso acontece, o *bytecode* e o armazenamento do contrato é deletado. A vulnerabilidade contrato suicida, também conhecida como suicídio desprotegido, acontece quando uma autenticação inadequada permite que algum invasor tome posse do contrato e execute a função para matar o contrato (CHEN *et al.*, 2020a).

Essa vulnerabilidade foi explorada em um segundo ataque efetuado contra a *Parity Wallet*. Como resposta ao primeiro ataque contra a *Parity Wallet*, foi adicionado um modificador, `only_uninitialized`, como ilustrado no Figura 12. Desta forma, pretendia-se proteger a função `initWallet()` de forma que uma reinicialização da `Wallet` por meio da chamada `delegatecall` teria como resposta o disparo de uma exceção, e então seria rejeitada pelo modificador. Porém, o próprio contrato `WalletLibrary` foi deixado como não inicializado, e o invasor pôde passar pelo modificador `only_uninitialized` e se autodeclarar o dono do contrato. Assim que assumiu o controle da biblioteca, o invasor invocou o método `suicide`

Figura 12 – Contrato da *Parity Wallet*, corrigido após o primeiro ataque

```
contract WalletLibrary {
    modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

    function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
        initDaylimit(_daylimit);
        initMultiowned(_owners, _required);
    }
    function kill(address _to) onlymanyowners(sha3(msg.data)) external {
    suicide(_to);
    }
}
```

Fonte: Suiche (2017).

para matar o contrato. Por consequência, todas as carteiras criadas pelo contrato `Wallet` que dependem da biblioteca foram inutilizadas, o que causou o bloqueio permanente de 280 milhões de dólares em Ether associados às carteiras (CHEN *et al.*, 2020a; DESTEFANIS *et al.*, 2018).

Outras vulnerabilidades

As vulnerabilidades discutidas nesta seção são o foco desta pesquisa. Entretanto, existem diversas vulnerabilidades encontradas na literatura, e algumas delas são listadas e brevemente descritas na Tabela 1. Diante dos ataques efetuados que exploraram vulnerabilidades nos CIs e suas consequências, uma série de pesquisas foram realizadas nos últimos anos com foco na verificação de CIs e detecção de vulnerabilidades, nas quais diversas técnicas têm sido empregadas (CHEN *et al.*, 2020a; ALMAKHOUR *et al.*, 2020; LIU; LIU, 2019; SINGH *et al.*, 2020). Algumas dessas técnicas são discutidas a seguir na Seção 2.4.

2.4 Verificação e validação

Grande parte das vulnerabilidades encontradas em CIs escritos em Solidity poderiam ter sido evitadas com a ajuda de análise formal e verificação desses contratos antes de serem implantados na blockchain (SINGH *et al.*, 2020; CHEN *et al.*, 2020a). Porém, as linguagens de domínio específico encontradas no estado da arte, como Solidity, não foram desenvolvidas com o intuito de serem verificadas formalmente, fazendo disso uma tarefa desafiadora. Além disso, Solidity não é uma linguagem perfeita para escrever CIs, já que é vulnerável à certos riscos [23], e suas características, que são diretamente relacionadas com a execução de CIs na Ethereum, muitas vezes não são compreendidas pelos desenvolvedores (SINGH *et al.*, 2020; ATZEI; BARTOLETTI; CIMOLI, 2017). Consequentemente, mesmo desenvolvedores experientes estão sujeitos a deixar vulnerabilidades de segurança e *bugs* em seus códigos. Devido a isso, organizações recorrem

[23] Solidity. *Security Considerations*.<https://docs.soliditylang.org/en/v0.8.4/security-considerations.html>

Tabela 1 – Tipos de vulnerabilidades em contratos inteligentes

Vulnerabilidade	**Descrição**
Ataque de profundidade da pilha de chamadas	Acontece quando é excedido o limite de chamadas ao método de um contrato.
Ataque DoS com operações ilimitadas	Essa vulnerabilidade é resultante de programação imprópria com operações ilimitadas em um contrato, que podem entrar em *loop* indefinidamente.
Autenticação com `tx.origin`	Ocorre quando um contrato utiliza `tx.origin` para autenticação do dono ou administrador do contrato ao invés de usar msg.sender. Assim, autenticação pode ser comprometida por um ataque *phishing*.
Bloqueio de Ether	Os fundos do contrato ou o saldo em ether são travados indefinidamente, isto é, o CI pode receber Ether mas não pode enviar. Essa vulnerabilidade é conhecida também como contrato guloso.
Consumo de *gas* ineficiente	Consumo de *gas* desnecessário na execução do código do contrato.
Contrato pródigo	O contrato pode liberar fundos ou saldo em Ether para usuários arbitrários.
Contrato *honeypot*	Decorre da inserção intencional de vulnerabilidades em um CI para atrair usuários maliciosos, que depositam Ether com a intenção de explorar o contrato, mas não conseguem recuperar a quantia depositada.
Controle de acesso vulnerável	Quando uma função que lida com informações sensíveis (e.g., um método construtor) permite o acesso de usuários arbitrários.
Dependência de informação do bloco	Também conhecido como dependência de variável previsível, ocorre quando um contrato utiliza informação de um bloco para disparar ações, ou como semente para geração de números aleatórios. Um minerador malicioso pode se aproveitar disso para obter benefícios.
Dependência de ordem da transação	Ordem das transações inconsistente em relação ao momento da invocação.
Dependência de *timestamp*	Ocorre quando um contrato que utiliza o *timestamp* de um bloco como parte da condição para acionar uma operação crítica (e.g. envio de Ether) é explorado por um minerador malicioso.
Desordem de exceções	Se dois ou mais contratos interagem entre si, então a função de um contrato pode depender da execução de uma função em outro contrato, e assim sucessivamente, criando uma corrente de chamadas de funções aninhadas. Caso a invocação de alguma função for feita por meio das chamadas `address.call()`, `address.delegatecall()`, ou `address send()` e ocorrer um erro nesta função, as transações não serão revertidas e os outros contratos não estarão cientes do erro, o que pode resultar em alterações inesperadas no estado dos contratos envolvidos.
Divisão por zero	É um erro aritmético que ocorre quando há uma divisão por zero ou módulo zero.
Endereço curto	A MVE não verifica a validade de endereços. Desta forma, se o tamanho de uma variável `address` na chamada de uma função for menor do que o esperado, então a quantidade de *bytes* no argumento da função também será menor do que o esperado, e os *bytes* restantes serão preenchidos com zeros hexadecimais, o que pode alterar o valor dos parâmetros.
Exceções não tratadas	Ocorre quando uma exceção é disparada em um CI por meio da chamada de outro contrato e não é tratada devidamente por aquele que fez a chamada.
Chamada externa não verificada	O valor de retorno de uma chamada à outro contrato não é verificado.
Integer *overflow* e *underflow*	Um *overflow*/*underflow* pode ocorrer quando são executadas operações de adição, subtração, ou armazenamento da entradas do usuário sobre variáveis inteiras com limitações de valor.
Gasto de *gas* descontrolado	Ocasiona um consumo de *gas* desnecessário na execução do código do contrato.

Fonte: Fu *et al.* (2019), Chen *et al.* (2020a), Lu *et al.* (2019), Huang *et al.* (2021), Ashraf *et al.* (2020), Jiang, Liu e Chan (2018), Torres, Schütte e State (2018).

à serviços de auditoria de segurança, como os oferecidos pela OpenZeppeling [24], Solidified [25] e SmartDec [26]. Segundo Dika e Nowostawski (2018), auditorias são a forma mais efetiva de garantir a segurança dos CIs antes de sua implantação. Entretanto, esses serviços podem ser muito custoso para pequenas organizações e desenvolvedores autônomos, que, por sua vez, têm que recorrer à frameworks e ferramentas de verificação de CIs (SINGH *et al.*, 2020).

Essas ferramentas e frameworks realizam dois tipos de verificação: (i) proativa; e (ii) reativa. A verificação proativa é aplicada sobre os CIs antes da sua implantação, enquanto que a

[24] <https://openzeppelin.com/>
[25] <https://solidified.io/>
[26] <https://smartcontracts.smartdec.net/>

reativa tem a função de reagir à potenciais explorações de vulnerabilidades sobre os CIs durante a fase de execução destes, e é referida também como verificação em tempo de execução (CHEN *et al.*, 2020a). Os métodos de verificação são aplicados por meio de diversas abordagens, tais como análise de código, métodos formais, *fuzzing*, e com o uso de técnicas de IA, que são discutidas adiante nas Seções 2.4.1, 2.4.2, 2.4.3 e 2.4.4, respectivamente. Na Seção 2.4.5 são levantados alguns aspectos referentes à verificação em tempo de execução. Enfim, como estão diretamente relacionadas com a proposta desta pesquisa, as lógicas temporais são abordadas na Seção 2.4.6.

2.4.1 Análise de código

A análise de código é uma estratégia de verificação que geralmente é executada de forma automatizada, e é utilizada como um detector de erros no processo de desenvolvimento de *software*. Ferramentas para verificação de *software* automatizam a detecção de certos tipos de anomalias, como as discutidas anteriormente, que, no contextos dos CIs, podem levar à exploração de vulnerabilidades. Para isso, diversos aspectos podem ser analisados, como fluxo de controle, fluxo de dados, interface, fluxo de informações, e análise de caminhos de execução (LUU *et al.*, 2016; ZHENG *et al.*, 2006). Os métodos de análise podem executar a verificação de forma estática, na qual o código é examinado sem a necessidade de executá-lo, ou de forma dinâmica, em que é preciso realizar ou simular a execução total ou parcial para inferência de padrões de execução que correspondem à algum comportamento indesejado. Há também métodos híbridos, que utilizam tanto a análise estática quanto a dinâmica. Em ambas as formas, normalmente obtém-se uma representação intermediária estruturada (RIE) do código fonte ou do *bytecode* dos CIs para realizar a verificação (LUU *et al.*, 2016; TSANKOV *et al.*, 2018; TIKHOMIROV *et al.*, 2018).

Ferramentas como SafeVM (ALBERT *et al.*, 2019), Securify (TSANKOV *et al.*, 2018) e GASTAP (ALBERT *et al.*, 2021) utilizam o *bytecode* dos CIs para obtenção de um grafo de fluxo de controle (GFC), e, a partir desta RIE, executam a análise estática em busca de padrões que representam vulnerabilidades ou a violação de propriedades de segurança. O GFC é uma das RIEs mais utilizadas para análise de CIs, inclusive para análise dinâmica, como nas ferramentas Oyente (LUU *et al.*, 2016) e sCompile (CHANG *et al.*, 2019). Já nas ferramentas NeuCheck (LU *et al.*, 2019) e SmartCheck (TIKHOMIROV *et al.*, 2018) é obtida uma árvore sintática em XML (sigla para *Extensible Markup Language*).

Entre os métodos empregados para análise dinâmica, um dos mais populares é a execução simbólica, utilizada em ferramentas como Oyente (LUU *et al.*, 2016), Osiris (TORRES; SCHÜTTE; STATE, 2018), GasChecker (CHEN *et al.*, 2020b) e Solc-Verify (HAJDU; JOVANOVIĆ, 2019). A execução simbólica é uma técnica para análise de programas que substitui o valor das variáveis do programa por expressões simbólicas com o intuito de descobrir todos os caminhos de execução viáveis por meio da construção de uma RIE, como um CGF (KING, 1976). Há condições para a execução de um caminho simbólico da RIE, e para que o caminho

seja viável a condição de seu caminho deve ser satisfatória. Desta forma, a RIE resultante é checada por meio de solucionadores SMT (do inglês, Satisfiability Modulo Theories solvers (SMT-solver)) para identificação e detecção de vulnerabilidades (ALMAKHOUR *et al.*, 2020).

2.4.2 Métodos formais

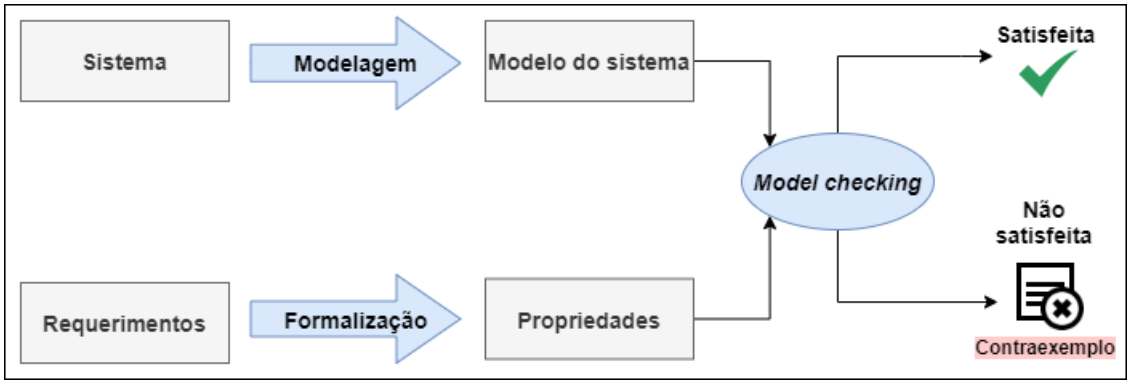
Métodos formais são uma coleção de técnicas baseadas na matemática e em lógicas para aprimorar a confiança em sistemas. Com o uso dessas técnicas, um sistema é modelado formalmente por meio de representações matemáticas, lógica de processos ou modelos baseados em estados. Dado o modelo, são definidas e descritas formalmente especificações que representem propriedades do sistema, como requisitos operacionais de segurança e vivacidade, por exemplo. O processo de verificação, que pode ser automático ou não, é então efetuado, e o modelo é verificado a partir das especificações fornecidas (PELED, 2019). Métodos formais exigem conhecimento dos formalismos utilizados, e podem ser custoso em relação ao tempo e recursos necessários, sendo, assim, aplicados em sistemas críticos, em que falhas podem levar à graves prejuízos, como é o caso dos CIs. Ao longo desta seção, são abordados três dos métodos formais mais utilizados para verificação de CIs: *model checking*; demonstração de teoremas; e verificação dedutiva.

Model checking

Model checking é uma técnica usada para verificação de sistemas de transição de estados que consiste em três etapas: (i) modelagem; (ii) especificação; e (iii) verificação. Com o *model checking*, dado um modelo de estados finito de um sistema, é checado se o modelo satisfaz determinadas propriedades. Para isso, os requerimentos do sistema são formalizados e representadas como propriedades do sistema. As propriedades são descritas por meio de algum formalismo lógico, como por exemplo, as lógicas temporais linear e ramificada, abordadas em detalhes na Seção 2.4.6. Uma vez obtidos o modelo do sistema e as propriedades, é iniciada a verificação, isto é, o processo de *model checking*, geralmente executado automaticamente por alguma ferramenta, chamada de *model checker*, que interpreta o modelo do sistema como um grafo de estados. Por meio de procedimentos que realizam uma pesquisa exaustiva sobre todo o espaço de estados do sistema, é verificado se, de acordo com o modelo, o sistema age da forma esperada e se o modelo é satisfeito pela propriedade. Quando uma propriedade não é aceita, um contra-exemplo é fornecido, como ilustrado na Figura 13 (CLARKE JR *et al.*, 2018). Desta forma, todas as possibilidades de interação são verificadas e aquelas que estiverem em inconformidade com as especificações do sistema são detectadas antes da sua implementação (PELED, 2019).

De forma geral, diversas linguagens formais e *model checkers* podem ser utilizadas para modelagem e verificação de CIs. No trabalho de Nehai, Piriou e Daumas (2018), um CI é modelado por meio da linguagem de entrada do *model checker* NuSMV, o qual é utilizado para verificação de propriedades funcionais especificadas em CTL. Na estudo desenvolvido por Wang,

Figura 13 – Procedimento do *model checking*



Fonte: Almakhour *et al.* (2020).

Yang e Li (2020), o código Solidity é traduzido para a linguagem formal MSVL, e o *model checking* é aplicado para detecção da vulnerabilidade de reentrância. Também há exemplos de uso de Redes de Petri (LIU; LIU, 2019) para modelagem, e de outros *model checkers* como SPIN (BAI *et al.*, 2018; OSTERLAND; ROSE, 2020) e Spacer (MARESCOTTI *et al.*, 2020).

Um dos maiores desafios enfrentados no desenvolvimento de algoritmos de *model checking* é a explosão de estados, que ocorre quando o modelo obtido a partir do código fornecido permite a exploração de uma quantidade excessivamente grande de estados, o que pode inviabilizar a aplicação desta técnica. Em razão disso, foram criados métodos alternativos de *model checking* para tentar evitar a construção completa do grafo de estados, como o *model checking* simbólico e o *model checking* limitado (PELED, 2019). Há também uma variação denominada *model checking* estatístico, utilizada no trabalho de Abdellatif e Brousmiche (2018), no qual o mecanismo de verificação calcula a probabilidade de sucesso em cada um dos cenários de ataque especificados.

Demonstração de teoremas

Na demonstração de teoremas, a modelagem do sistema e a especificação das propriedades é feita por meio de formalismos matemáticos (ALMAKHOUR *et al.*, 2020). Este método formal é utilizado para providenciar provas a partir de alguma lógica simbólica utilizando inferência dedutiva. Cada passo da demonstração introduz um axioma ou uma premissa e fornece uma afirmação, a qual consiste em uma consequência natural dos resultados previamente estabelecidos utilizando regras de inferência (SINGH *et al.*, 2020). Os formalismos comumente utilizados para modelagem e especificação de propriedades são: lógica proposicional; lógica temporal (LTL, CTL); lógica de ordem superior e lógica de primeira ordem (HARRISON, 2008; SUN; YU, 2020; YANG; LEI, 2019; LI *et al.*, 2019).

Verificação dedutiva

A verificação dedutiva consiste em gerar um conjunto de provas matemáticas a partir do sistemas e suas especificações. Se essas provas matemáticas se mostrarem verdadeiras, então isso implica na conformidade do sistema com sua especificação. Essa abordagem geralmente exige que seja fornecida uma sequência de teoremas que representam propriedades do sistema modelado e outras especificações como invariantes, pré-condições e pós-condições relacionadas com estas propriedades. O processo de verificação pode exigir trabalho manual, mas geralmente é executado com o auxílio de provadores de teoremas e SMT-solvers (AHRENDT *et al.*, 2016; PARK *et al.*, 2018; BEILLAHI *et al.*, 2020).

2.4.3 Fuzzing

Teste *Fuzz*, ou *fuzzing* é uma técnica para teste de software em que são fornecidos dados de entrada aleatórios chamados de *FUZZ* (ALMAKHOUR *et al.*, 2020). Uma ferramenta de teste *fuzz*, referida também como *fuzzer*, gera entradas de teste para um programa alvo de forma iterativa e aleatória (KLEES *et al.*, 2018).

Os *fuzzers* geralmente seguem o seguinte procedimento (KLEES *et al.*, 2018): (i) O processo é iniciado com a seleção de um conjunto de sementes de entrada com as quais o programa é testado; (ii) o *fuzzer* cria repetitivamente mutações dessas entradas e avalia o programa testado; (iii) se o resultado obtido for considerado satisfatório, então o *fuzzer* mantém a mutação de entrada para uso futuro e armazena o que foi observado; (iv) o *fuzzer* é encerrado em duas situações, quando um determinado objetivo é alcançado (e.g., um determinado erro é encontrado, ou quando uma entrada causa o travamento do programa testado), ou quando o limite de tempo é atingido.

2.4.4 Inteligência artificial

Recentemente, técnicas de IA têm sido utilizadas para verificação e detecção de vulnerabilidades em CIs, principalmente as técnicas de *machine learning* (XING *et al.*, 2020; SUN; GU, 2021; WANG *et al.*, 2020) e *deep learning* (GAO *et al.*, 2020; QIAN *et al.*, 2020). De forma geral, os métodos baseados em IA consistem na transformação de CIs com vulnerabilidades conhecidas para obtenção de vetores ou matrizes que representam padrões encontrados no código ou que indicam a presença de vulnerabilidades. Esses modelos de contratos vulneráveis são utilizados para treinar algoritmos de detecção, que verificam em outros contratos a presença de padrões similares aos encontrados nos contratos vulneráveis, indicando, assim, a presença de erros e vulnerabilidades. Na detecção de vulnerabilidades, são utilizadas técnicas como redes neurais, redes neurais convolucionais, floresta aleatória, redes de propagação de mensagens temporais, entre outras (XING *et al.*, 2020; ZHUANG *et al.*, 2020; SUN; GU, 2021; GAO *et al.*, 2020). Em alguns casos, os modelos de contratos vulneráveis são adquiridos por meio da utilização

de ferramentas baseadas em análise estática e execução simbólica, que são empregadas para rotulação das vulnerabilidades encontradas em cada contrato (WANG *et al.*, 2020; MOMENI; WANG; SAMAVI, 2019).

2.4.5 Verificação em tempo de execução

Métodos para verificação em tempo de execução consistem em instrumentalizar o código Solidity para que este tenha um sistema de monitoramento embutido. Desta forma, é possível monitorar os caminhos de execução e reagir à atividades suspeitas que podem levar à violação de propriedades predefinidas (WANG *et al.*, 2019; AZZOPARDI; ELLUL; PACE, 2018; LI; CHOI; LONG, 2020). Na ferramenta ContracLarva (AZZOPARDI; ELLUL; PACE, 2018), as partes envolvidas no projeto do contrato depositam uma quantia em Ether que fica bloqueada até o término da execução do contrato, e, em caso de detecção de violações de alguma propriedade, as partes prejudicadas são indenizadas, enquanto que aquelas que violaram alguma propriedade não recuperam o valor depositado. No estudo de Wang *et al.* (2019), a ferramenta ContractGuard instrumentaliza o código fonte com um sistema de intrusão baseado em anomalia. Entretanto, a inserção de um mecanismo de monitoramento nos contratos tem como consequência o aumento do custo de execução, isto é, o gasto excessivo de *gas*, o que representa a maior limitação encontrada em tais abordagens. (WANG *et al.*, 2019; AZZOPARDI; ELLUL; PACE, 2018).

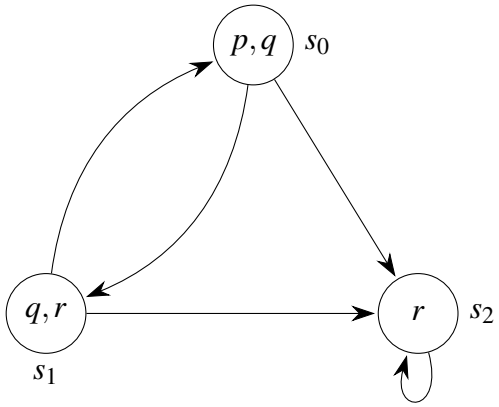
2.4.6 Lógica temporal

A lógica temporal é um formalismo usado para descrever sequências de transações entre estados em sistemas reativos (CLARKE JR *et al.*, 2018). Portanto, uma fórmula lógica temporal geralmente é avaliada sobre um sistema de transição que modela uma especificação de um sistema (CLARKE JR *et al.*, 2018). Na lógica temporal, o modelo é dado pela tupla $\mathscr{M} = (S, \rightarrow, L)$, em que S é o conjunto de estados, $\rightarrow$ (ou R) $\in S \times S$ é a relação de transição e L: $S \rightarrow 2^{AP}$ denota a função de rotulação dos estados com as proposições atômicas AP verdadeiras no estado em que se encontram. Um exemplo de sistema de transição é apresentado na Figura 14, definido por:

- $AP = \{p, q, r\}$
- $S = \{s_0, s_1, s_2\}$
- $\rightarrow = \{(s_0, s_1), (s_1, s_0), (s_0, s_2), (s_1, s_2), (s_2, s_2)\}$
- $L(s_0) = \{p, q\}, L(s_1) = \{q, r\}, L(s_2) = \{r\}$

Uma fórmula lógica temporal pode ser verdadeira ou falsa dependendo do modelo no qual é interpretada. O conjunto de estados representam a evolução do sistema ao longo do tempo e as fórmulas são avaliadas em cada estado do sistema. Portanto, a noção estática de

Figura 14 – Exemplo de um sistema de transição.



Fonte: Huth e Ryan (2004).

verdade é substituída pela noção dinâmica de evolução dos estados do sistema ao longo do tempo (CLARKE JR *et al.*, 2018). A lógica temporal pode ser dos tipos linear ou ramificada, as quais são discutidas a seguir.

Lógica temporal linear

Na lógica linear (do ingês, *linear tree logic* (LTL)), o tempo é tratado como se cada momento tivesse apenas um único futuro possível, logo, as fórmulas são interpretadas como sequências lineares, ou caminhos. A LTL possui conectivos que se referem ao futuro, e suas fórmulas são interpretadas sobre uma sequência de estados (i.e., o caminho). Um caminho é uma sequência finita não vazia denotada por $\pi = \langle \pi_0, \pi_1, \ldots, \pi_{n-1} \rangle$, em que os estados $\pi_0, \pi_1, \ldots, \pi_{n-1} \in S$ e $(\pi_i, \pi_{i-1}) \in R$ para todo $0 \leq i < n-1$. O tamanho de um caminho é denotado por $|\pi| = \infty$. Um caminho infinito é denotado por $\pi = \langle \pi_0, \pi_1, \pi_3, \ldots \rangle$ com tamanho $|\pi| = \infty$. Todo caminho que não pode ser estendido é considerado maximal, logo, qualquer caminho infinito é maximal (MULLER-OLM; SCHMIDT; STEFFEN, 1999). A sintaxe de uma fórmula LTL é composta de proposições atômicas e gerada conforme segue:

$$\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi U \varphi' \tag{2.1}$$

Assim como na lógica proposicional, os valores lógicos avaliados como verdadeiro e falso são representados pelos símbolos $\top$ e $\bot$, respectivamente, além dos operadores lógicos $\neg$, $\wedge$, $\vee$ e $\rightarrow$, que também têm o mesmo significado. Já o símbolo p representa um identificador das proposições, e os operadores X, F, G e U, são usados para especificar situações temporais de uma fórmula. A expressão $X\varphi$ (após φ) indica que a proposição φ é verdadeira no próximo estado. Já a expressão $F\varphi$ (afinal, φ) configura que em algum estado futuro da computação, φ é verdadeira, enquanto que $G\varphi$ (sempre φ) indica que φ é verdadeira em todos os estados do caminho de computação. Enfim, a expressão $\varphi U \varphi'$ (φ até que φ') significa que a proposição φ deve ser

verdadeira em todos os estados até que φ seja verdadeira (MURA, 2016; MULLER-OLM; SCHMIDT; STEFFEN, 1999).

Uma fórmula α em LTL é satisfeita se existe um modelo $\mathscr{M}$ e um caminho π tal que $\mathscr{M}, \pi \models \varphi$, ou seja, um modelo no qual o caminho π satisfaz a fórmula φ. Um caminho é formado por uma sequência de estados s e suas posições são indexadas partindo de π_1 até π_n, em que n determina a quantidade de posições de π. Sendo π um dado caminho de $\mathscr{M}$, as relações de satisfação $\models$ e $\not\models$ indicam se uma fórmula LTL é satisfeita ou não, respectivamente, por π, como exposto na Figura 15.

Figura 15 – Semântica da LTL.

$$\pi \models \top \tag{2.2}$$
$$\pi \not\models \bot \tag{2.3}$$
$$\pi \models p \iff p \in L(\pi_1) \tag{2.4}$$
$$\pi \models \neg\varphi \iff \pi \not\models \varphi \tag{2.5}$$
$$\pi \models \varphi \wedge \varphi' \iff \pi \models \varphi \text{ e } \pi \models \varphi' \tag{2.6}$$
$$\pi \models \varphi \vee \varphi' \iff \pi \models \varphi \text{ ou } \pi \models \varphi' \tag{2.7}$$
$$\pi \models \varphi \rightarrow \varphi' \iff \pi \models \varphi \text{ sempre que } \pi \models \varphi' \tag{2.8}$$
$$\pi \models X\varphi \iff \pi_2 \models \varphi \tag{2.9}$$
$$\pi \models G\varphi \iff \forall i \geq 1 \mid \pi_i \models \varphi \tag{2.10}$$
$$\pi \models F\varphi \iff \exists i \geq 1 \mid \pi_i \models \varphi \tag{2.11}$$
$$\pi \models \varphi U \varphi' \iff \exists i \geq 1 \mid \pi_i \models \varphi' \text{ e } \forall j = 1 \wedge j < i \mid \pi_j \models \varphi \tag{2.12}$$

Fonte: Huth e Ryan (2004).

Nas definições 2.2 e 2.3 da Figura 15 são indicadas as proposições verdadeiras e falsas, respectivamente. Na definição 2.4 é dito que a proposição p só é válida se estiver no primeiro estado do caminho. Nas proposições 2.5 à 2.8 são definidos os conectivos proposicionais da lógica proposicional. Em 2.9, o operador X denota que a fórmula φ é satisfeita no próximo estado, no caso, a segunda posição do caminho. As definições 2.10 e 2.11 representam os operadores G e F que apontam, respectivamente, que a fórmula φ é verdadeira em todas ou em pelo menos uma posição de π. Por fim, a definição 2.12 representa o operador U, expressando que, para toda posição antecessora à π_i, a fórmula φ é verdadeira, caso contrário, a proposição φ' é verdadeira.

Diversas propriedades podem ser verificadas com fórmulas LTL, tais como as de segurança, de vivacidade, e de progresso (HUTH; RYAN, 2004; MURA, 2016). Huth e Ryan (2004) descrevem algumas propriedades que podem ser representadas em sistemas reais, considerando um sistema especificado com as proposições *ocupado*, *requisitado*, *iniciado*, *pronto* e *habilitado*, como nos exemplos a seguir:

- O sistema não pode ter *iniciado* e ainda não estar *pronto* para atender requisições:

$$G\neg(iniciado \wedge \neg\ pronto) \tag{2.13}$$

- Em todo sistema, se é *requisitado* algum recurso, o requisitante deve ser *respondido*:

$$G(requisitado \rightarrow F\ respondido) \tag{2.14}$$

- Um processo é *habilitado* infinitas vezes em todo o caminho de computação:

$$GF\ habilitado \tag{2.15}$$

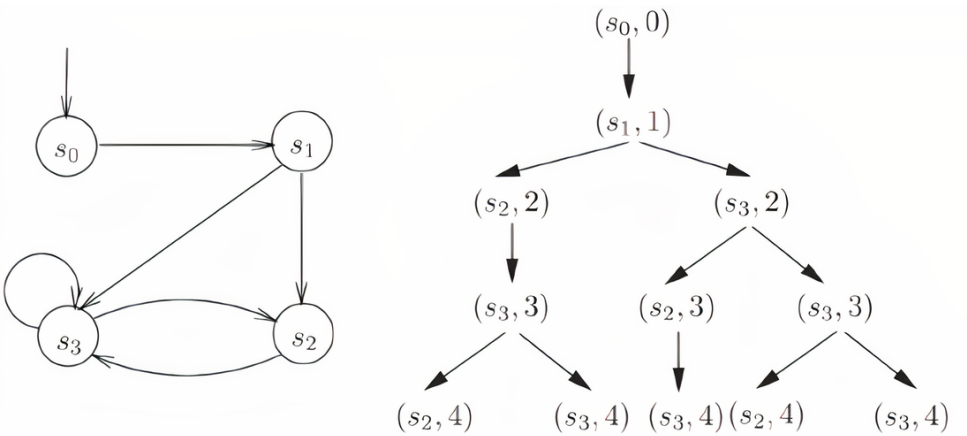
- Um elevador subindo para o segundo andar não muda sua direção quando houver passageiros que desejam ir para o quinto andar:

$$G(andar2 \wedge subindo \wedge pressionadoBotao5 \rightarrow (subindoUandar5)) \tag{2.16}$$

Lógica temporal ramificada

A lógica temporal ramificada (do inglês, *computational tree logic* (CTL)), ou lógica de computação em árvore (CTL), é capaz de representar sistemas de transições de estados em que o futuro não é determinado, ou seja, quando há uma variedade de possíveis caminhos que podem ser tomados a partir de um determinado estado. Sendo assim, a evolução do sistema ao longo do tempo pode ser representada por uma árvore que retrata todas as execuções possíveis (HUTH; RYAN, 2004). Na Figura 16, à esquerda, é apresentado um modelo de estados, e à direita, a árvore de computação que representa as mudanças de estados, retratados pelo rótulo do nó e o nível de profundidade da execução.

Figura 16 – Exemplo de árvore de computação num sistema de transição.



Fonte: Huth e Ryan (2004).

A sintaxe de construção de fórmulas CTL estende a LTL e adiciona os operadores existencial (E) e universal (A), e consiste em:

$$\begin{aligned}&\varphi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \\ &AX\varphi \mid EX\varphi \mid AF\varphi \mid EF\varphi \mid AG\varphi \mid EG\varphi \mid A[\varphi U \varphi] \mid E[\varphi U \varphi]\end{aligned} \tag{2.17}$$

Como exposto na definição 2.17, os conectivos temporais da CTL são usados em pares, sendo o primeiro um quantificador, denotado por A ou E, enquanto que o segundo é um operador temporal, usado da mesma forma que na LTL. O quantificador A expressa que, a partir de um determinado estado, a propriedade é verdadeira para todos os caminhos de computação possíveis. O operador E sugere que, a partir de um determinado estado, há ao menos um caminho em que a propriedade é verdadeira (KATOEN, 1999). Alguns exemplos de fórmulas CTL são apresentados a seguir:

- É possível chegar a um estado em que o sistema está *iniciado* mas não está *pronto*?

$$EF(iniciado \wedge \neg pronto) \tag{2.18}$$

- Em qualquer caminho e em todo este caminho, se algum recurso é *requisitado*, ele eventualmente é *reconhecido*?

$$AG(requisitado \rightarrow AF\ reconhecido) \tag{2.19}$$

- Um determinado processo está *habilitado* infinitamente em cada caminho de computação.

$$AG(AF\ habilitado) \tag{2.20}$$

Assim como na LTL, as fórmulas em CTL são interpretadas sobre os caminhos a partir de estados em um sistema de transição (KATOEN, 1999). Dado o modelo de transição $\mathscr{M} = (S, \rightarrow, L)$, a relação de satisfação $\mathscr{M}, s \models \varphi$ pode ser dada indutivamente conforme é exibido na Figura 17, em que $s \in \mathscr{M}$ e φ é uma fórmula em CTL (MURA, 2016).

Na CTL, os quantificadores existencial e universal podem ser aninhados para expressar propriedades mais complexas. Por exemplo, uma propriedade em que para todo caminho de computação sempre é possível retornar para o estado inicial, pode ser representada por $AG\ (EF\ (inicio))$. A partir desta fórmula CTL, interpreta-se que, em todo estado do sistema (G), para todo caminho de computação (A), existe a possibilidade (E) de eventualmente retornar ao início ($F\ inicio$) (BAIER; KATOEN, 2008).

Figura 17 – Semântica da CTL.

$$\mathscr{M},s \models \top \tag{2.21}$$
$$\mathscr{M},s \not\models \bot \tag{2.22}$$
$$\mathscr{M},s \models p \iff p \in L(s) \tag{2.23}$$
$$\mathscr{M},s \models \neg\varphi \iff \pi \not\models \varphi \tag{2.24}$$
$$\mathscr{M},s \models \varphi \wedge \varphi' \iff \mathscr{M},s \models \varphi \text{ e } \mathscr{M},s \models \varphi' \tag{2.25}$$
$$\mathscr{M},s \models \varphi \vee \varphi' \iff \mathscr{M},s \models \varphi \text{ ou } \mathscr{M},s \models \varphi' \tag{2.26}$$
$$\mathscr{M},s \models \varphi \rightarrow \varphi' \iff \mathscr{M},s \not\models \varphi \text{ ou } \mathscr{M},s \models \varphi' \tag{2.27}$$
$$\mathscr{M},s \models AX\varphi \iff \forall s_1 \mid s \rightarrow s_1 \text{ com } \mathscr{M},s_1 \models \varphi \tag{2.28}$$
$$\mathscr{M},s \models EX\varphi \iff \exists s_1 \mid s \rightarrow s_1 \text{ com } \mathscr{M},s_1 \models \varphi \tag{2.29}$$
$$\mathscr{M},s \models AG\varphi \iff \forall \pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \mid s_1 = s \text{ e } \forall s_i \in \pi \text{ com } \mathscr{M},s_i \models \varphi \tag{2.30}$$
$$\mathscr{M},s \models EG\varphi \iff \exists \pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \mid s_1 = s \text{ e } \forall s_i \in \pi \text{ com } \mathscr{M},s_i \models \varphi \tag{2.31}$$
$$\mathscr{M},s \models AF\varphi \iff \forall \pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \mid s_1 = s \text{ e } \exists s_i \in \pi \mid \mathscr{M},s_i \models \varphi \tag{2.32}$$
$$\mathscr{M},s \models EF\varphi \iff \exists \pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \mid s_1 = s \text{ e } \exists s_i \in \pi \mid \mathscr{M},s_i \models \varphi \tag{2.33}$$
$$\mathscr{M},s \models A[\varphi U \varphi'] \iff \forall \pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \mid s_1 = s \exists s_i \in \pi \mid \tag{2.34}$$
$$\mathscr{M},s_i \models \varphi' \text{ e } \forall j < i \text{ com } \mathscr{M},s_j \models \varphi \tag{2.35}$$
$$\mathscr{M},s \models E[\varphi U \varphi'] \iff \exists \pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots \mid s_1 = s \exists s_i \in \pi \mid \tag{2.36}$$
$$\mathscr{M},s_i \models \varphi' \text{ e } \forall j < i \text{ com } \mathscr{M},s_j \models \varphi \tag{2.37}$$

Fonte: Mura (2016).

CAPÍTULO

3

METODOLOGIA E TÉCNICAS DE PESQUISA

Este capítulo tem a finalidade de apresentar como a pesquisa foi estruturada. Primeiramente, foram realizadas buscas por trabalhos que abordam a tecnologia blockchain, como estudos secundários e livros. Desta forma, obteve-se um panorama sobre os conceitos gerais envolvidos no funcionamento da blockchain, assim como características específicas que diferenciam blockchains distintas, em particular, a Bitcoin e a Ethereum. Assim, também foi observado em vários trabalhos que aplicações que executam sobre a blockchain Ethereum sofreram diversos ataques, especialmente devido à exploração de vulnerabilidades sobre os CIs, que foram um elemento crucial para a criação da Ethereum, bem como para sua popularização. Com base nisso, foi empregada uma estratégia para revisão bibliográfica conhecida como Mapeamento Sistemático (MS), empregada neste trabalho com o objetivo de identificar e classificar o conteúdo relacionado à verificação para correção ou detecção de vulnerabilidades para aprimoramento da segurança dos CIs. O MS foi conduzido seguindo os métodos descritos por Nakagawa *et al.* (2017) e Kitchenham e Charters (2007). Para descrever como foi conduzido o processo de pesquisa, tal como os métodos empregados, foram seguidas as seguintes tarefas:

1. Planejamento e execução do MS;
2. Seleção do método de verificação e detecção de vulnerabilidades adequado para a proposta;
3. Seleção das vulnerabilidades abordadas por meio da proposta;
4. Definição da estratégia para validação da proposta.

3.1 Mapeamento Sistemático

O processo de MS empregado neste trabalho é realizado em três fases: (i) planejamento; (ii) condução; e (iii) publicação dos resultados. O planejamento do MS é a fase na qual é definido

o objetivo e o protocolo do MS. No protocolo são especificados os procedimentos necessários para identificação dos estudos primários, tais como: questões de pesquisa; estratégia de busca; fontes de pesquisa; *string* de busca; e critérios de seleção. A fase de condução consiste em identificar os estudos primários com base nas estratégias de busca e seleção, além de extrair e sintetizar os dados de forma a auxiliar o processo de resposta das questões de pesquisa. Por fim, na fase de publicação dos resultados, as questões de pesquisa são respondidas e os resultados são relatados e avaliados (NAKAGAWA *et al.*, 2017; KITCHENHAM; CHARTERS, 2007).

Adiante, na Seção 3.1.1, são descritas as fases de planejamento e condução do MS, nas quais os itens relacionados ao protocolo do MS são retratados, os estudos primários são identificados e submetidos à extração de seus dados, e a estratégia de sintetização dos dados é definida. Em seguida, na Seção 3.1.2, os resultados são obtidos e as questões de pesquisa respondidas. Por fim, na Seção 3.1.3 há uma discussão acerta dos resultados obtidos.

3.1.1 Planejamento e condução

O MS realizado neste trabalho tem o objetivo de revisar o estado da arte dos estudos realizados que abordam a verificação para correção ou detecção de vulnerabilidades para aprimoramento da segurança dos CIs, assim como classificar as abordagens empregadas e suas formas de implementação, identificar limitações existentes, e, por fim, analisar as estratégias de validação utilizadas.

Com o intuito de guiar o processo de revisão e satisfazer os objetivos, foram formuladas as questões de pesquisa. O foco das questões de pesquisas estão em identificar as abordagens utilizadas, assim como suas limitações e seus respectivos procedimentos de validação. Deste modo, baseado no objetivo deste MS, foram levantadas cinco questões de pesquisa (QP). São elas:

- **QP 1.** Quais abordagens têm sido propostas?
- **QP 2.** Quando e onde os estudos têm sido aplicados?
- **QP 3.** Quais problemas ou vulnerabilidades relacionados aos CIs têm sido abordados?
- **QP 4.** Quais estratégias de validação foram utilizadas?
- **QP 5.** Quais são as limitações presentes nas abordagens?

Para levantamento dos estudos necessários para responder as questões de pesquisa, foram utilizados os motores de busca *Engineering Village* [1], *Scopus* [2] e *Web of Science* [3]. Como

[1] <https://www.engineeringvillage.com/>
[2] <https://www.scopus.com/>
[3] <https://www.webofknowledge.com>

complemento, a pesquisa também foi realizada sobre as bases de dados bibliográficas *IEEE Xplore* [4] e *ACM Digital Library* [5]. Todos os motores de busca e bases bibliográficas selecionados permitem especificar os campos sobre os quais a busca é executada. Neste MS, os repositórios foram configurados para efetuar a busca sobre o título, o resumo e as palavras-chave dos estudos. Além disso, no motor de busca *Web of Science*, os resultados foram refinados para abranger apenas as categorias relacionadas à ciência da computação.

Com os repositórios de artigos definidos, o passo seguinte foi determinar a *string* de busca, composta por palavras-chave e operadores lógicos para conduzir o processo de busca nos repositórios. Como exposto por Nakagawa *et al.* (2017), o MS é um processo iterativo, pois, ao longo de seu desenvolvimento, pode-se notar a necessidade de ajustes no objetivo e no protocolo, que são então modificados, o protocolo é reavaliado, e o processo de condução recomeçado. Ao longo desta pesquisa, foram necessários alguns ajustes na *string* de busca, a fim de adequar as palavras-chave escolhidas para abranger mais artigos relacionados com o objetivo deste MS. Por fim, obteve-se a seguinte *string* de busca:

(*"smart contract" OR "ethereum bytecode"*) *AND* (*verification OR validation OR monitor* OR analysis OR formalization OR "formal methods" OR "security vulnerabilities" OR "security bugs" OR "vulnerability detection" OR "bug detection" OR optimiz**)

Tabela 2 – Quantidade de estudos encontrados nos repositórios

Repositórios	**Nº de estudos**
Engineering Village	865
Scopus	906
Web of Science	170
ACM Digital Library	65
IEEE Xplore	85
Total	**2091**

Fonte: Dados da pesquisa.

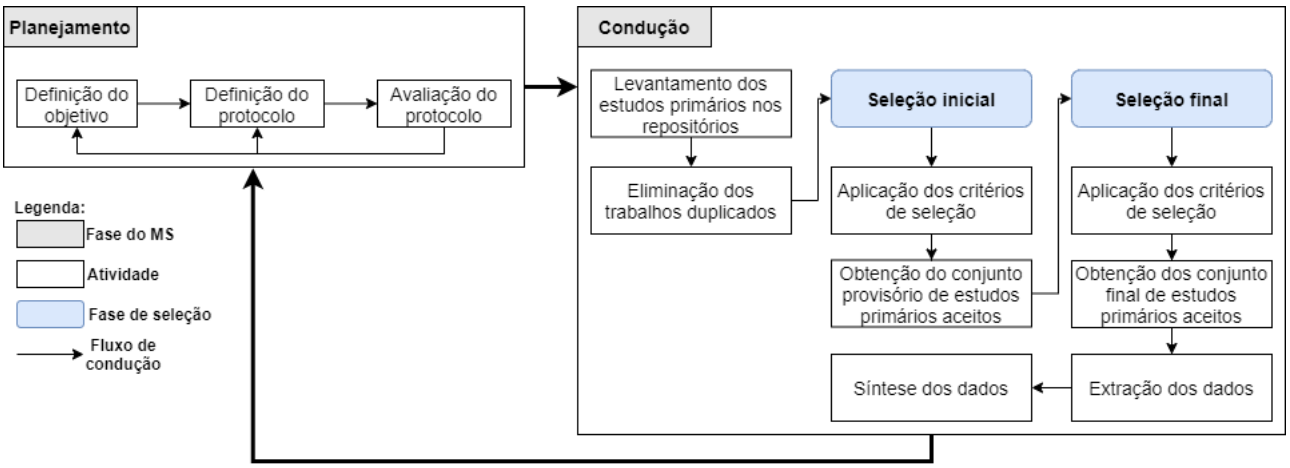
Uma vez definida a *string* de busca, as buscas nos repositórios foram realizadas. Considerando a eliminação de artigos duplicados, a quantidade de estudos encontrados em cada repositório é apresentada na Tabela 2. O levantamento dos trabalho ocorreu semanalmente, assim, o mecanismo de busca das bases bibliográficas e dos motores de busca foram configurados para enviarem atualizações semanais.

Após o levantamento dos artigos, foram aplicadas duas fases de revisão para selecionar apenas os trabalhos relevantes para esta pesquisa. Na primeira fase, a revisão inicial, o resumo de cada artigo foi lido, e, quando necessário, a introdução e a conclusão também, e então cada

[4] <https://ieeexplore.ieee.org/>
[5] <https://dl.acm.org/>

trabalho foi incluído ou excluído da revisão de acordo com critérios de seleção predefinidos. Em caso de dúvida, o estudo foi aceito na revisão inicial para ser novamente analisado na fase seguinte. Assim, obteve-se o conjunto provisório de estudos primários aceitos, sobre o qual foi realizada a fase de revisão final. Nesta fase, cada artigo teve todo seu texto lido, e então foi novamente submetido aos critérios de seleção. O processo de planejamento e condução deste MS é ilustrado na Figura 18.

Figura 18 – Fluxo de atividades das fases de planejamento e condução do MS



Fonte: Elaborada pelo autor.

Para selecionar os estudos relacionados ao tópico de pesquisa deste MS, foi definido o seguinte critério de inclusão (CrI):

- **CrI 1.** O estudo propõe uma abordagem que é utilizada para verificação formal para correção ou detecção de vulnerabilidades para garantia de segurança de CIs?

Os estudos excluídos foram rejeitados de acordo com sua adequação à pelo menos um dos seguintes critérios de exclusão (CE):

- **CE 1.** O estudo não propõe uma abordagem que é utilizada para verificação formal para correção ou detecção de vulnerabilidades para garantia de segurança de CIs?
- **CE 2.** O estudo não possui o texto completo publicado?
- **CE 3.** O estudo não é escrito na língua inglesa?
- **CE 4.** O estudo é uma versão antiga de outro artigo selecionado?
- **CE 5.** O estudo não é um estudo primário?
- **CE 6.** Não foi possível acessar o texto completo do estudo?

Depois da eliminação dos estudos duplicados e da execução das fases de seleção inicial e final, obteve-se os estudos primários necessários para responder às questões de pesquisa. A quantidade de estudos aceitos e rejeitados de cada repositório é exposta na Tabela 3, somando um total de 104 estudos selecionados. Na Tabela 4 é listada a citação e o identificador (*ID*) de cada estudo selecionado. No decorrer deste capítulo, os identificadores são utilizados para se referir à publicação correspondente.

Tabela 3 – Quantidade de estudos primários selecionados em cada repositório

Repositórios	**Aceitos**	**Rejeitados**
Engineering Village	41	824
Scopus	47	859
Web of Science	5	165
ACM Digital Library	7	58
IEEE Xplore	4	81
Total	**104**	**1987**

Fonte: Dados da pesquisa.

Tabela 4 – Estudos selecionados

ID	Citação	ID	Citação	ID	Citação
#01	Torres *et al.* (2020)	#36	Prechtel, Groß e Müller (2019)	#71	Li *et al.* (2020b)
#02	Fu *et al.* (2019)	#37	Kolluri *et al.* (2019)	#72	Li *et al.* (2020a)
#03	Sun e Yu (2020)	#38	Yang e Lei (2019)	#73	Albert *et al.* (2019)
#04	Wang, Yang e Li (2020)	#39	Nikolić *et al.* (2018)	#74	Chang *et al.* (2019)
#05	Park *et al.* (2018)	#40	Duo, Xin e Xiaofeng (2020)	#75	Hao *et al.* (2020)
#06	Du e Huang (2020)	#41	Bai *et al.* (2018)	#76	Tsankov *et al.* (2018)
#07	Yang, Lei e Qian (2020)	#42	Liu e Liu (2019)	#77	Li, Choi e Long (2020)
#08	Xing *et al.* (2020)	#43	Li *et al.* (2019)	#78	Kongmanee, Kijsanayothin e Hewett (2019)
#09	Horta *et al.* (2020)	#44	Abdellatif e Brousmiche (2018)	#79	Zhou *et al.* (2018)
#10	Marescotti *et al.* (2020)	#45	Qu *et al.* (2018)	#80	Peng, Akca e Rajan (2019)
#11	Lahbib *et al.* (2020)	#46	Madl *et al.* (2019)	#81	Feist, Grieco e Groce (2019)
#12	Weiss e Schütte (2019)	#47	Bhargavan *et al.* (2016)	#82	Tian (2019)
#13	Wang *et al.* (2020a)	#48	Wang *et al.* (2019)	#83	Yu *et al.* (2020)
#14	Sun e Gu (2021)	#49	Ding *et al.* (2020)	#84	Zhuang *et al.* (2020)
#15	Beillahi *et al.* (2020)	#50	Chen *et al.* (2020b)	#85	Tikhomirov *et al.* (2018)
#16	Gao *et al.* (2020)	#51	Ashraf *et al.* (2020)	#86	Zhang *et al.* (2020)
#17	Marescotti *et al.* (2018)	#52	Wüstholz e Christakis (2020)	#87	Alt e Reitwiessner (2018)
#18	Jiang, Liu e Chan (2018)	#53	Huang *et al.* (2021)	#88	Akca, Rajan e Peng (2019)
#19	Wang *et al.* (2019)	#54	Momeni, Wang e Samavi (2019)	#89	Hajdu e Jovanović (2019)
#20	Wang *et al.* (2020)	#55	Grech *et al.* (2018)	#90	Liao *et al.* (2019)
#21	Xue *et al.* (2020)	#56	Luu *et al.* (2016)	#91	Lai e Luo (2020)
#22	Shishkin (2019)	#57	Mossberg *et al.* (2019)	#92	Feng, Torlak e Bodik (2020)
#23	Nehaï e Bobot (2019)	#58	Osterland e Rose (2020)	#93	Krupp e Rossow (2018)
#24	Hirai (2017)	#59	Nehai, Piriou e Daumas (2018)	#94	Torres, Steichen *et al.* (2019)
#25	Ji *et al.* (2020)	#60	He (2020)	#95	Mavridou e Laszka (2018)
#26	Wang, Zhang e Su (2019)	#61	Azzopardi, Ellul e Pace (2018)	#96	Qian *et al.* (2020)
#27	Argañaraz *et al.* (2020)	#62	Zhang *et al.* (2019)	#97	Liu *et al.* (2020)
#28	Atzei *et al.* (2019)	#63	Lu *et al.* (2019)	#98	Chen *et al.* (2018)
#29	Albert *et al.* (2021)	#64	Wang *et al.* (2020b)	#99	Amani *et al.* (2018)
#30	Gao *et al.* (2019)	#65	Torres, Schütte e State (2018)	#100	Ahrendt *et al.* (2019)
#31	Brent *et al.* (2020)	#66	Yamashita *et al.* (2019)	#101	Nelaturu *et al.* (2020)
#32	Frank, Aschermann e Holz (2020)	#67	Chatterjee, Goharshady e Velner (2018)	#102	So *et al.* (2020)
#33	Ashouri (2020)	#68	Chinen *et al.* (2020)	#103	Mavridou *et al.* (2019)
#34	Schneidewind *et al.* (2020)	#69	Samreen e Alalfi (2020)	#104	Permenev *et al.* (2020)
#35	Zhang *et al.* (2020)	#70	Liu *et al.* (2018)		

Fonte: Dados da pesquisa.

A partir do conjunto final de estudos primários aceitos, foi feita a extração de dados desses trabalhos, na qual informações foram coletadas com o intuito de auxiliar na resposta das questões de pesquisa.

Para condução de um MS, é preciso determinar um esquema de classificação para guiar a extração e sintetização dos dados (PETERSEN *et al.*, 2008). Para as questões de pesquisa **QP 1**, **QP 3** e **QP 4** a estratégia de classificação foi definida levando em consideração informações coletadas dos próprios estudos selecionados. Para a **QP 2** não foi necessário determinar um esquema de classificação, já que trata apenas de informações como o ano e o veículo de publicação dos estudos. Para a **QP 4** alguns aspectos dos estudos foram analisados, mas nem todos possuem uma classificação direta, já que alguns dados foram coletados de forma não estruturada e sem uma classificação predefinida. As categorias e classificações definidas são expostas a seguir.

*Abordagens propostas (**QP 1**)*

Diversas técnicas foram propostas para verificação e correção de CIs com o intuito de mitigar problemas de segurança e a exposição à ataques que visam a exploração de vulnerabilidades no códigos dos contratos para obtenção de benefícios. A partir dos estudos selecionados, foram consideradas dez categorias. São elas: Análise estática; análise dinâmica; análise simbólica; execução simbólica; demonstração de teoremas; *model checking*; verificação dedutiva; *fuzzing*; IA; e Outras.

Nos trabalhos selecionados, observou-se quatro variações das abordagens baseadas em *model checking*, como o *model checking* tradicional, o simbólico, o limitado, e o estatístico. Entre as abordagens que utilizam IA, duas técnicas específicas são utilizadas: *machine learning*; e *deep learning*. As variações das abordagens de *model checking* e IA são consideradas como sub-categorias e são tratadas separadamente nos resultados da *QP 1*. Já as técnicas observadas em apenas um dos estudos selecionados são englobadas na categoria Outras.

As abordagens também são classificadas de acordo com o tipo de verificação, sendo que, neste MS, foram identificados dois tipos: proativa; e em tempo de execução.

*Ano e veículo de publicação dos estudos (**QP 2**)*

Para responder esta questão de pesquisa, foi contabilizada a quantidade de estudos publicados em cada ano. Para cada ano, também foi observado quantos estudos se enquadram em cada uma das abordagens para verificação definidas na **QP 1**. Ademais, os veículos de publicação foram classificados como Conferências, Periódicos e Workshops.

*Problemas e vulnerabilidades abordados (**QP 3**)*

Os estudos realizados para verificação de CIs têm como foco diversas vulnerabilidades e problemas para serem verificados. Além da detecção de vulnerabilidades específicas, várias abordagens permitem a detecção de violação de propriedades específicas para cada contrato, que podem ser propriedades funcionais, lógicas, ou baseadas nos requerimentos e requisitos de cada CI. Neste MS, a violação de propriedades é definida com uma categoria de problemas e vulnerabilidades abordadas. Ademais, algumas abordagens têm foco na correção sintática e

semântica dos CIs, o que define mais uma categoria a ser analisada. As vulnerabilidades de segurança tratadas pelos estudos são classificadas em 23 tipos. Na Tabela 5 são listadas todas as vulnerabilidades e problemas considerados para responder esta questão de pesquisa, cada um acompanhado da respectiva sigla.

*Estratégias de validação (**QP 4**)*

De forma geral, nas pesquisas desenvolvidas para verificação de CIs, alguma ferramenta ou framework é implementado para validação do método proposto, ou é utilizado algum já existente. Em alguns casos, o método proposto não é implementado, e sua aplicação é descrita apenas como um processo. Em todos os casos, alguma estratégia é utilizada para validação da proposta. Baseado nos estudos selecionados, os mecanismos de aplicação das propostas (*M*) são definidos conforme segue:

- *M1*. Implementação de um framework ou ferramenta;
- *M2*. Utilização de um framework ou ferramenta já existente;
- *M3*. Descrição de um processo.

As estratégias de validação das propostas são classificadas em três categorias, definidas com base nos estudos selecionados. São elas:

- **Experimento:** A avaliação da proposta é feita a partir da definição de métricas como acurácia, eficiência, custo, performance, entre outras. O processo de experimentação dos métodos de verificação de CIs englobam ao menos um dos seguintes procedimentos: (i) experimento em larga escala, no qual é obtida uma amostra grande, geralmente com milhares de CIs com vulnerabilidades previamente conhecidas ou não; (ii) experimento reduzido, em que é utilizada uma amostra relativamente pequena de CIs, que pode variar de algumas unidades até centenas de contratos, geralmente com vulnerabilidades previamente conhecidas. Em alguns casos, essa amostra de contratos é obtida a partir de critérios predefinidos, formando um *benchmark* de CIs; (iii) avaliação empírica, na qual uma ou mais ferramentas, frameworks ou técnicas conhecidas na literatura são utilizados no experimento para comparação dos resultados e comprovação de avanços e melhorias;
- **Estudo de caso:** Como prova de conceito, um ou mais CIs previamente conhecidos são analisados e verificados por meio do método proposto. Na sequência, os resultados são examinados, e a aplicabilidade da proposta é esclarecida;
- **Exemplo de aplicação:** Como prova de conceito, são descritos um ou mais exemplos simples de utilização da proposta.

Limitações (QP 5)

A verificação e detecção de vulnerabilidades em CIs é uma área de pesquisa recente, e, como relatado por Chen *et al.* (2020a) e Kim e Lee (2020), as abordagens existentes apresentam diversas limitações. Para responder esta questão de pesquisa, dois aspectos foram considerados para classificação sistemática dos estudos. São eles:

- **Abrangência limitada**: A verificação não abrange todos os elementos dos CIs, mas apenas algum sub-conjunto da linguagem Solidity;
- **Nível de automação**: Mesmo que a estratégia proposta execute a verificação de forma automática, alguns trabalhos manuais podem ser necessários, como tradução de código, obtenção do modelo do contrato, especificação de propriedades, e a análise e interpretação dos resultados.

Outros dois aspectos que foram aspectos observados, mas que não são classificados de forma sistemática, são:

- **Necessidade de conhecimento prévio**: Para aplicação da abordagem é necessário ter o domínio de alguma linguagem ou formalismo específico para modelagem do contrato ou para definição dos padrões de vulnerabilidades e propriedades a serem verificadas;
- **Acurácia**: A ferramenta ou framework proposto para detecção de vulnerabilidades de CIs retornar falsos positivos e falsos negativos entre seus resultados.

Dentre os aspectos analisados para definir as limitações das abordagens de verificação, os dados sobre a abrangência limitada e o nível de automação foram coletados de forma estruturada sobre todos os estudos. Para as demais limitações, não foi estabelecida uma classificação predefinida, pois estas podem variar de acordo com a abordagem empregada e não seguem um padrão que permita uma classificação sistemática dos estudos.

Tabela 5 – Vulnerabilidades e problemas alvos de verificação em CIs

Sigla	Vulnerabilidade / Problema	Sigla	Vulnerabilidade / Problema
V_1	Ataque de profundidade da pilha de chamadas	V_{14}	Dependência de *timestamp*
V_2	Ataque DoS com operações ilimitadas	V_{15}	Desordem de exceções
V_3	Autenticação com `tx.origin`	V_{16}	Divisão por zero
V_4	Bloqueio de Ether	V_{17}	Endereço curto
V_5	Consumo de *gas* ineficiente	V_{18}	Exceções não tratadas
V_6	Contrato guloso	V_{19}	Chamada externa não verificada
V_7	Contrato pródigo	V_{20}	*Integer overflow* e *underflow*
V_8	Contrato suicida	V_{21}	Gasto de *gas* descontrolado
V_9	Contrato *honeypot*	V_{22}	Problemas de concorrência
V_{10}	Controle de acesso vulnerável	V_{23}	Reentrância
V_{11}	*Delegatecall injection*	VP	Violação de propriedades
V_{12}	Dependência de informação do bloco	CSS	Correção sintática e semântica
V_{13}	Dependência de ordem da transação		

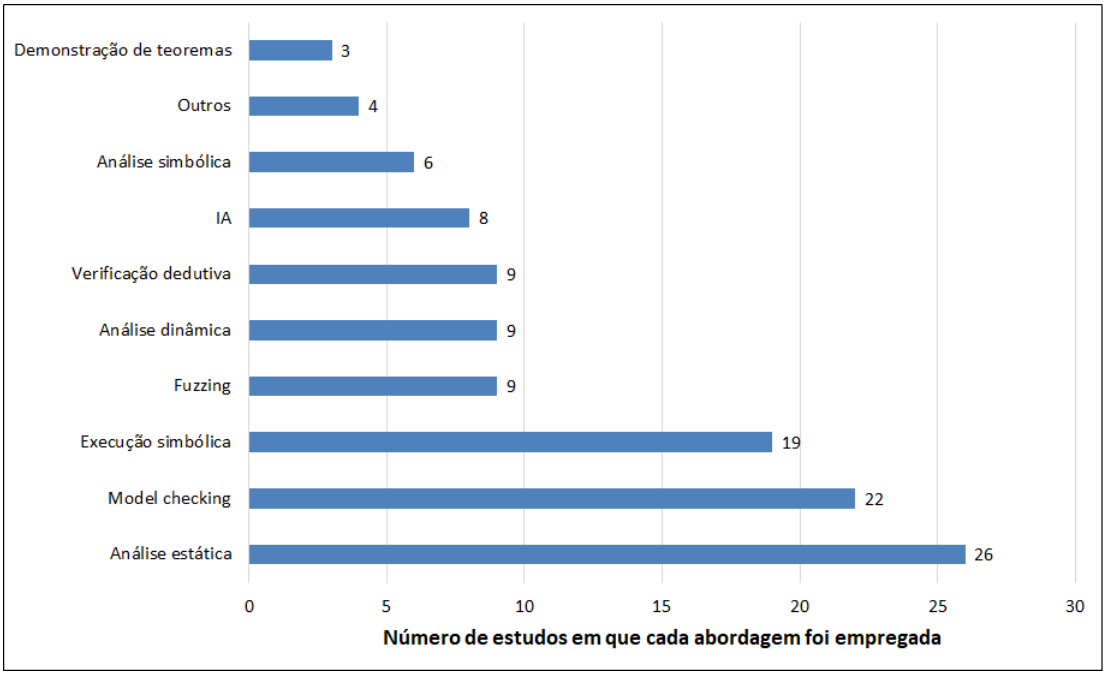
3.1.2 Resultados

O MS foi executado de acordo com os passos descritos na Seção 3.1.1. Nesta seção são apresentados os resultados para cada questão de pesquisa. Para isso, foi utilizado um formulário de extração de dados com o ID do estudo, dados bibliográficos e informações referentes às classificações utilizadas. Tais dados e informações foram utilizados para extrair as respostas das questões de pesquisa, e, posteriormente, para exibição dos resultados.

Abordagens para verificação de CIs (QP 1)

Em 93 dos trabalhos selecionados é utilizada apenas uma das abordagens categorizadas nesta questão de pesquisa, com exceção de 11 trabalhos, que utilizam abordagens híbridas, nas quais duas técnicas são utilizadas em conjunto. Com o intuito de expor as principais abordagens mais utilizadas para verificação para correção e aprimoramento da segurança de CIs, no gráfico da Figura 19 é apontado o número de estudos em que cada abordagem foi empregada dentre 104 selecionados, na qual pode-se observar uma prevalência das técnicas de execução simbólica, *model checking* e análise estática. Na tabela 6 são mostrados os estudos que empregaram abordagens híbridas.

Figura 19 – Abordagens para verificação de CIs utilizadas



Fonte: Dados da pesquisa.

Entre as abordagens baseadas em *model checking* observou-se quatro variações da aplicação desta técnica, consideradas como sub-categorias desta questão de pesquisa. Da mesma forma, também foram identificadas duas sub-categorias entre as abordagens de IA. Os estudos que utilizam *model checking* são expostos na Tabela 7. Das abordagens baseadas em IA, em

Tabela 6 – Abordagens híbridas para verificação de CIs

ID	Métodos de verificação
#02, #07, #62, #79	Análise estática e execução simbólica
#69, #76, #88	Análise estática e análise dinâmica
#37	Execução simbólica e *fuzzing*
#38	Execução simbólica e demonstração de teoremas
#64	Análise dinâmica e *fuzzing*
#90	*Fuzzing* e *machine learning*

Fonte: Dados da pesquisa.

#08,#14, #20, #54, #84 e #90 são aplicadas técnicas de *machine learning*, enquanto que em #16 e #96 é utilizado *deep learning*. Dos estudos englobados na categoria *outras*, foram aplicadas as seguintes técnicas de verificação: execução concólica (#12); refinamento de abstração (#67); pesquisa genética e teste de mutação (#83); e interpretação abstrata (#95).

Quanto ao tipo de verificação houve pouca variação, com 96% (100 de 104) dos estudos propondo uma estratégia de verificação proativa. A verificação em tempo de execução é pouco explorada, e está presente apenas em #19, #30, #61 e #77 (4%).

Tabela 7 – Técnicas de *model checking* empregadas para verificação de CIs

ID	Técnica de *model checking* empregada
#04, #10, #11, #28, #40, #41 #42, #45, #46, #47, #58, #59 #60, #78, #101, #103	*Model checking* tradicional
#26, #32, #48, #87	*Model checking* limitado
#17, #22	*Model checking* simbólico
#44	*Model checking* estatístico

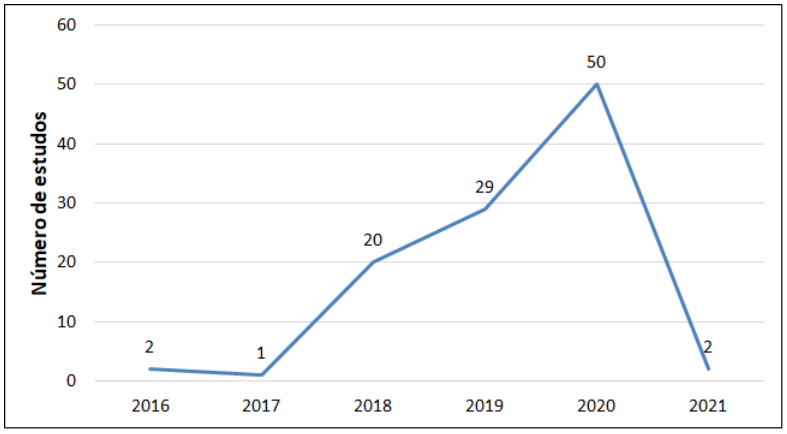
Fonte: Dados da pesquisa.

Classificação das publicações por ano e veículo de publicação (QP 2)

Com o propósito de oferecer uma visão geral dos esforços empregados para verificação de CIs, na Figura 20 é exibido a distribuição dos 104 estudos ao longo dos anos. Como pode-se observar, os esforços para verificação de CIs são recentes, iniciando em 2016 e crescendo rapidamente, principalmente a partir de 2018. Ademais, como complemento à **QP. 1**, no gráfico da Figura 21 é exposto a distribuição da utilização das abordagens para verificação de CIs em cada ano, e na Tabela 8 essa distribuição é exibida com a identificação dos estudos. Deste modo, observa-se que análise estática, *model checking* e análise simbólica, além de serem as abordagem mais utilizadas, são as que mais prevaleceram ao longo dos últimos anos.

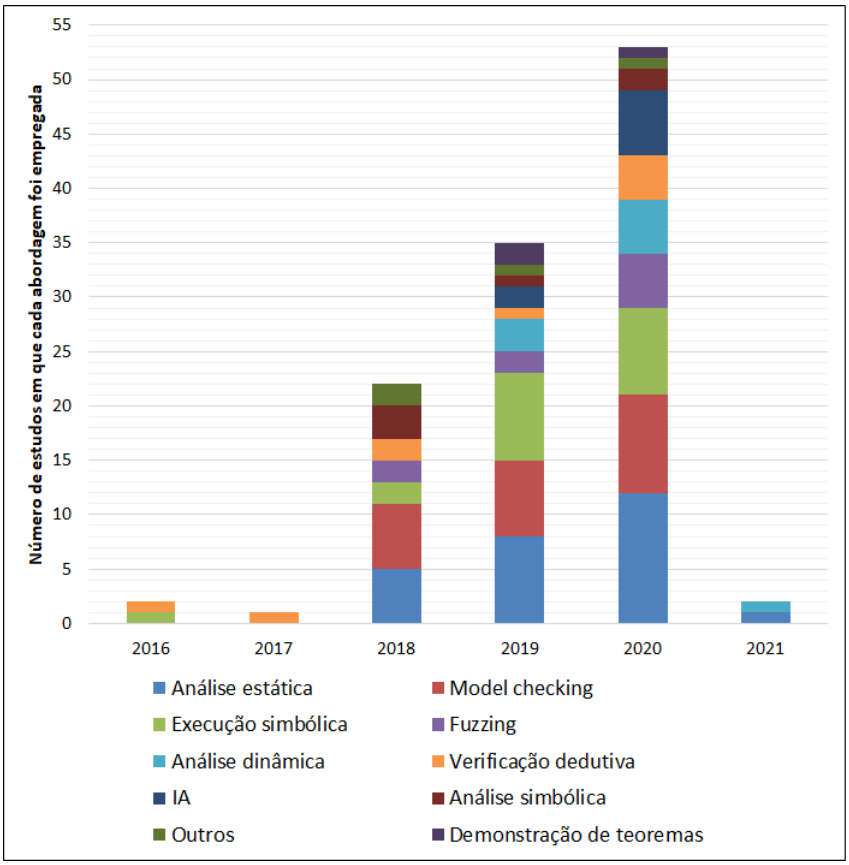
Os estudos selecionados foram publicados em Conferências, Periódicos e Workshops. Conferências têm sido o principal veículo de publicação, abrangendo 73% (76 de 104 estudos). Em seguida, os estudos publicados em Periódicos representam 22% (23 de 104). Por fim, apenas 5 estudos foram apresentados em Workshops (5%). Constata-se que, apesar da maioria dos estudos terem sido publicados em conferências, não há um meio de publicação padrão para este tipo de pesquisa, assim como os eventos e periódicos de publicação, que também são variados.

Figura 20 – Distribuição dos estudos selecionados ao longo dos anos



Fonte: Dados da pesquisa.

Figura 21 – Distribuição das abordagens para verificação de CIs empregadas ao longo dos anos



Fonte: Dados da pesquisa.

Vulnerabilidades e problemas tratados pelas abordagens de verificação de CIs (QP 3)

Na Figura 22 é exibido um gráfico que ilustra o número de estudos em que cada vulnerabilidade e problema é tratado. Desta forma, pode-se notar o foco dos esforços despendidos para

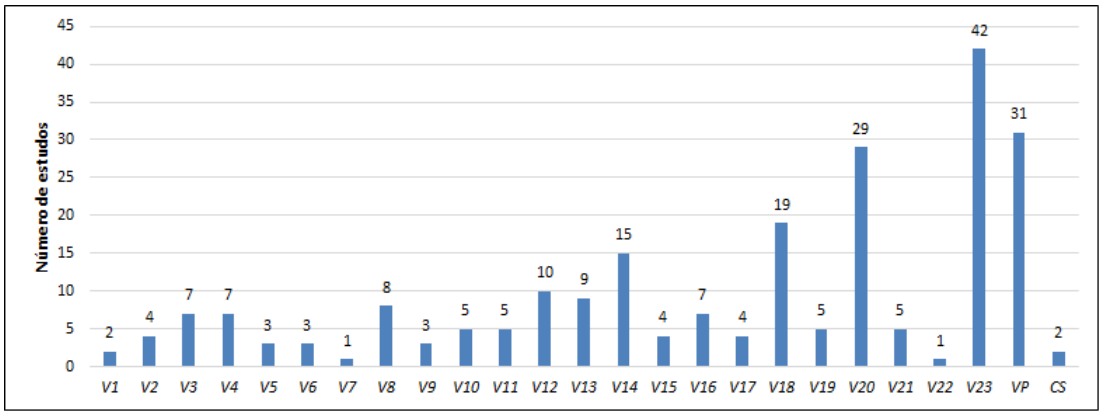
Tabela 8 – Abordagens para verificação de CIs empregadas ao longo dos anos

Abordagem	2016	2017	2018	2019	2020	2021
Análise estática			#55, #76, #79, #85 #98	#2, #62, #63, #66 #73, #80, #81, #88	#06, #07, #21, #27 #31, #34, #69, #71 #75, #77, #86, #91	#29
Model checking			#17, #41, #44, #45 #59, #87	#22, #26, #28, #42 #46, #78, #103	#04, #10, #11, #32 #40, #48, #58, #60 #101	
Execução simbólica	#56		#65, #79	#2, #36, #37, #38 #57, #62, #74, #82	#07, #13, #25, #50 #68, #89, #97, #104	
Fuzzing			#18, #70	#37, #90	#33, #35, #51, #52 #64	
Análise dinâmica				#30, #61, #88	#01, #19, #49, #64 #69	#53
Verificação dedutiva	#47	#24	#5, #99	#100	#09, #15, #23, #102	
IA				#54, #90	#08, #14, #16, #20 #84, #96	
Análise simbólica			#39, #76, #93	#94	#72, #92	
Outras			#67, #95	#12	#83	
Demonstração de teoremas				#38, #43	#03	

Fonte: Dados da pesquisa.

verificação de CIs, principalmente para as vulnerabilidades de reentrância V_{23}, *integer overflow* e *underflow* V_{20}, exceções não tratadas V_{18} e dependência de *timestamp*V_{14}, e também para a detecção de violação de propriedades (VP). Os estudos em que cada um dos itens são abordados são listados na Tabela 9.

Figura 22 – Distribuição das vulnerabilidades e problemas tratados na verificação de CIs



Fonte: Dados da pesquisa.

Estratégias de validação utilizadas (QP 4)

Dos 104 estudos selecionados, em 76 (73%) o método de verificação proposto é implementado por meio de um framework ou ferramenta (*M1*), e em 22 (21%) é utilizado um framework ou ferramenta já existente (*M2*). Nos outros 6 estudos (6%) a verificação é descrita como um processo (*M3*), e, portanto, não é retratado o desenvolvimento ou o uso de algum *software* para guiar ou automatizar o procedimento de verificação. Conforme ilustrado no gráfico

Tabela 9 – Estudos que abordam cada uma das vulnerabilidades e problemas dos CIs

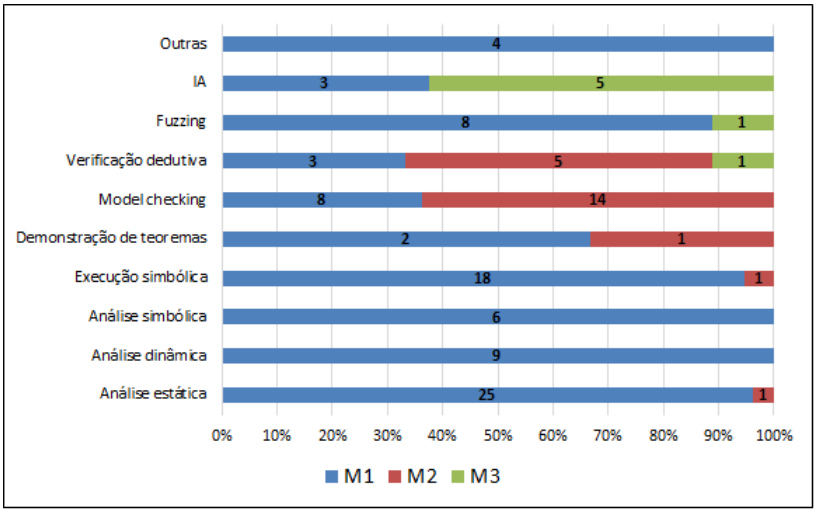
Vulnerabilidade/ Problema	Estudos	Vulnerabilidade/ Problema	Estudos
V_1	#20, #79	V_{14}	#14, #16, #18, #20, #26, #51, #56, #71 #79, #82, #84, #85, #88, #90, #92
V_2	#33, #75, #84, #90	V_{15}	#18, #51, #64, #83
V_3	#19, #27, #63, #71, #79, #85, #88	V_{16}	#65, #79, #80, #85, #87, #88, #102
V_4	#18, #28, #33, #63, #81, #85, #101	V_{17}	#08, #33, #90, #101
V_5	#50, #63, #98	V_{18}	#02, #07, #13, #18, #19, #33, #35, #36 #47, #51, #53, #56, #63, #76, #81, #85 #88, #92, #101
V_6	#08, #13, #39	V_{19}	#26, #63, #86, #90, #101
V_7	#39	V_{20}	#02, #03, #05, #07, #08, #14, #16, #19 #20, #30, #33, #52, #53, #55, #63, #56 #65, #71, #75, #80, #83, #86, #87, #88 #89, #90, #91, #92, #102
V_8	#02, #31, #32, #39, #62, #71, #81, #101	V_{21}	#16, #17, #29, #55, #98
V_9	#16, #53, #94	V_{22}	#45
V_{10}	#35, #53, #63, #75, #90	V_{23}	#01, #02, #04, #06, #14, #16, #18, #19 #20, #21, #24, #26, #27, #33, #34, #37 #47, #51, #52, #53, #56, #62, #63, #64 #68, #69, #70, #71, #75, #76, #79, #81 #82, #83, #84, #85, #86, #89, #90, #92 #96, #101
V_{11}	#02, #13, #18, #19, #31	VP	#07, #09, #10, #11, #15, #21, #22, #23, #24, #28, #34, #40, #41, #42, #43, #44, #46, #47, #58, #59, #60, #61, #73, #74, #77, #78, #89, #97, #99, #100, #103, #104
V_{12}	#02, #13, #18, #33, #51, #53, #63, #75 #90, #92	CSS	#07, #48
V_{13}	#02, #20, #26, #37, #56, #76, #79, #83 #90		

da Figura 23 a utilização de um dos mecanismos de aplicação das propostas pode variar de acordo a abordagem de verificação. Na maior parte das abordagens de verificação baseadas em *fuzzing* (8 dos 9 estudos), demonstração de teoremas (2 dos 3), execução simbólica (18 dos 19) e análise estática (25 dos 26), foram implementados frameworks ou ferramentas de verificação. A utilização de frameworks ou ferramentas já existentes prevaleceu principalmente nas categorias de verificação dedutiva e *model checking* (5 dos 9, e 14 dos 22 estudos, respectivamente). A aplicação da verificação como descrição de um processo ocorreu apenas em uma das propostas de verificação dedutiva e *fuzzing*, e também em 5 das 8 propostas baseadas em IA. Por fim, nas categorias de análise simbólica, análise dinâmica e em outras, todos os estudos empregam o mecanismo *M1*.

Entre as estratégias de validação das propostas, em 63% dos estudos foi aplicado algum experimento, enquanto que o estudo de caso e o exemplo de aplicação foram utilizados como prova de conceito em 25% e 14% dos estudos, respectivamente. Na Figura 24 é mostrada a frequência em as estratégias de validação são empregadas sobre cada uma das abordagens de verificação. Assim, nota-se uma quantidade relativamente baixa de experimentos nas propostas de verificação dedutiva e *model checking*, enquanto que na demonstração de teoremas nenhum experimento foi executado.

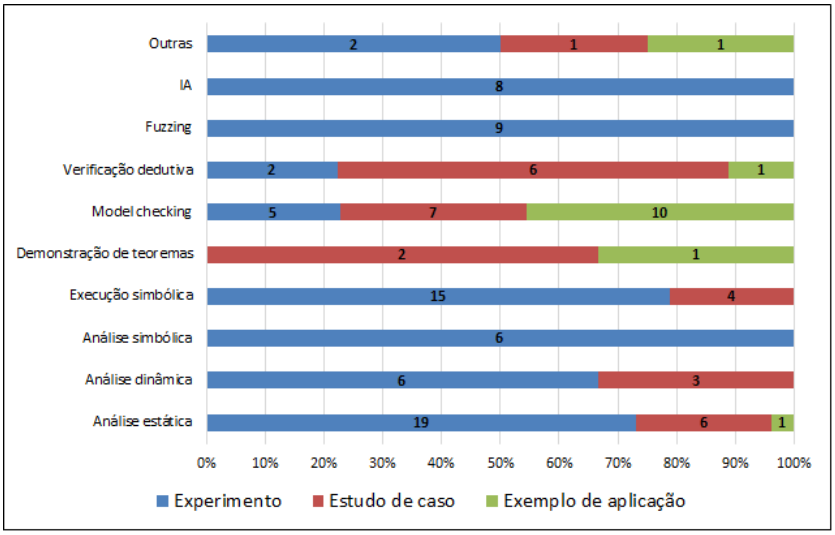
No total, 65 dos 104 estudos realizaram algum experimento para validação das pro-

Figura 23 – Utilização dos mecanismos de aplicação das propostas em cada abordagem de verificação



Fonte: Dados da pesquisa.

Figura 24 – Utilização das estratégias de validação em cada abordagem de verificação



Fonte: Dados da pesquisa.

postas, e, dentre esses, um ou mais procedimentos foram aplicados ao longo do processo de experimentação. Dos 65 estudos, em 47 (72%) é aplicado um experimento em larga escala, enquanto que em 38 (37%) é realizado um experimento reduzido e em 28 (43%) é acrescentado algum procedimento para avaliação empírica. A relação do número de estudos nos quais foram descritos ao menos um dos procedimentos experimentais citados é ilustrada no diagrama de Venn Figura 25. Vale salientar que, os números 14 e 18, que estão fora das intersecções, representam o número de estudos nos quais foi aplicado apenas o procedimento de experimento reduzido ou em larga escala, nesta ordem.

Figura 25 – Procedimentos aplicados sobre os estudos que realizaram validação experimental

Fonte: Dados da pesquisa.

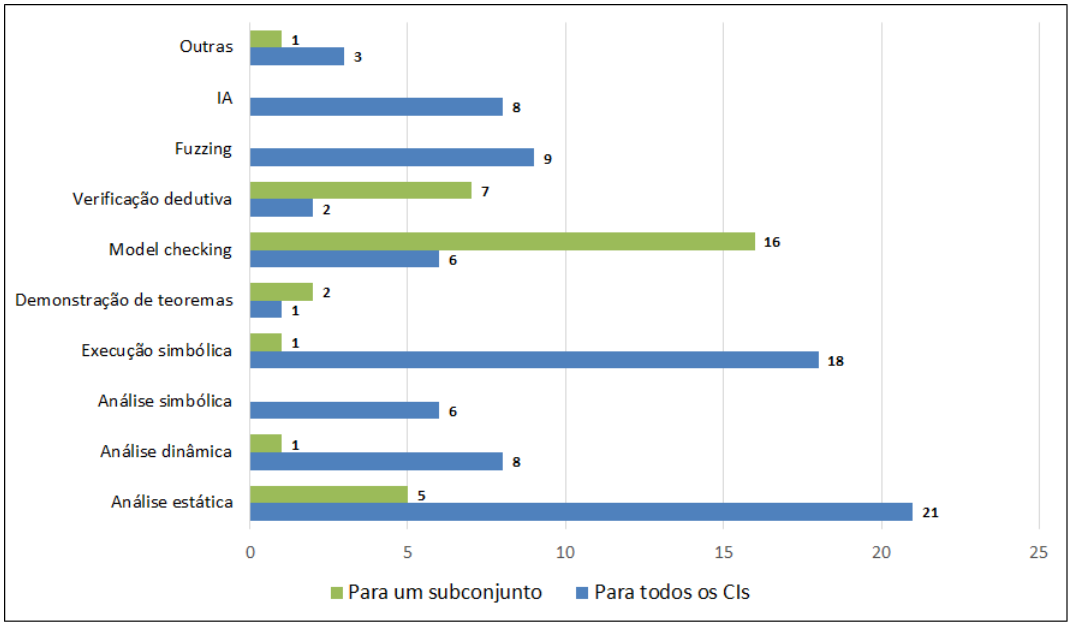
*Limitações presentes nas abordagens (**QP 5**)*

Em relação à abrangência da verificação de CIs, em 70% (73 de 104) dos estudos o método empregado engloba todos elementos da linguagem Solidity, ou outra linguagem para programação de CIs considerada, enquanto no restante (31 de 104) é considerado apenas algum subconjunto de elementos da linguagem. Quanto ao nível de automatização, em 66% (69 de 104) dos estudos é proposta uma estratégia automática para verificação, enquanto em 9% (9) a verificação é feita manualmente, ou então não é descrito nenhum mecanismo de automatização. Os outros 26 estudos (25%) empregam estratégias semi-automatizadas, das quais é necessário execução manual de ao menos um dos seguintes procedimentos: (i) tradução de código (9); obtenção do modelo a ser verificado (16); especificação das propriedades (22); e análise e interpretação dos resultados (9). Informações sobre a abrangência e o nível de automação de cada abordagem de verificação são expostas nas Figuras 26 e 27, respectivamente, e os estudos incluídos em cada uma das classificações são listados nas Tabelas 10 e 11.

Das abordagens de verificação, as únicas nas quais a maioria dos trabalhos não abrange todos os elementos dos CIs foram o *model checking* (16 entre 22 estudos), a verificação dedutiva (7 entre 9) e a demonstração de teoremas (2 entre 3). Já na análise estática, análise dinâmica, execução simbólica, e em Outras, na vasta maioria dos estudos todos os aspectos dos CIs são aceitos para verificação, enquanto que na análise simbólica, *fuzzing*, e IA isso se enquadra à todos os trabalhos selecionados.

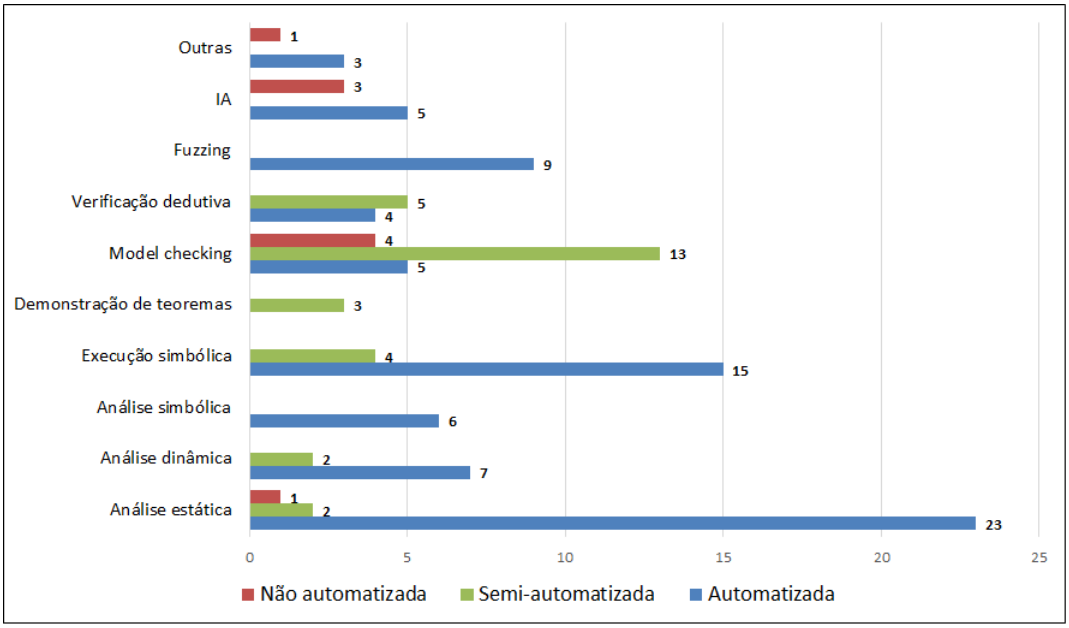
Quanto ao nível de automação, nota-se que há poucos (ou nenhum) trabalhos semi ou não automatizados entre as abordagens de análise estática, análise dinâmica, análise simbólica, *fuzzing*, e em Outras. Dentre os estudos focados em IA, 3 das 8 propostas foram classificadas como não automatizadas, pois não foram fornecidas informações suficientes para concluir que a verificação é feita de forma automática. Entre as propostas de demonstração de teoremas, *model*

Figura 26 – Abrangência das abordagem para verificação de CIs



Fonte: Dados da pesquisa.

Figura 27 – Nível de automação das abordagem para verificação de CIs



Fonte: Dados da pesquisa.

checking e verificação dedutiva, todas, ou a maioria não são automatizadas.

Com exceção da demonstração de teoremas, do *model checking* e da verificação dedutiva, foi observada a ocorrência de falsos positivos e falsos negativos entre as vulnerabilidades detectadas pelas ferramentas e frameworks implementados, principalmente quando utilizadas técnicas de análise estática e execução simbólica. De forma indireta, este problema atingiu

Tabela 10 – Abrangência e automação das abordagens de verificação de CIs

Abordagem	Abrangência		Automação		
	Para todos os CIs	**Para um subconjunto**	**Automatizada**	**Semi-Automatizada**	**Não automatizada**
Análise estática	#02, #21, #27, #29, #31 #34, #55, #62, #63, #71 #75, #76, #77, #79, #80 #81, #85, #86, #88, #91 #98	#06, #07, #66, #69, #73	#02, #21, #27, #29, #31 #34, #55, #62, #63, #66 #71, #73, #75, #76, #77 #79, #80, #81, #85, #86 #88, #91, #98	#07, #69	#06
Análise dinâmica	#39, #72, #76, #92, #93 #94		#39, #72, #76, #92, #93 #94		
Análise simbólica	#39, #72, #76, #92, #93 #94		#39, #72, #76, #92, #93 #94		
Execução simbólica	#02, #13, #25, #36, #37 #38, #50, #56, #57, #62 #65, #68, #74, #79, #82 #89, #97, #104	#07	#02, #13, #25, #36, #37 #50, #56, #57, #62, #65 #68, #74, #79, #82, #89	#07, #38, #97, #104	
Demonstração de teoremas	#38	#03, #43		#03, #38, #43	
Model checking	#10, #26, #32, #48, #101 #103	#04, #11, #17, #22, #28 #40, #41, #42, #44, #45 #46, #58, #59, #60, #78 #87	#10, #26, #32, #48, #87	#04, #11, #22, #28, #41 #44, #45, #46, #58, #59 #78, #101, #103	#17, #40, #42, #60
Verificação dedutiva	#100, #102	#05, #09, #15, #23, #24 #47, #99	#15, #23, #47, #102	#05, #09, #24, #99, #100	
Fuzzing	#18, #33, #35, #37, #51 #52, #64, #70, #90		#18, #33, #35, #37, #51 #52, #64, #70, #90		
IA	#08, #14, #20, #54, #84 #90, #16, #96		#20, #84, #90, #16, #96		#08, #14, #54
Outras	#12, #83, #95	#67	#12, #83, #95		#67

Fonte: Dados da pesquisa.

Tabela 11 – Procedimentos manuais realizados nos métodos semi-automatizados

Abordagem	Procedimentos manuais			
	Tradução de código	**Obtenção do modelo**	**Especificação das propriedades**	**Análise e interpretação dos resultados**
Análise estática	#69		#07	
Execução simbólica	#97		#07, #38, #97, #104	#38
Demonstração de teoremas		#03, #43	#38, #43	#38, #43
Model checking	#11, #22, #45, #59, #78	#11, #22, #41, #44, #45 #46, #58, #59, #78, #101 #103	#11, #22, #28, #41, #44 #45, #46, #58, #59, #78 #101, #103	#04, #41, #44, #45, #46 #78
Verificação dedutiva	#24, #99	#05, #24, #99	#05, #24, #99, #100	#09

Fonte: Dados da pesquisa.

também os estudos baseados em IA, já que, em alguns destes (#20, #54 e #90), os modelos utilizados para treinamento dos algoritmos de detecção são criados a partir dos resultados obtidos de ferramentas que apresentam altas taxas de falsos positivos e falsos negativos, o que pode ter influenciado a qualidade dos modelos. A ocorrência de falsos positivos e falsos negativos foi notada também em técnicas de análise dinâmica, análise simbólica e *fuzzing*.

As abordagens de demonstração de teoremas, *model checking* e verificação dedutiva destacam-se pela sua precisão, já que não produzem falsos positivos e negativos. Porém, estas baseiam-se na aplicação de métodos formais, e exigem algum conhecimento específico sobre técnicas de modelagem e verificação de propriedades, que geralmente não são dominadas por desenvolvedores.

3.1.3 Discussão

Plataformas blockchain baseadas na execução de CIs, como a Ethereum, têm ganhado notável destaque e popularidade nos últimos anos, e foram aplicadas em diversas áreas. Por outro lado, houve também ataques sobre diversos CIs nos quais usuários maliciosos de aproveitaram de vulnerabilidades de segurança presentes nos códigos dos CIs para transferir para a própria conta o Ether associado ao contrato. Estes ataques mobilizaram a academia e a indústria, e, nos últimos ano houve um aumento expressivo das pesquisas que buscam mitigar os riscos destas vulnerabilidades, que iniciaram-se em 2016. Assim, a verificação e detecção de vulnerabilidades em CIs é um tópico de pesquisa recente e diversas abordagens distintas têm sido empregadas. A identificação e classificação dessas abordagens foi o foco desse MS, que também apontou a frequência da utilização dessas abordagens ao longo dos anos, as vulnerabilidades e problemas alvos de verificação, as estratégias de validação empregadas e as principais limitações presentes nestes trabalhos.

Entre os 104 estudos selecionados, as abordagens de verificação mais utilizada, e que também foram as que mais prevaleceram ao longo dos anos foram as baseadas em execução simbólica, *model checking*, e análise estática. Foram classificados neste MS 25 problemas e vulnerabilidades alvos de verificação. Destes, nota-se que a Reentrância (V_{23}) foi a vulnerabilidade mais investigada pelos estudos, o que provavelmente deve-se ao fato desta ter sido a primeira a ser explorada, o que resultou no ataque ao contrato *The DAO* e mobilizou a comunidade em torno deste problema.

Uma forma mais genérica de verificar se a execução de um contrato é realizada conforme esperado é por meio da verificação da violação de propriedades (VP), as quais podem estar relacionadas com problemas de segurança, e também com o cumprimento de requisitos de projeto, funcionais, e lógicos. As propriedades são definidas especialmente para cada contrato, um trabalho manual que exige um alto nível de compreensão dos requisitos do contrato, assim como o conhecimento específico do formalismo ou linguagem de especificação utilizado. A violação de propriedades têm sido tratada principalmente pelas abordagens baseadas em *model checking* (em 15 estudos).

Outra questão analisada neste MS foram as estratégias da validação utilizada nos estudos (**QP 4**). Desta forma pôde-se observar desde estratégias simples e triviais como exemplos de aplicação e estudos de caso, até as mais robustas, compostas por vários procedimentos experimentais. Em muitos casos, devido a falta de uma validação mais completa e robusta, não foi possível tirar conclusões sobre questões como a eficiência e acurácia dos métodos propostos (como nos estudos #11, #15, #22, #23, #27, #58, #59, #60, #69, #70, #71, #75, #78, #82, #87, #95, #98, #100 e #103).

Ao longo dos resultados da **QP 5**, a abordagens foram classificadas em relação à abrangência da verificação e ao nível de automação da estratégia empregada, o que envolve não apenas

a verificação em si, mas também procedimentos aplicados antes e após. Apenas no *model checking*, na demonstração de teoremas, e na verificação dedutiva, os métodos propostos nos estudos foram majoritariamente aplicados apenas sobre um subconjunto da linguagem de programação de CIs considerada, o que indica necessidade de avanços nessa área para expandir a verificação sobre mais elementos dos CIs e abranger contratos mais complexos e variados. Também, nestas três abordagens, notou-se, na maior parte dos estudos, a aplicação de procedimentos manuais ao longo da estratégia de verificação. Entretanto, o fato de um método não ser totalmente automatizado não representa, necessariamente, uma desvantagem, pois depende do fim para o qual ele é proposto. Por exemplo, em muitos casos pode ser desejável que propriedades sejam definidas manualmente, pois cada contrato possui requisitos específicos relacionados ao seu projeto e seu propósito.

De forma geral, ainda não há uma convergência ou consenso sobre qual método de verificação é o melhor, pois ainda é uma área de estudo recente, e várias abordagens mostram-se promissoras, tantos aquelas utilizadas a mais tempo quantos a mais recentes, como as baseadas em IA. Contudo, ainda há diversas limitações presentes, que servirão de motivação das pesquisas e avanços futuros nesta área.

3.2 Seleção dos fundamentos da proposta

A realização do MS descrito na Seção 3.1 constituiu a primeira das quatro tarefas de pesquisa descritas no início deste Capítulo, e serve de base para a execução das próximas, que são: 2. Seleção do método de verificação e detecção de vulnerabilidades adequado para a proposta; 3. Seleção das vulnerabilidades abordadas por meio da proposta; e 4. Definição da estratégia para validação da proposta.

O método de verificação escolhido foi o *model checking*, pois há diversos trabalhos na literatura que utilizam esta técnica para verificação de vulnerabilidades de reentrância e para detecção de violação de propriedades. Desta forma, pretende-se focar em vulnerabilidades já conhecidas e também permitir a análise de propriedades funcionais de cada contrato individualmente. Além disso, das abordagens mais utilizadas, o *model checking* apresenta a melhor precisão, com raros casos de falsos positivos e falsos negativos (detectados apenas no estudo #26).

Das vulnerabilidades abordadas no MS, três foram selecionadas para serem o foco do método de verificação proposto, são elas: reentrância (V_{23}); *delegatecall injection* (V_{13}); e contrato suicida (V_8). A primeira foi explorada no *The DAO Attack*, e as outras duas nos ataques cometidos contra o contrato da *Parity Wallet*, ambos citados na Seção 2.3.1. Enquanto a V_8 e a V_{23} foram abordadas em alguns trabalhos que empregam o *model checking* (FRANK; ASCHERMANN; HOLZ, 2020; NELATURU *et al.*, 2020; WANG; YANG; LI, 2020; WANG; ZHANG; SU, 2019), a V_{13} ainda não foi tratada por meio desta técnica, o que seria um avanço

no estado da arte. Como estratégia de validação, pretende-se aplicar um experimento reduzido, com contratos cujas vulnerabilidades já são conhecidas, permitindo, assim, mensurar o nível de acurácia do método. Além disso, também deve ser aplicado um estudo de caso para verificação de violação de propriedades. Demais detalhes sobre o método proposto são descritos adiante no Capítulo 4.

CAPÍTULO

4

MÉTODO PARA VERIFICAÇÃO FORMAL DE CONTRATOS INTELIGENTES

No presente capítulo são descritos os detalhes do método para verificação de CIs proposto. Em seguida, é apresentada uma discussão acerca dos trabalhos relacionados com esta proposta. Por fim, é exposto o plano de trabalhos com as atividade planejadas ao longo deste curso.

4.1 Método proposto

Esta pesquisa propõe um método para verificação formal para detecção de vulnerabilidades em CIs escritos em Solidity na fase de pré-implementação. Especificamente, pretende-se detectar as vulnerabilidades de reentrância, *delegatecall injection* e contrato suicida. Para isso, foi escolhida a técnica de *model checking*, um método formal para verificação de sistemas de transição de estados que realiza uma pesquisa exaustiva sobre todo o espaço de estados do sistema para verificar se, conforme o modelo, o sistema age de acordo com propriedades predefinidas referentes à requerimentos de segurança e vivacidade do sistema.

Para isso, o código em Solidity deve ser convertido para algum formalismo baseado em estados que seja compatível com alguma ferramenta de verificação formal baseada em *model checking*, como NuSMV [1], nuXmv [2], SPIN [3], ou outra. Como as vulnerabilidades de reentrância e *delegatecall injection* acontecem a partir da interação entre contratos (i.e., o contrato explorado e o contrato atacante), então o modelo de estados utilizado deve permitir a modelagem de relações "intercontratuais".

No *model checking* as propriedades do sistema são especificadas por meio de fórmulas descritas em lógicas temporais LTL e CTL, ou alguma variação destas, como exposto nas Se-

[1] <https://nusmv.fbk.eu/>
[2] <https://nuxmv.fbk.eu/>
[3] <http://spinroot.com/>

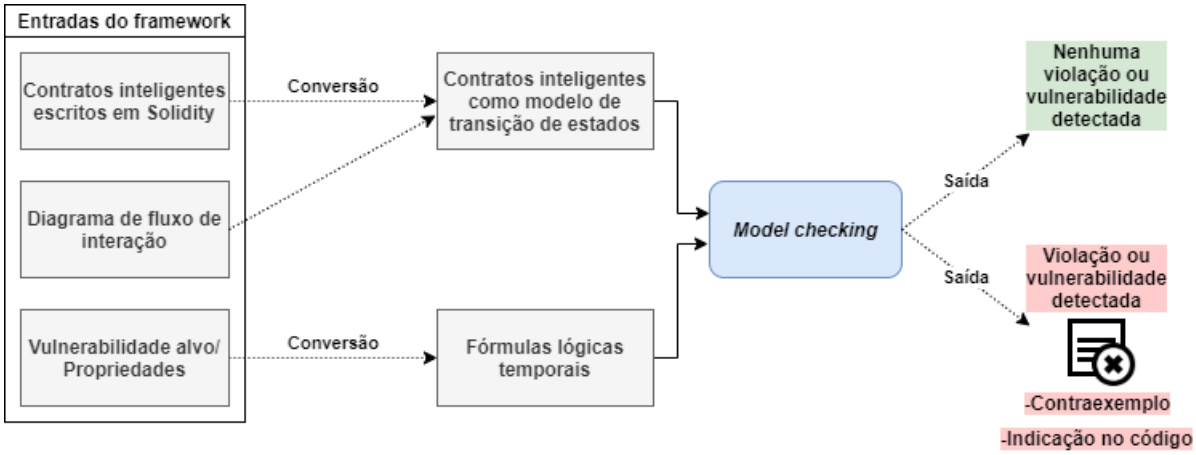
ções 2.4.2 e 2.4.6. Essas fórmulas são compostas por operadores temporais, quantificadores e elementos da lógica proposicional, técnicas que geralmente não são dominadas por desenvolvedores. Então, para facilitar a definição das propriedades, pretende-se desenvolver um modelo para descrição mais legível e mais próxima da linguagem natural, das propriedades que representem as vulnerabilidades abordadas. Vale destacar que, além das vulnerabilidades que são foco deste método de verificação, também será possível a inserção de qualquer propriedade relativa aos requisitos do CI, que ficará à critério do usuário. Este último ponto trata-se de algo que é intrínseco do *model checking*, e que já está inerentemente incluso nos *model checkers* existentes. Desta forma, também será possível a detecção de violações de propriedades funcionais.

No *model checking*, quando o modelo não satisfaz uma propriedade, então é detectada uma violação, e um contraexemplo é fornecido para descrever o caminho de execução até a violação. No caso das vulnerabilidades alvos desta pesquisa, a violação de uma propriedade indica a presença desta vulnerabilidade.

Para implementar a proposta, será desenvolvido um framework que integre as seguintes atividades: inclusão do código fonte em Solidity; conversão do código para o modelo de estados; exibição do modelo obtido; inserção das propriedades e seleção das vulnerabilidades a serem verificadas; conversão das propriedades e vulnerabilidades para a fórmula em lógica temporal; verificação sobre o modelo de estados do contrato por meio do *model checking*; e exibição dos resultados. Na inclusão do código fonte, mais de um arquivo pode ser inserido, já que o método proposto deve permitir a verificação considerando relações intercontratuais. Neste caso, devem ser indicadas manualmente pelo usuário as operações nas quais ocorre uma chamada externa à outro contrato. Desta forma, é obtido o diagrama de fluxo de interação entre os CIs, essencial para obtenção do modelo de transição de estados dos CIs. Uma vez fornecidos o código fonte e o diagrama de fluxo de interação dos CIs, é realizada a conversão automática para o modelo de transição de estados, que pode ser visualizado pelo usuário. Em seguida, as vulnerabilidades alvos da verificação são selecionadas, e, opcionalmente, outras propriedades também podem ser especificadas, e, adiante, são convertidas em fórmulas lógicas temporais. Enfim, o processo automático de *model checking* pode ser iniciado. Encerrada a verificação, os resultados para cada vulnerabilidade e propriedade são exibidos. Em caso de detecção de uma vulnerabilidade ou violação de alguma propriedade, um contraexemplo é fornecido, indicando o caminho de execução resultante para cada vulnerabilidade detectada ou propriedade violada, assim como o fragmento de código responsável pela vulnerabilidade ou violação. O fluxo de trabalho do framework é ilustrado na Figura 28. Nesta proposta, os únicos procedimento manuais necessários são a inserção das propriedades, e, quando for o caso, a descrição do diagrama de fluxo de interação dos contratos.

Como validação da proposta, será realizado experimento sobre um conjunto de contratos com vulnerabilidades já conhecidas, para, assim, poder avaliar a eficácia e acurácia da verificação, e outros aspectos como performance e a eficiência do framework proposto (i.e., consumo de

Figura 28 – Fluxo de trabalho do framework proposto.



Fonte: Elaborada pelo autor.

memória, processamento, e tempo de execução). Além disso, também deve ser aplicado ao menos um estudo de caso para tratar de propriedades funcionais relativas aos requisitos do CI.

4.2 Trabalhos relacionados

Esta seção tem o propósito de discutir a respeito das propostas encontradas na literatura para verificação de CIs baseadas em *model checking*. Para coleta dos trabalhos, foi aproveitado o levantamento bibliográfico feito no MS desenvolvido na Seção 3.1, no qual os estudos selecionados foram classificados de acordo com a abordagem de verificação empregada. Nesta seção, são discutidos alguns dos estudos que propõem a abordagem de *model checking*.

Os trabalhos de Mavridou *et al.* (2019) e Nelaturu *et al.* (2020) são os primeiros a serem discutidos, e são os que possuem maior semelhança com o método proposto na presente pesquisa. No primeiro, que é um dos mais citados sobre o assunto, foi desenvolvido o framework VeriSolid, que permite a modelagem gráfica de CIs por meio de uma máquina de estados finito (MEF) que estende a semântica operacional da linguagem Solidity utilizada no framework FSolidM (MAVRIDOU; LASZKA, 2018). Assim, o comportamento do contrato pode ser analisado com um alto nível de abstração, e então o código Solidity é gerado automaticamente a partir da MEF construída. Baseado na MEF obtida, pode-se definir propriedades de segurança e vivacidade para serem verificadas, porém, nenhuma vulnerabilidade específica foi abordada. Por meio da construção da MEF, diversos problemas são evitados, pois o código gerado do modelo é construído seguindo boas práticas de programação, como as recomendadas pela ConsenSys Diligence (CONSENSYS DILIGENCE, 2021). No trabalho de cite Nelaturu *et al.* (2020) é proposta uma extensão da VeriSolid, na qual os contratos em Solidity são automaticamente convertidos para uma MEF que é incrementada com um diagrama de implementação, o qual consiste em informações relacionadas à relação entre múltiplos contratos. Além disso, também é

acrescentado um módulo que realiza a implantação automática dos CIs gerados a partir da MEF. Nesse estudo, as vulnerabilidades de reentrância e de exceções não tratadas são prevenidas na própria construção da MEF, e as vulnerabilidades de endereço curto, contrato suicida e bloqueio de Ether são verificadas durante a modelagem ou na implantação do contrato.

O framework desenvolvido por Mavridou *et al.* (2019) e aperfeiçoado por Nelaturu *et al.* (2020) representa um dos maiores avanços na aplicação do *model checking* para verificação de CIs, pois não exige que o usuário possua experiência com métodos formais, tanto para modelagem, quanto para definição das propriedades, a qual é feita com descrições em linguagem natural. Uma das lacunas deixadas por Nelaturu *et al.* (2020) é que, para prevenir a reentrância, não é permitido que funções sejam chamadas externamente (e.g., por meio de uma função *fallback*) antes da finalização da transição atual, o que evita a recursividade de chamadas, porém, são desconsiderados casos em que uma função *fallback* é utilizada sem representar necessariamente uma vulnerabilidade. Na estratégia proposta por esta pesquisa, pretende-se tratar estes casos, e também incluir a verificação da vulnerabilidade *delegatecall injection*, explorada no ataque à *Parity Wallet*, mencionado na Seção 2.3.1.

Bai *et al.* (2018) também propõem a modelagem do CI como uma MEF, codificada diretamente em PROMELA, a linguagem de entrada do *model checker* SPIN [4]. Entretanto, não é realizada nenhuma equivalência com o código fonte em Solidity, que sequer é retratado na abordagem proposta. O principal problema disso é que, representar os elementos de um CI diretamente em alguma outra linguagem ou formalismo que não sejam próprios para programação de CIs, não assegura que as propriedades e vulnerabilidades verificadas serão evitadas na eventual escrita do código em Solidity (ou em outra linguagem de programação aceita na Ethereum). O mesmo problema também ocorre em Liu e Liu (2019), Madl *et al.* (2019), He (2020), Kongmanee, Kijsanayothin e Hewett (2019), Shishkin (2019) e Abdellatif e Brousmiche (2018). Na estratégia proposta nesta pesquisa, essa limitação é evitada, pois pressupõe que as vulnerabilidades e violações detectadas serão apontadas no código para correção.

No estudo de Wang, Zhang e Su (2019) é proposta a ferramenta de verificação NPChecker, cuja estratégia de modelagem e detecção de violação de propriedades nos contratos é feita com o objetivo de expor o não-determinismo da execução das transações na Ethereum como a raiz de vulnerabilidades relacionadas à *bugs* de pagamentos e transferências. Especificamente, são tratadas as vulnerabilidades de reentrância, dependência de ordem de transação e falha de chamada externa (também conhecida como chamada externa não verificada). Diferente do que é proposto na presente pesquisa, em Wang, Zhang e Su (2019) é empregada uma estratégia totalmente automatizada, na qual não é necessário analisar individualmente cada contrato, o que favorece a aplicação de experimentos em larga escala, porém, por outro lado, afeta a acurácia da verificação e introduz falsos positivos.

[4] <http://spinroot.com/>

4.2.1 Conclusão

Como exposto na Seção 3.1.2, o *model checking* é um dos métodos de verificação mais adotados em pesquisas recentes que abordam a verificação de CIs. Contudo, ainda há diversas limitações presentes na maior parte desses trabalhos. Um dos entraves para a adoção do *model checking* e outros métodos formais para verificação de CIs é a realização de procedimentos manuais que exigem conhecimento específico sobre os métodos formais empregados. Os trabalhos levantados nesta pesquisa que apresentam tal limitação são citados na Seção 3.1.2.

Ademais, poucos trabalhos consideram relações intercontratuais para modelagem e verificação. Neste sentido, o trabalho de Nelaturu *et al.* (2020) apresentou um dos maiores avanços nessa área, pois além de tratar as relações entre múltiplos contratos, também evita a necessidade de conhecimento sobre os métodos formais utilizados, já que os procedimentos de tradução do código fonte, obtenção do modelo, e análise e interpretação dos resultados são realizados automaticamente pelo framework. Além disso, apesar da especificação das propriedades ser feita manualmente, esse procedimento é simplificado por normas para especificação em linguagem natural.

Por meio do estratégia de verificação de CIs proposta neste trabalho, pretende-se usufruir dos avanços já realizados, e estender algumas funcionalidades como o tratamento especial das funções *fallback* para verificação da vulnerabilidade de reentrância, a detecção da vulnerabilidade *delegatecall injection* e a indicação do fragmento do código fonte que ocasionar o problema detectado.

4.3 Plano de trabalho

Na plano de trabalho exposto na Tabela 12, são expostas as atividades já desenvolvidas até então, e também as que serão realizadas até a defesa da dissertação para conclusão do mestrado.

Tabela 12 – Plano de trabalho das atividades do mestrado

	2020					2021						2022	
Atividade	Mar-Abr	Mai-Jun	Jul-Ago	Set-Out	Nov-Dez	Jan-Fev	Mar-Abr	Mai-Jun	Jul-Ago	Set-Out	Nov-Dez	Jan-Fev	Mar-Abr
Obtenção dos créditos obrigatórios	■	■	■	■	■								
Estudo da tecnologia blockchain			■	■	■								
Estudo da blockchain Ethereum e dos CIs			■	■	■	■	■	■	■				
Estudo dos métodos de verificação de CIs				■	■	■	■	■	■				
Realização do MS				■	■	■	■	■					
Preparação da qualificação						■	■	■					
Implementação do método proposto									■	■			
Avaliação experimental										■	■		
Redação da dissertação										■	■	■	
Defesa do mestrado													■

REFERÊNCIAS

ABDELLATIF, T.; BROUSMICHE, K.-L. Formal verification of smart contracts based on users and blockchain behaviors models. In: IEEE. **2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)**. [S.l.], 2018. p. 1–5. Citado nas páginas 49, 61 e 80.

AGUIAR, E. J. D.; FAIÇAL, B. S.; KRISHNAMACHARI, B.; UEYAMA, J. A survey of blockchain-based strategies for healthcare. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 53, n. 2, p. 1–27, 2020. Citado nas páginas 17, 23 e 37.

AHMED, M.; ELAHI, I.; ABRAR, M.; ASLAM, U.; KHALID, I.; HABIB, M. A. Understanding blockchain: Platforms, applications and implementation challenges. In: **Proceedings of the 3rd International Conference on Future Networks and Distributed Systems**. New York, NY, USA: Association for Computing Machinery, 2019. (ICFNDS '19). ISBN 9781450371636. Disponível em: <https://doi.org/10.1145/3341325.3342033>. Citado nas páginas 26, 28 e 29.

AHRENDT, W.; BECKERT, B.; BUBEL, R.; HÄHNLE, R.; SCHMITT, P. H.; ULBRICH, M. Deductive software verification–the key book. **Lecture Notes in Computer Science**, Springer, v. 10001, 2016. Citado na página 50.

AHRENDT, W.; BUBEL, R.; ELLUL, J.; PACE, G. J.; PARDO, R.; REBISCOUL, V.; SCHNEIDER, G. Verification of smart contract business logic. In: SPRINGER. **International Conference on Fundamentals of Software Engineering**. [S.l.], 2019. p. 228–243. Citado na página 61.

AKCA, S.; RAJAN, A.; PENG, C. Solanalyser: A framework for analysing and testing smart contracts. In: IEEE. **2019 26th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.], 2019. p. 482–489. Citado na página 61.

ALBERT, E.; CORREAS, J.; GORDILLO, P.; ROMÁN-DÍEZ, G.; RUBIO, A. Safevm: a safety verifier for ethereum smart contracts. In: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2019. p. 386–389. Citado nas páginas 47 e 61.

____. Don't run on fumes—parametric gas bounds for smart contracts. **Journal of Systems and Software**, Elsevier, v. 176, p. 110923, 2021. Citado nas páginas 47 e 61.

ALMAKHOUR, M.; SLIMAN, L.; SAMHAT, A. E.; MELLOUK, A. Verification of smart contracts: A survey. **Pervasive and Mobile Computing**, Elsevier, p. 101227, 2020. Citado nas páginas 19, 45, 48, 49 e 50.

ALT, L.; REITWIESSNER, C. Smt-based verification of solidity smart contracts. In: SPRINGER. **International Symposium on Leveraging Applications of Formal Methods**. [S.l.], 2018. p. 376–388. Citado nas páginas 19 e 61.

AMANI, S.; BÉGEL, M.; BORTIN, M.; STAPLES, M. Towards verifying ethereum smart contract bytecode in isabelle/hol. In: **Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs**. [S.l.: s.n.], 2018. p. 66–77. Citado na página 61.

ANDROULAKI, E.; BARGER, A.; BORTNIKOV, V.; CACHIN, C.; CHRISTIDIS, K.; CARO, A. D.; ENYEART, D.; FERRIS, C.; LAVENTMAN, G.; MANEVICH, Y.; MURALIDHARAN, S.; MURTHY, C.; NGUYEN, B.; SETHI, M.; SINGH, G.; SMITH, K.; SORNIOTTI, A.; STATHAKOPOULOU, C.; VUKOLIć, M.; COCCO, S. W.; YELLICK, J. Hyperledger fabric: A distributed operating system for permissioned blockchains. In: **Proceedings of the Thirteenth EuroSys Conference**. New York, NY, USA: Association for Computing Machinery, 2018. (EuroSys '18). ISBN 9781450355841. Disponível em: <https://doi.org/10.1145/3190508.3190538>. Citado na página 26.

ANTONOPOULOS, A. M. **Mastering Bitcoin: unlocking digital cryptocurrencies**. [S.l.]: "O'Reilly Media, Inc.", 2014. Citado na página 24.

ARGAÑARAZ, M.; BERÓN, M.; PEREIRA, M. J.; HENRIQUES, P. Detection of vulnerabilities in smart contracts specifications in ethereum platforms. In: SCHLOSS DAGSTUHL–LEIBNIZ-ZENTRUM FUER INFORMATIK. **9th Symposium on Languages, Applications and Technologies (SLATE 2020)**. [S.l.], 2020. v. 83, p. 1–16. Citado na página 61.

ASHOURI, M. Etherolic: a practical security analyzer for smart contracts. In: **Proceedings of the 35th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2020. p. 353–356. Citado na página 61.

ASHRAF, I.; MA, X.; JIANG, B.; CHAN, W. Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. **IEEE Access**, IEEE, v. 8, p. 99552–99564, 2020. Citado nas páginas 46 e 61.

ATZEI, N.; BARTOLETTI, M.; CIMOLI, T. A survey of attacks on ethereum smart contracts (sok). In: SPRINGER. **International conference on principles of security and trust**. [S.l.], 2017. p. 164–186. Citado nas páginas 18, 41, 42 e 45.

ATZEI, N.; BARTOLETTI, M.; LANDE, S.; YOSHIDA, N.; ZUNINO, R. Developing secure bitcoin contracts with bitml. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2019. p. 1124–1128. Citado na página 61.

AZZOPARDI, S.; ELLUL, J.; PACE, G. J. Monitoring smart contracts: Contractlarva and open challenges beyond. In: SPRINGER. **International Conference on Runtime Verification**. [S.l.], 2018. p. 113–137. Citado nas páginas 51 e 61.

BAI, X.; CHENG, Z.; DUAN, Z.; HU, K. Formal modeling and verification of smart contracts. In: **Proceedings of the 2018 7th International Conference on Software and Computer Applications**. [S.l.: s.n.], 2018. p. 322–326. Citado nas páginas 49, 61 e 80.

BAIER, C.; KATOEN, J. **Principles of Model Checking**. [S.l.]: MIT Press, 2008. ISBN 9780262026499. Citado na página 55.

BEILLAHI, S. M.; CIOCARLIE, G.; EMMI, M.; ENEA, C. Behavioral simulation for smart contracts. In: **Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2020. p. 470–486. Citado nas páginas 50 e 61.

BERTONI, G.; DAEMEN, J.; PEETERS, M.; ASSCHE, G. **The keccak SHA-3 submission. Submission to NIST (Round 3)**. 2011. Citado na página 22.

BHARGAVAN, K.; DELIGNAT-LAVAUD, A.; FOURNET, C.; GOLLAMUDI, A.; GONTHIER, G.; KOBEISSI, N.; KULATOVA, N.; RASTOGI, A.; SIBUT-PINOTE, T.; SWAMY, N.; ZANELLA-BÉGUELIN, S. Formal verification of smart contracts: Short paper. In: **Proceedings of the 2016 ACM workshop on programming languages and analysis for security**. [S.l.: s.n.], 2016. p. 91–96. Citado na página 61.

BOURAGA, S. A taxonomy of blockchain consensus protocols: A survey and classification framework. **Expert Systems with Applications**, Elsevier Ltd, v. 168, 2021. Citado nas páginas 25 e 26.

BRENT, L.; GRECH, N.; LAGOUVARDOS, S.; SCHOLZ, B.; SMARAGDAKIS, Y. Ethainter: a smart contract security analyzer for composite vulnerabilities. In: **Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2020. p. 454–469. Citado na página 61.

BUTERIN, V. [S.l.], 2014. Acesso em: 29 jan. 2021. Disponível em: <https://ethereum.org/en/whitepaper/>. Citado nas páginas 17, 22, 25, 29, 30, 32 e 35.

CASCARILLA, C. **Pax Gold white paper**. 2019. Acesso em: 23 fev. 2021. Disponível em: <https://www.paxos.com/pax-gold-whitepaper>. Citado na página 36.

CASTRO, M.; LISKOV, B. Practical byzantine fault tolerance. In: **OSDI**. [S.l.: s.n.], 1999. v. 99, n. 1999, p. 173–186. Citado na página 26.

CHANG, J.; GAO, B.; XIAO, H.; SUN, J.; CAI, Y.; YANG, Z. Scompile: Critical path identification and analysis for smart contracts. In: SPRINGER. **International Conference on Formal Engineering Methods**. [S.l.], 2019. p. 286–304. Citado nas páginas 47 e 61.

CHATTERJEE, K.; GOHARSHADY, A. K.; VELNER, Y. Quantitative analysis of smart contracts. In: SPRINGER, CHAM. **European Symposium on Programming**. [S.l.], 2018. p. 739–767. Citado na página 61.

CHEN, H.; PENDLETON, M.; NJILLA, L.; XU, S. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 3, jun. 2020. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3391195>. Citado nas páginas 18, 19, 31, 32, 38, 41, 42, 43, 44, 45, 46, 47 e 64.

CHEN, T.; FENG, Y.; LI, Z.; ZHOU, H.; LUO, X.; LI, X.; XIAO, X.; CHEN, J.; ZHANG, X. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. **IEEE Transactions on Emerging Topics in Computing**, IEEE, 2020. Citado nas páginas 47 e 61.

CHEN, T.; LI, Z.; ZHOU, H.; CHEN, J.; LUO, X.; LI, X.; ZHANG, X. Towards saving money in using smart contracts. In: IEEE. **2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)**. [S.l.], 2018. p. 81–84. Citado na página 61.

CHEN, T.; LI, Z.; ZHU, Y.; CHEN, J.; LUO, X.; LUI, J. C.-S.; LIN, X.; ZHANG, X. Understanding ethereum via graph analysis. **ACM Transactions on Internet Technology**, Association for Computing Machinery, v. 20, n. 2, 2020. ISSN 15576051. Citado nas páginas 29 e 32.

CHINEN, Y.; YANAI, N.; CRUZ, J. P.; OKAMURA, S. Ra: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis. In: IEEE. **2020 IEEE International Conference on Blockchain (Blockchain)**. [S.l.], 2020. p. 327–336. Citado na página 61.

CLARKE JR, E. M.; GRUMBERG, O.; KROENING, D.; PELED, D.; VEITH, H. **Model checking**. [S.l.]: MIT press, 2018. Citado nas páginas 20, 48, 51 e 52.

CONSENSYS DILIGENCE. **Ethereum Smart Contract Best Practices**. 2021. Acesso em: 20 fev. 2021. Disponível em: <https://consensys.github.io/smart-contract-best-practices/>. Citado nas páginas 43 e 79.

DESTEFANIS, G.; MARCHESI, M.; ORTU, M.; TONELLI, R.; BRACCIALI, A.; HIERONS, R. Smart contracts vulnerabilities: a call for blockchain software engineering? In: IEEE. **2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)**. [S.l.], 2018. p. 19–25. Citado nas páginas 18 e 45.

DI ANGELO, M.; SALZER, G. Tokens, types, and standards: Identification and utilization in ethereum. In: **2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)**. [S.l.: s.n.], 2020. p. 1–10. Citado na página 36.

DIKA, A.; NOWOSTAWSKI, M. Security vulnerabilities in ethereum smart contracts. 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. 955–962. Citado nas páginas 18, 19 e 46.

DING, Y.; WANG, C.; ZHONG, Q.; LI, H.; TAN, J.; LI, J. Function-level dynamic monitoring and analysis system for smart contract. **IEEE Access**, IEEE, 2020. Citado na página 61.

DINH, T. T. A.; LIU, R.; ZHANG, M.; CHEN, G.; OOI, B. C.; WANG, J. Untangling blockchain: A data processing view of blockchain systems. **IEEE Transactions on Knowledge and Data Engineering**, v. 30, n. 7, p. 1366–1385, 2018. Citado nas páginas 17, 23 e 26.

DRESCHER, D. **Blockchain basics: a non-technical introduction in 25 steps**. [S.l.]: Apress, 2017. Citado nas páginas 17, 22, 24, 25, 27, 28 e 29.

DU, S.; HUANG, H. A general framework of smart contract vulnerability mining based on control flow graph matching. In: SPRINGER. **International Conference on Artificial Intelligence and Security**. [S.l.], 2020. p. 166–175. Citado na página 61.

DUO, W.; XIN, H.; XIAOFENG, M. Formal analysis of smart contract based on colored petri nets. **IEEE Intelligent Systems**, IEEE, v. 35, n. 3, p. 19–30, 2020. Citado na página 61.

DWORKIN, M. J. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015. Citado na página 22.

EKBLAW, A.; AZARIA, A.; HALAMKA, J. D.; LIPPMAN, A. A case study for blockchain in healthcare:"medrec" prototype for electronic health records and medical research data. In: **Proceedings of IEEE open & big data conference**. [S.l.: s.n.], 2016. v. 13, p. 13. Citado na página 37.

ETHEREUM COMMUNITY. **Ethereum Homestead Documentation**. 2018. Acesso em: 20 fev. 2021. Disponível em: <https://ethdocs.org/en/latest/>. Citado na página 31.

FAN, C.; GHAEMI, S.; KHAZAEI, H.; MUSILEK, P. Performance evaluation of blockchain systems: A systematic survey. **IEEE Access**, IEEE, v. 8, p. 126927–126950, 2020. Citado na página 17.

FEIST, J.; GRIECO, G.; GROCE, A. Slither: a static analysis framework for smart contracts. In: IEEE. **2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)**. [S.l.], 2019. p. 8–15. Citado na página 61.

FENG, Y.; TORLAK, E.; BODIK, R. Summary-based symbolic evaluation for smart contracts. In: IEEE. **2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2020. p. 1141–1152. Citado na página 61.

FRANK, J.; ASCHERMANN, C.; HOLZ, T. Ethbmc: A bounded model checker for smart contracts. In: **29th USENIX Security Symposium (USENIX Security 20)**. [S.l.: s.n.], 2020. p. 2757–2774. Citado nas páginas 61 e 75.

FU, M.; WU, L.; HONG, Z.; ZHU, F.; SUN, H.; FENG, W. A critical-path-coverage-based vulnerability detection method for smart contracts. **IEEE Access**, IEEE, v. 7, p. 147327–147344, 2019. Citado nas páginas 46 e 61.

GAO, J.; LIU, H.; LIU, C.; LI, Q.; GUAN, Z.; CHEN, Z. Easyflow: Keep ethereum away from overflow. In: IEEE. **2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)**. [S.l.], 2019. p. 23–26. Citado na página 61.

GAO, Z.; JIANG, L.; XIA, X.; LO, D.; GRUNDY, J. Checking smart contracts with structural code embedding. **IEEE Transactions on Software Engineering**, IEEE, 2020. Citado nas páginas 50 e 61.

GRECH, N.; KONG, M.; JURISEVIC, A.; BRENT, L.; SCHOLZ, B.; SMARAGDAKIS, Y. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 2, n. OOPSLA, p. 1–27, 2018. Citado na página 61.

HAJDU, Á.; JOVANOVIĆ, D. Solc-verify: A modular verifier for solidity smart contracts. In: SPRINGER. **Working Conference on Verified Software: Theories, Tools, and Experiments**. [S.l.], 2019. p. 161–179. Citado nas páginas 47 e 61.

HAO, X.; REN, W.; ZHENG, W.; ZHU, T. Scscan: A svm-based scanning system for vulnerabilities in blockchain smart contracts. In: IEEE. **2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)**. [S.l.], 2020. p. 1598–1605. Citado na página 61.

HARRISON, J. Theorem proving for verification (invited tutorial). In: SPRINGER. **International Conference on Computer Aided Verification**. [S.l.], 2008. p. 11–18. Citado na página 49.

HE, X. Modeling and analyzing smart contracts using predicate transition nets. In: IEEE. **2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)**. [S.l.], 2020. p. 108–115. Citado nas páginas 61 e 80.

HEWA, T.; YLIANTTILA, M.; LIYANAGE, M. Survey on blockchain based smart contracts: Applications, opportunities and challenges. **Journal of Network and Computer Applications**, v. 177, 2021. Citado na página 30.

HIRAI, Y. Defining the ethereum virtual machine for interactive theorem provers. In: SPRINGER. **International Conference on Financial Cryptography and Data Security**. [S.l.], 2017. p. 520–535. Citado na página 61.

HORTA, L. P. A. da; REIS, J. S.; SOUSA, S. M. de; PEREIRA, M. A tool for proving michelson smart contracts in why3. In: IEEE. **2020 IEEE International Conference on Blockchain (Blockchain)**. [S.l.], 2020. p. 409–414. Citado na página 61.

HUANG, J.; HAN, S.; YOU, W.; SHI, W.; LIANG, B.; WU, J.; WU, Y. Hunting vulnerable smart contracts via graph embedding based bytecode matching. **IEEE Transactions on Information Forensics and Security**, IEEE, v. 16, p. 2144–2156, 2021. Citado nas páginas 46 e 61.

HUTH, M.; RYAN, M. **Logic in Computer Science: Modelling and Reasoning about Systems**. [S.l.]: Cambridge University Press, 2004. ISBN 9781139453059. Citado nas páginas 52, 53 e 54.

JI, R.; HE, N.; WU, L.; WANG, H.; BAI, G.; GUO, Y. Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts. **arXiv preprint arXiv:2006.06419**, 2020. Citado na página 61.

JIANG, B.; LIU, Y.; CHAN, W. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: IEEE. **2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2018. p. 259–269. Citado nas páginas 46 e 61.

KANNENGIESSER, N.; LINS, S.; DEHLING, T.; SUNYAEV, A. Trade-offs between distributed ledger technology characteristics. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 53, n. 2, p. 1–37, 2020. Citado nas páginas 17 e 19.

KATOEN, J.-P. **Concepts, algorithms, and tools for model checking**. [S.l.]: IMMD Erlangen, 1999. Citado na página 55.

KIM, K. B.; LEE, J. Automated generation of test cases for smart contract security analyzers. **IEEE Access**, IEEE, v. 8, p. 209377–209392, 2020. Citado nas páginas 19 e 64.

KING, J. C. Symbolic execution and program testing. **Communications of the ACM**, ACM New York, NY, USA, v. 19, n. 7, p. 385–394, 1976. Citado na página 47.

KING, S.; NADAL, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012. Acesso em: 05 fev. 2021. Disponível em: <https://www.peercoin.net/whitepapers/peercoin-paper.pdf>. Citado na página 26.

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering - version 2.3. Citeseer, 2007. Citado nas páginas 57 e 58.

KLEES, G.; RUEF, A.; COOPER, B.; WEI, S.; HICKS, M. Evaluating fuzz testing. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.: s.n.], 2018. p. 2123–2138. Citado na página 50.

KOBLITZ, N. Elliptic curve cryptosystems. **Mathematics of computation**, v. 48, n. 177, p. 203–209, 1987. Citado na página 27.

KOLLURI, A.; NIKOLIC, I.; SERGEY, I.; HOBOR, A.; SAXENA, P. Exploiting the laws of order in smart contracts. In: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2019. p. 363–373. Citado na página 61.

KONGMANEE, J.; KIJSANAYOTHIN, P.; HEWETT, R. Securing smart contracts in blockchain. In: IEEE. **2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)**. [S.l.], 2019. p. 69–76. Citado nas páginas 61 e 80.

KRUPP, J.; ROSSOW, C. Teether: Gnawing at ethereum to automatically exploit smart contracts. In: **27th {USENIX} Security Symposium ({USENIX} Security 18)**. [S.l.: s.n.], 2018. p. 1317–1333. Citado na página 61.

LAHBIB, A.; WAKRIME, A. A.; LAOUITI, A.; TOUMI, K.; MARTIN, S. An event-b based approach for formal modelling and verification of smart contracts. In: SPRINGER. **International Conference on Advanced Information Networking and Applications**. [S.l.], 2020. p. 1303–1318. Citado na página 61.

LAI, E.; LUO, W. Static analysis of integer overflow of smart contracts in ethereum. In: **Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy**. [S.l.: s.n.], 2020. p. 110–115. Citado na página 61.

LI, A.; CHOI, J. A.; LONG, F. Securing smart contract with runtime validation. In: **Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2020. p. 438–453. Citado nas páginas 51 e 61.

LI, X.; SU, C.; XIONG, Y.; HUANG, W.; WANG, W. Formal verification of bnb smart contract. In: IEEE. **2019 5th International Conference on Big Data Computing and Communications (BIGCOM)**. [S.l.], 2019. p. 74–78. Citado nas páginas 49 e 61.

LI, Y.; LIU, H.; YANG, Z.; REN, Q.; WANG, L.; CHEN, B. Safepay on ethereum: A framework for detecting unfair payments in smart contracts. In: IEEE. **2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)**. [S.l.], 2020. p. 1219–1222. Citado na página 61.

LI, Z.; GUO, W.; XU, Q.; XU, Y.; WANG, H.; XIAN, M. Research on blockchain smart contracts vulnerability and a code audit tool based on matching rules. In: **Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies**. [S.l.: s.n.], 2020. p. 484–489. Citado na página 61.

LIAO, J.-W.; TSAI, T.-T.; HE, C.-K.; TIEN, C.-W. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In: IEEE. **2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)**. [S.l.], 2019. p. 458–465. Citado na página 61.

LIU, C.; LIU, H.; CAO, Z.; CHEN, Z.; CHEN, B.; ROSCOE, B. Reguard: finding reentrancy bugs in smart contracts. In: IEEE. **2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)**. [S.l.], 2018. p. 65–68. Citado na página 61.

LIU, J.; LIU, Z. A survey on security verification of blockchain smart contracts. **IEEE Access**, IEEE, v. 7, p. 77894–77904, 2019. Citado nas páginas 18, 19, 41 e 45.

LIU, Y.; LI, Y.; LIN, S.-W.; ZHAO, R. Towards automated verification of smart contract fairness. In: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2020. p. 666–677. Citado na página 61.

LIU, Z.; LIU, J. Formal verification of blockchain smart contract based on colored petri net models. In: IEEE. **2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)**. [S.l.], 2019. v. 2, p. 555–560. Citado nas páginas 49, 61 e 80.

LU, N.; WANG, B.; ZHANG, Y.; SHI, W.; ESPOSITO, C. Neucheck: A more practical ethereum smart contract security analysis tool. **Software: Practice and Experience**, Wiley Online Library, 2019. Citado nas páginas 46, 47 e 61.

LUU, L.; CHU, D.-H.; OLICKEL, H.; SAXENA, P.; HOBOR, A. Making smart contracts smarter. In: **Proceedings of the 2016 ACM SIGSAC conference on computer and communications security**. [S.l.: s.n.], 2016. p. 254–269. Citado nas páginas 47 e 61.

MADL, G.; BATHEN, L.; FLORES, G.; JADAV, D. Formal verification of smart contracts using interface automata. In: IEEE. **2019 IEEE International Conference on Blockchain (Blockchain)**. [S.l.], 2019. p. 556–563. Citado nas páginas 61 e 80.

MAESA, D. D. F.; MORI, P. Blockchain 3.0 applications survey. **Journal of Parallel and Distributed Computing**, Elsevier, v. 138, p. 99–114, 2020. Citado nas páginas 17, 22, 35 e 36.

MANNING, A. **Comprehensive list of known attack vectors and common anti-patterns**. 2018. Acesso em: 17 mai. 2021. Disponível em: <https://github.com/sigp/solidity-security-blog>. Citado nas páginas 18 e 44.

MARESCOTTI, M.; BLICHA, M.; HYVÄRINEN, A. E.; ASADI, S.; SHARYGINA, N. Computing exact worst-case gas consumption for smart contracts. In: SPRINGER. **International Symposium on Leveraging Applications of Formal Methods**. [S.l.], 2018. p. 450–465. Citado na página 61.

MARESCOTTI, M.; OTONI, R.; ALT, L.; EUGSTER, P.; HYVÄRINEN, A. E.; SHARYGINA, N. Accurate smart contract verification through direct modelling. In: SPRINGER. **International Symposium on Leveraging Applications of Formal Methods**. [S.l.], 2020. p. 178–194. Citado nas páginas 49 e 61.

MARION, J. **Contabilidade basica**. [S.l.]: Atlas, 1985. ISBN 9788547210236. Citado na página 22.

MAVRIDOU, A.; LASZKA, A. Tool demonstration: Fsolidm for designing secure ethereum smart contracts. In: SPRINGER. **International conference on principles of security and trust**. [S.l.], 2018. p. 270–277. Citado nas páginas 61 e 79.

MAVRIDOU, A.; LASZKA, A.; STACHTIARI, E.; DUBEY, A. Verisolid: Correct-by-design smart contracts for ethereum. In: SPRINGER. **International Conference on Financial Cryptography and Data Security**. [S.l.], 2019. p. 446–465. Citado nas páginas 61, 79 e 80.

MAZIERES, D. The stellar consensus protocol: A federated model for internet-level consensus. **Stellar Development Foundation**, Citeseer, v. 32, 2015. Disponível em: <https://www.stellar.org/papers/stellar-consensus-protocol>. Citado na página 26.

MCGHIN, T.; CHOO, K.-K. R.; LIU, C. Z.; HE, D. Blockchain in healthcare applications: Research challenges and opportunities. **Journal of Network and Computer Applications**, Elsevier, v. 135, p. 62–75, 2019. Citado na página 37.

MOMENI, P.; WANG, Y.; SAMAVI, R. Machine learning model for smart contracts security analysis. In: IEEE. **2019 17th International Conference on Privacy, Security and Trust (PST)**. [S.l.], 2019. p. 1–6. Citado nas páginas 51 e 61.

MONRAT, A. A.; SCHELéN, O.; ANDERSSON, K. A survey of blockchain from the perspectives of applications, challenges, and opportunities. **IEEE Access**, v. 7, p. 117134–117151, 2019. Citado nas páginas 21 e 27.

MOSSBERG, M.; MANZANO, F.; HENNENFENT, E.; GROCE, A.; GRIECO, G.; FEIST, J.; BRUNSON, T.; DINABURG, A. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: IEEE. **2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2019. p. 1186–1189. Citado na página 61.

MULLER-OLM, M.; SCHMIDT, D.; STEFFEN, B. Invited talks and tutorials-model-checking. a tutorial introduction. **Lecture Notes in Computer Science**, Berlin: Springer-Verlag, 1973-, v. 1694, p. 330–354, 1999. Citado nas páginas 52 e 53.

MURA, W. A. D. **Detecção de conflitos em contratos multilaterais**. 133 p. Dissertação (Mestrado) — Universidade Estadual de Londrina, Londrina-PR, 2016. Citado nas páginas 53, 55 e 56.

NAKAGAWA, E. Y.; SCANNAVINO, K. R. F.; FABBRI, S. C. P. F.; FERRARI, F. C. Revisão sistemática da literatura em engenharia de software: teoria e prática. Elsevier Brasil, 2017. Citado nas páginas 57, 58 e 59.

NAKAMOTO, S. **Bitcoin: A peer-to-peer electronic cash system**. [S.l.], 2008. Acesso em: 29 jan. 2021. Disponível em: <https://bitcoin.org/bitcoin.pdf>. Citado nas páginas 17, 21, 22, 24, 25 e 27.

NEHAÏ, Z.; BOBOT, F. Deductive proof of industrial smart contracts using why3. In: SPRINGER. **International Symposium on Formal Methods**. [S.l.], 2019. p. 299–311. Citado na página 61.

NEHAI, Z.; PIRIOU, P.-Y.; DAUMAS, F. Model-checking of smart contracts. 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. 980–987. Citado nas páginas 48 e 61.

NELATURU, K.; MAVRIDOUL, A.; VENERIS, A.; LASZKA, A. Verified development and deployment of multiple interacting smart contracts with verisolid. In: IEEE. **2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)**. [S.l.], 2020. p. 1–9. Citado nas páginas 19, 61, 75, 79, 80 e 81.

NIKOLIĆ, I.; KOLLURI, A.; SERGEY, I.; SAXENA, P.; HOBOR, A. Finding the greedy, prodigal, and suicidal contracts at scale. In: **Proceedings of the 34th Annual Computer Security Applications Conference**. [S.l.: s.n.], 2018. p. 653–663. Citado na página 61.

OAPOS;DWYER, K. Bitcoin mining and its energy footprint. **IET Conference Proceedings**, Institution of Engineering and Technology, p. 280–285(5), January 2014. Citado na página 26.

OSTERLAND, T.; ROSE, T. Model checking smart contracts for ethereum. **Pervasive and Mobile Computing**, Elsevier, v. 63, p. 101129, 2020. Citado nas páginas 49 e 61.

PARK, D.; ZHANG, Y.; SAXENA, M.; DAIAN, P.; ROŞU, G. A formal verification tool for Ethereum VM bytecode. In: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2018. p. 912–915. Citado nas páginas 50 e 61.

PELED, D. A. Formal methods. In: **Handbook of Software Engineering**. [S.l.]: Springer, 2019. p. 193–222. Citado nas páginas 20, 48 e 49.

PENG, C.; AKCA, S.; RAJAN, A. Sif: A framework for solidity contract instrumentation and analysis. In: IEEE. **2019 26th Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.], 2019. p. 466–473. Citado na página 61.

PERMENEV, A.; DIMITROV, D.; TSANKOV, P.; DRACHSLER-COHEN, D.; VECHEV, M. Verx: Safety verification of smart contracts. In: IEEE. **2020 IEEE Symposium on Security and Privacy (SP)**. [S.l.], 2020. p. 1661–1677. Citado na página 61.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: **12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12**. [S.l.: s.n.], 2008. p. 1–10. Citado na página 62.

PRECHTEL, D.; GROSS, T.; MÜLLER, T. Evaluating spread of 'gasless send'in ethereum smart contracts. In: IEEE. **2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)**. [S.l.], 2019. p. 1–6. Citado na página 61.

QIAN, P.; LIU, Z.; HE, Q.; ZIMMERMANN, R.; WANG, X. Towards automated reentrancy detection for smart contracts based on sequential models. **IEEE Access**, IEEE, v. 8, p. 19685–19695, 2020. Citado nas páginas 50 e 61.

QU, M.; HUANG, X.; CHEN, X.; WANG, Y.; MA, X.; LIU, D. Formal verification of smart contracts from the perspective of concurrency. In: SPRINGER. **International Conference on Smart Blockchain**. [S.l.], 2018. p. 32–43. Citado na página 61.

SALAH, K.; REHMAN, M. H. U.; NIZAMUDDIN, N.; AL-FUQAHA, A. Blockchain for AI: Review and open research challenges. **IEEE Access**, v. 7, p. 10127–10149, 2019. Citado na página 17.

SAMREEN, N. F.; ALALFI, M. H. Reentrancy vulnerability identification in ethereum smart contracts. In: IEEE. **2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)**. [S.l.], 2020. p. 22–29. Citado na página 61.

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 25.

SAYEED, S.; MARCO-GISBERT, H.; CAIRA, T. Smart contract: Attacks and protections. **IEEE Access**, IEEE, v. 8, p. 24416–24427, 2020. Citado nas páginas 19 e 42.

SCHNEIDEWIND, C.; GRISHCHENKO, I.; SCHERER, M.; MAFFEI, M. ethor: Practical and provably sound static analysis of ethereum smart contracts. In: **Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.: s.n.], 2020. p. 621–640. Citado na página 61.

SCHWARTZ, D.; YOUNGS, N.; BRITTO, A. The ripple protocol consensus algorithm. **Ripple Labs Inc White Paper**, v. 5, n. 8, p. 151, 2014. Disponível em: <https://cryptoguide.ch/cryptocurrency/ripple/whitepaper.pdf>. Citado na página 26.

SHISHKIN, E. Debugging smart contract's business logic using symbolic model checking. **Programming and Computer Software**, Springer, v. 45, n. 8, p. 590–599, 2019. Citado nas páginas 61 e 80.

SIEGEL, D. **Understanding The DAO Attack**. 2020. Acesso em: 24 fev. 2021. Disponível em: <https://www.coindesk.com/understanding-dao-hack-journalists>. Citado nas páginas 18 e 41.

SINGH, A.; PARIZI, R. M.; ZHANG, Q.; CHOO, K.-K. R.; DEHGHANTANHA, A. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. **Computers & Security**, Elsevier, v. 88, p. 101654, 2020. Citado nas páginas 19, 45, 46 e 49.

SO, S.; LEE, M.; PARK, J.; LEE, H.; OH, H. Verismart: A highly precise safety verifier for ethereum smart contracts. In: IEEE. **2020 IEEE Symposium on Security and Privacy (SP)**. [S.l.], 2020. p. 1678–1694. Citado na página 61.

SOMPOLINSKY, Y.; ZOHAR, A. Secure high-rate transaction processing in bitcoin. In: SPRINGER. **International Conference on Financial Cryptography and Data Security**. [S.l.], 2015. p. 507–527. Citado nas páginas 27 e 35.

SUICHE, M. **The $280M Ethereum's Parity bug**. 2017. Acesso em: 17 mai. 2021. Disponível em: <https://blog.comae.io/the-280m-ethereums-bug-f28e5de43513>. Citado na página 45.

SUN, T.; YU, W. A formal verification framework for security issues of blockchain smart contracts. **Electronics**, Multidisciplinary Digital Publishing Institute, v. 9, n. 2, p. 255, 2020. Citado nas páginas 49 e 61.

SUN, Y.; GU, L. Attention-based machine learning model for smart contract vulnerability detection. In: IOP PUBLISHING. **Journal of Physics: Conference Series**. [S.l.], 2021. v. 1820, n. 1, p. 012004. Citado nas páginas 50 e 61.

SWAN, M. **Blockchain: Blueprint for a New Economy**. [S.l.]: O'Reilly Media, 2015. ISBN 9781491920473. Citado nas páginas 17, 21, 24 e 35.

SZABO, N. Formalizing and securing relationships on public networks. **First Monday**, v. 2, n. 9, Sep. 1997. Disponível em: <https://journals.uic.edu/ojs/index.php/fm/article/view/548>. Citado na página 39.

TIAN, Z. Smart contract defect detection based on parallel symbolic execution. In: IEEE. **2019 3rd International Conference on Circuits, System and Simulation (ICCSS)**. [S.l.], 2019. p. 127–132. Citado na página 61.

TIKHOMIROV, S.; VOSKRESENSKAYA, E.; IVANITSKIY, I.; TAKHAVIEV, R.; MARCHENKO, E.; ALEXANDROV, Y. Smartcheck: Static analysis of ethereum smart contracts. In: **Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain**. [S.l.: s.n.], 2018. p. 9–16. Citado nas páginas 47 e 61.

TORRES, C. F.; BADEN, M.; NORVILL, R.; PONTIVEROS, B. B. F.; JONKER, H.; MAUW, S. Ægis: Shielding vulnerable smart contracts against attacks. In: **Proceedings of the 15th ACM Asia Conference on Computer and Communications Security**. [S.l.: s.n.], 2020. p. 584–597. Citado na página 61.

TORRES, C. F.; SCHÜTTE, J.; STATE, R. Osiris: Hunting for integer bugs in ethereum smart contracts. In: **Proceedings of the 34th Annual Computer Security Applications Conference**. [S.l.: s.n.], 2018. p. 664–676. Citado nas páginas 46, 47 e 61.

TORRES, C. F.; STEICHEN, M. *et al.* The art of the scam: Demystifying honeypots in ethereum smart contracts. In: **28th USENIX Security Symposium (USENIX Security 19)**. [S.l.: s.n.], 2019. p. 1591–1607. Citado na página 61.

TSANKOV, P.; DAN, A.; DRACHSLER-COHEN, D.; GERVAIS, A.; BUENZLI, F.; VECHEV, M. Securify: Practical security analysis of smart contracts. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.: s.n.], 2018. p. 67–82. Citado nas páginas 47 e 61.

VACCA, A.; SORBO, A. D.; VISAGGIO, C. A.; CANFORA, G. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. **Journal of Systems and Software**, Elsevier, p. 110891, 2020. Citado na página 18.

VARELA-VACA, Á. J.; QUINTERO, A. M. R. Smart contract languages: A multivocal mapping study. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 54, n. 1, p. 1–38, 2021. Citado na página 18.

WANG, A.; WANG, H.; JIANG, B.; CHAN, W. Artemis: An improved smart contract verification tool for vulnerability detection. In: IEEE. **2020 7th International Conference on Dependable Systems and Their Applications (DSA)**. [S.l.], 2020. p. 173–181. Citado na página 61.

WANG, H.; LIU, Y.; LI, Y.; LIN, S.-W.; ARTHO, C.; MA, L.; LIU, Y. Oracle-supported dynamic exploit generation for smart contracts. **IEEE Transactions on Dependable and Secure Computing**, IEEE, 2020. Citado na página 61.

WANG, S.; DING, W.; LI, J.; YUAN, Y.; OUYANG, L.; WANG, F. Decentralized autonomous organizations: Concept, model, and applications. **IEEE Transactions on Computational Social Systems**, v. 6, n. 5, p. 870–878, 2019. Citado na página 37.

WANG, S.; ZHANG, C.; SU, Z. Detecting nondeterministic payment bugs in ethereum smart contracts. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 3, n. OOPSLA, p. 1–29, 2019. Citado nas páginas 19, 32, 33, 34, 61, 75 e 80.

WANG, W.; SONG, J.; XU, G.; LI, Y.; WANG, H.; SU, C. Contractward: Automated vulnerability detection models for ethereum smart contracts. **IEEE Transactions on Network Science and Engineering**, IEEE, 2020. Citado nas páginas 50, 51 e 61.

WANG, X.; HE, J.; XIE, Z.; ZHAO, G.; CHEUNG, S.-C. Contractguard: Defend ethereum smart contracts with embedded intrusion detection. **IEEE Transactions on Services Computing**, IEEE, v. 13, n. 2, p. 314–328, 2019. Citado nas páginas 51 e 61.

WANG, X.; YANG, X.; LI, C. A formal verification method for smart contract. In: IEEE. **2020 7th International Conference on Dependable Systems and Their Applications (DSA)**. [S.l.], 2020. p. 31–36. Citado nas páginas 19, 49, 61 e 75.

WANG, Y.; LAHIRI, S. K.; CHEN, S.; PAN, R.; DILLIG, I.; BORN, C.; NASEER, I.; FERLES, K. Formal verification of workflow policies for smart contracts in azure blockchain. In: SPRINGER. **Working Conference on Verified Software: Theories, Tools, and Experiments**. [S.l.], 2019. p. 87–106. Citado na página 61.

WEISS, K.; SCHÜTTE, J. Annotary: A concolic execution system for developing secure smart contracts. In: SPRINGER. **European Symposium on Research in Computer Security**. [S.l.], 2019. p. 747–766. Citado na página 61.

WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. **Ethereum project yellow paper**, v. 151, n. 2014, p. 1–32, 2014. Citado nas páginas 29, 30, 31, 32, 34 e 35.

WÜSTHOLZ, V.; CHRISTAKIS, M. Harvey: A greybox fuzzer for smart contracts. In: **Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2020. p. 1398–1409. Citado na página 61.

XIAO, Y.; ZHANG, N.; LOU, W.; HOU, Y. T. A survey of distributed consensus protocols for blockchain networks. **IEEE Communications Surveys Tutorials**, v. 22, n. 2, p. 1432–1465, 2020. Citado nas páginas 25 e 26.

XING, C.; CHEN, Z.; CHEN, L.; GUO, X.; ZHENG, Z.; LI, J. A new scheme of vulnerability analysis in smart contract with machine learning. **Wireless Networks**, Springer, p. 1–10, 2020. Citado nas páginas 50 e 61.

XUE, Y.; MA, M.; LIN, Y.; SUI, Y.; YE, J.; PENG, T. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: IEEE. **2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2020. p. 1029–1040. Citado na página 61.

YAMASHITA, K.; NOMURA, Y.; ZHOU, E.; PI, B.; JUN, S. Potential risks of hyperledger fabric smart contracts. In: IEEE. **2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)**. [S.l.], 2019. p. 1–10. Citado na página 61.

YANG, Z.; LEI, H. Fether: An extensible definitional interpreter for smart-contract verifications in coq. **IEEE Access**, IEEE, v. 7, p. 37770–37791, 2019. Citado nas páginas 49 e 61.

YANG, Z.; LEI, H.; QIAN, W. A hybrid formal verification system in coq for ensuring the reliability and security of ethereum-based service smart contracts. **IEEE Access**, IEEE, v. 8, p. 21411–21436, 2020. Citado na página 61.

YU, X. L.; AL-BATAINEH, O.; LO, D.; ROYCHOUDHURY, A. Smart contract repair. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 29, n. 4, p. 1–32, 2020. Citado na página 61.

ZHANG, Q.; WANG, Y.; LI, J.; MA, S. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In: IEEE. **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.], 2020. p. 116–126. Citado na página 61.

ZHANG, R.; XUE, R.; LIU, L. Security and privacy on blockchain. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 3, jul. 2019. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3316481>. Citado na página 17.

ZHANG, S.; LEE, J.-H. Analysis of the main consensus protocols of blockchain. **ICT express**, Elsevier, v. 6, n. 2, p. 93–97, 2020. Citado nas páginas 25 e 26.

ZHANG, W.; BANESCU, S.; PASOS, L.; STEWART, S.; GANESH, V. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In: IEEE. **2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.], 2019. p. 456–462. Citado na página 61.

ZHANG, Y.; MA, S.; LI, J.; LI, K.; NEPAL, S.; GU, D. Smartshield: Automatic smart contract protection made easy. In: IEEE. **2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.], 2020. p. 23–34. Citado na página 61.

ZHENG, J.; WILLIAMS, L.; NAGAPPAN, N.; SNIPES, W.; HUDEPOHL, J. P.; VOUK, M. A. On the value of static analysis for fault detection in software. **IEEE transactions on software engineering**, IEEE, v. 32, n. 4, p. 240–253, 2006. Citado na página 47.

ZHENG, Z.; XIE, S.; DAI, H.-N.; CHEN, W.; CHEN, X.; WENG, J.; IMRAN, M. An overview on smart contracts: Challenges, advances and platforms. **Future Generation Computer Systems**, v. 105, p. 475–491, 2020. ISSN 0167-739X. Citado nas páginas 17, 39, 40 e 41.

ZHOU, E.; HUA, S.; PI, B.; SUN, J.; NOMURA, Y.; YAMASHITA, K.; KURIHARA, H. Security assurance for smart contract. In: IEEE. **2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)**. [S.l.], 2018. p. 1–5. Citado na página 61.

ZHU, Q.; LOKE, S. W.; TRUJILLO-RASUA, R.; JIANG, F.; XIANG, Y. Applications of distributed ledger technologies to the internet of things: A survey. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 52, n. 6, p. 1–34, 2019. Citado na página 17.

ZHUANG, Y.; LIU, Z.; QIAN, P.; LIU, Q.; WANG, X.; HE, Q. Smart contract vulnerability detection using graph neural networks. In: **Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)**. [S.l.: s.n.], 2020. p. 3283–3290. Citado nas páginas 50 e 61.


ICMC
USP
SÃO CARLOS