YUDI KUBOTA DUARTE DE OLIVEIRA

ANÁLISE DE VULNERABILIDADES EM CONTRATOS INTELIGENTES DA ETHEREUM

(*versão pré-defesa, compilada em 31 de março de 2023*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Luiz Carlos Albini.

CURITIBA PR

2023

RESUMO

Contratos inteligentes são programas que rodam em uma camada de abstração sobre a blockchain Ethereum. Esses contratos habilitam o desenvolvimento de aplicações variadas, incluindo aplicações financeiras. No entanto, vulnerabilidades presentes nesses contratos permitem a exploração de falhas, oferecendo riscos de significativos prejuízos financeiros. Portanto, é necessário o esforço para a prevenção de vulnerabilidades e para isso o uso de ferramentas de análise de contratos é essencial. Porém, as revisões de literatura do campo recomendam atualizações regulares. Com isso, a presente monografia propõe um estudo empírico baseado no trabalho de Durieux et al. (2020). O estudo consiste na construção de um *dataset* atualizado e na execução das ferramentas de análise sobre essa extensiva coleção de contratos. Os resultados deste trabalho reportam sobre os tempos de execução e as vulnerabilidades encontradas pelas ferramentas de análise selecionadas de acordo com a taxonomia DASP. Ao final, conclui-se que houve redução na porcentagem de contratos acusados como vulneráveis em comparação ao estudo anterior.

Palavras-chave: Blockchain. Vulnerabilidades. Ethereum.

ABSTRACT

Smart contracts are programs that run on a layer of abstraction of the Ethereum blockchain. These contracts enable the development of several applications, including financial applications. However, vulnerabilities in these contracts allow for the exploitation of flaws, offering significant risks of financial losses. Therefore, an effort is required to prevent vulnerabilities, and contract analysis tools are essential for this purpose. However, literature reviews in the field recommend regular updates. Thus, this work proposes an empirical study based on the work of Durieux et al. (2020). The study involves building an updated dataset and running analysis tools on an extensive collection of contracts. The results of this work report on the execution times and vulnerabilities found by selected analysis tools according to the DASP taxonomy. In the end, it is concluded that there was a reduction in the percentage of contracts tagged as vulnerable compared to the previous study.

Keywords: Blockchain. Vulnerabilities. Ethereum.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

UFPR	Universidade Federal do Paraná
C3SL	Centro de Computação Científica
API	*Application Programming Interface* (Interface de programação de aplicação)
CFG	*Control Flow Graph* (Grafo de Fluxo de Controle)
DAG	*Directed Acyclic Graph* (Grafo acíclico direcionado)
CLI	*Command Line Interface* (Interface para linha de comando)
P2P	*Peer to Peer* (Par-a-par)
TOD	*Transaction Order Dependence* (Dependência da Ordem de Transações)
DASP	*Decentralized Application Security Project* (Projeto de Segurança das Aplicações Descentralizadas)
SWC	*Smart Contract Weakness Classification* (Classificação das Fraquezas de Contratos Inteligentes)
EIP	*Ethereum Improvement Proposal* (Proposta de Melhoria da Ethereum)
ERC	*Ethereum Request for Comment* (Requisição para comentários da Ethereum)
NFT	*Non-Fungible Token* (Token não fungível)
JSON	*Javascript Object Notation* (Notação de objeto javascript)
RAM	*Random Access Memory* (Memória de acesso aleatório)
CEI	*Checks, Effects, and Interactions* (Verificações, efeitos e interações)
DAO	*Descentralized Autonomous Organization* (Organização autônoma descentralizada)
EVM	*Ethereum Virtual Machine* (Máquina Virtual da Ethereum)

JVM	*Java Virtual Machine* (Máquina Virtual do Java)

LISTA DE SÍMBOLOS

σ	sigma, representa o estado do autômato da Ethereum
Υ	Upsilon, representa a função de transição do autômato da Ethereum

SUMÁRIO

1 INTRODUÇÃO

A necessidade de se realizar trocas de bens e serviços na sociedade gerou a demanda pela moeda (Asmundson e Oner, 2012). A moeda serve como um meio de troca comum que aumenta a eficiência do comércio, aumentando também a produtividade de uma sociedade (Mankiw, 2021). Naturalmente, tentativas da criação de uma moeda digital foram propostas. Contudo, essa proposta enfrenta desafios particulares ao depender de sistemas distribuídos. Foi apenas em 2009 que Satoshi Nakamoto (Nakamoto, 2023) resolveu o problema do gasto duplo a partir do sistema Bitcoin. Esse sistema propunha a criação de um mecanismo que foi posteriormente chamado de blockchain. Esse mecanismo possui características que possibilitam a criação de um sistema de pagamentos descentralizado, imutável e de livre entrada – efetivamente criando um novo tipo de moeda: a criptomoeda. O advento da tecnologia blockchain impulsionou outras iniciativas, entre elas a Ethereum.

Diferentemente do Bitcoin, a Ethereum permite a execução de programas arbitrários em sua plataforma, chamados de contratos inteligentes. Esses contratos habilitam o desenvolvimento de aplicações descentralizadas em uma plataforma de computação distribuída e segura. Dessa flexibilidade emergem inúmeras aplicações, desde a criação de novas moedas digitais a títulos de propriedade e acordos auto-executáveis (Buterin, 2014).

Consequentemente, ataques a esses sistemas são esperados. Vulnerabilidades presentes em contratos inteligentes permitem a exploração dessas falhas, causando consideráveis prejuízos financeiros. Portanto, se faz necessária a prevenção dessas falhas. Apesar dessa prevenção ser de responsabilidade dos desenvolvedores de contratos, as ferramentas automatizadas de análise também são essenciais nesse esforço.

Porém, revisões sistemáticas sugerem que o campo ainda seja imaturo. Em decorrência disso, Rameder et al. (2022) recomenda a revisão periódica dos resultados obtidos por estudos nessa área. Um trabalho relevante foi o estudo empírico proposto por Durieux et al. (2020). Nesse trabalho, os autores propõem a coleção de contratos inteligentes publicados na blockchain Ethereum e a análise dessa coleção por diversas ferramentas de análise.

1.1 OBJETIVOS

Na data de consolidação do *dataset* construído por Durieux et al. (2020), em 08/08/2019 (Smartbugs, 2020), a Ethereum possuia 17.503.263 contratos publicados em sua blockchain. Já em 09/01/2023 esse número passou para 56.386.406, um aumento de quase 3 vezes [1]. Essa massiva quantidade de novos contratos pode incorrer em mudanças nas características de contratos em produção que foram utilizados para construção do *dataset*. Apesar desse potencial, nenhum

[1] Números obtidos através do Google BigQuery. Detalhes sobre a query no Capítulo 6.

estudo em grande escala foi realizado até 2022 (Rameder et al., 2022), o que pode indicar que uma atualização de resultados seja necessária. Além disso, novas ferramentas foram incluídas ou atualizadas e outras ferramentas foram depreciadas ou abandonadas (Rameder et al., 2022) (Kushwaha et al., 2022).

Desta forma, o presente trabalho propõe um estudo empírico baseado no estudo realizado por Durieux et al. (2020), seguindo a recomendação de Rameder et al. (2022) de realizar atualizações frequentes no campo de ferramentas de análise de contratos inteligentes. Esse estudo tem como objetivos: 1. propor um *dataset* atualizado e representativo das condições de contratos em produção até 2022, 2. reportar sobre o estado das ferramentas selecionadas no estudo e 3. comparar os resultados com o estudo de referência. Com isso, espera-se obter novos resultados que confirmem ou atualizem resultados anteriores sobre a área de ferramentas de análise de vulnerabilidades para contratos inteligentes da Ethereum.

1.2 ESTRUTURA DA MONOGRAFIA

A presente monografia foi organizada da seguinte maneira: O Capítulo 2 apresenta o contexto histórico e econômico para a criação da tecnologia blockchain. O Capítulo 3 explora o funcionamento técnico da blockchain Ethereum. O Capítulo 4 discorre sobre o estado da arte na área de ferramentas de análise para contratos inteligentes da Ethereum. O Capítulo 5 descreve o artigo de referência usado como base de metodologia e comparações. O Capítulo 6 descreve a metodologia adotada para a execução do estudo empírico proposto. O Capítulo 7 apresenta os resultados do estudo empírico, discussões e comparações. Por fim, o capítulo 8 apresenta as considerações finais e sugestões de trabalhos futuros.

2 ECONOMIA E BLOCKCHAIN

Este capítulo visa ambientar o leitor a respeito do contexto histórico, econômico e tecnológico que proporcionou a criação da tecnologia blockchain. Discorreremos sobre a criação e evolução da moeda e sobre as tecnologias modernas que culminaram com a criação do sistema Bitcoin.

2.1 ECONOMIA

No livro "Introdução à Economia", Mankiw (2021) define: "Economia é o estudo de como a sociedade administra os seus recursos escassos". Uma atividade consequente dessa administração é o comércio, isto é, a troca de bens e serviços entre os participantes da sociedade. A presença do comércio impulsiona a produtividade da sociedade como um todo, possibilitando a produção de mais bens e serviços do que sem essa prática.

As primeiras formas de comércio recorriam ao escambo para a realização de trocas mutualmente benéficas. O escambo consiste na prática de realizar a troca de bens ou serviços por outro que seja de interesse alheio de maneira imediata. Para que essa troca ocorra é necessário que ambas as partes desejem um bem ou serviço que o outro possui no momento da negociação. Essa circunstância improvável caracteriza o que é chamado de **dupla coincidência de desejos**. Uma sociedade que pratica o comércio através do escambo teria dificuldades em alocar os seus recursos de forma eficiente, pois o custo de negociação seria alto e improvável. Em decorrência disso, as sociedades optaram por utilizar a moeda (Mankiw, 2021).

2.1.1 Moeda e Confiança

Para Dotsis (2019), a moeda atual é uma forma de IOU, do inglês "I Owe You", que é um documento que age como uma forma de reconhecimento de débito. Esse ativo financeiro é uma maneira de lidar com as ineficiências do escambo. Ela serve como meio de troca comum entre as partes, o que permite que a sociedade organize seus recursos de maneira mais eficiente, aumentando a produtividade (Mankiw, 2021) (Asmundson e Oner, 2012).

A moeda tem 3 funções na economia (McLeay et al., 2014) (Dotsis, 2019) (Mankiw, 2021) (Asmundson e Oner, 2012) , são elas servir como:

1. **Meio de troca:** é um ativo financeiro amplamente usado como intermédio para trocas de bens ou serviços;

2. **Reserva de valor:** é um ativo que pode ser trocado por algo de valor no futuro e que mantém seu valor no longo prazo;

3. **Unidade de conta:** é utilizada como um padrão adotado amplamente para medir e registrar valor econômico de bens e serviços na sociedade;

A moeda também pode assumir diferentes formas para exercer suas funções. De acordo com a história econômica, a moeda surgiu primeiramente como uma mercadoria, de forma que um ativo de valor intrínseco real é utilizado como moeda. A esse tipo de moeda se dá o nome de **moeda-mercadoria**. Exemplos de moeda-mercadoria são metais preciosos (como ouro e prata), produtos agrícolas ou mesmo um quilograma de sal (Dotsis, 2019) (Mankiw, 2021). Um exemplo de moeda-mercadoria inconvencional é o cigarro. Em algumas prisões, cigarros são usados como moeda pois satisfazem as 3 características descritas acima. Prisioneiros confiam que esse bem: 1. poderá ser trocado por outros bens e serviços, 2. manterá seu valor através do tempo e 3. será utilizado para medir outros bens e favores (Dotsis, 2019). Isso sugere que em alguns lugares a moeda corrente não necessariamente é a moeda oficial adotada pelo Estado. De fato, Ingham (2004) citado por Dotsis (2019) defende que o sistema monetário moderno surgiu como resultado da integração da moeda informal com a moeda estatal.

Ademais, a moeda pode não assumir necessariamente a forma de mercadoria. O papel-moeda (*banknotes*) é um objeto que não tem valor intrínseco, mas age na prática como um compromisso de débito – um IOU – e por esse motivo exerce as funções da moeda. Se a nota bancária pode ser convertida em algum ativo de valor real, como ouro, ela é caracterizada como uma **moeda representativa**. Quando a moeda representa o direito a uma quantidade em ouro (lastro em ouro), diz-se que o sistema monetário é padrão ouro. Porém, desde a queda do lastro do dólar em ouro, em 1971 [1] , se tem o que é chamado de **moeda de curso forçado** (também chamada moeda fiduciária ou *fiat*). Esse tipo de moeda se caracteriza por ser um ativo financeiro que não tem lastro em um ativo real (Dotsis, 2019).

Em retrospecto, a moeda parece resolver o problema da troca eficiente, mas dela emergem outros problemas. Para que uma moeda funcione, é preciso ter o consenso de uma sociedade ao usá-la. Indivíduos precisam ter a segurança de que a moeda poderá ser utilizada no futuro, seja amplamente aceita e que a sua produção será suficiente e responsável. Em outras palavras, é necessária a **confiança** alheia em um sistema monetário. Com efeito, Asmundson e Oner (2012) afirmam que "o dinheiro funciona porque as pessoas acreditam que vai funcionar"(tradução própria [2]). Contudo, a confiança na moeda pode desvanecer. Altas taxas de inflação e tensões políticas podem erodir a confiança na moeda (Asmundson e Oner, 2012). Nakamoto (2023) destaca que um sistema monetário baseado em criptografia removeria a necessidade de confiança, pois se basearia em provas criptográficas, permitindo transações entre participantes que não se confiam entre si ou sequer confiam em uma entidade centralizadora.

[1] Antes dessa data também se tem registros de uso de moeda de curso forçado. Consultar Dotsis (2019)

[2] No original: *"money works because people believe that it will"*

2.2 A PRIMEIRA BLOCKCHAIN: BITCOIN

A ideia de se criar uma moeda digital descentralizada tem sido discutida há décadas. A ideia baseia-se na premissa de que assim como a moeda fiduciária, uma representação digital também pode ser considerada moeda se for aceita e cumprir as três funções monetárias citadas anteriormente, mesmo sem ter lastro ou valor intrínseco. Asmundson e Oner (2012) dão o exemplo de que, apesar de ter sido criado em 1994, o Real brasileiro foi aceito pelos cidadãos "assim como a moeda fiduciária, se é aceita como moeda, então é moeda"(tradução própria [3]).

Em 1992, May (1992) destacou que as tecnologias discutidas no meio acadêmico na época permitiam a criação de um novo sistema político e monetário. Nesse novo sistema, a interação entre participantes poderia ser totalmente anônima e incensurável ao se beneficiar de aspectos da criptografia, como protocolos de chave pública e provas de zero-conhecimento.

Em 1998, Dai (1998), inspirado pelo manifesto de May (1992), propôs o protocolo descentralizado b-money. Nesse sistema, cada participante da rede detém uma cópia de um banco de dados que serve como livro-razão (*ledger*) com os estados (saldos) de todos os demais participantes. Nesse modelo a posse da moeda é definida pelo consenso dos participantes sobre o estado desse livro-razão. O autor sugere que a criação de moedas seja feita a partir da solução de problemas computacionalmente caros, mas que podem ser facilmente verificados publicamente. Um algoritmo que satisfaz essas condições foi proposto por Back (2002) em 1997, com o objetivo inicial de mitigar o abuso de recursos computacionais na internet, como o spam em e-mails. O artigo baseia-se no fato de que a função mais eficiente para encontrar o resultado de uma função hash é a força bruta [4]. Porém, foi apenas em 2004 que Finney (2023) uniu os dois conceitos e sugeriu o uso desse algoritmo como Prova de Trabalho (*Proof of Work* ou PoW) para a criação de *tokens* que poderiam ser utilizados como moeda a partir da mediação de um servidor central. Porém essa abordagem ainda dependia da confiança em um ponto central que poderia falhar ou ser corrompido.

A centralização era necessária para evitar o **problema do gasto duplo** e o **problema dos generais bizantinos**, decorrentes da utilização de um sistema distribuído. Chohan (2021) define: "O problema do gasto duplo é uma falha em pontencial em uma criptomoeda ou outro esquema de dinheiro digital onde o mesmo *token* digital pode ser gasto mais de uma vez"(tradução própria [5]). Tal problema inviabilizaria o uso dos *tokens* de um sistema vulnerável a essa falha como moeda. O problema dos generais bizantinos refere-se ao impasse de caso hipotético em que o exército bizantino precisa entrar em consenso sobre um plano de ação, mas está comandado por *n* generais que só podem se comunicar por mensageiros não confiáveis e a presença de generais desleais. Esse caso hipotético serve como uma abstração muito útil para problemas que envolvem

[3] No original: *"Just like fiat money. If it is accepted as money, it is money"*

[4] Detalhes sobre o funcionamento das funções hash e dos algoritmos citados neste capítulo serão explorados no próximo capítulo

[5] No original: *"the double spending problem is a potential flaw in a cryptocurrency or other digital cash scheme whereby the same single digital token can be spent more than once"*

o consenso de um sistema distribuído sobre uma determinada informação (Lamport et al., 1982). Por exemplo, ao considerar o sistema b-money o estado do livro-razão em disputa é análogo ao impasse dos generais, pois há incentivo financeiro para que um participante da rede se comporte de maneira desleal ao reportar uma quantidade incorreta de moedas.

Foi apenas em 2009 que Nakamoto (2023) propôs um mecanismo que não dependia da confiança em uma entidade central e que resolvia ambos os problemas supracitados. O artigo resolve a dependência de uma entidade central a partir de uma rede P2P (par-a-par) que consegue obter consenso devido ao algoritmo de prova de trabalho para a criação de um histórico canônico de transações. Esse histórico é constituído pela sequência cronológica de blocos de transações interligados criptograficamente por hash de referências a blocos anteriores, formando uma corrente de blocos – posteriormente denominado **blockchain** (corrente de blocos, em inglês) [6] . A blockchain não pode ser alterada sem a re-computação do algoritmo de consenso. Para isso, a maior cadeia de blocos é considerada o estado canônico do sistema, que contém a maior quantidade de computação dedicada historicamente. Isso mitiga a possibilidade de que atacantes maliciosos modifiquem o histórico ou o estado do sistema, pois para isso precisariam computar toda a cadeia mais longa e controlar mais do que 50% do poder computacional da rede como um todo, o que é extremamente improvável. Ademais, o protocolo incentiva a participação honesta dos nodos através da recompensa em bitcoins (*tokens* da blockchain Bitcoin). Essa dinâmica também limita a quantidade de bitcoins produzidos, pois essa produção depende da capacidade computacional da rede como um todo.

As propriedades do Bitcoin fazem com que esse sistema seja considerado uma forma de moeda robusta, isto é, moeda que é difícil de ser produzida. De forma semelhante ao ouro – que depende de trabalho e recursos físicos para extração, produção e comercialização – a produção de um *token* na rede Bitcoin requer quantidades expressivas de recursos computacionais. Isso torna a oferta monetária de moedas robustas muito mais controlada do que as moedas de curso forçado, o que pode ajudar a mitigar problemas como elevadas taxas de inflação e a consequente perda da confiança na moeda. Além disso, o protocolo Bitcoin impõe um limite rígido de 21 milhões de *tokens*, o que caracteriza-o como uma moeda não inflacionária (Guazzo, 2021) (C., 2022). Mudanças no protocolo também são improváveis, pois o Bitcoin não tem uma figura centralizadora, dado que o seu criador Satoshi Nakamoto é na verdade um pseudônimo desconhecido.

2.3 UM FUTURO DESCENTRALIZADO

Neste capítulo, foram apresentadas as diversas formas que a moeda pode assumir e como ela se relaciona com a confiança. Também foram apresentadas as tentativas de criação de uma moeda digital descentralizada e seus desafios, como o Problema dos Generais Bizantinos e

[6] Mais detalhes sobre esse mecanismo no próximo capítulo

o Problema do Gasto Duplo. Por fim, foi mostrado como o Bitcoin resolve esses problemas e como esse sistema pode ser considerado moeda.

A proposta do Bitcoin culminou na invenção da tecnologia blockchain, que pode oferecer consenso, confiança e controle da oferta monetária. Contudo, essa tecnologia permite não apenas a implementação de um sistema monetário, mas também de aplicações arbitrárias, pois oferece um livro-razão distribuído e confiável. A Ethereum, lançada em 2015, é uma plataforma que se propõe a habilitar a construção de aplicações descentralizadas a partir de programas arbitrários que são executados pela blockchain, chamados "contratos inteligentes"(Ethereum, 2023c). A existência de contratos inteligentes possibilita a implementação de aplicações que não necessitam da confiança mutua entre seus participantes, contribuindo para um futuro mais robusto e descentralizado.

3 ETHEREUM

Em 2014, Buterin (2014) publicou o *whitepaper* "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform"propondo um novo sistema baseado na tecnologia blockchain que havia sido criada e popularizada pelo sistema Bitcoin. Porém, enquanto o Bitcoin se propunha a ser um sistema eletrônico de pagamentos P2P, esse novo sistema – denominado Ethereum – visava se tornar uma plataforma distribuída de computação para uso genérico. No artigo, o autor destaca que apesar do sistema Bitcoin ter sido usado e alterado de maneira a estender suas funcionalidades, seu potencial era limitado, pois o sistema não oferecia uma plataforma de computação turing-completa de forma nativa (Ethereum, 2023c).

Neste capítulo, veremos como a Ethereum atinge o seu objetivo de ser uma plataforma de computação genérica e distribuída.

3.1 DEFINIÇÃO

Em uma perspectiva formal, de acordo com a especificação técnica (*yellowpaper*) de Wood (2022), a blockchain Ethereum pode ser considerada uma máquina de estados baseada em transações. Desta forma, o estado corrente em conjunto com o histórico das transações da máquina correspondem às entradas de um livro-razão distribuído. Formalmente:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T) \tag{3.1}$$

em que Υ é a função de transição do estado e σ é o estado global. A abstração σ permite que dados arbitrários sejam armazenados entre uma transição e outra. As transações na Ethereum representam transições válidas entre um estado e outro e são agrupadas em blocos, de forma semelhante ao Bitcoin.

3.2 CONTAS E TRANSAÇÕES

Na Ethereum, todas interações com a rede são iniciadas por uma **conta**. As contas podem ser controladas por agentes externos ou por código via contratos inteligentes. Quando são controladas por agentes externos são chamadas de *Externally-owned Accounts* (EOA). As EOAs são criadas a partir da geração de um par de chaves criptográficas. A chave pública é utilizada para derivar o endereço da conta e a chave privada é utilizada para assinar transações iniciadas pela mesma. Ambos os tipos de conta podem interagir com contratos inteligentes e enviar, receber e deter saldo e outros *tokens*. Uma conta de contrato inteligente só pode ser criada a partir da publicação de um contrato inteligente por uma EOA. Essas não possuem chaves criptográficas, pois elas não iniciam transações, apenas as recebem (Ethereum, 2023a).

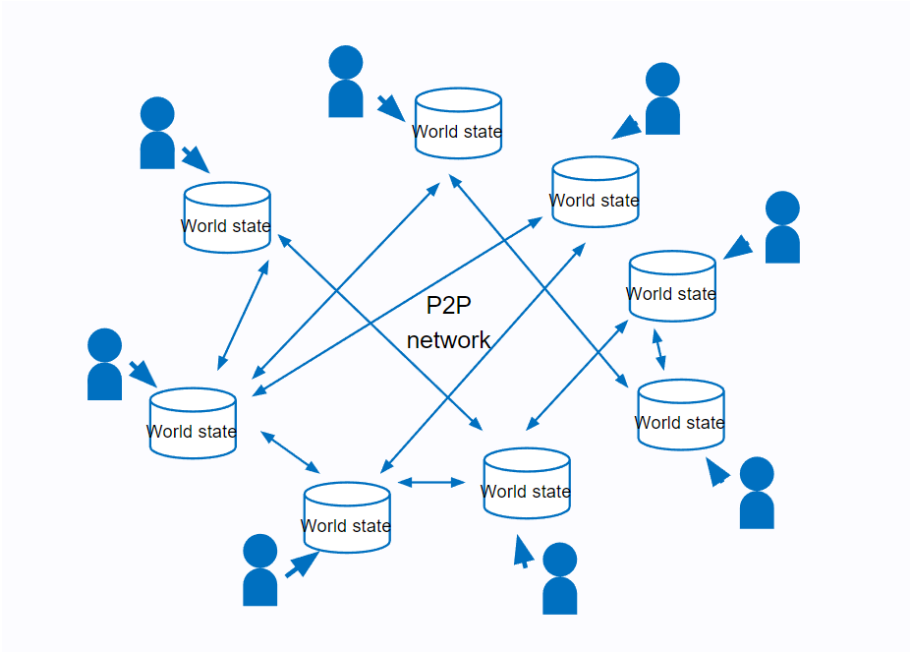

Figura 3.1: Blockchain é um banco de dados transacional, descentralizado e compartilhado a partir de uma rede P2P (Tani, 2018).

As contas possuem as seguintes propriedades (Ethereum, 2023a) (Wood, 2022):

- ***nonce:*** Um contador que indica o número de transações enviadas de uma EOA ou o número de contratos criados a partir de uma conta de contrato. Essa propriedade evita a duplicação de transações e ataques relacionados.
- ***balance:*** O saldo de ETH detido por essa conta, medido em wei (10^{-18} ETH).
- ***codeHash:*** Contém um hash do código do contrato inteligente atrelado à conta.
- ***storageHash:*** Contém um hash do nodo raíz de uma árvore Merkle Patricia que contém os dados de armazenamento arbitrário de uma conta, utilizado para valores de variáveis em contratos inteligentes.

As alterações ao estado global σ são chamadas de transações. Por meio de transações, EOAs conseguem requisitar alterações do estado global à rede. As transações representam arcos válidos entre um estado e outro. Para que uma transação seja considerada válida ela precisa ser assinada criptograficamente com a chave privada da conta que a requisitou e ser enviada a um nodo honesto da rede através da *mempool*. A *mempool* contém transações pendentes de confirmação pela rede. Além disso, a transação deve respeitar regras impostas pelo protocolo da rede. Por exemplo, se Alice envia uma transação assinada por sua chave privada que transfere 1 ETH para Bob, a transação só será considerada válida se a propriedade *balance* da EOA representada pela chave pública de Alice for maior ou igual a 1 ETH. Aliás, regras arbitrárias podem ser programadas e verificadas através de contratos inteligentes (Ethereum, 2023a) (Wood, 2022).

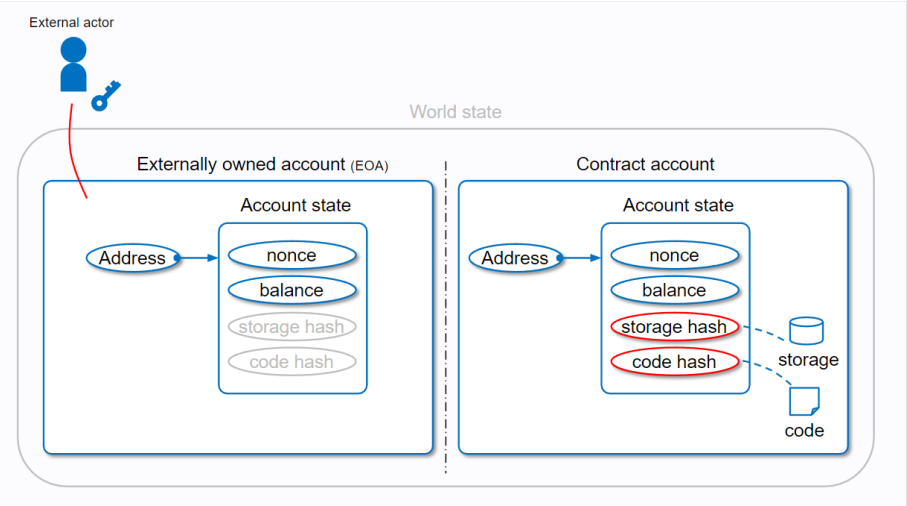

Figura 3.2: Tipos de conta e suas propriedades (Tani, 2018).

3.3 A CONSTRUÇÃO DA CADEIA DE BLOCOS

As transações são agrupadas em grupos, chamados de blocos. Um bloco é composto por um cabeçalho e pelas transações atreladas a ele. No cabeçalho do bloco existem informações como número do bloco, data e hora e uma referência ao bloco anterior (Ethereum, 2023a).

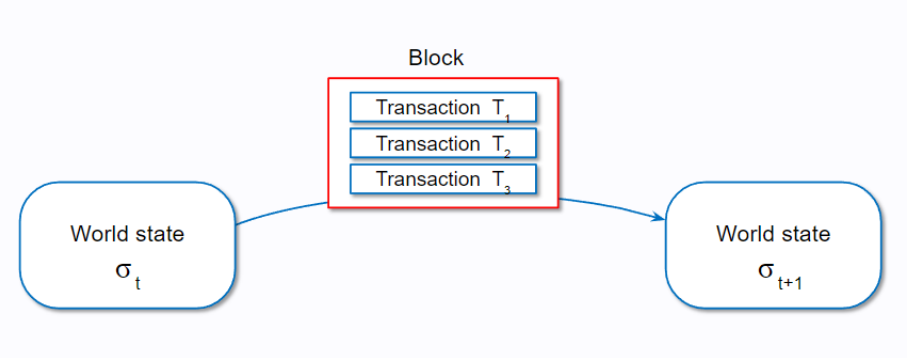

Figura 3.3: Blocos reúnem transações que alteram o estado global (Tani, 2018).

Os blocos contêm uma referência ao bloco válido anterior, criando uma corrente de blocos válidos, denominada **blockchain** (corrente de blocos). Essa cadeia de blocos é interligada através de funções criptográficas seguras. No caso da Ethereum essa função é a *keccak256*. Isso fortalece a conexão e cronologia entre os blocos, evitando que a cadeia seja alterada, o que empodera o sistema com a imutabilidade.

Para que um bloco seja criado e aceito na blockchain é preciso que ele seja validado pelos nodos da rede através de um algoritmo de consenso. A **mineração** é o processo de dedicar trabalho, na forma de recursos computacionais ou recursos financeiros, para a criação de um bloco de transações. No caso do algoritmo de consenso PoW, a criação de um bloco é feita a partir da resolução de um problema criptográfico que tem como objetivo comprovar o gasto de recurso computacional. Ao resolver esse problema computacional, o nodo responsável pela

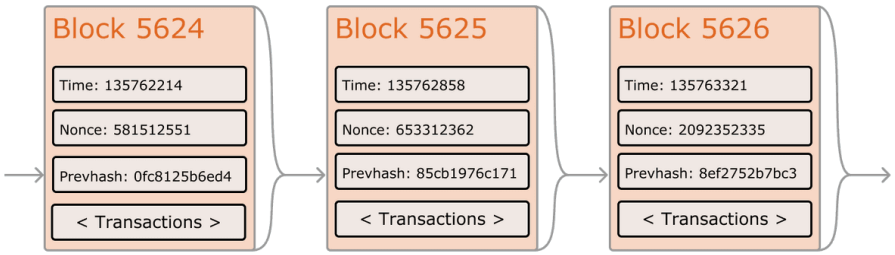

Figura 3.4: Corrente de blocos na Ethereum (Ethereum, 2023c).

solução é recompensado com *tokens* da rede. Esse mecanismo incentiva a participação de nodos honestos e constitui a única maneira de se produzir novos *tokens* ETH.

3.3.1 Algoritmos de consenso

Obter o consenso dos nodos da rede sobre o estado global σ é essencial, pois os participantes da rede esperam um estado coerente mesmo tendo as suas informações dispersas em um sistema distribuído. Esse é um problema análogo ao Problema dos Generais Bizantinos, sugerido por Lamport et al. (1982) em 1982. Para entender esse problema, suponha que o exército bizantino esteja acampado no exterior de uma cidade inimiga e precise executar um plano de ação, como invadir ou recuar. O exército é composto por várias divisões lideradas por comandantes dispersos e eles só podem se comunicar por mensageiros. Os comandantes podem ser leais ou desleais, assim como os mensageiros. A fim de executar um plano de ação eficaz, os comandantes leais precisam de um algoritmo que garanta que eles tomarão a decisão correta. Apesar de ser uma situação hipotética e útil para a compreensão do problema, termos como "decisão correta"não são claros o suficiente. Desta forma, Lamport et al. (1982) define esse problema formalmente da seguinte maneira:

Uma comandante-general precisa enviar uma ordem a seus $n - 1$ tenente-generais de forma que:

- **IC1.** Todos os tenente-generais leais obedeçam à mesma ordem.
- **IC2.** Se o comandante-general é leal, então todos os outros tenente-generais leais obedecem a ordem que ele envia.

Os algoritmos que resolvem esse problema são conhecidos como algoritmos de consenso. No contexto de tecnologia blockchain, esses algoritmos são utilizados de maneira garantir que o estado global do livro-razão seja aceito pelos nodos honestos. A blockchain Ethereum iniciou sua operação fazendo uso do algoritmo Proof of Work e passou a utilizar o algoritmo Proof of Stake a partir de uma atualização em 2022 (Ethereum, 2023a).

3.3.1.1 Proof of Work

No algoritmo Proof of Work (PoW) o consenso é obtido a partir da prova de que um nodo gastou uma quantidade específica de recursos computacionais para resolver um problema criptográfico. A solução para esse problema consiste em encontrar o valor *nonce* do cabeçalho do bloco tal que a *hash* do bloco seja menor do que um parâmetro de dificuldade. Esse parâmetro é ajustado bloco a bloco de forma a se adaptar ao poder computacional da rede e gerar blocos em intervalos de tempo regulares. Quando blocos estão sendo produzidos em tempos inferiores ao desejado a dificuldade aumenta. No Bitcoin esse tempo é de 10 minutos e na Ethereum o tempo é de 12 segundos (Wood, 2022).

Segundo a especificação formal de Wood (2022), criar um novo bloco é encontrar os valores m e n tal que:

$$m = H_m \quad \wedge \quad n \leq \frac{2^{256}}{H_d} \quad com \quad (m, n) = PoW(H_\alpha, H_n, d) \tag{3.2}$$

onde H_α é o cabeçalho do bloco a ser validado sem o *nonce*; H_n é o valor do *nonce*; H_d é a dificuldade do novo bloco, H_m é um hash utilizado em combinação com o *nonce*, d é um DAG (Grafo Acíclico Direcionado) usado na geração de *nonce* e PoW é a função de prova de trabalho. Na Ethereum, essa função é chamada de *Ethash*.

O nodo que encontra os valores m e n é o nodo que efetivamente submete a solução à rede. Os demais nodos podem confirmar a correta solução a partir da computação da função de prova de trabalho. Ao final, o nodo que resolveu o problema é recompensado com *tokens* e o bloco recém criado é considerado canônico por todos os nodos.

3.3.1.2 Proof of Stake

No algoritmo de consenso Proof of Stake (PoS), os nodos são incentivados a agirem honestamente mediante o risco de perdas financeiras. Para participar como um nodo na rede, um usuário deve primeiro depositar 32 ETH em um contrato inteligente. Esse saldo fica em risco de ser confiscado caso o nodo aja desonestamente ou preguiçosamente. Esse mecanismo permite que o tempo de cada bloco seja estático, não sendo necessário o parâmetro de dificuldade, como no PoW. Além disso, essa abordagem gasta menos energia elétrica, pois não exige significativos recursos computacionais para rodar, apenas financeiros. Apesar disso, esse mecanismo é mais complexo de ser implementado e mais jovem que o PoW (Ethereum, 2023a).

3.4 CONTRATOS INTELIGENTES

Uma das principais diferenças entre as blockchains Bitcoin e Ethereum é que a Ethereum é turing-completa e permite a execução de programas arbitrários, chamados de contratos inteligentes (Ethereum, 2023c). Esses contratos podem ser escritos em linguagens de programação de alto nível, como Solidity e Vyper, definindo regras para as transações na

blockchain. Tais programas podem ser utilizados para impor a execução honesta e distribuída de acordos imutáveis, abrindo possibilidades para inúmeras aplicações. Desta forma, a plataforma Ethereum habilita desenvolvedores a construírem aplicações descentralizadas que utilizam a tecnologia blockchain como plataforma de execução, autenticação, segurança e consenso (Ethereum, 2023a).

Os contratos inteligentes da Ethereum são o meio pelo qual os usuários da rede podem definir estados através de variáveis e regras para alterações desses estados através de funções. Segue abaixo um exemplo de contrato inteligente escrito na linguagem Solidity. Esse contrato foi adaptado de um repositório que demonstra um contrato vulnerável ao ataque de reentrada [1]. Ele implementa as funcionalidades de depósito e resgate de saldo em ETH.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;

import '@openzeppelin/contracts/utils/Strings.sol';

contract Vulneravel {
   mapping(address => uint256) public saldo;
   using Strings for uint256;
   using Strings for address;

   constructor() {
      //
   }

   function depositar() payable public {
      saldo[msg.sender] += msg.value;
   }

   function resgatar() public {
      // Vamos assumir que o resgatante éo chamador desta função
      address payable resgatante = payable(msg.sender);

      // Verifica se o resgatante tem saldo
      require(saldo[resgatante] > 0, "Nao ha saldo a ser resgatado.");

      // Realiza a transferência de todo o saldo disponível
      uint256 quantia = saldo[resgatante];
      (bool os, ) = resgatante.call{value: quantia}('');
      require(os, "Falha ao transferir fundos.");

      // Atualiza o saldo
      saldo[resgatante] = 0;
   }
```

[1] Mais detalhes sobre esse ataque no Capítulo 4. Código adaptado do repositório `https://github.com/yudikubota/exemplo_ataque_reentrada`

```
34
35      function saldoTotal() public view returns (uint256) {
36          return address(this).balance;
37      }
38
39      function saldoDe(address _address) public view returns (uint256) {
40          return saldo[_address];
41      }
42  }
```

Esse contrato inteligente define o estado *saldo* que mapeia endereços de contas a um inteiro que representa o saldo armazenado no contrato referente a cada conta. As funções *depositar()* e *resgatar()* definem a lógica para depósito e resgate do saldo pelas contas, ou seja, define as regras para uma alteração de estado válida – uma transação.

Para iniciar uma interação com esse contrato, uma EOA deve assinar uma transação com sua chave privada. Por exemplo, se uma EOA deseja chamar a função *depositar()* do contrato acima no valor de 1 ETH, ela deve enviar a seguinte transação a um nodo da rede:

```
1  {
2    "from": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
3    "to": "0xd9145CCE52D386f254917e481eB44e9943F39138",
4    "data": "0x3d922f90",
5    "value": 1000000000000000000
6  }
```

Nessa transação assumimos que o contrato foi publicado no endereço *0xd9145CCE52D386f254917e481eB44e9943F39138* e que o endereço da EOA é *0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2* . O campo *data* contém o nome da chamada de função e seus argumentos codificados e o campo *value* contém o valor em wei (10^{-18} ETH) a ser enviado para a função.

A funcionalidade de contratos inteligentes arbitrários pôde ser alcançada devido a implementação de uma máquina virtual que executa instruções básicas em cima da camada de blockchain. Essa máquina virtual é chamada de *Ethereum Virtual Machine* (EVM) e funciona de maneira similar a *Java Virtual Machine* (JVM) para a linguagem Java. Assim como na linguagem Java, o código de alto nível precisa ser compilado para *bytecode* para ser interpretado pela máquina virtual. O código EVM *bytecode* é então armazenado na blockchain e executado pela EVM quando requisitado (Ethereum, 2023c) (Wood, 2022).

Existem também especificações de interfaces frequentemente utilizadas no desenvolvimento de aplicações descentralizadas. Essas especificações são propostas a partir de *Ethereum Improvement Proposal* (EIPs) (Ethereum, 2023b). Dois padrões serão citados nesta monografia, são eles: ERC-20 e ERC-721, referentes às interfaces de *tokens* fungíveis e de *tokens* não fungíveis respectivamente. Diferentemente do *token* nativo da rede, como o ETH no caso da Ethereum, os *tokens* ERC são implementados por contratos inteligentes e não pelo protocolo.

Esses *tokens* podem ser criados para representar a posse de ativos financeiros reais ou totalmente digitais.

3.5 RESUMO

Este capítulo mostrou como a blockchain Ethereum pode ser utilizada como uma plataforma de computação generalizada. Em resumo, a Ethereum pode ser considerada uma máquina de estados baseada em transações. Alterações ao estado global dessa máquina são feitas através de transações que são agrupadas em blocos. Esses blocos são validados em uma rede P2P a partir de um algoritmo de consenso, que garante que o estado global da rede seja consistente. Além disso, o algoritmo de consenso também atua na criação de novos *tokens*. Uma diferença notável entre o Bitcoin e o Ethereum é que o Ethereum se propõe a ser uma plataforma de computação generalizada, enquanto o Bitcoin tem um escopo mais reduzido. A Ethereum oferece essa funcionalidade propondo a *Ethereum Virtual Machine* (EVM) – uma máquina virtual que executa instruções arbitrárias e roda sobre a camada da blockchain, de forma semelhante a JVM para a linguagem Java. Essa camada adicional de abstração permite o uso de linguagens de programação de alto nível, como Solidity e Vyper, para a criação de programas na blockchain, chamados contratos inteligentes. A partir disso, a plataforma habilita desenvolvedores a criarem aplicações descentralizadas de uso genérico com aplicações variando de criação de novas moedas, *tokens* e organizações descentralizadas (DAOs) a aplicações financeiras (DeFi), *exchanges* e títulos de propriedade (Ethereum, 2023c).

4 SEGURANÇA NA ETHEREUM

Em 2016, cerca de 3,6 milhões de ETH foram transferidos da organização descentralizada conhecida como “The DAO” para um atacante anônimo através de uma transação maliciosa em um *smart contract* vulnerável. A quantidade era equivalente a 60 milhões de USD na cotação da época. Esse foi o primeiro grande *exploit* enfrentado pela rede Ethereum, na época com apenas um ano de existência. O ataque gerou incertezas sobre a resposta da comunidade e o futuro da blockchain. Como cerca de 14% de todo ETH circulante na época estava envolvido no hack, a comunidade decidiu por fazer um *hard fork* (divisão) da rede, efetivamente anulando a transação do atacante. Isso gerou enormes controvérsias pois teoricamente blockchains seriam imutáveis. Os mineradores que não concordavam com essa alteração continuaram executando a versão original da blockchain, enquanto outros atualizaram seus clientes para legitimar a operação corretiva. Isso efetivamente causou a divisão das cadeias de blocos e dividiu a blockchain em duas: a Ethereum Classic (ETC) e a Ethereum (ETH) (Atzei et al., 2017) (jhimlic1, 2022).

Esse episódio marca a Ethereum como um dos eventos mais relevantes em sua história. A partir desse momento a segurança em blockchain foi reconhecida como uma importante área dessa tecnologia, pois falhas nesse pilar oferecem riscos à sua própria existência, credibilidade e adoção, além de causar grandes prejuízos financeiros.

Este capítulo explora aspectos de segurança na rede Ethereum, compreendendo revisões de literatura, vulnerabilidades, taxonomias, ferramentas e *benchmarks*.

4.1 ALGUMAS VULNERABILIDADES

Segundo Jimenez Freitez et al. (2009), "uma vulnerabilidade de software é uma falha ou defeito na construção de um software que pode ser explorada por um atacante para obter algum privilégio no sistema"(tradução própria [1]) .

As vulnerabilidades presentes no ecossistema Ethereum podem estar presentes em diversos níveis. A controvérsia sobre desfazer uma transação (como no caso The DAO) pode ser argumentada como uma vulnerabilidade presente no nível de comunidade, pois depende da discussão empreendida pelos membros da comunidade. Outras vulnerabilidades podem estar presentes no nível de tecnologia blockchain, que compreende as implementações de clientes da rede, a conexão entre os seus nodos e outras falhas sistêmicas reservadas às particularidades dessa tecnologia (Group, 2023) (Atzei et al., 2017) .

Neste trabalho, iremos focar exclusivamente em vulnerabilidades presentes na blockchain Ethereum no nível de código-fonte Solidity. Isto é, não consideraremos vulnerabilidades presentes na comunidade da Ethereum ou em suas diferentes implementações.

[1] No original: *"A software vulnerability is a flaw or defect in the software construction that can be exploited by an attacker in order to obtain some privileges in the system."*

4.1.1 Ataque de Reentrada

O ataque ao contrato da The DAO foi possível devido a uma vulnerabilidade que permitia o desvio do fluxo de execução do contrato inteligente. Essa vulnerabilidade ficou conhecida como Ataque de Reentrada e é uma das mais proeminentes da Ethereum [2].

O ataque funciona da seguinte maneira: durante a execução de uma função do contrato vulnerável, o fluxo é desvirtuado para um contrato atacante de maneira a subverter a lógica de verificação de estado do contrato atacado. Com a lógica comprometida, o contrato atacado permite a execução de ações que não deveriam ser permitidas. Essa falha permite, por exemplo, que o saldo de um contrato seja drenado por completo, sem as verificações de propriedade que seriam normalmente aplicadas (Daian, 2016). [3]

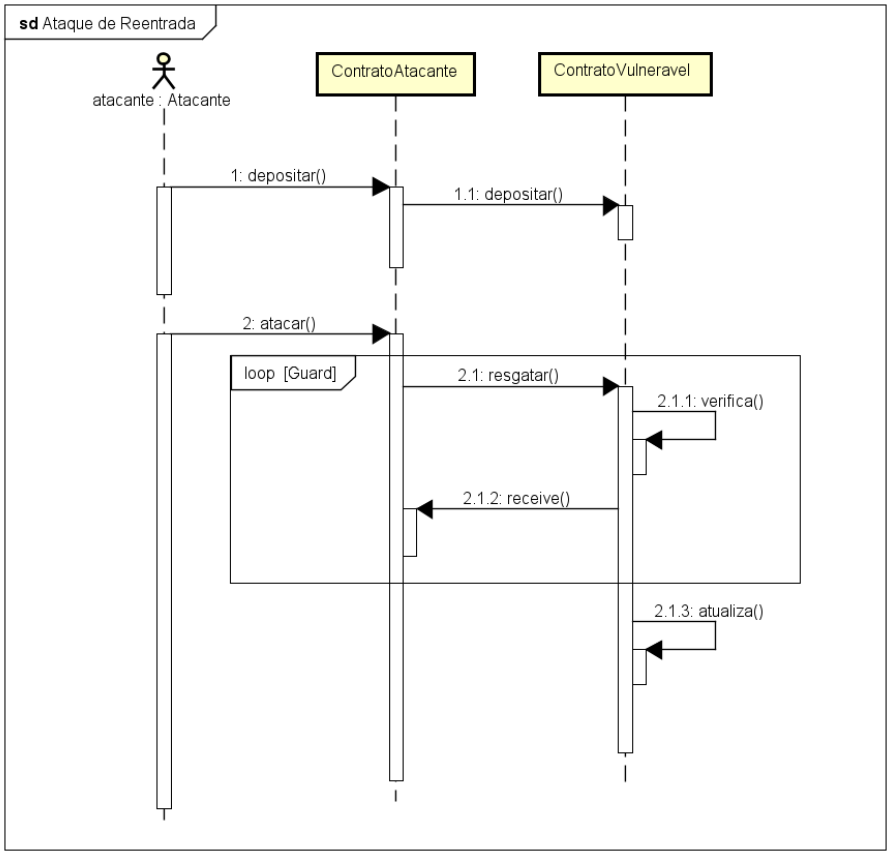

Figura 4.1: Diagrama de Sequência do Ataque de Reentrada.

A Figura 4.1 mostra um diagrama de sequência que exemplifica o ataque com os seguintes passos:

- ***1: depositar()*** – Primeiro, o ator atacante deposita uma quantidade arbitrária de fundos no contrato atacante;

[2]Essa vulnerabilidade é identificada como 1A na taxonomia de Rameder et al. (2022), correspondente a SWC-107, na taxonomia SWC e DASP-1 na taxonomia DASP.

[3]Um exemplo de ataque de reentrada pode ser encontrado no repositório `https://github.com/yudikubota/exemplo_ataque_reentrada`

- ***1.1: depositar()*** – O contrato atacante transfere seus fundos para o contrato vulnerável. Desta forma o saldo do atacante é maior que zero nos registros do contrato vulnerável. Esta etapa é necessária para o sucesso na verificação de saldo do passo *2.1.1: verifica()*;
- ***2: atacar()*** – Em seguida, em outra transação, o atacante inicia o ataque;
- ***2.1: resgatar()*** – A função desencadeia uma chamada no contrato vulnerável para o resgate dos fundos;
- ***2.1.1: verifica()*** – O contrato vulnerável verifica o saldo do contrato atacante. Essa verificação sempre resulta em sucesso, pois o saldo ainda não foi atualizado no fluxo de execução atual;
- ***2.1.2: receive()*** – Logo após receber os fundos, a função *receive()* é chamada no contrato atacante. Esta função desencadeia o resgate novamente completando um ciclo de reentrada. Esse ciclo é repetido até que os fundos sejam totalmente drenados do contrato vulnerável;
- ***2.1.3: atualiza()*** – Após a saída do ciclo, o contrato vulnerável atualiza o saldo do contrato atacante.

A vulnerabilidade está presente no fato de que a transferência de fundos é feita antes da atualização do estado da aplicação. Uma maneira de prevenir essa vulnerabilidade é a simples re-ordenação dos comandos para garantir que primeiro seja alterado o estado da blockchain e apenas depois seja feita a transferência de fundos. Essa ordenação se tornou uma boa prática e padrão de projeto difundida com o nome CEI *(Checks, Effects, and Interactions)* (Hacken, 2022) (Consensys, 2023a).

4.1.2 Uso de tx.origin

Um outro exemplo de vulnerabilidade é o uso da variável `tx.origin`. Essa variável contém o endereço da conta externa (EOA) que originou a transação corrente. Utilizar essa variável para autorizar outras ações oferece riscos, pois ela permanece constante mesmo quando a transação passa por outro contrato inteligente malicioso que pode interceptar o fluxo de execução e alterar valores de parâmetros. Isso somado a técnicas de engenharia social (*phishing*) faz com que usuários autorizem transações fraudulentas em seu endereço (Group, 2023). [4]

A recomendação é utilizar a variável `msg.sender`, que contém o endereço do chamador da função (que pode ser um outro contrato inteligente). Desta forma, é possível assegurar a origem dos parâmetros da função chamada e evitar ações não intencionadas.

[4]Essa vulnerabilidade é identificada como 6A na taxonomia de Rameder et al. (2022), correspondente a SWC-115, na taxonomia SWC e DASP-2 na taxonomia DASP.

4.1.3 Uso de geradores de números pseudo-aleatórios previsíveis

Sendo um sistema distribuído e assíncrono, a tecnologia blockchain não possui um relógio central [5]. Portanto, variáveis que indicam tempo, como `block.timestamp`, precisam ser definidas por entidades eleitas. No caso da Ethereum, esse valor é definido pelo minerador/validador do bloco corrente. Essa característica implica em variáveis com valores determinísticos, ou seja, previsíveis por determinados atores, o que gera assimetria de informação sobre o estado atual e esperado do sistema. Isso pode se tornar um problema quando uma aplicação requer o uso de valores aleatórios ou pseudo-aleatórios, pois um atacante pode tentar descobrir ou manipular esses valores em benefício próprio (Olickel, 2016) [6] .

O incentivo à manipulação de valores se torna ainda maior ao considerar a presença de aplicações financeiras no sistema, especialmente loterias – que fazem uso de geradores de números pseudo-aleatórios (PRNGs) para o sorteio do ganhador de um prêmio. Em 2017, a loteria SmartBillions foi alvo desse ataque culminando na perda de mais de 400 ETH (Group, 2023) (Reutov, 2018) .

Em 2018, Reutov (2018) realizou a análise de 3.648 contratos públicos da Ethereum que faziam uso de PRNGs. A análise resultou na identificação de 43 contratos considerados vulneráveis. O autor considerou 4 categorias de PRNGs.

- **que usam variáveis do bloco como fonte de entropia**: Essas variáveis são definidas pelo minerador, ou seja, não podem ser usadas como fonte de entropia, pois podem ser manipuladas pelo minerador.

- **que usam a função hash de algum bloco anterior**: Como esses valores são compartilhados no bloco como um todo, um atacante pode usar o mesmo código para descobrir o número gerado no contrato vulnerável e usar uma transação interna para executar o exploit.

- **que usam uma fonte de entropia considerada "privada"**: Todos os valores da blockchain são públicos. Mesmo que uma variável seja marcada como `private`, a transação que altera o seu valor permanece pública e acessível.

- **que são sucetíveis a front-running**: Um atacante pode monitorar a *mempool* procurando por transações que satisfaçam o sorteio. Uma vez encontrada, o atacante pode forçar a execução da sua transação maliciosa antes da transação original através da manipulação do preço do gas (*front-running*).

Essa vulnerabilidade pode ser mitigada através do uso de oráculos, técnicas avançadas de criptografia ou o método de sorteios commit-reveal. (Reutov, 2018)

[5] Ver Problema dos Generais Bizantinos

[6] Essa vulnerabilidade é identificada como 2C na taxonomia de Rameder et al. (2022), correspondente a SWC-120, na taxonomia SWC e DASP-6 na taxonomia DASP.

Através de oráculos, a fonte de entropia é gerada fora do sistema distribuído da blockchain. Apesar de funcional, essa abordagem apenas move a assimetria de informação do minerador para o gerador de entropia que agora passa a ter incentivos para a manipulação. Contudo, iniciativas de oráculos como a Chainlink (2023) visam descentralizar a geração de números aleatórios para garantir a verificabilidade da não manipulação de valores.

4.2 TAXONOMIA DE VULNERABILIDADES

Ao estudar sobre a segurança de um sistema é interessante classificar suas vulnerabilidades de acordo com alguns critérios. Essa classificação pode oferecer padrões, como nomenclatura, descrição, escopo e tipo, para auxiliar o entendimento e discussão dos problemas encontrados. Tal padronização também é benéfica para estudos e ferramentas de análise pois oferece uma base comum para a comparação de resultados de maneira objetiva.

Ferramentas de análise que não têm uma classificação própria geralmente optam por usar uma das seguintes taxonomias comunitárias:

- ***Decentralized Application Security Project***: Mais conhecido como DASP, é um projeto open-source iniciado pela empresa NCC Group que classifica as vulnerabilidades em 10 grupos. O projeto conta com descrição desses grupos, exemplos, eventos relevantes e mitigações (Group, 2023).
- ***Smart Contract Weakness Classification Registry***: A *SWC Registry* relaciona vulnerabilidades com a tipologia CWE (Mitre, 2023) e conta com 37 itens contendo mitigações, referências e exemplos (SWC, 2023).

Apesar de existirem iniciativas, ainda não existe uma taxonomia completa, compreensiva e em amplo uso por acadêmicos e pela indústria. Além disso, os repositórios das duas taxonomias supramencionadas parecem inativas. O repositório DASP não é mais mantido desde 2019 [7] e o repositório SWC não é mais mantido desde 2020. [8]

Rameder et al. (2022) notam que vários estudos e ferramentas de análise fazem uso dessas taxonomias, mas reconhecem que elas são insuficientes para classificar todas as vulnerabilidades. Diante disso, os autores propuseram uma nova taxonomia que reúne essas duas taxonomias em uma só consolidação. Todavia, existem vulnerabilidades que foram identificadas em trabalhos, mas que não têm correspondência nas taxonomias SWC e DASP.

4.3 MÉTODOS E FERRAMENTAS

Existem diversos métodos para a prevenção, detecção e mitigação de vulnerabilidades. Nesta seção, iremos apresentar o estado da arte de ferramentas de análise de contratos inteligentes da Ethereum bem como os métodos de análise empregados por essas ferramentas.

[7] Veja `https://github.com/CryptoServices/dasp/issues/17`
[8] Veja `https://github.com/SmartContractSecurity/SWC-registry/issues/276`

4.3.1 Métodos de análise

As ferramentas podem ser classificadas de acordo com o método empregado para a detecção de vulnerabilidades. Porém, por vezes as ferramentas misturam diversos métodos a fim de obter melhores resultados, não podendo ser classificada em somente uma categoria. A seguir, apresentamos as diferentes técnicas de análise identificadas nas revisões de Rameder et al. (2022), Kushwaha et al. (2022) e Jimenez Freitez et al. (2009).

4.3.1.1 Análise estática

Uma ferramenta é considerada de análise estática quando ela não necessita que o código-fonte analisado seja executado em seu ambiente usual para a detecção de vulnerabilidades e de outras informações. Porém, algumas técnicas podem criar um número exponencial de caminhos de exploração possíveis, o que pode tornar esse tipo de análise custoso. Além disso, a análise estática não garante a ausência de vulnerabilidades (falsos negativos) ou a presença de vulnerabilidades (falsos positivos).

A técnica de análise estática mais simples é a detecção de padrões de código inseguros. Exemplos de tais padrões são nomes de variáveis confusos ou intencionalmente errôneos, ou mesmo o uso de funções depreciadas e conhecidamente vulneráveis. Um exemplo disso seria o uso da função depreciada `gets()` na linguagem C, que torna o código vulnerável ao ataque de *buffer overflow*.

A **análise léxica** acrescenta uma etapa antes da detecção de padrões, pois transforma o texto em *tokens* que são processados em busca de padrões vulneráveis. Esse método é interessante ao considerar a Ethereum, pois no processo de compilação de código Solidity para EVM ocorre a transformação do código-fonte em uma representação intermediária com *tokens*. Desta forma, a entrada para esse tipo de análise já se encontra disponível após o processo de compilação. Contudo, essa abordagem gera um grande número de falsos positivos pois não considera a gramática e a sintaxe da linguagem analisada.

A **análise de fluxo de dados** considera todos os possíveis caminhos que o fluxo de execução de um programa pode seguir ignorando detalhes irrelevantes, como nomes de variáveis. Quando essa estrutura é organizada em forma de grafo, forma-se um *Grafo de Controle de Fluxo* (CFG). A detecção se dá a partir da análise dos caminhos presentes nesse grafo, considerando padrões e áreas identificados como vulneráveis. Apesar de rápida aplicação na linguagem Solitidy essa técnica tende a ser menos precisa quando uma vulnerabilidade não consegue ser expressa em termos de padrões no CFG. Essa é uma das abordagens usadas para detectar a vulnerabilidade *buffer overflow*.

A **execução simbólica** é uma técnica que executa os comandos do programa de maneira simbólica, isto é, substituindo os valores por símbolos. Esses símbolos são combinados de maneira a expressar diversas cadeias de execuções possíveis. Como essa abordagem cresce de maneira exponencial é necessário que a exploração da cadeia se encerre em um momento.

Quando essa técnica é aliada a execução "concreta"(com valores) ela é chamada de **Concolic** (**Conc**rete + Simb**olic**). Pode-se também combina-las com a **análise de contaminação**. Nessa técnica, uma porção de dados de entrada não confiável proveniente do usuário é marcada como contaminada. A análise consiste na verificação de que o fluxo de execução não atinja partes críticas do sistema sem primeiro tratar a contaminação previamente. Essa abordagem é útil na detecção de padrões de código não tratados.

Para lidar com o alto custo de exploração e a não completude da análise estática, pode-se optar pela interpretação abstrata. Por esse meio as entradas do programa são criadas de maneira a representar uma classe de entradas e não as suas relações exatas, como na execução simbólica. Esse tipo de análise pode garantir a ausência de certos tipos de falha.

4.3.1.2 Análise dinâmica

Na **análise dinâmica** o código-fonte analisado é executado no ambiente em que era esperado executá-lo com o objetivo de gerar informações acerca de sua execução, possibilitando a busca por vulnerabilidades e melhorias.

Uma técnica simples é o uso de valores de entrada aleatórios pertencentes ao domínio de entrada do programa. Essa técnica recebe o nome de *Fuzzing*. Um ponto positivo é que essa técnica não requer conhecimento sobre o funcionamento interno do sistema, apenas de suas entradas. Por esse meio é possível inspecionar o funcionamento do sistema quando ele recebe entradas não esperadas. Isso evidencia erros de validação e de estados não esperados. Entretando, essa técnica não garante que todos os fluxos de execução serão percorridos, possibilitando a presença de brechas na análise. Além disso, em alguns casos a execução do código pode ser excessivamente custosa, tornando-se inviável.

A **instrumentação de código** consiste em realizar alterações no código-fonte a ser analisado. Essas alterações podem ter diferentes objetivos. Um deles é a inserção proposital de falhas no sistema a fim de observar o comportamento do *software* quando erros são introduzidos. Um ponto negativo é que essa técnica requer conhecimento do funcionamento interno do sistema.

O **teste de mutação** é um tipo de instrumentação que visa atestar a qualidade dos casos de teste do sistema. Com essa técnica, uma parte do sistema é modificada de forma a serem inseridas alterações no código que imitam falhas de implementação – as mutações. Se os casos de teste forem robustos o suficiente, eles serão capazes de identificar essas falhas.

A técnica de **contaminação dinâmica** combina a análise de contaminação explicada anteriormente com a execução concreta do código. Desta maneira é possível acompanhar os dados que estão contaminados e avaliar o seu comportamento. Isso é útil para identificar as consequências de dados de entrada não tratados.

4.3.1.3 Especificação formal

A especificação formal consiste na representação intermediária do código-fonte em um modelo matemático formal. Essa técnica usa padrões e modelos para atestar a presença ou ausência de vulnerabilidades e é particularmente apropriada para sistemas que podem ser modelados matematicamente de maneira precisa. A especificação tem 3 fases:

1. especificação de propriedades de segurança
2. extração do modelo do programa
3. verificação das propriedades do modelo

Esse método permite a constatação da ausência de um tipo específico de vulnerabilidade a partir da verificação de modelos formais.

Segundo Rameder et al. (2022) a especificação formal parece oferecer um tipo promissor de análise para *smart contracts*, pois esse tipo de abordagem se beneficia de ambientes que podem ser modelados formalmente. A exemplo: blockchains podem ser modeladas como autômatos finitos. De fato, Wood (2022) afirma que "Ethereum, como um todo, pode ser vista como uma máquina de estados baseada em transações: começamos com um estado gênesis e executamos transações incrementalmente para transformá-la em um estado corrente."(tradução própria [9]). Os autores conjecturam que não foi observada uma ferramenta com esse método por conta do conhecimento específico que é requerido para essa técnica.

4.3.1.4 Inteligência artificial

O uso de inteligência artificial para a detecção de vulnerabilidades tem sido proposto. Técnicas de aprendizagem profunda, aprendizagem com reforço e modelos de linguagem são alternativas para o uso de inteligência artificial nessa área.

Um empecilho para o uso de inteligência artificial na detecção de vulnerabilidades de *smart contracts* é a falta de *benchmarks* e *datasets* de tamanho substancial para permitir o treinamento eficiente de modelos.

4.3.1.5 Auditoria especializada

Também é importante notar que uma parcela significativa da análise de segurança de software é feita de maneira não automatizada a partir de especialistas em segurança (Jimenez Freitez et al., 2009). Essa pode ser uma abordagem viável caso os participantes interessados na análise de um *smart contract* disponham de recursos financeiros suficientes para esse serviço. O ecossistema Ethereum dispõe de algumas organizações especializadas em segurança de *smart contracts*. Empresas como Certik (2023), Consensys (2023b) e Hacken (2023) oferecem serviços

[9] No original: *"Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some current state."*

de consultoria e de auditoria de segurança a outros participantes interessados em elevar os aspectos de segurança de suas implementações.

Uma outra maneira mais descentralizada de se atingir esse objetivo é adotar um programa de recompensas por bugs encontrados. Nesta modalidade, especialistas são incentivados a encontrar vulnerabilidades para obter recompensas financeiras e reputação.

4.3.2 Ferramentas de análise de contratos inteligentes

Algumas revisões foram conduzidas com o objetivo de avaliar a quantidade e a qualidade de artigos que propõem ou comparam ferramentas de análise e as técnicas de detecção de vulnerabilidades mais utilizadas.

A revisão recente mais compreensiva encontrada durante esta escrita foi a revisão sistemática conduzida por Rameder et al. (2022). De um total inicial de +1300 artigos foram selecionados 303 relacionados a ferramentas de análise de contratos inteligentes da Ethereum. Esses trabalhos foram filtrados e categorizados de acordo com o escopo, o tipo de pesquisa e a qualidade. Ao final do crivo dos autores, foram selecionadas 8 revisões sistemáticas, 38 *surveys* e 149 estudos primários. Dentre outros resultados, a revisão resume o estado da arte em relação a ferramentas e métodos disponíveis, além de expor taxonomias e *datasets*. Ao considerar apenas ferramentas de código aberto, os autores reduziram o escopo para 83 ferramentas. Apesar disso, grande parte das ferramentas analisadas encontram-se inativas. Ao final, somente 20 foram consideradas ativas pelos autores, veja Tabela 4.1.

No artigo de Kushwaha et al. (2022) foram analisadas 86 ferramentas em um subconjunto de contratos vulneráveis derivados do *dataset* SolidiFi. O objetivo desta pesquisa foi observar os métodos empregados pelas ferramentas e as vulnerabilidades que elas detectavam. Foi observado que cerca de 3/4 das ferramentas empregavam o método de análise estática. Além disso também foi observado o número de novas ferramentas, ano a ano, veja a Figura 4.2. Os autores também percebem a falta de um *benchmark* padrão amplamente adotado pelo campo e apontam que grande parte das ferramentas analisadas não são de código aberto, o que dificulta a verificação dos resultados por elas reportados.

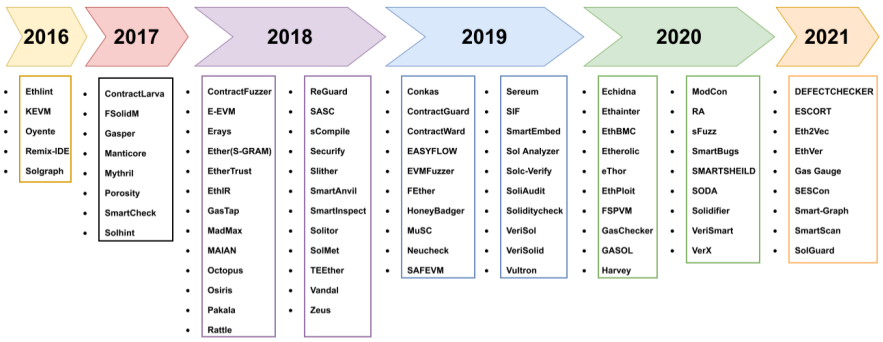

Figura 4.2: Evolução ano a ano das ferramentas de análise para contratos inteligentes baseados na blockchain Ethereum. (Kushwaha et al., 2022)

Tabela 4.1: Ferramentas publicadas em ou antes de 2019 que são mantidas e em uso (Rameder et al., 2022).

Ferramenta	**Publicado em**	**Repositório público em github.com**
ContractLarva	2019–08	gordonpace/contractLarva
Ethersplay	2018–05	crytic/ethersplay
Gigahorse	2019–01	nevillegrech/gigahorse-toolchain
ILF	2019–11	eth-sri/ilf
KEVM	2019–07	kframework/evm-semantics
KEVM	2018–10	runtimeverification/verified-smart-contracts
MadMax	2018–09	nevillegrech/MadMax
Manticore	2017–02	trailofbits/manticore
Mythril	2017–10	ConsenSys/mythril
Oyente	2016–01	enzymefinance/oyente
PASO	2019–09	aphd/paso
Remix	2014–11	ethereum/remix-project
Securify	2018–09	eth-sri/securify2
Slither	2018–10	crytic/slither
SmartBugs	2019–10	smartbugs/smartbugs
SmartCheck	2018–05	smartdec/smartcheck
SOLC	2019–07	SRI-CSL/solidity/blob/0.7/SOLC-VERIFY-README.md
Solhint	2017–10	protofire/solhint
teEther	2019–02	nescio007/teether
Vertigo	2019–08	JoranHonig/vertigo

O experimento empírico proposto por Durieux et al. (2020) analisou 9 ferramentas extensivamente com um grande número de contratos. Diferentemente dos outros trabalhos, este visou principalmente o aspecto empírico com contratos utilizados em ambiente de produção e não contratos construídos para expor vulnerabilidades específicas. Os resultados concluíram que 97% dos contratos foram considerados vulneráveis, o que pode indicar um grande número de falsos positivos. Além disso, também foi constatado que a combinação de certas ferramentas podem oferecer maior acurária ao considerar o tempo de execução das ferramentas.

Os 3 estudos supramencionados comentam e comparam os resultados das ferramentas, seus métodos e tempos de execução. Alguns pontos de consenso entre eles são: 1. A ferramenta Manticore tem o maior tempo médio de execução, 2. as ferramentas Slither, Mythril e Oyente constumam ter uma maior acurácia e 3. o campo carece de padrões de *benchmarks* e taxonomias.

Em suma, no tocante a quantidade de ferramentas disponíveis o campo conta com ampla variedade. Apesar disso, a qualidade das ferramentas é variável e nem sempre elas são mantidas a ponto de serem utilizadas no longo termo.

4.4 BENCHMARKS E DATASETS

Na revisão de Rameder et al. (2022), foram identificados 33 datasets públicos que poderiam ser usados independentemente da ferramenta escolhida. Desses, 22 contêm código-fonte Solidity. Alguns datasets incluem vulnerabilidades identificadas e revisadas para serem

falsos positivos ou verdadeiros positivos. Esse esforço se faz valioso ao comparar ferramentas de análise, pois como apontado por Durieux et al. (2020): "Ao considerarmos o maior dataset, observamos que 97% dos contratos foram etiquetados como vulneráveis, sugerindo um número considerável de falsos-positivos"(tradução própria [10]). O número de contratos anotados nos datasets varia de nenhum até 587 contratos. O número total de contratos não anotados varia de 6 até 47.541.

Já na revisão performada por Kushwaha et al. (2022), foram selecionados 30 contratos vulneráveis do benchmark SolidiFi (Ghaleb e Pattabiraman, 2020). Esse dataset também foi usado parcialmente por Durieux et al. (2020) para a construção do dataset $SB^{CURATED}$.

4.5 RESUMO

Ataques a contratos inteligentes na blockchain Ethereum podem resultar em prejuízos financeiros significativos, o que destaca a necessidade de prevenção por meio da detecção prévia de vulnerabilidades. Embora a segurança em *smart contracts* muitas vezes seja responsabilidade dos desenvolvedores e empresas de consultoria e auditoria de segurança, ferramentas automatizadas de análise também são essenciais para identificar vulnerabilidades e melhorar a segurança em blockchain.

No entanto, revisões como de Kushwaha et al. (2022) e Rameder et al. (2022) mostram que a qualidade das ferramentas de análise de *smart contracts* é questionável e muitas delas não são atualizadas regularmente. Além disso, a falta de padronização e taxonomia dificulta a comparação de resultados. A maioria das ferramentas de análise também depende de *datasets* e *benchmarks* pouco expressivos da real condição de contratos inteligentes publicados em produção na Ethereum, e ainda lidam com poucos contratos anotados para lidar com falsos positivos.

Todos esses fatores indicam um campo imaturo e em desenvolvimento, ainda muito dependente de análises manuais. Por isso, Rameder et al. (2022) sugere revisões regulares do estado da arte de ferramentas de análise para acompanhar a evolução do campo e garantir a melhoria contínua da segurança em *smart contracts*.

[10] No original: *"When considering the largest dataset, we observed that 97% of contracts are tagged as vulnerable, thus suggesting a considerable number of false positives."*

5 ARTIGO DE REFERÊNCIA

Em 2020, Durieux et al. (2020) realizaram a análise de 47.398 contratos inteligentes da Ethereum utilizando 9 ferramentas de análise de vulnerabilidades em um tempo de execução total de 564 dias e 3 horas. Esse estudo foi, no momento da publicação do artigo, o maior em termos de tempo de execução das análises.

No artigo, os autores percebem um considerável esforço na implementação de ferramentas de análise tanto pela indústria e comunidades quanto por meios acadêmicos. Contudo, não observam um meio comum para a execução das ferramentas e consequente comparação de resultados de maneira a atestar a efetividade das ferramentas. Por esse motivo, os autores propuseram um estudo que teve como contribuições a construção de dois novos *datasets*, o framework de execução *Smartbugs* e os resultados da execução desse framework em um grande número de smart contracts.

Neste capítulo, descreveremos esse estudo em mais detalhes, pois ele será usado como referência para comparação dos resultados emergentes desta monografia e norteador da metodologia aplicada.

5.1 DESENHO DO ESTUDO

O estudo foi desenhado de forma a reportar sobre o estado da arte dessas ferramentas, no tocante a efetividade, reprodutibilidade e performance.

5.1.1 Seleção das ferramentas

Primeiramente, foi feita a seleção das ferramentas a serem consideradas para o estudo. O levantamento inicial de ferramentas em potencial foi feito a partir de surveys, pesquisas de literatura acadêmica e buscas feitas pelos próprios autores em bases do Github e do Google. Esse levantamento gerou uma lista com 35 ferramentas, porém nem todas eram apropriadas para o estudo. Portanto, foram incluídas na seleção final somente as ferramentas que atendem a todos os seguintes critérios:

1. tem código-fonte disponível publicamente;
2. oferece interface para linha de comando (CLI);
3. tem como entrada código-fonte na linguagem Solidity;
4. requer somente o código-fonte para a sua execução;
5. identifica vulnerabilidades ou más práticas.

Esses critérios têm como objetivo incluir apenas ferramentas que seguem um padrão de escopo, entrada e saída da execução, de forma a facilitar a implementação e aplicação do framework de execução (Durieux et al., 2020). Os critérios também removem ferramentas que não disponibilizam o código-fonte publicamente, o que dificulta a reprodutibilidade e verificação dos resultados dos experimentos empíricos. Também nota-se que algumas ferramentas publicam resultados sem os *datasets* utilizados pelas mesmas (Rameder et al., 2022).

Após a aplicação dos critérios de seleção, foram selecionadas 9 ferramentas, sendo elas:

1. **Oyente**: Uma das primeiras ferramentas de análise, que é utilizada como base para outras ferramentas. Ela usa a execução simbólica sobre a EVM para identificar vulnerabilidades.

2. **HoneyBadger**: Baseado na ferramenta Oyente, também usa execução simbólica em conjunto com heurísticas para detectar *honeypots*. *Honeypots* são contratos feitos para parecerem vulneráveis, mas que na verdade não são.

3. **Maian**: Também baseada na ferramenta Oyente. Essa ferramenta tem um foco maior na detecção de fluxos de execução que bloqueiam ETH, seja por meio da destruição do contrato ou por pela falta funções de resgate de ETH. Além disso, essa ferramenta executa a verificação da vulnerabilidade a partir de uma blockchain privada.

4. **Manticore**: Essa ferramenta também usa a execução simbólica para explorar caminhos de execução que podem levar ao encerramento do contrato.

5. **Mythril**: Desenvolvida pela Consensys (2023b), essa ferramenta utiliza o método de execução simbólica e concreta (*concolic*), análise de contaminação e análise do CFG para a detecção de vulnerabilidades.

6. **Osiris**: Essa ferramenta amplia o escopo da ferramenta Oyente para poder detectar vulnerabilidades aritméticas (*overflow/underflow*).

7. **Securify**: Emprega a análise estática do bytecode EVM para coletar informações semânticas que auxiliam na prova de propriedades.

8. **Slither**: Ferramenta de análise estática que detecta vulnerabilidades através da conversão do código Solidity para uma representação intermediária e posterior análise via CFG e análise de contaminação.

9. **SmartCheck**: Utiliza análise léxica e sintática para detectar padrões vulneráveis e más práticas, se caracterizando como uma ferramenta de análise estática.

5.1.2 Construção dos Datasets

Um problema notado pelos autores foi a falta de *datasets* comuns às ferramentas de análise. Muitas ferramentas costumam ter os contratos vulneráveis presentes em seus repositórios, já outras usam *datasets* não padronizados. Para lidar com essa questão, os autores propuseram a construção de dois *datasets*:

- $\text{SB}^{\text{CURATED}}$: Um *dataset* que contém 69 contratos anotados com exemplos de vulnerabilidades conhecidas e bem estudadas para servir como benchmark.
- SB^{WILD}: Um *dataset* que contém 47.398 contratos em uso em ambiente real, extraídos da Ethereum e que satisfazem alguns critérios de seleção.

O *dataset* $\text{SB}^{\text{CURATED}}$, foi construído com base em contratos vulneráveis coletados em três fontes: 1. repositórios do Github, 2. postagens em blogs confiáveis que discorrem sobre segurança em contratos e 3. contratos conhecidamente vulneráveis da Ethereum. Os contratos foram anotados com base na taxonomia DASP (Group, 2023) resultando em um *dataset* com 115 vulnerabilidades identificadas e classificadas, distribuídas em um total de 3.799 linhas de código. Como esse *dataset* oferece informações sobre verdadeiros-positivos e falsos-positivos, ele pode ser utilizado como benchmark para avaliar a efetividade das ferramentas analisadas.

Em contrapartida, o dataset SB^{WILD} foi construído com o objetivo de analisar o comportamento das ferramentas de análise quando expostas a códigos-fonte de aplicações em ambiente de produção na Ethereum. Esse *dataset* oferece uma amostragem muito mais ampla e caótica do que o anterior.

Para a construção do *dataset* SB^{WILD}, foi necessário obter os endereços dos contratos publicados na blockchain Ethereum. Os endereços desses contratos foram obtidos através da base de dados do Google BigQuery (Day e Medvedev, 2018) com a seguinte query SQL:

```
SELECT contracts.address, COUNT(1) AS tx_count
 FROM `ethereum_blockchain.contracts` AS contracts
 JOIN `ethereum_blockchain.transactions` AS transactions
      ON (transactions.to_address = contracts.address)
 GROUP BY contracts.address
 ORDER BY tx_count DESC
```

A query foi executada em 08/08/2019 e retornou uma tabela com os endereços de 2.263.096 contratos e suas respectivas quantidades de transações. É importante notar que essa query retorna somente contratos com pelo menos uma transação confirmada, ou seja, não é representativa da quantidade total de contratos publicados na Ethereum.

Em posse dos endereços dos contratos relevantes à pesquisa, foi possível extrair o código-fonte pela plataforma Etherscan (2023) através de sua API pública. Essa plataforma oferece um serviço gratuito de verificação de código-fonte de contratos inteligentes da Ethereum em que usuários disponibilizam o código-fonte para auditoria pública e transparência.

A próxima etapa foi o tratamento dos contratos recuperados. Notou-se que 95% dos contratos extraídos durante a etapa anterior eram duplicatas. Conteúdos repetidos foram identificados a partir do hash md5 do código-fonte do contrato, retirando-se comentários e tabulações. A tabela 5.1 resume o resultado final da construção do dataset SB^{WILD} após o tratamento final.

Tabela 5.1: Estatísticas de coleção de contratos inteligentes solidity da blockchain Ethereum (Durieux et al., 2020)

Código-fonte não disponível	1.290.074
Código-fonte disponível	972.975
Código-fonte não acessível	47
Total	2.263.096
Contratos únicos	47.518
Linhas de código	9.693.457

5.1.3 Taxonomia

A taxonomia DASP foi adotada pelo estudo para a classificação das vulnerabilidades. Como as ferramentas não têm uma saída padronizada nessa taxonomia, os autores propuseram um mapeamento manual entre os achados de cada ferramenta e a classificação DASP. Para assegurar a classificação das vulnerabilidades encontradas corretamente, os resultados individuais de cada ferramenta foram mapeados na etapa de consolidação de resultados através de um script Python. A *dataset* $SB^{CURATED}$ também foi mapeado para conter anotações seguindo essa classificação.

Entretando, os autores reconhecem que essa taxonomia não é suficiente para a discriminação das vulnerabilidades encontradas pelas ferramentas, pois ela contém uma categoria coringa (DASP-10 Unknown unknowns), que compreende todas as vulnerabilidades não previstas por essa taxonomia. Apesar disso, a adoção de uma taxonomia, mesmo que insuficiente, permite a agregação e comparação de resultados entre as ferramentas, o que vai de acordo com os objetivos do estudo.

5.1.4 Ambiente de execução

Motivados por um notável incremento no desenvolvimento de ferramentas de análise para smart contracts, Ferreira et al. (2020) propuseram o framework *Smartbugs*. Esse framework possibilita a configuração de ambientes de execução baseados em imagens Docker (2023). Isso possibilita a execução das ferramentas em ambientes padronizados, independentes da plataforma na qual estão rodando. O framework também trata as entradas e saídas das execuções de maneira a padronizar os artefatos gerados no processo.

Como as análises são executadas dentro de imagens Docker, é garantido que o ambiente de execução seja igual, independentemente da plataforma de instalação do framework. Ademais, as execuções podem ser feitas paralelamente e em ambientes totalmente independentes. Isso possibilita o uso de diversos núcleos de processamento para a execução das análises.

O framework foi desenhado de forma a ser facilmente extensível para comportar novas ferramentas e suas atualizações. Há também a possibilidade de configurar diversas versões para a mesma ferramenta, pois existem versões das ferramentas que são compatíveis com certas versões da lingugagem Solidity. Atualmente, o repositório conta com 18 ferramentas pré-configuradas, das quais 13 aceitam código-fonte Solidity como entrada. Outras ferramentas aceitam bytecode EVM ou código de máquina [1].

5.2 RESULTADOS

Para avaliar a acurácia, as ferramentas foram executadas sobre os contratos do *dataset* $SB^{CURATED}$, pois esse *dataset* contém informações sobre verdadeiros-positivos e falsos-positivos.

No total, as ferramentas foram capazes de encontrar 42% (48 de 115) das vulnerabilidades das presentes. A ferramenta Myhtil obteve a melhor acurácia, detectando 27% de todas as vulnerabilidades presentes no *dataset*. Em seguida, a ferramenta Slither obteve 17%.

Todavia, nenhuma das 9 ferramentas foi capaz de detectar vulnerabilidades nas categorias Bad randomness (DASP-6) e Short Address (DASP-9). Esse resultado é esperado, pois as ferramentas não se propõem a detectar todas as vulnerabilidades. Em contrapartida, as vulnerabilidades Arithmetic (DASP-3), Reentrancy (DASP-1), Time Manipulation (DASP-8), Unchecked low level calls (DASP-4) e Unknown unknowns (DASP-10) foram as mais comumente encontradas. Isso evidencia a oportunidade para a criação e evolução de ferramentas capazes de detectar outras categorias de vulnerabilidades.

Os autores também testaram combinações de ferramentas a fim de considerar o custo-benefício entre acurácia e tempo de execução. Nesse quesito a combinação Mythril e Slither foi capaz de detectar 37% das vulnerabilidades. Isso ainda é inferior aos 42% resultantes de todas as ferramentas combinadas, mas ainda assim os autores recomendam essa combinação ao considerar o tempo de execução das análises.

Outra parte da pesquisa focou em analisar o comportamento das ferramentas para contratos em ambiente de produção, publicados na Ethereum, através do *dataset* SB^{WILD}. Ao considerar a análise das 9 ferramentas selecionadas sobre notou-se uma significativa diferença entre os tempos de execução das ferramentas, provavelmente atrelado aos métodos empregados pelas mesmas. A ferramenta Manticore obteve o maior tempo médio de execução, próximo de 24 minutos. Em comparação, as outras ferramentas tiveram tempos médios próximos a 2 minutos. A mais rápida entre elas – Slither – necessitou em média apenas 5 segundos para analisar os contratos. Os autores destacam que não observaram correlação entre a acurácia e o tempo de execução das ferramentas.

[1] Veja `https://github.com/smartbugs/smartbugs`

6 METODOLOGIA

Neste capítulo, apresentamos a metodologia adotada neste trabalho a respeito das etapas de pesquisa, desenho do estudo, implementação, processamento e análise de resultados.

6.1 PESQUISA

A pesquisa inicial deu-se a partir de buscas em bases de dados de artigos acadêmicos acessíveis a partir da rede da Universidade Federal do Paraná (UFPR), repositórios no Github, artigos e iniciativas comunitárias sobre segurança em *smart contracts* da Ethereum. Foram usadas as palavras-chave *"Survey", "Review", "Ethereum smart contract", "Security"e "Analysis tools"* em combinação. Para a fundamentação teórica foram consultadas as referências originais – citadas nos trabalhos de revisão – que deram início à tecnologia blockchain e particularmente a Ethereum, como Nakamoto (2023), Buterin (2014) e Wood (2022). Como o tema circunda uma tecnologia relacionada à economia também foi consultada a literatura referente a conceitos econômicos que podem ser relevantes para o entendimento do impacto deste trabalho, como a história da moeda a partir de livros e materiais de aula pelos autores Mankiw (2021) e Dotsis (2019).

Este trabalho foi desenhado de forma a replicar parte do estudo original proposto por Durieux et al. (2020) com dados atualizados. Isso corrobora com a recomendação de Rameder et al. (2022), sobre realizar revisões constantes sobre o estado das ferramentas de análise no campo.

6.2 SELEÇÃO DE FERRAMENTAS

A primeira etapa a ser feita para replicar os resultados de Durieux et al. (2020) é a seleção de ferramentas. O levantamento inicial será feito a partir das 19 ferramentas disponíveis no *framework* de execução *smartbugs*. Porém, devido ao tempo limitado disponível para o presente estudo, aliado ao fato de que o estudo de referência reporta um tempo de execução de 564 dias e 3 horas com apenas 9 ferramentas, será feita a redução do escopo de estudo no tocante a quantidade de ferramentas analisadas. Além disso, algumas ferramentas, apesar de suportadas pelo *framework*, não são mantidas atualizadas e não são compatíveis com as versões atuais da linguagem solidity. Para comportar esses novos requisitos serão considerados alguns critérios adicionais para a seleção de ferramentas. Para ser selecionada, a ferramenta:

1. **deve ter interface para a linha de comando (CLI)**;
2. **deve ter como entrada código solidity**;
3. **requer somente o código-fonte solidity para a sua execução**;

4. **identifica vulnerabilidades ou más práticas**;

5. **tem repositório com última data de atualização posterior a janeiro de 2021** – esse critério visa eliminar ferramentas que foram abandonadas ou que não comportam as versões recentes da linguagem solidity;

6. **tem tempo médio de execução inferior a 10 minutos** – tempo razoável para comportar as limitações deste estudo;

A tabela 6.1 mostra todas as ferramentas pré-configuradas no framework *smartbugs* e os resultados dos critérios de seleção expandidos.

Tabela 6.1: Ferramentas de análise e os critérios de seleção

Repositório da ferramenta	**Seleção**
`https://github.com/christoftorres/ConFuzzius`	sim
`https://github.com/smartbugs/conkas`	sim
`https://zenodo.org/record/3760403`	rejeitado pelo critério 2
`https://secpriv.wien/ethor/`	rejeitado pelo critério 2
`https://github.com/smartbugs/HoneyBadger`	rejeitado pelo critério 4
`https://github.com/nevillegrech/MadMax`	rejeitado pelo critério 2
`https://github.com/smartbugs/MAIAN`	sim
`https://github.com/trailofbits/manticore`	rejeitado pelo critério 6
`https://github.com/ConsenSys/mythril`	sim
`https://github.com/smartbugs/Osiris`	rejeitado pelo critério 5
`https://github.com/smartbugs/oyente`	sim
`https://github.com/palkeo/pakala`	rejeitado pelo critério 2
`https://github.com/eth-sri/securify`	rejeitado pelo critério 5
`https://github.com/duytai/sFuzz`	sim
`https://github.com/crytic/slither`	sim
`https://github.com/smartdec/smartcheck`	rejeitado pelo critério 5
`https://github.com/protofire/solhint`	rejeitado pelo critério 4
`https://github.com/protofire/solhint` 3.3.8	sim
`https://github.com/nescio007/teether`	rejeitado pelo critério 2
`https://github.com/usyd-blockchain/vandal`	rejeitado pelo critério 2

Ao final, foram selecionadas 8 ferramentas a serem consideradas para o estudo. São elas:

1. **Oyente**: Uma das primeiras ferramentas de análise, utilizada como base para outras. Utiliza execução simbólica para a detecção de vulnerabilidades.

2. **Maian**: Baseada na ferramenta Oyente, utiliza a execução simbólica para verificar vulnerabilidades que bloqueiam ETH.

3. **Mythril**: Desenvolvida pela empresa Consensys (2023b), utiliza execução simbólica e concreta (*concolic*), análise de contaminação e CFG para a detecção de vulnerabilidades.

4. **Slither**: Ferramenta de análise estática que detecta vulnerabilidades através da conversão do código Solidity para uma representação intermediária e posterior análise via CFG e análise de contaminação.

5. **ConFuzzius**: Ferramenta que usa fuzzing em conjunto com a execução simbólica para detectar vulnerabilidades, denominado fuzzer híbrido (Torres et al., 2021).

6. **Conkas**: Ferramenta modular que usa execução simbólica e representação intermediária para detectar vulnerabilidades em 5 das 10 categorias DAPS (Veloso, 2021).

7. **sFuzz**: Fuzzer de iniciativa *open-source* que faz uso do compilador Solidity.

8. **Solhint**: Ferramenta *open source* desenvolvida pela comunidade inicialmente como um *linter*, mas que passou a recomendar algumas alterações de segurança.

6.3 COLEÇÃO DOS CÓDIGOS-FONTE

A Ethereum possui hoje mais de 50 milhões de contratos publicados em sua blockchain (Day e Medvedev, 2018), portanto faz-se necessária uma redução do escopo de contratos a serem incluídos nesta análise para torná-la viável. Nesta seção, apresentaremos a metodologia de coleção de códigos-fonte utilizada neste trabalho.

6.3.1 Atualização do dataset SB$^{\text{WILD}}$

O dataset SB$^{\text{WILD}}$ utilizado por Durieux et al. (2020) foi consolidado em 08/08/2019 (Smartbugs, 2020). Como um dos objetivos do presente estudo é estudar e comparar os resultados em um *dataset* atual, é preciso atualizar tanto os códigos-fonte quanto os meta-dados dos contratos considerados para a análise.

O artigo de referência considerou relevantes apenas os contratos que possuem código-fonte disponível na plataforma *Etherscan* e cujo saldo em ETH fosse positivo. Neste trabalho também aplicaremos essas condições, além de filtros adicionais que garantem o funcionamento do *framework* de execução.

Para a obtenção dos códigos-fonte é necessário obter os endereços dos contratos relevantes a este estudo. Esses endereços serão obtidos através da base de dados do *Google BigQuery*, que disponibiliza os dados de transações e contratos da blockchain Ethereum a partir de um banco relacional (Day e Medvedev, 2018). A consulta será feita através da seguinte *query SQL*:

```
1 SELECT
2    contracts.address AS address,
3    COUNT(1) AS tx_count,
4    MAX(balances.eth_balance) AS eth_balance
5  FROM `bigquery-public-data.crypto_ethereum.contracts` AS contracts
6  JOIN `bigquery-public-data.crypto_ethereum.transactions` AS transactions
```

```
  ON (transactions.to_address = contracts.address)
JOIN `bigquery-public-data.crypto_ethereum.balances` AS balances
  ON (transactions.to_address = balances.address)
GROUP BY contracts.address
ORDER BY tx_count DESC, eth_balance DESC
```

Essa consulta retorna os endereços de todos os contratos com pelo menos uma transação confirmada na blockchain, bem como o saldo e a quantidade de transações de cada registro. Essas informações serão usadas para aplicar as condições de corte do dataset.

Em posse dos endereços dos contratos e seus respectivos saldos, será realizada a coleta dos códigos-fonte dos contratos com saldo positivo a partir da plataforma *Etherscan*. Por disponibilizar o seu serviço de forma pública, a plataforma limita a quantidade de requisições concorrentes permitidas em uma única chave de API. Isso limita a velocidade de obtenção dos códigos-fonte. Portanto, no interesse efetuar a coleta de maneira mais ágil, a etapa de corte do *dataset* será efetuada **antes** da etapa de coleta. No artigo de referência, os autores realizam a coleta dos códigos-fonte antes da remoção de duplicatas e de saldos nulos. O resultado dessa diferença na ordem das etapas será explicitado no capítulo de resultados.

A triagem será feita através de um *script* Python que itera sobre as linhas da tabela resultante da consulta de endereços e verifica se o saldo é positivo. Em caso verdadeiro, o *script* fará a requisição à API pública da plataforma *Etherscan* com uma chave de API e fará coleta o código-fonte do contrato em um diretório de saída.

6.3.2 Tratamento do dataset

Durante a etapa de coleção dos códigos-fonte, foi observado que o formato de resposta da API não era uniforme. As respostas às requisições respeitam 3 formatos:

1. Arquivo texto com o código-fonte puro em solidity e sem tratamentos adicionais;
2. Resposta em formato JSON, contendo meta-dados sobre o contrato publicado, bem como o código-fonte em si;
3. Resposta em formato JSON modificado. Esse formato não respeita exatamente o formato JSON e contém apenas a parte da estrutura que contém o código-fonte, semelhante à estrutura retornada no formato 2.

Porém, a entrada do framework de execução espera arquivos em código-fonte solidity e não em formato JSON. Os recursos do artigo de referência não fazem menção a qualquer tratamento adicional da resposta da API, sugerindo uma possível mudança no formato de resposta esperado. Na data de consolidação do dataset SB^{WILD} a linguagem solidity se encontrava na versão 0.5.10. Desde então houveram diversas atualizações da linguagem [1]. Certas atualizações

[1] https://github.com/ethereum/solidity/releases/tag/v0.5.10

corrigem algumas vulnerabilidades previstas pelas ferramentas de análise, como *overflow* e *underflow*. Já outra atualização permitiu o *upload* de diversos códigos-fonte de contratos em um mesmo endereço na blockchain. Isso possibilitou uma melhor organização dos arquivos-fonte ao considerar a experiência de desenvolvimento e publicação, impulsionando o desenvolvimento de contratos mais complexos. Contudo, o formato de retorno do Etherscan mudou para comportar essa nova estrutura e o novo formato não foi previsto pelo framework *Smartbugs*. Portanto, será necessária uma etapa intermediária de pré-processamento.

Nessa etapa, a resposta da API é primeiro classificada em um dos 3 formatos de resposta. Se a resposta respeita o formato 1, então ela é copiada de maneira integral ao diretório de saída, dado que ela já contém o código-fonte puro e no formato esperado, sem necessidade de tratamentos adicionais. Porém, se a resposta respeita os formatos 2 ou 3 é necessário que apenas o código-fonte relevante seja extraído. Para transformar essas respostas em arquivos solidity são realizadas as seguintes etapas:

1. Os códigos-fonte dos contratos são analisados de forma a serem extraídas informações sobre versão, definições de contratos e dependências de contratos;
2. Um grafo de dependências é construído com base nessas informações;
3. Um algoritmo de ordenação topológica é executado sobre o grafo de maneira a garantir a ordem correta das dependências;
4. O conteúdo dos contratos é tratado de forma a evitar repetições desnecessárias;
5. Todo o conteúdo tratado dos contratos é concatenado na ordem correta;
6. O resultado da concatenação é testado pelo compilador *solc* na versão relevante;
7. Caso o compilador tenha uma execução bem-sucedida, o conteúdo da concatenação é copiado para o diretório de saída. Caso contrário, ele é ignorado e marcado como inválido.

Essas etapas têm como objetivo adaptar grande parte das respostas para conter apenas o código-fonte solidity em um arquivo não acompanhado de seus meta-dados. Apesar desse esforço, a resposta da API não garante que a ordem dos contratos seja a correta. Em alguns casos os contratos têm dependências de outros contratos. Tais falhas são acusadas pelo compilador *solc*, que é utilizado para validar a correta ordenação.

Ao final desse processo os arquivos se encontram prontos para serem utilizados pelo framework *smartbugs*.

6.4 EXECUÇÃO DAS ANÁLISES

Para o processamento das análises, foi utilizado o framework de execução *smartbugs* descrito no capítulo anterior na versão 2.0.7 publicada em 17/02/2023. Os autores dessa ferramenta a desenharam de maneira a permitir a fácil configuração e execução de novas ferramentas e *datasets* de maneira independente de plataforma, sendo necessário apenas acesso à internet, python3 e docker como dependências. Uma máquina virtual com essas dependências foi requisitada para o trabalho e foi fornecida pelo C3SL (Centro de Computação Científica) da UFPR [2]. A máquina dispunha de 32 núcleos virtuais de processamento AMD EPYC 7401 24-Core Processor de 2 MHz, 32 GB de memória RAM e 100 GB de armazenamento para os resultados das análises. Nela, foram instaladas as dependências docker na versão 20.10.21 e python na versão 3.9.2. Esses recursos foram requisitados levando em consideração a capacidade de executar as análises de forma simultânea e independente.

O *dataset* atualizado foi dividido em 16 partes de tamanhos relativamente parecidos com cerca de 1700 contratos cada uma. Cada parte foi processada com 31 threads de processamento. Um núcleo de processamento foi deixado livre para acesso SSH e outras tarefas de coleção de resultados. Essa divisão foi necessária para comportar o limite de argumentos da linha de comando e para inspecionar erros que ocorriam em partes específicas do *dataset*. Isso possibilitou a identificação de mais contratos inválidos que foram rejeitados posteriormente.

6.5 PÓS-PROCESSAMENTO

Foi observado durante a execução das análises diversas situações não previstas. Alguns contratos não puderam ser compilados pelas ferramentas ou pelo compilador *solc*, mesmo tendo sido aprovados na etapa de pré-processamento e sendo executados corretamente em outras ferramentas. Algumas ferramentas encontraram exceções no código da ferramenta durante a sua execução. Outras execuções foram interrompidas pois excederam o tempo máximo previsto para uma análise. Todavia, o artigo de referência não faz menção a qualquer tratamento de erros a não ser *timeouts*. Os autores destacam que a única ferramenta que teve a dua execução interrompida em decorrência disso foi a ferramenta Manticore, que não foi incluída neste trabalho exatamente por esse motivo. Diante disso, a abordagem adotada no presente estudo foi considerar as análises com casos adversos como inválidas e calcular as métricas de resultados somente sobre casos bem-sucedidos.

Em seguida, foi feita a compilação dos resultados a partir dos arquivos de saída das ferramentas em diretórios para um arquivo *csv* resumido, contendo o endereço do contrato analisado, a duração do experimento, falhas e achados de cada ferramenta. O programa responsável por essa etapa foi implementado como parte do *framework Smartbugs*. Neste módulo,

[2] `https://www.c3sl.ufpr.br/`

cada ferramenta contém arquivos correspondentes que especificam como executar a ferramenta e como extrair os seus resultados [3].

Algumas ferramentas continham sua classificação própria e outras adotavam DASP ou SWC como suas taxonomias. Além disso, a nomenclatura das vulnerabilidades varia entre ferramentas. Por exemplo, a vulnerabilidade DASP-7 Front Running também é chamada de Transaction Order Dependence (TOD) em algumas ferramentas e na taxonomia SWC. Portanto, foi feito o mapeamento das vulnerabilidades reportadas em cada ferramenta para a classificação da taxonomia DASP. Porém, ao executar essa etapa notou-se que o mapeamento proposto por Durieux et al. (2020) estava desatualizado e não correspondia aos nomes de vulnerabilidades reportados pelas ferramentas. Com isso, fez-se necessária uma etapa de classificação manual entre as vulnerabilidades encontradas pelas ferramentas e a taxonomia adotada originalmente. Essa classificação levou em consideração a descrição das vulnerabilidades presentes nos repositórios e artigos de cada ferramenta. Quando uma ferramenta adota a taxonomia SWC, utilizou-se a taxonomia consolidada de Rameder et al. (2022) para encontrar uma correspondente na taxonomia DASP. A tabela A.1 presente no apêndice A mostra todas as vulnerabilidades encontradas pelas ferramentas e a classificação adotada em cada uma delas. É importante notar que alguns itens reportados pelas ferramentas não se caracterizam como vulnerabilidades e sim como otimizações de código ou sugestões de alterações para comportar padrões de projeto amplamente aceitos. Esses itens foram ignorados.

[3]Cada ferramenta tem seu tratamento específico. Veja: `https://github.com/smartbugs/smartbugs/tree/master/tools`

7 RESULTADOS

Neste capítulo, apresentaremos os resultados desta monografia, compreendendo a construção do *dataset* atualizado, a execução das análises de vulnerabilidades, comparações com o artigo de referência e, por fim, a discussão dos resultados obtidos.

7.1 DATASET ATUALIZADO

A construção do novo *dataset* iniciou no dia 09/01/2023. Nesse dia, foi realizada a consulta ao banco relacional do Google BigQuery iniciando a coleta de endereços dos contratos que respeitam os critérios especificados no capítulo anterior. Essa data foi escolhida de forma a incluir todos os contratos publicados na Ethereum até o final de 2022 e em parte de 2023. Por conta disso, o *dataset* atualizado foi nomeado SB^{2022}.

A consulta inicial retornou um total de 7.029.505 resultados, contendo endereços de contratos, quantidades de transações e saldos. Os dados foram salvos em um arquivo no formato *csv* e estão disponíveis no repositório do *dataset* [1] . Em seguida, foi realizado o primeiro corte que manteve somente contratos cujo saldo fosse positivo. Nessa etapa, observou-se que 6.574.880 (93%) contratos não possuiam saldo positivo. Dos restantes, apenas 297.993 (65%) continham código-fonte disponível na plataforma Etherscan. Por motivos variados, como por exemplo a falta da diretiva *pragma* indicando versões, 1.446 contratos foram considerados inválidos e foram removidos do *dataset*. Para a remoção de duplicatas e contagem de linhas de código, foi utilizado o programa *cloc*, assim como no artigo de Durieux et al. (2020). Foi observado que 269.873 (90%) contratos foram considerados duplicatas. Por fim, obtiveram-se 26.679 contratos únicos em mais de 14 milhões de linhas de código.

Essa etapa resultou no corte de 93% do *dataset* inicial. Apesar do corte expressivo, essa etapa possibilitou a redução do escopo para a realização do estudo empírico. A tabela 7.1 resume a quantidade de contratos considerados em cada etapa do processo de construção e mostra uma comparação entre os *datasets* SB^{2022} construído neste trabalho e SB^{WILD} proposto por Durieux et al. (2020).

7.1.1 Discussão e comparação

É importante destacar que neste trabalho, o corte de contratos sem saldo foi feito antes da identificação de duplicatas. Isso explica a diferença notável entre o número de contratos com código fonte disponível. Além disso, a blockchain Ethereum passou a ser utilizada de forma extensiva para a venda e comercialização de *tokens* fungíveis (ERC20) e não fungíveis (NFTs, ERC721). É possível que um grande número de contratos e contas externas (EOA) detenham

[1] Repositório do *dataset* SB^{2022}: `https://github.com/yudikubota/smartbugs-wild-2022`

Tabela 7.1: Comparação entre estatísticas de coleção de contratos inteligentes da Ethereum

Métrica	**SB^{2022}**	Durieux et al. (2020)
Total de contratos na Ethereum	56.386.406	17.503.263
Contratos com pelo menos uma transação	7.029.505	2.263.096
Contratos sem saldo	6.574.880	867.427
Contratos com saldo	454.625	1.395.669
Código-fonte não disponível	156.627	1.290.074
Código-fonte disponível	297.998	972.975
Código-fonte não acessível ou inválido	1.446	47
Duplicatas	269.873	925.457
Contratos únicos	26.679	47.518
Linhas de código (cloc)	14.757.303	9.693.457

ativos de valor na forma de *tokens* e não somente em ETH. Ambos os estudos consideram que o saldo é composto apenas por ETH.

Uma métrica de comparação com resultado peculiar é o número de linhas de código em cada *dataset*. Observa-se que apesar de conter menos contratos, o *dataset* SB^{2022} contém mais linhas de código. Um possível motivo para esse resultado é o fato de que a linguagem solidity passou a permitir o *upload* de mais de um contrato por endereço, possibilitando a implementação de contratos mais complexos, isto é, com mais linhas de código. De fato, os contratos do dataset atual têm, em média, 553 linhas de código, ao passo que o dataset SB$^{\text{WILD}}$ tem uma média de 204 linhas de código por contrato.

Outras métricas seguem resultados semelhantes ao estudo de referência. Em ambos, o número de duplicatas continua próximo a 90%. Já a porcentagem de contratos com código-fonte disponíveis no Etherscan teve um aumento, era de 43% e passou para 65%.

7.1.2 Métricas sobre o dataset

A fim de entender o impacto das atualizações da linguagem solidity no *dataset* e nos resultados, foi feita a extração das versões utilizadas nos contratos. A figura 7.1 mostra a distribuição das versões solitiy presentes no *dataset* SB^{2022}. A parte esquerda refere-se às versões em ordem cronológica, enquanto a parte direita refere-se às versões em ordem de quantidade. É possível perceber que grande parte do *dataset* é compatível com versões mais atuais da linguagem. Isso limita a efetividade de ferramentas que não oferecem suporte às versões mais recentes da linguagem, como as ferramentas Securify e Smartcheck.

Ao analisar as duplicatas, notou-se que grande parte se deve à implementação de contratos *proxy*. Esses contratos encaminham todas as chamadas para outro contrato já existente, agindo na prática como uma cópia da implementação. Esse padrão foi proposto pelo EIP-1167 [2] para permitir a clonagem simples e barata de implementações já existentes na blockchain. A tabela 7.2 apresenta os endereços dos 5 contratos mais duplicados, todos eles são *proxy*.

[2] `https://eips.ethereum.org/EIPS/eip-1167`

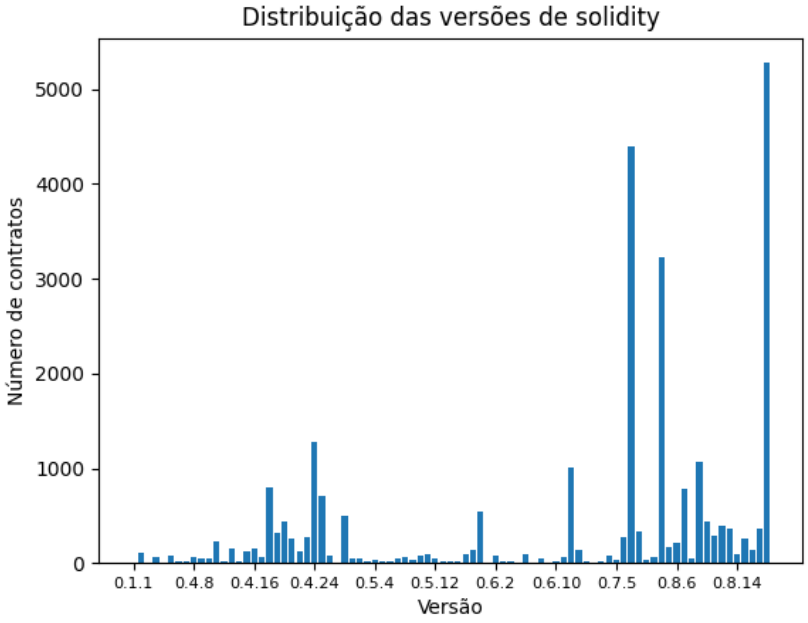

Figura 7.1: Distribuição das versões no dataset SB[2022].

Tabela 7.2: Endereços dos 5 contratos mais duplicados

#	**Endereço do contrato**	**Número de duplicatas**
1	0x3e4a5476621eb336cdbce827207202920169c538	1572
2	0xc4fe2f245724d02550b724906f03bf6af1c2ee8f	538
3	0xb5a0f8f5eb5f27ed44901279642798a2af2d4241	290
4	0x00bb11cc72b33ef257d2651cb3a6ac64d0fc535b	205
5	0x1db92e2eebc8e0c075a02bea49a2935bcd2dfcf4	158

7.2 RESULTADOS DAS ANÁLISES

Nesta sessão apresentaremos os resultados das análises de vulnerabilidades obtidas sobre o *dataset* SB[2022].

7.2.1 Tempo de execução

A primeira métrica a ser considerada foi o tempo de execução das ferramentas quando expostas a contratos presentes em ambiente de produção. No total, foram performadas 186.786 análises em um tempo de execução total de aproximadamente 340 dias e 20 horas, executadas paralelamente em 31 núcleos de processamento. Essa soma não inclui o tempo de pré-processamento e de compilação de resultados. O tempo médio de execução de cada análise considerando todas as ferramentas ficou em 2 minutos e 35 segundos, inferior ao resultado obtido por Durieux et al. (2020) de 4 minutos e 31 segundos provavelmente devido a remoção da ferramenta Manticore. A ferramenta com maior tempo médio foi Mythril com 6 minutos e 56 segundos e a de menor tempo médio foi Slither com apenas 16 segundos.

A ferramenta Sfuzz não foi considerada nos resultados, pois a saída produzida por essa ferramenta ocupou todo o espaço em disco restante das máquinas utilizadas para processamento. Isto é, os 100 GB previstos inicialmente não foram suficientes para comportar os resultados da execução de todas as ferramentas.

Também pôde-se notar que algumas ferramentas que não tiveram *timeout* durante sua execução no artigo de referência passaram a ter neste experimento. As ferramentas Confuzzius, Conkas, Maian e Mythril encontraram pelo menos uma análise interrompida por *timeout*. Isso se deve provavelmente ao aumento de número médio de linhas de código por contrato no *dataset*. Além disso, com intuito de viabilizar a análise, dado o tempo disponível, o tempo limite definido para cada análise foi configurado para 10 minutos, inferior aos 30 minutos do artigo de referência. A tabela 7.3 expõe os tempos médio e totais de cada ferramenta, enquanto que a tabela 7.4 expõe os mesmos dados sobre o artigo de referência no *dataset* SB^{WILD}.

Tabela 7.3: Tempo de execução médio para cada ferramenta

#	Ferramentas	Tempo de execução		Timeout
		Médio	Total	
0	confuzzius	0:03:21	62 dias, 7:36:38	sim
1	conkas	0:00:43	13 dias, 12:28:56	sim
2	maian	0:06:33	122 dias, 0:26:48	sim
3	mythril	0:06:56	126 dias, 6:34:56	sim
4	oyente	0:00:20	6 dias, 6:15:22	não
5	slither	0:00:16	4 dias, 23:09:35	não
6	solhint	0:00:17	5 dias, 11:27:06	não
Total		0:02:37	340 dias, 19:59:24	

Tabela 7.4: Tempo de execução médio para cada ferramenta (adaptado de Durieux et al. (2020))

#	Ferramentas	Tempo de execução		timeout
		Médio	Total	
0	Honeybadger	0:01:38	23 dias, 13:40:00	não
1	Maian	0:05:16	49 dias, 10:06:15	não
2	Manticore	0:24:28	184 dias, 01:59:02	sim
3	Mythril	0:01:24	46 dias, 07:46:55	não
4	Osiris	0:00:34	18 dias, 10:19:01	não
5	Oyente	0:00:30	16 dias, 04:50:11	não
6	Securify	0:06:37	217 dias, 22:46:26	não
7	Slither	0:00:05	2 dias, 15:09:36	não
8	Smartcheck	0:00:10	5 dias, 12:33:14	não
Total		0:04:31	564 dias, 03:10:39	

Todas as métricas da tabela 7.3 consideram o tempo de execução tanto para casos bem sucedidos como para falhas. Além do *timeout*, as análises encontraram outros problemas, como a falhas na compilação do código-fonte, falhas na execução da ferramenta e falhas na leitura de resultados de saída, resultando em 109.911 análises que ocorreram sem nenhuma

falha. É importante destacar que apesar de ocorrerem falhas durante a execução das análises, algumas ferramentas foram capazes de acusar vulnerabilidades antes da saída do programa. Essas vulnerabilidades foram consideradas válidas.

7.2.2 Vulnerabilidades

No tocante a vulnerabilidades, no total, foram encontradas 161.138 instâncias de 110 tipos de vulnerabilidades diferentes. A tabela A.1 presente no apêndice A especifica as vulnerabilidades que cada uma das ferramentas utilizadas foram capazes de detectar e o seu respectivo mapeamento para a taxonomia DASP. 27 itens foram ignorados por não se caracterizarem como vulnerabilidades, mas sim como recomendações de implementação e itens informacionais.

Um resultado significativo obtido do estudo empírico de Durieux et al. (2020) foi que 97% dos contratos do *dataset* SB^{WILD} foram acusados como vulneráveis. Nesse estudo, as ferramentas Oyente, Smartcheck e Mythril acusaram que 73%, 52% e 48% do dataset era vulnerável, respectivamente. No presente estudo, essas porcentagens caíram consideravelmente, com a ferramenta Mythril acusando 15% do dataset, enquanto que a ferramenta Oyente acusa 17%. A ferramenta Smartcheck não foi incluída, pois não oferece suporte às versões mais recentes da linguagem Solidity (consultar tabela 6.1). Ao considerar todas as ferramentas, exceto solhint, obtivemos que 36% do *dataset* é acusado de ter pelo menos uma vulnerabilidade. A tabela 7.5 mostra a quantidade de contratos com ao menos uma vulnerabilidade de cada tipo e detectada por cada ferramenta.

—

Tabela 7.5: Número total de contratos com pelo menos uma vulnerabilidade

Categoria	**confuzzius**	**conkas**	**maian**	**mythril**	**oyente**	**slither**	**Total**
DASP-1	289 1%	2.958 11%	0 0%	1.223 4%	117 0%	2.128 7%	4.668 17%
DASP-2	2 0%	0 0%	2.289 8%	402 1%	0 0%	1.481 5%	3.974 14%
DASP-3	1.316 4%	3.264 12%	0 0%	1.203 4%	4.595 17%	0 0%	5.932 22%
DASP-4	425 1%	298 1%	0 0%	131 0%	0 0%	1.002 3%	1.555 5%
DASP-5	14 0%	0 0%	0 0%	841 3%	292 1%	792 2%	1.755 6%
DASP-6	1.380 5%	0 0%	0 0%	250 0%	0 0%	0 0%	1.466 5%
DASP-7	886 3%	930 3%	0 0%	0 0%	1.212 4%	0 0%	2.331 8%
DASP-8	0 0%	1.328 4%	0 0%	1.361 5%	356 1%	712 2%	2.436 9%
DASP-9	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%	0 0%
DASP-10	2.204 8%	0 0%	0 0%	2.222 8%	0 0%	4.267 15%	5.691 21%
Total	3.592 13%	4.168 15%	2.289 8%	4.266 15%	4.772 17%	5.227 19%	9.739 36%

A ferramenta Solhint foi desconsiderada na tabela 7.5, pois acusava que 99% dos contratos eram vulneráveis, sugerindo um grande número de falsos positivos e definições de vulnerabilidades muito amplas. Outro ponto é que essa ferramenta foi desenvolvida inicialmente como um *linter*, ou seja, não tinha como principal objetivo a detecção de vulnerabilidades. A

tabela 7.6 mostra as categorias mais encontradas pela ferramenta Solhint. Nota-se que a categoria coringa DASP-10 *Unknown Unknowns* é a mais proeminente com 98%, seguida da categoria DASP-2 *Access Control* com 88%.

Tabela 7.6: Número de contratos com pelo menos uma vulnerabilidade segundo a ferramenta Solhint

Categoria	**solhint**
DASP-1	2.888 10%
DASP-2	23.666 88%
DASP-3	0 0%
DASP-4	428 1%
DASP-5	309 1%
DASP-6	0 0%
DASP-7	0 0%
DASP-8	16.914 63%
DASP-9	0 0%
DASP-10	26.254 98%
Total	26.538 99%

Também foi observado que um tempo de execução maior empreendido pela ferramenta não significa uma maior quantidade de contratos acusados como vulneráveis. Esse resultado corrobora com o que foi apontado por Durieux et al. (2020). Veja a figura 7.2.

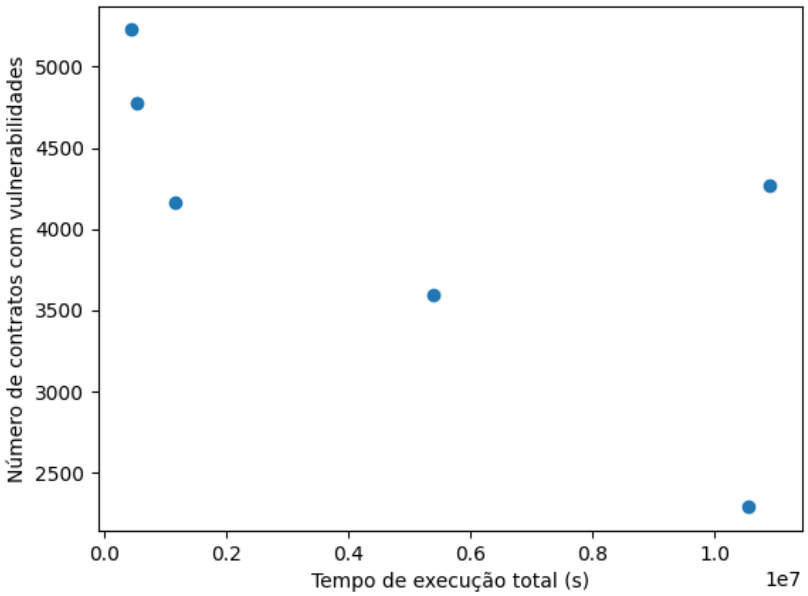

Figura 7.2: Dispersão da quantidade de contratos com vulnerabilidades em relação ao tempo de execução

7.3 RISCOS E LIMITAÇÕES DO ESTUDO

Este estudo compartilha algumas limitações e riscos com o estudo em que foi baseado. Assim como no estudo empírico de Durieux et al. (2020), o mapeamento de vulnerabilidades foi

feito de maneira manual, ou seja, está exposto ao risco de mapeamento incorreto. Para mitigar esse risco, foram utilizadas as documentações oficiais, artigos e códigos-fonte das ferramentas consideradas no estudo. Além disso, a taxonomia consolidada proposta por Rameder et al. (2022) foi usada para encontrar correspondentes entre as taxonomias SWC e DASP.

Um outro risco é a possibilidade da presença de falhas na implementação do *framework* de execução *Smartbugs* e nos scripts de coleta, pré-processamento e compilação de resultados. Esse risco foi parcialmente mitigado ao coletar informações sobre contratos inválidos e análises incompletas.

Uma limitação é que o desenho do estudo não permite a identificação do motivo da redução da porcentagem de contratos considerados vulneráveis pelas ferramentas, o que abre espaço para novas hipóteses e estudos futuros.

O *dataset* foi construído com base em critérios que excluem contratos que não possuem saldo em ETH. Apesar disso, essa não é a única forma de armazenar valor financeiro na blockchain Ethereum. A existência de *tokens* fungíveis (ERC-20) e não fungíveis (NFTs, ERC-721) sugere a possibilidade de armazenamento de valor financeiro significativo em contratos que não foram incluídos no *dataset* e por consequência na análise final.

7.4 RESUMO E DISCUSSÃO DOS RESULTADOS

Este capítulo mostrou os resultados obtidos a partir da construção do *dataset* SB^{2022} e subsequente análise do mesmo por 7 ferramentas de análise. O novo *dataset* apresenta uma quantidade menor de contratos, mas mais linhas de código, o que sugere uma maior complexidade dos códigos-fonte em comparação ao dataset SB^{WILD}. Outras métricas vão de acordo com o resultado obtido pelo artigo de referência, como a alta incidência de duplicatas e de contratos sem saldo em ETH. Os resultados das análises mostram uma redução significativa na porcentagem contratos acusados como vulneráveis ao desconsiderar a ferramenta Solhint. O desenho do estudo não permite atestar o real motivo dessa redução, porém a redução de falsos-positivos por parte das ferramentas e a melhor robustez de código perante vulnerabilidades por parte de esforços de desenvolvedores são hipóteses possíveis. No tocante a falsos-positivos, foi mostrado que a quantidade de contratos considerados vulneráveis pela ferramenta Solhint não é condizente com os resultados das outras ferramentas, mesmo quando consideradas em conjunto, o que pode indicar um número alto de falsos-positivos ou regras muito amplas e pouco representativas. Este estudo também confirma a observação de que um maior tempo de execução não indica maior quantidade de contratos acusados como vulneráveis.

8 CONCLUSÃO

Ataques a contratos inteligentes na blockchain Ethereum podem resultar em prejuízos financeiros significativos, o que destaca a necessidade de prevenção por meio da detecção prévia de vulnerabilidades. Embora a segurança em *smart contracts* muitas vezes seja responsabilidade dos desenvolvedores, ferramentas automatizadas de análise também são essenciais para identificar vulnerabilidades e melhorar a segurança em blockchain.

O presente estudo propôs a construção do *dataset* atualizado SB^{2022} e a execução de análises sobre os contratos desse *dataset* com o objetivo de reportar sobre os aspectos das ferramentas selecionadas neste estudo e comparar seus resultados com estudos anteriores. A construção da nova coleção de contratos mostra métricas condizentes com o estudo de referência, como a alta incidência de duplicatas e de contratos sem saldo. Os resultados das análises mostraram que o número de vulnerabilidades encontradas pelas ferramentas caiu em termos percentuais e que a ferramenta Solhint não condiz com os resultados das outras ferramentas.

Desta forma, a presente monografia almeja suportar e melhorar o entendimento sobre segurança e ferramentas e análise de vulnerabilidades para contratos inteligentes da Ethereum.

8.1 TRABALHOS FUTUROS

O presente estudo evidencia mudanças no campo de ferramentas de análise para contratos inteligentes da Ethereum em comparação a estudos anteriores. Todavia, o campo ainda carece de estudos e existem lacunas no entendimento dos resultados.

Um possível trabalho futuro seria a ampliação do dataset SB^{2022} para conter contratos que contenham valores significativos em outros *tokens* e não somente em ETH. Tal esforço poderia evidenciar ainda mais contratos vulneráveis e assim prevenir prejuízos financeiros futuros, além de reportar métricas sobre um *dataset* ainda mais representativo das aplicações em produção na Ethereum.

Outra lacuna presente é entender o motivo da redução da porcentagem de contratos acusados como vulneráveis. Essa redução pode ter sido causada pela melhor qualidade das ferramentas ou pela melhor qualidade do código-fonte do *dataset*. Entender o motivo dessa redução seria importante para garantir o esforço contínuo para a prevenção de ataques a contratos inteligentes.

Por fim, outra melhoria se aplica ao método de mapeamento de vulnerabilidade. Neste estudo e no estudo de referência esse mapeamento foi feito de maneira manual a critério dos autores. Um futuro mapeamento sistemático seria bem-vindo, pois eliminaria dúvidas sobre os critérios utilizados para a classificação na taxonomia escolhida.

REFERÊNCIAS

Asmundson, I. e Oner, C. (2012). What is money? *FINANCE & DEVELOPMENT*, 49(3).

Atzei, N., Bartoletti, M. e Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). Em Maffei, M. e Ryan, M., editores, *Principles of Security and Trust*, páginas 164–186, Berlin, Heidelberg. Springer Berlin Heidelberg.

Back, A. (2002). Hashcash - a denial of service counter-measure. `http://www.hashcash.org/papers/hashcash.pdf`. Acessado em 25/02/2023.

Buterin, V. (2014). Ethereum: A next-generation smart contract and decentralized application platform. `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf`. Acessado em 12/02/2023.

C., R. (2022). Bitcoin is the hardest money ever created. here's what that actually means. `https://medium.com/thepowerbrunch/bitcoin-is-the-hardest-money-ever-created-heres-what-that-actually-means-659f3ae64aad`. Acessado em 27/02/2023.

Certik (2023). Web3 security leaderboard. `https://www.certik.com/`. Acessado em 31/01/2023.

Chainlink (2023). Introduction to chainlink VRF | chainlink documentation. `https://docs.chain.link/vrf/v2/introduction`. Acessado em 28/01/2023.

Chohan, U. W. (2021). The double spending problem and cryptocurrencies. `https://ssrn.com/abstract=3090174`. Acessado em 27/02/2023.

Consensys (2023a). Reentrancy - Ethereum smart contract best practices. `https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/`. Acessado em 08/02/2023.

Consensys (2023b). Smart contract audits | consensys diligence. `https://consensys.net/diligence/`. Acessado em 31/01/2023.

Dai, W. (1998). B-money. `http://www.weidai.com/bmoney.txt`. Acessado em 25/02/2023.

Daian, P. (2016). Analysis of the DAO exploit. `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`. Acessado em 21/01/2023.

Day, A. e Medvedev, E. (2018). Ethereum in bigquery: a public dataset for smart contract analytics. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics. Acessado em 24/01/2023.

Docker (2023). Docker documentation: How to build, share, and run | docker documentation. https://docs.docker.com/. Acessado em 10/02/2023.

Dotsis, G. (2019). *IOU: Money, Banking and Cryptocurrencies*, páginas 1–34. Utopia Publishing.

Durieux, T., Ferreira, J. F., Abreu, R. e Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 ethereum smart contracts. Em *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, páginas 530–541.

Ethereum (2023a). Ethereum development documentation. https://ethereum.org/en/developers/docs/. Acessado em 27/02/2023.

Ethereum (2023b). Ethereum improvement proposals. https://eips.ethereum.org/. Acessado em 28/02/2023.

Ethereum (2023c). Ethereum whitepaper | ethereum.org. https://ethereum.org/en/whitepaper/. Acessado em 22/02/2023.

Etherscan (2023). Introduction - etherscan. https://docs.etherscan.io/. Acessado em 23/01/2023.

Ferreira, J. F., Cruz, P., Durieux, T. e Abreu, R. (2020). Smartbugs: A framework to analyze solidity smart contracts. Em *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, páginas 1349–1352.

Finney, H. (2023). RPOW theory. https://nakamotoinstitute.org/finney/rpow/theory.html. Acessado em 26/02/2023.

Ghaleb, A. e Pattabiraman, K. (2020). How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. Em *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Group, N. (2023). DASP - top 10. https://dasp.co/index.html. Acessado em 20/01/2023.

Guazzo, G. (2021). Why bitcoin is the hardest money in the world. https://medium.datadriveninvestor.com/why-bitcoin-is-the-hardest-money-in-the-world-8c0f6487122f. Acessado em 27/02/2023.

Hacken (2022). Reentrancy attacks and how to deal with them. https://hacken.io/discover/reentrancy-attacks/. Acessado em 08/02/2023.

Hacken (2023). Smart contract security audit - hacken. https://hacken.io/services/blockchain-security/smart-contract-security-audit/. Acessado em 31/01/2023.

jhimlic1 (2022). What was the DAO hack? - geeksforgeeks. https://www.geeksforgeeks.org/what-was-the-dao-hack/. Acessado em 21/01/2023.

Jimenez Freitez, W. R., Mammar, A. e Cavalli, A. R. (2009). Software vulnerabilities, prevention and detection methods : a review. Em *SEC-MDA 2009 : Security in Model Driven Architecture* , páginas 1 – 11, Enschede, Netherlands.

Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M. e Lee, H.-N. (2022). Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062.

Lamport, L., Shostak, R. e Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

Mankiw, G. (2021). *Introdução à Economia – Tradução da 8ª edição norte-americana*, páginas 1–72, 484–499. Cengage Learning Edições Ltda., 4ª edition.

May, T. C. (1992). The crypto anarchist manifesto. https://activism.net/cypherpunk/crypto-anarchy.html. Acessado em 25/02/2023.

McLeay, M., Radia, A. e Thomas, R. (2014). Money in the modern economy: an introduction. https://www.bankofengland.co.uk/quarterly-bulletin/2014/q1/money-in-the-modern-economy-an-introduction. Acessado em 02/02/2023.

Mitre, C. (2023). CWE - common weakness enumeration. https://cwe.mitre.org/index.html. Acessado em 20/01/2023.

Nakamoto, S. (2023). Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf. Acessado em 12/02/2023.

Olickel, H. (2016). Why smart contracts fail: Undiscovered bugs and what we can do about them. https://hrishiolickel.medium.com/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007. Acessado em 28/01/2023.

Rameder, H., di Angelo, M. e Salzer, G. (2022). Review of automated vulnerability analysis of smart contracts on ethereum. *Front. Blockchain*, 5(814977).

Reutov, A. (2018). Predicting random numbers in ethereum smart contracts. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620. Acessado em 28/01/2023.

Smartbugs (2020). This repository contains 47,398 smart contracts extracted from the ethereum network. https://github.com/smartbugs/smartbugs-wild. Acessado em 23/01/2023.

SWC (2023). Overview · smart contract weakness classification and test cases. https://swcregistry.io/. Acessado em 20/01/2023.

Tani, T. (2018). Ethereum evm illustrated. https://github.com/takenobu-hs/ethereum-evm-illustrated. Acessado em 27/02/2023.

Torres, C., Iannillo, A., Gervais, A. e State, R. (2021). Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. Em *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, páginas 103–119. IEEE Computer Society.

Veloso, N. (2021). Conkas: A modular and static analysis tool for ethereum bytecode. https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997262417/94080-Nuno-Veloso_resumo.pdf. Acessado em 28/02/2023.

Wood, G. (2022). Ethereum: A secure decentralised generalised transaction ledger - berlin version beacfbd. https://github.com/ethereum/yellowpaper. Acessado em 08/02/2023.

APÊNDICE A – MAPEAMENTO DE VULNERABILIDADES

A tabela a seguir mostra os nomes de vulnerabilidades encontradas por cada ferramenta e a classificação adotada neste trabalho usando as taxonomias SWC e DASP. A coluna "Ignorada"indica itens que foram reportados pelas ferramentas mas não se caracterizam como vulnerabilidades e sim como otimizações de código ou sugestões de alterações para comportar padrões de projeto e boas práticas.

Tabela A.1: Mapeamento de vulnerabilidades

Ferramenta	**Nome da vulnerabilidade**	**SWC**	**DASP**	**Ignorada**	**Severidade**
confuzzius	Arbitrary-Memory-Access	124	10		Alta
confuzzius	Assertion-Failure	110	10		Média
confuzzius	Block-Dependency	120	6		Baixa
confuzzius	Integer-Overflow	101	3		Alta
confuzzius	Leaking-Ether	105	10		Alta
confuzzius	Locking-Ether	132	10		Média
confuzzius	Reentrancy	107	1		Alta
confuzzius	Transaction-Order-Dependency	114	7		Média
confuzzius	Unhandled-Exception	104	4		Média
confuzzius	Unprotected-Selfdestruct	106	5		Alta
confuzzius	Unsafe-Delegatecall	112	2		Alta
conkas	Integer-Overflow		3		Alta
conkas	Integer-Underflow		3		Alta
conkas	Reentrancy		1		Alta
conkas	Time-Manipulation		8		Baixa
conkas	Transaction-Ordering-Dependence		7		Baixa
conkas	Unchecked-Low-Level-Call		4		Média
maian	Destructible-verified	106	2		
maian	Destructible	106	2		
maian	Ether-leak-verified		2		
maian	Ether-leak		2		
maian	Ether-lock-Ether-accepted-without-send		2		
maian	Ether-lock		2		
maian	No-Ether-leak-no-send			Sim	Informativa

maian	No-Ether-lock-Ether-refused			Sim	Informativa
maian	Not-destructible-no-self-destruct			Sim	Informativa
mythril	Delegatecall-to-user-supplied-address-SWC-112	112	2		
mythril	Dependence-on-predictable-environment-variable-SWC-116	116	8		
mythril	Dependence-on-predictable-environment-variable-SWC-120	120	6		
mythril	Dependence-on-tx-origin-SWC-115	115	2		
mythril	Exception-State-SWC-110	110	10		
mythril	External-Call-To-User-Supplied-Address-SWC-107	107	1		
mythril	Integer-Arithmetic-Bugs-SWC-101	101	3		
mythril	Jump-to-an-arbitrary-instruction-SWC-127	127	10		
mythril	Multiple-Calls-in-a-Single-Transaction-SWC-113	113	5		
mythril	State-access-after-external-call-SWC-107	107	1		
mythril	Unchecked-return-value-from-external-call-SWC-104	104	4		
mythril	Unprotected-Ether-Withdrawal-SWC-105	105	10		
mythril	Unprotected-Selfdestruct-SWC-106	106	5		
mythril	Write-to-an-arbitrary-storage-location-SWC-124	124	10		
oyente	Callstack-Depth-Attack-Vulnerability		5		
oyente	Integer-Overflow	101	3		
oyente	Integer-Underflow	101	3		
oyente	Parity-Multisig-Bug-2			Sim	
oyente	Re-Entrancy-Vulnerability	107	1		
oyente	Timestamp-Dependency		8		

oyente	Transaction-Ordering-Dependence-TOD	114	7		
slither	arbitrary-send		2		Alta
slither	assembly			Sim	Informativa
slither	calls-loop		5		Baixa
slither	constable-states			Sim	Informativa
slither	constant-function	110	10		
slither	controlled-delegatecall	112	10		
slither	deprecated-standards			Sim	Informativa
slither	erc20-indexed		10		Média
slither	erc20-interface		10		Média
slither	external-function			Sim	Informativa
slither	incorrect-equality	132	10		Média
slither	locked-ether		2		
slither	low-level-calls		4		
slither	naming-convention			Sim	Informativa
slither	pragma			Sim	Informativa
slither	reentrancy-benign	107	1		Baixa
slither	reentrancy-eth	107	1		Alta
slither	reentrancy-no-eth	107	1		Média
slither	shadowing-abstract	119	10		Média
slither	shadowing-builtin	119	10		Média
slither	shadowing-local	119	10		Média
slither	shadowing-state	119	10		Média
slither	solc-version			Sim	Informativa
slither	suicidal	106	2		Alta
slither	timestamp		8		Baixa
slither	tx-origin	115	2		
slither	uninitialized-local	109	10		Alta
slither	uninitialized-state	109	10		Alta
slither	uninitialized-storage	109	10		Alta
slither	unused-return	131	10		Média
slither	unused-state			Sim	Informativa
solhint	avoid-call-value	111	10		
solhint	avoid-low-level-calls	127	10		
solhint	avoid-sha3	111	10		
solhint	avoid-suicide	111	10		

solhint	avoid-throw	111	10		
solhint	avoid-tx-origin	115	2		
solhint	check-send-result	104	4		
solhint	compiler-version	102	10		
solhint	const-name-snakecase			Sim	Informativa
solhint	contract-name-camelcase			Sim	Informativa
solhint	event-name-camelcase			Sim	Informativa
solhint	func-name-mixedcase			Sim	Informativa
solhint	func-visibility	100	2		
solhint	imports-on-top			Sim	Informativa
solhint	max-states-count			Sim	Informativa
solhint	multiple-sends		10		
solhint	no-complex-fallback		10		
solhint	no-empty-blocks	135	10	Sim	Informativa
solhint	no-inline-assembly		10		
solhint	no-unused-vars	131	10	Sim	Informativa
solhint	not-rely-on-block-hash		10		
solhint	not-rely-on-time		8		
solhint	payable-fallback	100	5		
solhint	quotes			Sim	Informativa
solhint	reason-string			Sim	Informativa
solhint	reentrancy	107	1		
solhint	state-visibility	108	2		
solhint	use-forbidden-name			Sim	Informativa
solhint	var-name-mixedcase			Sim	Informativa
solhint	visibility-modifier-order			Sim	Informativa
solhint	indent			Sim	Informativa
solhint	max-line-length			Sim	Informativa