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

06 de agosto 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.

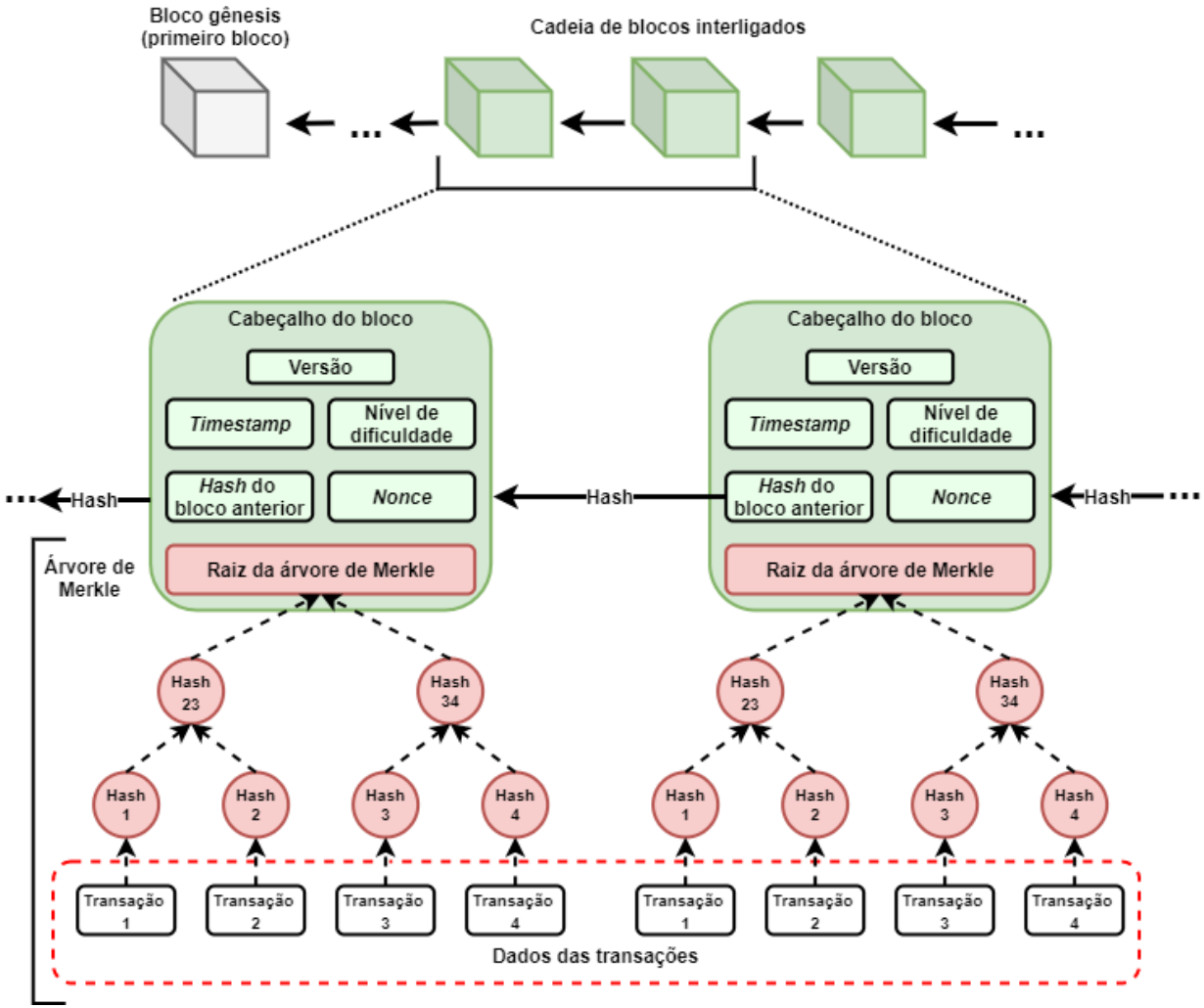
Blockchain Bitcoin

- 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 - 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.

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;
- Aplicações: Cuidados médicos, inteligência artificial, internet das coisas, governança descentralizada, entre outras.

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

Ethereum

- Permite a geração, transferência e gerenciamento de sua criptomoeda, o **Ether**;
- Seu funcionamento é baseado na implantação dos **contratos inteligentes** (CI):
  - Programas de computador;
  - Executam automatica e obrigatoriamente aquilo que foi programado;
  - Estabelecem um acordo entre os envolvidos;
  - Geração de transações.
- Opera de forma semelhante à Bitcoin na garantia da **imutabilidade**.

Ethereum - Propriedades

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

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

Ethereum - Aplicações

- Sistema de **tokens**:
  - Tokenização;
  - 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 médicos e serviços de saúde:
  - Integração de dados de prontuários.

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.

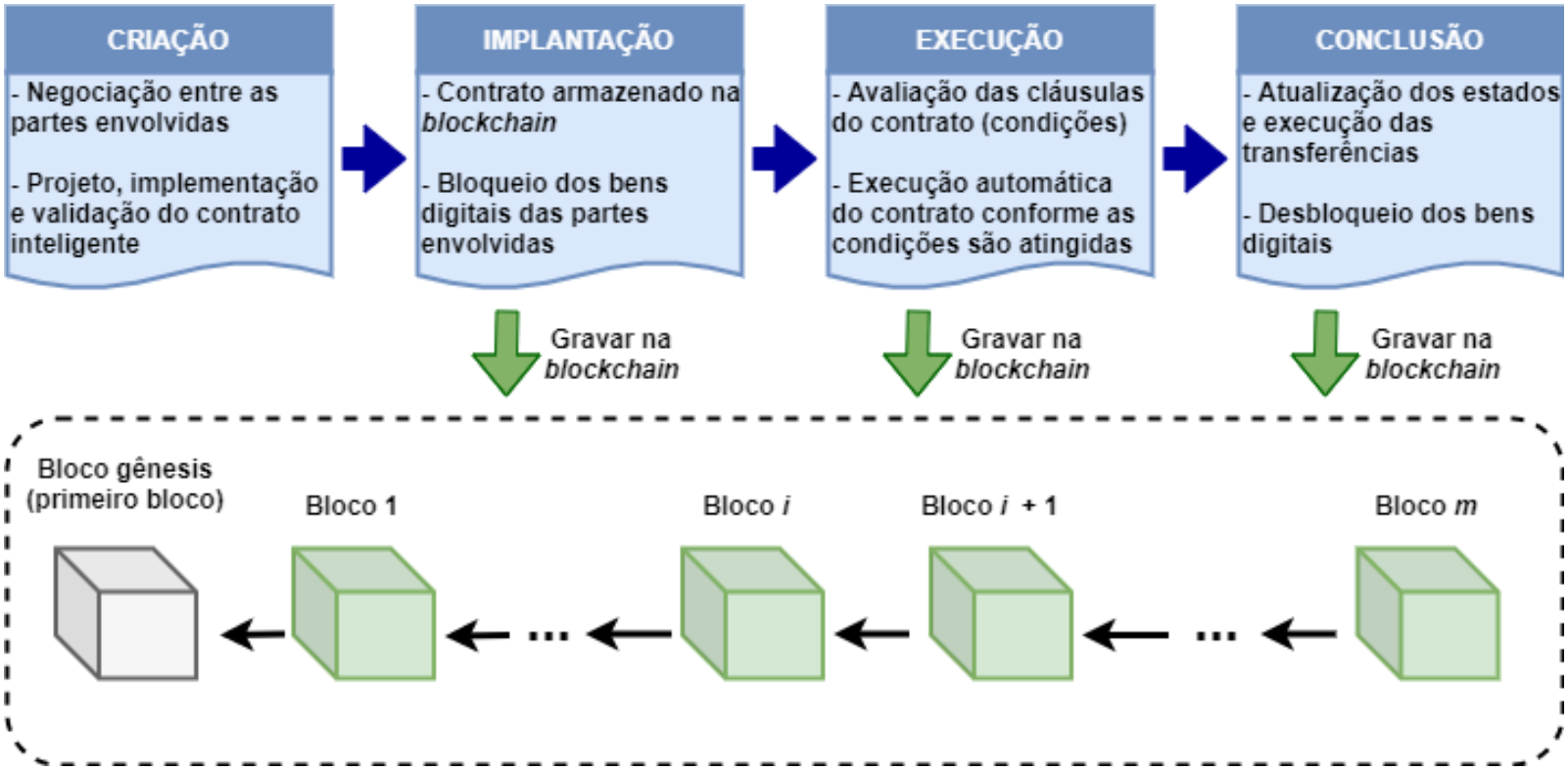
- Solidity: linguagem desenvolvida para CIs na Ethereum;
- CIs são sempre compilados para *bytecode*;
- O *bytecode* é executado na MVE.

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.

Aplicações desenvolvidas por meio de CIs costumam envolver transferências e gerenciamento de grandes quantias em Ether.

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

Contratos inteligentes - Vulnerabilidades e ataques

- 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

- 2º Passo: Doação para o contrato utilizando sua conta.

```
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*

- Explorada 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;
- O invasor atribuiu para si a posse do contrato e realizou a transferência para sua conta.

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 a *Parity Wallet*;
- **280 milhões** de dólares em Ether foram permanentemente **bloqueados**.

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**.

Questão de pesquisa

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

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 custosos 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**.

Verificação de CIs - Frameworks e ferramentas

Métodos de verificação:

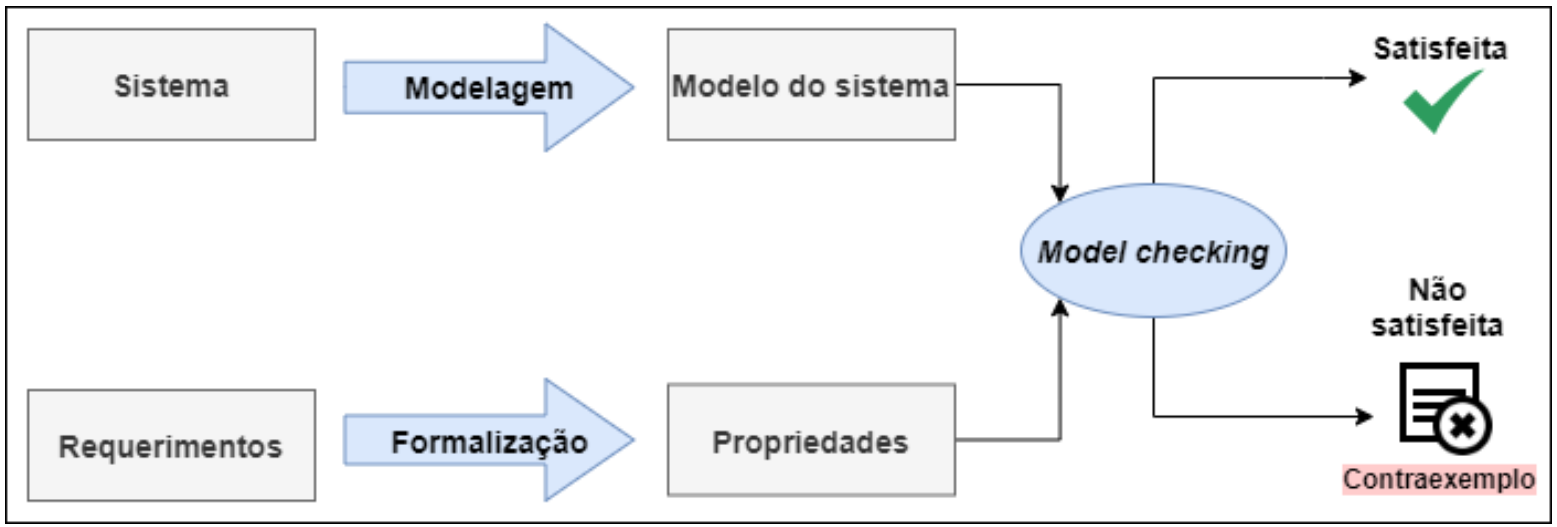
- Análise de código:
  - Pode ser **estática**, **dinâmica** ou **híbrida**;
  - **Execução simbólica**.
- Métodos formais:
  - *Model checking*;
  - Demonstração de teoremas;
  - Verificação dedutiva.

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**.

Model checking

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



Verificação de CIs - Frameworks e ferramentas

Métodos de verificação:

- *Fuzzing*:
  - Geração de mutações.
- Técnicas de Inteligência Artificial (IA):
  - *Machine learning* e *deep learning*.
- Em tempo de execução:
  - Instrumentalização de código.

Sumário

Objetivo da pesquisa

Objetivo geral

Propor uma estratégia para verificação formal para aprimoramento de segurança aplicada na fase de pré-implantação de CIs escritos em **Solidity** para detecção das vulnerabilidades de **reentrância**, **delegatecall injection** e **contrato suicida**, por meio da técnica de **model checking**.

Metodologia e técnicas de pesquisa

Objetivo do Mapeamento Sistemático (MS) realizado

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

Fundamentado no MS:

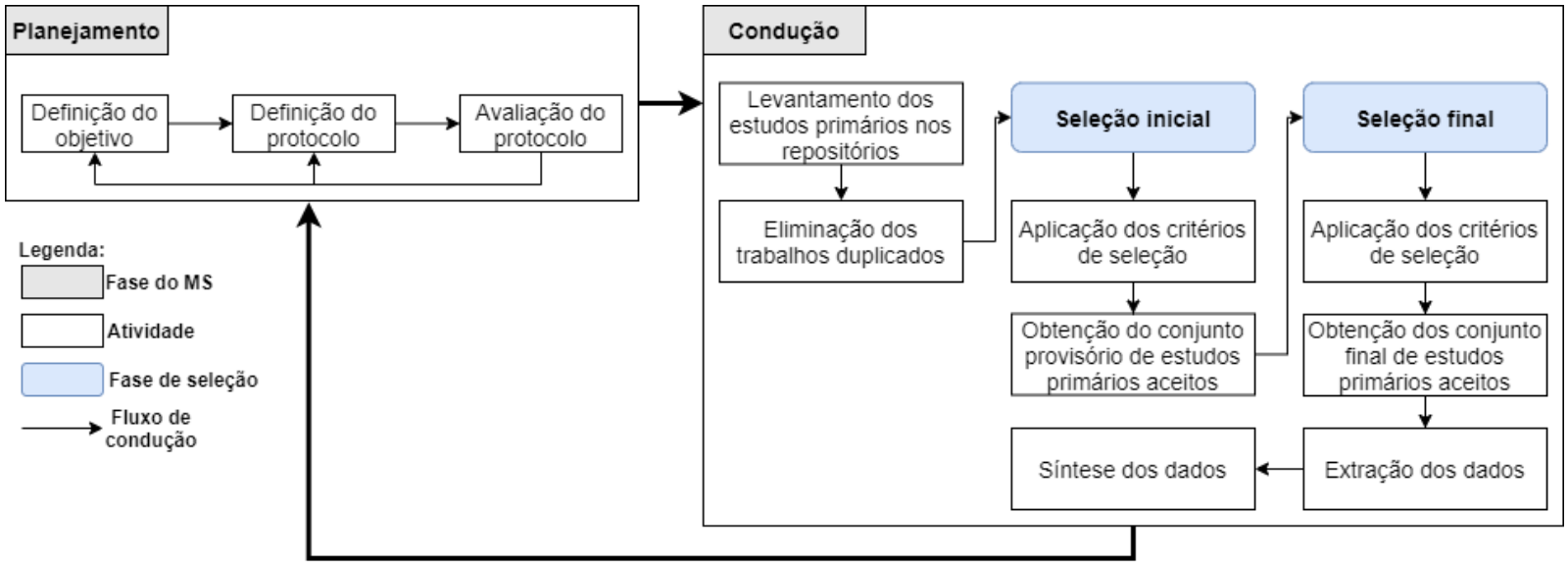
- Seleção do método de verificação e detecção de vulnerabilidades;
- Seleção das vulnerabilidades abordadas;
- Definição da estratégia para validação da proposta.

Sumário

Mapeamento Sistemático

Três fases:

- Planejamento;
- Condução;
- Publicação 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*.

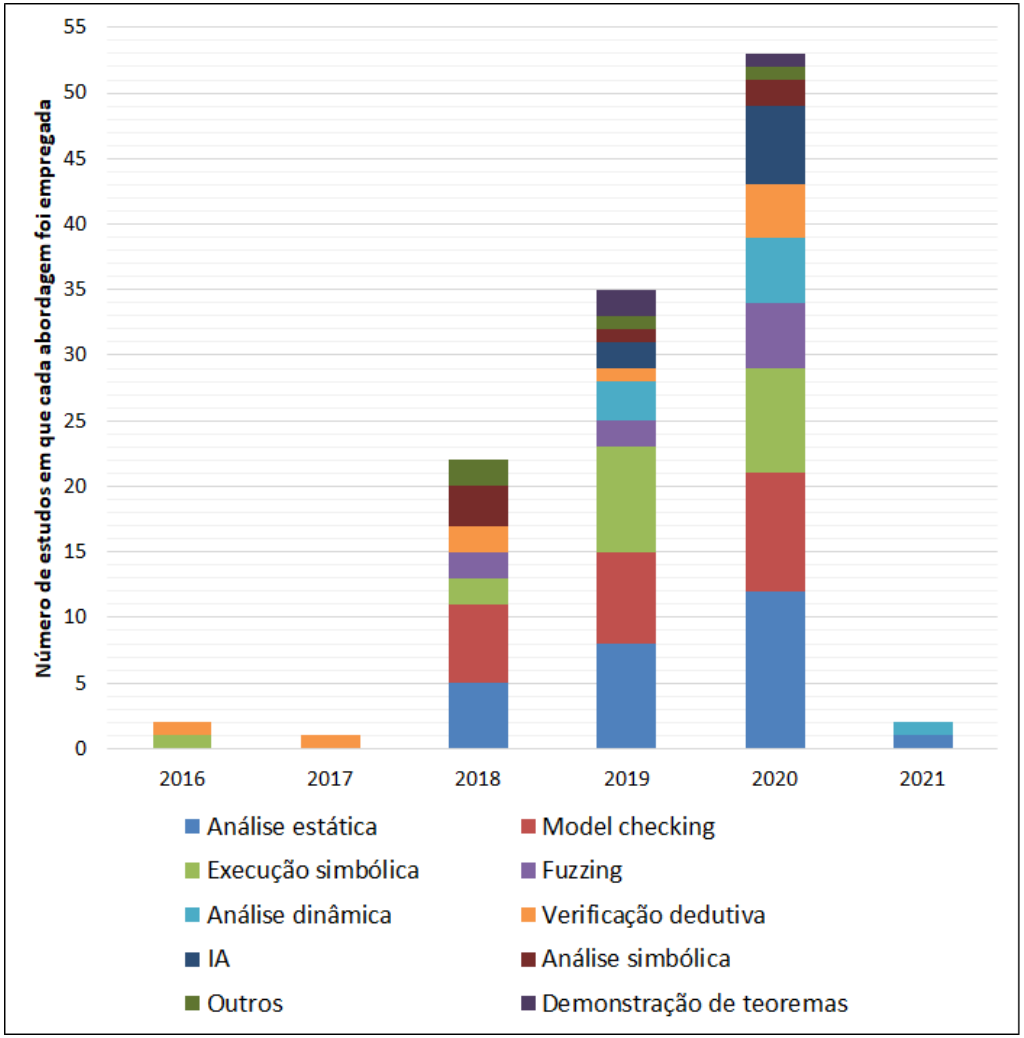
Planejamento e condução

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

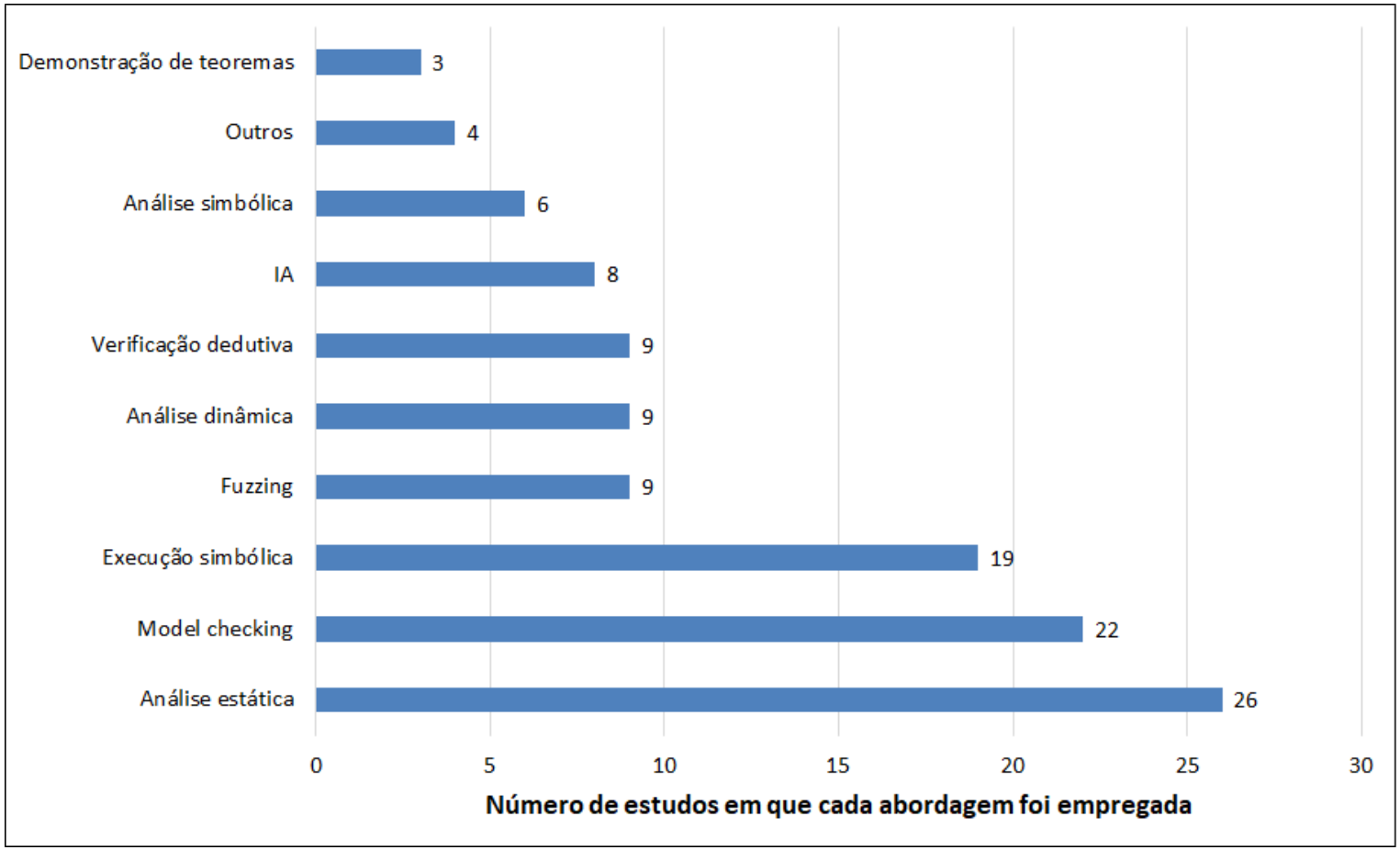
Discussão - *Model checking*

- **Está entre as abordagens mais utilizadas nos últimos anos**;
- Há poucos estudos com validação experimental;
- Reentrância foi a vulnerabilidade mais abordada;
- Violação de propriedades:
  - Amplamente abordada;
  - Principalmente *model checking*.
- Está entre as que apresentam melhor precisão e acurácia.

Discussão - *Model checking*



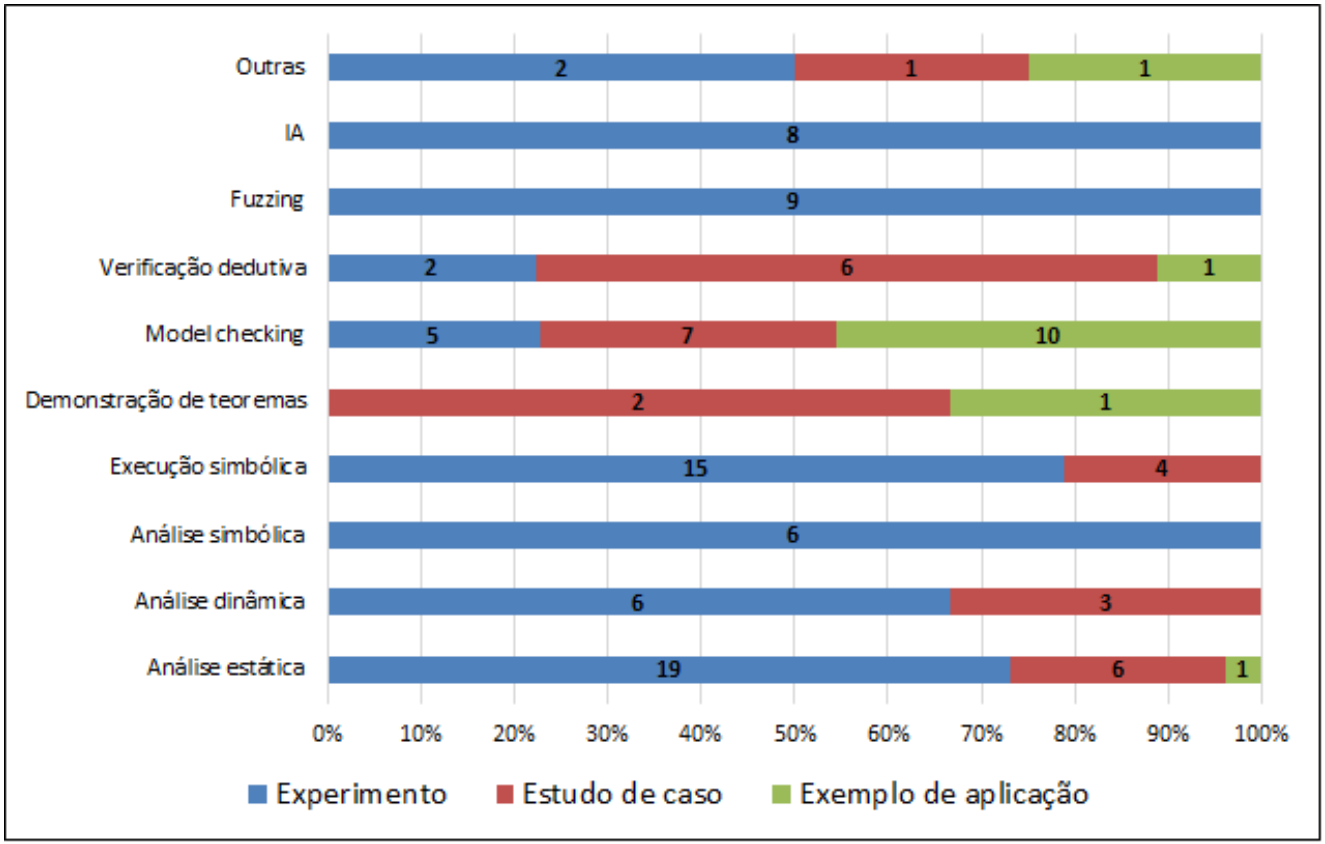
Discussão - *Model checking*



Discussão - *Model checking*

- Está entre as abordagens mais utilizadas nos últimos anos;
- **Há poucos estudos com validação experimental**;
- Reentrância foi a vulnerabilidade mais abordada;
- Violação de propriedades:
  - Amplamente abordada;
  - Principalmente *model checking*.
- Está entre as que apresentam melhor precisão e acurácia.

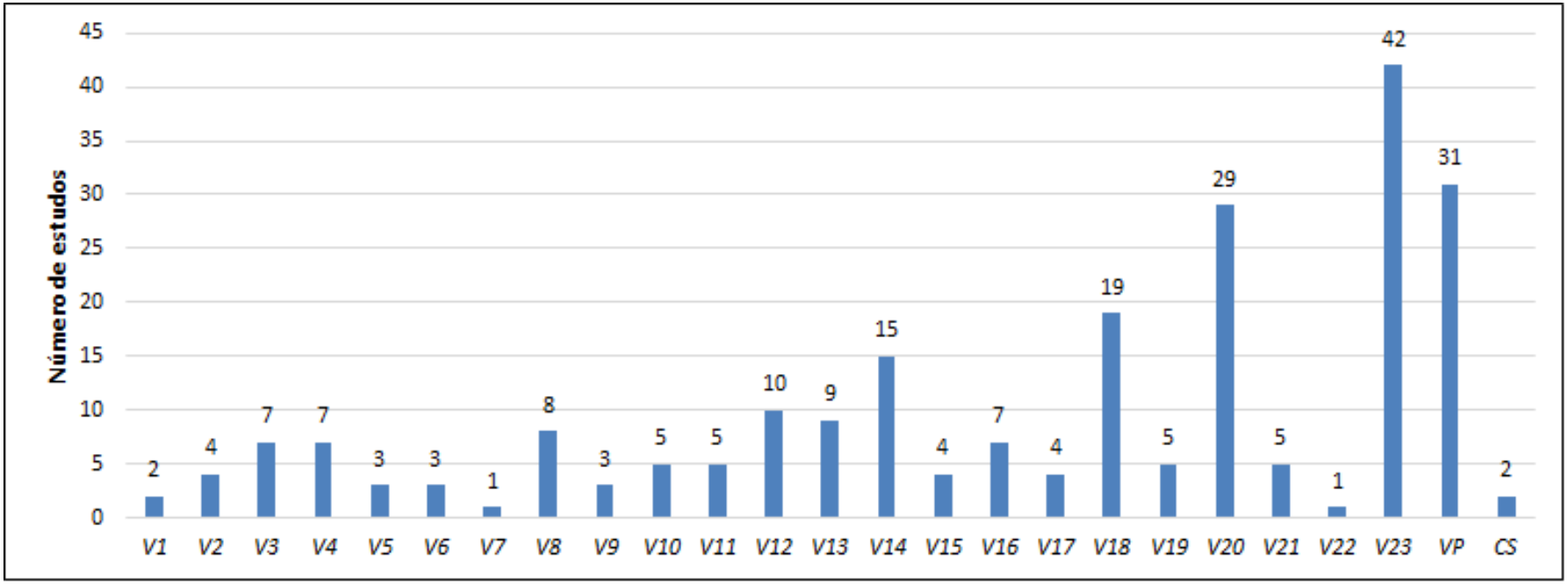
Discussão - *Model checking*



Discussão - *Model checking*

- Está entre as abordagens mais utilizadas nos últimos anos;
- Há poucos estudos com validação experimental;
- **Reentrância foi a vulnerabilidade mais abordada**;
- **Violação de propriedades**:
  - Amplamente abordada;
  - Principalmente *model checking*.
- Está entre as que apresentam melhor precisão e acurácia.

Discussão - *Model checking*



Discussão - *Model checking*

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		

Discussão - *Model checking*

- Está entre as abordagens mais utilizadas nos últimos anos;
- Há poucos estudos com validação experimental;
- Reentrância foi a vulnerabilidade mais abordada;
- Violação de propriedades:
  - Amplamente abordada;
  - Principalmente *model checking*.
- **Está entre as que apresentam melhor precisão e acurácia.**

Sumário

Objetivo

Objetivo geral

Propor uma estratégia para verificação formal para aprimoramento de segurança aplicada na fase de pré-implantação 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.

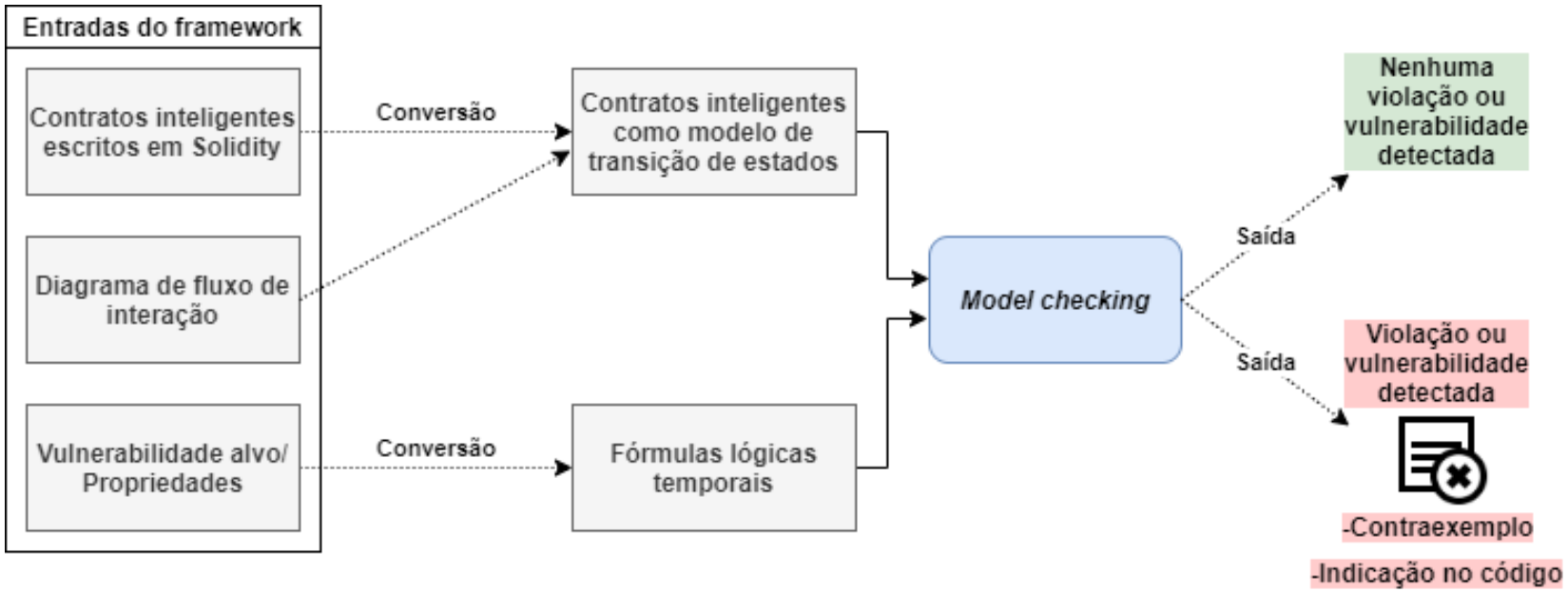
Proposta

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.

Framework proposto



Proposta - Validação

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

Experimento:

- 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.

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;
- Solidity $\rightarrow$ MEF;
- Permite a relação entre múltiplos contratos:
  - Diagrama de implementação.
- Implantação automática dos CIs gerados;
- Vulnerabilidades de reentrância e exceções não tratadas:
  - Prevenidas na construção de MEF.
  - Verificadas na modelagem ou implantação.
- Propriedades descritas por meio de modelos 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	Mai-Jun	Jul- Ago
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

06 de agosto de 2021