Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DogeFuzz: um framework extensível para estudos de fuzzing na análise dinâmica de Smart Contracts

Ismael C. Medeiros

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2023

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

DogeFuzz: um framework extensível para estudos de fuzzing na análise dinâmica de Smart Contracts

Ismael C. Medeiros

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Wilkerson Andrade (DSC - UFCG)

Prof. Dr. Genaína Nunes Rodrigues (CIC - UnB)

do Programa de Pós-graduação em Informática

Brasília, 30 de fevereiro de 2023

Dedicatória

Dedico esta tese de mestrado aos pilares do meu apoio incondicional: meus amados pais, Manoel e Vera, minha querida namorada, Agatha, e aos meus preciosos amigos, Bruno Rodrigues, Bruno Bergamaschi, Fabio, Fernando, Iure, Gabriel, entre outros.

A vocês, meus pais, que estiveram ao meu lado desde o início desta jornada acadêmica, sou imensamente grato por todo o amor, suporte e sacrifícios que fizeram por mim. Seu incentivo constante, palavras de encorajamento e crença em meu potencial foram a força que me impulsionou em direção aos meus objetivos. Sua presença, apoio emocional e exemplo de determinação e dedicação foram fundamentais para que eu pudesse alcançar essa conquista.

À minha amada Agatha, meu porto seguro e minha maior fonte de inspiração, agradeço por todo o seu amor, paciência e compreensão ao longo deste tempo. Você sempre esteve ao meu lado, oferecendo suporte emocional, encorajamento e palavras de motivação quando eu mais precisei. Sua presença em minha vida é um presente que valorizo imensamente, e sua confiança em mim é o combustível para alcançar meus sonhos.

Aos meus amigos fiéis, Bruno Rodrigues, Bruno Bergamaschi, Fabio, Fernando, Iure, Gabriel e tantos outros que me acompanharam nesta jornada, agradeço por seu apoio constante, por estarem sempre disponíveis para ouvir minhas ideias, compartilhar experiências e incentivar meu crescimento acadêmico. Suas amizades são inestimáveis e sou grato por termos construído memórias e momentos inesquecíveis juntos.

A todos vocês, pais, namorada e amigos, meu profundo agradecimento por serem a minha rede de apoio durante este tempo. Suas palavras encorajadoras, seu apoio incondicional e sua presença constante foram o suporte que me manteve firme durante os desafios e obstáculos enfrentados ao longo desta caminhada.

Esta conquista é dedicada a vocês, em reconhecimento a todo amor, confiança e apoio que têm depositado em mim. Vocês são a razão pela qual meus sonhos se tornam realidade, e sou eternamente grato por ter cada um de vocês em minha vida.

Muito obrigado!

Agradecimentos

Gostaria de expressar meus sinceros agradecimentos a todos aqueles que contribuíram de maneira significativa para a conclusão deste trabalho de mestrado.

Em primeiro lugar, agradeço aos renomados professores da Universidade de Brasília (UnB), que foram responsáveis por transmitir conhecimentos valiosos ao longo do meu percurso acadêmico. Sua dedicação, paixão pela educação e compromisso com a excelência foram inspiradores. Sou grato pelas inúmeras horas de aulas, discussões estimulantes e orientações especializadas que moldaram meu conhecimento e me ajudaram a amadurecer como pesquisador.

Meu profundo agradecimento vai ao meu orientador Rodrigo Bonifácio, cuja orientação e apoio foram cruciais durante todo o processo de pesquisa. Sua visão, orientação perspicaz e disponibilidade incansável foram fundamentais para o desenvolvimento deste trabalho. Sou grato por seu comprometimento em me guiar, encorajar meu crescimento acadêmico e me ajudar a superar obstáculos. Sua expertise e mentoria foram fundamentais para o sucesso deste estudo.

Gostaria de estender meu agradecimento aos meus chefes Alexandre, Bruno e Leonardo do trabalho, que compreenderam e apoiaram minhas responsabilidades acadêmicas. Sua flexibilidade e apoio inabalável foram essenciais para que eu pudesse equilibrar minhas obrigações profissionais com a finalização deste mestrado. Sou grato por terem confiado em minha capacidade e fornecido um ambiente favorável para meu crescimento profissional e acadêmico.

Não poderia deixar de expressar minha imensa gratidão a meus pais Manoel e Vera e à minha amada namorada Ágatha. Seu apoio incondicional, compreensão e incentivo constante foram a força motriz que me impulsionou ao longo dessa jornada. Vocês foram meu porto seguro, oferecendo suporte emocional, paciência e encorajamento. Sou profundamente grato por compartilharem minha alegria nas conquistas e por me apoiarem nos momentos desafiadores. Vocês são minha base, e sem vocês, esta jornada não teria sido possível.

Por último, mas não menos importante, agradeço aos amigos e colegas que estiveram ao meu lado, fornecendo suporte moral e intelectual. Sua presença e incentivo foram

inestimáveis e contribuíram para o enriquecimento desta pesquisa.

A todos os mencionados e a todos aqueles que, direta ou indiretamente, contribuíram para este trabalho, expresso minha mais profunda gratidão. Seus esforços, ensinamentos e apoio foram inestimáveis e deixaram uma marca indelével em minha jornada acadêmica e pessoal.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Smart contracts são programas *Turing*-completo que são executados em uma rede *Blockchain*. Muitas vezes, este tipo de programa armazena ativos digitais valiosos e em um *Blockchain* como o *Ethereum*, o *bytecode* de cada *smart contract* está público e transparente, e por isso, pode ser acessado por qualquer um. Isso faz com que este tipo de programa seja alvo constante de ataques e que a segurança de um contrato seja algo crítico. Este trabalho visa apresentar uma ferramenta flexível (chamada DogeFuzz) de forma que seja possível experimentar diferentes técnicas de *fuzzing* na análise dinâmica de *smart contracts*.

Entre as estratégias de *fuzzing* existentes, exploramos os benefícios de se utilizar as técnicas de *greybox fuzzing* e *directed greybox fuzzing* direcionada a instruções críticas de um *smart contract*. Foi mostrado, por meio de um experimento com cerca de 300 *smart contracts*, que estas técnicas conseguiram encontrar vulnerabilidades mais rapidamente que uma estratégia de *fuzzing* simples como *blackbox fuzzing*, que gera *inputs* completamente aleatórios. Além disso, concluímos que uma ferramenta de *fuzzing* necessita ter tempo para gerar uma quantidade grande de *inputs* para explorar melhoras as vulnerabilidades existentes nos *smart contracts* e que novas pesquisas em cima do DogeFuzz devem levar isto em consideração na hora de planejar o experimento. E por fim, foi feita uma análise da relação de métricas (como cobertura, tempo de execução, e instruções críticas executadas) e a detecção de vulnerabilidades, e foi notado que algumas vulnerabilidades possuíam uma certa relação com estas métricas que foram colhidas durante a execução do experimento.

Palavras-chave: blockchain, smart contracts, fuzzing, testes, instruções críticas, vulnerabilidades, arquitetura

Abstract

Smart contracts are Turing-complete programs that run on a *Blockchain* network. Often, this type of program stores valuable digital assets and on a *Blockchain* like Ethereum, the bytecode of each *smart contract* is public and transparent, and therefore can be accessed by anyone. This makes this type of program a constant target of attacks and the security of a contract is something critical. This work aims to present a flexible tool (called DogeFuzz) so that it is possible to experiment with different *fuzzing* techniques in the dynamic analysis of *smart contracts*.

Among the existing *fuzzing* strategies, we explore the benefits of using *greybox fuzzing* and *directed greybox fuzzing* directed at critical instructions of a *smart contract*. It was shown, through an experiment with about 300 *smart contracts*, that these techniques managed to find vulnerabilities faster than a simple *fuzzing* strategy like *blackbox fuzzing*, which generates *inputs* completely random. In addition, we conclude that a *fuzzing* tool needs to have time to generate a large amount of *inputs* to better exploit existing vulnerabilities in *smart contracts* and that further research on DogeFuzz should take this into account when designing the experiment. And finally, an analysis was made of the relationship between metrics (such as coverage, execution time, and critical instructions executed) and the detection of vulnerabilities, and it was noted that some vulnerabilities had a certain relationship with these metrics that were collected during execution of the experiment.

Keywords: blockchain, smart contracts, fuzzing, tests, critical instructions, vulnerabilities, architecture

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ABI Application Binary Interface.

AFL American Fuzzy Lop.

API Application Programming Interface.

BFS Breadth-First Search.

CFG Control-Flow Graph.

CSV Comma-separated Values.

CWE Common Weakness Enumeration.

DAO Decentralized Autonomous Organization.

DeFi Descentralized Finance.

EIP Ethereum Improvement Proposals.

ERC Ethereum Request for Comments.

EVM Ethereum Virtual Machine.

FIFO First in, First out.

HTTP Hypertext Transfer Protocol.

JSON JavaScript Object Notation.

NFT Non-Fungible Token.

P2P Peer-to-peer.

PC Program Counter.

PPMCC *Pearson product-moment correlation coefficient.*

REST Representational State Transfer.

RPC Remote Procedure Call.

SWC Smart Contract Weakness Classification.

WSL2 Windows Subsystem for Linux 2.

Capítulo 1

Introdução

A arquitetura *Blockchain* envolve uma rede descentralizada que tem a capacidade de validar transações monetárias entre pares não confiáveis. Isso é feito sem a necessidade de um intermediário confiável, como um banco, para validar estas transações. Além de ser utilizado em aplicações financeiras, as tecnologias que utilizam *Blockchain* vêm evoluindo a passos largos como o uso de *smart contracts* em aplicações descentralizadas.

Os *smart contracts* são programas *Turing*-completo que são executados de maneira descentralizada por nós em uma rede *Blockchain*. Toda operação feita com esse tipo de programa é registrada em *Blockchain* e por isso, é imutável e transparente. A principal rede é a do projeto *Ethereum* [1], onde foi apresentada inicialmente a ideia da execução de *smart contracts* numa rede *Blockchain* e onde estão os principais projetos envolvendo *smart contracts*.

Por ser uma tecnologia recente, muitos casos de uso ainda estão sendo descobertos e validados. Além disso, o desenvolvimento de aplicações utilizando *smart contracts* ainda não é bem consolidado pela comunidade em comparação com o desenvolvimento de software para arquiteturas mais convencionais. Muitos problemas ainda estão sendo descobertos pela comunidade e metodologias de desenvolvimento neste tipo de ambiente ainda estão sendo aperfeiçoadas. Por isso, *Blockchain* e *smart contracts* vêm sendo um alvo frequente de pesquisa tanto pela indústria quanto pela academia.

Em particular, como os *smart contracts* armazenam ativos digitais valiosos, a arquitetura tem se tornado alvo de ataques, e avanços recentes de pesquisa têm contribuído com novas técnicas e ferramentas para a identificação de vulnerabilidades em *smart contracts*.

1.1 Caracterização do Problema

Problema geral. Em uma rede *Blockchain* como a do *Ethereum*, o conteúdo compilado de cada contrato é público e pode ser analisado por qualquer interessado.

Por isso, diversos ataques tiraram proveito de alguma vulnerabilidade de um *smart contract*. Entre elas, pode-se citar a vulnerabilidade de *Dangerous Delegate Call* (SWC-112 [2]) que foi explorada pelo ataque conhecido como *Parity Wallet Hack* [3]. Este ataque causou um prejuízo de cerca de 30 milhões de dólares às carteiras da rede *Ethereum* além do prejuízo causado pela perda de confiança dos usuários nesta tecnologia—algo crítico para um sistema financeiro. Neste tipo de sistema, uma simples vulnerabilidade pode causar um grande prejuízo monetário. Por isso, é importante verificar e identificar o máximo possível de falhas antes que o contrato seja publicado e fique exposto a uma eventual vulnerabilidade.

Problema Específico. Para mitigar a presença de vulnerabilidades em *smart contracts*, a academia vem desenvolvendo diversas técnicas de detecção automática dessas falhas. Dentre elas, pode-se citar as técnicas *grammar-based fuzzing* e *mutation-based fuzzing* utilizadas pela ferramenta ContractFuzzer [4], onde, a partir de uma *seed*, o *fuzzer* gera *inputs* que seguem os formatos definidos pela especificação ABI de um *smart contract*. Esta abordagem gera *inputs* que possuem chances de serem sintaticamente válidos, por outro lado muitas vezes não conseguem explorar tão bem as camadas mais profundas e/ou críticas do programa. Ou seja, existe uma lacuna na literatura sobre as implicações potenciais com o uso de técnicas mais avançadas de *fuzzing* (como *directed greybox fuzzing*) para o domínio de *smart contracts*.

1.2 Objetivos

O principal objetivo desse trabalho é apresentar uma ferramenta extensível que possibilita a exploração de técnicas de *fuzzing* na detecção de vulnerabilidades em *smart contracts*, se baseando no trabalho feito pelo AFL++ [5]. Além disso, o objetivo secundário deste o estudo foi a exploração da técnica *directed greybox fuzzing* direcionada em instruções críticas, conceito que apresentamos neste trabalho. Este tipo de técnica visa gerar inputs que exploram caminhos que chegam mais perto de certas instruções do *bytecode* na EVM, que são consideradas críticas para o funcionamento do *smart contract*. Como será discutido na Seção 2.3, técnicas mais avançadas de *fuzzing* ainda não foram tão exploradas no domínio de *smart contracts*. E para atingir este objetivo, este trabalho planeja contribuir com:

- Construção de uma nova ferramenta que vise a extensibilidade e a reprodução para possibilitar a exploração de diversas estratégias de *fuzzing* neste estudo e em pesquisas futuras;

- Implementação de estratégias de fuzzing clássicas como *blackbox fuzzing*, *greybox fuzzing*, e *directed greybox fuzzing*;
- Construir um *dataset* de *smart contracts* para ser utilizado no experimento deste trabalho, e que também possa ser usado por outros pesquisadores. Aqui, é importante selecionar *smart contracts* que tenham vulnerabilidades conhecidas para poder analisar a capacidade de detecção de vulnerabilidades da ferramenta a partir de um *ground truth*; e
- Colher dados de *benchmark* para que o DogeFuzz possa ser analisado de forma qualitativa acerca das estratégias de *fuzzing* implementadas. Particularmente, analisaremos principalmente a estratégia de *directed greybox fuzzing* que foi apresentada neste trabalho.

Como será descrito no Cap. 3, este projeto escolheu criar uma arquitetura robusta o suficiente para seja possível estendê-la para suportar diversas estratégias *fuzzing* para *smart contracts* de maneira simples. E com essa arquitetura, implementar algumas estratégias de *fuzzing* clássicas para serem exploradas no domínio de *smart contracts*.

1.3 Hipótese

A hipótese a ser investigada nesse trabalho é que ao criar uma ferramenta extensível, seja possível explorar técnicas mais avançadas de *fuzzing*, como *greybox fuzzing* e *directed greybox fuzzing* e que se estas técnicas apresentem uma melhora na capacidade da ferramenta em explorar vulnerabilidades, ou seja, é esperado que a quantidade de vulnerabilidades encontradas aumente em comparação com os resultados obtidos por uma técnica simples de *fuzzing*, como *blackbox fuzzing*.

Além disso, conforme as estratégias de *fuzzing* fossem implementadas, verificaremos a facilidade e extensibilidade da ferramenta desenvolvida para poder verificar a capacidade do *fuzzing* de ser utilizado por outros pesquisadores, o qual foi o objetivo principal deste trabalho.

1.4 Justificativa

O tema de segurança em *smart contracts* foi escolhido por ser um alvo constante de pesquisa nos últimos anos. Os conceitos de *Blockchain* e *smart contract* são recentes, então muitos casos de uso ainda estão sendo explorados e aperfeiçoados. E, como qualquer outra tecnologia que manipule ativos digitais valiosos, é preciso ter metologias que priorizem a segurança destes tipos de programa.

Na indústria, a verificação de um *smart contract* é feita de maneira semi-manual, ou seja, um ou mais especialistas em segurança de *smart contracts* realizam uma verificação do código e tentam encontrar alguma vulnerabilidade naquele contrato. Para complementar este processo, muitas vezes se utiliza alguma ferramenta de detecção automática de vulnerabilidades, que irá encontrar possíveis falhas no contrato que está sendo analisado. Por isso, muitas pesquisas focam no aperfeiçoamento destas técnicas a fim de aumentar a confiabilidade sobre a segurança de um *smart contract*.

Capítulo 2

Revisão Teórica e Estado da Arte

Nesta seção, abordaremos o conceito de *Blockchain* e *smart contracts*, e suas vulnerabilidades mais conhecidas. Por fim, iremos apresentar o estado-da-arte em segurança de *smart contracts*.

2.1 *Bitcoin & Blockchain*

O *Bitcoin* foi apresentado por Satosh Nakamoto em um Whitepaper [6], onde descreve o conjunto que algoritmos e protocolos necessários para a criação de uma moeda que não precise da confiança em autoridades centralizadas, como bancos.

Como uma moeda digital, o Bitcoin precisa de uma estratégia que previna o problema de *Double-spending*, ou seja, é preciso monitorar a quantidade de moedas que um dono pode enviar para outras pessoas, correspondendo à quantidade de moedas que ele possuía anteriormente. Uma solução comum para este problema é introduzir uma autoridade centralizada que seja confiável para fazer esta verificação. Porém, esta abordagem faz com que o sistema monetário inteiro tenha que confiar nesta autoridade centralizada, gerando assim um ponto único de falha neste sistema.

O Bitcoin e a tecnologia de *Blockchain* foram propostos como uma solução descentralizada para o problema de *Double-spending*. Numa rede *Blockchain*, os nós se comunicaram de maneira descentralizada utilizando comunicação P2P. Os nós desta rede utilizam algoritmos de criptografia e um protocolo de consenso (intitulado *Proof-of-work*) para garantir que o histórico de transações não possa ser facilmente modificado. Adicionalmente, a arquitetura *Blockchain* usa um sistema de incentivo para que não seja rentável aos mineradores participarem de ataques em um *Blockchain*.

2.1.1 Transações

Em um *Blockchain*, as transações são armazenadas em blocos. E cada transação é transmitida para os mineradores da rede que irão validar esta transação, criar um novo bloco e transmiti-lo de volta para a rede.

Conforme ilustrado na Figura 2.1, cada bloco possui diversos dados, como as transações feitas, a data e hora de criação deste bloco, o número dele no *Blockchain*, e o *hash* do bloco anterior. Este último dado é essencial para a imutabilidade de um *Blockchain*, pois se qualquer informação de um bloco existente for modificada, seu *hash* será diferente, e consequentemente o bloco seguinte se tornará inválido pois conterá o valor anterior deste *hash*.

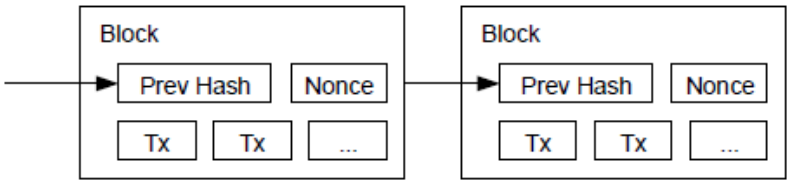

Figura 2.1: Blocos em um *Blockchain* (Fonte: [6]).

Como um sistema distribuído, blocos podem ser minerados ao mesmo tempo e a lista de blocos receberá ramificações e, por meio do algoritmo de consenso *Proof-of-work*, será definido qual ramificação é a correta.

2.1.2 Algorimo de Consenso

Para adicionar um novo bloco à rede, um minerador deverá resolver um problema criptográfico que irá consumir um grande poder computacional. Chamamos este processo de "mineração" e é utilizado pelo protocolo de consenso *Proof-of-work* para validação do *Blockchain*. O minerador deverá encontrar um valor para o campo *nonce* de forma que ao calcular o *hash* deste novo bloco se obtenha um valor que tenha um determinado prefixo, o qual é definido pela rede *Blockchain*. Este prefixo é reajustado periodicamente para que seja cada vez mais difícil encontrar um valor para o campo *nonce*.

2.1.3 Incentivo aos mineradores

Como convenção, a primeira transação em um bloco é uma transação especial que cria uma nova moeda que ficará em posse do criador deste bloco. Isso é feito para remunerar o poder computacional gasto durante a mineração e incentivar o suporte destes mineradores na rede *Blockchain*. O incentivo também é feito pela taxa paga em cada transação, onde

o criador desta transação oferecerá uma taxa para que sua transação seja processada. O valor desta taxa é variável, uma vez que influencia diretamente no tempo em que uma transação vai ser processada. Dessa forma, os mineradores priorizam as transações com taxas maiores.

2.2 *Ethereum & Smart Contracts*

O conceito de uma rede *Blockchain* foi concebido com o único objetivo de ser uma rede que processa transferências de cripto-moedas entre usuários sem precisar de uma autoridade central e confiável para mediar estas transações. Porém, além de processar transações, os componentes dessa rede foram estendidos de forma a processar outros tipos de dados, como realizar a execução de programas *Turing Complete*, chamados de *smart contracts*.

Assim, a rede *Ethereum* foi criada por Vitalik Buterin et al. [1] com o intuito de criar uma linguagem de programação *Turing*-completa que pode ser executada em cada nó de forma descentralizada no *Blockchain*. Cada participante desta rede possui uma implementação da EVM (Ethereum Virtual Machine (EVM)), máquina virtual que executa o código de um *smart contract*. A principal linguagem de programação, que é compilada para a EVM, se chama Solidity. E como pode ser visto no código abaixo, esta linguagem, baseada em JavaScript, é estaticamente tipada, suporta herança e a definição de tipos complexos utilizando *structs* [7]. Além disso, atualmente existem bibliotecas Solidity que podem ser reusadas pelos programadores de *smart contracts*.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract HelloWorld {
    string public greet = "Hello World!";
}
```

A comunidade de desenvolvimento em *Ethereum* cresceu muito nos últimos anos. Foram criados diversos tipos de aplicações com essa tecnologia e foram feitas diversas melhorias em cima do protocolo de *Blockchain*. Atualmente, a rede está passando por atualizações importantes que visam melhorias na performance e escalabilidade da rede. Por exemplo, o algoritmo de consenso principal foi alterado, de *proof-of-work* para *proof-of-stake*. Nesse algoritmo, os nós possuem poder de voto na validação de transações, onde o poder de voto é baseado na quantidade de cripto-moedas *ETH* que ele possui, ou seja, mais tem a perder caso valide uma transação maliciosa. Essa mudança visa acabar com a necessidade da mineração como prova de trabalho e o problema de consumo de energia que este tipo de operação gera em redes baseadas em *proof-of-work*. Além disso, é

possível citar que a comunidade de *Blockchain* preza bastante pelo uso de seus projetos *open-source* e isso fez com que diversos projetos de aplicações descentralizadas fossem criados pela comunidade, como:

- Descentralized Finance (DeFi): sistemas financeiros que possibilitam interação entre usuários sem a necessidade de validação de alguma entidade, como um banco ou empresa. Por exemplo: Uniswap [8] que possibilita troca de cripto-moedas entre usuários utilizando apenas suas carteiras e AAVE [9] que possibilita empréstimos de cripto-moedas entre usuários;
- Sites de apostas, sistemas que aproveitam da questão da transparência do *Blockchain* para registrar resultados das apostas;
- Decentralized Autonomous Organization (DAO), comunidades onde se utilizam *smart contracts* para votações e governança; e
- Non-Fungible Token (NFT), tokens únicos utilizados em diversas áreas, como ativos de jogos, digitalização de ativos físicos, direitos autorais e mais recentemente como representação de identidade na web.

Para participar da rede Ethereum como um nó, é preciso escolher uma implementação da EVM e executá-la. Este nó irá realizar toda a lógica de processamento de transações, execução dos *smart contracts*, interações com as aplicações, e comunicação com outros nós via comunicação Peer-to-peer (P2P). Atualmente, existem diversas implementações em diferentes linguagens de programação:

- Go-Ethereum - Geth (Go)[1]
- Nethermind (C#)
- Erigon (Go)
- Besu (Java)

Existem alguns detalhes da implementação da EVM que são essenciais para a construção de um *fuzzer*. Entender como a máquina virtual funciona, como os estados são armazenados ou como uma aplicação se comunica com o nó da rede é importante para descrever como as vulnerabilidades ocorrem nesse tipo de programa e como guiar um *fuzzer* a aproximar a execução do *smart contract* com estes casos. As próximas seções apresentam informações relevantes para uma melhor compreensão deste trabalho.

[1]Baseado no site ethernodes.org, cerca de 70% da rede principal de *Ethereum* utiliza a implementação *Geth* [10]

2.2.1 *Ethereum Virtual Machine*

Um *smart contract* possui algumas características que não se encontram em programas de computadores comuns. Estes contratos gerenciam valores que foram criados para serem utilizados em uma aplicação que vai ser executada por vários nós de uma rede *Blockchain*. Valores como endereços, *gas*, e cripto-moedas. Conforme mostrada na Figura 2.2, a estrutura interna de um EVM é formada por elementos comuns de uma máquina virtual, como *program counter*, memória e stack, mas possui elementos específicos de *Blockchain*, como a contabilização de *gas* e o armazenamento do *bytecode* dos *smart contracts*.

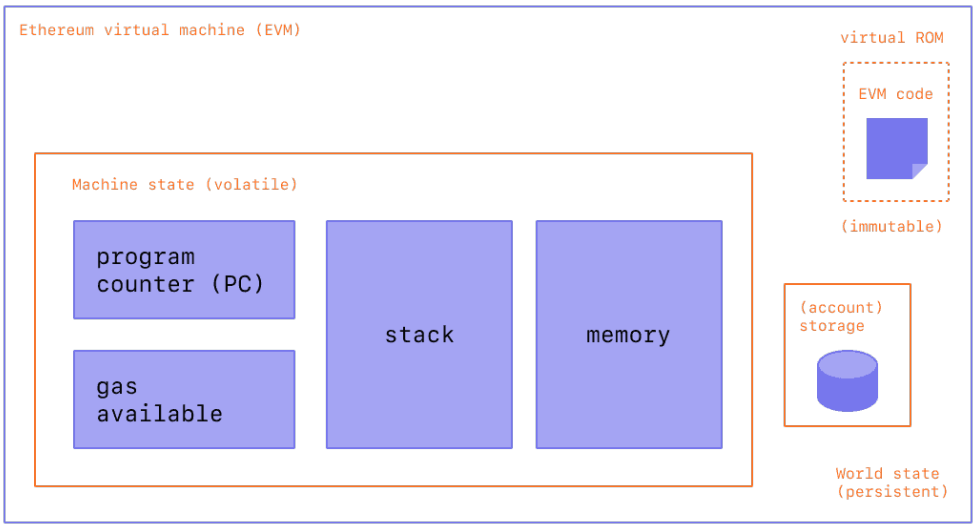

Figura 2.2: Representação da Ethereum Virtual Machine (EVM) (Fonte: [11]).

Os *smart contracts* e carteiras de usuários são tratados de maneiras semelhantes. Ambas são identificados por seus respectivos endereços na rede. O *bytecode* de um contrato é armazenado em um armazenamento global do sistema e é identificado por um endereço do *Blockchain* para ser tratado como uma carteira de um usuário comum, ou seja, pode receber e enviar cripto-moedas para outros endereços. Além disso, estes endereços podem ser manipulados a nível de código *Solidity*, conforme mostrado no código abaixo que define a assinatura do método de transferência de *tokens* que seguem o padrão ERC-20[12]:

```
interface IERC20 {
    function transfer(address recipient, uint amount) external returns (bool);
}
```

Para esta execução, a rede *Ethereum* possui um sistema de *gas*, que simboliza o poder computacional gasto em cada execução. Este conceito é necessário para dois cenários. Primeiro, para poder incentivar nós da rede a executarem o código—uma vez que recebem

uma recompensa baseada na quantidade *gas* gasto naquela execução. Além disso, cada chamada possui um limite de *gas* a ser utilizado, por isso, esse sistema previne que ocorra ataques que visam ser aproveitar do poder computacional dos nós e também prevenir que ocorram problemas de *loop* infinito na lógica do contrato.

Os protocolos que a rede *Ethereum* segue foram definidos formalmente por Gavin Wood em um *yellow paper* [13]. Neste artigo, é descrito em detalhes todas as operações que são realizadas dentro da EVM. A nível de *bytecode* é feita a definição do quanto cada operação irá custar de poder computacional (*gas*). Cada `OPCODE` possui um custo diferente, alguns são definidos estaticamente e outros dinamicamente como `OPCODE` de chamadas a outros contratos (e.g., `CALL` e `DELEGATECALL`).

Conforme visto na Figura 2.2, a EVM utiliza diversos tipos de armazenamento de estado na rede. É preciso saber a importância dessas estruturas e como são modificadas pelas transações, onde a interação com cripto-moedas é essencial para a modificação deste estado.

2.2.2 Máquina de Estado

Como mencionado acima, a Ethereum Virtual Machine (EVM) é um ambiente de *runtime* que executa *smart contracts* no *Ethereum*. Uma das principais características da EVM é a capacidade de armazenamento—que permite que *smart contracts* armazenem dados no *Blockchain*. A mudança de estado impacta diretamente no consumo de *gas* de uma chamada, e também são essenciais para o funcionamento .

O *Blockchain Ethereum* pode ser interpretado como uma máquina de estado descentralizada. Em toda transação feita a um *smart contracts*, é salvo um *snapshot* do estado atual daquele contrato. Ao fim da execução, um novo estado é persistido no *Blockchain*, e aquele *snapshot* é atualizado. Caso ocorra algum erro durante a execução, o *snapshot* do estado inicial é restaurado e o estado permanece inalterado. Uma chamada a um *smart contract* pode ser visto como a seguinte formula:

$$F(S, T) = S'$$

Onde,

- S é o estado atual (*snapshot*) antes da execução;
- S' é o novo estado gerado depois da execução do *smart contract*;
- T é transação realizada para chamar a função do *smart contract*; e
- $F(S, T)$ é função executada no *smart contract*

Durante a execução de um *smart contract*, o estado pode ser armazenado de três maneiras. Por exemplo, para armazenar o código dos contratos, contas e balanços, o *Blockchain* utiliza um armazenamento global que contém todos os endereços e contratos publicados naquela rede. A EVM carrega o código de execução de um *smart contract* no momento que ele é chamado. Para armazenar o estado que persiste entre chamadas daquele *Blockchain*, a EVM utiliza um armazenamento de chave-valor chamado *Storage* que são acessados utilizando os `OPCODEs`: `SSTORE` e `SLOAD`. E, por fim, para armazenar o estado transiente, ou seja, armazenar estado que será utilizado apenas no escopo da função que será utilizada, a EVM utiliza a estrutura chamada *Memory*. A EVM realiza operações com estes dados utilizando os `OPCODEs` `MSTORE` e `MLOAD`. Por padrão, as instruções que manipulam o *Storage* consomem mais *gas* pois manipulam o estado do *smart contract* e pode influenciar seu comportamento. Por essa razão, os validadores são recompensados proporcionalmente.

Muitos dados sensíveis são armazenados na estrutura de *storage*, ou seja, estes dados persistem no *Blockchain* e podem ditar o comportamento futuro daquele *smart contract*. Esse tipo de estrutura armazena dados importantes da lógica de execução do *smart contract*. Por exemplo, um *smart contract* de um *token* ERC-20 [12] pode armazenar nessa estrutura os endereços de cada usuário que possui alguma quantidade de *token* na conta dele. Neste exemplo, um atacante pode se aproveitar de um mal funcionamento do contrato e modificar seu próprio balanço de *tokens* de forma maliciosa.

O entendimento de como um *smart contract* armazena seus dados é essencial para a análise de vulnerabilidades e como utilizam estas características para explorar comportamentos excepcionais. Toda mudança de estado (no contrato ou no *Blockchain*) gera uma transação que é armazenado de maneira imutável pela rede, e, por isso, está atrelado diretamente as formas de interações que aplicações externas podem se comunicar com o contrato e como esta aplicação irá criar esta transação.

2.2.3 Interação entre Aplicações e *smart contracts*

Uma aplicação externa se comunica com o *Blockchain* por meio de uma biblioteca cliente. Esta biblioteca realiza algumas operações necessárias para que a comunicação com o *smart contract* seja feita maneira correta com o nó do *Blockchain*.

Para operações que mudam o estado do *smart contract*, a biblioteca deve enviar transações convencionais (de transferências de cripto-moedas) para um *smart contract* com informações adicionais importantes para sua execução. Estas transações devem conter um valor em cripto-moedas que será transferido, sendo aceitável transferir o valor *zero* na transação. Porém, quando se cria um *smart contract*, é possível especificar que um

método só será executado se receber um valor monetário naquela transação. Isso é feito com a palavra-chave **payable**, como pode ser visto no programa abaixo:

```
contract MyBank {
    mapping(address => uint256) deposits;

    function deposit() public payable {
      deposits[msg.sender] += msg.value;
    }
}
```

Neste tipo de método, o objeto `msg` é preenchido, o qual contém a propriedade `msg.value` que armazena o valor transferido para aquele *smart contract* e a propriedade **msg.sender** que armazena o endereço de qual conta enviou aquela transação. Com estes dados, os *smart contracts* podem interagir com as contas que estão se comunicando com eles.

Dentro do conteúdo da transação, são codificadas todas as informações necessárias para execução do *smart contract*. As bibliotecas clientes utilizam a especificação ABI que é disponibilizada no momento da compilação do *smart contract* para realizar a criação da requisição. Esta especificação possui metadados acerca do contrato, como nomes dos métodos, tipos dos parâmetros desses métodos, e informações relacionadas à mudança de estado. Esta especificação é essencial para mapear os parâmetros corretos aos componentes de uma aplicação externa. No momento em que a transação é criada, estes parâmetros são codificados segundo a especificação e enviados no formato JSON-RPC para ser decodificado por um nó do *Blockchain*.

A especificação ABI é útil quando se quer executar algum método específico do *smart contract*, mas existem outro tipo de interação com o contrato. É possível transferir dinheiro para um contrato como se fosse um conta normal. Porém, o contrato deve implementar um dos métodos obrigatórios para receber transferências (`receive()` e `fallback()`):

```
contract MyBank {
    mapping(address => uint256) deposits;

    fallback() external payable {
      deposits[msg.sender] += msg.value;
    }

    receive() external payable {
      deposits[msg.sender] += msg.value;
    }
}
```

Estes métodos devem conter a palavra-chave `payable` por manipularem os metadados de quem enviou a transferência utilizando as propriedades `msg.value` e `msg.sender`. Note que a única diferença entre os métodos `fallback()` e `receive()` é que o método `fallback()` pode receber dados codificados por meio da propriedade `msg.data`.

Com isso, é possível entender como as aplicações externas se comunicam com um *smart contract*, mas este tipo de programa pode realizar outro tipo de comunicação. Contratos podem ser comunicar com outros *smart contracts* e isso pode levar a diversos complicações que impactam a segurança deste tipo de programa.

2.2.4 Interação entre *smart contracts*

Os *smart contracts* podem se comunicar de maneira simples dentro de um *Blockchain*. Como todos os contratos possuem uma especificação ABI, a comunicação entre eles podem ser feita por meio de RPC podendo chamar diretamente métodos de outros contratos. Também é possível programar um contrato para transferir cripto-moedas para outro contrato, e o contrato alvo irá tratar essa transferência da mesma maneira como é feito caso um usuário ou aplicação transfira dinheiro para aquele contrato.

Para realizar a interação entre contratos como um RPC, o contrato utiliza o método **call()** do tipo **address** para realizar a chamada ao contrato. Note que é preciso saber previamente o endereço do contrato alvo, o nome do método, e seus argumentos codificados para realizar a chamada. Utilizando este método, o *gas* será gasto no outro contrato até que o limite definido naquela transação seja ultrapassado. Um exemplo de chamada RPC pode ser visto no código abaixo:

```
contract ContractCaller {
    function CallContract(
        address contractAddress,
        bytes calldata data
    ) public payable {
        (success, ) = contractAddress.call{value: msg.value}(data);
        assert(success);
    }
}
```

Outra forma de comunicação é a transferência de cripto-moedas entre contratos. Para isso, é possível utilizar os métodos `transfer()` e `send()` para transferir cripto-moedas para outro contrato ou usuário, passando o endereço alvo e o valor a ser transferido. O contrato que não possui o valor irá causar um erro. Estes métodos possui um limite de *gas* de 2300, e depois que a execução passar esse limite, um erro será lançado. Um exemplo de transferência pode ser visto no contrato abaixo:

```
contract ContractTransfer {
    function TransferToContract(address payable to) public payable {
        to.transfer(msg.value);
    }

    function SendToContract(address payable to) public payable {
        bool success = to.send(msg.value);
        assert(success);
    }
}
```

Como pode ser visto nos contratos apresentados acima, cada um destes métodos possui uma forma de checar se houve algum erro durante a chamada àquele contrato. Em caso de o contrato chamado lançar algum erro, o não-tratamento explicito do retorno da chamada pode causar comportamentos inesperados, podendo ser explorados por atacantes.

Com diversas formas de interação disponíveis, o desenvolvimento de *smart contracts* deve ser feito de maneira minuciosa para que estas chamadas não possibilitem que ataques sejam feitos. Para isso, alguns padrões foram definidos para que leve a um desenvolvimento correto e seguro dos contratos.

2.2.5 Padronizações EIPs e ERCs

A comunidade do *Ethereum* está em constante expansão e a plataforma está se desenvolvendo cada vez mais. Para padronizar as melhorias da rede, a comunidade criou os EIPs, que são documentos que definem propostas de melhoria da rede *Ethereum*. Estes documentos passam por diversos níveis de validação e revisão feitos pela comunidade antes que sejam de fato utilizados em aplicações reais.

Estas propostas podem ser a nível de aplicação, como os ERCs que definem interfaces que padronizam certos tipos de *smart contracts* (e.g. ERC-20 [12] para *tokens* e ERC-721 [14] para NFTs), e também a nível de protocolo, como os EIP-1559 [15] e EIP-2982 [16] que compõem propostas de melhorias para a rede *Ethereum*.

Uma proposta importante para a análise de vulnerabilidades em *smart contracts* é a EIP-1470 [17] que define um esquema de catalogação de vulnerabilidades em *smart contract* da rede *Ethereum*, chamado de SWC, *Smart Contract Weakness Classification*. A ideia se baseia na popular lista de vulnerabilidades CWE, *Common Weakness Enumeration*, que é comumente utilizada pela comunidade de segurança [18].

Neste trabalho, o padrão EIP-1470 será utilizada como referência para identificar comportamentos incomuns em contratos, onde cada vulnerabilidade é identificada com base em padrões já identificados em contratos com aquela vulnerabilidade. Com isso, é

Título	SWC Relacionado
Transaction Order Dependence	SWC-114
Timestamp Dependence	SWC-120
Mishandled Exceptions	SWC-113
Reentrancy	SWC-107

Tabela 2.1: Vulnerabilidades Detectadas pela Ferramenta OYENTE

possível compreender como outros estudos sobre a segurança de *smart contracts* foram feitos e quais vulnerabilidades foram mais exploradas nestes estudos.

2.3 Segurança em *smart contracts*

A pesquisa para mitigar vulnerabilidades em *smart contracts* tem explorado diferentes abordagens, incluindo análise dinâmica (e.g., *fuzzing* e análise do histórico de transações), análise estática (e.g., execução simbólica, verificação formal, e *model checking*) e *deep learning* para analisar *smart contracts*. Por exemplo, Luu et al. [19] apresentaram uma ferramenta chamada OYENTE, que utiliza execução simbólica para encontrar potenciais *bugs* de segurança em *smart contracts*. Esta ferramenta foi desenvolvida para detectar algumas vulnerabilidades específicas, como é mostrado na Tabela 2.1. De um dataset de 19,366 contratos, a ferramenta OYENTE conseguiu sinalizar 8,833 contratos como vulneráveis (cerca de 46%), ou seja, encontrou algum indício de que pelo menos uma das vulnerabilidades da Tabela 2.1 existia nos *smart contracts*.

O estudo de Luu et al. trouxe um impacto significativo, sendo citado por diversos outros trabalhos de pesquisa. Devido a tal impacto, utilizamos este trabalho como referência principal para encontrar outros estudos que utilizaremos como a base teórica desta pesquisa (seguindo a estratégia de *snowballing*). Para escolhermos os artigos mais relevantes, fizemos um estudo exploratório das ferramentas de detecção de vulnerabilidades em *smart contract*, selecionadas seguindo os seguintes critérios:

- Citou o artigo da ferramenta OYENTE
- Publicado em um veículo relevante (*Journal* ou conferência A*)
- Apresentou uma nova ferramenta

Ou seja, seguindo uma estratégia de *snowballing*, primeiro filtramos os artigos que citaram o artigo de Luu et al. [19]. Segundo, selecionamos apenas os artigos que foram publicados em *journals* ou conferências consideradas de maior relevância, que são classificadas como A* no Ranking Core [20]. Por fim, escolhemos apenas os artigos que

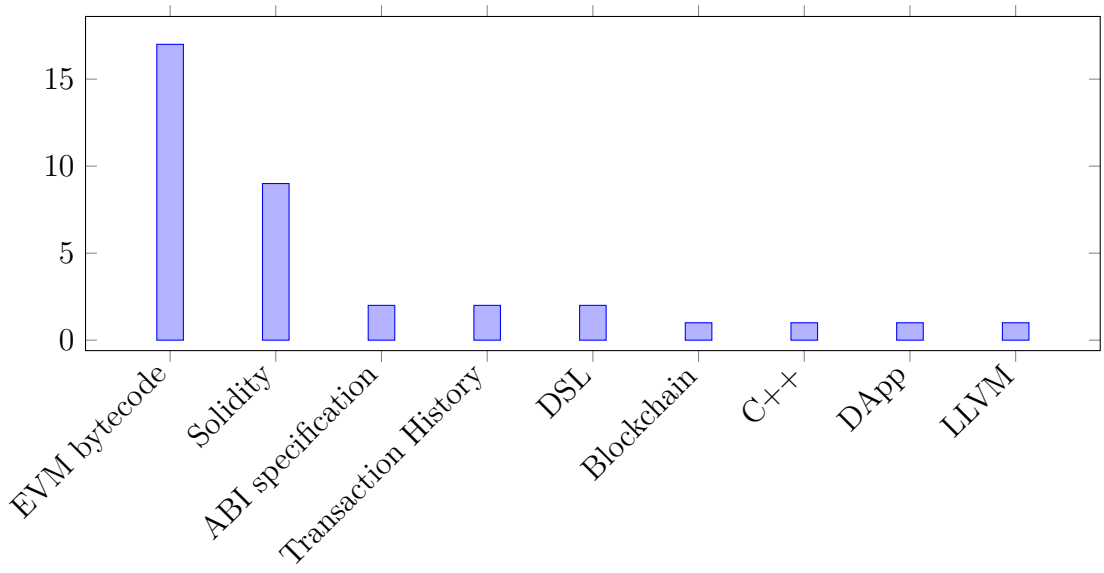

Figura 2.3: Domínios das Análises

apresentaram uma nova ferramenta, para excluir artigos de outros estudos exploratórios do estado-da-arte. Com estes filtros, fizemos um levantamento de 33 ferramentas que podem ser vistas na Tabela 2.2. Esta tabela mostra o tipo de análise que cada ferramenta utilizou e em qual domínio foi feita esta análise. Além disso, organizamos a tabela pelo número de citações para priorizarmos os artigos mais citados em nosso estudo.

Com este estudo exploratório, podemos extrair algumas informações sobre o estado-da-arte na exploração de vulnerabilidades em *smart contracts*, como os domínios abordados pelas ferramentas de análise, as vulnerabilidades exploradas, e tipos de análise que foram feitos.

2.3.1 Domínios da Análise

Como pode ser visto na Figura 2.3, podemos destacar uma predominância da análise do *bytecode* dos *smart contracts* e do código *Solidity* deste contrato. O alto número de pesquisas que analisam o *bytecode* pode ser explicado pelo fato de que os *bytecodes* de todos os *smart contracts* publicados até hoje estão disponíveis publicamente no *Blockchain* do *Ethereum* e podem ser facilmente extraídos de lá. E somente parte do código fonte dos *smart contracts* está disponível publicamente, que mostra uma tendência dos desenvolvedores de escolherem segurança por obscurantismo para proteger seus *smart contracts*. Além disso, podemos destacar a análise da especificação ABI que é utilizada pelas aplicações para se comunicar com um *smart contract*. Nesta especificação, existem as informações sobre os métodos públicos deste *smart contract* que poderão ser chamados por aplicações externas.

#	Artigo	Tipo de Análise	Domínio
1	Kalra et al. [21]	Estática	LLVM
2	Jiang et al. [4]	Dinâmica	ABI specification
3	Krupp e Rossow [22]	Estática	EVM bytecode
4	Liu et al. [23]	Dinâmica	C++
5	Grishchenko et al. [24]	Estática	EVM bytecode
6	Permenev et al. [25]	Estática	EVM bytecode
7	Coblenz [26]	Estática	DSL
8	Mossberg et al. [27]	Estática	EVM bytecode
9	Grech et al. [28]	Estática	EVM bytecode
10	Liu et al. [29]	Estática	Solidity
11	Wang et al. [30]	Dinâmica	EVM bytecode
12	Nguyen et al. [31]	Dinâmica	EVM bytecode
13	Gao et al. [32]	Estática	Solidity
14	Frank et al. [33]	Estática	EVM bytecode
15	Gao et al. [34]	Estática	EVM bytecode
16	Bhargavan et al. [35]	Estática	Solidity
17	Sunbeom et al. [36]	Estática	Solidity
18	Wustholz e Christakis [37]	Dinâmica e Estática	EVM bytecode
19	Wang et al. [38]	Dinâmica	EVM bytecode
20	Zhang et al. [39]	Dinâmica	Transaction History
21	Park et al. [40]	Estática	EVM bytecode
22	Zhang et al. [41]	Estática	Solidity
23	Almashaqbeh et al. [42]	Estática	Blockchain Design
24	Chen et al. [43]	Estática	EVM bytecode
25	Xue et al. [44]	Estática	Solidity
26	Feng et al. [45]	Estática	DSL
27	Yang et al. [46]	Estática	EVM bytecode
28	Li [47]	Estática	EVM bytecode
29	Wang et al. [48]	Dinâmica	Transaction History
30	Gao [49]	Estática	Solidity
31	Chen [50]	Estática	Solidity
32	So et al. [51]	Estática	Solidity
33	Gao [52]	Dinâmica	Application

Tabela 2.2: Ferramentas de Detecção de Vulnerabilidades em *smart contracts*.

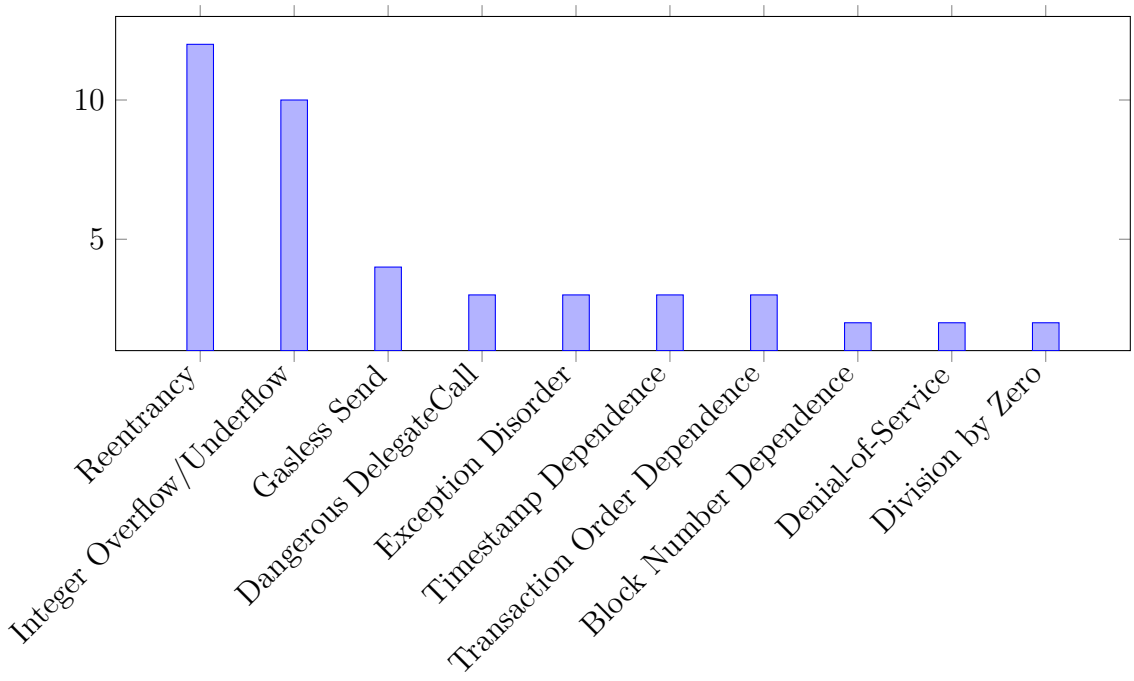

Figura 2.4: Ranking das Vulnerabilidades Exploradas

2.3.2 Vulnerabilidades Exploradas

A maior parte das ferramentas identificadas na revisão de literatura exploram um conjunto restrito de vulnerabilidades. Como pode ser visto na Tabela 2.3, algumas vulnerabilidades são conhecidas na comunidade e já foram exploradas por algum atacante, possuindo registros em SWC. A Figura 2.4 mostra o ranking de vulnerabilidades exploradas. Entre elas, podemos destacar a vulnerabilidade *Dangerous Delegate Call*, que foi explorada no famoso ataque *Parity Wallet Hack* [3] que resultou em milhões de dólares em perdas, mas não foi tão explorado pela literatura.

A análise de vulnerabilidades em *smart contracts* vem sendo feito utilizando diferentes técnicas. Podemos citar a técnica de *fuzzing* que analisa o contrato de forma dinâmica, ou seja, durante a sua execução em um *Blockchain*. Esta técnica vem sido explorada por alguns pesquisadores e trouxe avanços na área de segurança em *smart contracts*. Além disso, esta área chamou a atenção do grupo de pesquisa deste estudo por interesse dos pesquisadores em explorar técnicas de análise em *smart contracts*.

2.3.3 Tipos de Análise

A partir das informações colhidos, podemos fazer um levantamento de quais técnicas de análise de programas foram utilizadas. E como pode ser visto na Figura 2.5, a maior parte dos estudos utilizam alguma técnica de análise estática. Isso mostra uma tendência da

Título	SWC Relacionado
Reentrancy	SWC-107
Integer Overflow/Underflow	SWC-101
Gasless Send	SWC-126
Dangerous DelegateCall	SWC-112
Exception Disorder	SWC-113
Timestamp Dependence	SWC-120
Transaction Order Dependence	SWC-114
Block Number Dependence	SWC-120
Denial-of-Service	SWC-128 e SWC-113
Division by Zero	

Tabela 2.3: Vulnerabilidades Exploradas

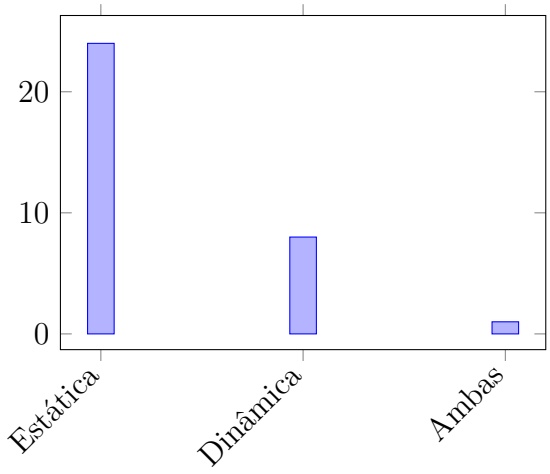

Figura 2.5: Tipos de Análise Utilizadas nos Estudos

academia para explorar este tipo de análise. As técnicas de análise dinâmica não foram tão exploradas neste domínio de *smart contracts*, o que leva a novas oportunidades de pesquisa nesta área.

Dentre as técnicas de análise dinâmica, podemos destacar *fuzzing*, que é a técnica de geração de casos de teste para a exploração do comportamento de um programa. Esta técnica vem sendo utilizada na exploração de vulnerabilidades em *smart contracts* em alguns estudos, como Jiang et al. [4], Liu et al. [23] e Nguyen et al. [31].

2.4 *Fuzzing*

O conceito de *Fuzzing* foi apresentado pelo professor Barton Miller da University of Wisconsin-Madison, onde ele propôs o seguinte exercício: Cada aluno deveria construir um programa que gere um *stream* de caracteres aleatórios que seriam utilizados como *in-*

put de um programa utilitário do UNIX. A ideia do exercício foi tentar atacar o máximo possível de programas utilitários e testar resiliência destes programas a *inputs* incomuns. Como resultado, seus alunos conseguiram quebrar 25-33% dos utilitários testados [53]. Apesar de ser um programa simples de ser implementado, *fuzzing* se mostrou uma ferramenta bastante eficaz na exploração de comportamentos atípicos de um programa, sendo útil na exploração de vulnerabilidades em programas.

A pesquisa em *Fuzzing* vem evoluindo constantemente, gerando diversas variações como *grammar-based*, *mutation-based* e *greybox fuzzing*. Estas variações apresentaram contribuições positivas para esta área. E nas próximas sessões, iremos descrever cada uma destas variações e para resolver quais problemas elas foram criadas.

2.4.1 *Grammar & Mutation-based Fuzzing*

Inputs gerados automaticamente muitas vezes são sintaticamente inválidos e podem ser facilmente rejeitados pelos programas. Como o objetivo do *fuzzing* é explorar as partes mais profundas do programa e não só a parte que processa e valida os *inputs*, é preciso aumentar as chances desse *input* ser sintaticamente válido. Para resolver este problema, as técnicas como *grammar-based fuzzing* e *mutation-based* foram criadas.

A técnica de *grammar-based fuzzing* foi criada para testar programas que exigiam um *input* com formato complexo, como visualizadores de PDF e compiladores. Esta técnica recebe como parâmetro uma especificação no formato de dicionário de como um *input* deve ser gerado para satisfazer os pré-requisitos do programa. Assim, o *fuzzer* conseguirá gerar inputs que se aproximem aos inputs que o programa está esperando. Contudo, o uso de gramáticas para testar programas vem sido feito antes mesmo do conceito de *fuzzing* ser apresentado. Esta técnica foi primeiro explorada por Burkhardt [54] em 1967.

A técnica de *mutation-based fuzzing* consiste em utilizar um *input* inicial, o *seed*, que possui um formato sintaticamente válido para o programa que está sendo testado. E então, este *input* sofrerá uma pequena mutação que gerará um novo *input* semelhante. Este novo valor será usado para testar o programa e depois sofrerá uma outra mutação, gerando um novo *input* e assim, explorará o programa fazendo estas pequenas mutações no *seed* atual. A premissa desta técnica é que a chance de um *input*, que foi gerado a partir de uma pequena modificação de um *input* válido, ser válido é maior do que simplesmente gerar um *input* aleatoriamente. Esta técnica foi utilizada principalmente pela ferramenta AFL [55] que será descrita também como uma ferramenta de *greybox fuzzer* a seguir.

Além disso, é importante citar que muitas vezes, um *fuzzer* pode utilizar a combinação das técnicas de *mutation-based* e *grammar-based fuzzing*, se aproveitando dos benefícios de cada uma.

2.4.2 *Greybox Fuzzing*

Nas sessões anteriores, apresentamos técnicas de geração de teste blackbox, ou seja, não faz nenhuma análise sobre o conteúdo do programa. Este tipo de análise possui algumas vantagens como simplicidade e velocidade de execução, porém apresenta algumas limitações na exploração do programa. A probabilidade de explorar caminhos mais profundos do programa é pequena quando se utiliza este tipo de análise.

Em contrapartida, existem as técnicas *whitebox* que envolvem uma análise completa sobre o programa. Este tipo de análise é bastante poderoso, mas pode consumir bastante tempo na sua execução. Quando um programa possui muitas *branches*, o tempo de execução cresce exponencialmente. Este problema é chamado de *path explosion*.

Uma ferramenta *greybox*, como *greybox fuzzing*, se aproveita da velocidade de uma ferramenta *blackbox* e também de uma análise do programa mais profunda que é feita por uma ferramenta *whitebox*. Neste tipo de ferramenta, o programa é analisado de maneira simples e utilizamos o resultado desta análise como parâmetro na escolha dos novos *inputs* para testar o programa mais profundamente. Uma das ferramentas *greybox* mais utilizadas é o AFL [55], que serviu como base para diversos outros estudos nesta área de exploração de vulnerabilidades.

Muitas ferramentas analisam a cobertura do código atingida por determinado *input* para priorizar a mutação de *inputs* que mais exploraram o programa. Para isso, foi introduzido o conceito de *power schedule* por Böhme et al. [56]. Um *power schedule* distribui o tempo em que o *fuzzer* será executado entre os seeds do *fuzzer*. Aqui, chamamos a chance de um *seed* ser escolhido como "enegia" deste *seed*. Então, devemos decidir apenas um método responsável por definir a energia do *seed*. Ferramentas utilizaram estratégias como atribuir mais energia para as *seeds* que são menores, que executam mais rápido, ou que causam um aumento na cobertura mais frequentemente.

Além disso, existem algumas variações de *Greybox Fuzzing* como *Boosted Greybox Fuzzing* e *Directed Greybox Fuzzing*. O *Boosted Greybox Fuzzing* possui um método para atribuição de energia de uma *seed* onde as *seeds* que explorarem caminhos dos programas incomuns terão mais energia. A probabilidade de cada caminho é computada a partir da quantidade de vezes que o caminho foi executado por *inputs* anteriores. A ferramenta AFLFast [56] utiliza esta estratégia de *Boosted Greybox Fuzzing* para atribuir energia aos seus *seeds* durante a execução.

O *Directed Greybox Fuzzing*, que foi apresentado por Böhmer et al. [57] com a ferramenta AFLGo, possui uma metodologia de *power schedule* diferente das outras. Este tipo de *fuzzing* foca em atingir pontos que são considerados críticos ao programa. A função de *power schedule* irá atribuir mais energia às *seeds* que se aproximaram mais destes pontos críticos. Para isto, precisamos computar esta distância utilizando um grafo de fluxo de

controle computado por uma análise estática prévia do programa. Em mais detalhes, primeiro, geramos um *call graph* do programa e para deixar o *fuzzer* mais eficiente, já pré-calculamos as distâncias entre os métodos. Além disso, instrumentamos as chamadas aos métodos para computarmos dinamicamente quais métodos foram cobertos na execução do *fuzzer*. Ao fim de cada execução, computamos o caminho percorrido e a menor distância média dos pontos críticos e com estes valores, podemos calcular a energia da *seed* e continuarmos a execução do *fuzzer*.

2.5 Estado-da-arte em Fuzzing

Tanto na indústria quanto academia, o uso de *fuzzer* vem crescendo bastante nos últimos anos. Muitas técnicas e ferramentas foram criadas e aperfeiçoadas. Entre elas podemos citar algumas que apresentaram contribuições importantes para esta área.

2.5.1 AFL

A ferramenta AFL [55] é um *greybox fuzzing* baseado em cobertura e teve resultados bastante promissores. Este *fuzzer* utiliza uma abordagem *mutation-based* com diversos tipos de mutação do *seed* e utiliza a cobertura de código para definir a energia destes *seeds*. Esta cobertura é computada por um instrumentação simples das *branches* do código. Isso faz com que o *fuzzer* seja mais eficiente e tenha um *overhead* menor na execução do programa em teste.

2.5.2 AFL++

Com a popularização do projeto AFL, a comunidade *open-source* criou o projeto AFL++ [5]. O projeto tem como principal melhoria a extensibilidade, que possibilitou que outros desenvolvedores a adicionares novas estratégias de *fuzzing* e novas integrações de maneira mais simples. Isso foi atingido pela modularização de pontos utilizados em outras estratégias de *fuzzing*, como a API de mutação ou de geração de *seeds*.

2.5.3 AFLFast e AFLGo

Muitos outros trabalhos de pesquisa utilizaram os projetos AFL e AFL++ como base e implementaram algum extensão para a ferramenta, como o AFLFast e o AFLGo.

Foi observado que o AFL acessava caminhos do programas frequentes e não conseguia explorar o programa mais profundamente. Com isso, a ferramenta AFLFast [56] extraiu o conceito de *power schedule* do AFL e ajustou ele para explorar caminhos acessados

menos frequentemente. As *seeds* que exploravam caminhos menos frequentes recebiam mais energia no *power schedule* e prioridade na execução.

Já o AFLGo [57] apresentou a abordagem de *Directed Greybox Fuzzing*, que foi descrita na seção 2.4.2. A ferramenta foi desenvolvida para reservar o seu tempo de execução em partes especificas do programa sem gastar recursos em componentes não importantes. Os alvos mais interessantes para este tipo de abordagem foram: testes sobre um novo *patch* do programa, reprodução de um *crash* e detecção de *sinks* no programa.

Capítulo 3

DogeFuzz

Conforme discutido na Seção 2.3, existe ainda uma lacuna na literatura onde poucos trabalhos exploram técnicas de análise dinâmica para encontrar vulnerabilidades em *smart contracts*, em comparação com aqueles abordagens que utilizaram análise estática. Com a revisão de literatura conduzida, também foi possível observar que recentes avanços na área de *fuzzing*, os quais foram descritos na sessão 2.5, ainda não foram explorados no domínio de *smart contracts*. Ou seja, técnicas que obtiveram sucesso nos testes em outros tipos programas ainda não foram exploradas neste domínio. Com o intuito de suprir essas lacunas, esta pesquisa tem como principal objetivo explorar técnicas mais avançadas de *fuzzing* para a análise de vulnerabilidades em *smart contracts*. Inspirado no trabalho do AFL++, contribuímos com uma arquitetura extensível (DogeFuzz) que possibilita a integração de diferentes estratégias de *fuzzing*. Com isso, podemos experimentar diferentes alternativas de *fuzzing*, como *greybox fuzzing* e *directed greybox fuzzing* para identificar vulnerabilidades em *smart contracts*. Este resultado de engenharia foi fundamental para a condução dos experimentos descritos no próximo capítulo, bem como está abrindo novas possibilidades de pesquisas na área de segurança de *smart contracts* na UnB. Este capítulo se concentra na descrição das decisões arquiteturalmente relevantes do DogeFuzz bem como algumas decisões de implementação.

3.1 Princípios de Design Envolvidos no DogeFuzz

Para realizar uma pesquisa que possa ser reproduzível e extensível, deve ser necessário identificar as características que um *fuzzer* deve ter para que possibilite o trabalho de outros pesquisadores no projeto.

Seguindo padrões da indústria, onde a reprodutibilidade é essencial para o desenvolvimento de projetos, é preciso ter um *script* de execução bem definido e de fácil execução. Por isso, foi definido que o DogeFuzz deve depender de *containers* para ser executado.

Desta forma, trazemos os benefícios de um virtualização leve, onde o tamanho da imagem do *container* será mínima.

Além da facilidade de execução, o DogeFuzz deve ter um arquitetura configurável, para que os programas sejam executados de maneiras diferentes e com diferentes estratégias. Por isso, o projeto deve ser modularizado nos pontos essenciais em que uma extensão é possível em uma estratégia de *fuzzing*. Então, foi feita uma análise sobre as estratégias de fuzzing descritas em 2.5 para que fossem identificados os pontos em que o *fuzzer* pudesse ser estendido, como a escolha dos *seeds*, a escolha da estratégia onde a cobertura será calculada e outros pontos que serão descritos nas próximas seções.

Tendo uma arquitetura com as características descritas acima, deve ser possível investigar os usos de diferentes estratégias de *fuzzing*. Com isso, poderemos investigar estes usos e identificar quais apresentam uma melhor performance (em termos de acurácia, tempo necessário para identificar vulnerabilidades etc.).

3.2 Estratégias de *Fuzzing*

Para investigar o uso de *fuzzing*, foram implementadas diferentes estratégias para a geração de inputs para *smart contracts*. Com isso, poderemos analisar quais estratégias são mais eficazes em encontrar vulnerabilidades nos contratos que serão testados.

As estratégias escolhidas foram extraídas de *fuzzers* já consolidados que trouxeram conceitos importantes para o estudo de *fuzzing*. Por isso, um estudo foi feito em cima das implementações feitas pela ferramentas AFL, AFL++, AFLFast, e AFLGo; além das suas respectivas documentações. O intuito desse estudo foi colher as decisões feitas nestes *fuzzers* em seus usos e como estas decisões poderiam ajudar na implementação do DogeFuzz, um *fuzzer* customizado para as particularidades de *smart contracts*. Além disso, foi implementada uma estratégia básica de *fuzzing* para servidor de comparação com as estratégias utilizadas nestas ferramentas.

As estratégias *greybox fuzzing* e *directed greybox fuzzing* foram escolhidas pare implementação na primeira versão do DogeFuzz. Particularmente, exploramos a estratégia de *directed greybox fuzzing* que guia o *fuzzer* a explorar caminhos que levam a pontos específicos do código (identificados por instruções críticas). Como será investigada a relação da execução de instruções críticas e vulnerabilidades encontradas, esta estratégia de *fuzzing* se mostrou útil conforme mostrado no trabalho apresentado com a ferramenta AFLGo [57].

3.3 Arquitetura Proposta

Para atingir os objetivos propostos nesta pesquisa, o desenho arquitetural do DogeFuzz precisa atender aos requisitos descritos na sessão anterior. Esta seção apresenta detalhes sobre esta arquitetura, desde as decisões mais gerais feitas sobre a arquitetura até detalhes de implementação do *fuzzer*.

Primeiro, é importante definir aspectos mais gerais do projeto. Definir como o *fuzzer* é executado é essencial para facilitar a reprodução dos experimentos realizados. Além disso, isolar a execução também é útil para que as execuções do *fuzzer* sejam independentes, possibilitando estudar de forma mais aprofundada o comportamento do DogeFuzz.

Por não ser um *fuzzer* de programa comum, o planejamento do *fuzzer* é essencial para o seu funcionamento. Em um *Blockchain*, o processamento das transações é feito de maneira independente ao processo do *fuzzer*. Por isso, para implementar técnicas avançadas de *fuzzing*, o DogeFuzz deve colher informações sobre a execução, mesmo com a execução independente do fuzzer e do *Blockchain*. Para realizar esta implementação, a utilização de *webhooks* se mostrou essencial, bem como a realização de processamentos assíncronos, para que o *fuzzer* conseguisse controlar o comportamento dos *smart contracts* em um ou mais nós que estão participando de um *Blockchain* local.

Como mencionado anteriormente, técnicas mais avançadas de *fuzzing* utilizam dados sobre execuções anteriores para ajudar a gerar novos *inputs* válidos. Por isso, o DogeFuzz precisa colher dados como as instruções executadas e vulnerabilidades encontradas durante estas execuções. Para implementar técnicas como *greybox fuzzing* e *directed greybox fuzzing*, também é preciso computar dados como cobertura e distância a pontos críticos do código, que são essenciais para o funcionamento dessas estratégias. Para isso, foram implementados recursos no DogeFuzz para incluir a capacidade de gerar uma representação de um *smart contract* em um Control-Flow Graph (CFG). Esta representação contém dados importantes sobre os caminhos que um programa pode ser executado e, com isso, tornou-se computar os valores necessários para o *fuzzer*.

Como um arquitetura orientada a eventos, o DogeFuzz precisou utilizar um banco de dados para armazenar os dados de execução para que sejam o ponto central de comunicação entre diferentes partes do programa que são executados em paralelo e de forma independente. Para que o DogeFuzz suporte novas técnicas de *fuzzing*, foi necessário implementar padrões de design que possibilitam a extensibilidade do programa, com o padrão *strategy*. Neste padrão, o componente depende apenas do contrato definido por uma *interface* e a implementação feita passa a ser escolhida por meio de um seletor.

Essas decisões simplificam tanto o desenvolvimento quanto a execução do DogeFuzz. Nesta sessão, descrevemos cada ponto mencionada de forma que fique claro os benefí-

cios da arquitetura escolhida no desenvolvimento do DogeFuzz. A Figura 3.1 mostra a representação geral da arquitetura do projeto.

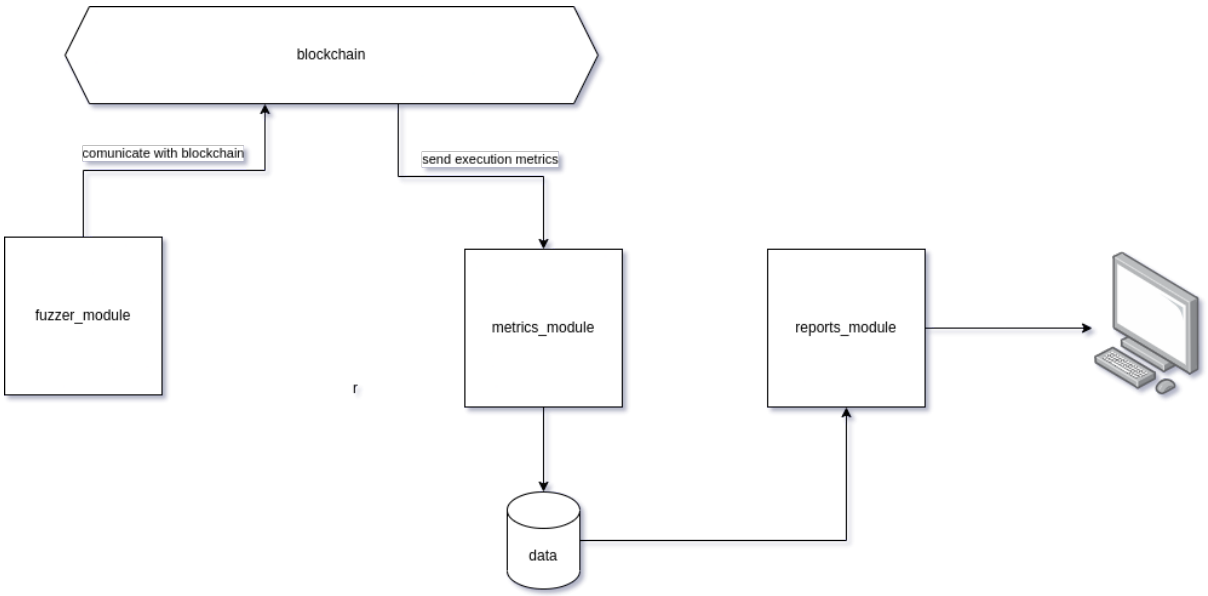

Figura 3.1: Representação da arquitetura do DogeFuzz.

3.3.1 Reprodutibilidade

Durante o estudo realizado sobre o estado-da-arte em *fuzzing* para *smart contracts*, visto na sessão 2.3, passamos por diversos problemas na reprodução dos resultados mostrados por estas pesquisas. Isso impossibilitou a comparação dos resultados com o mesmo conjunto de *smart contracts* explorado em pesquisas anteriores. Além disso, os projetos de pesquisa investigados utilizaram diferentes procedimentos para prepararem os contratos para serem analisados. No caso de ferramentas de análise dinâmica, os contratos eram instalados em um *Blockchain* local para que fossem executados numa máquina virtual semelhantes as que são utilizadas em uma rede *Blockchain* real.

De forma que a execução do DogeFuzz tenha menos atrito, o ambiente de execução do DogeFuzz deve estar desacoplado do computador que está executando a aplicação, ou seja, o ambiente deverá ser virtualizado. Na indústria, a tecnologia de contêineres Docker é bastante utilizada para isolar a execução de aplicações em ambientes controlados como a computação em nuvem. Esta tecnologia possibilita a virtualização de maneira leve e a criação de *snapshots* de um sistema operacional que possibilita o mesmo contêiner ser iniciado com os mesmo pacotes e versões independentemente do computador que está executando aquele contêiner.

Um *fuzzer* para *smart contract* precisa se comunicar com um *Blockchain* local para executar os contratos em um ambiente operacional. Para isso, é preciso que o *fuzzer* seja capaz de realizar a preparação do contrato para ser executado no *Blockchain*.

Primeiro, o contrato deve ser compilado da linguagem *solidity* para o formato de *bytecode* que o *Blockchain* Ethereum aceita. Nesse passo, utilizamos a ferramenta *solc* que realiza duas operações importantes. A partir de código fonte do *smart contract*, esta ferramenta compila o código em bytecode compatível com a EVM e gera a especificação ABI que é utilizada para a comunicação com aquele contrato.

Segundo, é preciso que o DogeFuzz possua uma lógica para publicar os contratos na rede local. Aqui, escolhemos a biblioteca cliente *geth*—a biblioteca mais utilizada e que possui métodos importantes para a comunicação com o *Blockchain* e o mapeamento da especificação ABI na linguagem de programação Go. Além disso, a biblioteca possui a funcionalidade de executar um nó do *Blockchain* localmente, e servir como uma rede local para que o DogeFuzz possa se comunicar e interagir com os contratos publicados neste nó.

De forma que técnicas mais elaboradas de *fuzzing* sejam implementadas, o DogeFuzz precisa computar uma representação de um *smart contract* em um Control-Flow Graph. Esta representação é necessária para analisar quais caminhos do programa estão sendo executados e, com isso, computar dados como cobertura e aproximação de instruções críticas que serão apresentadas nas implementações de *fuzzing* suportadas. Para a construção do Control-Flow Graph, o DogeFuzz se beneficia da ferramenta Vandal [58]; que realiza análises de segurança de *bytecode* de *smart contracts*. Para fins de simplicidade na integração com *fuzzer*, foi criado um servidor REST em cima desta ferramenta que contém um único *endpoint* que recebe o *bytecode* do contrato e retorna a representação correspondente do CFG.

Como pode ser visto na figura 3.2, a arquitetura do DogeFuzz foi feita utilizando contêineres para garantir a reprodução dos experimentos de maneira fácil. Utilizando esta arquitetura, pudemos integrar o DogeFuzz com um nó de um *Blockchain* local e com a ferramenta Vandal. Aqui, a ferramenta utilitária *docker-compose* [59] foi utilizada para coordenar as execuções dos contêineres Docker e para também executá-los em uma rede privada onde os contêineres poderão de comunicar de maneira simples, onde cada um terá seu próprio domínio na rede privada.

3.3.2 Uso de Comunicação Assíncrona na Arquitetura

Como mencionado, um *fuzzer* robusto, que teste *smart contracts* em um *Blockchain* local, deve apresentar um comportamento assíncrono. A execução de uma *smart contract* é feita dentro da máquina virtual do EVM e para o *fuzzer* colher dados da execução do contrato,

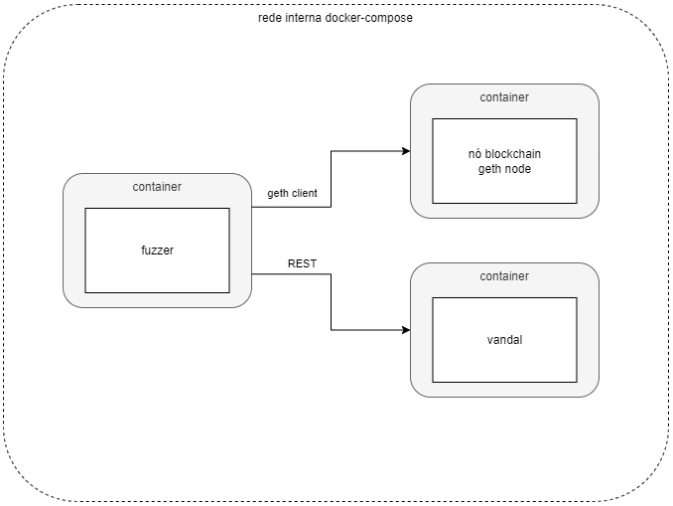

Figura 3.2: Arquitetura de containers do DogeFuzz.

é preciso que a máquina virtual se comunique com o *fuzzer* para enviar as instruções executadas. Esta troca de dados é feita por meio de requisições HTTP e *webhooks*. Para sincronização e detecção do fim das execuções, o DogeFuzz utiliza *jobs* recorrentes que rodam em segundo plano e checam dados da execução (como o sucesso ou não das transações).

Para a EVM se comunicar com o *fuzzer*, a EVM deve se comunicar utilizando algum protocolo pré definido. Escolhemos o protocolo HTTP por ser amplamente utilizado e utilizar um codificação simples como o formato JSON. O DogeFuzz funciona como um servidor HTTP, recebendo da máquina virtual requisições contendo as instruções executadas e as vulnerabilidades encontradas. Além disso, o *fuzzer* pode receber requisições para iniciar o processo de execução e, assim, possibilitando a integração do DogeFuzz com outras aplicações, como, por exemplo, o componente de *benchmark* que usamos para executar os experimentos que visam estimar a efetividade do DogeFuzz.

O DogeFuzz também foi desenhado para depender exclusivamente de *webhooks*, tanto para atualizar seu estado quanto para atualizar outras aplicações a cerca da sua execução. Esta é a alternativa mais simples para atualizar o estado interno do *fuzzer* sem precisar de uma conexão persistente. Como foi mencionado acima, a máquina virtual realiza requisições HTTP para o *fuzzer* contendo os dados colhidos com a instrumentação feita dentro da EVM.

Como algumas implementações de *fuzzing* dependem de execuções anteriores, o DogeFuzz armazena os dados das execuções em um banco de dados e utiliza *jobs* recorrentes que controlam as execuções feitas e sincroniza a execução do *fuzzer* definindo quando novos inputs serão gerados e quando a execução deverá ser finalizada.

Para que todo o sistema execute de forma assíncrona, com todos os componentes

mencionados acima, a arquitetura do DogeFuzz é baseada em eventos. Desta forma, os componentes se comunicam de forma desacoplada e reativa conforme novos eventos ocorrem durante a execução do *fuzzer*. Estes eventos são enviados para uma fila defina por um tópico, onde cada tópico é processado utilizando a estratégia First in, First out (FIFO).

Tópico *task-start*

O tópico *task-start* é utilizado para processar o inicio da execução do DogeFuzz em um *smart contract*. Desta forma, a interface do *fuzzer* pode ser implementado utilizando diferentes estratégias, como uma REST API ou um aplicação CLI. Neste projeto, apenas o servidor HTTP foi implementado para facilitar a comunicação com o *benchmark*. Assim que o evento entra no tópico, o componente do DogeFuzz, que é responsável por processar esse evento, recebe automaticamente esta mensagem e começa a preparação do contrato para ser testado. Para iniciar a execução, o *fuzzer* recebe alguns dados importantes, como:

- O código-fonte do(s) *smart contract(s)*;
- O nome do contrato principal;
- A estratégia de *fuzzing*; e
- O tempo de execução.

Na Figura 3.3, é possível notar que o tópico *task-start* pode receber mensagens de diferentes componentes do programa, ou seja, podem ser escolhidas diferentes estratégias para iniciar a execução do DogeFuzz. Neste projeto, foi escolhida a estratégia de utilizar um servidor REST para iniciar o programa. Desta forma, uma execução em um contêineres Docker é facilitada pois este tipo de virtualização já possui funcionalidades prontas para a comunicação HTTP como o mapeamento de portas e resolução de domínio dentro de um rede interna no Docker.

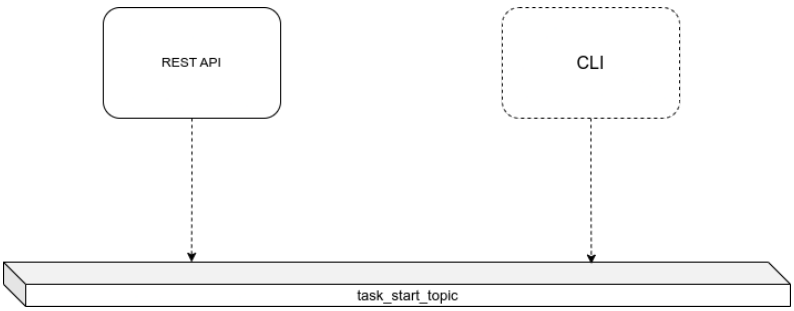

Figura 3.3: Componentes que publicam mensagens no tópico *task-start*.

Toda vez que uma mensagem é enviada ao tópico, um componente do DogeDuzz, que é responsável por esta mensagem, deve ser executado. Este componente foi nomeado de *contract_deployer_listener* e sua composição pode ser vista na Figura 3.4. Esse componente é responsável por fazer a preparação do *fuzzer* para testar o contrato em questão. Em mais detalhes, o componente compila o contrato utilizando a ferramenta *solc*, que gera o *bytecode* do contrato e a especificação ABI correspondente. Com o *bytecode* do contrato, é possível publicar o contrato em um nó local do *Blockchain*, que será feito também por este componente. Além disso, é possível chamar o serviço Vandal, o qual ainda será descrito nesse capítulo, que gera uma representação CFG do contrato. Com todos estes dados sobre os contratos, o componente irá armazená-los em um banco de dados para que sejam recuperados por outros componentes do *fuzzer* durante a sua execução.

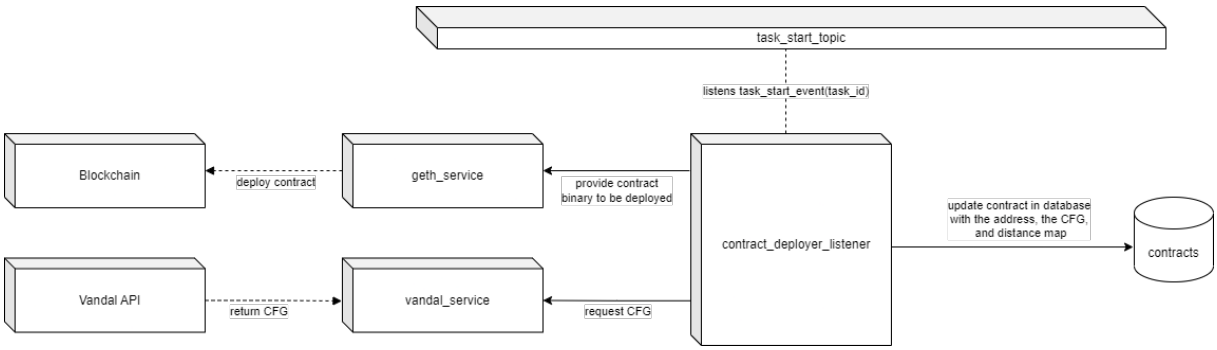

Figura 3.4: Componente *contract_deployer_listener*.

Depois que a preparação é finalizada, o DogeFuzz publica uma nova mensagem ao tópico *task-input-request*, que é responsável pela geração de novos inputs. Isso se repete até que a execução do *fuzzer* seja concluída. Este tópico corresponde ao componente principal do DogeFuzz. Assim que o tempo de execução se encerra, um evento é enviado ao tópico *task-finish*, responsável pela finalização do processo de execução e envio do relatório de execução, com as vulnerabilidades detectadas.

Tópico *task-finish*

Quando o tempo de execução chega ao fim, o tópico *task-finish* é acionado, causando a execução do componente responsável pela emissão do relatório de execução, que contém informações como vulnerabilidades encontradas, cobertura atingida e instruções críticas executadas. O DogeFuzz foi implementado utilizando um *job* periódico que checa quais execuções já acabaram com base no tempo de limite calculado com o parâmetro de entrada do *fuzzer*. Assim que este tempo expirar, o tópico *task-finish* é acionado para que o componente responsável possa gerar o relatório de execução. A implementação da geração

deste relatório foi feita de maneira flexível, com vários formatos de de relatório úteis tanto para uma execução local quanto para execução de um experimento com vários contratos.

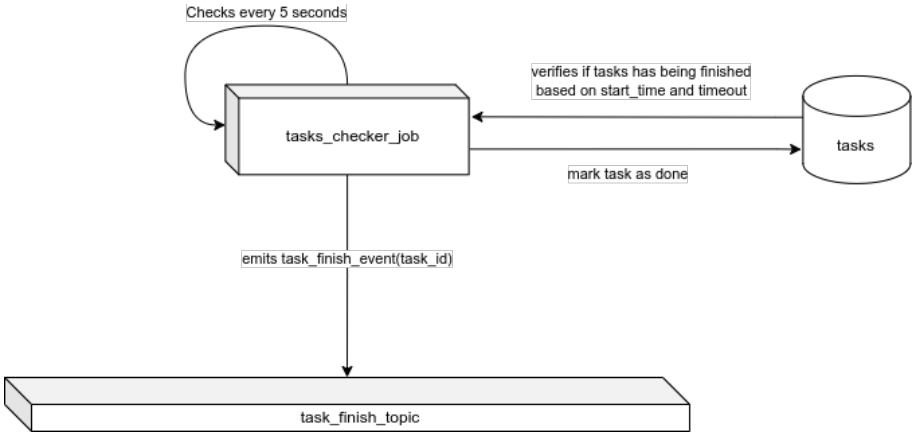

Figura 3.5: Job que publica mensagens no tópico *task-finish*.

Para checar o tempo de execução, o *fuzzer* possui um *job* periódico que a cada 5 segundos checa no banco de dados por tarefas que tenham extrapolado o período limite. Como pode ser visto na Figura 3.5, para cada tarefa encontrada pelo *job*, um evento é enviado para o tópico *task-finish*. O cálculo do tempo de execução é feito com base no tempo em que o componente *contract_deployer_listener* acaba sua execução, ou seja, a partir do momento em que o contrato está pronto para ser chamado pelo *fuzzer*. Assim, o tempo de execução, que é definido na entrada do usuário, começa a valer apenas quando o *fuzzer* estiver realmente explorando o contrato.

O componente responsável por processar o evento do tópico *task-finish* gera o relatório de execução do *fuzzer*, como pode ser visto na Figura 3.6. Para seguir o objetivo de flexibilidade do projeto, a geração do relatório também foi implementada utilizando o padrão de projeto *strategy* de programação orientada à objetos, onde várias implementações podem ser feitas seguindo a mesma interface e o programa possui algum condição para escolher qual das implementações será executadas. Assim, a geração foi implementada com base na estratégia de relatório recebida como parâmetro do *fuzzer*. Então, os seguintes tipos de relatório foram implementadas:

- Escrita no *log* com os principais resultados;
- Escrita em um arquivo local no formato JSON; e
- Envio de uma requisição HTTP para um *webhook*.

Particularmente, a implementação da estratégia *webhook* foi essencial para executarmos o experimento. Isso deve pelo fato do experimento, que será detalhado posteriormente, ser executado utilizando um *container* para facilitar a sua execução em qualquer

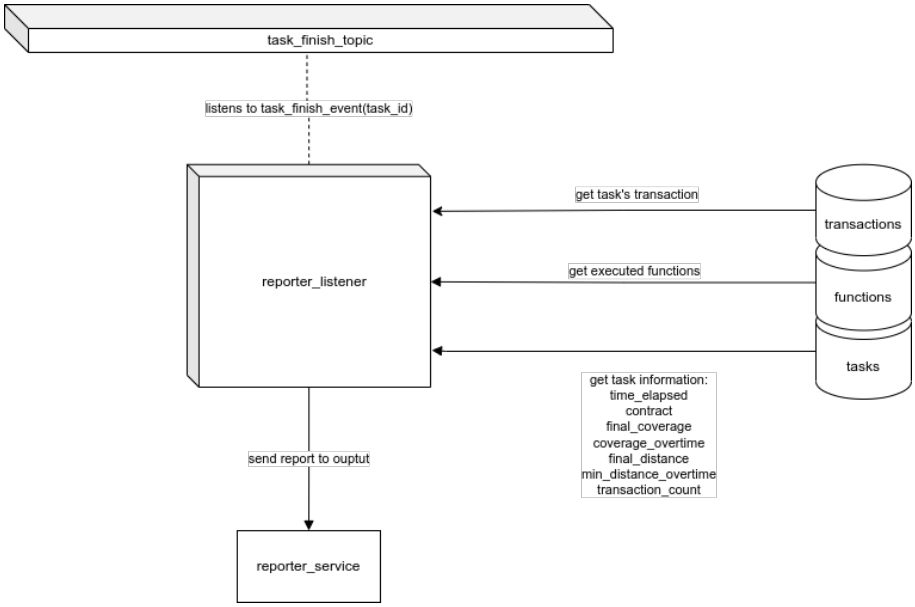

Figura 3.6: Componente *reporter_listener*.

ambiente, e assim, seguir o objetivo de garantir a reprodução dos experimentos. Os relatórios contém informações essenciais para analisar a efetividade do *fuzzer*, por isso a requisição que será feita será codificada no formato JSON e conterá as seguintes informações sobre a execução:

- Vulnerabilidades encontradas;
- Cobertura atingida;
- Instruções críticas executadas;
- Inputs gerados; e
- Instruções executadas.

Portanto, utilizando os tópicos *task-start* e *task-finish*, é possível definir o começo e o fim da execução de um *fuzzer*. Mas como o DogeFuzz funciona durante este período será definido pelo tópico *task-input-request*, que é responsável por gerar novos *inputs* e várias chamadas ao *smart contract* que está sendo testado.

Tópico *task-input-request*

O tópico *task-input-request* dispara a execução do componente central do DogeFuzz, onde ocorre a chamada ao nó do *Blockchain* e a geração de novos *inputs*. Este tópico recebe mensagens de dois componentes do *fuzzer*. O componente *contract_deployer_listener*,

que depois de preparar a execução do fuzzer, chama este tópico para iniciar a execução do *fuzzer*. E para controlar a execução do *fuzzer* durante o tempo de execução definido e requisitar novos *inputs*, o *fuzzer* utiliza um *job* periódico que checa se todas as transações criadas foram finalizadas e, caso afirmativo, requisita que novos *inputs* sejam gerados. Portanto, toda vez que um evento for publicado neste tópico, será iniciada a execução de um componente que será responsável pela geração de *inputs* baseado na escolha da estratégia de *fuzzing* para aquela execução.

Como mencionado anteriormente, o componente *contract_deployer_listener* prepara o *smart contract* para execução. Depois que isso é feito, um evento é publicado no tópico *task-input-request* para que comece a geração de inputs e execução do *fuzzer*. E para que o resto da execução seja controlada e monitorada, um *job* periódico foi implementado de forma a checar se as transações realizadas já receberam algum retorno da *EVM* ou se ouve algum erro com aquela transação. Este *job* também é essencial para controlarmos a quantidade de inputs gerados de forma que não sobrecarregue o nó do *Blockchain* que consegue processar um número finito de transação num curto espaço de tempo.

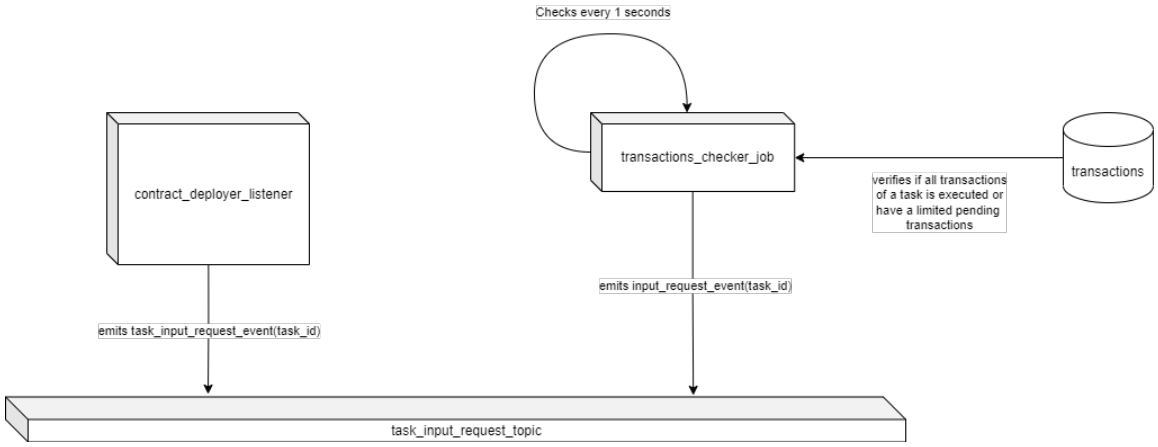

Figura 3.7: Publicação de mensagens no tópico *task-input-request*.

Para que se otimize a geração de novos inputs sem sobrecarregar o nó, existe uma lógica que evita gerar novos inputs para limitar o número de transação pendentes, ou seja, que não receberam o retorno da sua execução. Com esta lógica é possível gerar um número de transações maior e ao mesmo tempo esperar que o nó do *Blockchain* execute as transação já realizadas. Na Figura 3.7, é possível checar um diagrama que mostra a publicação de mensagens no tópico *task-input-request* e interação do *job* periódico, chamado de *transaction_checker_job*, com o banco de dados para buscar as informações sobre as transações anteriores.

fuzzer_listener é o componente que executa assim que um evento é publicado no tópico *task-input-request*. Esse componente é responsável por duas tarefas. Primeiro, pela

geração de novos *inputs* seguindo a estratégia de *fuzzing* definida para aquela execução. E, assim que tiver os *inputs* gerados, será executado um lógica para chamar o *smart contract*. Aqui, o contrato é chamado de diversas maneiras como mencionado na Sessão 2.2.3 e a escolha da estratégia é feita de maneira aleatória. A descrição em detalhes do funcionamento do *fuzzer* será feita na Sessão 3.4.

Com isso, a parte do *fuzzer* que são gerados os *inputs* de teste do *smart contract* foi finalizada. Esta parte é executada de maneira independente dos componentes que analisam a execução do *smart contract* na *EVM*. Para isso, o *fuzzer* utiliza o tópico *instrument-execution* e algumas APIs para receber os dados da execução do *smart contract* como instruções executadas e indícios de vulnerabilidades encontrados.

Tópico *instrument-execution* e detecção de vulnerabilidades

Durante a execução de um *smart contract*, a EVM fornece dados importantes para o *fuzzer*. Dois *webhooks* são usados para compartilhar os dados. Um *webhook* é utilizado para receber os *oracles* implementados que servirão para detectar vulnerabilidades durante a execução. E o outro *webhook* é utilizado para receber dados das instruções executadas pela EVM. Em cada execução de contrato ou execução em cadeia de mais de um contrato, são enviados requisições para estes *webhooks* que processaram estes dados.

A detecção de vulnerabilidades depende das informações enviadas pelos *oracles*, que são explicados mais a fundo na Sessão 3.5. Os *oracles* são recebidos por meio de um *webhook* do *fuzzer* que, por meio de uma requisição, HTTP recebe indícios de que uma vulnerabilidade foi encontrada. Assim que o *fuzzer* recebe essa informação, ele armazena em banco de dados quais vulnerabilidades foram encontradas na execução daquela transação.

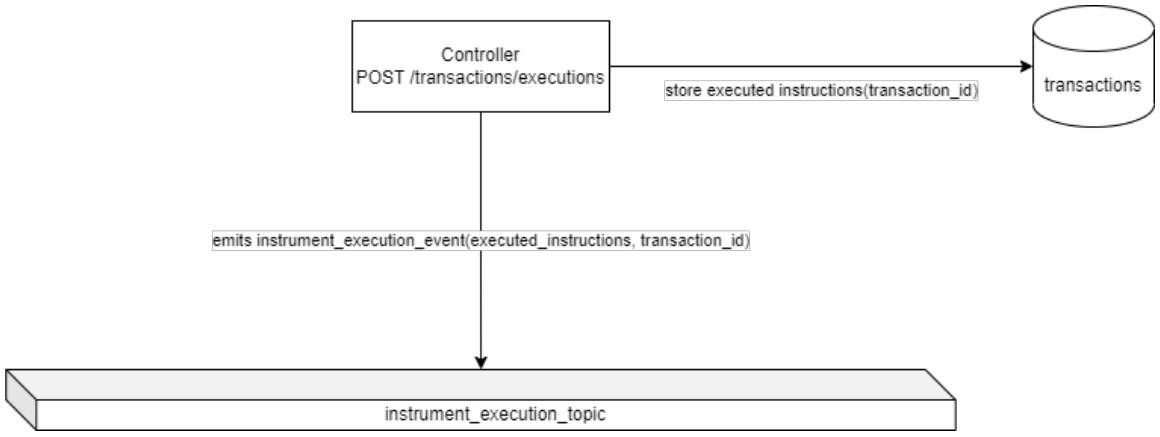

Figura 3.8: Publicação de mensagens no tópico *instrument-execution*.

Como pode ser visto na Figura 3.8, o *webhook* responsável por receber as instruções executadas, são imediatamente armazenadas em um banco de dados para serem executadas posteriormente. Assim que for executado, o *webhook* publica um evento *instrument-execution*. As instruções executadas são associadas ao *hash* da transação que a identifica unicamente no banco de dados—para que possa ser facilmente recuperada por outros componentes do programa.

Portanto, para processar estas instruções, o tópico *instrument-execution* é utilizado, sendo aplicada uma lógica para o cálculo da cobertura atingida e se a execução do *smart contract* se aproximou das instruções críticas. Assim todos estes dados serão armazenados em banco de dados para ser utilizado pelo componente do *fuzzer*, onde existem implementações que precisam destes dados de cobertura das execuções anteriores.

3.3.3 Cálculo da Cobertura e Distância de Instruções Críticas

Estratégias de *fuzzing* que dependem de elementos contextuais da execução, como a cobertura atingida e a aproximação da execução de instruções específicas do *bytecode*, precisam computar estes dados de forma que não ocorra um impacto no desempenho do *fuzzer*. Por isso, todos cálculos de distâncias e representações do *bytecode* são feitos na preparação do DogeFuzz, antes que o processo de geração dos *inputs* seja iniciado.

Construção do Control-Flow Graph (CFG) para Smart Contracts

A representação Control-Flow Graph (CFG) é muito utilizada para análise dos diferentes caminhos que podem ocorrer durante a execução de um programa. Como pode ser visto na Figura 3.9, um Control-Flow Graph possui o formato de um grafo, onde as arestas são bifurcações que o código em questão pode ter, como *loops* e condições, e os nós do grafo representam os blocos que são executados em cada bifurcação. A implementação de uma conversão do *bytecode* em uma representação CFG foi delegada à ferramenta Vandal, apresentada por Lexi Brant et al. [58]. Essa ferramenta, que é escrita em Python, é um *framework* que objetiva facilitar a análise estática de *smart contracts*. A ferramenta Vandal recebe o *bytecode* do contrato e retorna um CFG no formato textual ou em formato visual, por meio de um diagrama. Para este projeto, encapsulamos esta ferramenta em um servidor REST para que o DogeFuzz possa fazer chamadas simples durante a sua execução e receber a representação CFG do contrato que está sendo testado.

Instrumentação

Em um CFG, os blocos de código, que são representados pelos nós do grafo, são identificados pelo valor do Program Counter (PC) da primeira instrução daquele bloco. O

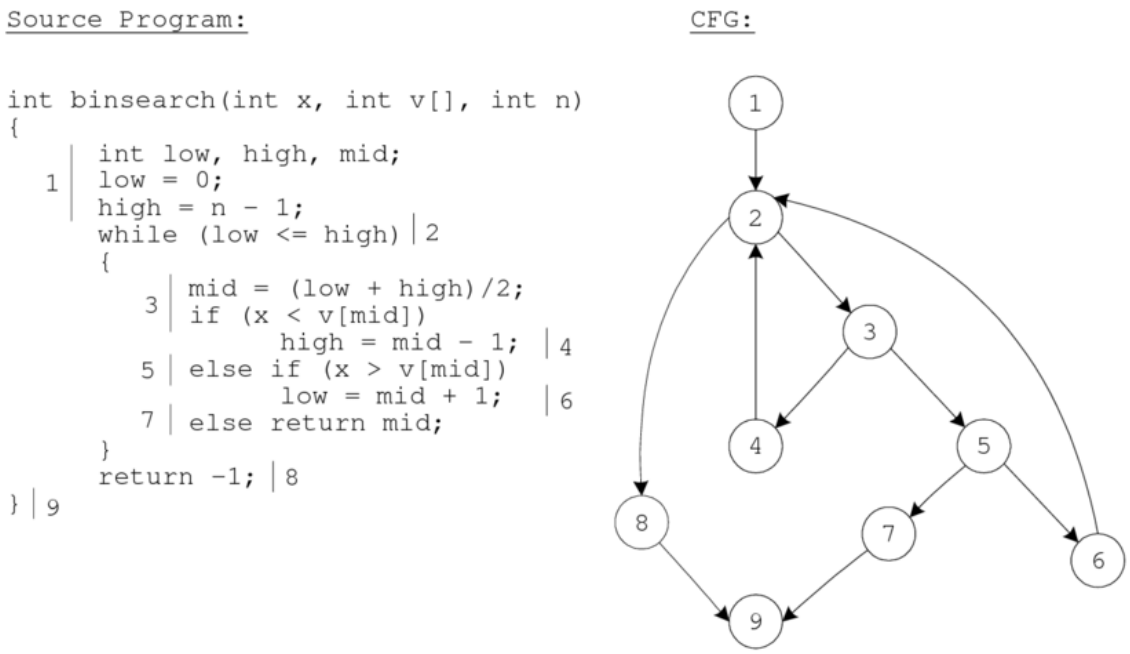

Figura 3.9: Representação Control-Flow Graph de uma busca binária.

fuzzer utiliza essa informação para associar as instruções executadas com os blocos do CFG. E para colher quais instruções foram executadas, modificamos a *EVM* de forma que, toda vez que uma execução de um contrato for finalizada, seja feita uma requisição ao DogeFuzz. Tal requisição informa o conjunto de instruções do *smart contract* que foram executadas.

Além disso, é importante citar que as instruções são identificadas simplesmente pelo valor do Program Counter, e, por isso, foi feita uma análise profunda da compatibilidade entre os valores encontrado no CFG e a instruções executadas na EVM. Com isso, foi identificado que a ferramenta *solc*, utilizada para compilar o código-fonte do *smart contract*, retorna dois tipos de *bytecode*: o *bytecode* de publicação do contrato e o *bytecode* de execução.

Apesar dos dois tipos de *bytecode* serem gerados a partir de um mesmo contrato, eles possuem utilidades diferentes e principalmente os valores de PCs de instruções são diferentes, o que influencia na associação que é feita entre as instruções do CFG e as instruções executadas. O *bytecode* de publicação contém o *bytecode* de execução e os parâmetros de inicialização do *smart contract*. Quando o contrato é publicado no *Blockchain*, apenas o *bytecode* de execução é armazenado e executado pela EVM. Por isso, o DogeFuzz deve se basear apenas no *bytecode* execução na sua execução e dessa forma, os valores do PCs vindos da representação CFG serão compatíveis com as instruções executadas na EVM.

Com as lógicas de instrumentação e a representação CFG feitas, será possível computar a cobertura atingida pelo *fuzzer* e saber quais instruções críticas foram executadas.

Cobertura

A cobertura de código é uma métrica utilizada muitas vezes para medir o quanto testes exploraram algum programa. Por mais que essa métrica não esteja diretamente relacionada com a chance de encontrar vulnerabilidades, essa métrica é importante para o *fuzzer* utilizar com o objetivo de gerar *inputs* que explorem novos caminhos do programa. Com isso, uma estratégia de como computar a cobertura deverá ser definido.

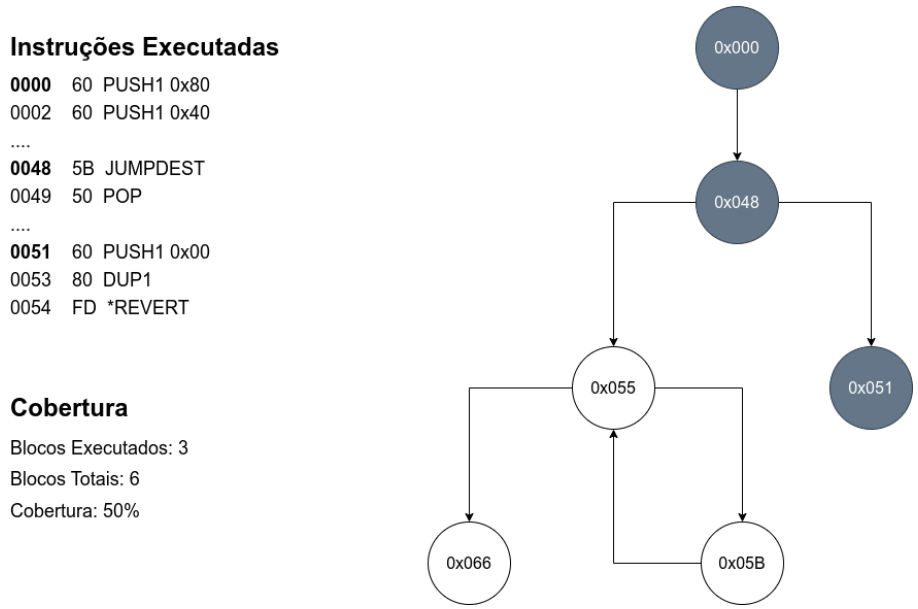

Figura 3.10: Cálculo da cobertura.

A estratégia definida foi feito com base no blocos de código representados pelos nós do CFG. Como cada bloco é identificado pelo valor de PC da primeiro instrução daquele bloco, o DogeFuzz consegue estimar a quantidade de blocos executados com base na lista de instruções executadas e o CFG. A Figura 3.10 representa como é feito o calculo da cobertura. Aqui, fazemos os mapeamento das instruções executadas e os blocos e, assim, é possível computar quais blocos foram executados. A porcentagem de cobertura é estimada com base no número total de blocos CFG.

Além da cobertura, o DogeFuzz necessita computar o alcance às instruções críticas durante a sua execução. Para isso, precisa utilizar estratégias que focam na geração de inputs que exploram partes do programa que levam à execução das instruções críticas.

Instruções Críticas

Uma implementação de *fuzzer* que se guie pela exploração de pontos específicos do código como o *directed greybox fuzzing* necessita de uma estratégia para computar a aproximação destes pontos. Particularmente, foi escolhido uma lógica que guia o DogeFuzz na aproximação de algumas instruções consideradas críticas. Este conceito de instruções críticas foi baseado no estudo feito por Krupp e Rossow [22], que utilizaram execução simbólica para explorar instruções críticas em *smart contracts*. Nesse estudo, as seguintes instruções da EVM foram consideradas críticas:

CALL: Cria transações entre usuários

SELFDESTRUCT: Destrói um contrato, transferindo as cripto-moedas armazenadas no contrato para um endereço específico.

CALLCODE e DELEGATECALL: Possibilitam injeção de código em *smart contracts*.

Parte do motivo de utilizarmos estas mesmas instruções é que as vulnerabilidades, que serão exploradas neste projeto de pesquisa, se aproveitam de forma direta ou indiretamente destas instruções. Além disso, instruções, como `CALL`, realizam transferências de cripto-moedas e, por isso, é alvo de diversos outros tipos de ataques que focam em recompensa monetário.

Como foi mencionado anteriormente, alguns cálculos são feitos na preparação do DogeFuzz para otimizar a performance do *fuzzer*, evitando que cálculos sejam feitos de forma repetitiva. Um deles é a geração do mapa de distâncias às instruções críticas. Este mapa contem as distâncias entre cada bloco e os blocos que contém as instruções críticas. Blocos que não podem chegar em uma instrução crítica armazenam a distância máxima. A Figura 3.11 representa como é feito o mapeamento, onde o nó preenchido possui uma instrução crítica e os outros nós mapeiam as distância deste nó.

Além disso, é possível notar que a geração do mapa é feito com base no CFG. Criamos um grafo copiando os mesmos nós do CFG, onde cada nó é identificado pelo valor de PC da primeira instrução do bloco, mas também contem a informação da distância no grafo até o bloco com alguma instrução crítica. Primeiramente, iniciamos este grafo com valores máximos de distância e começamos a preencher os nós que possuem algum caminho até instrução crítica. Com o mapa de distância iniciado, é computado um grafo inverso ao CFG que serve como referência para computar o caminho até o bloco com as instruções críticas. Com este grafo reverso, poderemos realizar um algoritmo Breadth-First Search (BFS) para percorrer o caminho que um programa levaria para instrução crítica. Com isso, podemos percorrer este caminho e computar as distâncias com base no número de

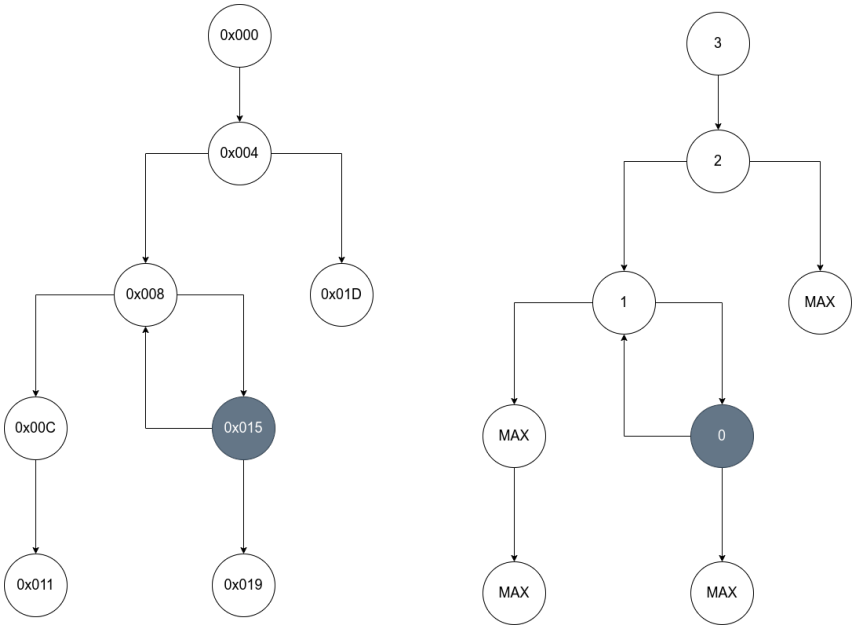

Figura 3.11: Geração do mapa de distâncias.

blocos de código que compõem este caminho no CFG reverso. Conforme for percorrendo o caminho reverso, é incrementado o valor da distância do bloco anterior. Assim que o grafo não tiver nós para serem percorridos, o mapa de distâncias já terá sido preenchido com todas as distâncias de todos caminhos possíveis para o bloco com a instrução crítica. A implementação da inversão deste grafo pode ser visto a seguir:

```
func (g CFG) GetReverseGraph() map[string][]string {
    reversed := make(map[string][]string)
    for node, edges := range g.Graph {
        for _, edge := range edges {
            reversed[edge] = append(reversed[edge], node)
        }
    }
    return reversed
}
```

Enquanto que a implementação do algoritmo de BFS para computar as distancias e preencher o mapa de distancias pode ser visto no trecho de código a seguir:

```
distanceMap[targetBlock][targetBlock] = 0
queue := []string{targetBlock}
for len(queue) > 0 {
    current := queue[0]
    queue = queue[1:]
    for _, edge := range reversedCFG[current] {
        if distanceMap[edge][targetBlock] == math.MaxUint32 {
```

```
                distanceMap[edge][targetBlock] =
                    distanceMap[current][targetBlock] + 1
                queue = append(queue, edge)
            }
        }
    }
```

Como um *smart contract* pode conter mais que um instrução crítica, é necessário também ter uma lógica para armazenar todas as distâncias em cada nó do mapa de distâncias. Por isso, os códigos acima (escritos na linguagem de programação *Go*) serão executados para cada instrução crítica encontrada no código. E, em vez de armazenar somente uma distância de bloco, cada nó irá armazenar um dicionário de distâncias para cada instrução crítica.

Para o *fuzzer* computar todas as distâncias atingidas, é feito um mapeamento das instruções executadas e o mapa de distâncias. Também é computado o valor da mínima distância, ao percorrer todos os blocos executados, gerando um dicionário contendo as mínimas distâncias atingidas com aquela execução. Para fins de comparação entre outras execuções, o dicionário é representado pelo somatório das suas distâncias mínimas, ou seja, caso este somatório seja menor que outra execução, é possível afirmar que o *fuzzer* conseguiu se aproximar mais de alguma instrução crítica sem precisar saber qual instrução.

Portanto, foram definidos os componentes que auxiliam na execução do DogeFuzz. O cálculo da cobertura necessário para a implementação de um *greybox fuzzing* e o cálculo das distâncias as instruções críticas que são necessários para implementação de um *directed greybox fuzzing* guidado por estas instruções.

3.4 Detalhes de Implementação

3.4.1 Fuzzing

Como mencionado anteriormente, a lógica de *fuzzing* é executado assim que o tópico *input-request* recebe uma requisição. O componente *fuzzer_listener*, que é executado assim que a mensagem for recebida no tópico, faz algumas operações necessárias para gerar e enviar novos *inputs* ao *smart contract*. Aqui, a especificação ABI é essencial para a escolha de qual métodos será chamado e quais parâmetros este método possui. Além disso, para manter o requisito arquitetural de extensibilidade, esse componente deve suportar a execução de diferentes estratégias e implementações de *fuzzing* para gerar os parâmetros dos métodos escolhidos. Para isso, é preciso integrar este componente com o banco de dados, onde são armazenados os dados sobre a cobertura e distâncias atingidas às instruções criticas, descritas anteriormente neste documento, de execuções anteriores.

Por fim, este componente também é responsável por escolher qual tipo de interação com *smart contract* será feito, algo essencial para a detecção de algumas vulnerabilidades.

3.4.2 Escolha do Método do Contrato

A primeira parte da execução do *fuzzer* é a escolha de qual método do contrato será testado naquele momento. Aqui, a especificação ABI do contrato fornece dados importantes que serão utilizados na estratégia de escolha do método e dos dados sobre a interação do contrato e o *fuzzer*, como a visibilidade do método, mutabilidade do método (ele pode mudar o estado do *smart contract*) e se o método é *payable*, ou seja, se é necessário enviar uma quantia de criptomoedas para ser manipulada por aquele método.

A especificação ABI é essencial para a comunicação entre o *fuzzer* e o *smart contract*. Tal especificação lista todos os métodos de um contrato. É possível checar a visibilidade dos métodos e encontrar aqueles métodos públicos, ou seja, métodos que podem ser chamados externamente. Na descrição de cada método, existe também a classificação do método quanto à mutabilidade—indicando se o método muda o estado interno do *Blockchain* ou não. Estas duas informações foram utilizados para filtrar a lista de métodos que realmente impactam no funcionamento do contrato e, consequentemente, focar a execução do *fuzzer* nestes métodos. Com essa lista de métodos, a estratégia de escolha do método foi definida como a simples escolha aleatória de um dos métodos públicos. Assim, todos os métodos são executados de uniformemente.

Assim que o método alvo é definido, a especificação ABI é utilizada para recuperar a lista de parâmetros daquele método e quais tipos de dados cada parâmetro exige, os quais serão essenciais para estratégias de geração de inputs. Além disso, para cada tipo da linguagem *Solidity*, foram implementados vários componentes que lidam com as operações dos parâmetros do seu respetivo tipo. Estes componentes são utilizados pelas estratégias de *fuzzing* que foram implementadas nesse projeto.

3.4.3 Tipos da Linguagem *Solidity*

Como foi mencionado anteriormente no Seção 2.2, *Solidity* é uma linguagem estaticamente tipada, ou seja, todos as variáveis e parâmetros têm tipos bem definidos. E após um estudo na documentação oficial da linguagem, foi possível notar que ela possui tipos bem comuns em linguagens de programação; mas também possui tipos muitos específicos do contexto de *Blockchain*, como o tipo *address* que representa o endereço de um conta no *Blockchain*. Portanto, para que o DogeFuzz possa ser implementado, é preciso criar componentes que consigam realizar operações com cada um destes tipos de forma a abranger todos

Solidity	*Go*
uint8, uint16, uint32, uint64	uint8, uint16, uint32, uint64
int8, int16, int32, int64	int8, int16, int32, int64
uint24, uint40, ..., uint256	*big.Int
int24, int40, ..., int256	*big.Int
bool	bool
string	string
address	*common.Address
bytesN (fixed-sized)	[N]byte
bytes (dynamically-sized)	[ ]byte
type[N] (static array)	[N]interface{} (array)
type[ ] (dinamic array)	[ ]interface{} (slice)
high-order functions	[ ]byte
structs	map[string]interface{}

Tabela 3.1: Equivalência de tipos entre *Solidity* e Golang

os tipos de contratos com seus diferentes parâmetros. Este componente é chamado de *type_handler*.

Na linguagem *Solidity*, existem tipos simples, como inteiros e strings, e tipos mais complexos, como *arrays* e *structs*. E para mapearmos os tipos de *Solidity* em tipos da linguagem Go, foi utilizado a biblioteca `go-ethereum` que possui uma vasta API de geração de código Golang com base na especificação ABI. A Tabela 3.1 apresenta a equivalência de tipos da linguagem *Solidty* com os tipos da linguagem Golang.

Alguns tipos exigiram um tratamento especial na hora de implementar um componente *type_handler* para sua manipulação em Golang. Por exemplo, tipos dinâmicos como *arrays*, listas e *structs* exigiram que o DogeFuzz possuísse uma lógica de mapeamento em recursão para que os componentes *type_handler* para esse tipos dinâmicos utilizem os *type_handlers* respectivos aos tipos base. Particularmente, o tipo *address* possui uma lógica mais diferente dos outros tipos, pois ele deve representar um endereço real do *Blockchain*. Por isso, foi necessário criar uma lógica para utilizar dados contextuais do *Blockchain* dentro do *type_handler*. O *type_handler* para o tipo *address* foi implementado para manipular apenas endereços existentes, ou seja, endereços de outros usuários e contratos publicados. É importante citar que foi dado uma atenção especial neste componente, pois algumas vulnerabilidades, que são relacionadas à interação entre contratos, dependiam diretamente da geração de endereços válidos.

Cada *type_handler* deve conter a implementação de um conjunto de métodos que são úteis na geração de inputs pelas estratégias de *fuzzing*. Os seguintes métodos devem ser implementados em cada *type_handler*:

- `LoadSeedsAndChooseOneRandomly(seeds common.Seeds)`: Lê do arquivo de configuração as seeds iniciais para execução do *fuzzer*. Utiliza o método `Ddeserialize()` para converter em um tipo válido;
- `Serialize()`: Converte um valor do tipo do *type_handler* no formato texto;
- `Deserialize(value string)`: Converte do formato texto em um valor do tipo do *type_handler*;
- `Generate()`: Gera randomicamente um valor do tipo do *type_handler* [1]; e
- `GetMutators()`: Retorna a lista de funções de mutadores que irão realizar pequenas mudanças no valor do tipo do *type_handler* [2].

Com todos os componentes implementados, foi possível criar as diferentes estratégias de geração de *inputs* que serão descritas a seguir.

3.4.4 Estratégias de *Fuzzing*

Para atingir um dos objetivos desse projeto, o componente *fuzzer_leader* foi criado de forma que generalize a execução de um estratégia de *fuzzing* e possibilite a criação de novas estratégias para o projeto. A Figura 3.12 mostra a representação deste componente e as implementações das estratégias de *fuzzing* escolhidas para a primeira parte desta pesquisa.

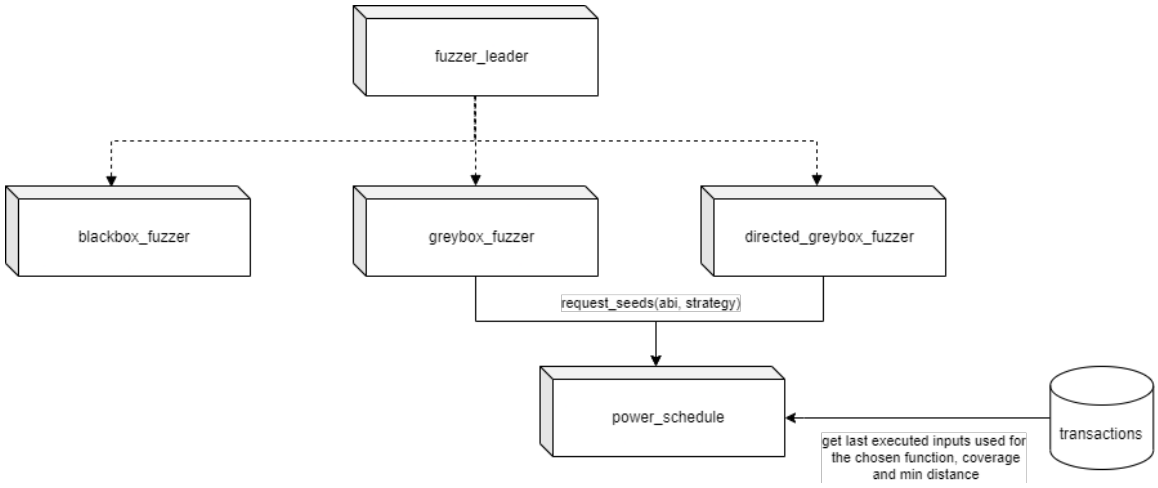

Figura 3.12: Componente *fuzzer_leader*.

[1] Utilizado na estratégia de *blackbox fuzzing*
[2] Utilizado nas estratégias *greybox fuzzing* e *directed greybox fuzzing*

Todas as estratégias de *fuzzing* devem implementar o método `GenerateInput()`, de acordo com a listagem de código a seguir. O método implementado deve receber o identificar do método, onde poderá buscar no banco de dados informações sobre ele como a sua especificação ABI, e deve retornar a lista de parâmetros do método alvo preenchidos com valores gerados pela estratégia de *fuzzing*.

```
type Fuzzer interface {
    GenerateInput(functionId string) ([]interface{}, error)
}
```

Assim, as seguintes estratégias foram escolhidas e implementadas neste projeto:[3]

Blackbox Fuzzing : geração de *inputs* totalmente aleatórios;

Greybox Fuzzing : geração de *inputs* que priorize uma maior cobertura atingida pelos testes; e

Directed Greybox Fuzzing Focado em Instruções Críticas: geração de *inputs* que priorize a aproximação de instruções críticas

A estratégia *blackbox fuzzing* utiliza uma abordagem totalmente randômica. Primeiro, ela extrai a lista de parâmetros da especificação ABI respetivo do método escolhido. E para cada parâmetro, o valor daquele *input* é gerado randomicamente utilizando principalmente o método `Generate()` do componente *type_handler* do tipo daquele parâmetro.

Já a estratégia *greybox fuzzing*, baseado nas ferramentas AFL [55] e AFLFast [56], utiliza dados das transações anteriores para geração de novos *inputs*. Em cada transação feita, é feito um análise sobre a cobertura atingida—conforme descrita na Sessão 3.3.3. Nesta análise, é computado a variação de cobertura atingida naquele momento, ou seja, é armazenado no banco de dados todas as instruções executadas para calculo da cobertura total atingida e se compara com a cobertura atingida naquela transação. Com este dados, é possível implementar o componente *power_schedule* [56], definido no estudo do AFLFast [56], que escolhe uma transação entre aquelas que obtiveram maior variação de cobertura e retorna os *inputs* utilizados naquela transação. Com estes *inputs*, esta estratégia utiliza o método `GetMutators()` e escolher um método para realizar pequenas mudanças nesses *inputs* de forma que gere outros inputs semelhantes e que possivelmente exploram partes próximas do que já foi explorado no *smart contract*.

A estratégia *directed greybox fuzzing*, baseado na implementação feita pela ferramenta AFLGo [57], tenta aproximar a execução do *smart contract* das instruções críticas definidas na configuração do *fuzzer*. Depois de ter os parâmetros e os tipos do método que

[3]Todas as estratégias utilizaram a especificação ABI para gerar *inputs* que sejam válidos para o contrato, sendo classificados como estratégias *grammar-based*.

Instrução Crítica	Utilização
CALL	Cria uma transação regular
SELFDESTRUCT	Inutiliza um contrato
CALLCODE	Chamada de funções em outros contatos dentro do contexto do contrato **origem**
DELEGATECALL	Chamada de funções em outros contatos dentro do contexto do contrato **destino**

Tabela 3.2: Instruções Críticas

devem ser testado, o *fuzzer* recupera todas as transações realizadas com a aquele método e os *inputs* utilizados nessas transações. O componente *power_schedule* também é utilizado para ordenar as transações com base em dois parâmetros com pesos diferentes:

- Instruções críticas executadas (peso 0.9); e
- Redução na distância de alguma instruções críticas (peso 0.1).

O cálculo desse parâmetro foi representado em forma de porcentagem para serem normalizadas e ser possível fazer um cálculo com pesos. O número de instruções críticas executadas foi representado como a parcela das instruções executadas pelo total de instruções criticas encontradas em cada contrato. E as distâncias atingidas nas instruções foram representadas como a parcela do somatório das máximas distâncias de totais às instruções críticas encontradas. Assim, com esta ordenação, podemos priorizar *inputs* que mais se aproximam da execução do máximo de instruções críticas possível.

Depois que as transações forem organizadas, a estratégia de *directed greybox fuzzing* seleciona randomicamente uma das cinco primeiras transações[4] e utiliza os *inputs* utilizados naquela transação para servir de *seed* para geração do próximo *input*. Essa estratégia também utiliza o método `GetMutators()` de cada *type_handler* para recuperar a lista de funções para realizar mudanças no valor desse *input* de forma a gerar uma execução semelhante.

As instruções definidas como críticas foram extraídas pelo estudo feito por Krupp e Rossor [22]. Estas instruções críticas podem ser vistas na Tabela 3.2.

Com as estratégias de *fuzzing* definidas, o componente *fuzzer_listener* está pronto para enviar transações para o *Blockchain*. Para isso, utiliza o componente *agent* que é responsável por se comunicar como o *smart contract* das diferentes maneiras possíveis.

[4]Este parâmetro é configurável e possui o valor padrão de 5 primeiras transações.

3.4.5 Componente *agent*

Com as lógica de geração de *inputs* pronta, o que falta para a comunicação com o *smart contract* ficar completa é o componente *agent*, que tem a responsabilidade de transformar os *inputs* gerados em uma transação e enviá-la para o *smart contract*. Para explorar todas as formas de comunicação com o contrato, o componente *agent* escolhe randomicamente diversas estratégias de comunicação com o *fuzzing*. Como um *smart contract* pode ser chamado por uma aplicação ou por outro contrato, o *fuzzer* foi implementado de forma a se comunicar com o *Blockchain* de forma direta por meio de transações ou de forma indireta por meio de contratos auxiliares.

Para se comunicar de forma direta, o *agent* pode utilizar duas formas. Um delas é o envio de uma transação escolhendo o método e enviando junto com os *inputs* gerados. E a outra forma é o envio de *criptomoedas* para aquele contrato de forma que ative os métodos `receive()` ou `fallback()` do contrato. Particularmente, o método `fallback()` foi o único método explorado pelo *agent*, pois este método, além de receber uma transação monetária, pode processar dados da transação por meio da variável `msg.data`. Isso faz esse método mais provável de causar uma execução não esperada.

De forma que o *fuzzer* possa explorar contratos que possuem uma lógica de interação com outros contratos por meio dos métodos `send()`, `transfer()`, e `call`, contratos auxiliares foram implementados com métodos `fallback()` e `receive()` que ajudam a causar vulnerabilidades especificas que se aproveitam dessa interação. Além disso, alguns contratos simples com somente estes métodos implementados foram publicados no *Blockchain* na preparação do *fuzzer* para que ser utilizado em contratos que recebam algum parâmetro do tipo *address*, onde passamos o endereços destes contratos publicados.

Para explorar as vulnerabilidades de *reentrancy*, por exemplo, foi implementado um contrato auxiliar que força a reentrada do contrato alvo. Então foi feita uma lógica para chamar o contrato novamente assim que o método `fallback()` ou `receive()` é executado. A implementação pode ser vista na seguinte listagem.

```
fallback() external payable {
    balance += msg.value;
    if (!sent[globalCalledAddress]) {
        sent[globalCalledAddress] = true;
        (bool success, ) = globalCalledAddress.call(globalData);
        assert(success);
    } else {
        sent[globalCalledAddress] = false;
    }
}
```

Contratos que possuem as vulnerabilidades de *exception disorder* e *gasless send* não costumam tratar a chamada realizada para outro contrato, por isso as implementações dos contratos auxiliares foram feitas para causar, respectivamente, uma exceção ou o consumo excessivo de *gas* no método `fallback()` ou `receive()`. Assim, o contrato contrato que está sendo testado pode chamar estes contratos auxiliares e o *fuzzer* consegue detectar se houve um tratamento correto destes erros por meio dos *oracles*, que serão descritos na próxima seção.

Um exemplo de implementação dos método `fallback()` para as vulnerabilidades *exception disorder* e *gasless send* pode ser visto na próxima listagem.

```
// exception disorder
fallback() external payable {
    revert("Exception thrown");
}

// gasless send
fallback() external payable {
    int64 value = 0;
    while (value < 100) {
        value++;
    }
}
```

Assim, o componente responsável por se comunicar com o *Blockchain* está pronto e com isso, é possível descrever como é feita a detecção de vulnerabilidades no DogeFuzz, que ocorre depois de toda execução de alguma transação criada. Além poderemos checar como a implementação do *fuzzer* foi desenvolvida de forma a explorar melhor estas vulnerabilidades.

3.5 Oracles

A detecção de vulnerabilidades foi baseada na proposta feita por outro estudo apresentado por Bo Jiang et al. [4], onde apresentou a ferramenta *ContractFuzzer*. Esta ferramenta foi implementada com uma estratégia de *fuzzing* bem simples e bem semelhante a estratégias *blackbox fuzzing* deste projeto. Mas essa ferramenta também implementou um lógica interessante de detecção de vulnerabilidades. Primeiro, foi feito uma instrumentação da EVM para que se detecte eventos que ocorrem durante a execução de um contrato, os componentes que fazem esta detecção foram chamados de *event_instr* e foram exportados para este projeto. Com o resultado dos *oracles* de cada execução, o *ContractFuzzer*

determina algumas regras simples que, se ocorrerem, a ferramenta reporta que alguma vulnerabilidade ocorreu.

3.5.1 Instrumentação da EVM

Para a implementação dos *event_instr*, é preciso instrumentar a EVM de forma que os *event_instr* consigam extrair informações importantes sobre a execução de uma transação. Como existe a interação entre contratos, o trace das chamadas é construída para que possa ser analisado pelos *event_instr*. Em cada chamada a um contrato, também é feita uma checagem sobre (a) mudanças de estado do contrato, (b) erros que ocorreram durante a execução, (c) instruções executadas, entre outros.

Todas estas informações são armazenado em uma estrutura de dados para que possa ser acessado facilmente pelos *event_instr*. Esta checagem ocorre assim que o último contrato for executado, para que o *trace* de chamadas fique completo. Assim, os *event_instr* conseguem verificar vulnerabilidades relacionadas às interações entre contratos.

Para que seja analisada a execução de cada contrato contido no *trace*, a instrumentação foi feita para checar se houve alguns tipos de comportamentos. É checado mudanças no estado do contrato, seja ela no *storage* ou no balanço de cripto-moedas daquele contrato. Erros ocorridos e instruções executadas também são colhidos pois são necessárias para a análise de alguns *event_instr*.

Definição dos Eventos

Para que seja possível detectar as vulnerabilidades de maneira semelhante ao *ContractFuzzer*, alguns *oracles* foram implementados baseado na implementação original do estudo. Estes *event_instr* são:

- Delegate (D)
- GaslessSend (GS)
- SendOp (SO)
- ExceptionDisorder (ED)
- BlockNumber (BN)
- Timestamp (T)
- Reentrancy (R)
- StorageChanged (SC)
- EtherTransfer (ET)

O *event_instr Delegate* é ativado quando ocorre a execução da instrução $DELEGATECALL$, onde a instrução utilizará um valor que foi recebido como parâmetro do método executado naquela transação. Se o *trace* de execução possuir mais de um contrato, apenas o contrato alvo dos testes deverá ser checado quanto à execução da instrução mencionada.

O *event_instr GaslessSend* é ativado quando ocorre um erro de falta de *gas* durante alguma chamado do *trace* de execução e se esse erro ocorreu durante a execução do método `send()` da linguagem *Solidity*. Como este método não possui um `OPCODE` específico, a detecção do uso deste método é feito com base no *gas limit* dessa chamada (o método `send()` sempre terá o limite de 2300) e no parâmetro de entrada da chamada (o método `send()` sempre iniciará com o valor de entrada zero). De uma maneira semelhante, o *oracle SendOp* é ativado quando é detectado que o método `send()` foi executado. O *event_instr ExceptionDisorder* é ativado quando ocorrer uma exceção em qualquer chamada do *trace* de execução, e esta exceção não ser propagada para chamada ao primeiro contrato do *trace*, ou seja, o contrato que está sendo testado pelo *fuzzer*.

Os *event_instr BlockNumber* e $Timestamp$ são ativados quando há a ocorrência dos `OPCODE`s $TIMESTAMP$ e $NUMBER$, respectivamente, na lista de instruções executadas. Estes `OPCODE`s são utilizados para recuperar os dados relacionados ao processamento daquela transação: o tempo de criação e o número do bloco que conterá a transação criada. O *event_instr Reentrancy* é ativado quando, analisando o trace de execução, é possível checar uma reentrada ao contrato que está sendo testado, ou seja, se o contrato foi chamado mais de um vez dentro de uma mesma transação. O *event_instr StorageChanged* é ativado quando há a execução do `OPCODE` $SSTORE$, que é responsável pela mudança de estado do *storage*, ou seja, a memória que é persistente entre execuções daquele *smart contract*. Finalmente, o *event_instr* $EtherTransfer$ é ativado quando, analisando o trace de execução, é checado se o valor de `msg.value` da chamada é maior que zero. Assim, é verificado que houve uma transferência de *ether* entre os contratos chamados.

3.5.2 Definição dos *Oracles*

Como mencionado anteriormente, em toda execução de uma transação será gerado uma lista de *event_instr* que foram detectados naquela execução. Com esta lista, chamada de *execution_snapshot*, podemos checar as vulnerabilidades de maneira semelhando ao que foi feito no estudo do *ContractFuzzer*.

A Tabela 3.3 apresenta as vulnerabilidades exploradas nesse projeto e seu respectivo identificador no registro SWC. Além disso, é possível notar quais combinações de *event_instr* deverão ser ativados para que a vulnerabilidade seja detectada.

É possível notar na tabela que as vulnerabilidades de *Dangerous Delegate Call*, *Gasless Send*, *Exception Disorder* são detectados utilizando apenas a ativação de seus respetivos

Oracle	Registro SWC	Combinação de Eventos
Reentrancy	SWC-108	R **AND** (SC **OR** ET **OR** SO)
Dangerous Delegate Call	SWC-112	D
Gasless Send	SWC-128	ED
Exception Disorder	SWC-113	GS
Number Dependency	SWC-120	BN **AND** (SC **OR** ET **OR** SO)
Timestamp Dependency	SWC-120	T **AND** (SC **OR** ET **OR** SO)

Tabela 3.3: Vulnerabilidade Esploradas pelo DogeFuzz

oracles (*Delegate*, *GaslessSend*, e *ExceptionDisorder*). Já as vulnerabilidades de *Reentrancy*, *Number Dependency*, *Timestamp Dependency* dependem da combinação de seus respetivos *oracles* (*Reentrancy*, *BlockNumber*, e *Timestamp*), com outros relacionados a interação entre contratos como a transferência de cripto-moedas, mudanças no *storage* do contrato, e o uso do método *send*().

Capítulo 4

Avaliação da Pesquisa

Este capítulo descreve o experimento feito para comparar as diferentes estratégias de *fuzzing* implementadas na ferramenta DogeFuzz. O experimento é descrito por meio de questões de pesquisa que são respondidas pelos resultados obtidos com a execução dos *fuzzers* considerando um *benchmark* específico[1].

Os resultados do experimento foram analisados de forma quantitativa e qualitativa. Neste experimento, visamos analisar as implementações feitas em cima da ferramenta DogeFuzz. Particularmente, analisaremos o desempenho da estratégia *directed greybox fuzzing* direcionada a instruções críticas, que foi apresentada neste trabalho.

4.1 Esboço do Experimento

Esta seção descreve o experimento utilizando a abordagem GQM (*goal, questions, and metrics*). O objetivo central deste estudo empírico é comparar as diferentes estratégias de *fuzzers* para identificar vulnerabilidades em *smart contracts*. As próximas seções descrevem as questões de pesquisa e as métricas utilizadas.

4.1.1 Questões de Pesquisa

Para compreender melhor o eficácia das estratégias de *greybox fuzzing* e *directed greybox fuzzing* na detecção de vulnerabilidades, comparamos essas abordagens com a implementação de uma estratégia de *blackbox fuzzing* (geração de *inputs* totalmente randômico), que se aproxima da implementação da ferramenta *ContractFuzzer*. Portanto, iremos explorar as seguintes questões de pesquisa:

RQ1: A implementação da estratégia de *greybox fuzzing* ajudou no aumento da cobertura atingida pelo DogeFuzz? O objetivo desta questão de pesquisa

[1] https://github.com/dogefuzz/benchmark

é compreender se a implementação desta estratégia aumenta cobertura da ferramenta DogeFuzz, ou seja, se conseguiu explorar mais o *smart contract* do que uma estratégia *blackbox*.

RQ2: A implementação da estratégia de *directed greybox fuzzing* ajudou na saturação das execução das instruções críticas? O objetivo desta questão de pesquisa é compreender se a implementação desta estratégia executa mais instruções consideradas críticas em comparação com uma estratégia de *blackbox fuzzing*.

RQ3: As implementações das estratégias de *directed greybox fuzzing* e *greybox fuzzing* impactam na quantidade de inputs que serão gerados? O objetivo desta questão de pesquisa é compreender se estratégias mais avançadas de *fuzzing* levam a algum impacto na performance do DogeFuzz em termos de geração de novos inputs.

RQ4: As estratégias de *greybox fuzzing* e *directed greybox fuzzing* melhoraram a eficácia do DogeFuzz na detecção de vulnerabilidades? O objetivo desta questão de pesquisa é compreender se estratégias mais avançadas de *fuzzing* impactam na detecção de vulnerabilidades.

RQ5: Existe alguma relação entre as métricas analisadas nas questões anteriores e a detecção de vulnerabilidades? E quais vulnerabilidades tiveram mais relação com estas métricas? O objetivo desta questão de pesquisa é compreender se as métricas colhidas possuem influência no número de vulnerabilidades encontradas num mesmo contrato, quando for executado utilizando diversas estratégias de *fuzzing*.

4.1.2 Métricas

Para responder a questão RQ1, o DogeFuzz foi executada com um conjunto de *smart contracts* para testar as estratégias de *greybox fuzzing* e *blackbox fuzzing*. A cobertura de cada execução foi colhida e analisada de forma a checar se houve alguma melhoria nessa métrica com a estratégia de *greybox fuzzing*.

Para responder a questão RQ2, o DogeFuzz foi executada com um conjunto de *smart contracts* para testar as estratégias de *directed greybox fuzzing* e *blackbox fuzzing*. As métricas colhidas nesse experimento estimam a quantidade de instruções executadas e as soma das distâncias que foram atingidas utilizando as duas estratégias.

Para responder a questão RQ3, o DogeFuzz foi executada com um conjunto de *smart contracts* para testar as diferentes estratégias de *fuzzing*; sendo computada a quantidade de transações que estas execuções geraram e assim comparar a performance de cada estratégia.

Para responder a questão RQ4, o DogeFuzz foi executada utilizando um conjunto de *smart contracts*, onde cada contrato foi classificado pelo tipo de vulnerabilidade—já

detectada por outra ferramenta. Com isso, é possível comparar a capacidade de detecção das implementações com esse conjunto de contratos.

Para responder a questão RQ5, foi utilizado o mesmo conjunto de contratos utilizados para responder as questões RQ2, RQ3 e RQ4. E, com isso, foram utilizadas as métricas colhidas para responder estas questões para serem analisados de sua relação com a quantidade vulnerabilidades encontradas.

4.2 Configuração do Experimento

O experimento foi planejado para responder as questões de pesquisas definidos anteriormente. Utilizamos uma estrutura em *Docker* para facilitar a reprodução dos experimentos. A escolha dos contratos que compõem o experimento foi feita baseada nos experimentos feitos pelo estudo do *ContractFuzzer*, mas para isso foi preciso realizar um filtro durante a carga de contratos.

4.2.1 Reprodução e Ambiente de Execução

Um dos principais características de um boa pesquisa é a capacidade desse estudo de ser reproduzido por outros pesquisadores. Por isso, foram escolhidas algumas ferramentas que facilitam esta execução, desde da linguagem de programação até o ambiente de execução. E para demostrar como os resultados foram gerados, foram descritos também de forma que outros pesquisadores possam executar de forma semelhante ao que foi feito nesse estudo.

Com o intuito de utilizar ferramentas de análise de dados, a linguagem de programação *Python* foi escolhida para criar o *script* de *benchmark* desse estudo. Esta linguagem possui diversas ferramentas que facilitam a manipulação de estruturas de dados, como JSON e CSV. Além disso, *Python* é um linguagem bastante popular na comunidade de programação e relativamente simples de ser utilizada e executada comparada com outras linguagens de programação. Para o experimento receber informações sobre a execução do *fuzzer*, a infraestrutura inicia um servidor HTTP em uma *thread* paralela que recebe, por um *webhook*, o resultado da execução via requisição HTTP.

Para garantir a facilidade de reprodução, o experimento foi executado dentro de um contêiner *Docker* e se aproveitou a estrutura do DogeFuzz, que também utiliza *Docker*, de forma que a comunicação via HTTP entre a infraestrutura do experimento e o *fuzzer* seja configurada de maneira simples. E foi utilizado a ferramenta *Docker Compose* para gerenciar os containers que serão executados.

O experimento foi executados diversas vezes com diferentes parâmetros, mas todas as execuções foram feitas utilizando o mesmo computador, com o intuito de obtermos os resultados mais estáveis possíveis. Este computador possui as seguintes especificações:

- Sistema Operacional: Linux 64-bits via WSL2 (virtualizado em um PC Windows)
- Processador: AMD Ryzen 5 5600X
- Memória RAM: 32 GB DDR4
- Memória SSD: 500GB NVMe WD_BLACK SN750

Com isso, o ambiente de execução foi configurado com o intuito de ser reproduzido por outros pesquisadores. E para coletar as métricas de execução e responder às questões de pesquisa definidos anteriormente, houve um esforço para selecionar os contratos que são executados pelo *fuzzer*.

4.2.2 Escolha dos Contratos

Para curar um conjunto de contratos estáveis, foi feito uma análise em cima dos contratos utilizados no estudo do *ContractFuzzer*, que contabilizaram 459 contratos. Este contratos podem ser encontrados no repositório da pesquisa original [2]. No mesmo repositório foi categorizado quais tipos de vulnerabilidades cada contrato possui.

É importante citar que apenas os contratos com alguma vulnerabilidade estavam no repositório, e não os contratos "saudáveis" utilizados no experimento feito por Bo Jiang et al. Por isso, apenas 459 contratos dos 9.960 contratos do experimento original foram encontrados e analisados. Além disso, apenas contratos com versão até 0.4.26 da linguagem *Solidity* foram utilizados no experimento, pois era a versão mais atual da linguagem no momento do estudo original.

Nessa lista de contratos, foram feitas algumas filtragens para possibilitar a execução destes contratos no DogeFuzz. A primeira filtragem foi feita com base no componente que compila dos contratos no DogeFuzz. Apenas os contratos que puderam ser compilados foram utilizados no nosso experimento. A segunda filtragem foi feita na execução do contrato pelo DogeFuzz. Como ocorreram alguns problemas de compatibilidade entre o contrato e a ferramenta *Vandal*, alguns contratos também precisaram ser removidos do *benchmark*.

Portando, o nosso *dataset* usado no experimento contempla 305 contratos. Todos estes contratos possuem algum tipo de vulnerabilidade e foram organizados quanto a isso. O experimento utiliza esta classificação como parâmetro para estimarmos a taxa de detecção das estratégias de *fuzzing* medidos pela ferramenta DogeFuzz.

[2] https://github.com/gongbell/ContractFuzzer

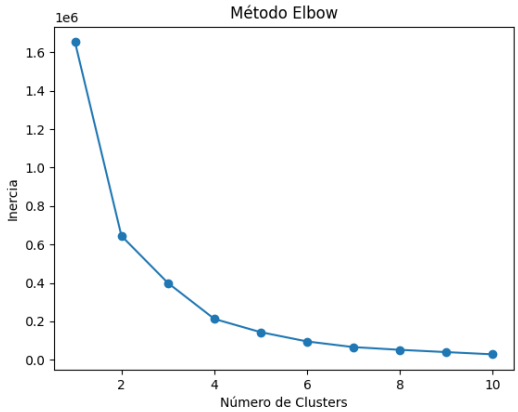

Figura 4.1: Método Elbow

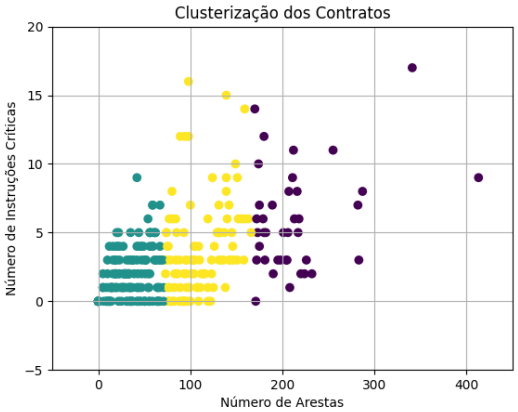

Figura 4.2: Clusterização dos Contratos

Grupos	Cor	Contratos
Grupo A (Complexidade Baixa)	Verde	107
Grupo B (Complexidade Média)	Amarelo	159
Grupo C (Complexidade Alta)	Roxo	40

Tabela 4.1: Categorização de Contratos segundo Clusterização

4.2.3 Caracterização dos Contratos

Os contratos utilizados no experimento possuem diferentes características e estruturas de código. Para analisá-los de forma mais detalhada, foi preciso categorizar os contratos conforme dois parâmetros: a estrutura do contrato e a classificação quanto ao tipo de vulnerabilidade que ele apresenta.

Para classificar os contratos quanto à complexidade da sua estrutura, foi necessário um cálculo de clusterização utilizando o algoritmo *KMeans*. Para definir a estrutura de um contrato, foi colhido duas métricas: número de vértices no CFG e número de instruções críticas presentes no *bytecode* do contrato.

Primeiro, utilizamos o método Elbow com o intuito de computar um número ótimo de *clusters*, que pode ser visto na Figura 4.1, e que foi decidido utilizar 3 clusters para classificar os contratos. O algoritmo *KMeans* foi executado com 3 *clusters* e 100 repetições, o que resultou na clusterização que pode ser vista no gráfico representado na Figura 4.2. No gráfico mostrado, os contratos foram classificados de forma que cada grupo foi representado por uma cor. E é possível visualizar na Tabela 4.1 quantos contratos foram classificados em cada grupo.

Os contratos também foram pré-classificados pelo estudo do *ContractFuzzer* a cerca da vulnerabilidade que cada contrato apresenta. Na Tabela 4.2, é possível consultar a quantidade de contratos que possui determinada vulnerabilidade e que foram utilizadas para computar algumas métricas no experimento. É importante citar que alguns contratos

Vulnerabilidade	Contratos
Reentrancy	11
Dangerous Delegate Call	15
Gasless Send	13
Exception Disorder	24
Number Dependency	71
Timestamp Dependency	110
Outras	62

Tabela 4.2: Categorização de Contratos segundo Vulnerabilidade

possuíam a vulnerabilidade de *freezing_ether*, que não foi explorado por este projeto, mas foram utilizados no experimento.

Com isso, os contratos foram caracterizados de forma possibilitar uma análise mais detalhada das métricas colhidas neste experimento. Apesar de ser bem estruturado, o *benchmark* possui algumas limitações as quais discutimos a seguir.

4.2.4 Limitações do *Benchmark*

O experimento possui algumas limitação que devem ser discutidas para que sirva como ponto de partida de novos estudos que explorem melhor o funcionamento do DogeFuzz. A quantidade de contratos pode ser considerada baixa comparado com a quantidade de contratos já publicados no *Blockchain*.

Como foi citado, os contratos utilizados foram classificados com pelo menos uma vulnerabilidades pelo artigo original que introduziu o *ContractFuzzer*, ou seja, não foram utilizados contratos que não tenham vulnerabilidades. Por isso, não foi possível fazer uma análise de falsos positivos do DogeFuzz e apenas a análise de positivos verdadeiros com base na pré-classificação feita pelo *ContractFuzzer*. Ou seja, não foi possível fazer a análise de detecção de alguma vulnerabilidade que não havia sido classificado anteriormente, que necessitaria também de uma análise qualitativas dos contratos marcados como vulneráveis e que não tinham sido classificados anteriormente.

Como poderá ser visto na métrica de performance da ferramenta (discutidos na questão RQ3) medido no experimento, foi possível notar que as diferentes estratégias de *fuzzing* não conseguiram atingir uma alta taxa de geração de inputs. Para um *fuzzer*, é importante que se gere uma quantidade substancial de inputs (na casa de centenas por segundo) para que seja maior a chance de executar uma parte do código com alguma vulnerabilidade. Isso motivou uma análise mais aprofundada da execução do DogeFuzz. Feita esta análise, foi possível notar que o principal gargalo de execução foi o nó que compõe a rede *Blockchain*

local[3]. Um nó só consegue processar um número baixo de transações por segundo, pois, a cada bloco, um processamento é feito em cima das transações para que sejam ordenadas, na qual impacta o estado final daquele contrato depois daquele bloco. Isso limitou em muito a taxa de geração de novos inputs, onde o DogeFuzz poderia gerar vários inputs, mas não poderia enviar todos de uma vez porque o nó iria ficar sobre-carregado e não responderia de volta com os resultados dessas execuções em tempo hábil.

Além disso, os contratos são compilados com uma versão mais antiga da linguagem *Solidity* (até 0.4.26), e contratos atuais são compilados com a versão 0.8+ da linguagem. Isso restringe os testes da ferramenta o DogeFuzz quanto à compatibilidade com versões mais recentes do *Solidity*—o DogeFuzz pode, inclusive, detectar algo que possa ter sido resolvido sintaticamente com melhorias da linguagem a longo do tempo.

Apesar das limitações, tais decisões foram necessárias, para permitir uma comparação mais fidedigna com o que foi reportado previamente na literatura. Pretendemos mitigar essas limitações como trabalhos futuros.

4.3 Análise Quantitativa

Esta seção apresenta os resultados quantitativos do experimento realizado, respondendo as questões de pesquisa definidas anteriormente. Com o conjunto de contratos definidos, foi possível planejar um experimento que testa diferentes estratégias de *fuzzing*, variando o tempo de execução para analisarmos o comportamento do *fuzzing* em diferentes configurações.

4.3.1 Procedimentos

Para explorar o DogeFuzz, foi feito um planejamento do experimento de forma que o mesmo explore as três estratégias de *fuzzing* implementadas: *blackbox*, *greybox* e *directed greybox fuzzing*. Cada estratégia foi executada por um período de tempo definido e checaremos as taxas de detecção destas implementações. Além disso, algumas métricas foram coletadas (conforme discutido anteriormente) para ajudar nas análises de cada estratégia de *fuzzing*. Os resultados do experimento estão disponíveis em um repositório público: https://github.com/dogefuzz/results.

Em linhas gerais, o experimento possibilita comparar as estratégias de *fuzzing* que foram implementadas neste projeto. Em particular, é possível comparar as técnicas de

[3]Recentemente, um dos principais problemas enfrentados pela industria de *Blockchain* é a escala e performance, que estão sendo trabalhados com novas camadas em cima do *Blockchain* (Layer 2) e novas tecnologias como o *Zero-Knowledge Proofs*

Estratégia	1m	5m	10m
blackbox	39.11%/contract	43.73%/contract	44.39%/contract
greybox	39.25%/contract	43.86%/contract	44.46%/contract
directed greybox	39.91%/contract	43.96%/contract	44.57%/contract

Tabela 4.3: Máxima Cobertura Atingida pelas Estratégias de *Fuzzing* em Diferentes Tempos de Execução

Estratégia	1m	5m	10m
blackbox	11.77%/tx	12.03%/tx	12.08%/tx
greybox	11.75%/tx	12.14%/tx	12.04%/tx
directed greybox	11.85%/tx	12.18%/tx	12.15%/tx

Tabela 4.4: Cobertura Média Atingida pelas Estratégias de *Fuzzing* em Diferentes Tempos de Execução

fuzzing avançadas em relação a um estratégia *blackbox* de *fuzzing*, que consiste na simples geração randômica de inputs ao *smart contract*.

O tempo de execução é fundamental para a análise de uma estratégia de *fuzzing*. Como a geração de inputs ocorre de maneira aleatório, ou semi aleatória, e algumas estratégias de *fuzzing* se adaptam conforme o tempo de execução avança, o DogeFuzz precisa executar por um período de tempo para que se possa notar alguma diferença entre as estratégias utilizadas. Para este experimento, as estratégias exercitam cada contrato pelos períodos de tempo de 1, 5 e 10 minutos. Coletamos os respectivos resultados após cada execução.

4.3.2 RQ1: Cobertura

Para computar a cobertura atingida pelas estratégias de *fuzzing*, duas métricas foram coletadas em cada execução. Primeiro, a cobertura máxima atingida foi coletada para checar quantas instruções do programa foram exploradas. Segundo, estimamos a métrica de cobertura média atingida, que avalia a média de cobertura que cada transação atingiu.

A relação da máxima cobertura atingida na execução de cada estratégia de *fuzzing* em um período fixo de tempo pode ser vistos na Tabela 4.3. Pode-se notar que os resultados foram bastante semelhantes, mas que com um tempo de execução maior (mais de 5 minutos), esta métrica de cobertura máxima foi melhorando, o que prova que quanto mais tempo durar a execução, mais chances do *fuzzer* explorar novas partes do programa, independentemente da estratégia de *fuzzing* utilizada.

Na Tabela 4.4, podemos ver a relação de cobertura média atingida pelas execuções dos *smart contracts*. Aqui, foi possível observar que a diferença de cobertura média atingida pelas estratégias foi praticamente nula. As estratégias de *fuzzing* mais avançadas tiveram uma melhoria pequena em relação à estratégia de *blackbox fuzzing*.

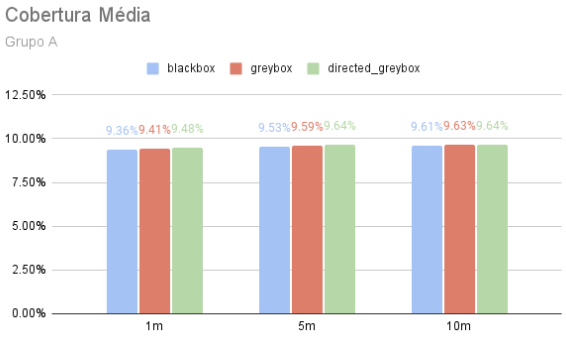

Figura 4.3: Cobertura Média (Grupo A)

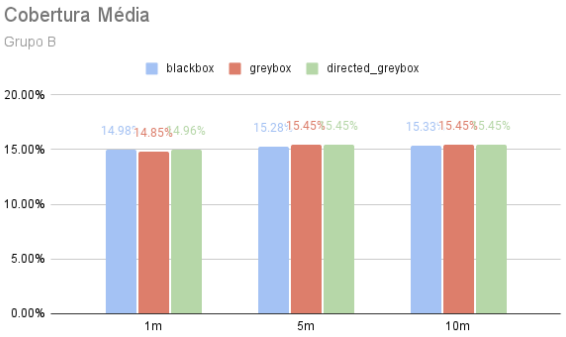

Figura 4.4: Cobertura Média (Grupo B)

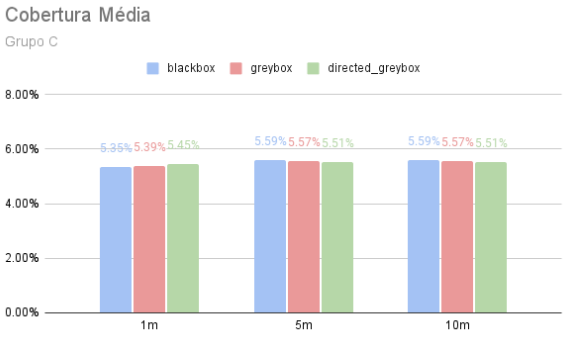

Figura 4.5: Cobertura Média (Grupo C)

Para analisarmos melhor a relação das estratégias implementadas e a cobertura atingida, utilizamos a classificação de contratos feita no sub-seção 4.2, com base na complexidade do contrato. Assim, é possível analisar a execução da ferramenta DogeFuzz em contratos de diferentes tipos de complexidade e tamanho. Como pode ser visto nas Figuras 4.3, 4.4, 4.5, também não houve uma diferença significativa entre as estratégias mesmo executando em diferentes conjuntos de contratos.

Estes resultados nos mostram que a estratégia de *greybox fuzzing* implementada, que se esperava possuir um resultado melhor nos testes, não obteve um resultado tão diferente das outras estratégias em termos de cobertura de instruções. Mas apesar de pequena, é possível checar uma leve melhora na cobertura média nas estratégias de *greybox fuzzing* e *directed greybox fuzzing* em relação à estratégia de *blackbox fuzzing*, que pode impactar na performance do *fuzzer* em outras métricas quando a ferramenta for executada por muito tempo.

4.3.3 RQ2: Instruções Críticas

A análise da execução de instruções críticas foi feita em cima da quantidade de vezes que uma instrução foi executado durante a execução dos contratos no experimento. Para

Estratégia	1m	5m	10m
blackbox	10.99 hits/tx	11.32 hits/tx	11.09 hits/tx
greybox	11.44 hits/tx	11.68 hits/tx	11.15 hits/tx
directed greybox	10.91 hits/tx	11.46 hits/tx	11.59 hits/tx

Tabela 4.5: Média de Instruções Críticas Executadas pelas Estratégias de *Fuzzing* em Diferentes Tempos de Execução

obter um resultado mais estável, foi computado a média de instruções executadas por transação para normalizar os resultados entre estratégias que conseguiram criar maiores quantidades de transações que as outras. Na Tabela 4.5 podemos ver que as estratégias geraram resultados praticamente iguais. Porém, é possível notar que a estratégia de *directed greybox fuzzing* apresentou um crescimento dessa taxa de execuções conforme o tempo de execução aumentou.

Para realizar uma análise mais aprofundada, utilizamos também a classificação dos contratos com base na complexidade das estruturas dos contratos que foi feito Seção 4.2. Observando as Figuras 4.6, 4.7, e 4.8, podemos observar um comportamento mais estável nas três classificações de contratos conforme a sua complexidade. Em todos os grupos, a estratégia de *greybox fuzzing* teve uma maior taxa de execução de instruções críticas em períodos de 1 minuto, mas a partir de 5 minutos, a estratégia de *directed greybox fuzzing* se sobressaiu sobre as outras estratégias.

Portanto, com base nas métricas computadas, é possível conjecturar que, independentemente da complexidade do contrato executado, a estratégia de *directed greybox fuzzing* leva a um aumento da execução de instruções críticas, conforme o tempo de execução do DogeFuzz aumente. Além disso, nota-se que a estratégia de *greybox fuzzing* também apresentou uma boa taxa de execução em comparação com a estratégia de *blackbox fuzzing* em contratos de de complexidade baixa e média.

4.3.4 RQ3: Performance da DogeFuzz

A métrica de quantidade de transação geradas foi utilizado para medir a performance de cada estratégia. Aqui, o objetivo é checar se estratégias mais complexas de *fuzzing* têm impacto na quantidade de transações geradas num período de tempo. Aqui computamos a média de transações que foram executados por minuto, para ser possível comparar a performance das estratégias com diferentes períodos de tempo de execução.

Analisando a Tabela 4.6, é possível observar a média de transações por minuto de cada estratégia—todas as estratégias possuem uma performance bem semelhante. Porém, a partir de 5 minutos, podemos observa-se uma leve diminuição da performance do DogeFuzz com as estratégias de *greybox fuzzing* e *directed greybox fuzzing*.

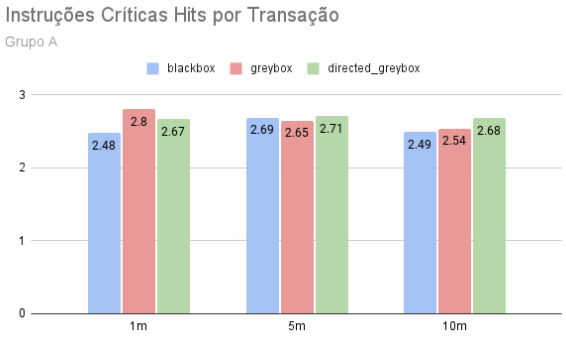

Figura 4.6: Instruções Críticas Hits por Transação (Grupo A)

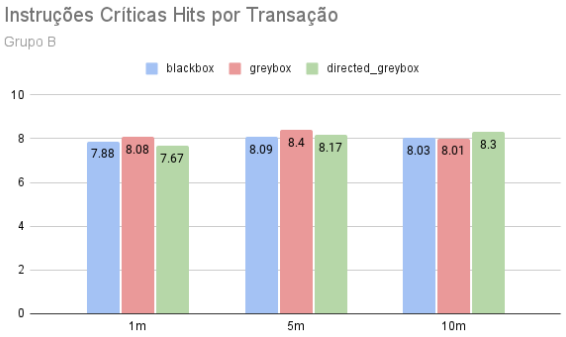

Figura 4.7: Instruções Críticas Hits por Transação (Grupo B)

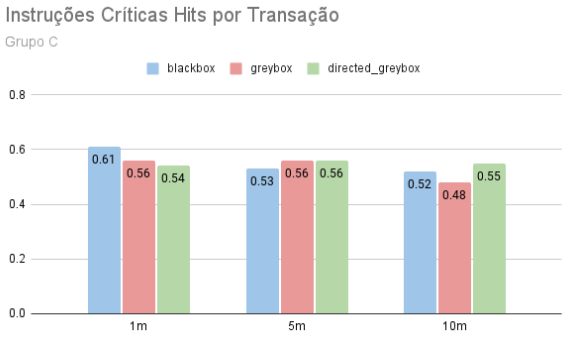

Figura 4.8: Instruções Críticas Hits por Transação (Grupo C)

Estratégia	1m	5m	10m
blackbox	34.51 txs/min	32.43 txs/min	32.10 txs/min
greybox	34.99 txs/min	31.56 txs/min	30.48 txs/min
directed greybox	34.61 txs/min	31.244 txs/min	31.61 txs/min

Tabela 4.6: Taxa de Geração de Inputs pelas Estratégias de *Fuzzing* em Diferentes Tempos de Execução

Estratégia	1m	5m	10m
blackbox	23.71%	29.94%	30.55%
greybox	23.26%	32.66%	34.66%
directed greybox	27.95%	29.60%	34.62%

Tabela 4.7: Taxa de Detecção das Estratégias de *Fuzzing* em Diferentes Tempos de Execução

Esses resultados sugerem que estratégias mais avançadas começam a reduzir a performance da ferramenta DogeFuzz conforme o tempo vai passando. Isso pode ser explicado pelo fato destas estratégias terem que checar o histórico de transações e processá-los a cada ciclo de geração de novos inputs. Por isso, em tempos maiores, essas análise pode ser custosa à performance da ferramenta e impactar na eficiência do *fuzzer*, ou seja, no tempo necessário para identificar vulnerabilidades.

4.3.5 RQ4: Vulnerabilidades Encontradas

Para analisar a eficácia do DogeFuzz utilizando as diferentes estratégias implementadas, foram coletadas as taxas de detecção de vulnerabilidades de cada estratégia. A taxa de detecção foi computada a partir das vulnerabilidades detectadas pela ferramenta e a pré-classificação dos contratos a partir da vulnerabilidade encontrada no estudo original do *ContractFuzzer*. Aqui, os resultados também foram separados por tipos de complexidade de contratos para analisarmos as performance de detecção em cada tipo de contrato.

Observando a Tabela 4.7, nota-se que, em um tempo curto de execução, a técnica de *directed greybox fuzzing* se sobressaiu, e conseguiu uma taxa de detecção maior do que as outra estratégias. Mas, conforme o tempo de execução aumentou para 5 ou 10 minutos, todas as estratégias apresentaram uma melhora na taxa de execução. Isso sugere a importância do tempo de execução nas análises. Além disso, a Tabela 4.7 aponta que as estratégias de *greybox fuzzing* e *directed greybox fuzzing* se sobressairam em relação à estratégia de *blackbox fuzzing* conforme o tempo de execução foi aumentando.

Para analisar de forma mais detalhada, computamos as métricas de detecção nos diferentes grupos de contratos classificados na Seção 4.2. Os resultados são apresentados nas Figuras 4.9, 4.10, e 4.11. Observando os resultados obtidos, é possível notar que a diferença de detecção ocorreu mais nos contratos de complexidade simples, onde com um período de 1 minuto, a estratégia de *directed greybox fuzzing* se sobressaiu, e com um período de 5 e 10 minutos, a estratégia de *greybox fuzzing* teve um resultado superior. Pode-se dizer que a taxa de detecção nos contratos de complexidade baixa impactou na taxa de detecção média mostrada na Tabela 4.7, pois no caso de detecção nos contratos

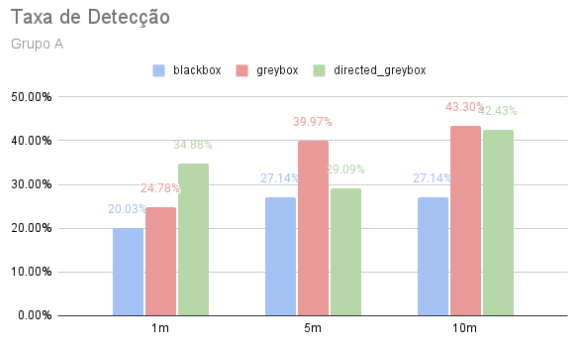

Figura 4.9: Taxa de Detecção (Grupo A)

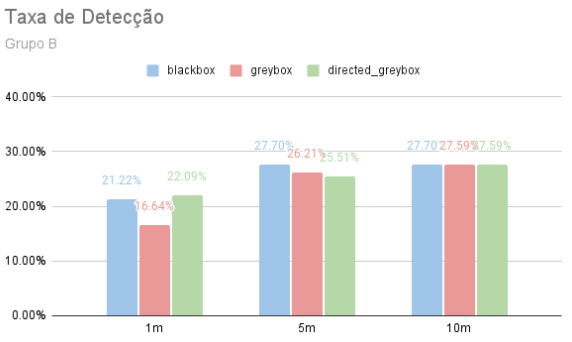

Figura 4.10: Taxa de Detecção (Grupo B)

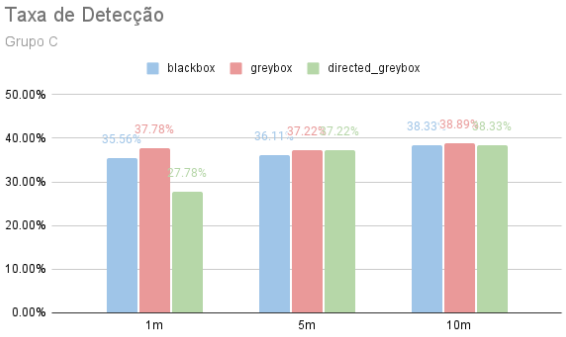

Figura 4.11: Taxa de Detecção (Grupo C)

de complexidade média e alta, a taxa de detecção das ferramentas foram bem parecidas entre as estratégias, principalmente em tempos de execução maiores.

Foi possível observar que ouve uma melhoras significativa na taxa de detecção de vulnerabilidades, quando foi utilizado uma estratégia mais avançada de fuzzing. Particularmente, é possível citar a estratégia de *directed greybox fuzzing*, que obteve um melhor resultado quando foi executado em um curto período de tempo; bem como a estratégia de *greybox fuzzing* que melhorou bastante a sua taxa de detecção quando a execução foi mais longa, se tornando a estratégia com melhor taxa de detecção (de 32.66%).

4.3.6 RQ5: Métricas x Vulnerabilidades Encontradas

Com intuito de checar a relação das métricas computadas e as vulnerabilidades encontradas, os contratos foram separados conforme a vulnerabilidade apresentada pelo estudo do *ContractFuzzer*. Dessa forma, as métricas foram recalculadas sobre os sub-conjunto de contratos que possuíam determinada vulnerabilidade, tornando a análise mais exata pois apenas as métricas referentes aos contratos com certa vulnerabilidade são levadas em consideração.

Taxa de Detecção	CM	MEIC	TE
1.00000000	0.45530829	0.09342296	0.81872538
0.45530829	1.00000000	0.68589278	0.36460618
0.09342296	0.68589278	1.00000000	-0.0422412
0.81872538	0.36460618	-0.0422412	1.00000000

Tabela 4.8: Matriz de Correlação entre Métricas e Taxa de Detecção de *Dangerous Delegate Call*

A Tabela 4.2 mostra a quantidade de contratos classificados em cada vulnerabilidade, e como mencionado, as métricas para somente os contratos que tenham sido classificado com a vulnerabilidade em questão. Para computar a correlação entre as métricas e taxa de detecção, foi utilizado a medida de correlação *Pearson product-moment correlation coefficient* (PPMCC). As seguintes métricas foram escolhidas para checar a correlação entre elas e a detecção de cada tipo de vulnerabilidade:

- Cobertura Média (CM);
- Média de Execuções de Instruções Críticas (MEIC); e
- Tempo de Execução (TE).

Dangerous Delegate Call

A Tabela 4.8 mostra a correlação entre as métricas escolhidas e a taxa de detecção da vulnerabilidade *Dangerous Delegate Call*. Estas taxas foram computadas durante a execução do experimento utilizando as diferentes estratégias de *fuzzing*. Analisando a matriz de correlação, é possível notar três coisas: uma correlação forte com o tempo de execução, uma correlação moderada com a taxa de cobertura média e uma correlação fraca com a taxa média de execução de instruções críticas por transação. Isso mostra que um alto tempo de execução, e consequentemente um grande número de transações geradas, possuem impacto maior na detecção da vulnerabilidade *Dangerous Delegate Call*.

Gasless Send

A Tabela 4.9 mostra a correlação entre as métricas escolhidas e a taxa de detecção da vulnerabilidade *Gasless Send*. Estas taxas foram colhidas durante a execução do experimento, utilizando as diferentes estratégias de *fuzzing*. Analisando a matriz de correlação computada com os contratos com a vulnerabilidade *Gasless Send*, pode-se notar que houve uma correlação moderada negativa entre a taxa de detecção dessa vulnerabilidade e a cobertura média. Nota-se também uma correlação fraca entre a taxa de detecção e a taxa de execução média de instruções críticas.

Taxa de Detecção	CM	MEIC	TE
1.00000000	-0.4701351	0.13006650	-0.0000000
-0.4701351	1.00000000	-0.1179740	0.49158947
0.13006650	-0.1179740	1.00000000	0.11379761
-0.0000000	0.49158947	0.11379761	1.00000000

Tabela 4.9: Matriz de Correlação entre Métricas e Taxa de Detecção de *Gasless Send*

Taxa de Detecção	CM	MEIC	TE
1.00000000	0.06136674	-0.5264051	0.89624377
0.06136674	1.00000000	0.01652113	-0.0362486
-0.5264051	0.01652113	1.00000000	-0.3668422
0.89624377	-0.0362486	-0.3668422	1.00000000

Tabela 4.10: Matriz de Correlação entre Métricas e Taxa de Detecção de *Exception Disorder*

Exception Disorder

A Tabela 4.10 mostra a correlação entre as métricas escolhidas e a taxa de detecção da vulnerabilidade *Exception Disorder*. Estas taxas foram computadas durante a execução do experimento, utilizando as diferentes estratégias de *fuzzing*. Analisando a matriz de correlação para a taxa de detecção dessa vulnerabilidade, nota-se uma forte correlação entre o tempo de execução e uma correlação moderada negativa com a taxa de execução média de instruções críticas.

Number Dependency

A Tabela 4.11 mostra a correlação entre as métricas escolhidas e a taxa de detecção da vulnerabilidade *Number Dependency*. Estas taxas foram computadas durante a execução do experimento, utilizando diferentes estratégias de *fuzzing*. Segundo a matriz de de correlação com os contratos classificados em a vulnerabilidade *Number Dependency*, é possível checar que todas as métricas computadas tiveram uma correlação entre fraca e moderada com a taxa de detecção desta vulnerabilidade.

Taxa de Detecção	CM	MEIC	TE
1.00000000	0.34543137	0.23083599	0.34896971
0.34543137	1.00000000	0.35815643	0.78260276
0.23083599	0.35815643	1.00000000	0.02755821
0.34896971	0.78260276	0.02755821	1.00000000

Tabela 4.11: Matriz de Correlação entre Métricas e Taxa de Detecção de *Number Dependency*

Taxa de Detecção	CM	MEIC	TE
1.00000000	0.51981772	0.16536603	0.65871538
0.51981772	1.00000000	0.13696987	0.79169245
0.16536603	0.13696987	1.00000000	0.21412355
0.65871538	0.79169245	0.21412355	1.00000000

Tabela 4.12: Matriz de Correlação entre Métricas e Taxa de Detecção de *Timestamp Dependency*

Timestamp Dependency

A Tabela 4.12 mostra a correlação entre as métricas escolhidas e a taxa de detecção da vulnerabilidade *Timestamp Dependency*. Estas taxas foram computadas durante a execução do experimento, utilizando diferentes estratégias de *fuzzing*. Analisando a matriz de correlação para esta vulnerabilidade, é possível notar uma correlação moderada entre a cobertura média e o tempo de execução, e uma correlação fraca entre a taxa de detecção e o taxa de execução média de instruções críticas. O que se conclui que duas métricas (CM e TE) tiveram mais impacto na detecção dessa vulnerabilidade.

Reentrancy

A relação da detecção da vulnerabilidade *Reentrancy* com as métricas não foi possível ser feita pois a ferramenta não conseguiu detectar a vulnerabilidade em nenhum dos 10 contratos que haviam sido classificados com essa vulnerabilidade no estudo do *ContractFuzzer* durante os períodos de execuções que foram escolhidos para este experimento.

4.4 Discussão

Esta seção apresenta as lições aprendidas depois da execução dos experimentos. As métricas computadas foram essenciais para analisar o comportamento do DogeFuzz, de forma precisa e para tirar conclusões sobre o comportamento das diferentes estratégias de *fuzzing*. Além disso, esta seção também apresenta as ameaças à validade, bem como os aspectos que podem ter impactado no resultados obtidos.

4.4.1 Lições Aprendidas

Depois de executar o experimento com diferentes configurações, variando os parâmetros de tempo de execução, a estratégia de *fuzzing*, e um conjunto pré-definidos de contratos, algumas lições foram aprendidas, que foram separadas abaixo pelos tópicos explorados neste estudo

Sobre a vulnerabilidades e métricas coletadas neste estudo, as seguinte lições foram aprendidas:

- A métrica de cobertura média apresentou uma correlação moderada com as vulnerabilidades de *Dangerous Delegate Call*, *Number Dependency* e *Timestamp Dependency*;
- A métrica de instruções críticas executadas por transação apresentou uma correlação fraca entre as vulnerabilidades *Gasless Send*, *Number Dependency* e *Timestamp Dependency*; e
- O tempo de execução apresentou uma correlação forte com as seguintes vulnerabilidades: *Dangerous Delegate Call*, *Exception Disorder* e *Timestamp Dependency*.

Sobre as estratégias de *fuzzing* implementadas, as seguintes informações foram extraídas do estudo:

- Todas as estratégias de *fuzzing* conseguiram uma cobertura bem parecida do programa e com leve melhorias nas estratégias avançadas de *fuzzing*;
- A estratégia de *directed greybox fuzzing* teve uma maior taxa de execução de instruções críticas que as outras estratégias; e
- As estratégias de *greybox fuzzing* e *directed greybox fuzzing* tiveram uma taxa de detecção de vulnerabilidade maiores nos períodos de tempo explorados neste experimento (1, 5, e 10 minutos).
- Conforme o tempo de execução aumentou, as estratégias mais avançadas de *fuzzing* tiveram uma redução na performance, ou seja, na taxa de geração de inputs.

Sobre o DogeFuzz e a arquitetura implementada, as seguintes lições foram aprendidas:

- Um *fuzzer* robusto deve produzir uma grande quantidade de inputs; e
- Um *fuzzer* para *smart contract* deve possuir uma estrutura bem complexa para lidar com especifidades da arquitetura de *smart contracts* e de *Blockchain*.

Por fim, podemos concluir que o experimento foi bem sucedido em questões de explorar o comportamento do DogeFuzz de diversas maneiras e que o estudo trouxe algumas contribuições para a pesquisa de segurança em *smart contracts*.

4.4.2 Ameaças à Validade

Como qualquer outro experimento, este trabalho possui algumas características que possam impactar no resultado obtido neste estudo. Por ser uma ferramenta com comportamento aleatório, os resultados objetivos podem não ter sido tão testados de forma que os resultados convirjam em resultados mais normalizados.

O experimento utilizou um número limitado de contratos, que foram tirados de outro estudo, e por mais que sejam contratos que foram extraídos do *Blockchain Ethereum* oficial, ainda é um número pequeno de contratos que podem ser testados dentre aqueles que existe na rede *Blockchain*. Isso limitou a exploração da ferramenta entre as diferentes lógicas e padrões de implementação de *smart contracts* existentes.

Além disso, é possível citar o número de execuções que o experimento foi feito. Por motivos de performance, o experimento só foi executado apenas uma vez em cada configuração. Isso pode ter impactado nos resultados, porque o comportamento aleatório do DogeFuzz pode ter feito com que a ferramenta não explorasse bem o contrato naquela ocasião. Para explorar melhor o DogeFuzz, seria necessário a re-execução do experimento várias vezes para que os resultados possa ser normalizados.

Comparando com outros experimentos em *fuzzing* para outros tipos de programa, como os *fuzzer* que este estudo se baseou (AFL, AFLFast, e AFLGo), podemos ver que estes *fuzzers* foram testados durante períodos de até 24 horas. Neste experimento, isso não foi possível pela grande quantidade de contratos que foram utilizados neste experimento, e seria inviável testar cada contrato por um período maior que uma hora, ou seja, mais de 300 horas por configuração do experimento. E como foi visto no experimento, períodos maiores de execução impactam na taxa de detecção do DogeFuzz.

Contudo, o experimento conseguiu explorar com sucesso diversos elementos da ferramenta DogeFuzz apesar destas ameaças á validade. Isso possibilitou tirar conclusões úteis para a área de *fuzzing* para *smart contracts*.

Capítulo 5

Conclusão e Trabalhos Futuros

Neste capítulo, será discutido o conjunto de contribuições que o projeto trouxe para a área de segurança de *smart contracts* e *Blockchain*. Porém, as limitações da ferramenta também serão apresentadas de forma que os próximos passos para evoluir a ferramenta e trabalhos futuros desta área de pesquisa possam ser sumarizados para outros pesquisadores possam seguir em frente na pesquisa *fuzzing* para *smart contracts*.

5.1 Contribuições

Nesta sessão, serão apresentados as contribuições que esta pesquisa trouxe para área de pesquisa de *fuzzing* para *smart contracts*. Entre elas, podemos citar as seguintes contribuições:

- Uma arquitetura flexível para a implementação de novas estratégias de *fuzzing* e de fácil utilização;
- A apresentação de uma estratégia de *fuzzing* direcionada a instruções críticas no domínio de *smart contracts*;
- Um *benchmark* de *fuzzers* para *smart contracts*, no qual é fácil reproduzir os resultados obtidos
- Uma análise quantitativa entre a relação entre métricas (como cobertura, instruções críticas executadas, e tempo de execução) e a taxa de detecção de vulnerabilidades em *smart contracts*.

5.2 Limitações da Ferramenta

Durante o experimento, algumas limitações foram notadas na ferramenta DogeFuzz. A taxa de geração de inputs da ferramenta foi considerada baixa se compararmos com *fuzzers* de outros domínios, como o AFL. AFLFast, e AFLGo. E como o *fuzzer* se baseou em outros estudos, como ContractFuzzer [4] e Vandal [58], apenas os contratos, que estes estudos suportavam na época, foram utilizados e testados neste experimento.

Como foi mencionado, a quantidade de *inputs* gerados pelo DogeFuzz foi pequena se comparamos com *fuzzers* do estado-da-arte. Enquanto estes *fuzzer* geram centenas de milhares de inputs por minuto, este *fuzzer* para *smart contract* conseguiu uma taxa de dezenas de inputs por minuto. Isso deve pela forma que uma *smart contract* é executado na EVM e para melhorar esta taxa seria necessário uma nova reformulação da arquitetura da ferramenta.

Durante o experimento, foi notado também que alguns contratos não puderam ser executados por incompatibilidade do *smart contract* por alguns motivos. Podemos citar que alguns contratos não puderam ser compilados utilizando ferramentas de compilação que estão disponíveis atualmente. Geralmente, foram contratos com versões muito antigas ou descontinuadas da linguagem *Solidity*. Além disso, uma pequena quantidade de contratos teve que ser retirado do experimento pelo fato da ferramenta *Vandal* não se comportar bem com estes contratos, gerando *loops* infinitos ou CFG vazios.

Pelo fato da ferramenta ser uma implementação recente, ela ainda tem muito caminhos para evolução e um dos principais objetivos que este projeto foi realizado foi para possibilitar melhorias e evolução de uma maneira fácil e que possa ser continuado por outros pesquisadores. Todas estas limitações que foram citadas podem ser melhorados e serão tratadas em trabalhos futuros da pesquisa.

5.3 Trabalhos Futuros

Como foi mencionado diversas vezes neste documento, um dos principais objetivos do DogeFuzz é possibilitar a evolução para explorar novos caminhos e estratégias na análise de segurança de *smart contracts* utilizando *fuzzing*. Por isso, novas estratégias de *fuzzing* poderão ser exploradas em cima do DogeFuzz para continuar a pesquisa nessa área. Além disso, podemos citar algumas melhorias que a ferramenta DogeFuzz e o experimento que poderão complementar o estudo feito neste projeto. Além do experimento ter suas limitações que foram principalmente ligadas a quantidade de contratos e o tempo de execução, podemos citar também que a ferramenta apresentou um performance baixa em relação a ferramentas de *fuzzing* de outros domínios.

A ferramenta *DogeFuzz* foi desenhada de forma flexível para possibilitar a pesquisa de novas estratégias de *fuzzing* para *smart contracts*. E por isso, o principal trabalho futuro vai ser utilizar esta arquitetura para estudar novas estratégias e continuar a pesquisa de segurança em *smart contracts* dando apoio necessário para os pesquisadores.

E para complementar a análise feita neste estudo e as novas implementações de *fuzzer* em cima da ferramenta DogeFuzz, será necessário construir um experimento com uma vasta quantidade de contratos e que possam ser executados em períodos maiores de tempo. Além isso, selecionar contratos com versões mais recentes da linguagem *Solidity* para explorar novas funcionalidades da linguagem no experimento.

Por fim, o DogeFuzz apresentou algumas limitações que deverão ser melhoradas de forma a não impactar os resultados dos experimentos futuros. A principal mudança necessária é reformulação da arquitetura para utilizar algoritmos de distribuição de carga como líder-seguidores para reduzir o gargalo que um nó causa ao funcionamento da ferramenta como um todo. Além disso, será necessário atualizar partes da ferramenta *Vandal* para poder suportar todos os contratos que serão utilizados no experimento.

Referências

[1] Buterin, V.: *A next-generation smart contract and decentralized application platform.* 2014. https://ethereum.org/whitepaper/. 1, 7

[2] *Swc-112 - delegatecall to untrusted callee.* https://swcregistry.io/docs/SWC-112, 2023. [Online; accessed 23-June-2023]. 2

[3] *The parity wallet hack explained.* https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7, 2023. [Online; accessed 23-June-2023]. 2, 18

[4] Jiang, B., Y. Liu e W. K. Chan: *Contractfuzzer: fuzzing smart contracts for vulnerability detection.* ASE 2018: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software, páginas 259–269, 2018. 2, 17, 19, 48, 71

[5] Fioraldi, Andrea, Dominik Maier, Heiko Eißfeldt e Marc Heuse: *Afl++ combining incremental steps of fuzzing research.* Em *Proceedings of the 14th USENIX Conference on Offensive Technologies*, páginas 10–10, 2020. 2, 22

[6] Nakamoto, S.: *Bitcoin: A peer-to-peer electronic cash system.* 2008. 5, 6

[7] *Solidity documentation.* https://docs.soliditylang.org, 2023. [Online; accessed 9-April-2023]. 7

[8] *Uniswap - github.* https://github.com/Uniswap, 2023. [Online; accessed 23-June-2023]. 8

[9] *Aave - github.* https://github.com/aave, 2023. [Online; accessed 23-June-2023]. 8

[10] *Ethernodes.* https://ethernodes.org/, 2023. [Online; accessed 10-April-2023]. 8

[11] *Ethereum virutal machine - ethereum docs.* https://ethereum.org/en/developers/docs/evm/, 2023. [Online; accessed 10-April-2023]. 9

[12] Community, Ethereum: *Erp-20: Token standard.* https://eips.ethereum.org/EIPS/eip-20, 2015. [Online; accessed 30-September-2021]. 9, 11, 14

[13] Wood, Gavin *et al.*: *Ethereum: A secure decentralised generalised transaction ledger.* Ethereum project yellow paper, 151(2014):1–32, 2014. 10

[14] Community, Ethereum: *Erp-721: Non-fungible token standard.* https://eips.ethereum.org/EIPS/eip-721, 2018. [Online; accessed 30-September-2021]. 14

[15] Community, Ethereum: *Eip-1559: Fee market change for eth 1.0 chain.* `https://eips.ethereum.org/EIPS/eip-1559/`, 2019. [Online; accessed 30-September-2021]. 14

[16] Community, Ethereum: *Eip-2982: Serenity phase 0.* `https://eips.ethereum.org/EIPS/eip-2982`, 2020. [Online; accessed 30-September-2021]. 14

[17] Community, Ethereum: *Eip-1470: Smart contract weakness classification (swc).* `https://eips.ethereum.org/EIPS/eip-1470`, 2018. [Online; accessed 30-September-2021]. 14

[18] Christey, Steve, J Kenderdine, J Mazella e B Miles: *Common weakness enumeration.* Mitre Corporation, 2013. 14

[19] Luu, Loi, Duc Hiep Chu, Hrishi Olickel, Prateek Saxena e Aquinas Hobor: *Making smart contracts smarter.* CCS '16, página 254–269, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450341394. `https://doi.org/10.1145/2976749.2978309`. 15

[20] Education, Computing Research : *Core ranking.* `http://portal.core.edu.au/conf-ranks/`, 2013. [Online; accessed 27-September-2021]. 15

[21] Kalra, S., S. Goel, M. Dhawan e S. Sharma: *Zeus: Analyzing safety of smart contracts.* 25th Annual Network and Distributed System Security Symposium, {NDSS}, 2018. 17

[22] Krupp, J. e C. Rossow: *teether: Gnawing at ethereum to automatically exploit smart contracts.* 27th USENIX Security Symposium (USENIX Security 18), páginas 1317–1333, 2018. 17, 39, 46

[23] Liu, C., H. Liu, Z. Cao, Z. Chen, B. Chen e B. Roscoe: *Reguard: Finding reentrancy bugs in smart contracts.* Em *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, páginas 65–68, 2018. 17, 19

[24] I., Grishchenko, Maffei M. e Schneidewind C.: *Foundations and tools for the static analysis of ethereum smart contracts.* Computer Aided Verification., 2018. 17

[25] Permenev, A., D. Dimitrov, P. Tsankov, D. Drachsler-Cohen e M. Vechev: *Verx: Safety verification of smart contracts.* Em *2020 IEEE Symposium on Security and Privacy (SP)*, páginas 1661–1677, 2020. 17

[26] Coblenz, M.: *Obsidian: A safer blockchain programming language.* Em *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, páginas 97–99, 2017. 17

[27] Mossberg, M., F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson e A. Dinaburg: *Manticore: A user-friendly symbolic execution framework for binaries and smart contracts.* Em *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 1186–1189, 2019. 17

[28] Grech, N., L. Brent, B. Scholz e Y. Smaragdakis: *Gigahorse: Thorough, declarative decompilation of smart contracts*. Em *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, páginas 1176–1186, 2019. 17

[29] Liu, H., C. Liu, W. Zhao, Y. Jiang e J. Sun: *S-gram: Towards semantic-aware security auditing for ethereum smart contracts*. Em *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 814–819, 2018. 17

[30] Wang, H., Y. Li, S. Lin, L. Ma e Y. Liu: *Vultron: Catching vulnerable smart contracts once and for all*. Em *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, páginas 1–4, 2019. 17

[31] Nguyen, T., L. Pham, J. Sun, Y. Lin e Q. Minh: *Sfuzz: An efficient adaptive fuzzer for solidity smart contracts*. ICSE '20, página 778–788, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450371216. https://doi.org/10.1145/3377811.3380334. 17, 19

[32] Gao, Zhipeng, Lingxiao Jiang, Xin Xia, David Lo e John Grundy: *Checking smart contracts with structural code embedding*. IEEE Transactions on Software Engineering, páginas 1–1, 2020. 17

[33] Frank, Joel, Cornelius Aschermann e Thorsten Holz: *ETHBMC: A bounded model checker for smart contracts*. Em *29th USENIX Security Symposium (USENIX Security 20)*, páginas 2757–2774. USENIX Association, agosto 2020, ISBN 978-1-939133-17-5. https://www.usenix.org/conference/usenixsecurity20/presentation/frank. 17

[34] Gao, Jianbo, Han Liu, Chao Liu, Qingshan Li, Zhi Guan e Zhong Chen: *Easyflow: Keep ethereum away from overflow*. ICSE '19, página 23–26. IEEE Press, 2019. https://doi.org/10.1109/ICSE-Companion.2019.00029. 17

[35] Bhargavan, Karthikeyan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy e Santiago Zanella-Béguelin: *Formal verification of smart contracts: Short paper*. Em *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, página 91–96, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450345743. https://doi.org/10.1145/2993600.2993611. 17

[36] So, Sunbeom, Myungho Lee, Jisu Park, Heejo Lee e Hakjoo Oh: *VERISMART: A highly precise safety verifier for ethereum smart contracts*. Em *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, páginas 1678–1694. IEEE, 2020. https://doi.org/10.1109/SP40000.2020.00032. 17

[37] Wüstholz, Valentin e Maria Christakis: *Targeted greybox fuzzing with static lookahead analysis*. Em *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, página 789–800, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450371216. https://doi.org/10.1145/3377811.3380388. 17

[38] Wang, X., J. He, Z. Xie, G. Zhao e S. Cheung: *Contractguard: Defend ethereum smart contracts with embedded intrusion detection*. IEEE Transactions on Services Computing, 13(02):314–328, apr 2020, ISSN 1939-1374. 17

[39] Zhang, Mengya, Xiaokuan Zhang, Yinqian Zhang e Zhiqiang Lin: *TXSPECTOR: uncovering attacks in ethereum from transactions*. Em Capkun, Srdjan e Franziska Roesner (editores): *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, páginas 2775–2792. USENIX Association, 2020. `https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-mengya`. 17

[40] D., Park, Zhang Y. e Rosu G: *End-to-end formal verification of ethereum 2.0 deposit smart contract*. Computer Aided Verification, 2020. 17

[41] Zhang, Pengcheng, Jianan Yu e Shunhui Ji: *ADF-GA: data flow criterion based test case generation for ethereum smart contracts*. Em *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, páginas 754–761. ACM, 2020. `https://doi.org/10.1145/3387940.3391499`. 17

[42] Almashaqbeh, Ghada, Allison Bishop e Justin Cappos: *Abc: A cryptocurrency-focused threat modeling framework*. Em *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, páginas 859–864, 2019. 17

[43] Chen, Jiachi, Xin Xia, David Lo, John Grundy, Xiapu Luo e Ting Chen: *Defectchecker: Automated smart contract defect detection by analyzing evm bytecode*. IEEE Transactions on Software Engineering, página 1–1, 2021, ISSN 2326-3881. `http://dx.doi.org/10.1109/TSE.2021.3054928`. 17

[44] Xue, Yinxing, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye e Tianyong Peng: *Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts*. Em *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 1029–1040, 2020. 17

[45] Feng, Yu, Emina Torlak e Rastislav Bodik: *Summary-based symbolic evaluation for smart contracts*. Em *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 1141–1152, 2020. 17

[46] Yang, Zhiqiang, Han Liu, Yue Li, Huixuan Zheng, Lei Wang e Bangdao Chen: *Seraph: Enabling cross-platform security analysis for evm and wasm smart contracts*. Em *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, página 21–24, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450371223. `https://doi.org/10.1145/3377812.3382157`. 17

[47] Li, Yue: *Finding concurrency exploits on smart contracts*. Em *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, páginas 144–146, 2019. 17

[48] Wang, Bin, Han Liu, Chao Liu, Zhiqiang Yang, Qian Ren, Huixuan Zheng e Hong Lei: *Blockeye: Hunting for defi attacks on blockchain*, 2021. 17

[49] Gao, Zhipeng: *When deep learning meets smart contracts.* Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Dec 2020. http://dx.doi.org/10.1145/3324884.3418918. 17

[50] Chen, Jiachi: *Finding ethereum smart contracts security issues by comparing history versions*, 2020. 17

[51] So, Sunbeom, Seongjoon Hong e Hakjoo Oh: *Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution.* Em *30th USENIX Security Symposium (USENIX Security 21)*, páginas 1361–1378. USENIX Association, agosto 2021, ISBN 978-1-939133-24-3. https://www.usenix.org/conference/usenixsecurity21/presentation/so. 17

[52] Gao, Jianbo: *Guided, automated testing of blockchain-based decentralized applications.* Em *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, páginas 138–140, 2019. 17

[53] Miller, Barton P., Louis Fredriksen e Bryan So: *An empirical study of the reliability of unix utilities.* Commun. ACM, 33(12):32–44, dezembro 1990, ISSN 0001-0782. https://doi.org/10.1145/96267.96279. 20

[54] Burkhardt, Walter H: *Generating test programs from syntax.* Computing, 2:53–73, 1967. 20

[55] *American fuzzy lop - github.* https://github.com/google/AFL, 2018. [Online; accessed 3-October-2021]. 20, 21, 22, 45

[56] Böhme, Marcel, Van Thuan Pham e Abhik Roychoudhury: *Coverage-based greybox fuzzing as markov chain.* Em *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, página 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450341394. https://doi.org/10.1145/2976749.2978428. 21, 22, 45

[57] Böhme, Marcel, Van Thuan Pham, Manh Dung Nguyen e Abhik Roychoudhury: *Directed greybox fuzzing.* CCS '17, página 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450349468. https://doi.org/10.1145/3133956.3134020. 21, 23, 25, 45

[58] Brent, Lexi, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz e Bernhard Scholz: *Vandal: A scalable security analysis framework for smart contracts.* arXiv preprint arXiv:1809.03981, 2018. 28, 36, 71

[59] *Docker compose.* https://docs.docker.com/compose/, 2023. [Online; accessed 25-April-2023]. 28