Detecção de Vulnerabilidades em Contratos Inteligentes Utilizando Arvore Sintática Abstrata

Eduardo V. B. Esquivel[1], Josué Nunes Campos[1], Ronan Dutra Mendonça[1], Alex Borges Vieira[2], José Augusto Miranda Nacif[1]

[1]Instituto de Ciências Exatas e Tecnológicas, *Campus* UFV-Florestal
Universidade Federal de Viçosa (UFV) – Florestal, MG – Brazil

[2]Departamento de Informática, Universidade Federal de Juiz de Fora
(UFJF) – Juiz de Fora, MG – Brazil

{eduardo.esquivel,josue.campos,ronan.dutra,jnacif}@ufv.br
alex.borges@ufjf.edu.br

Abstract. *As we move towards new decentralized application solutions, we can observe a growing number of smart contracts on blockchain networks, ranging from more straightforward implementations to more complex and valuable ones. However, this significant increase brings along a concerning amount of vulnerable contracts, susceptible to attacks from malicious actors aiming to steal resources or cause application failures. This paper aims to provide documentation on detecting the main vulnerabilities in Solidity and present an automated detection solution that utilizes static analysis techniques to identify vulnerabilities in large datasets of smart contracts. Based on this, performance tests and comparisons were conducted with the leading current tools in the smart contract verification field, In these experiments, our tool achieved good results in terms of both execution time and detection capability compared to the others.*

Resumo. *À medida que avançamos em direção a novas soluções de aplicações descentralizadas, observa-se um crescente número de contratos inteligentes nas redes blockchain, desde implementações mais simples até as mais complexas e valiosas. No entanto, esse aumento significativo traz consigo uma preocupante quantidade de contratos vulneráveis, que estão suscetíveis a ataques de invasores mal-intencionados com o objetivo de roubar recursos ou causar falhas nas aplicações. O objetivo deste artigo é fornecer documentação sobre a detecção das principais vulnerabilidades do Solidity e apresentar uma solução de detecção automatizada que utiliza técnicas de análise estática para identificar vulnerabilidades em grandes conjuntos de dados de contratos inteligentes. Para tanto, foram realizados testes de desempenho e comparações com as principais ferramentas atuais da área de verificação de contratos inteligentes, nesses experimentos a nossa ferramenta apresentou bons resultados tanto em termos de tempo de execução como em capacidade de detecção em relação às demais.*

1. Introdução

Em decorrência do rápido avanço das tecnologias blockchain, os contratos inteligentes emergiram como uma forma revolucionária de automatizar e executar acordos digitais de

forma confiável, transparente e imutável. Os contratos inteligentes são programas auto executáveis e armazenados nas redes blockchain, que foram projetados para automatizar e garantir a execução de transações entre as partes envolvidas, sem a necessidade de intermediários.

Um dos sistemas mais populares para implementar contratos inteligentes é a Ethereum Virtual Machine (EVM). Ela é uma "máquina virtual" criada especialmente para executar esses contratos na blockchain Ethereum. Isso permite que desenvolvedores implantem os contratos escritos em linguagem de programação Solidity na blockchain para serem executados de forma confiável [Dannen 2017].

No entanto, à medida que o número de contratos inteligentes aumenta rapidamente, também surgem preocupações sobre a segurança e a detecção de vulnerabilidades nesses contratos. Erros de programação, falhas de projeto e brechas de segurança podem causar consequências graves, como a perda de recursos financeiros ou até mesmo serem explorados por pessoas, como relatado em [Ndiaye and Konate 2021]. Sendo assim, muitas das vulnerabilidades que ocorreram e ainda ocorrem nos contratos inteligentes foram documentadas por meio de especialistas de segurança, pesquisadores e usuários. O conhecimento e documentação das vulnerabilidades é algo essencial para a compreensão e estudo dessa área, evitando que programas futuros sofram dos mesmos ataques e sejam explorados da mesma forma.

Este artigo explora os conceitos fundamentais da análise estática em Solidity, abordando também a documentação de vulnerabilidades comuns já conhecidas. Serão destacadas as vulnerabilidades encontradas em registros diversos, classificadas por tipo e método de detecção baseado no código fonte. Uma solução chamada ASTSecurer é apresentada como uma técnica de análise estática para detectar vulnerabilidades, usando um analisador semântico com base em vulnerabilidades conhecidas. Isso possibilita a análise automatizada de código extenso, identificando vulnerabilidades e sua localização por meio da análise da Árvore Sintática Abstrata (AST), que revela padrões de vulnerabilidade. A ferramenta inclui uma interface para o usuário final por meio de linha de comando, permitindo a verificação de vulnerabilidades em conjuntos de dados selecionados, fornecendo resultados em tempo de execução.

Ademais serão apresentados ao longo do artigo tópicos explicativos sobre os principais conceitos que envolvem o reconhecimento de vulnerabilidades, assim como técnicas que são constantemente utilizadas para a verificação do código de contratos inteligentes.

2. Visão Geral

Ao abordar a verificação de contratos inteligentes, é necessário explorar técnicas e conceitos específicos. Estes contratos apresentam peculiaridades em sua implementação, o que os torna mais suscetíveis a serem verificados por meio de abordagens distintas.

2.1. Análise Estática e Análise Dinâmica

Existem duas maneiras de realizar testes em contratos inteligentes: análise estática e análise dinâmica.

A análise estática de código é uma técnica que envolve a revisão do código-fonte do software sem a necessidade de executá-lo. Geralmente, são utilizadas ferramentas de

software para examinar o código-fonte em busca de erros de programação, violações de padrões de codificação, vulnerabilidades de segurança e outros problemas. Por outro lado, a análise dinâmica envolve a execução do software e a observação de seu comportamento durante a execução. Essa técnica é realizada por meio de testes de software, que simulam diferentes situações para detectar erros e falhas de segurança [Adrion et al. 1982].

Existem diversos métodos para realizar os testes, incluindo testes formais, *machine learning*, [Durelli et al. 2019] métodos baseados em modelos [El-Far and Whittaker 2002] e métodos baseados em pesquisas [McMinn 2011].

Atualmente, a maioria das ferramentas existentes utilizam análise estática para detecção de vulnerabilidades em contratos inteligentes, como documentado em [Marijan and Lal 2022]. Isso se deve ao fato de que as técnicas de análise estática têm se mostrado mais eficientes em termos de tempo na detecção de um maior número de vulnerabilidades em contratos inteligentes até o momento, uma vez que para executar um contrato inteligente é necessário simular todo um ambiente de execução [Luu et al. 2016].

2.2. Verificação Formal

A verificação formal é uma abordagem que se utiliza de métodos lógicos e matemáticos para detectar e verificar sistemas de software. Nesse processo é realizada a modelagem formal do sistema, expressando suas propriedades e requisitos em termos de matemática lógica. Essa modelagem matemática permite uma análise precisa e abrangente do sistema, ajudando a identificar erros e garantir a exatidão ou precisão do software [Hasan and Tahar 2015].

No contexto da tecnologia blockchain, a verificação formal pode ser aplicada ao código dos contratos inteligentes, o que é extremamente útil para detectar diferentes tipos de vulnerabilidades. Através da modelagem matemática, é possível identificar de forma mais precisa e eficiente as potenciais fragilidades no código. Ao utilizar técnicas de verificação formal, é possível analisar minuciosamente o contrato inteligente e identificar condições que possam tornar o código vulnerável a ataques ou explorações indesejadas.

Ao contrário da verificação tradicional, que é baseada em testes e verificações manuais, a verificação formal usa métodos automatizados para realizar uma análise completa do contrato inteligente. Isso inclui a verificação de todos os possíveis caminhos de execução do contrato, bem como a verificação de todas as condições de entrada e saída. Nos últimos anos, uma série de técnicas de verificação têm sido amplamente empregadas para detecção de erros e análise de vulnerabilidades. Dentre essas técnicas, destacam-se o *model checking, theorem proving, symbolic execution*.

2.2.1. Model Checking

O *model checking* é uma técnica de verificação formal amplamente empregada na engenharia de software e hardware para assegurar a exatidão de sistemas complexos, traduzindo os mesmos para a linguagem SMT a fim de verificar propriedades desejadas do modelo criado [Clarke 1997].

Uma vez que o modelo formal é criado, é possível realizar a verificação por meio do *model checking*. Essa verificação consiste em explorar todas as possíveis execuções

do sistema, em busca de estados indesejados ou violações de propriedades especificadas. Essas propriedades podem descrever requisitos de segurança, exatidão funcional ou comportamental do sistema.

No contexto dos contratos inteligentes, o *model checking* é frequentemente realizado utilizando ferramentas como NuSMV e nuXmv [Cimatti et al. 1999] [Cavada et al. 2014], que se valem da técnica do *model checking* para analisar expressões como LTL (Linear Temporal Logic) ou CTL (Computation Tree Logic). Além disso, outras aplicações fazem uso de solucionadores SMT para analisar a linguagem simbólica dos contratos.

2.2.2. Symbolic Execution

Symbolic execution é uma técnica de análise estática de código que é usada para verificar o comportamento do programa de forma simbólica, em vez de executá-lo de fato. Ela permite analisar o comportamento do programa para uma entrada genérica, em vez de analisar o comportamento específico para um conjunto finito de entradas de teste [King 1976].

Durante o processo o código é executado simbolicamente, substituindo os valores das variáveis por expressões simbólicas que representam suas possíveis atribuições. Essas expressões simbólicas são manipuladas de acordo com as instruções do programa, produzindo novas expressões simbólicas que representam o estado do programa em diferentes pontos de execução. Os SMT *solvers* são amplamente utilizados na *symbolic execution* para verificar se as expressões simbólicas geradas durante a execução simbólica satisfazem certas propriedades ou restrições.

Em solidity a grande maioria das ferramentas de verificação de código e vulnerabilidade se utiliza da técnica *symbolic execution*, uma vez que para vulnerabilidades mais complexas nas quais os valores de entradas estão diretamente ligados a exploração de erros, ele se mostra mais eficaz, produzindo menor número de falsos positivos no geral, como reportado em [Luu et al. 2016].

2.2.3. Theorem Proving

Theorem Proving consiste na técnica de codificar o sistema e suas propriedades em uma lógica matemática particular. Após isso se tenta obter uma prova formal na qual se faça a satisfação das propriedades do sistema codificado. Diferente do *model checking* que apenas consegue verificar sistemas de estados finitos, o *Theorem proving* consegue verificar sistemas de estados infinitos [Green 1981].

2.3. SMT Solver

SMT (Satisfiability Modulo Theories) solver é um software projetado para resolver problemas de satisfação, justificação, explicação de fórmulas de lógica de primeira ordem em teorias específicas, como aritmética, teoria dos conjuntos, teoria da igualdade, entre outras. No contexto de Solidity muitas ferramentas utilizam de SMT *solvers* para realizar a detecção de vulnerabilidades e bugs no código. Dentre os *solvers* mais utilizados está o solver Z3 [De Moura and Bjørner 2008].

3. Trabalhos Relacionados

Durante os últimos anos, surgiram várias ferramentas de verificação de vulnerabilidades em contratos inteligentes. Muitas dessas ferramentas utilizam técnicas avançadas de análise estática para identificar possíveis bugs em contratos vulneráveis.

Uma das ferramentas disponíveis para análise do *bytecode* EVM é o Mythril [ConsenSys 2018], que utiliza a execução simbólica para identificar certos tipos de vulnerabilidades. Embora tenha demonstrado resultados satisfatórios em relação a algumas vulnerabilidades, a análise de contratos simples usando essa ferramenta requer um tempo significativo.

O Slither [Feist et al. 2019] é uma ferramenta adicional de análise que emprega métodos de análise estática para detectar vulnerabilidades específicas. Ele faz uso de técnicas de representação intermediária para identificar essas vulnerabilidades durante o processo de análise, o que torna o processo um pouco mais rápido do que as técnicas que fazem uso de modelagem complexas, entretanto possui resultados menos precisos para alguns tipos de vulnerabilidades.

Além disso, em 2019 foi publicada a Manticore, uma ferramenta popular descrita em [Mossberg et al. 2019], que utiliza a técnica de execução simbólica e se comporta de maneira semelhante ao Mythril, analisando certos caminhos do contrato inteligente e verificando certas propriedades para encontrar vulnerabilidade, também demandando certo esforço computacional, tornando assim todo o processo mais lento.

4. Metodologia

A ferramenta de detecção ASTSecurer foi desenvolvida com base na documentação SWC (Smart Contract Weakness Classification and Test Cases) [SmartContractSecurity 2020], que é amplamente reconhecida por programadores de contratos inteligentes, sendo bastante abrangente em relação às vulnerabilidades da linguagem Solidity. A partir dessa documentação, foram criados artefatos úteis para utilização no desenvolvimento e aprimoramento da nossa ferramenta de detecção.

Para a elaboração da ferramenta de detecção, foi empregada uma estratégia baseada na análise estática da AST (Árvore de Sintaxe Abstrata) gerada a partir do código fonte de contratos inteligentes. Foi desenvolvido um algoritmo para identificar e detectar vulnerabilidades previamente selecionadas no registro. Essa abordagem foi cuidadosamente escolhida devido a uma série de motivos fundamentais descritos a seguir.

Em primeiro lugar, a análise estática da AST permite examinar o código-fonte do contrato de forma minuciosa e abrangente. Ao analisar a estrutura sintática do código, é possível identificar padrões, fluxos de controle e relacionamentos entre os elementos do contrato. Isso possibilita a detecção de vulnerabilidades em um nível mais profundo, permitindo uma maior precisão na identificação dos problemas de segurança.

Além disso, a análise estática da AST tem a vantagem de ser eficiente e escalável. Ao contrário da execução do contrato, que exigiria a simulação de todas as possíveis interações com o contrato para detectar vulnerabilidades, a análise estática pode ser realizada antes mesmo da implantação do contrato. Isso reduz significativamente o tempo necessário para verificar a segurança do código, permitindo uma identificação mais rápida e eficiente de possíveis vulnerabilidades.

O método proposto dispõe da ocorrência de que algumas vulnerabilidades podem ser identificadas por meio de padrões semânticos específicos. Isso elimina a necessidade de converter as propriedades do código em modelos complexos, permitindo uma análise mais rápida e prática. Entretanto a utilização dessa técnica limita o grupo de vulnerabilidades que podem ser detectadas, assim sendo necessário delimitar os tipos de vulnerabilidades a serem suportadas na ferramenta de detecção.

Atualmente, existem muitos dados e contratos inteligentes escritos, e há várias ferramentas que propõem a detecção de vulnerabilidades. No entanto, muitas dessas ferramentas requerem um esforço computacional significativo, o que torna a análise lenta e ineficiente quando aplicada a conjuntos de dados extensos de contratos inteligentes.

Portanto, foi desenvolvida uma ferramenta que realiza uma filtragem inicial de certas vulnerabilidades de contratos inteligentes em tempo hábil. Essa ferramenta utiliza a técnica de análise dos nós da AST para exibir em quais métodos ou variáveis foram encontradas as vulnerabilidades no código fonte em linguagem Solidity.

4.1. Documentação de Vulnerabilidades

Com base nas vulnerabilidades registradas no SWC, foi realizado um estudo detalhado e minucioso, resultando na criação de um artefato abrangente contendo detalhes importantes. Esse artefato foi convertido cuidadosamente em uma tabela, a partir disso nós documentamos as informações essenciais para a compreensão das vulnerabilidades em contratos inteligentes.

Dentro da Tabela 1, é possível encontrar o nome específico de cada vulnerabilidade, bem como o código de identificação correspondente. Além disso, uma breve descrição é fornecida, oferecendo uma visão geral das características e potenciais impactos de cada vulnerabilidade identificada. Para auxiliar ainda mais na detecção e compreensão dessas vulnerabilidades, cada uma delas é acompanhada de um método específico de detecção em contratos inteligentes, nos quais descrevem um passo a passo para identificar as vulnerabilidades de forma prática.

A Tabela 1 é uma representação abrangente dos tipos de vulnerabilidades que são suportados pelo mecanismo de detecção do ASTSecurer. Através dessa tabela, é possível visualizar de forma clara e organizada os diferentes tipos de vulnerabilidades que o ASTSecurer tem a capacidade de identificar em contratos escritos em Solidity.

Tabela 1. Detalhes das Vulnerabilidades

ID/Nome	Descrição	Detecção
SWC-100 Function Default Visibility	Funções com tipo de visibilidade não especificado são tratadas como públicas por padrão.Em casos do programador esquecer de definir a visibilidade correta podem ocorrer ataques maliciosos.	Para verificar esse tipo de vulnerabilidade basta conferir se todas as funções do contrato inteligente estão com o tipo de visibilidade definida , caso negativo o contrato é suscetível a este tipo de vulnerabilidade.

SWC-102 Outdated Compiler Version	Ao se utilizar uma versão desatualizada do compilador o código pode estar suscetível a bugs e vulnerabilidades expostas.	Basta verificar se o seu contrato está sendo executado pela versão mais recente do compilador do Solidity.
SWC-104 Unchecked Call Return Value	Ocorre quando os valores de retornos das chamadas não são verificados, a execução continua mesmo após haver uma exceção, podendo assim ocorrer resultados indesejados.	Verifique se há chamadas a funções externas sem verificar o valor de retorno. Verifique também se o código está corretamente lidando com o caso em que a função externa falha.
SWC-108 State Variable Default Visibility	Variáveis com tipo de visibilidade não especificado são tratadas como públicas por padrão.Em casos do programador esquecer de definir a visibilidade correta podem ocorrer ataques maliciosos.	Para verificar esse tipo de vulnerabilidade basta conferir se todas as variáveis do contrato inteligente estão com o tipo de visibilidade definida , caso negativo o contrato é suscetível a este tipo de vulnerabilidade.
SWC-111 Use of Deprecated Solidity Functions	Ocorre devido ao uso de funções obsoletas ou descontinuadas na linguagem de programação Solidity. Se um desenvolvedor ainda as usa em seu código, isso pode resultar em problemas de segurança ou problemas de eficiência.	Conferir se as funções presentes no código são obsoletas conforme a documentação mais atualizada do compilador do Solidity.
SWC-115 Authorization through tx.origin	Consiste no uso incorreto da variável “tx.origin”, onde se utilizada para validação de autoridade de forma errada pode implicar em roubos de fundos por contratos maliciosos com controle de autorização deste contrato.	A detecção pode ser realizada verificando se o contrato possui em seu código a utilização do “tx.origin” para autorização de alguma funcionalidade ou requisição.
SWC-116 Block values as a proxy for time	Devido à mineração de novos blocos ser dificultada e à possibilidade de congestionamento da rede, a velocidade de adição de blocos pode variar. Além disso, os mineradores podem atrasar intencionalmente a mineração para manipular o valor do bloco.	Verificar se o contrato utiliza as variáveis tais como “block.timestamp” ou “block.number”, ou algum tipo de má aleatoriedade para ativar certas condições do contrato.

SWC-127 Arbitrary Jump with Function Type Variable	Solidity possui suporte para tipos de funções, ou seja pode se atribuir uma função a variável.Entretanto o uso incorreto disso juntamente com a utilização com uso de assembly pode acarretar nos usuários podendo alterar diretamente essas funcionalidades.	Verificar se o código faz uso de variáveis do tipo função utilizando isso em conjunto com a utilização de assembly.
SWC-130 Right-To-Left-Override control character	Os Usuários podem utilizar o caractere único de Right-To-Left-Override para forçar uma renderização de texto RTL.	Verificar se o contrato permite entradas inválidas em certas áreas visíveis para outros usuários , tais como o caractere Right-To-Left-Override control.
SWC-131 Presence of unused variables	Consiste na existência de variáveis não utilizadas, que podem acarretar em um maior gasto de gás (taxa paga na rede para o uso do poder computacional da plataforma) , e ruídos no código.	Verificar se o código possui variáveis declaradas e que não são utilizadas em todo o contrato inteligentes.
SWC-134 Message call with hardcoded gas amount	As funções “send()” e “transfer()”, utilizam fixamente 2300 de gás, entretanto esses valores podem ser modificados durantes os “hard forks”, quebrando certas lógicas de contratos que supõem apenas esses valores fixos.	Verificar se o contrato inteligente utiliza funcionalidades tais como send() ou transfer(), apenas esperando valores fixos de gás já predefinidos.

4.2. Árvore de Sintaxe Abstrata

A AST é uma estrutura de dados que descreve sintaticamente um código, indicando várias informações úteis em seus nós, como o tipo de dado de uma variável ou as expressões que compõem outras expressões.

O uso de uma árvore de sintaxe abstrata é comum em compiladores, interpretadores e ferramentas de análise estática de código. Ela permite que o código-fonte seja processado e analisado de forma mais eficiente, facilitando a detecção de erros, a transformação do código e a geração de código intermediário ou otimizado. Resumidamente a Árvore Sintática Abstrata é uma representação intermediária e abstrata do código-fonte de um programa, organizada em uma estrutura de árvore que facilita a análise e manipulação do código.

Podemos observar um exemplo de AST de um contrato inteligente na (Figura 1).

Nesse exemplo é demonstrado a estrutura de um código em Solidity utilizando uma representação AST, assim tendo os seus nós correspondendo às variáveis e expressões de um contrato inteligente. Esse contrato exibe uma mensagem “Hello Word” ao acionar uma determinada funcionalidade nomeada “sayHell”.

A AST é uma estrutura muito facilitadora para análise e verificação de código, uma vez que, podemos analisar semanticamente certas expressões que podem se encaixar em um padrão vulnerável. Por conta dessa praticidade o ASTSecurer se utiliza dessa estrutura para a detecção de vulnerabilidades utilizando diretamente a AST para analisar certos padrões específicos no código de contratos inteligentes, a fim de detectar trechos vulneráveis e que são suscetíveis a ataques de autores maliciosos na blockchain, e a partir disso exibe para o usuário onde se encontra localizado cada trecho vulnerável no código base, ou seja o local exato onde está presente aquele tipo de vulnerabilidade.

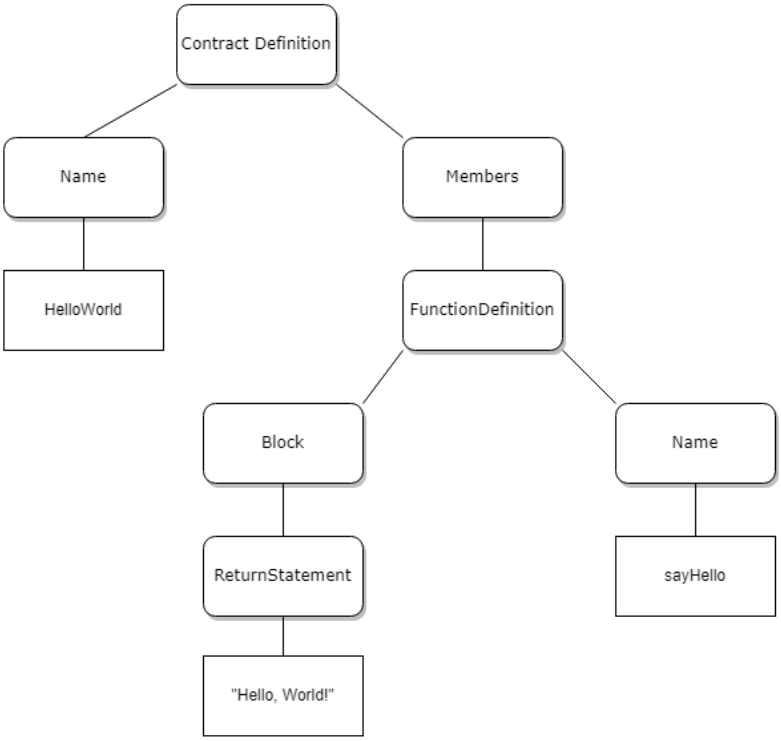

Figura 1. Representação AST

5. Resultados

Nesta seção, serão apresentados os resultados das execuções do ASTSecurer, bem como métricas para avaliar a confiabilidade e capacidade de detecção de vulnerabilidades. Essas métricas foram obtidas por meio de diversas execuções do programa em diferentes conjuntos de dados de contratos inteligentes.

Foram analisados 10480 contratos inteligentes aleatórios reais através da execução da ferramenta, podendo esses códigos do conjunto de dados possuírem vulnerabilidades

ou não, isso foi feito a fim de medir o tempo médio de execução da ferramenta para análise de vulnerabilidades em um conjunto de dados de contratos inteligentes relativamente grande. Os resultados conforme mostrado na Tabela 2 foram bastante satisfatórios tendo um tempo médio de execução por contrato de apenas 1,46 segundos.

Tabela 2. Tempo de Análise ASTSecurer

Número de contratos inteligentes	Tempo de Análise (s)
10.480	15.250,97

Além disso, foram realizadas análises comparativas do tempo de verificação de um conjunto de dados de códigos utilizando outras ferramentas de detecção de contratos inteligentes, nomeadamente Mythril, Oyente, Slither e Manticore. Essas análises foram executadas em uma mesma máquina, com a mesma configuração. O método de verificação semântica do ASTSecurer demonstrou resultados significativamente mais rápidos em comparação às outras ferramentas. Foram testados no total 39 códigos de contratos inteligentes distintos, cada um contendo vulnerabilidades diferentes, com isso foi realizada uma análise do tempo de execução de todas as ferramentas conforme é apresentado na Tabela 3.

Tabela 3. Tempo de Execução de Ferramentas de Análise

Ferramenta	Número de Contratos Inteligentes	Tempo de Análise Total (s)
ASTSecurer	39	61,62
Mythril	39	3.388.85,24
Oyente	39	473,20
Slither	39	107,81
Manticore	39	22.986,46

Na (Figura 2), é apresentado o tempo médio de execução em segundos para a análise de cada código nas diversas ferramentas. É possível observar uma notável disparidade de velocidade entre as ferramentas que empregam técnicas mais complexas de análise estática em comparação àquelas que utilizam representação intermediária ou análise semântica direta.

Através da análise dos resultados obtidos em execuções do mesmo conjunto de dados, foi realizado um estudo sobre as vulnerabilidades detectadas em determinados contratos inteligentes vulneráveis. Esses contratos inteligentes foram previamente classificados em relação às suas vulnerabilidades por meio de uma revisão manual.

Os contratos foram divididos com base nas vulnerabilidades presentes em cada um, sendo possível que um mesmo contrato possua várias vulnerabilidades do mesmo tipo ou de tipos diferentes. As vulnerabilidades correspondentes aos contratos são: "*Function Default Visibility*", "*Outdated Compiler Version*", "*Unchecked Call Return Value*", "*Use of Deprecated Solidity Functions*", "*Authorization through tx.origin*", "*Block values as a proxy for time*", e "*Presence of unused variables*". Essas vulnerabilidades estão relacionadas no registro SWC, respectivamente, aos seguintes códigos: SWC-100, SWC-102, SWC-104, SWC-111, SWC-115, SWC-116 e SWC-131.

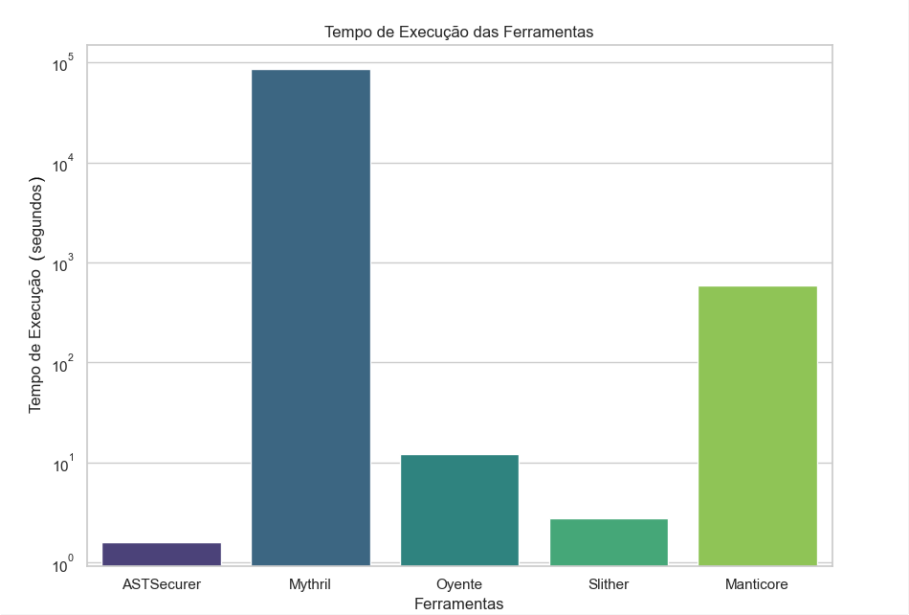

Figura 2. Comparação de Tempo de Execução

A Tabela 4 contém o total de vulnerabilidades presentes no conjunto de dados, enquanto os resultados do experimentos estão presentes na (Figura 3), onde é possível observar que o ASTSecurer conseguiu identificar diversas vulnerabilidades distintas, apresentando resultados muito satisfatórios em comparação às demais ferramentas testadas. Apesar do ASTSecurer captar a grande maioria das vulnerabilidades ainda assim algumas podem passar despercebidas, como alguns casos de código que continham a ”SWC-116”. Mesmo a ferramenta possuindo um tempo de execução baixo, como mencionado anteriormente, o ASTSecurer consegue realizar a detecção de forma eficiente em comparação às demais ferramentas testadas.

Tabela 4. Total de Tipos de Vulnerabilidades no Conjunto de Dados

SWC-100	SWC-102	SWC-104	SWC-111	SWC-115	SWC-116	SWC-131
5	15	12	3	7	5	6

Por fim, foi testada a ferramenta em conjunto de dados de contratos vulneráveis popular obtido em [SmartBugs 2020], este repositório contém diversos códigos em Solidity com vulnerabilidades classificadas em cada um, sendo constantemente alimentados e atualizados pela comunidade, a partir do mesmo foram feitas diversas execuções do ASTSecurer a fim de averiguar a precisão da ferramenta na detecção de certos tipos de vulnerabilidades.

Foram selecionados três conjuntos de dados principais, cada um representando uma vulnerabilidade específica: “*Unchecked Calls*” (Chamadas Não Verificadas), “*Bad Randomness*” (Aleatoriedade Ruim) e “*Access Control*” (Controle de Acesso). Cada conjunto de dados apresentava diferentes tamanhos. Os resultados desse experimento foram bastante satisfatórios, conforme evidenciado na (Figura 4). Nela, pode-se observar que o ASTSecurer conseguiu detectar a maioria das vulnerabilidades nos conjuntos de dados de

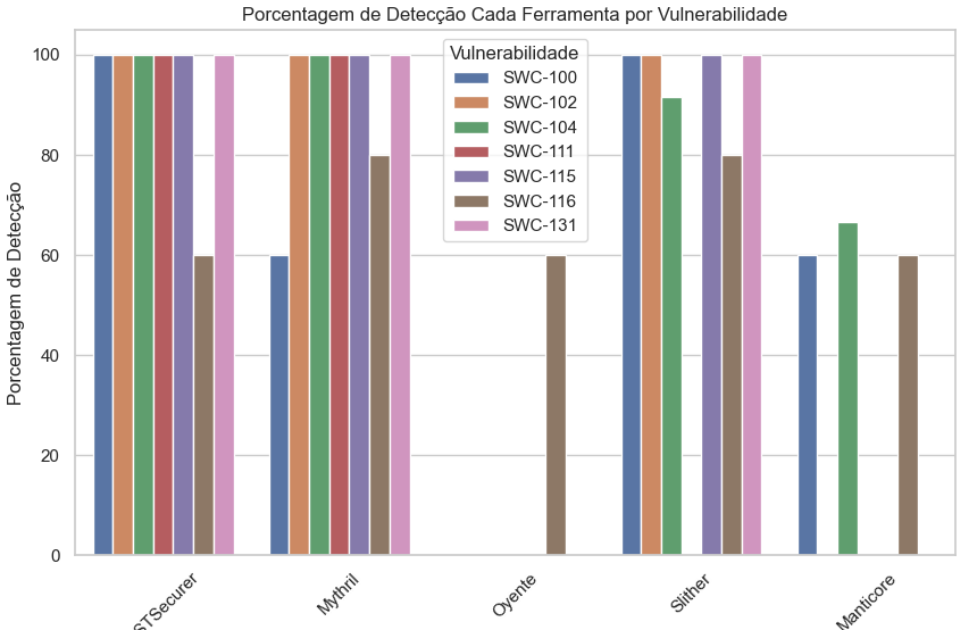

Figura 3. Comparação de Capacidade de Detecção das Ferramentas

contratos reais, mesmo em casos onde os conjuntos de dados eram diversificados e continham tipos de vulnerabilidades para os quais a ferramenta não foi originalmente projetada para detectar.

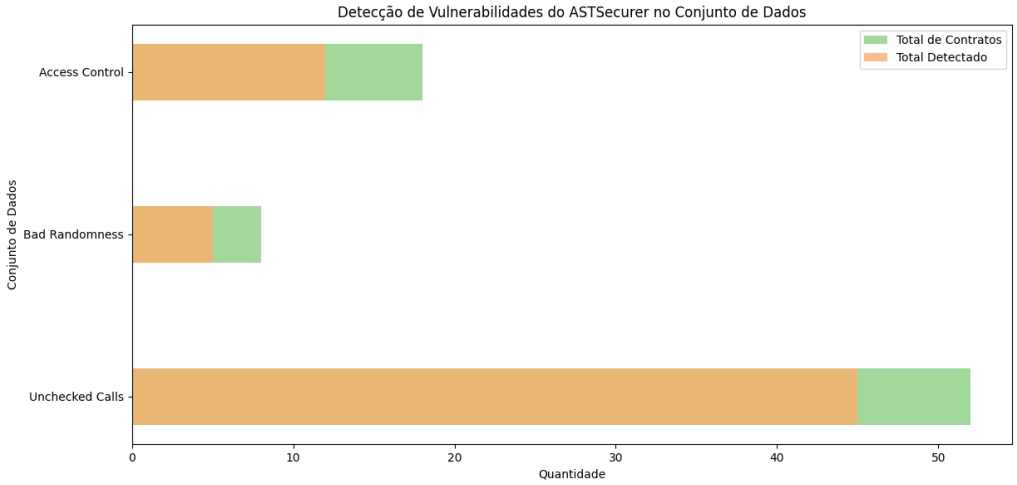

Figura 4. Teste em Conjuntos de Dados Reais

6. Conclusão

A precisão e a confiabilidade dos resultados obtidos pelo ASTSecurer são notáveis. A ferramenta demonstrou uma capacidade robusta de identificar vulnerabilidades específicas, oferecendo aos desenvolvedores uma valiosa análise de segurança para seus contratos

inteligentes. Com suas funcionalidades avançadas e detecção precisa, o ASTSecurer se destaca como uma solução eficiente no setor, contribuindo para a mitigação de riscos e a proteção dos contratos inteligentes contra explorações maliciosas. Esses resultados promissores reforçam a importância de utilizar ferramentas de análise de segurança como o ASTSecurer para garantir a integridade e a confiança dos contratos inteligentes no volumoso sistema da blockchain.

Além disso a ferramenta como demonstrado nos resultados apresenta um baixo tempo de execução, sendo bastante útil para ser utilizada em grande volumes de dados de contrato inteligentes escritos em Solidity, sendo recomendado principalmente para realização de filtragens iniciais em contratos inteligentes, revelando se o mesmo possui algumas das vulnerabilidades documentadas em seu código fonte.

Para futuros trabalhos, há planos de expansão do suporte para detecção de novos tipos de vulnerabilidades em Solidity, além do aprimoramento dos recursos de detecção já existentes no código. O objetivo é aumentar o número de códigos detectados e melhorar a eficiência do ASTSecurer.

Uma abordagem promissora é a utilização de recursos de aprendizado de máquina para detectar padrões na *Abstract Syntax Tree* (AST). Isso pode acelerar o processo de detecção e permitir a identificação de um espectro ainda maior de vulnerabilidades pelo programa. A aplicação de técnicas de *machine learning* na análise de contratos inteligentes pode proporcionar uma detecção mais precisa e automatizada, aprimorando a capacidade da ferramenta de encontrar vulnerabilidades sutis ou complexas.

Essas direções futuras visam a contínua evolução do ASTSecurer, tornando-o mais abrangente, eficiente e capaz de lidar com uma ampla variedade de vulnerabilidades em contratos inteligentes.

Referências

Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C. (1982). Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192.

Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S. (2014). The nuxmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 334–342. Springer.

Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M., et al. (1999). Nusmv: A new symbolic model verifier. In *CAV*, volume 99, pages 495–499. Citeseer.

Clarke, E. M. (1997). Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer.

ConsenSys (2018). Mythril. `https://github.com/ConsenSys/mythril`. Acessado em: 2023.

Dannen, C. (2017). *Introducing Ethereum and solidity*, volume 1. Springer.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference,*

TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14, pages 337–340. Springer.

Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., and Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212.

El-Far, I. K. and Whittaker, J. A. (2002). Model-based software testing. *Encyclopedia of software engineering*.

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.

Green, C. (1981). Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier.

Hasan, O. and Tahar, S. (2015). Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI global.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269.

Marijan, D. and Lal, C. (2022). Blockchain verification and validation: Techniques, challenges, and research directions. *Computer Science Review*, 45:100492.

McMinn, P. (2011). Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE.

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

Ndiaye, M. and Konate, P. K. (2021). Cryptocurrency crime: Behaviors of malicious smart contracts in blockchain. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–8. IEEE.

SmartBugs (2020). SmartBugs Curated Repository. `https://github.com/smartbugs/smartbugs-curated`. Acesso em: 8 de junho de 2023.

SmartContractSecurity (2020). Smart contract weakness classification and test cases. `https://swcregistry.io`. Acessado em 25 de maio de 2023.