Towards the Evolution of Tools for Detecting Vulnerabilities in Smart Contracts: A Case Study of Mythril and Slither

Felipe Mello Fonseca[1], **Matheus dos Santos Moura**[1],
Pedro Henrique Gonzalez[2], **Diogo Silveira Mendonça**[1]

[1]Programa de Pós-Graduação em Ciência da Computação (PPCIC)
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET-RJ)
Rio de Janeiro, RJ, Brasil

[2]Programa de Pós-Graduação em Engenharia de Sistemas e Computação (PESC)
Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia (COPPE)
Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro, RJ, Brasil

{felipe.mello, matheus.moura}@aluno.cefet-rj.br

pegonzalez@cos.ufrj.br, diogo.mendonca@cefet-rj.br

Abstract. *Blockchain is an innovative technology applied in various areas such as finance, record management, electronic voting, and gaming. Transactions on the blockchain are often executed by smart contracts, which are considered critical in terms of security. There are several tools designed to automatically identify vulnerabilities in smart contracts; however, as previous studies have shown, the effectiveness of these tools is usually low. Therefore, identifying vulnerabilities in smart contracts through automation remains a significant challenge. In this study, we investigate the evolution of two important tools for detecting vulnerabilities in smart contracts: Mythril and Slither, evaluating their effectiveness in identifying vulnerabilities and whether they have evolved compared to earlier versions. To do this, we ran the tools with the latest versions on a set of 69 smart contracts previously analyzed in an earlier study. The results were compared with the already classified vulnerabilities and subjected to manual validation to assess their accuracy. The experiments show that Mythril demonstrated improvements in reducing false positives, while Slither improved the detection of Access Control-related flaws. However, the tools showed limitations in their evolution for certain types of vulnerabilities. These findings reinforce the ongoing need for the continuous improvement and evaluation of automated security analysis tools for smart contracts.*

Resumo. *A Blockchain é uma tecnologia inovadora aplicada em diversas áreas como finanças, gestão de registros, votação eletrônica e jogos. As transações em blockchain são frequentemente executadas por smart contracts, código considerado crítico em termos de segurança. Existem diversas ferramentas que se propõe a identificar vulnerabilidades de forma automatizada em smart contracts, contudo, conforme estudos anteriores mostram, a eficácia nesta tarefa é normalmente baixa. Desse modo, identificar vulnerabilidades de forma automatizada em smart contracts continua sendo um grande desafio. Neste estudo*

investigamos a evolução de duas importantes ferramentas para detecção de vulnerabilidades em smart contracts: Mythril e Slither, avaliando sua eficácia na identificação de vulnerabilidades e se evoluíram comparadas a uma versão anterior. Para isto, executamos as ferramentas com versões mais rescentes em um conjunto de 69 smart contracts previamente analisados em um estudo anterior. Os resultados foram comparados com as vulnerabilidades já classificadas e submetidos a uma validação manual para aferir sua precisão. Os experimentos demonstram que Mythril apresentou melhorias na redução de falsos positivos, enquanto Slither aprimorou a detecção de falhas relacionadas a Access Control. Contudo, as ferramentas apresentaram limitações na sua evolução para alguns tipos de vulnerabilidades. Esses achados reforçam a necessidade contínua de aprimoramento e avaliação das ferramentas de análise automatizada de segurança em smart contracts.

1. Introdução

Uma *blockchain* é um livro-razão distribuído que gerencia ativos entre usuários [Maesa and Mori 2020]. A tecnologia conhecida como *blockchain* foi apresentada pela primeira vez por Satoshi Nakamoto [Nakamoto 2008], cujo objetivo principal foi resolver o problema de estabelecer confiança em um sistema distribuído. Mais especificamente, trata-se de um desafio relacionado à criação de um sistema de armazenamento de documentos distribuído, capaz de manter um registro confiável das datas e horários de cada documento, assegurando que nenhuma entidade possa modificar o conteúdo dos dados ou os registros temporais sem que tal atividade seja detectada [Di Pierro 2017].

A *blockchain* ganhou notoriedade principalmente no contexto das criptomoedas e no setor bancário [Guo and Liang 2016]. No entanto, sua aplicação não se restringe a esses campos; ela pode ser utilizado em qualquer sistema industrial que requeira tomada de decisão descentralizada, robusta e confiável [Belotti et al. 2019]. Na *blockchain*, utilizam-se os *smart contracts*, programas que codificam as regras para a transferência de ativos. As transferências ocorrem dentro de transações que são armazenadas de maneira persistente. Uma das plataformas mais populares que suporta *smart contracts* é o Ethereum [Buterin et al. 2014], cuja linguagem principal, Solidity, é influenciada pelo JavaScript [Eshghie et al. 2021]. Para a execução de operações em um *smart contract*, é necessária uma taxa denominada *gas*. Esses *smart contracts* podem implementar uma ampla gama de casos de uso, incluindo aplicações financeiras, governança, votação eletrônica, jogos e Internet das Coisas [Maesa and Mori 2020, Singh et al. 2021].

Os contratos inteligentes manipulam criptomoedas que somam bilhões de dólares, tornando-os alvos atrativos para invasores [Qian et al. 2020]. Portanto, é crucial que sejam previamente verificados quanto a potenciais vulnerabilidades. Em 2016, por exemplo, um ataque ao DAO (*Decentralised Autonomous Organisation*) resultou na perda de 60 milhões de dólares devido a falhas nos *smart contracts* [Liao et al. 2019]. Neste contexto, a detecção de vulnerabilidades nesses contratos é um desafio crescente, e as ferramentas de análise automatizada desempenham um papel essencial ao permitir a identificação de falhas sem a necessidade de execução dos contratos. No entanto, a eficácia dessas ferramentas varia significativamente, e novas ameaças continuam surgindo à medida que a tecnologia *blockchain* evolui [Bhushan et al. 2021].

Diante desse cenário, este estudo investiga a evolução da precisão e das limitações de duas das principais ferramentas de análise automatizada de código: Mythril e Slither. O objetivo é avaliar sua capacidade de detecção de vulnerabilidades em *smart contracts* escritos em Solidity. Para isso, foi conduzido um experimento controlado utilizando um conjunto de 69 *smart contracts* previamente analisados no estudo SmartBugs [Ferreira et al. 2020]. As ferramentas foram executadas nesses contratos e seus resultados foram comparados com classificações de vulnerabilidades de referência. Além disso, foi realizada uma validação manual para avaliar a precisão das detecções.

Os resultados indicam que a versão mais recente do Mythril reduziu significativamente os falsos positivos, enquanto o Slither apresentou melhorias na detecção de falhas de controle de acesso, mas ainda demonstra limitações na identificação de vulnerabilidades relacionadas à manipulação de tempo. Esses achados reforçam a necessidade contínua de aprimoramento das técnicas de análise automatizada e sugerem que a combinação de múltiplas ferramentas pode aumentar a eficácia na detecção de falhas em *smart contracts*.

O restante deste artigo está estruturado da seguinte forma: a Seção 2 apresenta uma revisão dos trabalhos relacionados ao tema, destacando as principais abordagens e limitações das ferramentas de análise de vulnerabilidades. A Seção 3 fornece a fundamentação teórica, abordando conceitos essenciais sobre *blockchain*, *smart contracts* e análise automatizada de código. A Seção 4 detalha a metodologia adotada para a condução do experimento. Os resultados obtidos são discutidos na Seção 5, seguidos das conclusões na Seção 6.

2. Trabalhos Relacionados

O artigo de Ye et al. [Ye et al. 2019] analisou ferramentas de detecção de bugs em *smart contracts*, avaliando seus prós e contras. Utilizando 1.010 *smart contracts* extraídos do Etherscan, o estudo examinou os mecanismos de detecção das ferramentas Slither e Oyente. Os resultados indicaram que ambas identificam problemas de *reentrancy*, porém o Slither detectou 24 arquivos com o bug, enquanto o Oyente encontrou apenas 8. Para *integer over/underflow*, o Oyente detectou 546 casos, enquanto o Slither não suportou essa detecção. Concluiu-se que o Slither apresenta altas taxas de falsos positivos para *reentrancy*, e que a falta de verificação do modificador de propriedade torna os resultados do Oyente não confiáveis para essa vulnerabilidade.

O trabalho de [Durieux et al. 2020] tem como objetivo relatar o estado da arte das ferramentas de análise automatizada para *smart contracts*, utilizando a estrutura extensível SmartBugs para facilitar a reprodutibilidade e a comparação. O estudo analisou 9 ferramentas, selecionadas entre 35, com base em critérios como disponibilidade e aceitação de entradas em Solidity. Foram criados dois conjuntos de dados: um curado, com 69 contratos, e outro com 47.518 contratos extraídos da *blockchain* Ethereum. O Mythril se destacou pela precisão, detectando 27% das vulnerabilidades, enquanto a combinação de Mythril e Slither identificou 37% das vulnerabilidades únicas. O estudo concluiu que as ferramentas analisadas conseguem detectar 42% das vulnerabilidades, e a combinação de Mythril e Slither a mais eficaz.

O estudo de Zheng et al. [Zheng et al. 2023] analisou a eficácia de cinco ferramentas — Oyente, Mythril, Securify, Smartian e Sailfish — na detecção de *reentrancy* em

smart contracts. Foram coletados 230.548 contratos do Etherscan, filtrando-se 139.424 contratos únicos. A análise identificou 21.212 contratos com *reentrancy*, mas a validação manual revelou que 99,8% eram falsos positivos. Os achados indicam que as ferramentas atuais têm baixo desempenho e não detectam ataques recentes. O trabalho notou que as ferramentas falham em detectar problemas de *reentrancy* nos contratos recentemente atacados. Com base nos resultados, o trabalho concluiu que os estudos existentes na detecção de *reentrancy* têm baixo desempenho e podem estar desatualizados, e os pesquisadores devem direcionar sua atenção do *call.value()* para a descoberta e detecção de novos padrões de *reentrancy*.

Este trabalho segue a linha dos estudos referenciados, incluindo o uso dos *smart contracts* acurados do trabalho de et al. [Durieux et al. 2020] e de seus dados para compará-los com nossas análises. Além disso, identificamos a relevância de questões como o problema dos falsos positivos, que também fazem parte da nossa investigação. No entanto, nosso foco principal está na evolução da ferramenta, e não na comparação entre diferentes soluções.

3. Ferramentas de Análise de Segurança em Código Solidity

Nesta seção explicaremos os conceitos envolvidos e as ferramentas de análise automatizada de código fonte de *smart contracts* avaliadas neste trabalho.

3.1. Static Analysis

A *static code analysis* é uma abordagem eficaz para detectar vulnerabilidades sem a necessidade de executar o programa. Essas ferramentas identificam automaticamente erros de programação relacionados à segurança, oferecendo uma alternativa mais rápida e abrangente à revisão manual [Chess and McGraw 2004]. Diferente dos testes dinâmicos, que apenas expõem sintomas, a *static analysis* revela a causa raiz dos problemas de segurança, permitindo correções mais precisas [Chess and West 2007]. Além disso, possibilita a detecção precoce de falhas, reduzindo custos e fornecendo feedback imediato aos desenvolvedores, ao mesmo tempo em que facilita a reavaliação de grandes bases de código para novas ameaças, sendo particularmente útil na revisão de sistemas legados.

No entanto, essa abordagem apresenta desafios, como a alta incidência de falsos positivos, que podem comprometer sua eficiência. Se não forem gerenciados adequadamente, esses alertas excessivos podem resultar em desperdício de tempo e até na negligência de vulnerabilidades reais, minando a segurança do software [Chess and West 2007].

3.2. Symbolic Execution

A *symbolic execution* é uma técnica usada para analisar programas explorando múltiplos caminhos de execução simultaneamente et al. [Baldoni et al. 2018]. Em vez de usar valores concretos para entradas, usa variáveis simbólicas. Assim, gera-se expressões condicionais e as resolve usando *SMT solvers* para determinar valores que levam a falhas. Em vez de utilizar valores concretos, a análise simbólica usa variáveis simbólicas para representar entradas, permitindo identificar vulnerabilidades potenciais et al. [Tsankov et al. 2018].

A *symbolic execution* é usada principalmente para teste de falha de software e segurança. Os principais desafios dessa tecnologia estão relacionados ao que se chama de

path explosion [Baldoni et al. 2018]. Na *path explosion*, cada bifurcação no código duplica os caminhos explorados e o número de execuções cresce exponencialmente. Assim, resolver expressões simbólicas complexas pode ser computacionalmente caro, especialmente no contexto da *blockchain* [Tsankov et al. 2018]. Reduzir o número de caminhos cobertos para uma maior velocidade de execução pode gerar um maior aparecimento de falsos positivos [Baldoni et al. 2018].

3.3. Slither

O Slither [Feist et al. 2019] é um framework de análise estática para *smart contracts* no Ethereum, escrito em Python 3. Ele converte contratos Solidity em uma representação intermediária, SlithIR, baseada em Static Single Assignment (SSA), preservando informações semânticas essenciais. Isso permite a aplicação de técnicas como análise de fluxo de dados e rastreamento de contaminação.

O framework é atualmente usado para o seguinte:

- **Detecção automatizada de vulnerabilidades:** diversas falhas em *smart contracts* podem ser identificadas sem intervenção do usuário.
- **Detecção automatizada de otimizações de código:** o Slither encontra otimizações de código que o compilador não percebe.
- **Compreensão do código:** logs de resumo, que exibem informações dos contratos.
- **Revisão de código assistida:** o usuário pode interagir com o Slither por meio de sua API.

3.4. Mythril

Mythril é uma ferramenta de análise de segurança voltada para *smart contracts* da Ethereum. Desenvolvido pela ConsenSys, ela utiliza técnicas como *concolic analysis*, análise de contaminação e verificação do fluxo de controle do bytecode da EVM para reduzir o espaço de busca e identificar valores que possam explorar vulnerabilidades em *smart contracts* [Durieux et al. 2020]. O projeto é de código aberto, baseado em console e está em desenvolvimento ativo. Seu mecanismo de análise é capaz de detectar diversas classes de vulnerabilidades de segurança em *smart contracts*. Para isso, utiliza a máquina virtual simbólica LASER, um módulo próprio do Mythril, que realiza a execução simbólica do código [Prechtel et al. 2019].

A máquina virtual LASER é integrada ao Mythril, substituindo o bytecode Ethereum bruto por um *disassembly* gerado pelo Mythril. Esse *disassembly* mantém as mesmas instruções do bytecode nativo da Ethereum, mas oferece informações adicionais úteis para a análise. O *disassembly* do Mythril pode ser gerado a partir de fontes Solidity ou do próprio bytecode Ethereum [Prechtel et al. 2019]. O LASER é capaz de modelar a maioria das funcionalidades da máquina virtual Ethereum oficial, embora não consiga abordar algumas funcionalidades, como a modelagem do consumo de gás em contratos inteligentes.

4. Metodologia

O principal objetivo deste artigo é analisar duas das principais ferramentas de análise automatizada de *smart contracts*, identificadas no trabalho de [Di Angelo and Salzer 2019] e aprimoradas no estudo de [Durieux et al. 2020]. O foco está nas ferramentas Mythril

e Slither, que segundo [Durieux et al. 2020], são as melhores quando utilizadas em conjunto. Os dados apresentados por esse estudo continuam sendo relevantes para análises contemporâneas, pois se tratam de uma das pesquisas mais abrangentes sobre o tema, com um *dataset* de alta precisão. Além disso, o trabalho introduziu a solução SmartBugs [Ferreira et al. 2020], conforme detalhado nas referências mencionadas. Este estudo compara as análises geradas pelo SmartBugs com as versões mais recentes do Mythril e Slither.

Com o objetivo de examinar um grande conjunto de códigos em Solidity, a abordagem inicial consistiu na construção de scripts automatizados capazes de executar as ferramentas em todos os arquivos Solidity. Para garantir que o código fosse o mais genérico possível e facilitasse sua aplicação em outros repositórios, diversos testes foram realizados e etapas de refatoração implementadas.

Durante o desenvolvimento, foi constatado que o Mythril não oferece suporte ao sistema operacional Windows. Por esse motivo, as ferramentas e o desenvolvimento dos scripts de automação foram realizados em uma máquina com Linux Mint 21.2, equipada com um processador Core i5 de 8ª geração e 8 GB de RAM.

4.1. Etapas da Análise

No script automatizado, cada ferramenta é acionada por comandos de linha. O script ajusta automaticamente o compilador Solidity (solc) conforme o pragma do arquivo, garantindo a compilação mais próxima possível das especificações do contrato. O estudo de Ibba et al. [Ibba et al. 2024] propôs uma abordagem semelhante, utilizando um script em Python para detectar e trocar o solc conforme o pragma. Após a execução das ferramentas, é gerado um arquivo JSON com os resultados da análise.

Concluída a etapa de execução, inicia-se o processo de extração de dados. O código responsável por essa extração percorre os arquivos JSON gerados em busca de chaves que contenham informações sobre vulnerabilidades detectadas. A partir desses dados, constrói-se um dataframe onde os nomes das vulnerabilidades são utilizados como títulos das colunas, enquanto cada linha representa um arquivo Solidity analisado. Para cada arquivo, registra-se a quantidade de ocorrências de cada vulnerabilidade identificada.

Na sequência, as vulnerabilidades detectadas são mapeadas para a taxonomia Decentralized Application Security Project (DASP)[1]. Esse mapeamento é realizado por meio de um dicionário que associa as vulnerabilidades às respectivas categorias da DASP. Como diferentes vulnerabilidades podem pertencer à mesma categoria, os valores são consolidados, resultando em um novo dataframe que apresenta as categorias DASP e a contagem agregada de vulnerabilidades para cada arquivo analisado.

Por fim, os dados extraídos foram comparados com as análises realizadas pelo SmartBugs. O objetivo dessa comparação foi avaliar o volume de detecções das ferramentas. Além disso, realizou-se uma análise manual para verificar a precisão das detecções, garantindo que os problemas identificados correspondessem corretamente aos trechos de código afetados. Esses aspectos serão detalhados na seção de resultados.

O código automatizado do experimento[2] pode ser resumido nos seguintes passos:

[1] `https://dasp.co/`

[2] `https://github.com/drinith/analysis_mestrado`

1. Executar as ferramentas em cada arquivo Solidity, gerando como saída um arquivo JSON.
2. Extrair as vulnerabilidades da estrutura dos arquivos JSON.
3. Criar um dataframe com a contagem das vulnerabilidades.
4. Converter as vulnerabilidades do dataframe para as categorias DASP.
5. Criar um dataframe com a contagem consolidada de vulnerabilidades DASP.
6. Comparar cada vulnerabilidade encontrada com o gabarito criado a partir dos JSONs precisos no DASP.
7. Realizar uma análise manual e comparar os resultados obtidos.

Alguns pontos são importantes serem colocados para futuras análises e comparações com este trabalho. Nas primeiras análises, observou-se uma discrepância significativa no número de arquivos Solidity em relação à pesquisa do SmartBugs. Uma investigação mais profunda revelou que os pesquisadores haviam acrescentado mais códigos ao repositório para análise, além dos 69 apontados no trabalho base. Para a comparação, foi necessário procurar um local que disponibilizasse as informações das análises acuradas publicadas pelo SmartBugs. Após uma busca mais aprofundada no repositório do GitHub[3], foram encontrados os arquivos JSON gerados pelo SmartBugs. Assim, tornou-se possível separar os arquivos que foram realmente analisados.

Outro ponto notado na pesquisa é que o Mythril, na análise do SmartBugs, possui uma tabela de conversão[4] que apresenta vulnerabilidades diferentes das encontradas nas análises atuais. Na tabela do repositório do artigo no GitHub, encontramos as seguintes vulnerabilidades: *DELEGATECALL to a user-supplied address*, *Dependence on predictable environment variable*, *Dependence on predictable variable*, *Ether send*, *Exception state*, *Integer Overflow*, *Integer Underflow*, *Message call to external contract*, *Multiple Calls*, *State change after external call*, *Transaction order dependence*, *Unchecked CALL return value*, *Unchecked SUICIDE* e *Use of tx.origin*.

No entanto, o Mythril gera atualmente as seguintes vulnerabilidades como saída: *Write to Arbitrary Storage Location (SWC-124)*, *Integer Overflow and Underflow (SWC-101)*, *Timestamp Dependence (SWC-116)*, *Assert Violation (SWC-110)*, *Reentrancy (SWC-107)*, *DoS with Failed Call (SWC-113)*, *Unprotected Ether Withdrawal (SWC-105)*, *Delegatecall to Untrusted Callee (SWC-112)*, *Authorization through tx.origin (SWC-115)*, *Unchecked Call Return Value (SWC-104)*, *Weak Sources of Randomness from Chain Attributes (SWC-120)*, *Unprotected SELFDESTRUCT Instruction (SWC-106)* e *Transaction Order Dependence (SWC-114)*.

Assim, foi necessário criar uma tabela de conversão das vulnerabilidades, classificadas pela *Smart Contract Weakness Classification* (SWC)[5] para o DASP. As informações para a conversão foram encontradas em um arquivo do próprio SmartBugs, que acreditamos ter sido atualizado para essa saída[6].

A motivação do uso do DASP está no fato de que as saídas das ferramentas ap-

[3] `https://github.com/SmartBugs/SmartBugs-results/tree/master/results/Slither/curated`

[4] `https://github.com/SmartBugs/SmartBugs/wiki/Vulnerabilities-mapping`

[5] `https://swcregistry.io/`

[6] `https://github.com/SmartBugs/SmartBugs/blob/master/tools/Mythril-0.23.15/findings.yaml`

resentam vulnerabilidades com nomenclaturas distintas. Para viabilizar a comparação entre elas, foi realizada a conversão das vulnerabilidades para o padrão DASP. O dicionário utilizado para essa conversão baseia-se nas informações apresentadas anteriormente. Após a conversão, foi realizada uma nova contagem das vulnerabilidades, permitindo assim comparar os resultados desta análise com aqueles apresentados no artigo [Durieux et al. 2020]. Nem todas as vulnerabilidades das ferramentas possuem uma equivalência direta no DASP. Assim, para comparação como feito no trabalho base, comparamos mais a frente somente a vulnerabilidades que foram traduzidas. As outras vulnerabilidades não entram na comparação, pois os arquivos acurados também encontravam-se na classificação DASP. Desta forma, a classificação *Other* não será usada.

Outro aspecto importante a ser analisado no trabalho de [Durieux et al. 2020] é a ausência de uma especificação das versões das ferramentas Mythril e Slither utilizadas na época. Ao examinar o repositório do SmartBugs, identificamos que o commit referente à análise foi realizado em 24 de agosto de 2019. Com base nessa data, verificamos que as versões específicas do Mythril e do Slither correspondentes a esse commit são, respectivamente, 0.21.15 e 0.6.6. Assim, adotamos essas versões como referência para nossa análise.

Durante os testes de revisão do trabalho, após corrigirmos bugs no código das ferramentas para que fossem executadas nas versões mais atuais do Python, conseguimos rodar as ferramentas nas versões estimadas. Observou-se que a saída gerada era equivalente à do trabalho estudado. Portanto, é provável que os autores tenham utilizado versões próximas a essas.

4.1.1. Análise Manual

Embora as análises automáticas realizadas por meio do script em Python tenham sido úteis para proporcionar uma visão geral dos dados, tornou-se necessária a realização de uma análise manual ao longo do trabalho. Essa necessidade surgiu ao examinar os arquivos de saída das ferramentas, levantando a seguinte questão: o arquivo classificado como contendo uma determinada vulnerabilidade poderia apresentar essa mesma vulnerabilidade em outra parte do código?

Para verificar essa hipótese, foi realizada uma análise detalhada com o Slither (conforme os exemplos apresentados nos anexos a seguir). Observou-se que as vulnerabilidades nem sempre apareciam exatamente nos locais indicados pelas informações classificadas como precisas [7].

Um outro exemplo que demonstra a necessidade da análise manual foi disponibilizado on-line[8]. O levantamento aponta a necessidade da alteração dos valores detectados no trabalho do SmartBugs para a vulnerabilidade *Time Manipulation*

Com a análise manual, notou-se outra questão: algumas contagens nas análises anteriores não estavam realmente corretas. Assim, para este estudo, realizamos uma re-

[7] https://github.com/drinith/analysis_mestrado?tab=readme-ov-file#anexo-i

[8] https://github.com/drinith/analysis_mestrado?tab=readme-ov-file#anexo-ii

contagem das vulnerabilidades encontradas no trabalho anterior. Notamos que o comportamento do arquivo *FibonacciBalance.sol*, explicado no apêndice online [9] , acrescentou uma detecção adicional.

Uma segunda questão observada nas contagens manuais foi o aumento na quantidade de vulnerabilidades apontadas nos contratos Solidity. Identificamos mais duas vulnerabilidades, indicando, em tese, a existência de dois novos casos relacionados a *Access Control*, o que também explicaria a diminuição nos apontamentos de *Unchecked Low Level Calls*, já que esse arquivo estava incluído. Contudo, na contagem manual houve um aumento na análise do Mythril do SmarBugs para *Unchecked Low Level Calls*. E não foi encontrado um fato concreto que sustentasse essas diferenças , conforme indicado acima e nos anexos. Acredita-se que os autores, ao longo de seus estudos, refinaram as análises e identificaram novas informações sobre vulnerabilidades no código. No repositório do projeto, observamos que o commit mais antigo não coincide com a época da publicação do artigo; portanto, é plausível que tenham ocorrido alterações [10].

Dessa forma, tornou-se imprescindível realizar uma análise manual, revisando as saídas das ferramentas e inspecionando os trechos de código Solidity identificados como relevantes. Essa situação impactava a contagem automatizada, pois, embora o arquivo fosse classificado como contendo uma determinada vulnerabilidade e esta estivesse presente, ela não aparecia exatamente no ponto esperado. Isso exigiu que a contagem automatizada para a confirmação da detecção fosse desconsiderada.

Para fins de comparação, mesmo com a identificação de diferença nas vulnerabilidades apontadas, neste trabalho, foram realizadas novamente as análises e detecções nas análises anteriores. Assim, as análises manuais foram comparadas tanto com as saídas das análises anteriores quanto com as realizadas atualmente, permitindo, assim, uma base comparativa consistente. Desta forma, mesmo com mudanças nas contagens, a comparação é possível, já que os acertos aconteceram nas duas análises.

5. Resultados

As tabelas a seguir encontram-se também no repositório do trabalho [11], além de tabelas intermediárias que nos trouxeram ao resultado consolidado. As vulnerabilidades apontadas utilizam a nomenclatura padrão DASP.

A tabela 1 apresenta a quantidade de alertas de vulnerabilidades geradas pelas ferramentas em uma versão anterior [Durieux et al. 2020], e na versão avaliada neste trabalho. Dentre as diferenças observadas, destaca-se a vulnerabilidade aritmética. No estudo comparativo, a análise anterior realizada com a ferramenta Mythril detectou 92 ocorrências dessa vulnerabilidade, enquanto a análise deste trabalho identificou apenas 21. O trabalho comparado indica a presença de 22 vulnerabilidades aritméticas encontradas nos arquivos Solidity. Dessa forma, a análise do SmartBugs apresenta um número significativamente maior de falsos positivos em relação à análise atual.

Outra diferença notável diz respeito à vulnerabilidade de dependência de *times-*

[9] https://github.com/drinith/analysis_mestrado?tab=readme-ov-file#anexo-iii

[10] https://github.com/SmartBugs/SmartBugs-curated/tree/main/dataset

[11] https://github.com/drinith/analysis_mestrado

	Mythril<=0.21.15	Mythril 0.24.7	Slither<=0.6.6	Slither 0.10.0
Access Control	24	14	20	7
Arithmetic	92	21	0	0
Bad Randomness	0	4	0	0
Denial Service	0	2	2	1
Front Running	21	18	0	0
Reentrancy	16	24	15	16
Time Manipulation	0	7	5	23
Unchecked Low Calls	30	11	13	11
Other	32	38	28	31

Table 1. Número de alertas gerados pelas ferramentas em versão anterior [Durieux et al. 2020] e mais recente.

tamp. Na análise do trabalho anterior, não foram detectadas vulnerabilidades desse tipo com a ferramenta Mythril. No entanto, em nossa análise, identificamos uma vulnerabilidade verdadeira positiva no arquivo *roulette.sol*. Além disso, foram detectados, no total, sete casos desse tipo de vulnerabilidade. Esse ponto chamou a atenção, pois, conforme indicado pela tabela de tradução das vulnerabilidades e por algumas detecções realizadas, o Mythril passou a apontar a possibilidade de identificar as categorias *bad randomness* e *denial of service*, vulnerabilidades que não eram detectadas na versão anterior da ferramenta.

Seguindo a metodologia do trabalho de [Durieux et al. 2020], comparamos a quantidade de vulnerabilidades realmente encontradas com a quantidade indicada nos comentários dos códigos (*Recall*). Para isso, repetimos as análises do trabalho comparado e as nossas, cujos resultados estão sumarizados na Tabela 2.

Uma análise mais aprofundada do trabalho comparado revelou um possível aumento na quantidade de vulnerabilidades relacionadas ao controle de acesso. Ao examinar o repositório do artigo, identificamos uma possível explicação: o arquivo *FibonacciBalance.sol* foi movido da categoria *unchecked_low_level_calls* para a categoria *access_control* após a realização da análise original. Essa mudança pode ter influenciado os resultados obtidos. Essa informação pode ser confirmada no repositório do trabalho [12].

Vulnerabilidade	**Mythril<=0.21.15**	**Mythril 0.24.7**	**Slither<=0.6.6**	**Slither 0.10.0**
Access Control	5/24	7/24	6/24	7/24
Arithmetic	15/22	14/22	0/22	0/22
Denial Service	0/14	0/14	0/14	0/14
Front Running	2/7	1/7	0/7	0/7
Reentrancy	5/8	5/8	7/8	8/8
Time Manipulation	0/5	1/5	1/5	1/5
Unchecked Low Calls	7/9	5/9	1/9	1/9

Table 2. Quantidade de vulnerabilidades encontradas pelas ferramentas e quantidade de vulnerabilidades reais [Durieux et al. 2020].

Com as análises manuais, foi possível corrigir a contagem, como mostrado nos anexos. Alguns apontamentos que não seriam identificados pelas análises automáticas

[12]https://github.com/SmartBugs/SmartBugs-curated/blob/main/ICSE2020_curated_69.txt

foram devidamente ajustados. Dessa forma, constatamos que, no Mythril, as detecções relacionadas a *Arithmetic*, *Front Running* e *Unchecked Low Calls* apresentaram uma piora, com uma redução de uma unidade nas detecções. A nova análise não conseguiu detectar a vulnerabilidade presente no arquivo *timelock.sol*.

Buscou-se entender por que apenas esse código apresentou um resultado diferente. Considerou-se a hipótese de que uma falha na saída do *SmartBugs* pudesse ter apontado incorretamente a vulnerabilidade. No entanto, também era possível que a versão do Mythril utilizada na época não fosse capaz de detectá-la. Para investigar, realizou-se uma nova análise com a versão 0.21.5 do Mythril. Durante esse processo, surgiram problemas de compatibilidade, que se acredita estarem relacionados à versão do Python utilizada. Para resolver isso, foi necessário corrigir alguns problemas no código do Mythril para que ele pudesse ser executado.

Com a análise realizada na versão 0.21.5, o Mythril conseguiu identificar exatamente a vulnerabilidade apontada pelo *SmartBugs*. Dessa forma, não restam dúvidas de que, para esses contratos específicos, houve uma piora na capacidade de detecção.

Outra vulnerabilidade que também teve uma piora foi a Front Running. A diferença encontrada foi no arquivo eth_tx_order_dependence_minimal.sol. A análise do Smartbugs mostrou duas detecções verdadeiras e a análise deste trabalho somente uma. O trabalho de Munir e Reichenbach [Munir and Reichenbach 2023] tem o foco somente voltado para a análise da vulnerabilidade Transaction Ordering Dependency, que para o DASP é a Front Running. Neste trabalho também estudam o código eth_tx_order_dependence_minimal.sol e confirmam a primeira vulnerabilidade indicada no SmartBugs.

Dentre as vulnerabilidades que apresentaram melhorias na detecção com o Mythril, destacam-se Access Control e Time Manipulation. Além da melhoria na detecção de verdadeiros positivos, um ponto importante evidenciado, e que também é apontado como uma dificuldade nos trabalhos referenciados, foi a evolução na identificação dessas falhas. Além da melhoria dos falsos positivos, observamos que tanto Access Control quanto Time Manipulation no Mythril conseguiram detectar mais de duas vulnerabilidades que não haviam sido identificadas na análise anterior. Por outro lado, a vulnerabilidade Reentrancy manteve os mesmos valores da análise anterior.

Analisando o Slither notamos que a análise da versão mais nova teve nas vulnerabilidades Access Control e Reentrancy uma melhoria de detecção das vulnerabilidade acuradas. No Access Control o arquivo solidity parity_wallet_bug2.sol teve a diferença em um. Na vulnerabilidade de Reentrancy a diferença aparece no arquivo modifier_reentrancy.sol.

Apesar da pequena diferença na detecção das vulnerabilidades existentes entre as duas versões, é possível destacar um ponto relevante na análise de precisão. Nas análises mais recentes, o Mythril apresentou uma melhor precisão em comparação com a versão anterior. Já o Slither obteve uma precisão de 1 em Access Control, acertando todas as suas detecções.

Como discutido na fundamentação teórica [Chess and West 2007], ferramentas de análise automatizada enfrentam desafios relacionados a falsos positivos. Essa questão também é abordada em estudos anteriores, como [Zheng et al. 2023] e [Ye et al. 2019].

Precision	Mythril 0.21.15	Mythril 0.24.7	Slither 0.6.6	Slither 0.10.0
Access Control	20,83%	50,00%	30,00%	100,00%
Arithmetic	16,30%	66,67%	0,00%	0,00%
Front Running	09,52%	05,56%	0,00%	0,00%
Reentrancy	31,25%	20,83%	46,67%	50,00%
Time Manipulation	0,00%	14,29%	20,00%	04,35%
Unchecked Low Calls	23,33%	45,45%	07,69%	0,00%

Table 3. Tabela de Precisão das ferramentas.

No Mythril, houve melhorias de precisão nas categorias Access Control, Arithmetic, Time Manipulation e Unchecked Low Calls. Por outro lado, o Slither apresentou melhorias em Access Control e Reentrancy, mas registrou piora em Time Manipulation e Unchecked Low Calls.

6. Conclusão

A análise das ferramentas Mythril (versões 0.21.15 e 0.24.7) e Slither (versões 0.6.6 e 0.10.0) aplicada aos 69 arquivos Solidity do SmartBugs revelou diferenças notáveis na detecção de vulnerabilidades. As atualizações nas versões das ferramentas causaram mudanças significativas nos resultados, tanto na detecção de vulnerabilidades reais quanto na quantidade de falsos positivos. No caso do Mythril, observou-se uma redução expressiva nas vulnerabilidades aritméticas, que caíram de 92 para 21 ocorrências, indicando uma diminuição considerável dos falsos positivos. Contudo, também foi identificado um impacto negativo na detecção de algumas vulnerabilidades, como *Front Running* e *Unchecked Low Calls*, que apresentaram uma diminuição de uma ocorrência em comparação com a análise anterior.

Além disso, uma melhoria importante foi observada na detecção da vulnerabilidade *Time Manipulation*, que antes não era identificada, mas passou a ser reconhecida na versão 0.24.7, como evidenciado no arquivo *roulette.sol*. Esse avanço demonstra a evolução da ferramenta na capacidade de identificar vulnerabilidades mais complexas, ampliando seu alcance. No caso do Slither, a versão 0.10.0 obteve uma taxa de 100% de acertos na categoria *Access Control*, destacando-se pela precisão na detecção dessa vulnerabilidade. No entanto, a ferramenta ainda apresenta limitações, como evidenciado pela detecção incorreta de *Time Manipulation* no arquivo *ether_lotto.sol*, onde a vulnerabilidade foi identificada na linha errada.

Apesar das melhorias observadas, a análise mostrou que, em alguns aspectos, a precisão das ferramentas ainda apresenta desafios. Mesmo com as atualizações, erros de classificação e localização das vulnerabilidades continuam ocorrendo, o que destaca a necessidade de uma revisão manual para garantir a acuracidade dos resultados. Essa constatação aponta para a importância de utilizar uma abordagem híbrida, combinando a análise automatizada com a revisão humana para aumentar a confiabilidade das detecções.

Em conclusão, as atualizações nas ferramentas Mythril e Slither resultaram em avanços significativos, principalmente na redução de falsos positivos. No entanto, ainda há espaço para melhorias, especialmente na detecção precisa de vulnerabilidades críticas como *Time Manipulation*. A pesquisa futura deverá explorar abordagens híbridas e outras estratégias para aprimorar ainda mais a acuracidade das ferramentas, garantindo maior

confiabilidade nas análises de segurança em contratos inteligentes.

References

Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39.

Belotti, M., Božić, N., Pujolle, G., and Secci, S. (2019). A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838.

Bhushan, B., Sinha, P., Sagayam, K. M., et al. (2021). Untangling blockchain technology: A survey on state of the art, security threats, privacy services, applications and future research directions. *Computers & Electrical Engineering*, 90:106897.

Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*, 3(37).

Chess, B. and McGraw, G. (2004). Static analysis for security. *IEEE security & privacy*, 2(6):76–79.

Chess, B. and West, J. (2007). *Secure programming with static analysis*. Pearson Education.

Di Angelo, M. and Salzer, G. (2019). A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*, pages 69–78. IEEE.

Di Pierro, M. (2017). What is the blockchain? *Computing in Science & Engineering*, 19(5):92–95.

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

Eshghie, M., Artho, C., and Gurov, D. (2021). Dynamic vulnerability detection on smart contracts using machine learning. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, pages 305–312.

Feist, J., Grieco, G., and Groce, A. (2019). Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE.

Ferreira, J. F., Cruz, P., Durieux, T., and Abreu, R. (2020). Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 1349–1352.

Guo, Y. and Liang, C. (2016). Blockchain application and outlook in the banking industry. *Financial innovation*, 2:1–12.

Ibba, G., Aufiero, S., Neykova, R., Bartolucci, S., Ortu, M., Tonelli, R., and Destefanis, G. (2024). A curated solidity smart contracts repository of metrics and vulnerability. In *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 32–41.

Liao, J.-W., Tsai, T.-T., He, C.-K., and Tien, C.-W. (2019). Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 458–465. IEEE.

Maesa, D. D. F. and Mori, P. (2020). Blockchain 3.0 applications survey. *Journal of Parallel and Distributed Computing*, 138:99–114.

Munir, S. and Reichenbach, C. (2023). Todler: A transaction ordering dependency analyzer-for ethereum smart contracts. In *2023 IEEE/ACM 6th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16. IEEE.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260.

Prechtel, D., Groß, T., and Müller, T. (2019). Evaluating spread of 'gasless send'in ethereum smart contracts. In *2019 10th IFIP international conference on new technologies, mobility and security (NTMS)*, pages 1–6. IEEE.

Qian, P., Liu, Z., He, Q., Zimmermann, R., and Wang, X. (2020). Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access*, 8:19685–19695.

Singh, S., Hosen, A. S., and Yoon, B. (2021). Blockchain security attacks, challenges, and solutions for the future distributed iot network. *IEEE Access*, 9:13938–13959.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., and Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82.

Ye, J., Ma, M., Peng, T., Peng, Y., and Xue, Y. (2019). Towards automated generation of bug benchmark for smart contracts. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 184–187. IEEE.

Zheng, Z., Zhang, N., Su, J., Zhong, Z., Ye, M., and Chen, J. (2023). Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. *arXiv preprint arXiv:2303.13770*.