Verificação formal para detecção de vulnerabilidades em contratos inteligentes

Gustavo Oliveira Dias[1]
Prof. Dr. Adenilso da Silva Simão[1]

[1]Instituto de Ciências Matemáticas e de Computação - ICMC

05 de julho de 2021

Sumário

Sumário

Blockchain

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))

Principais características:

- Armazenamento descentralizado e distribuído;
- Imutabilidade;
- Transparência;
- Dispensa da necessidade de confiança em uma terceira parte.

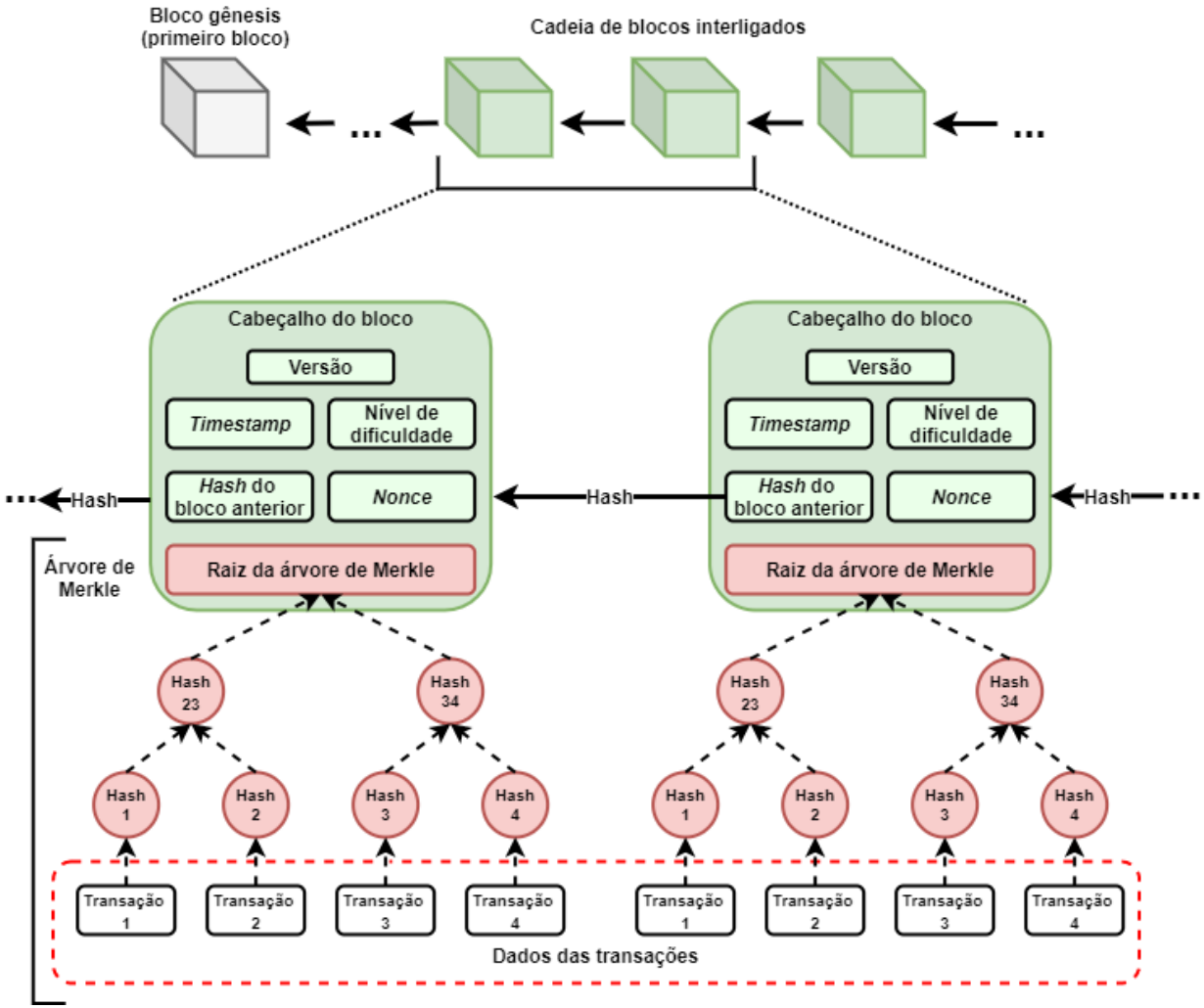
Blockchain Bitcoin

- Primeiro caso de êxito: **Bitcoin**;
- Criada em 2008 por Satoshi Nakamoto;
- Criptomoeda gerada e gerenciada de forma distribuída;
- Não possui entidades centralizadoras;
- Formada por uma rede de nós conectados auto-gerenciáveis;
- Nós trabalham para manter a integridade do sistema.

Blockchain - Transações e blocos

- Sempre que a posse de uma unidade ou fração de Bitcoin é transferida, uma **transação** é gerada;
- Quando uma nova transação ocorre, suas informações são transmitidas pela rede, tais como:
  - Contas envolvidas;
  - Quantia transferida;
  - Assinatura digital;
  - Horário da transação.
- Os nós da rede, conhecidos como **mineradores**, coletam as transações e as armazenam em **blocos**;
- Transações de um bloco são organizadas eu uma **árvore de Merkle**:
  - Nós folhas: transações;
  - Demais nós: Referências de *hash*.

Blockchain - Estrutura



Blockchain - Cabeçalho do bloco

1. Versão;
2. Hash do bloco anterior;
3. Raiz da árvore de Merkle;
4. *Timestamp*;
5. Nível de dificuldade;
6. *nonce*.

Criação do bloco:

- O minerador forma um bloco preliminar com os dados 1 ao 5;
- Por meio da resolução de um quebra-cabeça computacional, é obtido o *nonce* do bloco;
  - **Mineração** do bloco.
- O *nonce* é adicionado ao bloco;
- O bloco é transmitido pela rede.

Blockchain - Validação dos blocos

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.

Os nós são estimulados a manter a **integridade** das transações por meio de **mecanismos de incentivos**.

Blockchain - Validação dos blocos

Algoritmo de consenso

- Regras a serem seguidas pelos nós para criação e validação dos blocos;
- Recompensa para os nós honestos;
- Garante que todos os nós da rede concordem com o histórico de transações que compõem os blocos da blockchain;

Blockchain - Algoritmos de consenso

Proof-of-Work (PoW)

- Utilizado nas blockchains Bitcoin e Ethereum;
- Consiste na competição entre os mineradores para resolução do *nonce* do bloco;
- O vencedor transmite o bloco pela rede para validação;
- Quanto maior o poder computacional do minerador, maior a chance de vencer;
- Resulta em um alto esforço computacional e gasto energético;
- O vencedor receber um valor da criptomoeda como recompensa (e.g., Bitcoin);
- **Desvantagem**: Esforço computacional elevado.

Blockchain - Algoritmos de consenso

Proof-of-Stake(PoS)

- Nós **validadores** participam da criação dos blocos;
- Validadores investem um **valor de participação**;
- Quanto maior o valor, maior a chance de ser escolhido para validar o próximo bloco;
- Ethereum 2.0: migração para o protocolo PoS.

Blockchain - Algoritmos de consenso

Practical Byzantine Fault Tolerance(PBFT)

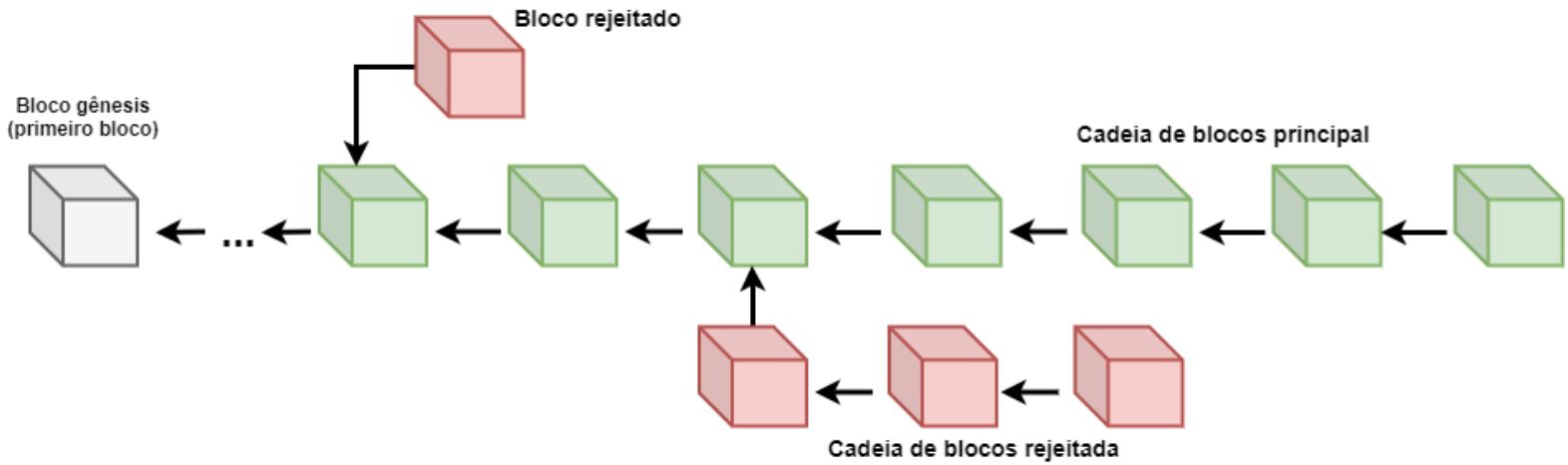
- Há dois tipos de nós: **cliente** e **servidor**;
- Um nó cliente envia um bloco aos nós servidores;
- Se o bloco for validado por um número suficiente de nós, então é adicionado à blockchain;
- Utilizado na blockchain *Hyperledger Fabric*.

Blockchain - Escolha do histórico de transações

- O trabalho dos mineradores consiste na alternação de duas tarefas:
  - Analisar a validade de um novo bloco;
  - Criar o próximo bloco.
- A transmissão e entrega de novos blocos sobrem influências externas;
- Assim, nós da rede podem ter informações distintas à sua disposição ao mesmo tempo;
- Quando um nó recebe mais de um nó com o mesmo bloco pai, é criado um *fork*;

Blockchain - Escolha do histórico de transações

- Na Bitcoin, a **cadeia mais longa** é escolhida;
- A ramificação escolhida pela maioria dos nós é considerada a **cadeia de blocos principal**.

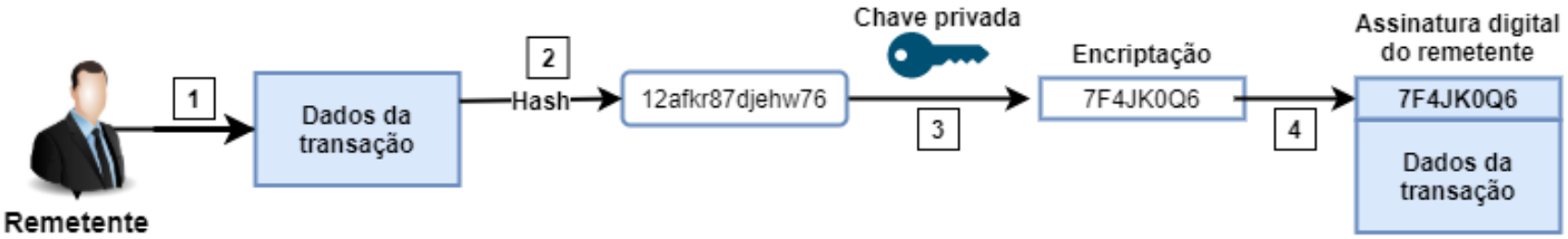


Blockchain - Criptografia e autorização de transações

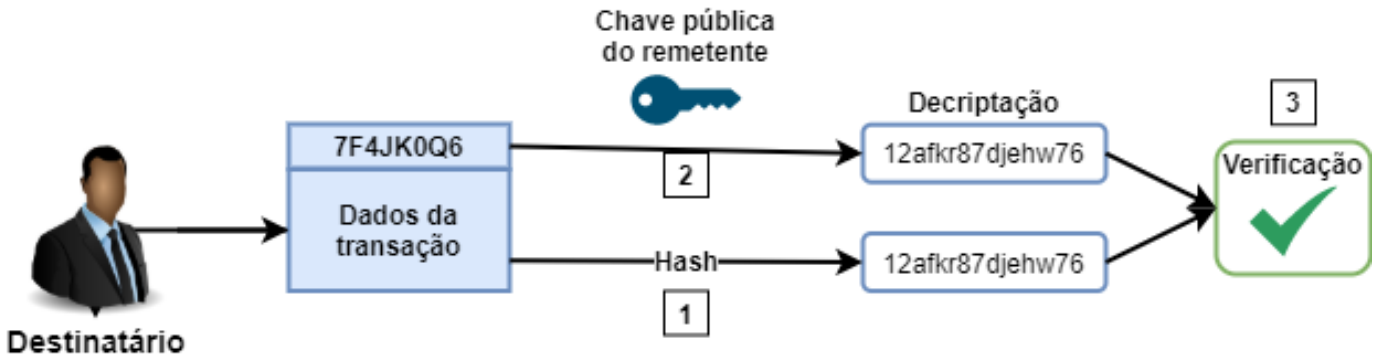
- Um dos pilares da blockchain consiste na **segurança** e **integridade** das transações;
- Apenas o proprietário legítimo pode realizar a transferência de uma criptomoeda para outra conta;
- Para isso, é utilizada uma **assinatura digital**;
  - Criptografia assimétrica;
  - Chaves pública e privada.
- A assinatura é utilizada na **assinatura** e na **verificação** de uma transação.

Blockchain - Criptografia e autorização de transações

- **Assinatura:**



- **Verificação:**



Sumário

Ethereum

- Apesar da blockchain ter sido criada para uma aplicação financeira (i.e., a Bitcoin), seus conceitos e protocolos podem ser estendidos para diversas áreas;
- É apenas um fim para um meio;

- Em 2014 foi criada a blockchain **Ethereum**;
- Introdução dos contratos inteligentes;
- Pioneira na expansão das aplicações da blockchain.

Ethereum

- Definida como uma plataforma de computação **distribuída** composta por uma rede de computadores que operam de forma **descentralizada**, **autônoma** e **democrática**;
- Permite a geração, transferência e gerenciamento de sua criptomoeda, o **Ether**;
- Seu funcionamento é baseado na implantação da **contratos inteligentes** (CI):
  - Programas de computador;
  - Executam automatica e obrigatoriamente aquilo que foi programado;
  - Estabelecem um acordo entre os envolvidos;
  - Quando compilado, é convertido em *bytecode*;
  - Executado na Máquina Virtual Ethereum (MVE);
  - Geração de transações.

Ethereum

- Transações são coletadas e estruturadas nos blocos em uma *trie*;
- Opera de forma semelhante à Bitcoin na garantia da **imutabilidade**;
- Aplicações Descentralizadas (do inglês, *Descentralized Applications* (DApps)):
  - Mais de 3 mil utilizam a Ethereum [1].
- Em 2021, seu valor de mercado superou $400 bilhões [2];
- Há ao menos 2 milhões de CIs já executados [3].

[1] <https://www.stateofthedapps.com/>
[2] <https://coinmarketcap.com/>
[3] <https://etherscan.io/>

Ethereum - Propriedades

As aplicações que executam sobre a Ethereum dispõem de uma série de propriedades:

- Descentralização;
- Imutabilidade;
- Persistência de dados;
- Execução autônoma;
- Acurácia.

Ethereum - Contas

Dois tipos:

- Conta de Propriedade Externa (CPE):
  - Armazena os fundos dos usuários em Wei (1 Ether = 10^{18} Wei);
  - Associadas e controladas por uma chave privada.
- Conta de contrato (CC):
  - Conta associada à um CI;
  - Controladas pelo código de um *bytecode* executável.

Ethereum - Contas e mensagens

Contas:

- Estado global da Ethereum = estado de todas as contas;
- Estado dinâmico definido por:
  - *nonce:*
    - CPE: transações iniciadas pelo proprietário;
    - CC: contratos criados pela conta.
  - *balance:* saldo em Wei da CPE ou CC;
  - *storageRoot:* CC - *Hash* da raiz da *trie*;
  - *codeHash: Hash* do *bytecode* da CC correspondente.

Mensagem:

- Meio de interação CPE -> CC e CPE -> CPE;
- Especifica uma instrução a ser executada;
- Usadas para transferência de Ether e execução de CIs;
- Sempre que uma CC recebe uma mensagem seu código é ativado

Ethereum - Transações e transição de estados

Transação:

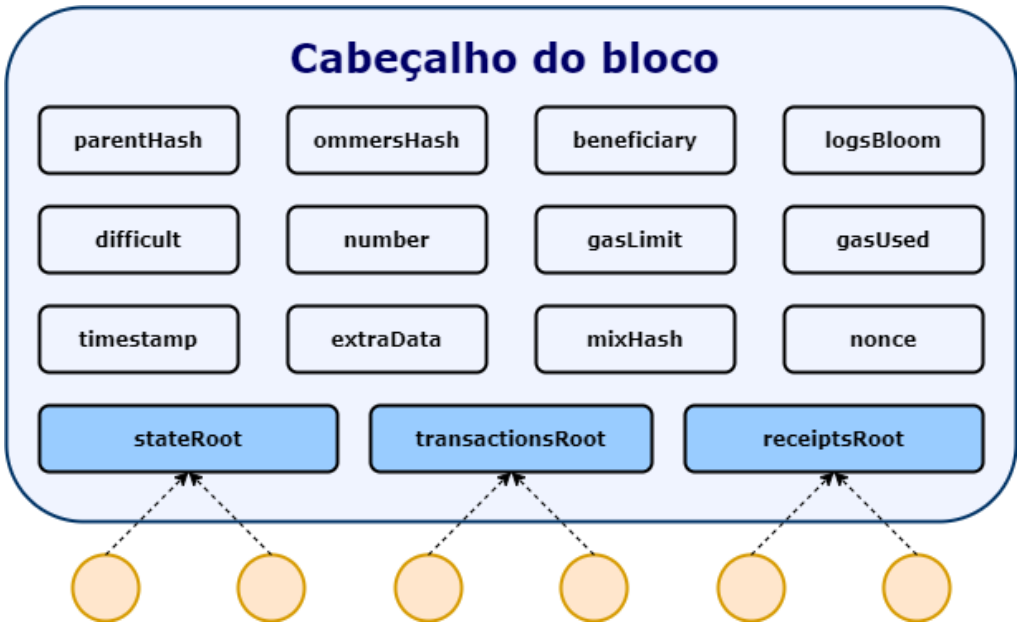
- Pacote de dados que armazena uma mensagem;
- Execução resultado em um custo computacional (*gas*);
- *gasPrice:* custo, em Wei, por unidade de *gas*;
- Quanto maior o *gasPrice*, maior a recompensa para o minerador;
- *gasLimit:*
  - Evita o consumo indefinido de *gas*.

Transição de estados:

- Quando uma transação é executada, algum atributo de alguma conta é alterado.
  - Ex: saldo da conta ou variável de um CI.
- Transição: $(S, TX) \longrightarrow S'$.

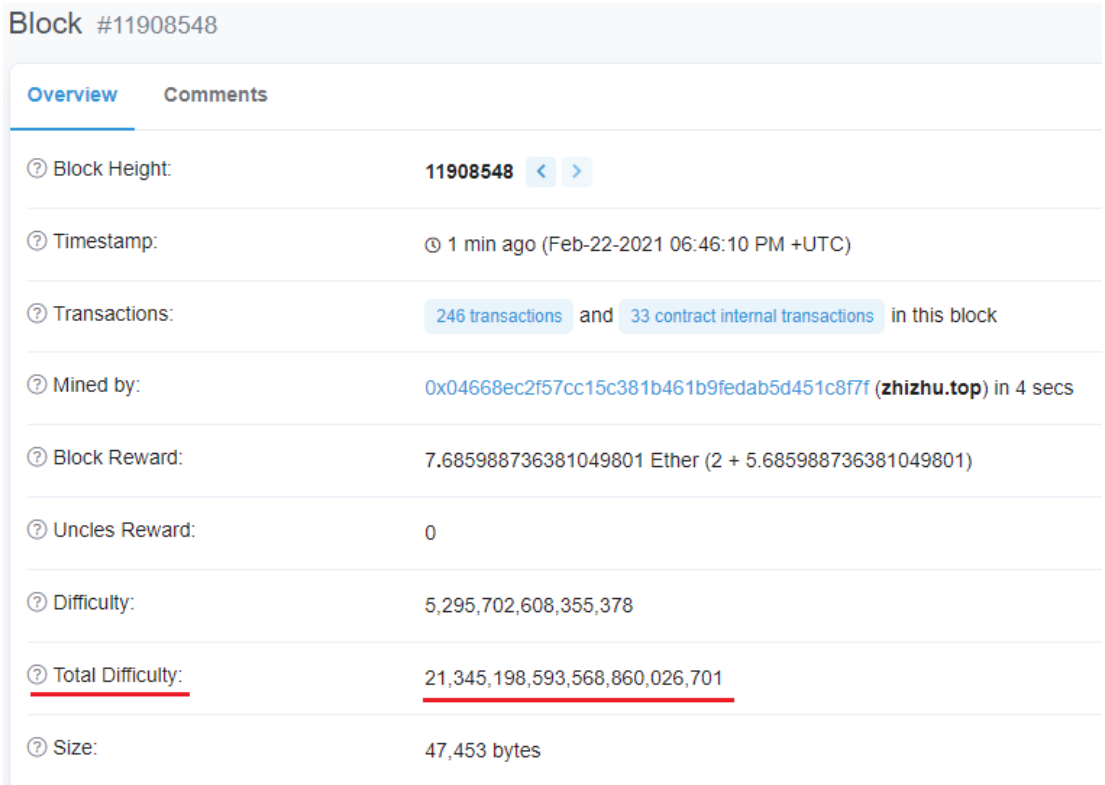
Ethereum - Formação dos blocos

- Transações são transmitidas pela rede;
- Mineradores possuem uma *pool* de transações pendentes;
- Transações da *pool* são escolhidas para formar os blocos.



Ethereum - Validação

- Na Ethereum podem ocorrer *forks* com versões distintas do histórico de transações;
- Cadeia principal: aquela com a maior dificuldade acumulada.



Blockchain - Estágios evolutivos

Após o êxito da Bitcoin, outras criptomoedas e tecnologias baseadas na blockchain foram desenvolvidas.

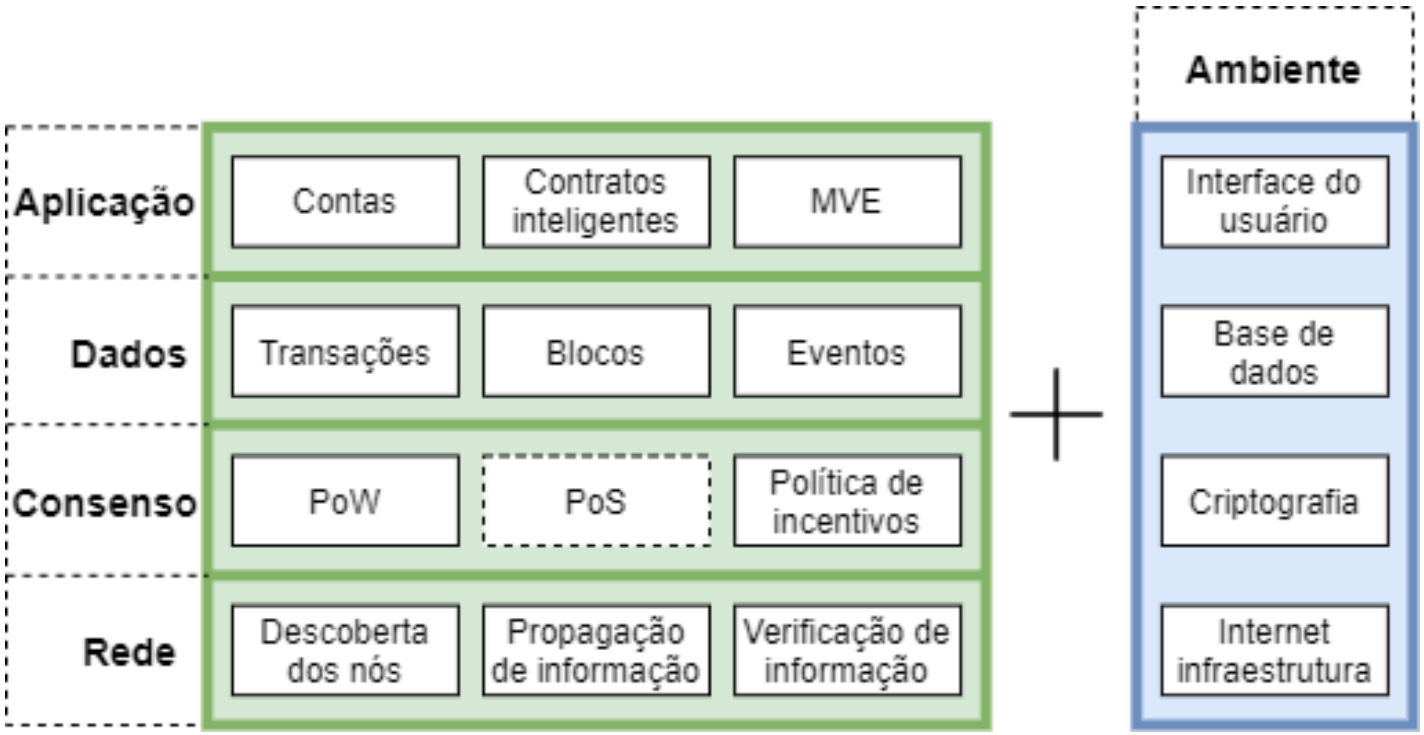
Esses avanços são classificados como:

- Blockchain 1.0 (Bitcoin):
  - Geração e gerenciamento de criptomoedas.
- Blockchain 2.0 (Ethereum):
  - Junção entre CIs e criptomoedas;
  - Aplicações financeiras;
  - Ex: DApps, Organizações Autônomas Descentralizadas (OADs).
- Blockchain 3.0:
  - Aplicações em outras áreas além da financeira;
  - Ex: Cuidados médicos, inteligência artificial, internet das coisas, governança descentralizada, entre outras.

Ethereum - Aplicações

- Sistema de **tokens**:
  - Tokenização;
  - *Stable coins*: Tether, USD Coin, Pax Gold;
  - Padrão ERC-20.
- Organização Autônoma Descentralizada:
  - Regras operacionais e de gerenciamento são programadas em um CI;
  - Abolição de modelos baseados em hierarquia;
  - Redução de custos;
  - Primeira OAD: *The DAO*.
- Cuidados métodos e serviços de saúde:
  - Integração de dados de prontuários.

Ethereum - Arquitetura em camadas



Sumário

Contratos inteligentes

- 1997: Primeira proposta;
- 2014: Primeira aplicação - blockchain Ethereum;
- Outras plataformas que implementam CIs: Hyperledger Fabric, Corda e Stellar.

Desenvolvimento de um CI

Cláusulas contratuais estabelecidas em comum acordo entre as partes envolvidas são expressas por meio de programas de computador executáveis.

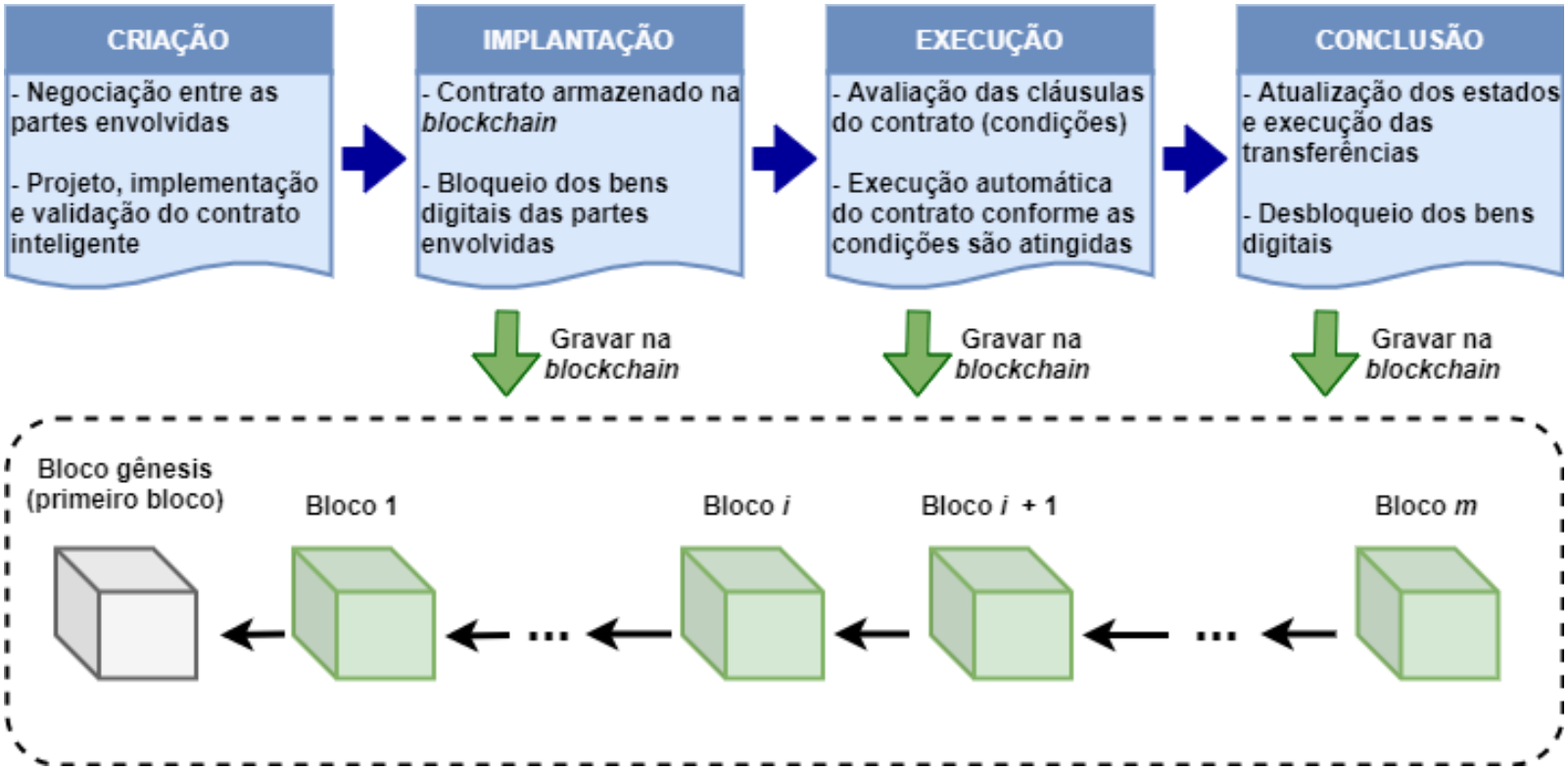
- Solidity: linguagem desenvolvida para CIs na Ethereum;
- CIs são sempre compilados para *bytecode*;
- O *bytecode* é executado na MVE.

Contratos inteligentes

```
1  pragma solidity ^0.5.9;
2
3  contract Coin {
4      address public minter;
5      mapping (address => uint) public balances;
6
7      event Sent(address from, address to, uint amount);
8
9      constructor() public{
10         minter = msg.sender;
11     }
12
13     function mint(address receiver, uint amount) public  {
14         if (msg.sender != minter) return;
15         balances[receiver] += amount;
16     }
17
18     function send(address receiver, uint amount) public {
19         if (balances[msg.sender] < amount) return;
20         balances[msg.sender] -= amount;
21         balances[receiver] += amount;
22         emit Sent(msg.sender, receiver, amount);
23     }
24 }
```

Contratos inteligentes

A utilização dos CIs possui um ciclo de vida de quatro fases: **criação**; **implantação**; **execução**; e **conclusão**.



Sumário

Contratos inteligentes

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.

Vulnerabilidades deixadas nos códigos dos CIs podem torná-los alvos de **ataques**.

Contratos inteligentes - Vulnerabilidades e ataques

Aplicações desenvolvidas por meio de CIs costumam envolver transferências e gerenciamento de grandes quantidades de bens digitais.

- Primeiro ataque: *The DAO Attack* (2016);]
- Vulnerabilidade explorada: **Reentrância**;
- Um usuário malicioso transferiu **3,6 milhões** em Ether para sua conta;
- Equivalente a **50 milhões** de dólares.

Este caso teve grande repercussão e motivou estudos e estratégias para **detecção** e **prevenção** de **vulnerabilidades** em CIs.

Contratos inteligentes - Vulnerabilidades e ataques

Parte dos erros e vulnerabilidades explorados em ataques contra os CIs são ocasionados pelo desalinhamento que há entre a semântica da linguagem Solidity e a intuição dos desenvolvedores.

Solidity possui elementos similares ao de outras linguagens, mas que não são implementados da mesma forma.

Há registro de uma série de vulnerabilidades que resultaram na exploração de CIs e enormes perdas financeiras.

Contratos inteligentes - Vulnerabilidades e ataques

Vulnerabilidades alvos desta pesquisa:

- Reentrância;
- *Delegatecall injection*;
- Contrato suicida.

Vulnerabilidade - Reentrância

- Explorada no *The DAO Attack*;
- Pode ocorrer quando um contrato pode ser invocado recursivamente sucessivas vezes;
- Ocorre por meio de uma função *callback*.

```
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 }
```

Exploração da reentrância

- $1^{\underline{o}}$ Passo: Publicação do contrato.

```
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 }
```

Exploração da reentrância

- $2^{\underline{o}}$ Passo: Doação para o contrato utilizando sua CPE.

```
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 }
```

Exploração da reentrância

- 3º Passo: Invocação da função *fallback*, que invoca a função `withdraw`;
- Ether é transferido para o `Attacker`.

```
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 }
```

Exploração da reentrância

- 4º Passo: Chamadas sucessivas à função `withdraw`.
- Da forma como é usada, a função `call`, que é uma função *callback*, permite nova invocação da função `withdraw` antes da execução do comando `credit[msg.sender] -= amount;`
- Verificação da linha 8 tem êxito novamente.

```
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 }
```

Exploração da reentrância

- As transferências são realizadas sucessivamente até ocorrer um dos seguintes eventos:
  - Todo *gas* é utilizado;
  - Pilha de chamadas da MVE é totalmente preenchida;
  - Saldo do `SimpleDAO` é zerado.
- Como evitar:
  - Atualizar as variáveis de estado antes da invocação de outro contrato;
  - Trava *mutex*;
  - Utilizar o método `transfer`.

Os danos causados pelo *The DAO Attack* foram revertidos por meio de um *hard fork*, que causou uma divisão dos mineradores da Ethereum.

Vulnerabilidade - *Delegatecall injection*

- Explorado no ataque contra a *Parity Multsignature Wallet*;
- Em 2017, 31 milhões de dólares foram subtraídos em um ataque;
- `delegatecall`: Usado para inserir o *bytecode* de um contrato no *bytecode de outro contrato*;
- O contrato que é chamado pode alterar as variáveis de estado do contrato que o invocou.

Exploração da *delegatecall injection*

```
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 }
```

Exploração da *delegatecall injection*

- Envio de uma transação com o campo `msg.data` contendo `initWallet` como a função a ser chamada;
- Os endereços de posse do contrato (`_owners`) foram substituídos pelo endereço do atacante;
- Foi realizada a transferência de 31 milhões de dólares em Ether.
- Como evitar:
  - Declarar contrato a ser compartilhado (`WalletLibrary`) como uma biblioteca (`library`).

Vulnerabilidade - Contrato suicida

- Um contrato pode ser "morto" pelo dono do contrato ou outra parte confiável;
- Métodos: `suicide` ou `selfdestruct`;
- O *bytecode* e o armazenamento do contrato é deletado;

A vulnerabilidade **contrato suicida**, acontece quando uma **autenticação inadequada** permite que algum invasor tome posse do contrato e execute a função para matar o contrato.

- Foi explorada em um segundo ataque contra *Parity Wallet*;
- **280 milhões** de dólares em Ether foram permanentemente **bloqueados**.

Exploração da vulnerabilidade contrato suicida

- Como resposta ao primeiro ataque, foi adicionado o modificador `only_uninitialized`;
- Porém, o contrato `WalletLibrary` foi deixado como não inicializado;
- O invasor passou pelo modificador, se declarou dono do contrato, e invicou o método suicide.

```
1  contract WalletLibrary {
2      modifier only_uninitialized { if (m_numOwners > 0) throw; _; }
3
4      function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
5          initDaylimit(_daylimit);
6          initMultiowned(_owners, _required);
7      }
8      function kill(address _to) onlymanyowners(sha3(msg.data)) external {
9      suicide(_to);
10     }
11 }
```

Ethereum - Vulnerabilidades

Além da reentrância, *delegatecall injection* e contrato suicida, existem diversas outras vulnerabilidades encontradas na literatura, e que devem ser evitadas.

Desde o *The DAO Attack*, muitos esforços foram realizados para desenvolvimento de métodos, ferramentas e frameworks para **verificação de CIs** e **detecção de vulnerabilidades**.

Sumário

Vulnerabilidades em CIs

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.

- Linguagens como Solidity não foram desenvolvidas para serem verificadas formalmente;
- Solidity não é uma linguagem perfeita;
- Questões relacionadas com a execução dos CIs na Ethereum muitas vezes não são compreendidas pelos desenvolvedores.

Vulnerabilidades em CIs - Como evitar?

Serviços de auditoria:

- Organizações: OpenZeppeling, Solidified e SmartDec;
- Podem ser muito custosas para pequenas organizações e desenvolvedores autônomos.

Solução

Frameworks e ferramentas de verificação de CIs

Verificação de CIs - Frameworks e ferramentas

Dois tipos de verificação:

- Proativa:
  - Aplicada antes da implantação.
- Reativa:
  - Monitoramento do contrato durante a execução;
  - Verificação em **tempo de execução**.

Métodos de verificação:

- Análise de código;
- Métodos formais;
- *Fuzzing*;
- Técnicas de Inteligência Artificial (IA);
- Em tempo de execução.

Verificação de CIs - Análise de código

- Estratégia geralmente automatizada;
- Detecção de erros;
- Identificação de vulnerabilidades;
- Aspectos analisados:
  - Fluxo de controle;
  - Fluxo de dados;
  - Interface;
  - Fluxo de informações;
  - Caminhos de execução.
- Verificação pode ser **estática**, **dinâmica** ou **híbrida**;
- Normalmente é utilizada alguma representação intermediária estruturada:
  - Grafo de fluxo de controle;
  - Árvore sintática em XML.
- **Execução simbólica**.

Verificação de CIs - Métodos formais

Modelagem formal de sistems:

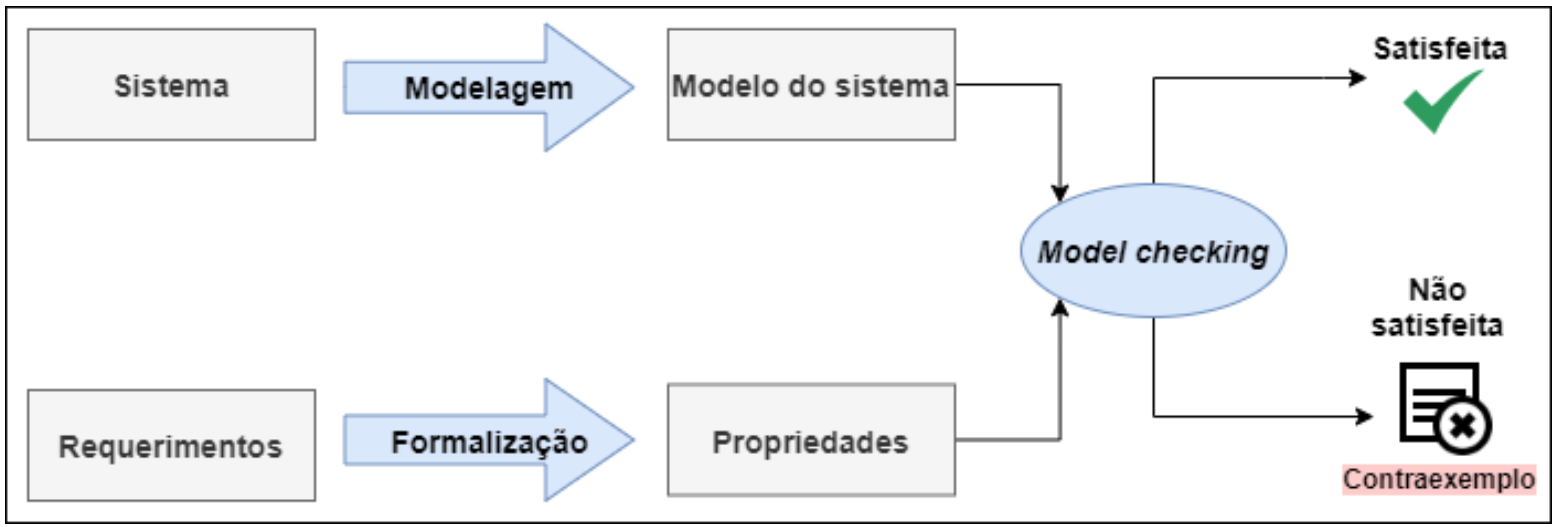
- Representações matemáticas;
- Lógica de processos;
- Modelos baseados em estados;
- Especificação formal de propriedades;
- Verificação é realizada a partir das especificações fornecidas;
- Podem ser custosos em relação ao tempo e recursos;
- Aplicados em sistemas críticos.

Principais métodos:

- *Model checking*;
- Demonstração de teoremas;
- Verificação dedutiva.

Model checking

- Verificação de sistemas de transição de estados;
- Três etapas:
  - Modelagem;
  - Especificação: especificação formal dos requerimentos;
    - Propriedades: Lógicas temporais.
  - Verificação.



Model checking

Problema: explosão de estados

Ocorre quando o modelo obtido a partir do código fornecido permite a exploração de uma quantidade excessivamente grande de estados.

Alternativas:

- *Model checking* simbólico;
- *Model checking* limitado;
- *Model checking* estatístico.

Métodos formais - Demonstração de teoremas

- Modelagem do sistema e a especificação das propriedades é feita por meio de formalismos matemáticos;
- Uso de inferência dedutiva para produção de provas;
- Demonstrações são realizadas por meio de regras de inferência;
- Formalismos utilizados:
  - Lógica proposicional;
  - Lógica temporal;
  - Lógica de ordem superior;
  - Lógica de primeira ordem.

Métodos formais - Verificação dedutiva

- Geração de provas matemáticas a partir do sistema e suas especificações;
- Se essas provas se mostrarem verdadeiras, então isso implica na conformidade do sistema com sua especificação.

Verificação de CIs - *Fuzzing*

- Técnica para teste de software;
- *Fuzzer*: Ferramenta para geração de entradas de teste de forma iterativa e aleatória;
- Baseado na geração de mutações de um conjunto de entradas;
- O *fuzzer* encerra quando:
  - Um objetivo é alcançado (ex: erro ou travamento do sistema);
  - Tempo limite é atingido.

Verificação de CIs - Inteligência artificial

- **Machine learning** e **deep learning**;
- CIs vulneráveis são convertidos em vetores ou matrizes;
- Modelos são utilizados para treinar os algoritmos;
- Em alguns casos, os modelos são obtidos com auxílio de outras ferramentas para verificação de CIs.

Verificação em tempo de execução

- Código é instrumentalizado;
- É embutido um sistema de monitoramento;
- Assim, é possível reagir à atividades suspeitas;
- Violações de propriedades podem ser verificadas e prevenidas durante a execução do contratado;
- Método pouco utilizado.

Sumário

Metodologia e técnicas de pesquisa

Mapeamento Sistemático (MS)

Identificar e classificar o conteúdo relacionado à **verificação para correção** ou **detecção de vulnerabilidades** para aprimoramento da segurança dos CIs.

Com base no MS:

- Seleção do método de verificação e detecção de vulnerabilidades;
- Seleção das vulnerabilidades abordadas;
- Definição da estratégia para validação da proposta.

Sumário

Mapeamento Sistemático

Três fases:

- Planejamento;
  - Objetivo e protocolo do MS.
- Condução;
  - Identificação dos estudos primários;
  - Extração e síntese dos dados.
- Publicação dos resultados.
  - Questões de pesquisa são respondidas;
  - Relato dos resultados.

Protocolo do MS

Questões de pesquisa:

- **QP 1.** Quais abordagens têm sido propostas?
- **QP 2.** Quando e onde os estudos têm sido publicados?
- **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?

Protocolo do MS

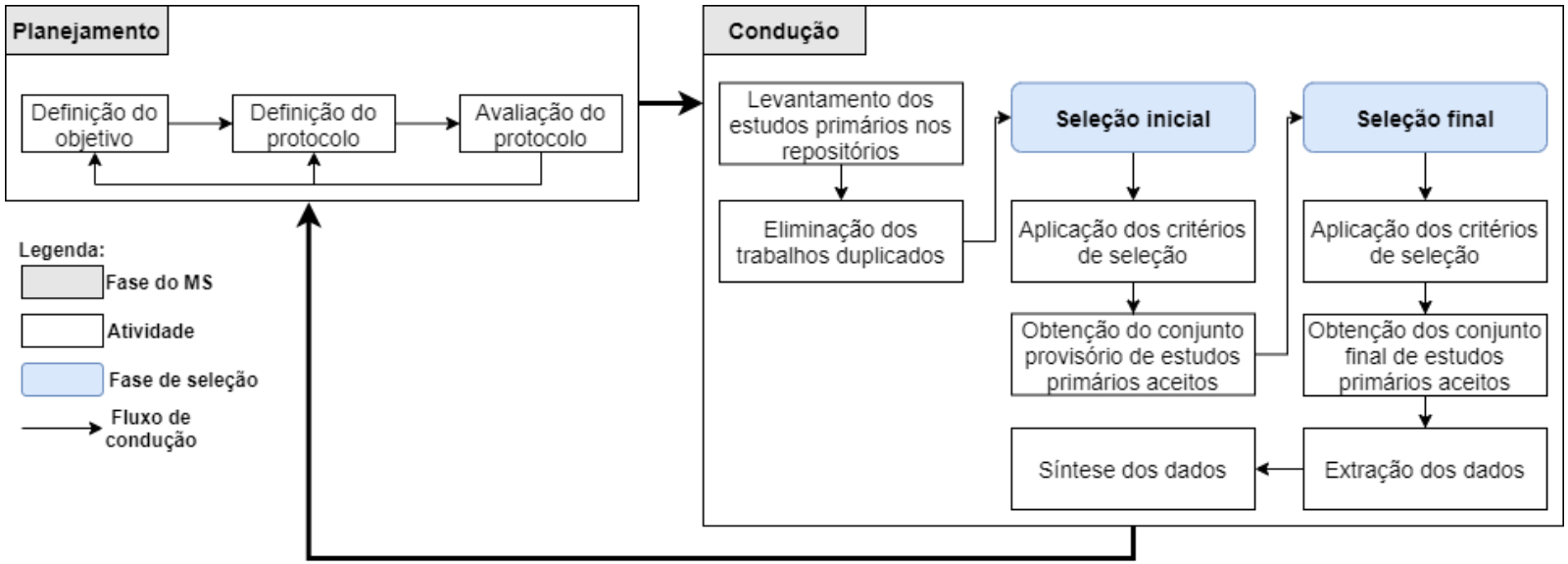
String de busca:

(*"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**)

Motores de busca e bases bibliográficas:

- *Engineering Village*;
- *Scopus*;
- *Web of Science*;
- *IEEE Xplore*;
- *ACM Digital Library*.

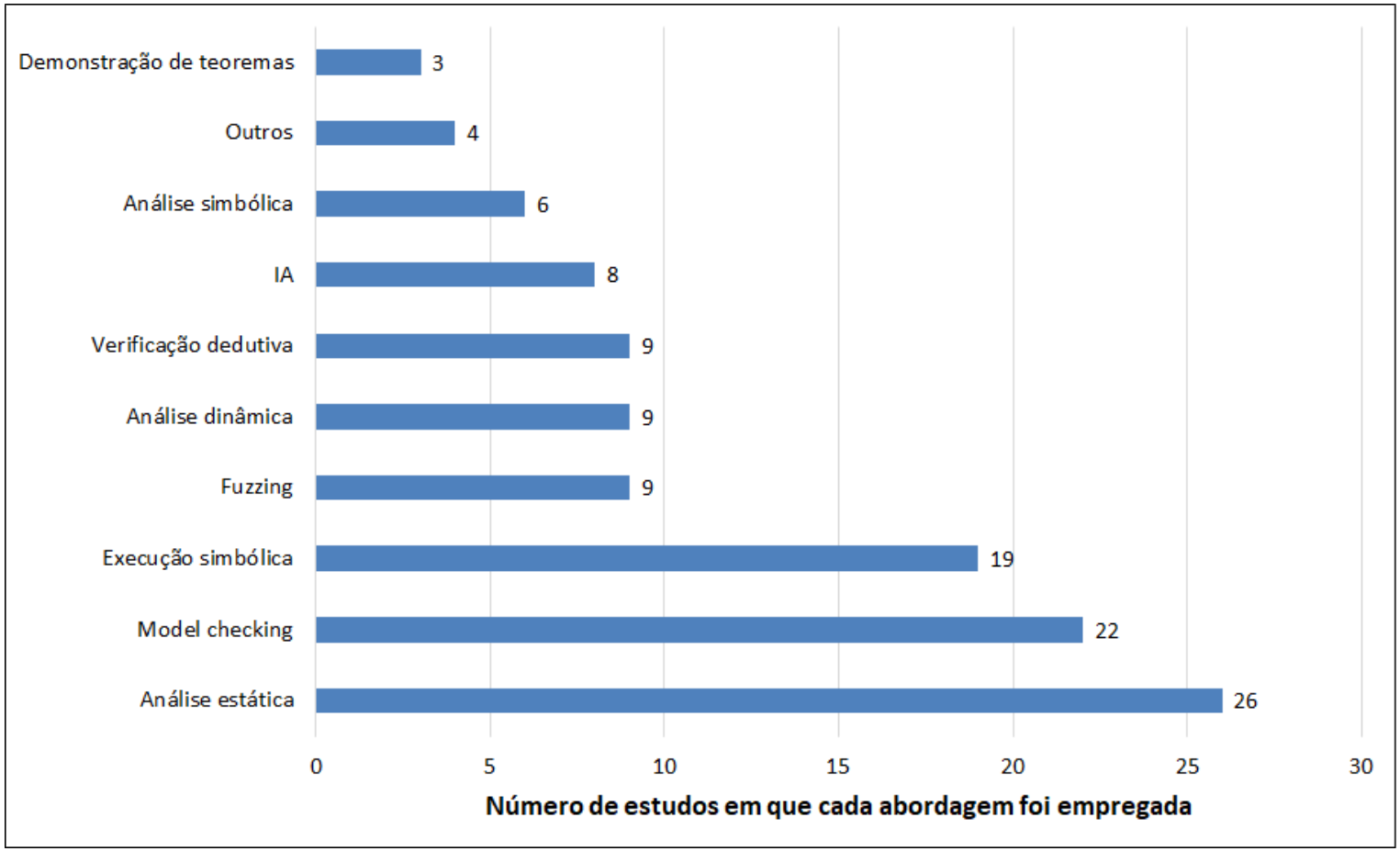
Planejamento e condução



- Total de estudos selecionados: 2091
- Total de estudos aceitos: **104**

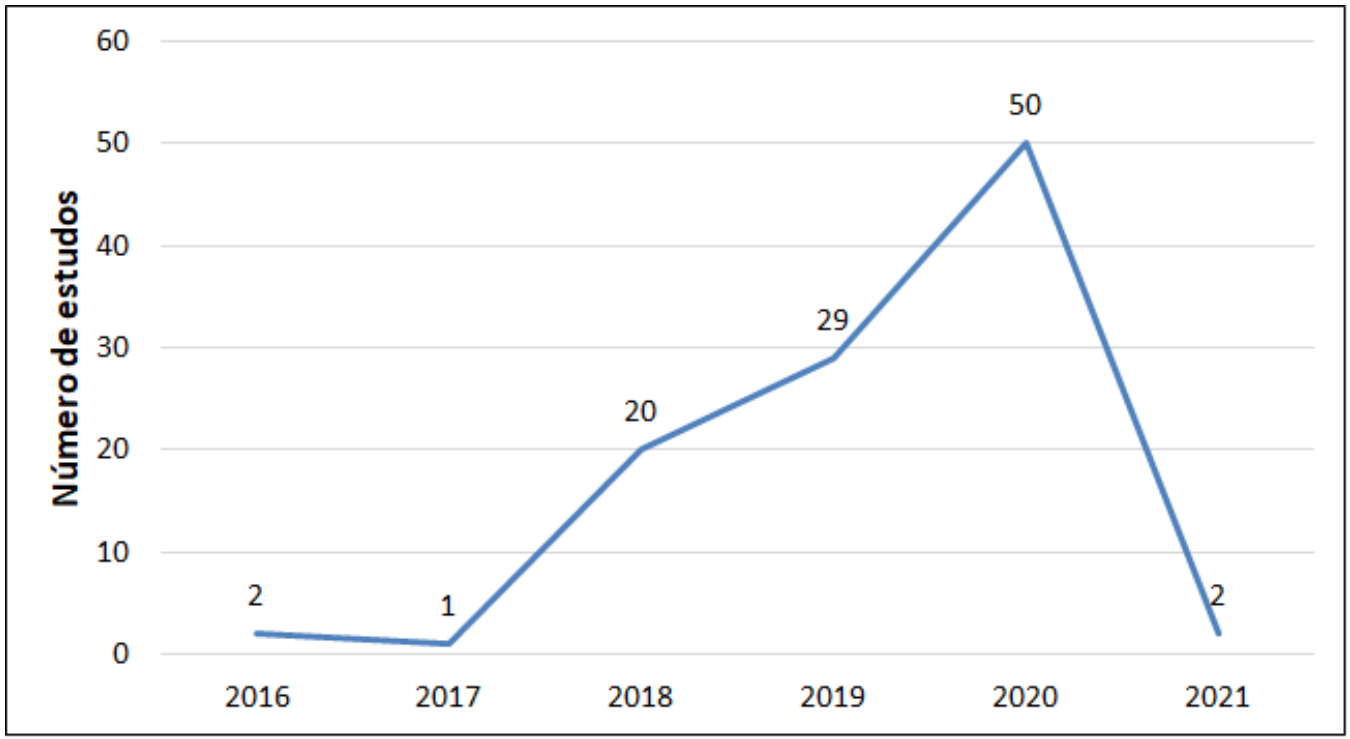
Resultados - QP1

- **QP1.** Quais abordagens têm sido propostas?



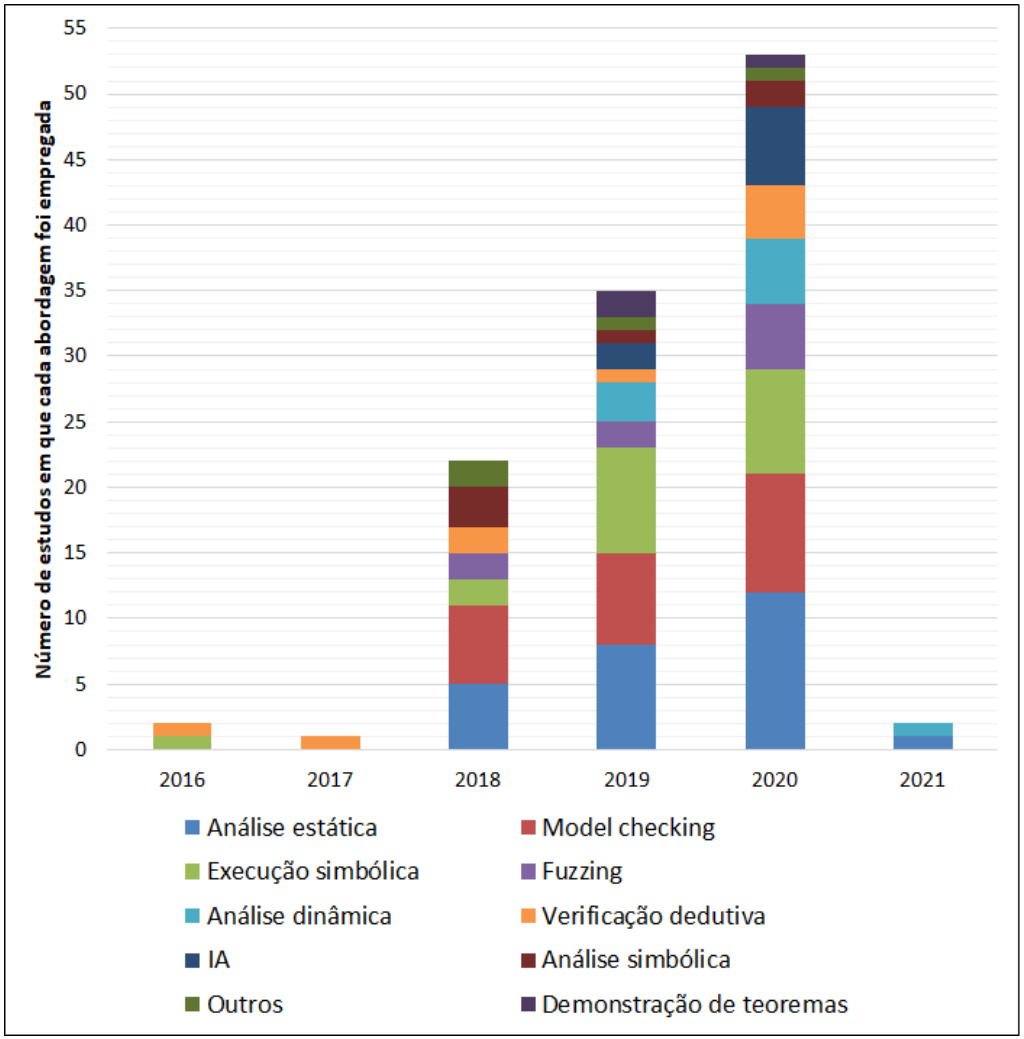
Resultados - QP2

- **QP2.** Quando e onde os estudos têm sido publicados?



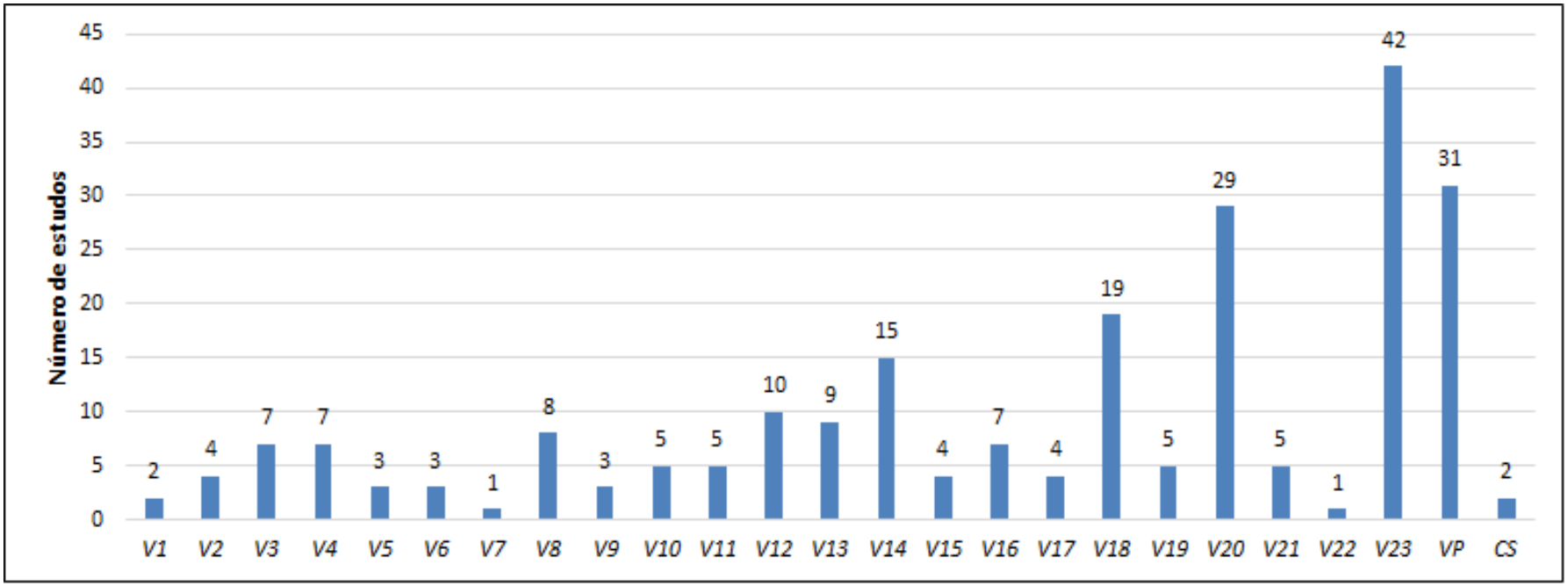
Resultados - QP2

- **QP2.** Quando e onde os estudos têm sido publicados?



Resultados - QP3

- **QP3.** Quais problemas ou vulnerabilidades relacionados aos CIs têm sido abordados?



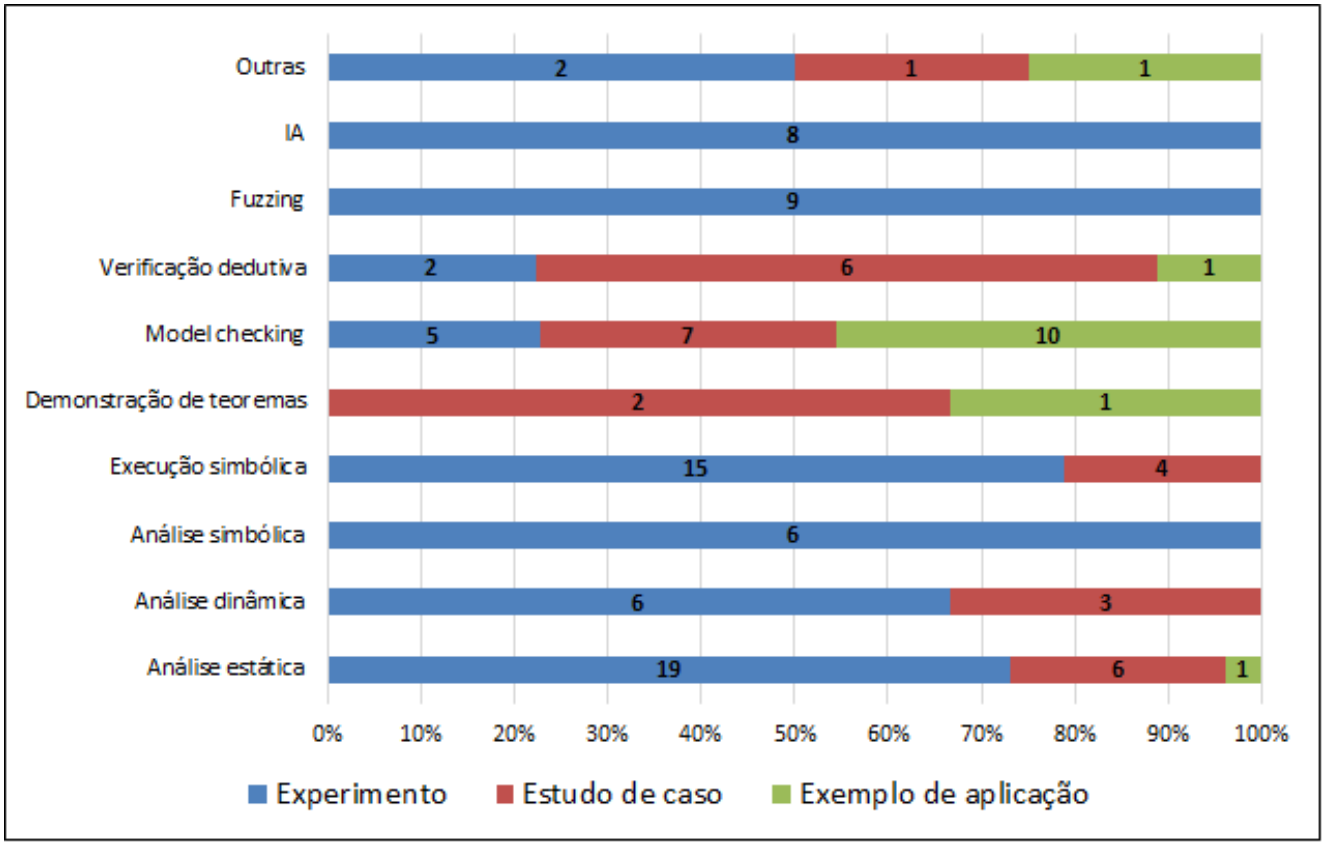
Resultados - QP3

- **QP3.** Quais problemas ou vulnerabilidades relacionados aos CIs têm sido abordados?

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		

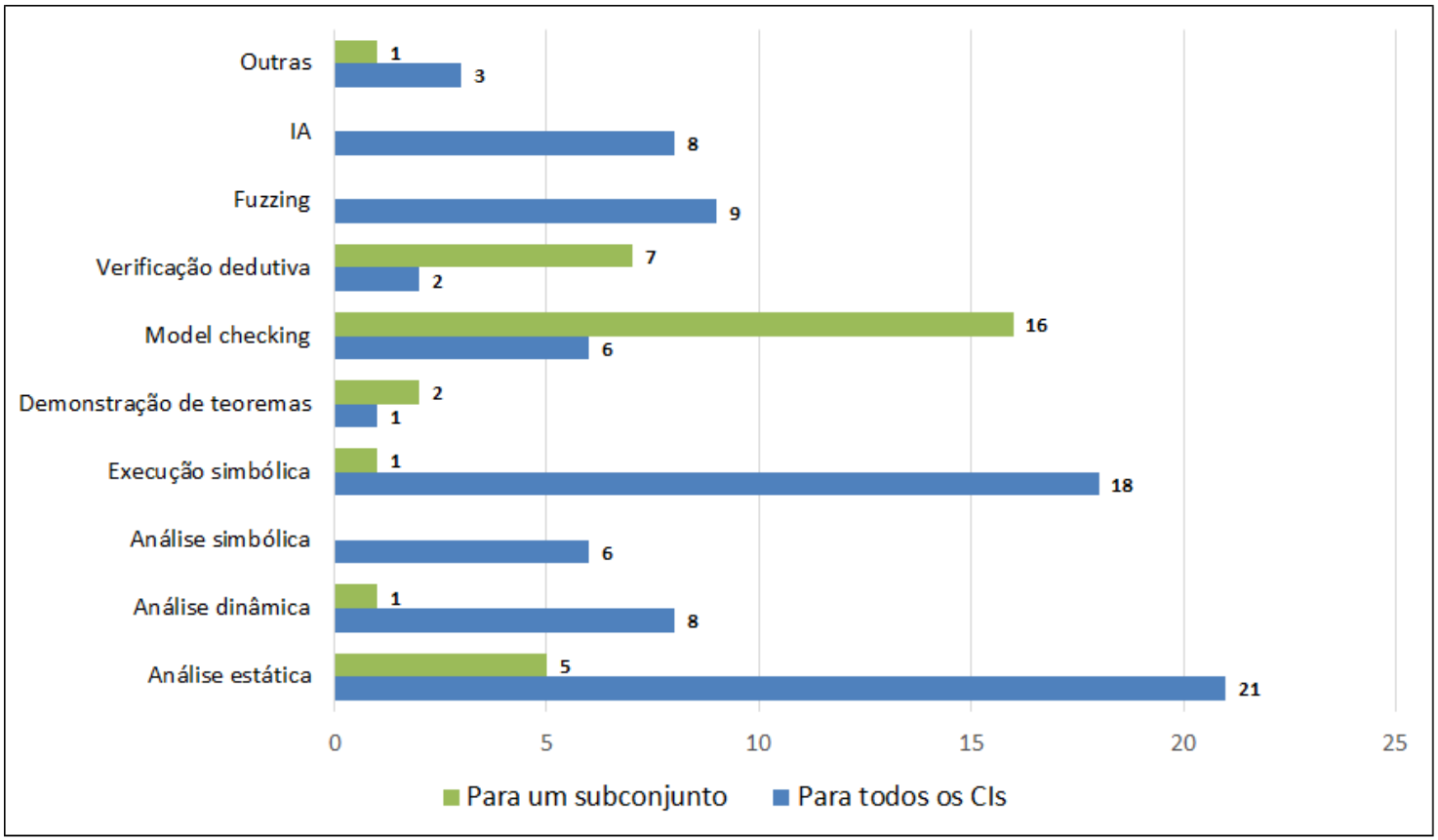
Resultados - QP4

- **QP4.** Quais estratégias de validação foram utilizadas?



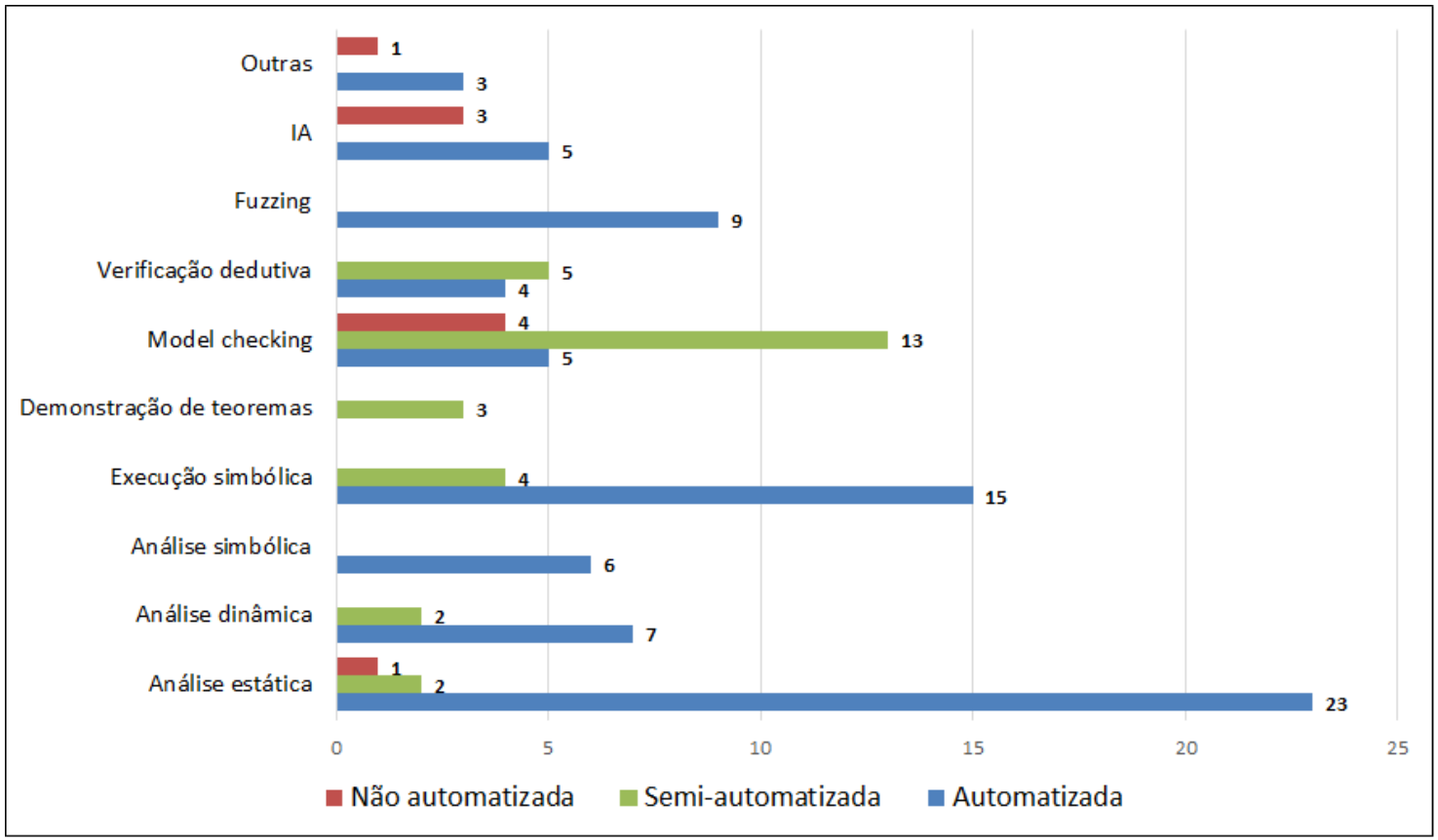
Resultados - QP5

- **QP5.** Quais são as limitações presentes nas abordagens?



Resultados - QP5

- **QP5.** Quais são as limitações presentes nas abordagens?



Discussão

- *Model checking* está entre as abordagens mais utilizadas nos últimos anos;
- Há poucos estudos com validação experimental;
- Reentrância foi a vulnerabilidade mais abordada;
- A violação de propriedades

Sumário

Objetivo

Objetivo geral

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**.

Objetivos específicos:

- 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.

Proposta

Framework para verificação formal de CIs escritos em Solidity na fase de pré-implantação.

Tarefas realizadas por meio do framework:

- Inclusão do código fonte em Solidity;
  - Pode ser mais de um;
  - Relações intercontratuais;
  - Diagrama de fluxo de interação entre CIs.
- Conversão do código para modelo de estados;
- Exibição do modelo obtido;
- Inserção das propriedades e seleção das vulnerabilidades;
  - Modelo para descrição;
  - Mais próximo de linguagem natural.

Proposta

Framework para verificação formal de CIs escritos em Solidity na fase de pré-implantação.

Tarefas realizadas por meio do framework:

- Conversão para lógica temporal;
- Verificação;
- Exibição dos resultados.
  - Indicação das vulnerabilidades no código.

Proposta

Framework para verificação formal de CIs escritos em Solidity na fase de pré-implantação.

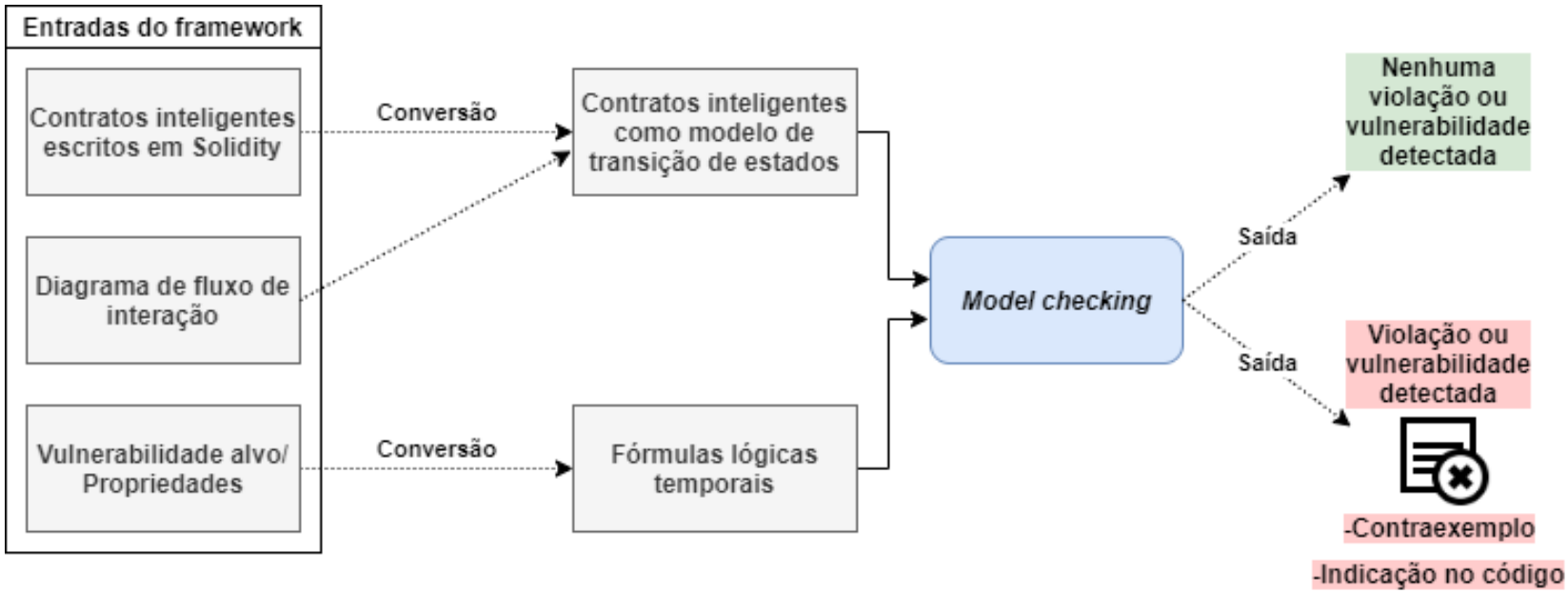
Validação:

- Experimento sobre um conjunto de CIs com vulnerabilidades já conhecidas;
- Avaliar a eficácia e acurácia da verificação;
- Eficiência;
  - Consumo de memória, processamento e tempo de execução.

Estudo de caso:

- Aplicado para tratar de propriedades funcionais relativas aos requisitos do CI.

Framework proposto



Sumário

O framework de Mavridou et al. (2019)

- Framework VeriSolid;
- Modelagem gráfica dos CIs como máquina de estados finito (MEF);
- Extensão do framework FSolidM, de Mavridou e Laszka (2018);
- MEF $\rightarrow$ Código Solidity;
  - Erros evitados direto na construção.
- Baseada na MEF, propriedades podem ser especificadas;
- Não é abordada nenhuma vulnerabilidade específica.

O framework de Nelaturu *et al.* (2020)

- Extensão do VeriSolid;
- Permite a relação entre múltiplos contratos;
  - Diagrama de implementação.
- Implantação do automática dos CIs gerados;
- Vulnerabilidades de reentrância e exceções não tratadas;
  - Prevenidas na construção de MEF.
- Vulnerabilidades de endereço curto, contrato suicida e bloqueio de Ether;
  - Verificadas na modelagem ou implantação.
- Propriedades descritas por meio de *templates* em linguagem natural;
- Reentrância:
  - Não permite a execução de funções *callback*;
  - Desconsidera casos onde a função é usada sem causar, necessariamente, uma vulnerabilidade.

Framework proposto

Proposta do trabalho:

- Relações intercontratuais: Diagrama de fluxo de interação;
- Modelagem de funções *callback*;
- *Delegatecall injection*:
  - Não é abordada em nenhum trabalho por meio do *model checking*.
- Apontamento das vulnerabilidades no código;
- Tarefas manuais:
  - Especificação e seleção das vulnerabilidades/propriedades;
  - Diagrama de fluxo de interação.

Sumário

Plano de trabalho

Atividade	2020					2021						2022	
	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													■

Verificação formal para detecção de vulnerabilidades em contratos inteligentes

Gustavo Oliveira Dias[1]
Prof. Dr. Adenilso da Silva Simão[1]

[1]Instituto de Ciências Matemáticas e de Computação - ICMC

05 de julho de 2021