CEFET/RJ
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO

DETECÇÃO DE VULNERABILIDADES EM CONTRATOS INTELIGENTES: UM ESTUDO COM FERRAMENTAS AUTOMATIZADAS

Felipe Mello Fonseca

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação, Centro Federal de Educação Tecnológica Celso Suckow da Fonseca CEFET/RJ, como parte dos requisitos necessários à obtenção do grau de mestre.

Orientadores:
Diogo Silveira Mendonça (orientador)
Pedro Henrique Gonzalez (coorientador)

Rio de Janeiro,
Setembro de 2026

Detecção de Vulnerabilidades em Contratos Inteligentes: Um Estudo com Ferramentas Automatizadas

Dissertação de Mestrado em Ciência da Computação, Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ.

Felipe Mello Fonseca

Aprovada por:

Presidente, Prof. Diogo Silveira Mendonça, D.Sc. (orientador)

Pedro Henrique Gonzalez, D.Sc. (coorientador)

Kele Teixeira Belloze, D.Sc.

José Augusto Miranda Nacif, D.Sc.

Rio de Janeiro,

Setembro de 2026

Ficha catalográfica elaborada pela Biblioteca Central do CEFET/RJ

F676 Fonseca, Felipe Mello
Detecção de vulnerabilidades em contratos inteligentes: um estudo com ferramentas automatizadas / Felipe Mello Fonseca. — 2026.
73f + apêndice. : il. color. , enc.

Dissertação (Mestrado) Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, 2026.
Bibliografia : f. 68-73
Orientador: Diogo Silveira Mendonça
Coorientador: Pedro Henrique Gonzalez

1. Computadores - Medidas de segurança. 2. Blockchains (Base de dados). 3. Contratos eletrônicos. 4. Análise de sistemas. I. Mendonça, Diogo Silveira. (Orient.). II. Gonzalez, Pedro Henrique. (Coorient.) III. Título.

CDD 005.8

Elaborada pela bibliotecária Tania Mello – CRB/7 nº 5507/04

DEDICATÓRIA

Dedico este trabalho à minha família, pelo apoio incondicional, paciência e incentivo em cada passo desta jornada. Aos meus professores, pelo conhecimento compartilhado, pela orientação e pela inspiração constante.

AGRADECIMENTOS

"Agradeço ao CEFET-RJ por me oportunizar, mais uma vez, crescimento pessoal e acadêmico."

RESUMO

Detecção de Vulnerabilidades em Contratos Inteligentes: Um Estudo com Ferramentas Automatizadas

Felipe Mello Fonseca

Orientadores:
Diogo Silveira Mendonça (orientador)
Pedro Henrique Gonzalez (coorientador)

Resumo da Dissertação submetida ao Programa de Pós-graduação em Ciência da Computação do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca CEFET/RJ como parte dos requisitos necessários à obtenção do grau de mestre.

A *blockchain* é uma tecnologia com aplicações em diversas áreas, como finanças, votação eletrônica e Internet das Coisas. Nesse contexto, as transações são frequentemente mediadas por contratos inteligentes, programas autoexecutáveis escritos predominantemente em Solidity. Por manipularem valores expressivos em criptomoedas, esses contratos são alvos frequentes de ataques, tornando a detecção automatizada de vulnerabilidades um desafio técnico relevante.

O objetivo deste trabalho é analisar, por meio de dois experimentos, o desempenho de ferramentas de análise estática tradicionais, Mythril e Slither, e de Grandes Modelos de Linguagem (LLMs), Gemini 2.0 Flash e GPT-4, na detecção de vulnerabilidades em contratos inteligentes. No primeiro, avalia-se a evolução das ferramentas de análise estática sobre os contratos do conjunto de dados SmartBugs Curated, segundo a taxonomia DASP, confrontando os resultados com classificações previamente validadas manualmente. No segundo, avalia-se o desempenho do Gemini 2.0 Flash na mesma tarefa de detecção, comparando-o aos resultados reportados para o GPT-4 na literatura.

Os resultados indicam avanços nas versões mais recentes das ferramentas de análise estática: o Mythril reduziu falsos positivos em *Arithmetic* (*F1-score* de 0,26 para 0,65), enquanto o Slither eliminou falsos positivos em *Access Control* e atingiu cobertura total em *Reentrancy* (*F1-score* de 0,67), a melhor combinação observada, com regressão em *Front Running* e *Unchecked Low-Level Calls*. Na avaliação dos LLMs, o Gemini 2.0 Flash apresentou desempenho estatisticamente equivalente ao GPT-4 na maioria das categorias, com vantagem significativa do GPT-4 em quatro delas e vantagem pontual do Gemini em *Access Control*. A auditoria manual de 139 relatórios por modelo revelou um padrão de erro comum: a dependência de indícios sintáticos em detrimento da análise semântica da lógica de negócio. Em conjunto, os resultados reforçam que a combinação de múltiplas abordagens, aliada à validação humana, é o caminho mais promissor para aumentar a confiabilidade dos contratos inteligentes.

Palavras-chave: Blockchain; Smart contracts; Slither; Mythril; LLMs.

Rio de Janeiro,

Setembro de 2026

ABSTRACT

Detection of Vulnerabilities in Smart Contracts: A Study with Automated Tools

Felipe Mello Fonseca

Advisors:
Diogo Silveira Mendonça (orientador)
Pedro Henrique Gonzalez (coorientador)

Abstract of dissertation submitted to Programa de Pós-graduação em Ciência da Computação - Centro Federal de Educação Tecnológica Celso Suckow da Fonseca CEFET/RJ as partial fulfillment of the requirements for the degree of master.

Blockchain is a technology with applications across multiple domains, including finance, electronic voting, and the Internet of Things. In this context, transactions are frequently mediated by smart contracts — self-executing programs written primarily in Solidity. Because they handle significant cryptocurrency values, these contracts are frequent targets of attacks, making automated vulnerability detection a relevant technical challenge.

The objective of this work is to analyze, through two experiments, the performance of traditional static analysis tools, Mythril and Slither, and Large Language Models (LLMs), Gemini 2.0 Flash and GPT-4, in the detection of vulnerabilities in smart contracts. In the first experiment, the evolution of the static analysis tools is evaluated on the contracts of the SmartBugs Curated dataset, according to the DASP taxonomy, comparing the results with classifications previously validated manually. In the second, the performance of Gemini 2.0 Flash is evaluated on the same detection task, comparing it to the results reported for GPT-4 in the literature.

The results indicate advancements in the latest versions of the static analysis tools: Mythril reduced false positives in Arithmetic (F1-score from 0.26 to 0.65), while Slither eliminated false positives in Access Control and achieved full coverage in Reentrancy F1-score of 0.67), the best combination observed, with regression in Front Running and Unchecked Low-Level Calls. In the evaluation of LLMs, Gemini 2.0 Flash showed statistically equivalent performance to GPT-4 in most categories, with a significant advantage for GPT-4 in four of them and a specific advantage for Gemini in Access Control. The manual audit of 139 reports per model revealed a common error pattern: reliance on syntactic cues rather than semantic analysis of business logic. Together, the results reinforce that combining multiple approaches, coupled with human validation, is the most promising path to improving the reliability of smart contracts.

Key-words: Blockchain; Smart contracts; Slither; Mythril; LLMs.

Rio de Janeiro,

Setembro de 2026

Sumário

I Introdução **1**

II Fundamentação Teórica **4**

II.1 Blockchain 4

II.2 Ethereum 5

II.3 Static Analysis 5

II.4 Symbolic Execution 6

II.5 Slither 6

II.6 Mythril 7

II.7 Smart Contract Weakness Classification 7

II.8 Decentralized Application Security Project 10

II.9 Grandes Modelos de Linguagem 12

II.9.1 GPT-4 12

II.9.2 Gemini 2.0 Flash 13

II.10 Métricas de Avaliação 14

III Trabalhos Relacionados **15**

III.1 Método de Pesquisa 15

III.2 Análise Estática e Simbólica 17

III.3 Grandes Modelos de Linguagem 23

IV Metodologia **27**

IV.1 Primeiro Experimento: Análise de Vulnerabilidades do Mythril e Slither 28

IV.1.1 Análise Manual: Mythril e Slither 31

IV.2 Segundo Experimento: Análise de Vulnerabilidades com LLMs 32

IV.2.1 Detecção de Vulnerabilidades 33

IV.2.2 Padronização Semântica 34

IV.2.3 Armazenamento e Avaliação 34

IV.2.4 Configuração do Ambiente 35

IV.2.5 Análise Manual: LLM 35

V Resultados **38**

V.1 Primeiro Experimento 38

V.2 Segundo Experimento 44

V.2.1 Resultado Análise Manual 45

V.2.2 Apontamentos nas análises manuais 48

V.2.3 Gemini 49

V.2.4 GPT-4 56

V.3 Discussão Integrada: Ferramentas Estáticas vs. LLMs 62

V.4 Ameaças à Validade 64

VI Conclusões **65**

Referências 68

A Análise do Slither — Ether Lotto (Time Manipulation) **74**

B Contratos com Vulnerabilidade em Time Manipulation **76**

C Análise do FibonacciBalance.sol **78**

D Correção de Contagem em Denial of Service **80**

Lista de Figuras

II.1 Cadeia de transações do *blockchain*, assegurando a validade das transações [Nakamoto, 2008] 4

IV.1 Fluxo geral da metodologia do Primeiro Experimento — Análise Estática com Mythril e Slither. 27

IV.2 Fluxo geral da metodologia do Segundo Experimento — Análise LLMs. 28

Lista de Tabelas

II.1 Vulnerabilidades do SWC Registry 8

II.2 Correlação do DASP vs SWC [Salzano et al., 2026] 11

III.1 Comparação entre os trabalhos relacionados de análise estática/simbólica e o presente estudo 22

III.2 Comparação entre trabalhos relacionados e o presente estudo 26

V.1 Análise com Mythril versão 0.24.7 dos 69 arquivos Solidity validados do SmartBugs Durieux et al. [2020] 38

V.2 Análise com Slither versão 0.10.0 dos 69 arquivos Solidity validados do SmartBugs [Durieux et al., 2020] 39

V.3 Contagem de detecção das análises do SmartBugs [Durieux et al., 2020] comparadas com as deste trabalho 40

V.4 Análise baseada no trabalho de Durieux et al. [2020]. 41

V.5 Análise de diferença das detecções Mythril. 41

V.6 Análise de diferença das detecções Slither. 41

V.7 Tabela de precisão das ferramentas (Valores arredondados para duas casas decimais). 43

V.8 Tabela de cobertura das ferramentas (valores arredondados para duas casas decimais). 43

V.9 Tabela de F1-score das ferramentas (Valores arredondados para duas casas decimais). 43

V.10 Comparação da Análise Binária: Gemini 2.0 vs. GPT-4 45

V.11 Comparativo da Análise Manual: Gemini 2.0 vs. GPT-4 46

V.12 F1-score com IC95% via *bootstrap* pareado por arquivo (10.000 reamostras, *seed*=42). Asterisco indica $p < 0{,}05$ para H_1: GPT-4 > Gemini. 48

V.13 Síntese comparativa de F1-score (%) entre a melhor ferramenta estática por categoria (Primeiro Experimento, 69 contratos) e os Grandes Modelos de Linguagem (LLMs) avaliados na análise manual (Segundo Experimento, 142 contratos). 63

Lista de Abreviações

CVE *Common Vulnerabilities and Exposures*. 19, 24

CWE *Common Weakness Enumeration*. 8

DAO *Decentralised Autonomous Organisation*. 1, 2

DASP *Decentralized Application Security Project*. 2, 4, 10, 11, 17, 23, 24, 26, 33, 44

DeFi *Decentralized Finance*. 25, 26

GPT *Generative Pre-trained Transformer*. 2, 12, 44

GSVD *Generalized Smart Contract Vulnerability Dataset*. 20

IR *Intermediate Representation*. 20

LLaMA *Large Language Model Meta AI*. 12

LLMs Grandes Modelos de Linguagem. 2–4, 12, 14, 16, 23–28, 33, 36, 37, 48–51, 62–65, 67

NLP *Natural Language Processing*. 12

PaLM *Pathways Language Model*. 12

RLHF *Reinforcement Learning with Human Feedback*. 12, 13

ROC *Receiver Operating Characteristic*. 18

SWC *Smart Contract Weakness Classification*. 4, 7, 8, 10, 30

Capítulo I Introdução

Uma *blockchain* é um livro-razão distribuído que gerencia ativos entre usuários [Maesa and Mori, 2020]. Essa tecnologia foi apresentada pela primeira vez por Satoshi Nakamoto [2008], com o objetivo de resolver o problema de estabelecer confiança em um sistema distribuído. Mais especificamente, trata-se de criar 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 os 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 utilizada em qualquer sistema que exija decisões descentralizadas, robustas e confiáveis [Belotti et al., 2019]. Na *blockchain*, utilizam-se os contratos inteligentes, programas que codificam as regras para a transferência de ativos. As transferências ocorrem dentro de transações armazenadas na *blockchain* de maneira persistente.

Uma das plataformas mais populares que suporta contratos inteligentes é o Ethereum [Buterin et al., 2014]. No Ethereum, os contratos inteligentes são frequentemente escritos em Solidity, uma linguagem influenciada pelo JavaScript [Durieux et al., 2020]. Para a execução de operações em um contrato inteligente, é necessária uma taxa denominada *gas*. Esses contratos 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, ocorreu o ataque *Decentralised Autonomous Organisation* (DAO), um dos episódios mais emblemáticos da história do Ethereum, que resultou na perda de 60 milhões de dólares devido a contratos inteligentes vulneráveis [Liao et al., 2019]. Mais recentemente, em julho de 2025, o protocolo GMX, uma das maiores plataformas de negociação de derivativos descentralizados construída sobre Solidity e implantada na rede Arbitrum (*Layer 2* do Ethereum), sofreu um ataque de reentrância em seu contrato *PositionManager*, resultando em perdas de aproximadamente 42 milhões de dólares [Sherlock and Panprog, 2026]. Em novembro do mesmo ano, o protocolo Balancer V2, uma das maiores plataformas de liquidez descentralizada implantada na rede Ethereum, foi explorado por meio de um erro de precisão na divisão inteira de

Solidity em seus *Composable Stable Pools*, resultando em perdas superiores a 121 milhões de dólares [OpenZeppelin, 2025]. Esses incidentes evidenciam que, quase uma década após o ataque DAO original, classes distintas de vulnerabilidade, de reentrância a erros aritméticos, continuam comprometendo protocolos Solidity, reforçando a necessidade de abordagens complementares de detecção automatizada.

Diante desse cenário, o objetivo deste trabalho é analisar, por meio de dois experimentos, o desempenho de ferramentas de análise estática tradicionais e de Grandes Modelos de Linguagem (LLMs) na detecção de vulnerabilidades em contratos inteligentes.

Este trabalho se diferencia da literatura existente em três pontos. Primeiro, diferentemente de Durieux et al. [2020], que avaliou o Mythril e o Slither em um único momento, este estudo mede a evolução dessas ferramentas entre versões, mostrando que os ganhos de precisão não são uniformes: categorias como *Arithmetic* apresentaram melhora expressiva, enquanto *Front Running* e *Unchecked Low-Level Calls* apresentaram regressão na cobertura. Segundo, diferentemente de Chen et al. [2025], restrito a modelos pagos da OpenAI, este trabalho mostra que um modelo com nível de acesso gratuito (Gemini 2.0 Flash) atinge desempenho estatisticamente equivalente ao GPT-4 na maioria das categorias avaliadas, com significância verificada por bootstrap pareado, resultado relevante para pesquisadores e equipes sem orçamento para APIs comerciais. Terceiro, a auditoria manual de 139 relatórios por modelo permitiu caracterizar padrões recorrentes de erro (como a dependência de indícios sintáticos em detrimento da lógica de negócio do contrato), contribuição qualitativa ausente nos trabalhos comparáveis.

No primeiro experimento, avaliou-se a capacidade de detecção de vulnerabilidades em contratos inteligentes escritos em Solidity. Para isso, utilizou-se um conjunto de 69 contratos do conjunto de dados SmartBugs Curated [Durieux et al., 2020], classificados segundo a taxonomia *Decentralized Application Security Project* (DASP). As ferramentas foram executadas em duas versões distintas de cada ferramenta, e seus resultados comparados com classificações de vulnerabilidades previamente validadas manualmente, permitindo caracterizar a evolução da precisão e da cobertura entre versões.

No segundo experimento, investigou-se o desempenho do Gemini 2.0 Flash, modelo com nível de acesso gratuito, na detecção das mesmas categorias de vulnerabilidade, utilizando o mesmo conjunto de dados e a mesma taxonomia do primeiro experimento. A avaliação foi conduzida por meio da replicação da metodologia proposta por Chen et al. [2025], que avaliou o *Generative Pre-trained Transformer* (GPT) na mesma tarefa, permitindo uma comparação direta entre os dois modelos. Os resultados foram obtidos a partir da normalização semântica das respostas e da aplicação de métricas quantitativas, complementadas por uma auditoria manual dos relatórios gerados, com o objetivo de identificar padrões recorrentes de erro e compreender as limitações de cada modelo na interpretação do código Solidity.

Os experimentos foram orientados por algumas questões de pesquisa. O primeiro experimento foi orientado pela seguinte questão de pesquisa: (RQ0) houve evolução na precisão e na cobertura de detecção do Mythril e do Slither entre as versões avaliadas? O segundo experimento foi orientado por três questões de pesquisa: (RQ1) qual é a diferença de desempenho entre Gemini 2.0 Flash e GPT-4 na detecção de vulnerabilidades em contratos inteligentes, (RQ2) qual é o impacto na avaliação de desempenho ao considerar a presença da vulnerabilidade no arquivo em vez de sua localização exata na linha de código e (RQ3) se os falsos positivos e falsos negativos produzidos pelos modelos podem ser explicados por padrões recorrentes de erro.

Os resultados obtidos mostram que as versões mais recentes das ferramentas de análise estática apresentaram avanços concretos, especialmente na redução de falsos positivos em categorias como *Arithmetic* e *Access Control*, embora com regressões em *Front Running* e *Unchecked Low-Level Calls*. Na comparação entre LLMs, o Gemini 2.0 Flash atingiu desempenho estatisticamente equivalente ao GPT-4 na maioria das categorias avaliadas, com vantagem significativa do GPT-4 restrita a quatro categorias. A auditoria manual dos relatórios gerados revelou um padrão de erro estrutural comum a ambos os modelos: a dependência de indícios sintáticos em detrimento da análise semântica da lógica de negócio dos contratos.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica; o Capítulo 3 revisa os trabalhos relacionados; o Capítulo 4 descreve a metodologia; o Capítulo 5 discute os resultados; e o Capítulo 6 apresenta as conclusões. Por fim, os apêndices apresentam análises complementares dos resultados obtidos.

Capítulo II Fundamentação Teórica

Nesta seção, serão apresentados os conceitos necessários para a compreensão da pesquisa. Primeiramente, serão abordados os fundamentos da *blockchain*; em seguida, serão apresentados o Ethereum, os conceitos de *Static Analysis* e *Symbolic Execution*. Posteriormente, serão descritas as ferramentas Mythril e Slither, seguidas das taxonomias de classificação de vulnerabilidades *Smart Contract Weakness Classification* (SWC) e DASP. Por fim, serão discutidos conceitos gerais sobre LLMs, Gemini e o GPT.

II.1 Blockchain

As criptomoedas tornaram-se um grande ativo financeiro, dentre as quais a mais famosa é o Bitcoin [Nofer et al., 2017]. A principal tecnologia que estrutura o Bitcoin é o *blockchain*, proposto pela primeira vez em 2008 e implementado em 2009 [Nakamoto, 2008]. O *blockchain* pode ser considerado um livro-razão público, onde todas as transações confirmadas são armazenadas em uma lista de blocos. Essa lista de blocos cresce à medida que novos blocos são acrescentados continuamente [Zheng et al., 2017]. A tecnologia *blockchain* geralmente apresenta características-chave, como descentralização, persistência, anonimato e capacidade de auditoria [Zheng et al., 2017].

A Figura II.1 ilustra a estrutura básica do *blockchain*, formada por uma cadeia de assinaturas digitais encadeadas.

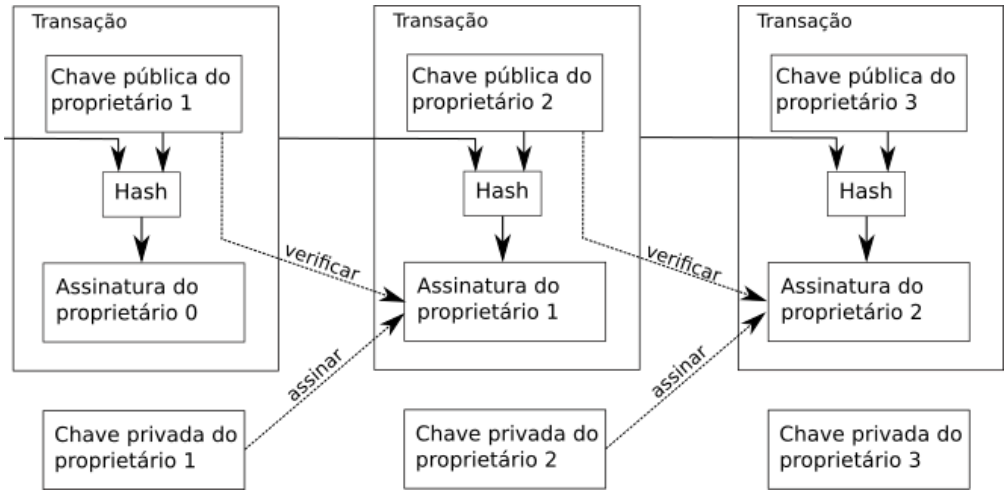


Figura II.1: Cadeia de transações do *blockchain*, assegurando a validade das transações [Nakamoto, 2008]

Na imagem, é possível observar o encadeamento dos blocos, gerado por uma transação anterior que sofre *hash* junto com a chave pública do receptor de um valor, por exemplo. Os clientes transferem uma moeda para o próximo cliente assinando a *hash* da transação anterior e a chave pública do próximo proprietário. Assim, um beneficiário pode verificar as assinaturas e confirmar se aquela cadeia é válida. Dessa forma, a cada transação há um acúmulo das anteriores, sendo este um dos principais pontos de segurança do *blockchain*.

II.2 Ethereum

Ethereum é uma rede *blockchain*, introduzida em 2014 e lançada em 2015 [Buterin et al., 2014]. Ao contrário das gerações anteriores de linguagens de programação de *blockchain*, como o *script* do Bitcoin, a linguagem do Ethereum é Turing completa [Buterin et al., 2013], capaz de expressar lógica com maior complexidade. No Ethereum, os desenvolvedores escrevem contratos em linguagens de alto nível, a mais popular é uma linguagem semelhante ao JavaScript, chamada Solidity, que é compilada em *bytecode* para a *Ethereum Virtual Machine* (EVM), uma *virtual machine* (VM) baseada em pilha, operando com palavras de 256 bits [Bhargavan et al., 2016] [Tikhomirov et al., 2018].

Para realizar operações dentro da rede Ethereum, como a mineração, é necessário um mecanismo para a transferência de valor. O Ethereum possui uma moeda interna, o Ether (ETH), cuja menor subdenominação é chamada Wei, sendo esta a unidade mínima de contagem de valor. Um Ether é definido como 10^{18} Weis [Wood et al., 2014]. O protocolo do Ethereum especifica um custo para cada operação na EVM, denominado *gas*. Os usuários pagam antecipadamente pela quantidade estimada de *gas* necessária para que a transação ocorra. Caso a transação seja bem-sucedida, o usuário recebe um reembolso parcial correspondente ao *gas* efetivamente consumido. Entretanto, se ocorrer uma exceção por falta de *gas*, todas as alterações no estado são revertidas, mas o valor do *gas* não é devolvido ao usuário. O preço do Ether por unidade de *gas* é determinado pelo mercado [Tikhomirov et al., 2018].

II.3 Static Analysis

A análise de segurança de código-fonte usando ferramentas de *static analysis* ajuda a detectar problemas de segurança examinando o código em si, sem precisar executar o programa. Essas ferramentas identificam automaticamente erros de programação comuns relacionados à segurança. Essa abordagem é mais rápida e abrangente do que a revisão manual de código, pois examina o código sem executá-lo. Ao contrário do *dynamic testing*, que necessita rodar o programa, a *static analysis* pode ser realizada antes mesmo do software estar completo [Chess and McGraw, 2004].

Ferramentas de *static analysis* examinam diretamente o código, identificando problemas de segurança de forma consistente e sem viés humano. Diferentemente dos testes dinâmicos, que podem apenas revelar sintomas, as ferramentas de *static analysis* permitem identificar a causa raiz dos problemas de segurança, conduzindo a correções mais adequadas [Chess and West, 2007].

Além disso, possibilitam detecção e *feedback* precoces: erros podem ser identificados antes mesmo da execução do programa, reduzindo custos de correção e fornecendo orientações rápidas aos programadores. Também funcionam como ferramentas de transferência de conhecimento, apresentando cenários de ataque e construções inseguras de código. Outro benefício é a possibilidade de reavaliar grandes bases de código com novos métodos de ataque recém-descobertos, tornando-as especialmente úteis para revisão de código legado.

Contudo, as ferramentas de *static analysis* também possuem desvantagens. Elas frequentemente produzem falsos positivos, alertas para problemas inexistentes. Isso pode levar à análise de avisos irrelevantes e até à negligência de vulnerabilidades reais [Chess and West, 2007].

II.4 Symbolic Execution

A *symbolic execution* é uma técnica utilizada para analisar programas explorando múltiplos caminhos de execução simultaneamente [Baldoni et al., 2018]. Em vez de utilizar valores concretos para entradas, são usadas variáveis simbólicas. Dessa forma, são geradas expressões condicionais, que são resolvidas por meio de *SMT solvers* para determinar valores que podem levar a falhas. Ao representar entradas como variáveis simbólicas, é possível identificar vulnerabilidades potenciais de forma mais ampla [Tsankov et al., 2018].

A *symbolic execution* é aplicada principalmente em testes de falha de software e segurança. Um dos principais desafios dessa técnica está no fenômeno conhecido como *path explosion* [Baldoni et al., 2018]. Cada bifurcação no código duplica o número de caminhos explorados, fazendo com que a quantidade de execuções cresça exponencialmente. Resolver expressões simbólicas complexas também pode ser computacionalmente custoso, especialmente no contexto de *blockchain* [Tsankov et al., 2018]. Para melhorar o desempenho, pode ser necessário reduzir o número de caminhos explorados, mas isso aumenta o risco de falsos positivos [Baldoni et al., 2018].

II.5 Slither

O Slither é um *framework* de *static analysis* projetado para fornecer informações detalhadas sobre contratos inteligentes no ecossistema Ethereum. É uma das ferramentas de *static analysis* mais amplamente consolidadas na literatura recente, sendo identificada entre as principais soluções de segurança adotadas entre 2020 e 2024 [Mishra and Phansalkar, 2025]. A ferramenta é implementada em Python 3 com aproximadamente 16 mil linhas de código. O seu funcionamento baseia-se na

conversão de contratos Solidity para uma representação intermediária chamada SlithIR [Feist et al., 2019]. Esta representação utiliza a forma de *Static Single Assignment* (SSA) com um conjunto reduzido de instruções. Essa abordagem facilita a implementação de análises complexas. Além disso, ela preserva informações semânticas que poderiam ser perdidas na conversão direta para *bytecode*. O Slither permite aplicar técnicas clássicas de análise de programas, como o rastreamento de contaminação e a análise de fluxo de dados.

Atualmente, o *framework* desempenha funções fundamentais na auditoria de contratos inteligentes. Ele permite a detecção automatizada de diversas vulnerabilidades sem a necessidade de intervenção manual. A ferramenta também identifica otimizações de código que os compiladores convencionais não conseguem perceber. Além disso, o Slither auxilia na compreensão da lógica do contrato ao gerar *logs* de resumo detalhados. Por fim, a sua API dedicada possibilita a realização de revisões de código assistidas de forma integrada.

II.6 Mythril

O Mythril é uma ferramenta de análise de segurança para contratos inteligentes na Ethereum. Desenvolvido pela ConsenSys, 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 encontrar valores que permitam explorar vulnerabilidades em contratos inteligentes [Durieux et al., 2020]. É um projeto de código aberto, baseado em console e em desenvolvimento ativo. Seu mecanismo de análise identifica diversas classes de vulnerabilidades, utilizando a máquina virtual simbólica LASER, um módulo do próprio Mythril, para execução simbólica de código [Prechtel et al., 2019]. O Mythril também é uma das mais populares ferramentas da sua categoria [Zheng et al., 2022]. A ferramenta figura entre as principais soluções de detecção de vulnerabilidades consolidadas na literatura entre 2020 e 2024, ao lado do Slither [Mishra and Phansalkar, 2025].

A máquina virtual LASER está intimamente integrada ao Mythril. Em vez de trabalhar com o *bytecode* bruto do Ethereum, utiliza o *disassembly* gerado pelo Mythril, que preserva as mesmas instruções do *bytecode* nativo, mas fornece informações adicionais para a análise. Esse *disassembly* pode ser obtido tanto a partir de código Solidity quanto de *bytecode* da Ethereum [Prechtel et al., 2019]. Apesar de modelar a maioria das funcionalidades da máquina virtual oficial, o LASER não aborda aspectos como a modelagem do consumo de *gas* nos contratos inteligentes.

II.7 Smart Contract Weakness Classification

A SWC é uma taxonomia padronizada criada para organizar fraquezas comuns em contratos inteligentes da plataforma Ethereum. Proposta inicialmente como *Ethereum Improvement Proposal* (EIP-1470), essa classificação tornou-se referência para auditorias e ferramentas automatizadas de

análise, como Mythril e Slither, que utilizam seus identificadores para categorizar vulnerabilidades detectadas [Zheng et al., 2024].

Os principais objetivos do SWC concentram-se em fornecer uma forma simples de classificar fraquezas, permitindo a identificação de falhas que levam a vulnerabilidades de segurança. Além disso, a taxonomia busca definir uma linguagem comum para descrever problemas de arquitetura, design e código, servindo como suporte fundamental para o desenvolvimento e operação de ferramentas de análise de segurança.

Apesar de sua ampla adoção por ferramentas como Mythril e Slither, o SWC apresenta limitações significativas [Vidal et al., 2024]. Primeiramente, a ausência de atualizações desde 2018 torna o registro desatualizado frente às vulnerabilidades modernas. Essa defasagem é evidenciada pela cobertura restrita a 37 vulnerabilidades, enquanto estudos recentes identificam mais de 90 categorias. Adicionalmente, observa-se uma nomenclatura inconsistente entre ferramentas, o que dificulta comparações, além de uma granularidade inadequada com categorias genéricas como *Denial of Service*. Por fim, a taxonomia frequentemente apresenta uma mistura de conceitos, confundindo a própria vulnerabilidade com seu impacto ou método de exploração.

Cada entrada no SWC inclui um identificador único (SWC-ID), descrição da fraqueza, exemplos de código vulnerável e corrigido, além de relação com outras taxonomias, como *Common Weakness Enumeration* (CWE). A estrutura do SWC é plana, composta por uma lista de vulnerabilidades sem hierarquia definida. Nesse contexto, cada fraqueza no SWC é formalmente documentada por meio de seu título e descrição, um identificador único (SWC-ID) e as respectivas orientações para mitigação.

A seguir, são apresentadas as principais vulnerabilidades classificadas pelo SWC Registry, conforme detalhado na Tabela II.1:

Tabela II.1: Vulnerabilidades do SWC Registry

ID	**Vulnerabilidade**	**Descrição Resumida**
SWC-100	Function Default Visibility	Funções sem visibilidade definida podem ser acessadas por qualquer pessoa.
SWC-101	Integer Overflow and Underflow	Erros aritméticos que ultrapassam os limites numéricos do tipo de dado.
SWC-102	Outdated Compiler Version	O uso de versões antigas do compilador expõe o contrato a bugs já conhecidos.
SWC-103	Floating Pragma	Não travar a versão do compilador pode causar comportamentos inconsistentes.

Continua na próxima página...

Continuação da Tabela II.1

ID	Vulnerabilidade	Descrição Resumida
SWC-104	Unchecked Call Return Value	Ignorar o retorno de chamadas externas pode esconder falhas de execução.
SWC-105	Unprotected Ether Withdrawal	Falta de controle de acesso em funções que enviam fundos (Ether).
SWC-106	Unprotected SELFDESTRUCT	Permite que atacantes destruam o contrato permanentemente.
SWC-107	Reentrancy	Chamadas externas que permitem reentrada antes da atualização do saldo.
SWC-108	State Variable Default Visibility	Variáveis de estado sem visibilidade explícita podem ser lidas indevidamente.
SWC-109	Uninitialized Storage Pointer	Ponteiros vazios que podem sobrescrever dados críticos no armazenamento.
SWC-110	Assert Violation	O uso de `assert()` para condições que podem falhar por entrada do usuário.
SWC-112	Delegatecall to Untrusted Callee	Executa código de terceiros no contexto do seu próprio contrato.
SWC-113	DoS with Failed Call	Uma falha em uma chamada externa trava toda a execução do contrato.
SWC-115	Authorization through tx.origin	Uso de `tx.origin` torna o contrato vulnerável a ataques de phishing.
SWC-116	Block Values as a Proxy for Time	Uso de dados do bloco para tempo, que podem ser levemente manipulados.
SWC-120	Weak Sources of Randomness	Uso de variáveis previsíveis (como hashes de bloco) para sorteios.
SWC-124	Write to Arbitrary Storage Location	Permite que um atacante altere qualquer slot de memória do contrato.
SWC-135	Code With No Effects	Código morto que consome Gas e pode indicar erros de lógica.

II.8 Decentralized Application Security Project

No ecossistema de segurança de contratos inteligentes, o DASP[1] consolidou-se como uma das taxonomias mais influentes para a classificação de falhas de segurança em contratos Solidity. Diferente de registros mais granulares e técnicos, como o SWC Registry, o DASP foca em dez categorias principais (Top 10) que refletem os problemas mais críticos e recorrentes enfrentados pela comunidade de desenvolvedores [Salzano et al., 2026]. Neste trabalho, o DASP é utilizado como ferramenta fundamental para a classificação final das vulnerabilidades, seguindo a abordagem das obras que serviram de base para esta pesquisa. Em outros textos de referência, ele também é citado como taxonomia participante.

A seguir, apresentam-se as vulnerabilidades que compõem o DASP Top 10, conforme detalhado no estudo empírico de Salzano et al. [2026]:

- **Reentrância (*Reentrancy*):** Ocorre quando um contrato de destino realiza uma chamada recursiva ao contrato de origem antes que este tenha finalizado a atualização de seu estado interno. Essa inconsistência temporária permite que atacantes drenem fundos repetidamente.
- **Controle de Acesso (*Access Control*):** Refere-se à ausência de autorizações adequadas em funções sensíveis. Falhas nesta categoria permitem que usuários mal-intencionados acessem valores privados ou executem instruções críticas, como o `SELFDESTRUCT`, sem permissão.
- **Aritmética (*Arithmetic*):** Envolve erros de *overflow* e *underflow* em variáveis de dimensões fixas. Quando uma operação ultrapassa o limite máximo ou mínimo de um tipo numérico, o valor retorna ao início da escala, comprometendo a lógica financeira do contrato.
- **Chamadas Não Verificadas (*Unchecked Low Level Calls*):** O Solidity oferece chamadas de baixo nível, como `call()`, que não propagam erros automaticamente. A falha em verificar explicitamente o retorno booleano pode fazer com que a execução prossiga sob a falsa premissa de sucesso.
- **Negação de Serviço (*Denial of Service - DoS*):** Esta falha torna o contrato inoperante. Exemplos comuns incluem o estouro do limite de gás do bloco através de loops excessivamente grandes ou falhas intencionais em funções de transferência.
- **Aleatoriedade Ruim (*Bad Randomness*):** Devido à natureza determinística do *blockchain*, gerar números aleatórios verdadeiros é um desafio. O uso de atributos previsíveis da rede (como o *hash* do bloco) permite que atacantes antecipem e manipulem resultados.

[1] NCC Group. *Decentralized Application Security Project (DASP) Top 10.* Disponível em: `https://web.archive.org/web/20250113205456/https://www.dasp.co/`. Acesso em: 21 jun. 2026. Página original (`https://dasp.co/`) atualmente indisponível.

- **Antecipação de Transações (*Front Running*):** Baseia-se na dependência da ordem das transações no *mempool*. Atacantes podem observar transações pendentes e enviar transações próprias com taxas de *gas* superiores para garantir prioridade de execução.

- **Manipulação de Tempo (*Time Manipulation*):** Ocorre quando decisões lógicas dependem do `block.timestamp`. Como este valor é fornecido pelos mineradores, ele pode ser ligeiramente manipulado para satisfazer condições temporais específicas.

- **Endereço Curto (*Short Address*):** Relacionada à forma como o Solidity preenche argumentos curtos para 32 bytes. Um atacante pode enviar dados malformados, induzindo o contrato a ler mais dados do que o pretendido.

- **Desconhecidos (*Unknowns*):** Esta categoria abrange novas vulnerabilidades que ainda não foram catalogadas formalmente, reconhecendo a natureza evolutiva das ameaças em sistemas descentralizados.

A Tabela II.2 apresenta a correlação entre as vulnerabilidades listadas pelo DASP e o padrão SWC , conforme detalhado no trabalho de Salzano et al. [2026].

Tabela II.2: Correlação do DASP vs SWC [Salzano et al., 2026]

ID DASP	**Nome (DASP)**	**ID SWC**	**Nome (SWC)**
DASP-01	Reentrancy	SWC-107	Reentrancy
DASP-02	Access Control	SWC-105	Unprotected Ether Withdrawal
DASP-02	Access Control	SWC-106	Unprotected SELFDESTRUCT Instruction
DASP-03	Arithmetic	SWC-101	Integer Overflow and Underflow
DASP-04	Unchecked Low-Level Calls	SWC-104	Unchecked Call Return Value
DASP-04	Unchecked Low-Level Calls	SWC-123	Requirement Violation
DASP-05	Denial of Service	SWC-113	DoS with Failed Call
DASP-05	Denial of Service	SWC-128	DoS with Block Gas Limit
DASP-06	Bad Randomness	SWC-120	Weak Sources of Randomness
DASP-07	Front Running	SWC-114	Transaction Order Dependence
DASP-08	Time Manipulation	SWC-116	Timestamp Dependence
DASP-09	Short Address Attack	SWC-102	Outdated Compiler Version*
DASP-10	Unknowns	N/A	N/A

* O mapeamento entre *Short Address Attack* e SWC-102 (Outdated Compiler Version) é baseado em [Salzano et al., 2026], embora a correspondência semântica seja indireta. O SWC Registry não possui uma entrada específica para esse tipo de ataque.

II.9 Grandes Modelos de Linguagem

Os LLMs são modelos avançados de linguagem baseados na arquitetura *Transformer*, com bilhões ou até trilhões de parâmetros, treinados em grandes conjuntos de dados textuais; essa escala é central para sua capacidade de compreender e gerar texto de forma ampla [Zhao et al., 2023; Minaee et al., 2024]. Historicamente, os LLMs resultam da evolução de décadas em modelagem de linguagem, passando por modelos estatísticos, redes neurais e, mais recentemente, transformadores. A literatura descreve quatro gerações principais: *Small Language Models* (SLMs), *Neural Language Models* (NLMs), *Pre-trained Language Models* (PLMs) e LLMs [Zhao et al., 2023].

Um traço distintivo dos LLMs é o surgimento de habilidades emergentes com o aumento da escala, como *in-context learning*, seguimento de instruções e raciocínio em múltiplas etapas, expandindo o escopo de uso para além das tarefas clássicas de *Natural Language Processing* (NLP). Essas capacidades permitem empregar LLMs como base de agentes de IA que interagem com usuários, acessam ferramentas externas e incorporam *feedback* para melhoria contínua [Zhao et al., 2023; Minaee et al., 2024].

Tecnicamente, a arquitetura *Transformer* e o mecanismo de *self-attention* viabilizam o processamento eficiente de sequências longas e contextuais, sustentando famílias influentes como GPT, *Pathways Language Model* (PaLM) e *Large Language Model Meta AI* (LLaMA). O desempenho e a utilidade prática são ampliados por técnicas como ajuste fino, alinhamento com preferências humanas, por exemplo, via *Reinforcement Learning with Human Feedback* (RLHF), e integração com bases externas e ferramentas, expandindo a capacidade dos modelos para resolver tarefas complexas e incorporar conhecimento atualizado [Zhao et al., 2023; Minaee et al., 2024].

Na prática, os LLMs já são aplicados em assistentes virtuais, tradução automática, geração de código, apoio educacional e pesquisa científica, além de compor agentes autônomos de propósito geral [Minaee et al., 2024]. Apesar do impacto, persistem desafios, como alto custo computacional, dependência de dados de qualidade, viés, segurança e questões éticas de alinhamento, temas centrais para uma evolução responsável e segura desses sistemas [Zhao et al., 2023; Minaee et al., 2024].

II.9.1 GPT-4

O GPT-4 é um modelo de linguagem de grande escala desenvolvido pela OpenAI, pertencente à família de modelos generativos baseados na arquitetura *Transformer*. Esse tipo de modelo é treinado utilizando grandes volumes de dados textuais com o objetivo de prever o próximo token em uma sequência, permitindo a geração de texto coerente, interpretação de linguagem natural e execução de diversas tarefas relacionadas ao processamento de linguagem natural [OpenAI, 2023; Achiam et al., 2023].

Em comparação com versões anteriores, como o GPT-3.5, o GPT-4 apresenta avanços significativos em capacidade de raciocínio, interpretação de instruções complexas e geração de respostas mais precisas e contextualizadas. De acordo com o relatório técnico do modelo, o GPT-4 demonstrou desempenho elevado em diversos *benchmarks* acadêmicos e profissionais, alcançando resultados comparáveis ou superiores ao desempenho humano em alguns exames padronizados, como o Uniform Bar Exam, no qual o modelo atingiu pontuação próxima aos melhores participantes [Achiam et al., 2023]. Esses avanços são resultados de melhorias no processo de treinamento e alinhamento do modelo, incluindo técnicas de aprendizado com *feedback* humano RLHF, que visam tornar as respostas mais úteis, seguras e alinhadas às preferências humanas [OpenAI, 2023; Achiam et al., 2023].

Outra característica relevante do GPT-4 é sua natureza multimodal, permitindo o processamento de diferentes tipos de entrada, incluindo texto e imagens, com geração de saídas textuais. Essa capacidade amplia o escopo de aplicações do modelo em áreas como análise de documentos, programação assistida, educação e suporte à tomada de decisão baseada em linguagem natural [Achiam et al., 2023]. O relatório técnico também destaca estudos sobre *predictable scaling*, indicando que o desempenho do modelo pode ser estimado a partir do comportamento de modelos menores durante o processo de desenvolvimento, o que auxilia no planejamento e otimização do treinamento em larga escala [Achiam et al., 2023].

Apesar dos avanços, o modelo ainda apresenta limitações, como a possibilidade de gerar informações incorretas ou imprecisas, fenômeno frequentemente denominado *hallucination*. Além disso, o modelo pode apresentar dificuldades em cenários altamente especializados ou em tarefas que exigem raciocínio profundo em múltiplas etapas. Dessa forma, recomenda-se a supervisão humana e a validação das respostas em aplicações críticas ou de alto impacto [OpenAI, 2023; Achiam et al., 2023].

II.9.2 Gemini 2.0 Flash

O Gemini 2.0 Flash representa uma evolução significativa na linha de modelos multimodais da Google, sendo especificamente otimizado para aplicações agênticas que exigem baixa latência e alto desempenho. Conforme a documentação técnica oficial da Google Cloud [2026], a infraestrutura do modelo sustenta-se em uma arquitetura multimodal nativa, permitindo o processamento integrado de entradas de texto, áudio, vídeo e imagem sem a necessidade de codificadores externos ou modelos intermediários de tradução de modalidade. Além disso, o Gemini 2.0 Flash destaca-se por suportar uma janela de contexto de até 1 milhão de *tokens*, o que viabiliza a análise de volumes extensos de dados, bases de código complexas ou sessões prolongadas de interação com alto índice de recuperação de informações. Sua capacidade agêntica é reforçada pelo suporte nativo ao uso de ferramentas,

permitindo que o modelo invoque autonomamente serviços como pesquisa na web e execução de código, tornando-o uma solução robusta para cenários que demandam respostas em tempo real com alta precisão no seguimento de instruções complexas.

Essa robustez técnica não se limita à eficiência operacional, mas fundamenta um novo paradigma de interação homem-máquina, frequentemente referido pela Google DeepMind como a "era agêntica" [Google DeepMind, 2024]. Esta abordagem é demonstrada através de protótipos de investigação, como o Project Astra, que explora a multimodalidade em tempo real para criar assistentes universais, e o Project Mariner, focado na navegação autônoma em ambientes de navegador para a execução de tarefas complexas. Estes desenvolvimentos sublinham a capacidade do modelo em realizar o planejamento de múltiplas etapas, fundamental para a transição de sistemas de linguagem estáticos para agentes autônomos capazes de interagir com o mundo real sob supervisão humana.

II.10 Métricas de Avaliação

Para quantificar o desempenho das ferramentas de análise estática e dos LLMs na detecção de vulnerabilidades, este trabalho adota três métricas consolidadas na literatura de classificação binária [Powers, 2020]: precisão, cobertura (*recall*) e *F1-score*. O cálculo dessas métricas parte da contagem de Verdadeiros Positivos (TP), Falsos Positivos (FP) e Falsos Negativos (FN), definidos formalmente no Capítulo IV para o contexto específico deste estudo.

A precisão mede a proporção de detecções realizadas pelo modelo que de fato correspondem a vulnerabilidades reais, penalizando o excesso de falsos positivos:

$$\text{Precisão} = \frac{TP}{TP + FP} \tag{II.1}$$

A cobertura, por sua vez, mede a proporção de vulnerabilidades reais que o modelo foi capaz de identificar, penalizando a omissão de vulnerabilidades existentes (falsos negativos):

$$\text{Cobertura} = \frac{TP}{TP + FN} \tag{II.2}$$

Como precisão e cobertura frequentemente apresentam *trade-off* entre si, o *F1-score* é utilizado como medida de síntese, correspondendo à média harmônica entre as duas métricas:

$$F_1 = 2 \times \frac{\text{Precisão} \times \text{Cobertura}}{\text{Precisão} + \text{Cobertura}} \tag{II.3}$$

Capítulo III Trabalhos Relacionados

III.1 Método de Pesquisa

Neste capítulo, são apresentados os trabalhos relacionados aos métodos e às tecnologias voltados à análise de ferramentas de detecção de vulnerabilidades, com foco em soluções automatizadas de análise de código-fonte [Chess and McGraw, 2004]. Foram selecionados estudos que incluem experimentos avaliando as duas principais ferramentas de análise automatizada de contratos inteligentes.

Este trabalho teve como base inicial uma linha de pesquisa envolvendo *machine learning*. Essa abordagem, à época, impunha como requisito fundamental a disponibilidade de uma grande quantidade de contratos verificados, uma vez que tais dados constituem a base para o treinamento dos modelos. Contudo, mesmo nos trabalhos relacionados, observou-se uma dificuldade recorrente em encontrar conjuntos de dados com volume e qualidade suficientes para justificar a viabilidade do treinamento.

Diante desse cenário, nesta etapa foi aplicada a metodologia de *Systematic Literature Review* (SLR) proposta por Kitchenham and Charters [2007], combinada com a estratégia híbrida de busca, associando pesquisa em base de dados com *snowballing* bidirecional (para trás e para frente) como em Mourão et al. [2020], seguindo a mesma condução metodológica adotada por Neto et al. [2021]. Tal metodologia permite construir uma visão abrangente do estado da arte, funcionando como uma fotografia dos estudos atuais sobre gerenciamento de risco e identificando as principais discussões e tendências da área.

No contexto da pesquisa em *machine learning*, optou-se inicialmente por utilizar termos de busca amplos, com o objetivo de maximizar a recuperação de trabalhos relacionados a essa abordagem. Entretanto, não foi possível dar continuidade a essa linha de investigação devido à dificuldade em encontrar conjuntos de dados extensos contendo código verificado. Ainda assim, durante a busca pelos principais repositórios de dados, identificou-se um trabalho de maior relevância, apresentado por Durieux et al. [2020]. Embora esse estudo disponibilize um conjunto de dados interessante, a quantidade de arquivos ainda se mostrou insuficiente para aplicações robustas de *machine learning*.

Apesar dessa limitação, o trabalho de Durieux et al. [2020] reforça a importância das abordagens tradicionais de análise de código. Os autores destacam ferramentas de análise estática como principais representantes dessa categoria, apontando o Slither como uma das ferramentas mais relevantes,

além de enfatizarem sua combinação com o Mythril para uma análise mais abrangente de contratos inteligentes.

Em razão da dificuldade em obter conjuntos de dados verificados em larga escala, a estratégia deste trabalho foi reformulada. No primeiro experimento, o foco passou a ser a análise das ferramentas tradicionais de detecção de vulnerabilidades. Inicialmente, realizou-se uma busca aberta com o objetivo de compreender a amplitude dos temas relacionados. Para isso, utilizou-se o Google Scholar, devido à sua capacidade de agregar trabalhos provenientes de diferentes repositórios.

Manteve-se a estratégia de empregar consultas amplas em um primeiro momento, restringindo-as progressivamente à medida que os resultados eram analisados. Contudo, devido ao elevado volume de resultados retornados pelo Google Scholar, tornou-se necessário o desenvolvimento de um programa em Python[1] para coletar e organizar os dados em uma planilha, contemplando informações como ano de publicação, resumo e número de citações.

As pesquisas foram realizadas por meio de um *script* de *scraping*, utilizado para extrair dados do Google Scholar, uma vez que a plataforma não disponibiliza um mecanismo de ranqueamento estruturado dos resultados. A escolha do Google Scholar teve como objetivo obter uma visão abrangente dos trabalhos mais citados sobre o tema, incluindo publicações oriundas de diferentes repositórios acadêmicos.

O uso do Python possibilitou a extração, organização e classificação sistemática dos artigos recuperados. Nessa etapa inicial, foi conduzida uma análise baseada na relevância e na aderência temática dos trabalhos mais citados, permitindo a identificação das contribuições de maior impacto na área de estudo. Por fim foram aplicadas também tecnicas como *snowballing* para buscar o máximo de artigos.

Essa abordagem permitiu identificar e organizar os trabalhos de maior relevância, especialmente com base no número de referências, facilitando a identificação dos estudos mais influentes. Como parte da busca mais abrangente, foram utilizadas expressões como (“Static Analysis”) AND (“Smart Contracts”), trazendo aproximadamente quinhentos trabalhos a serem ranqueados. Nessa busca, o próprio trabalho do Slither [Feist et al., 2019] destacou-se como um dos mais referenciados, reforçando sua importância no cenário de análise de contratos inteligentes.

Seguindo a estratégia definida para este estudo, que consiste na análise de duas ferramentas com maior representatividade na literatura, foram realizadas buscas específicas utilizando os termos (Mythril) AND (Slither) AND (“Smart Contracts”), com o objetivo de identificar trabalhos diretamente alinhados à proposta desta pesquisa.

No segundo experimento, voltado à análise por meio de LLMs, empregou-se a seguinte string de busca: (“Large Language Models” OR LLM OR ChatGPT OR GPT-4 OR Gemini OR Grok) AND

[1] `https://github.com/drinith/scrapinggooglescholar`

("smart contract" OR Solidity) AND (vulnerability OR security OR auditing OR "bug detection"). Com base na experiência adquirida no experimento anterior, optou-se por uma busca mais ampla, priorizando trabalhos de maior relevância para o tema, especialmente aqueles publicados em revistas de prestígio e com elevado número de citações. Utilizando a mesma ferramenta desenvolvida em Python apresentada anteriormente, a busca retornou um total de 397 trabalhos. Dos artigos retornados, 56 apresentavam um resumo que trazia uma possível avaliação e somente 6 artigos possuíam abordagens semelhantes com a proposta do projeto.

III.2 Análise Estática e Simbólica

O trabalho de Durieux et al. [2020] teve como principal objetivo relatar, na época, o estado da arte das ferramentas de análise automatizada disponíveis para contratos inteligentes. Para facilitar a reprodutibilidade e a comparação entre ferramentas, o estudo foi executado utilizando uma nova estrutura de execução extensível chamada SmartBugs. Nesse estudo, buscou-se responder a três questões: qual era a efetividade das ferramentas naquele momento; quais vulnerabilidades estavam presentes na *blockchain* Ethereum; e quanto tempo as ferramentas levavam para analisar os contratos inteligentes.

Com base no trabalho de Di Angelo and Salzer [2019], foi levantado um conjunto de ferramentas a serem analisadas. Das 35 ferramentas identificadas, algumas foram selecionadas com base em três critérios: disponibilidade, execução em linha de comando, aceitação de entradas em Solidity e capacidade de identificar vulnerabilidades. Assim, nove ferramentas foram escolhidas.

O estudo criou dois conjuntos de dados de contratos inteligentes em Solidity. O primeiro conjunto, curado, continha 69 contratos inteligentes, nos quais foram identificadas vulnerabilidades. O segundo conjunto, denominado *wild*, consistia em 47.518 contratos extraídos da *blockchain* Ethereum. O trabalho utilizou a taxonomia apresentada pelo DASP para descrever vulnerabilidades. Como resultado da base curada, o Mythril apresentou a maior precisão entre as nove ferramentas, detectando 27% de todas as vulnerabilidades, enquanto a média das demais ferramentas foi de 12%.

O estudo concluiu que o Mythril não é suficientemente poderoso para substituir todas as ferramentas; juntas, elas detectaram 42% das vulnerabilidades. Considerando que a combinação de todas as ferramentas pode não ser viável devido ao poder de computação necessário, propôs-se a combinação de Mythril e Slither, que detectou 37% das vulnerabilidades únicas. Assim, o trabalho indicou que as ferramentas, em conjunto, são capazes de detectar até 42% das vulnerabilidades em códigos curados. Em relação à proposta deste trabalho, este estudo indica as duas ferramentas que mais se destacaram em funcionamento conjunto, Slither e Mythril, embora não apresente um comparativo de evolução entre elas.

O artigo de Parizi et al. [2018] fornece uma avaliação empírica inicial de ferramentas de análise

estática de contratos inteligentes. O estudo teve como objetivo avaliar quatro ferramentas: Oyente, Mythril, Securify e SmartCheck, com base em sua eficácia na detecção de vulnerabilidades e na precisão. O trabalho investigou algumas questões: quão eficazes são as ferramentas automatizadas de teste de segurança de contratos inteligentes, em termos de capacidade de detecção de vulnerabilidades? Qual é a ferramenta mais eficaz? E quais são as pontuações de precisão obtidas pelas ferramentas de teste de segurança automatizadas na detecção de vulnerabilidades reais?

O experimento consistiu na seleção aleatória de uma ferramenta de teste de segurança, seguida da escolha aleatória de um contrato inteligente do conjunto de contratos. Para a análise das ferramentas, foram utilizadas duas fontes de contratos intencionalmente vulneráveis: contratos desenvolvidos pela Trail of Bits, uma empresa especializada em segurança Ethereum, e contratos do Wargame Ethernaut, um desafio online de exploração de contratos inteligentes. Foram selecionados dez contratos escritos em Solidity, contendo vulnerabilidades como *Integer Overflow*, *Missing Constructor*, *Reentrancy*, *Unchecked External Call*, *Unprotected Function*, *Wrong Interface*, *Callstack Depth Attack*, *Assertion Failure*, *Timestamp Dependency*, *Parity Multisig Bug* e *Transaction-Ordering Dependence*, entre outros.

Para avaliar a eficácia das ferramentas, utilizou-se a análise da *Receiver Operating Characteristic* (ROC). A precisão foi testada por meio dos testes estatísticos ANOVA eLSD. Os resultados indicaram que o SmartCheck se destacou estatisticamente em relação às demais ferramentas automatizadas de teste de segurança, com nível de significância de 95% ($p < 0,05$). Além disso, o Mythril apresentou precisão significativa ($p < 0,05$), emitindo o menor número de alarmes falsos entre as ferramentas avaliadas.

No artigo de Ye et al. [2019] pretendeu-se realizar uma análise criteriosa em ferramentas de análises de bugs em contratos inteligentes e resumir seus prós e contras. O projeto aplicou o mesmo conjunto de contratos inteligentes e analisou os resultados. Além disso, para compreender os mecanismos exatos de detecção das ferramentas, foi analisado o código-fonte. O artigo teve como experimento o uso de 1.010 contratos inteligentes em 200 arquivos. Todos esses arquivos foram selecionados aleatoriamente no banco de dados de contratos inteligentes do trabalho, rastreados do Etherscan. O trabalho tentou responder algumas perguntas, como: qual é a eficácia do Slither e do Oyente no conjunto de dados levantado? O que causa falsos positivos?

O artigo concluiu que o Slither e o Oyente verificam vulnerabilidades de *Reentrancy*. Contudo, o Slither detectou que 24 arquivos continham bugs de reentrada, enquanto o Oyente detectou apenas 8 arquivos. Em relação ao *bug* de *integer over/underflow*, o Oyente apresentou 546 detecções, e o trabalho classificou o Slither como não capaz de suportar esse tipo de vulnerabilidade. O projeto concluiu que o Slither apresenta altas taxas de falsos positivos para detectar *Reentrancy*. O trabalho também concluiu que, no caso do Oyente, a falta de verificação do modificador, que

permite identificar se o cliente é o proprietário, torna os resultados do Oyente não confiáveis para *Reentrancy*.

No trabalho de So et al. [2021], foi apresentado o SMARTEST, um novo analisador de segurança para contratos inteligentes do Ethereum. O trabalho afirma que a principal característica do SMARTEST, que o diferencia crucialmente dos analisadores existentes, é que ele encontra efetivamente sequências de transações vulneráveis em contratos inteligentes. Assim, o objetivo do SMARTEST não é apenas detectar bugs em contratos inteligentes, mas também gerar automaticamente sequências de transações vulneráveis que comprovem as imperfeições encontradas.

O primeiro conjunto de dados utilizado é composto por 443 contratos inteligentes com relatórios *Common Vulnerabilities and Exposures* (CVE) sobre vulnerabilidades aritméticas. O segundo conjunto consiste em 104 contratos com vulnerabilidades relacionadas ao controle de acesso, como *ether-leaking* e *suicidal contracts*. O trabalho comparou o SMARTEST com o Mythril e o Manticore. Os resultados mostram que o SMARTEST encontrou 93% das vulnerabilidades conhecidas, enquanto o Mythril e o Manticore encontraram coletivamente apenas 37,2%. No segundo conjunto de dados, que envolvia *ether-leaking* e *suicidal contracts*, o SMARTEST foi comparado com quatro outras ferramentas: Maian, teEther, Mythril, Manticore e ILF. Os resultados mostram que o SMARTEST encontrou efetivamente mais vulnerabilidades do que todas essas ferramentas.

Neste trabalho Smaragdakis et al. [2021], foi apresentada uma análise que tenta responder ao desafio relatado sobre as ferramentas de análise automatizada que sacrificam a precisão com o objetivo de alcançar tempos de execução realistas. Para a solução desenvolvida, os autores deram o nome de Symvalic. O trabalho propôs a comparação de sua solução com outras duas ferramentas: Manticore e Mythril. A proposta do Symvalic envolve o desenvolvimento de uma solução para análise simbólica, usando o que chamam de "ponto de fixação de fluxo de valor".

O estudo utilizou 109 contratos do conjunto SmartBugs rotulados. O Manticore apresentou um tempo médio de execução de 185 segundos, mas sua porcentagem de vulnerabilidades detectadas ficou atrás das outras ferramentas. O Mythril e a análise Symvalic detectaram um número significativo de vulnerabilidades. A análise do Symvalic exibiu uma cobertura muito alta, de 96%, contra 86% do Mythril e 74% do Manticore. No total, a análise do Symvalic detectou 80 das 109 vulnerabilidades, enquanto o Mythril detectou 60 e o Manticore apenas 27.

No trabalho de Dia et al. [2021], foi definido um esquema de classificação de defeitos de contratos inteligentes, usando a Classificação Ortogonal de Defeitos como estrutura. O trabalho extraiu contratos de conjuntos de dados existentes para criar um conjunto composto por 222 contratos. Os contratos são organizados em três grupos: contratos com defeitos de software conhecidos, incluindo vulnerabilidades; contratos onde esses defeitos tenham sido corrigidos; e contratos concebidos para enganar a verificação de ferramentas. O trabalho analisou a eficácia das versões mais recentes de

três ferramentas de verificação de contratos inteligentes: Mythril, Securify2 e Slither.

Dentro das análises em relação à capacidade de detecção de defeitos, o Slither apresenta os melhores resultados, com 44%, seguido pelo Securify2 com 30% e o Mythril com 18%. Contudo, o Slither apresenta um número relativamente elevado de alertas falsos. O trabalho levantou que a ordem é invertida se considerada a proporção de alertas comparados aos resultados de detecção corretos, pois, em média, para cada defeito detectado, o Mythril produz apenas 5 alertas, enquanto o Securify2 produz 11 e o Slither produz 14 alertas.

O estudo, em relação às análises de verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos, mostrou que nenhuma das ferramentas de verificação analisadas neste estudo é adequada para qualquer um dos cenários, devido ao baixo número de verdadeiros positivos e ao elevado número de falsos positivos. No entanto, entre as três ferramentas, o Slither geralmente apresenta melhores resultados e possui uma cobertura de mais de 50%.

O artigo de Hu et al. [2023b] argumenta que existem principalmente dois desafios na detecção de defeitos em contratos inteligentes. O primeiro desafio é projetar uma *Intermediate Representation* (IR), que consiste em uma conversão feita no código-fonte e que seja fácil de entender e expandir com abrangência semântica. O segundo desafio é detectar defeitos em contratos inteligentes da maneira mais abrangente possível, garantindo uma localização precisa e mantendo baixas taxas de falsos positivos. O trabalho teve como contribuição a criação da ferramenta chamada SoliDetector, abrangendo 20 tipos de defeitos. Por fim, conduziu mais experimentos comparando o SoliDetector com outras ferramentas em termos de eficácia e tempo de execução.

O trabalho fez um experimento de comparação de três ferramentas: SmartCheck, Slither e Mythril. No experimento, a comparação com o Slither levou às seguintes conclusões: as detecções de defeitos de *Front Running* e *Arithmetic* não foram suportadas pelo Slither. O Slither teve alguns falsos negativos ao detectar *Unchecked Call* e *Reentrancy*. Semelhante ao SoliDetector, o Slither não suporta o modificador de detecção de defeito de *Reentrancy*, resultando em 61 falsos negativos. Também não reconhece a operação de transferência `address.call.value(value)`, resultando em 214 falsos negativos para o defeito *Unchecked Call*. No caso do Mythril, ocorreram falsos negativos significativos, variando de 736 a 1415 entre seis tipos de defeitos diferentes. Mythril é uma ferramenta implementada com base no método de *symbolic execution*, o que requer mais tempo para explorar o caminho de execução. Devido ao aumento significativo de linhas de contrato, o Mythril é ineficiente em contratos com grande número de defeitos.

O artigo de Shen et al. [2023] descreve a construção do conjunto de dados *Generalized Smart Contract Vulnerability Dataset* (GSVD), composto por quatro conjuntos de dados offline que utilizam contratos inteligentes em duas cadeias, Polygon e BSC. Estes conjuntos incluem dois conjuntos pequenos de Solidity, contendo 153 códigos-fonte de contratos inteligentes rotulados, utilizados para

avaliar o desempenho de ferramentas de mineração de vulnerabilidades. Além disso, o trabalho utilizou dois grandes conjuntos com 52.202 códigos de contrato reais não rotulados, empregados para verificar a validade de diversas teorias e ferramentas em condições de dados do mundo real. O GSVD consiste inteiramente em contratos inteligentes do mundo real escritos em Solidity e podem ser divididos em duas categorias. A primeira categoria é GSVDMinor, que contém vários contratos inteligentes rotulados com vulnerabilidades e podem ser usados para testar o desempenho das ferramentas de mineração de vulnerabilidades. A segunda categoria é GSVDMajor, que contém vários contratos inteligentes com vulnerabilidades não rotuladas e podem ser usados para verificar a exatidão de diversas teorias e ferramentas sob um grande número de condições de dados reais. A construção inicial foi concluída com base em duas plataformas, subdivididas em GSVDMinor-Polygon com 56 contratos, GSVDMinorBSC com 97 contratos, GSVDMajorPolygon com 10.600 contratos e GSVDMajorBSC com 41.602 contratos.

Nas análises do trabalho, chegou-se à conclusão de que o Slither tem a maior precisão no conjunto de dados Polygon, que tem um número muito menor de contratos (45%). Por sua vez, o Mythril tem a maior precisão no conjunto de dados BSC, que possui uma quantidade muito maior de dados (48%). O conjunto de dados Minor do GSVD foi testado com três ferramentas (Slither, Manticore, Mythril), mantidas atualizadas, revelando que o uso combinado detectou 61,1% das vulnerabilidades rotuladas. Dentre essas ferramentas, o Mythril obteve a maior taxa de detecção, atingindo 42,6%. O trabalho conclui que ainda há amplo espaço para avanços nas atuais ferramentas de mineração de vulnerabilidades em contratos inteligentes, devido aos seus métodos subjacentes.

No trabalho de Zheng et al. [2023] realizou-se um estudo empírico em larga escala sobre a capacidade de abordagens existentes na detecção de *Reentrancy* em contratos inteligentes. O trabalho coletou o código verificado em Solidity de 230.548 contratos no Etherscan. O estudo selecionou cinco ferramentas capazes de identificar funções possivelmente defeituosas com problemas de *Reentrancy*: Oyente, Mythril, Securify, Smartian e Sailfish. O artigo filtrou contratos duplicados com o mesmo *bytecode*, obtendo 139.424 contratos. Em outra etapa, as ferramentas foram usadas para analisar os contratos, resultando em 21.212 contratos com *Reentrancy*. O trabalho também contou com uma etapa que examinou manualmente as funções defeituosas dos contratos com *Reentrancy*, recrutando 50 participantes: 27 graduandos, 21 mestrandos e dois doutorandos. Assim, o trabalho criou um conjunto de 34 contratos verdadeiramente positivos com *Reentrancy* e um conjunto de 21.178 contratos falsos positivos sem *Reentrancy*.

O estudo teve como resultado que mais de 99,8% dos contratos de *Reentrancy* detectados pelas ferramentas são falsos positivos. Os contratos de *Reentrancy* verdadeiramente positivos estão todos relacionados ao `call.value()`, dos quais 58,8% podem ser identificados pelo Remix. 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 métodos 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*.

A seguir, os trabalhos relacionados à análise estática e simbólica são agrupados de acordo com sua semelhança em relação à proposta deste estudo, e sintetizados na Tabela III.1, com foco nas ferramentas avaliadas, na taxonomia adotada e no tamanho dos conjuntos de dados utilizados.

O primeiro grupo reúne trabalhos que realizam avaliações empíricas multi-ferramenta sobre conjuntos de dados curados ou de tamanho moderado, sem propor uma nova técnica de detecção: Durieux et al. [2020], Parizi et al. [2018], Ye et al. [2019], Dia et al. [2021] e Shen et al. [2023]. Desses, Durieux et al. [2020] constitui o trabalho base deste estudo: dele são herdados o conjunto de dados SmartBugs Curated. Ainda assim, mesmo tendo avaliado nove ferramentas, os autores o fizeram em um único ponto no tempo, sem medir se atualizações posteriores dessas ferramentas alteram sua precisão ou cobertura, lacuna que este trabalho aborda diretamente. Os demais trabalhos desse grupo usam taxonomias próprias ou baseadas nos nomes dos detectores das ferramentas, e, assim como Durieux et al. [2020], nenhum deles mede a evolução das ferramentas entre versões.

O segundo grupo é composto por trabalhos que propõem uma nova técnica de detecção, utilizando Mythril e/ou Slither apenas como linha de base de comparação: So et al. [2021] (SMARTEST), Smaragdakis et al. [2021] (Symvalic) e Hu et al. [2023b] (SoliDetector). Nesses casos, o Mythril e o Slither não são o objeto central da investigação, mas parâmetros de referência para validar o desempenho da nova solução proposta.

Por fim, Zheng et al. [2023] constitui um caso à parte: um estudo em larga escala (139.424 contratos) restrito exclusivamente à vulnerabilidade *Reentrancy*, com validação manual conduzida por 50 participantes, mas sem curadoria prévia do conjunto de dados nem comparação entre versões das ferramentas.

Em comum, nenhum dos nove trabalhos analisados mede a evolução da precisão e da cobertura de Mythril e Slither entre versões consecutivas, lacuna que motiva o primeiro experimento deste estudo.

Tabela III.1: Comparação entre os trabalhos relacionados de análise estática/simbólica e o presente estudo

Trabalho	Mythril	Slither	Nova técnica de detecção	DASP	SB Curated	Nº contratos	Evolução entre versões
Durieux et al. [2020][2]	Sim	Sim	Não	Sim	Sim	69 / 47.518	Não
Parizi et al. [2018]	Sim	Não	Não	Não	Não	10	Não
Ye et al. [2019]	Sim	Sim	Não	Não	Não	1.010	Não
Dia et al. [2021]	Sim	Sim	Não	Não	Não	222	Não
Shen et al. [2023]	Sim	Sim	Não	Parcial	Não	153 / 52.202	Não
So et al. [2021]	Sim	Não	Sim	Não	Não	104 / 2.743	Não
Smaragdakis et al. [2021]	Sim	Não	Sim	Sim	Sim	109 / 166	Não
Hu et al. [2023b]	Sim	Sim	Sim	Parcial	Sim	191 / 37.862	Não
Zheng et al. [2023]	Sim	Não	Não	Não	Não	139.424	Não
Este trabalho	**Sim**	**Sim**	**Não**	**Sim**	**Sim**	**69**	**Sim**

III.3 Grandes Modelos de Linguagem

O artigo de Chen et al. [2025] apresenta um estudo empírico sobre a aplicação do ChatGPT (gpt-3.5-turbo,gpt-4o e gpt-4) na detecção de vulnerabilidades em contratos inteligentes escritos em Solidity. Utilizando o conjunto de dados *smartbugs-curated*, os autores avaliaram a eficácia do modelo em nove tipos de falhas comuns, conforme a taxonomia DASP. Os testes mostraram que, embora o ChatGPT apresente alta taxa de cobertura (até 88,2% no GPT-4), sua precisão é baixa (cerca de 22%), resultando em muitos falsos positivos. As principais causas desses erros incluem viés pela ausência de mecanismos de proteção, má interpretação de variáveis específicas do domínio, interferência de comentários e análise incorreta de código inativo.

A pesquisa também comparou o desempenho do ChatGPT com 14 ferramentas especializadas, como Slither, Mythril e Oyente, revelando que, apesar de sua ampla cobertura e velocidade superior, o modelo ainda é superado em precisão por diversas dessas soluções. Além disso, foram identificadas limitações, como a instabilidade das respostas (42% de inconsistência em execuções repetidas) e a restrição no tamanho de código analisável devido ao limite de *tokens*. Os autores sugerem que a integração de conhecimento externo via *prompt* e técnicas como ajuste de temperatura podem melhorar do desempenho, apontando caminhos promissores para o uso de LLMs na auditoria de segurança de contratos inteligentes.

Já o trabalho de Sun et al. [2024] propõe a ferramenta GPTScan, que combina LLMs, especificamente o GPT-3.5, com análise estática para detectar vulnerabilidades lógicas em contratos inteligentes. A motivação parte da limitação das ferramentas tradicionais, como Slither e Mythril, que não conseguem identificar falhas relacionadas à lógica de negócio dos contratos. A proposta utiliza o GPT para entendimento semântico do código, classificando vulnerabilidades com base em cenários e propriedades, e validando os resultados por meio de análise estática, o que reduz os falsos positivos. O processo é dividido em três etapas: filtragem de funções candidatas, análise semântica com GPT e confirmação estática.

A avaliação foi realizada em três conjuntos de dados, totalizando 400 projetos e mais de 3 mil arquivos Solidity. O GPTScan obteve precisão de 90,91% no conjunto de dados DefiHacks, precisão de 57,33% no Web3Bugs. Além disso, a ferramenta identificou nove vulnerabilidades não percebidas por auditores humanos. O artigo conclui que a combinação de LLMs com análise estática é eficaz na detecção de falhas lógicas, superando abordagens puramente baseadas em GPT ou ferramentas tradicionais.

O artigo de Hu et al. [2023a] apresenta análise sistemática sobre o uso de LLMs, especificamente o GPT-4, para detecção de vulnerabilidades em contratos inteligentes. Os autores identificam um dilema central: gerar mais respostas com maior aleatoriedade aumenta a chance de detectar vulne-

rabilidades reais, mas eleva inevitavelmente o número de falsos positivos. Para resolver esse conflito, propõem o *framework* GPTLENS, que divide a detecção em dois estágios, geração e discriminação, nos quais o LLM assume os papéis de Auditor e Crítico, respectivamente. O Auditor gera um amplo espectro de vulnerabilidades candidatas com alta aleatoriedade, enquanto o Crítico avalia cada candidata com base em corretude, severidade e lucratividade. A abordagem adota *open-ended prompting*, sem categorias predefinidas, conferindo maior generalidade ao método, incluindo a capacidade de identificar falhas desconhecidas ou não categorizadas.

A avaliação foi conduzida em 13 contratos inteligentes reais coletados do Etherscan, todos com vulnerabilidades confirmadas no banco CVE. Os resultados mostram que o GPTLENS eleva o taxa de acerto por contrato de 38,5% para 76,9% em relação à detecção convencional de um estágio, e a taxa de acerto por tentativa sobe de 33,3% para 59,0% com dois auditores. Entre as limitações apontadas estão o tamanho reduzido do conjunto de dados, a restrição de *tokens* que impede configurações maiores, e inconsistências no *scoring* quando as vulnerabilidades precisam ser divididas em lotes. O código está disponível publicamente em `https://github.com/git-disl/GPTLens`.

O artigo de Boi et al. [2024] propõe o VulnHunt-GPT, uma ferramenta de detecção de vulnerabilidades em contratos inteligentes Ethereum baseada no modelo GPT-3.5-turbo da OpenAI. A motivação central é a falta de completude das ferramentas tradicionais, nenhuma das quais cobre todas as categorias do DASP Top 10. A abordagem combina duas técnicas: *prompt engineering*, que define os papéis do modelo via API para retornar saídas estruturadas em vulnerabilidades e remediações; e enriquecimento de contexto (*context enrichment*), realizado com as bibliotecas Python LangChain e LlamaIndex para indexar as descrições textuais das vulnerabilidades e gerar vetores semânticos. A arquitetura opera em três fases sequenciais: pré-processamento do contrato, comunicação com a API da OpenAI e análise pelo modelo.

A avaliação foi conduzida sobre o conjunto de dados SmartBugs Curated, composto por 69 contratos inteligentes categorizados segundo o DASP Top 10, comparando o VulnHunt-GPT com nove ferramentas especializadas, entre elas Slither, Mythril, Manticore e Oyente. O principal resultado é que o VulnHunt-GPT foi a única ferramenta a detectar ao menos uma vulnerabilidade em cada uma das dez categorias, totalizando 128 vulnerabilidades identificadas, incluindo a categoria *Bad Randomness*, não coberta por nenhuma das demais. Em tempo de execução, a ferramenta apresentou média de 23 segundos por contrato, sendo uma das mais rápidas avaliadas. Os autores concluem que, apesar da ampla cobertura, modelos baseados em GPT devem ser utilizados como componentes de um conjunto de testes mais abrangente, e apontam como limitação a ausência de avaliação de falsos positivos, deixada como trabalho futuro.

O artigo de David et al. [2023] investiga a viabilidade do uso de LLMs para a realização de auditorias de segurança em contratos inteligentes, uma tarefa tradicionalmente custosa e demorada.

A motivação central é o crescente número de ataques a protocolos *Decentralized Finance* (DeFi), que acumularam prejuízos superiores a 956 milhões de dólares nos 52 contratos analisados. A abordagem combina engenharia de prompt com classificação binária e não-binária, teste de mutação sobre cinco contratos sintéticos com vulnerabilidades inseridas manualmente, e dois estudos de caso com *chain-of-thought prompting*. Os modelos avaliados, GPT-4-32k e Claude-v1.3-100k, foram submetidos a um total de 3.952 chamadas de API, cobrindo 38 tipos de vulnerabilidades extraídos do DeFi Attack SoK, utilizando um modelo aleatório como *baseline*.

A avaliação foi conduzida sobre um conjunto de 52 contratos DeFi previamente comprometidos, com *ground-truth* extraído do DeFi Attack SoK e código-fonte coletado via API do Etherscan. O principal resultado é que ambos os modelos identificaram corretamente o tipo de vulnerabilidade em 40% dos casos, superando o *baseline* aleatório em aproximadamente 20% no F1-score, enquanto no teste de mutação o GPT-4-32k atingiu até 78,7% de taxa de verdadeiros positivos. Contudo, ambos os modelos apresentaram alta taxa de falsos positivos, exigindo contínua supervisão humana. Os autores concluem que LLMs representam um avanço promissor para triagem automatizada de vulnerabilidades, mas não substituem auditores manuais no estágio atual, e apontam como limitações o possível viés oriundo dos dados de treinamento, o truncamento ingênuo de contratos extensos e a degradação de desempenho em contextos mais longos.

O artigo de Ma et al. [2024] propõe o iAudit, um *framework* que combina ajuste fino e agentes baseados em LLMs para auditoria intuitiva de contratos inteligentes com justificativas. A motivação central é a limitação dos LLMs de propósito geral, como GPT-4, que sem ajuste fino específico para Solidity atingem apenas 30% de precisão quando decisão e justificativa são avaliadas conjuntamente. A abordagem é inspirada no comportamento de auditores humanos experientes: primeiro perceber intuitivamente se há algo errado, depois analisar a causa. Para isso, o *framework* emprega dois modelos ajustados via LoRA sobre CodeLlama-13b, o Detector, que classifica o código como vulnerável ou seguro, e o Reasoner, que gera explicações para as vulnerabilidades detectadas, ambos treinados com múltiplos prompts e decisão por maioria de votos. Adicionalmente, dois agentes LLM baseados no Mixtral 8x7B-Instruct, o Ranker e o Critic, colaboram iterativamente para selecionar a explicação mais adequada entre as candidatas produzidas pelo Reasoner.

A avaliação foi conduzida sobre um conjunto de dados balanceado com 1.734 amostras positivas extraídas de 263 relatórios de auditoria reais e 1.810 amostras negativas geradas por método próprio de data enhancement com GPT-4 e GPTScan. O iAudit foi comparado com modelos ajuste fino tradicionais (CodeBERT, GraphCodeBERT, CodeT5, UnixCoder) e LLMs em modo zero-shot (GPT-4, GPT-3.5, CodeLlama-13b/34b). Os resultados mostram que o iAudit atingiu F1 de 91,21% e acurácia de 91,11%, superando significativamente os *baselines*, que ficaram na faixa de 60% a 89%. No alinhamento das explicações com as causas reais, o iAudit alcançou 37,99% de con-

sistência, contra 24,13% do segundo colocado (GPT-4). Em aplicação real, a ferramenta identificou 6 vulnerabilidades críticas em projetos do Code4rena, uma das quais não havia sido detectada por nenhuma outra ferramenta.

O presente trabalho se diferencia dos estudos anteriores por avaliar o Gemini 2.0 Flash, modelo com nível de acesso gratuito, utilizando o mesmo *pipeline* experimental de Chen et al. [2025], incluindo o mesmo conjunto de dados (*smartbugs-curated*, 142 contratos), a mesma taxonomia DASP e os mesmos *scripts*, o que permite uma comparação controlada entre modelos. Enquanto Sun et al. [2024] e Ma et al. [2024] buscam elevar a precisão por meio da combinação de LLMs com análise estática e ajuste fino, respectivamente, o presente trabalho avalia modelos sem adaptação de domínio, priorizando reprodutibilidade e acessibilidade. Hu et al. [2023a] também dispensa análise estática e ajuste fino, mas opera com prompts sem categorias predefinidas sobre apenas 13 contratos e sem métricas padronizadas. Boi et al. [2024] é o trabalho mais próximo em simplicidade, também aplicando GPT-3.5 sem adaptação sobre o SmartBugs Curated, porém sem calcular precisão, cobertura ou F1, e sem análise manual; este trabalho supera essa limitação ao introduzir avaliação por ocorrência com protocolo manual e intervalos de confiança via *bootstrap*. Por fim, David et al. [2023] avalia GPT-4 e Claude sobre 52 contratos DeFi com foco em ataques reais de alto impacto financeiro, enquanto o presente trabalho prioriza base estável e comparação controlada entre modelos. Em conjunto, esses trabalhos apontam para um consenso: LLMs apresentam alta cobertura mas baixa precisão sem adaptação de domínio, e a avaliação binária, predominante na literatura, superestima o desempenho real, contribuição metodológica central deste trabalho. A Tabela III.2 sintetiza essas diferenças, comparando o presente trabalho aos estudos relacionados quanto ao modelo avaliado, conjunto de dados utilizado, taxonomia adotada, métricas calculadas e demais critérios metodológicos.

Tabela III.2: Comparação entre trabalhos relacionados e o presente estudo

Critério	**Chen et al. [2025]**	**Sun et al. [2024]**	**Hu et al. [2023a]**	**Boi et al. [2024]**	**David et al. [2023]**	**Ma et al. [2024]**	**Este trabalho**
Modelo avaliado	GPT-3.5/4	GPT-3.5	GPT-4	GPT-3.5	GPT-4/Claude	CodeLlama	Gemini 2.0
Modelo gratuito	Não	Não	Não	Não	Não	Não	Sim
Análise estática combinada	Não	Sim	Não	Não	Não	Não	Não
Taxonomia DASP	Sim	Não	Não	Sim	Não	Não	Sim
Dataset SmartBugs Curated	Sim	Não	Não	Sim	Não	Não	Sim
Número de contratos	142	470+	13	69	52	3.544	142
Calcula Precisão/Cobertura/F1	Sim	Sim	Não	Não	Sim	Sim	Sim
Análise manual dos resultados	Sim	Sim	Sim	Não	Sim	Sim	Sim
Avaliação dos relatórios	Não	Não	Não	Não	Não	Não	Sim
Intervalos de confiança (*bootstrap*)	Não	Não	Não	Não	Não	Não	Sim
Artefatos públicos	Sim	Sim	Sim	Não	Não	Sim	Sim

Capítulo IV Metodologia

O principal objetivo deste trabalho é analisar ferramentas automatizadas de detecção de vulnerabilidades em contratos inteligentes, combinando abordagens tradicionais e técnicas baseadas em LLMs. Inicialmente, foram estudadas duas das principais ferramentas de análise automatizada identificadas por Di Angelo and Salzer [2019] e refinadas em Durieux et al. [2020]: Mythril e Slither, apontadas como as mais eficazes quando utilizadas em conjunto. Esses estudos permanecem relevantes por apresentarem um dos conjuntos de dados mais precisos e abrangentes sobre o tema. Além disso, foi considerada a solução SmartBugs [Ferreira et al., 2020], que integra diferentes ferramentas para análise de vulnerabilidades. Para examinar um grande conjunto de códigos em Solidity, desenvolveu-se uma abordagem automatizada baseada em *scripts* capazes de executar as ferramentas em todos os arquivos do repositório. Foram realizados testes e refatorações para garantir que o código fosse genérico e reutilizável. Durante essa etapa, constatou-se que o Mythril não oferece suporte ao sistema operacional Windows, o que levou à adoção de um ambiente Linux Mint 21.2, com processador Core i5 (8ª geração) e 8 GB de RAM. Este primeiro experimento do trabalho fez parte da publicação do artigo Fonseca et al. [2025].

A Figura IV.1 apresenta o fluxo geral da metodologia adotada no primeiro experimento, ilustrando as entradas e saídas de cada etapa do processo de análise estática com Mythril e Slither.



Figura IV.1: Fluxo geral da metodologia do Primeiro Experimento — Análise Estática com Mythril e Slither.

No segundo experimento, este trabalho propõe aplicar LLMs na análise de vulnerabilidades,

fundamentando-se na metodologia apresentada por Chen et al. [2025], cuja replicação visa assegurar coerência e reprodutibilidade dos resultados. O experimento utilizou o mesmo conjunto de dados (*smartbugs-curated*) e os *scripts* originais disponibilizados pelos autores no repositório oficial[1], permitindo comparações diretas entre diferentes LLMs sob condições equivalentes.

De forma análoga ao estudo de referência, adotou-se a taxonomia DASP e o prompt otimizado para a análise, bem como estratégias de verificação inspiradas no código disponibilizado no repositório[2]. Dessa forma, este estudo combina métodos consolidados com abordagens inovadoras, buscando avaliar a eficácia das ferramentas tradicionais e dos LLMs na detecção de vulnerabilidades em contratos inteligentes, oferecendo uma análise abrangente e comparativa. O segundo experimento deste trabalho originou um artigo aceito no Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain 2026) [Fonseca et al., 2026]. A versão apresentada nesta dissertação estende e aprimora os resultados publicados.

A Figura IV.2 apresenta o fluxo geral da metodologia adotada no segundo experimento, ilustrando as entradas e saídas de cada etapa do processo de análise com os LLMs.

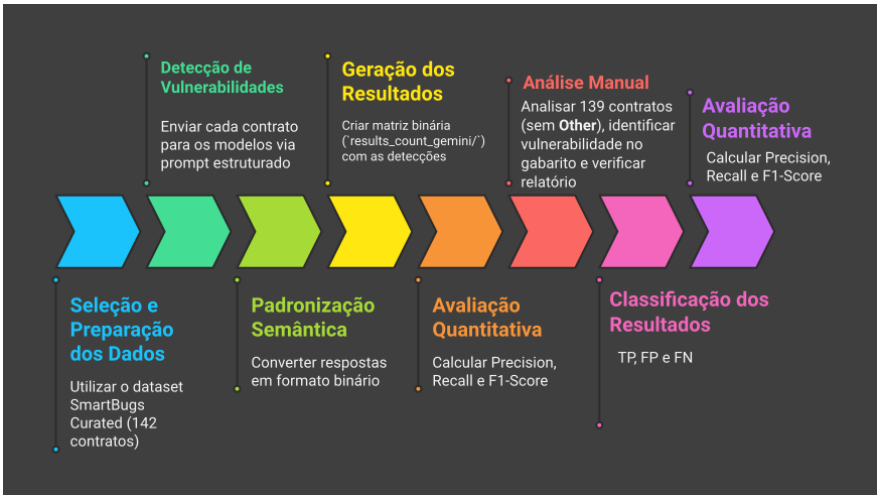


Figura IV.2: Fluxo geral da metodologia do Segundo Experimento — Análise LLMs.

IV.1 Primeiro Experimento: Análise de Vulnerabilidades do Mythril e Slither

No primeiro experimento do trabalho, que analisa as duas principais ferramentas de análise de contratos inteligentes, aplicou-se um *script* automatizado, onde cada ferramenta é acionada por meio de comandos de linha. O *script* inclui um mecanismo para reconhecer o *pragma* especificado no arquivo e ajustar automaticamente o compilador Solidity (*solc*) adequado para o código em questão. Este trabalho, ao buscar novos conjuntos de dados para análises adicionais, encontrou no trabalho de Ibba et al. [2024] a abordagem para enfrentar o mesmo problema. No estudo de

[1] `https://github.com/BugmakerCC/GPT4scv/tree/master`
[2] `https://github.com/BugmakerCC/GPT4scv/blob/master/output_limit.py`

Ibba, há uma descrição muito semelhante. Neste trabalho, também foi utilizado um *script* em Python para detectar e trocar o *solc* conforme os diferentes valores de *pragma*, garantindo que cada contrato inteligente fosse compilado o mais próximo possível de suas especificações originais. Com base nessa identificação, o *script* monta o comando apropriado para analisar o arquivo Solidity correspondente. Após a execução automatizada das ferramentas em cada arquivo, é gerado um arquivo JSON contendo os resultados da análise.

Concluída a etapa de execução, inicia-se o processo de extração de dados. O código de extração busca, nos arquivos JSON gerados, chaves que contenham informações sobre vulnerabilidades detectadas. A partir desses dados, é criado um *dataframe* em que os nomes das vulnerabilidades servem como títulos das colunas, e cada linha representa um arquivo Solidity. Para cada linha, é registrada a quantidade de ocorrências de cada vulnerabilidade identificada no respectivo arquivo.

Em seguida, as vulnerabilidades detectadas são traduzidas para a taxonomia DASP. Essa tradução é realizada por meio de um mapeamento definido em um dicionário. Como algumas vulnerabilidades distintas podem pertencer à mesma categoria na taxonomia DASP, o *script* consolida esses valores. O resultado é um novo *dataframe* que apresenta as categorias DASP e a quantidade consolidada de vulnerabilidades para cada arquivo analisado.

Por fim, os dados gerados pelas ferramentas foram utilizados para comparar os resultados com as análises realizadas no SmartBugs. Essa comparação teve como objetivo avaliar o volume de detecções das ferramentas. Além disso, foi realizada uma análise manual para verificar a precisão das detecções, garantindo que os problemas identificados correspondessem aos locais corretos nos arquivos analisados. Esses aspectos serão detalhados na seção de resultados.

O código automatizado do experimento[3] pode ser resumido nos seguintes passos:

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.

[3] `https://github.com/drinith/mythril-slither-case-study-wblockchain-2025`

Alguns pontos são importantes de serem colocados para futuras análises e comparações com este trabalho. Nas primeiras análises, observou-se uma discrepância entre o número de arquivos Solidity presentes no repositório atual do SmartBugs e os 69 contratos reportados no estudo base de [Durieux et al., 2020]. Uma investigação mais profunda revelou que o repositório do SmartBugs havia sido atualizado ao longo do tempo, com a inclusão de novos contratos além dos 69 originalmente analisados, sendo essa, inclusive, a origem dos 142 contratos utilizados posteriormente no segundo experimento deste trabalho (Seção IV.2). Para garantir uma comparação justa com o estudo base, foi necessário identificar precisamente quais arquivos correspondiam aos 69 contratos originais. Para isso, buscou-se um local que disponibilizasse as informações das análises validadas e publicadas pelo SmartBugs. Após uma busca mais aprofundada no repositório do GitHub[4], foram encontrados os arquivos JSON gerados pelo SmartBugs, o que tornou possível separar os arquivos que foram realmente analisados no estudo original.

Outro ponto notado na pesquisa é que o Mythril, na análise do SmartBugs, possui uma tabela de conversão[5] que apresenta vulnerabilidades diferentes das encontradas nas análises atuais. Na tabela do repositório do artigo no GitHub, foram encontradas 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: *Integer Overflow and Underflow* (SWC-101), *Unchecked Call Return Value* (SWC-104), *Unprotected Ether Withdrawal* (SWC-105), *Unprotected SELFDESTRUCT Instruction* (SWC-106), *Reentrancy* (SWC-107), *Assert Violation* (SWC-110), *Delegatecall to Untrusted Callee* (SWC-112), *DoS with Failed Call* (SWC-113), *Transaction Order Dependence* (SWC-114), *Authorization through tx.origin* (SWC-115), *Timestamp Dependence* (SWC-116), *Weak Sources of Randomness from Chain Attributes* (SWC-120) e *Write to Arbitrary Storage Location* (SWC-124).

Assim, foi necessário criar uma tabela de conversão das vulnerabilidades, classificadas pela SWC[6] para o DASP. As informações para a conversão foram encontradas em um arquivo do próprio SmartBugs, então adotaram-se esses dados para essa saída[7].

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

[4] `https://github.com/SmartBugs/SmartBugs-results/tree/master/results/Slither/curated`
[5] `https://github.com/SmartBugs/SmartBugs/wiki/Vulnerabilities-mapping`
[6] `https://swcregistry.io/`
[7] `https://github.com/smartbugs/smartbugs/blob/master/tools/mythril-0.23.15/findings.yaml`

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, são comparadas mais à frente somente as vulnerabilidades que foram traduzidas. As outras vulnerabilidades não entram na comparação de acertos, pois os arquivos validados também se encontravam na classificação DASP. A classificação *Other* será utilizada apenas para a contagem total de detecções (Tabela V.3), mas será excluída nas análises de acurácia e nas métricas de desempenho (precisão, cobertura e *F1-score*).

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, identificou-se que o *commit* referente à análise foi realizado em 24 de agosto de 2019. Com base nessa data, constatou-se que as versões específicas do Mythril e do Slither correspondentes a esse *commit* são, respectivamente, 0.21.15 e 0.6.6. Assim, essas versões foram adotadas como referência para a análise realizada neste trabalho.

Durante os testes de revisão do trabalho, após a correção de falhas no código das ferramentas para que fossem executadas nas versões mais recentes do Python, foi possível executá-las com as versões estimadas. Observou-se que a saída gerada foi equivalente à do estudo original. Portanto, é provável que os autores tenham utilizado versões próximas a essas.

IV.1.1 Análise Manual: Mythril e Slither

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 nos relatórios do Slither. Observou-se que as vulnerabilidades nem sempre apareciam exatamente nos locais indicados pelas informações classificadas como precisas, como apresentado no Apêndice A.

Um outro exemplo que demonstra a necessidade da análise manual pode ser notado no Apêndice B. 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, foi realizada uma recontagem das vulnerabilidades encontradas no trabalho base. Notou-se que o comportamento do arquivo *FibonacciBalance.sol*, explicado no Apêndice C, acrescentou uma detecção adicional.

Uma segunda questão observada nas contagens manuais foi o aumento na quantidade de vulnerabilidades apontadas nos contratos Solidity. Conforme destacado no Apêndice C, foram identificadas 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 SmartBugs para *Unchecked Low-Level Calls*. E não foi encontrado um fato concreto que sustentasse essas diferenças, conforme indicado nesta seção e nos apêndices. 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, observou-se que o *commit* mais antigo não coincide com a época da publicação do artigo; portanto, é plausível que tenham ocorrido alterações[8].

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 as diferenças identificadas, as análises manuais revisaram as saídas das ferramentas tanto nas versões anteriores do SmartBugs, quanto nas atuais. Isso permitiu estabelecer uma base comparativa consistente, garantindo que a comparação entre as versões fosse válida, pois os acertos foram verificados em ambas as análises, apesar das mudanças nas contagens.

IV.2 Segundo Experimento: Análise de Vulnerabilidades com LLMs

O segundo experimento deste trabalho tem como objetivo central avaliar o desempenho dos Grandes Modelos de Linguagem na detecção de vulnerabilidades em contratos inteligentes, por meio de uma abordagem experimental baseada em replicação de estudos existentes.

A metodologia adotada segue o arcabouço proposto no trabalho base, utilizando o conjunto de dados SmartBugs Curated, o que permite a comparação direta com resultados previamente publicados. O processo experimental foi estruturado em duas etapas principais: (i) detecção de vulnerabilidades e (ii) padronização dos resultados para análise quantitativa.

Na etapa de detecção, os contratos foram submetidos ao Gemini 2.0 Flash, utilizando *prompt* estruturado. Em seguida, as respostas geradas foram processadas e convertidas para uma representação padronizada baseada na taxonomia DASP Top 10.

Essa padronização possibilitou a avaliação de desempenho por meio de métricas quantitativas,

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

incluindo precisão, cobertura e *F1-score*, permitindo uma análise comparativa entre os modelos.

O código-fonte desenvolvido neste trabalho está disponível no repositório[9].

O delineamento no segundo experimento foi inspirado no estudo de referência [Chen et al., 2025], que avaliou a capacidade do ChatGPT (nas versões GPT-3.5-turbo, GPT-4o e GPT-4) em identificar falhas no conjunto de dados *smartbugs-curated*. A mesma metodologia foi replicada para assegurar coerência e reprodutibilidade, sendo adaptada à API do Gemini, modelo da Google, a fim de permitir a comparação entre famílias distintas de LLMs. A escolha do modelo Gemini 2.0 Flash para a replicação da metodologia de análise foi fundamentada não apenas em critérios técnicos para a comparação entre arquiteturas de LLMs, mas também na sua viabilidade operacional para o projeto. Em contraste com os modelos utilizados no estudo de referência (OpenAI), que operam sob um regime comercial de custos por *token* para acesso via API, o Gemini disponibiliza uma API acessível via Python e um nível de uso gratuito que suporta a execução de experimentos em larga escala. Essa característica é crucial, pois elimina as barreiras financeiras associadas a ferramentas comerciais.

Para este segundo experimento, foram utilizados os contratos do conjunto de dados público SmartBugs Curated. Este conjunto de dados, em sua versão atualizada, é composto por 142 contratos vulneráveis, organizados segundo a taxonomia DASP.

É importante notar que este número diverge dos 69 contratos utilizados no primeiro experimento. Isso ocorre porque o primeiro experimento replicou a metodologia do estudo-base de Durieux et al. [2020], que utilizou o conjunto de dados naquela configuração. Para este experimento de análise com LLMs, foram adotados os dados expandidos de 142 contratos, alinhando a metodologia à do trabalho de referência mais recente, de Chen et al. [2025] e permitindo uma comparação direta com seus achados.

Antes da análise, aplicou-se um pré-processamento que removeu anotações de vulnerabilidade e comentários específicos (como `// <yes> <report>` e `vulnerable_at_lines`), evitando viés nos resultados.

IV.2.1 Detecção de Vulnerabilidades

Nesta etapa, o *script* `experiment_gemini.py` automatiza o envio de cada contrato ao modelo Gemini 2.0 Flash, que recebe um *prompt* estruturado no formato proposto por Chen et al. [2025]:

```
You are a vulnerability detector for a smart contract.
Here are nine common vulnerabilities: [...]
Think step by step, carefully.
Check the following smart contract for the above vulnerabilities:
```

[9] https://github.com/drinith/FONSECA_DVCIEFA_LLMs

```
<código do contrato>
```

As respostas são armazenadas em formato textual livre (*results_ gemini/*) e representam a inferência do modelo sobre a presença ou ausência das vulnerabilidades.

O *script* implementa controle de taxa de requisições (*rate limit*), *timeout* de 100 segundos e atraso de 4 segundos entre as chamadas, respeitando os limites da API gratuita. Em casos de erro 429, é aplicado o tempo de espera sugerido pelo cabeçalho de resposta do servidor, garantindo a continuidade da execução em larga escala.

Cada contrato foi submetido ao Gemini 2.0 Flash em uma única execução, com a temperatura *default* da API (*temperature* = 1,0), uma vez que o *script* não especifica um objeto `generation_config` customizado. A opção pelo valor padrão teve como objetivo observar o comportamento natural do Gemini sob sua configuração, sem ajustes que pudessem favorecer artificialmente o desempenho do modelo. Os dados de referência do GPT-4 correspondem à saída disponibilizada no repositório oficial de Chen et al. [2025], reaproveitada por questões de viabilidade operacional, já que a execução própria via API da OpenAI implicaria custos financeiros.

IV.2.2 Padronização Semântica

O segundo *script*, *output_ limit_ gemini.py*, realiza a normalização semântica das respostas. O objetivo é converter as saídas textuais em um formato binário padronizado, com valores 0 (ausência) ou 1 (presença) para cada vulnerabilidade, permitindo a aplicação de métricas quantitativas de desempenho (precisão, cobertura e F1-score).

O *prompt* de análise semântica segue o mesmo modelo do estudo base:

```
You are a semantic analyzer of text.
Here are nine common vulnerabilities: [...]
The following text is a vulnerability detection result for a smart contract.
Use 0 or 1 to indicate whether there are specific types of vulnerabilities.
Example: "Reentrancy: 1"
The input is: <resposta do Gemini>
```

Assim como em Chen et al. [2025], o modelo atua como um "analisador semântico" independente, reduzindo ruídos contextuais e garantindo consistência entre as inferências livres e os resultados finais.

IV.2.3 Armazenamento e Avaliação

Os resultados são salvos em `results_count_gemini/`, compondo uma matriz binária de detecção. A comparação entre o Gemini e o GPT é realizada com base nas métricas de precisão, cobertura

e *F1-score* para cada categoria de vulnerabilidade, conforme a metodologia do artigo original. Essa abordagem assegura comparabilidade direta entre as famílias de modelos, avaliando não apenas sua capacidade de detecção, mas também o grau de consistência semântica das respostas.

IV.2.4 Configuração do Ambiente

Para a execução dos experimentos com o Gemini 2.0 Flash, foi necessário configurar o acesso à API do Google. O gerenciamento das dependências foi realizado via `pip`, instalando as seguintes bibliotecas Python:

```
pip install google-generativeai python-dotenv func-timeout
```

A autenticação é feita por meio de uma chave de API obtida gratuitamente no Google AI Studio[10]. Para evitar a exposição da chave no código-fonte, ela foi armazenada em um arquivo `.env` na raiz do projeto:

```
GOOGLE_API_KEY=sua_chave_aqui
```

No *script* `experiment_gemini.py`, a chave é carregada automaticamente pela biblioteca `python-dotenv` e utilizada para inicializar o cliente da API:

```
from dotenv import load_dotenv
import google.generativeai as genai
import os

load_dotenv()
api_key = os.environ.get("GOOGLE_API_KEY")
genai.configure(api_key=api_key)
model = genai.GenerativeModel('gemini-2.0-flash')
```

O nível gratuito da API impõe limitações de requisições por minuto. Por esse motivo, o *script* implementa um mecanismo de *timeout* de 100 segundos por requisição e um controle de erros 429 (*rate limit exceeded*), aguardando o tempo sugerido pelo cabeçalho de resposta do servidor antes de retomar a execução.

IV.2.5 Análise Manual: LLM

O trabalho base quantifica os resultados de suas análises por meio de um analisador semântico que converte a explicação textual em uma saída binária por categoria, denominação adotada neste

[10] `https://aistudio.google.com/`

trabalho para a classificação que atribui 0 à ausência e 1 à presença da vulnerabilidade no arquivo, sem indicar sua localização exata no código. No estudo comparado, os autores realizaram uma análise manual em 50 contratos para validar se essa representação era suficiente para quantificar os resultados com precisão.

Contudo, a presença de múltiplas vulnerabilidades em um único arquivo Solidity levanta um questionamento relevante: se uma métrica binária é capaz de capturar a totalidade das vulnerabilidades reais, motivando a análise manual descrita a seguir.

Por esse motivo, definiu-se como etapa essencial a análise manual dos 142 relatórios, abrangendo tanto os resultados do Gemini quanto os da versão GPT-4 (disponibilizada no repositório original). Dos 142 relatórios, 3 arquivos Solidity estão classificados como *Other*, correspondentes à 10ª categoria do DASP, denominada *Unknown Unknowns* [Chen et al., 2025], que não é discutida no trabalho base e não possui mapeamento direto em nenhuma das nove categorias avaliadas. Esses arquivos foram excluídos da análise manual, resultando em 139 contratos por modelo.

O protocolo de anotação seguiu três etapas para cada contrato. Primeiro, identificou-se no arquivo `.sol` a função e a instrução apontadas pelo gabarito como vulneráveis. Segundo, o relatório gerado pelo modelo foi lido na íntegra para verificar se mencionava especificamente aquela função e instrução. Terceiro, cada detecção foi classificada segundo os seguintes critérios:

- **Verdadeiro Positivo (TP):** o modelo identificou corretamente a vulnerabilidade presente no gabarito, apontando a função ou instrução vulnerável correspondente. Cada ocorrência confirmada foi registrada individualmente, somando uma unidade na coluna da categoria correspondente àquele arquivo.
- **Falso Positivo (FP):** o modelo apontou uma vulnerabilidade em uma função ou instrução que não consta no gabarito. Cada detecção indevida por categoria foi contabilizada individualmente.
- **Falso Negativo (FN):** calculado pela diferença entre o total de ocorrências no gabarito e os TPs confirmados para aquela categoria ($FN = \text{Gabarito} - TP$).

Os resultados foram consolidados em planilhas estruturadas de TPs e FPs para cada modelo, disponíveis no repositório público do trabalho[11]. Para reduzir possíveis erros de contagem, utilizou-se um *script* de apoio e uma verificação de consistência com diferentes LLMs em suas versões gratuitas, incluindo o próprio Gemini (Gemini 2.0 Flash), o Groq com llama-3.3-70b-versatile e o Mistral com mistral-large-latest. O papel desses LLMs limitou-se à conferência da soma aritmética de TPs e FPs já registrados pelo autor nas planilhas, comparando o total calculado manualmente

[11] `https://github.com/drinith/FONSECA_DVCIEFA_LLMs/tree/master/manual_analysis`

com o total recalculado por cada modelo; em nenhum momento os LLMs participaram da decisão sobre a validade semântica de uma detecção, ou seja, sobre se um relatório de fato identificava corretamente a vulnerabilidade do gabarito, etapa que permaneceu integralmente sob julgamento humano, conforme os critérios descritos acima. Divergências de contagem superiores a dois casos entre o total do autor e o total recalculado pelos LLMs foram revisadas manualmente, consultando diretamente os relatórios gerados e o gabarito do conjunto de dados. A análise semântica foi conduzida, portanto, por um único avaliador humano, limitação discutida na Seção V.4.

Capítulo V Resultados

V.1 Primeiro Experimento

Ao executar a análise e seguir os três primeiros passos da metodologia, foram obtidos os dados apresentados nas Tabelas V.1 e V.2. Essas tabelas também estão disponíveis no repositório do trabalho[1]. As vulnerabilidades identificadas utilizam a nomenclatura padrão de cada ferramenta; entretanto, para fins de comparação, todas foram posteriormente convertidas para o padrão DASP. Em todas as tabelas desta seção, as colunas com versões $\leq$0.21.15 e $\leq$0.6.6 referem-se às versões do Mythril e do Slither empregadas por Durieux et al. [2020], enquanto as colunas 0.24.7 e 0.10.0 correspondem às versões utilizadas neste trabalho.

A Tabela V.1 apresenta a contagem total de detecções feitas pelo Mythril, na versão 0.24.7, para cada vulnerabilidade encontrada. Já a Tabela V.2 exibe a contagem correspondente para o Slither, na versão 0.10.0.

Tabela V.1: Análise com Mythril versão 0.24.7 dos 69 arquivos Solidity validados do SmartBugs Durieux et al. [2020]

Vulnerabilidade	**Soma Total**
Reentrancy	24
Integer Overflow and Underflow	21
Transaction Order Dependence	18
Assert Violation	17
Unchecked Call Return Value	11
Unprotected Ether Withdrawal	9
Timestamp Dependence	7
Weak Sources of Randomness from Chain Attributes	4
Delegatecall to Untrusted Callee	3
Requirement Violation	3
Write to Arbitrary Storage Location	2
DoS with Failed Call	2
Authorization through tx.origin	2
Unprotected SELFDESTRUCT Instruction	1

Como explicado na metodologia, o resultado das Tabelas V.1 e V.2 é convertido para o DASP. Esta consolidação é apresentada na Tabela V.3.

Na Tabela V.3, dentre as diferenças observadas, destaca-se a vulnerabilidade aritmética (*Arith-*

[1] `https://github.com/drinith/mythril-slither-case-study-wblockchain-2025`

Tabela V.2: Análise com Slither versão 0.10.0 dos 69 arquivos Solidity validados do SmartBugs [Durieux et al., 2020]

Vulnerabilidade	**Soma Total**
naming-convention	191
solc-version	130
deprecated-standards	59
reentrancy-unlimited-gas	38
unchecked-send	25
timestamp	23
incorrect-equality	22
constable-states	21
controlled-array-length	16
external-function	13
missing-zero-check	12
arbitrary-send-eth	11
low-level-calls	11
divide-before-multiply	9
events-maths	9
too-many-digits	9
reentrancy-events	8
boolean-equal	8
reentrancy-eth	8
weak-prng	8
tautology	7
erc20-interface	6
dead-code	6
incorrect-modifier	5
events-access	5
reentrancy-no-eth	5
uninitialized-state	4
cache-array-length	3
similar-names	3
constant-function-asm	3
reentrancy-benign	3
unused-state	3
assembly	3
controlled-delegatecall	3
uninitialized-storage	3
costly-loop	3
suicidal	2
tx-origin	2
uninitialized-local	2
redundant-statements	1
missing-inheritance	1
calls-loop	1
shadowing-local	1

Tabela V.3: Contagem de detecção das análises do SmartBugs [Durieux et al., 2020] comparadas com as deste trabalho

	Mythril		**Slither**	
Categoria (DASP)	≤0.21.15	0.24.7	≤0.6.6	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

metic). No estudo comparativo, a análise do SmartBugs Durieux et al. [2020], realizada com a ferramenta Mythril, detectou 92 ocorrências dessa vulnerabilidade, enquanto a análise deste trabalho identificou apenas 21. O SmartBugs indica a presença de 22 vulnerabilidades aritméticas validadas 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 *Time Manipulation*. Na análise do trabalho anterior, não foram detectadas vulnerabilidades desse tipo com a ferramenta Mythril. No entanto, na análise realizada neste trabalho, foi identificada uma vulnerabilidade verdadeira positiva no arquivo *roulette.sol*. Além disso, foram detectados, no total, sete casos dessa vulnerabilidade. Esse ponto merece destaque, pois, conforme indicado pela tabela de tradução das vulnerabilidades e por algumas detecções realizadas, o Mythril passou a considerar as categorias *Bad Randomness* e *Denial of Service*, que não eram contempladas na versão anterior da ferramenta.

Seguindo a metodologia descrita em Durieux et al. [2020], foi comparada a quantidade de vulnerabilidades efetivamente encontradas com a quantidade indicada nos comentários dos códigos. Para isso, foram repetidas tanto as análises do trabalho comparado quanto as deste estudo, cujos resultados estão sumarizados na Tabela V.4.

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, foi identificada uma provável 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 diretamente no repositório do trabalho[2].

Para mapear as diferenças entre as análises e investigar possíveis causas para as divergências, foi elaborada uma tabela. Essa tabela lista os arquivos Solidity que apresentaram discrepâncias nas

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

Tabela V.4: Análise baseada no trabalho de Durieux et al. [2020].

	Mythril		**Slither**	
Análise	≤0.21.15	0.24.7	≤0.6.6	0.10.0
Access Control	5/24	7/24	6/24	7/24
Arithmetic	15/22	14/22	0/22	0/22
Denial Service	0/7	0/7	0/7	0/7
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

Tabela V.5: Análise de diferença das detecções Mythril.

Arquivo	**Vulnerabilidade**	**0.21.15**	**0.24.7**	**Linha**	**0.21.15 Linha**	**0.24.7 Linha**
FibonacciBalance.sol	Access Control	1	2	31,38	38	31,38
mapping_write.sol	Access Control	0	1	20		20
etherpot_lotto.sol	Unchecked Low Calls	1	0	109,141	141	
king_of_the_ether_throne.sol	Unchecked Low Calls	4	2	110,118,132,174	110,118,132,174	110,174
unchecked_return_value.sol	Unchecked Low Calls	0	1	17		17
roulette.sol	Time Manipulation	0	1	18,20		18
eth_tx_order_dependence_minimal.sol	Front Running	2	1	23,31	23	31
timelock.sol	Arithmetic	1	0	22	22	
tokensalechallenge.sol	Arithmetic	1	1	23,25,33	25	23

análises, incluindo uma coluna que indica a linha onde cada vulnerabilidade foi detectada, bem como colunas que mostram a quantidade de vulnerabilidades e as linhas identificadas por cada ferramenta. A Tabela V.5 refere-se ao Mythril, enquanto a Tabela V.6 corresponde ao Slither. Em ambas, a primeira coluna de contagem corresponde à versão empregada por Durieux et al. [2020] (≤0.21.15 para o Mythril e ≤0.6.6 para o Slither), e a segunda coluna à versão utilizada neste trabalho (0.24.7 e 0.10.0, respectivamente). As colunas de linha indicam os números de linha do contrato onde cada versão registrou a detecção.

As correções realizadas para fins de comparação neste trabalho não alteram o objetivo principal, pois equivalem a refazer as análises anteriores com o intuito de comparar diferentes versões das ferramentas. São apontadas apenas as diferenças, uma vez que foi utilizado o trabalho SmartBugs como referência, e os valores divergentes poderiam gerar dúvidas.

Com as análises manuais, foi possível corrigir a contagem, conforme apresentado nos apêndices. Alguns apontamentos que não seriam identificados pelas análises automáticas foram devidamente ajustados. Assim, constatou-se que, no Mythril, as detecções relacionadas a *Arithmetic* e *Front Running* apresentaram uma piora de uma unidade cada, e *Unchecked Low Calls* apresentou uma

Tabela V.6: Análise de diferença das detecções Slither.

Arquivo	**Vulnerabilidade**	**0.6.6**	**0.10.0**	**Linha**	**0.6.6 Linha**	**0.10.0 Linha**
parity_wallet_bug_2.sol	Access Control	0	1	226,233		233
modifier_reentrancy.sol	Reentrancy	0	1	15		15

piora mais acentuada, com uma redução de duas unidades nas detecções. A nova análise não conseguiu identificar a vulnerabilidade presente no arquivo *timelock.sol*.

Buscou-se compreender por que apenas esse código apresentou um resultado diferente. Considerou-se a hipótese de que uma falha na saída do SmartBugs pudesse ter indicado incorretamente a vulnerabilidade. Contudo, 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 0.21.15 do Mythril. Durante esse processo, surgiram problemas de compatibilidade, possivelmente relacionados à versão do Python utilizada. Para contornar isso, foi necessário corrigir alguns problemas no código do Mythril para que a ferramenta pudesse ser executada.

Com a análise realizada na versão 0.21.15, o Mythril conseguiu identificar exatamente a vulnerabilidade apontada pelo SmartBugs. Dessa forma, confirma-se que, para esses contratos específicos, houve uma piora na capacidade de detecção.

Outra vulnerabilidade que também apresentou piora foi a *Front Running*. A diferença foi observada no arquivo *eth_tx_order_dependence_minimal.sol*. A análise do SmartBugs indicou duas detecções verdadeiras, enquanto a análise deste trabalho identificou apenas uma. O estudo de Munir and Reichenbach [2023] foca exclusivamente na vulnerabilidade *Transaction Ordering Dependency*, correspondente à *Front Running* no DASP. Nesse estudo, o mesmo arquivo é analisado e confirma a primeira vulnerabilidade reportada pelo SmartBugs.

Entre as vulnerabilidades que apresentaram melhorias na detecção pelo Mythril destacam-se *Access Control* e *Time Manipulation*. Além do aumento de detecções verdadeiras, essa melhora reflete um ponto frequentemente citado como dificuldade nos trabalhos referenciados.

No caso específico do Mythril, *Access Control* e *Time Manipulation* apresentaram duas vulnerabilidades adicionais não detectadas na análise anterior. Já a vulnerabilidade *Reentrancy* manteve os mesmos valores das análises anteriores.

Quanto ao Slither, a versão mais recente mostrou melhorias nas vulnerabilidades *Access Control* e *Reentrancy*, com detecções mais precisas. No *Access Control*, a diferença ocorreu no arquivo *parity_wallet_bug2.sol*, enquanto na vulnerabilidade *Reentrancy* a diferença foi registrada no arquivo *modifier_reentrancy.sol*.

As Tabelas V.7, V.8 e V.9 apresentam, respectivamente, os valores de precisão, cobertura e F1-score para as categorias *Access Control*, *Arithmetic*, *Front Running*, *Reentrancy*, *Time Manipulation* e *Unchecked Low Calls*. As categorias *Denial of Service* e *Bad Randomness* foram excluídas. As categorias *Denial of Service* e *Bad Randomness* foram excluídas das Tabelas V.7, V.8 e V.9 pelos seguintes motivos: para *Denial of Service*, nenhuma das quatro combinações de ferramenta e versão avaliadas obteve detecções verdadeiras positivas frente ao gabarito de sete ocorrências do conjunto de dados original de Durieux et al. [2020] (Tabela V.4), o que torna a precisão e o *F1-score*

matematicamente indefinidos (divisão por zero). Já *Bad Randomness* não possui ocorrências catalogadas no gabarito de 69 contratos de Durieux et al. [2020], inviabilizando o cálculo de métricas de desempenho para essa categoria neste experimento. Ambas as categorias são retomadas no Segundo Experimento (Seção V.2), cujo conjunto de dados de 142 contratos contém ocorrências suficientes para sua avaliação quantitativa.

Tabela V.7: Tabela de precisão das ferramentas (Valores arredondados para duas casas decimais).

	Mythril		**Slither**	
Precisão	0.21.15	0.24.7	0.6.6	0.10.0
Access Control	0,20	0,50	0,30	1,00
Arithmetic	0,16	0,66	0,00	0,00
Front Running	0,09	0,05	0,00	0,00
Reentrancy	0,31	0,20	0,46	0,50
Time Manipulation	0,00	0,14	0,20	0,04
Unchecked Low Calls	0,23	0,45	0,07	0,09

Tabela V.8: Tabela de cobertura das ferramentas (valores arredondados para duas casas decimais).

	Mythril		**Slither**	
Cobertura	0.21.15	0.24.7	0.6.6	0.10.0
Access Control	0,21	0,29	0,25	0,29
Arithmetic	0,68	0,64	0,00	0,00
Front Running	0,29	0,14	0,00	0,00
Reentrancy	0,63	0,63	0,88	1,00
Time Manipulation	0,00	0,20	0,20	0,20
Unchecked Low Calls	0,78	0,56	0,11	0,11

Tabela V.9: Tabela de F1-score das ferramentas (Valores arredondados para duas casas decimais).

	Mythril		**Slither**	
F1-score	0.21.15	0.24.7	0.6.6	0.10.0
Access Control	0,21	0,37	0,28	0,45
Arithmetic	0,26	0,65	0,00	0,00
Front Running	0,14	0,07	0,00	0,00
Reentrancy	0,42	0,30	0,61	0,67
Time Manipulation	0,00	0,17	0,20	0,07
Unchecked Low Calls	0,36	0,50	0,09	0,09

A Tabela V.9 sintetiza os resultados por meio do *F1-score*, métrica que equilibra a precisão e a cobertura. A análise integrada das Tabelas V.7, V.8 e V.9 evidencia alguns pontos relevantes sobre a evolução das ferramentas ao longo do tempo.

Na categoria de *Reentrancy*, o Slither 0.10.0 apresentou o desempenho mais consistente, com *F1-score* elevado sustentado pela cobertura próxima do ideal e precisão moderada. Isso mostra que, apesar da presença de falsos positivos, a ferramenta conseguiu garantir cobertura completa para essa vulnerabilidade, configurando-se como a mais confiável no conjunto de dados avaliado.

Em *Arithmetic*, o Mythril 0.24.7 mostrou uma melhora expressiva na precisão (0,66), refletindo um maior controle dos falsos positivos. Diferente das outras categorias, essa evolução não prejudicou significativamente a cobertura, que se manteve estável em 0,64 (comparado a 0,68 na versão anterior). Isso resultou no maior salto de *F1-score* observado, indicando que a versão mais recente da ferramenta tornou-se muito mais eficaz e confiável para identificar vulnerabilidades aritméticas.

Na categoria de *Access Control*, o Slither 0.10.0 alcançou precisão máxima, eliminando falsos positivos. Esse comportamento elevou o *F1-score*, mesmo que a cobertura tenha permanecido em níveis moderados, indicando ganho de confiabilidade nos alertas.

Por outro lado, em *Front Running*, o Mythril 0.24.7 apresentou regressão, com queda no *F1-score* devido à redução na capacidade de detecção. Já em *Unchecked Low Calls*, o cenário é misto: houve um aumento no *F1-score* (de 0,36 para 0,50) impulsionado pela melhor precisão, apesar de uma queda notável na cobertura, indicando que a ferramenta reporta menos alertas, porém mais confiáveis.

De forma geral, os resultados reforçam o dilema recorrente nas ferramentas de análise estática, em que a busca por maior precisão tende a implicar perda de cobertura. As melhorias observadas priorizaram a redução de falsos positivos em categorias críticas, mas ao custo de menor detecção em vulnerabilidades relevantes, confirmando a tendência já discutida na literatura [Chess and West, 2007]. Em relação à RQ0, houve evolução na precisão e na cobertura de detecção entre as versões avaliadas, mas de forma não uniforme: melhorias consideráveis em *Arithmetic* e *Access Control* foram acompanhadas de regressão em *Front Running* e *Unchecked Low Calls*, indicando que a evolução das ferramentas não é linear entre categorias.

V.2 Segundo Experimento

Na segunda etapa, aplicou-se a análise do modelo Gemini 2.0 Flash com base na classificação de vulnerabilidades via normalização semântica. Alinhada à metodologia de Chen et al. [2025], essa abordagem produz uma saída binária por categoria: indica se a vulnerabilidade está presente no arquivo, mas não sua localização exata no código. Os autores avaliaram três versões do modelo GPT, consolidando os resultados por meio de média aritmética entre múltiplas iterações, adotando um gabarito binário por arquivo em que cada contrato é classificado apenas pela presença ou ausência de uma vulnerabilidade específica, ignorando o número de ocorrências. Em ambos os casos, o resultado final atesta a existência ou ausência de cada vulnerabilidade nas nove classes definidas pelo DASP.

Para permitir uma comparação justa, replicou-se essa abordagem utilizando os dados dos arquivos binários disponíveis no repositório oficial[3], comparando o Gemini 2.0 Flash ao GPT-4, o LLM de melhor desempenho entre os avaliados. A Tabela V.10 apresenta os resultados dessa análise

[3] `https://github.com/BugmakerCC/GPT4scv/tree/master/gpt4`

binária.

Tabela V.10: Comparação da Análise Binária: Gemini 2.0 vs. GPT-4

Vulnerabilidade	Métrica	Gemini 2.0 (%)	GPT-4 (%)
Reentrancy	Precisão	26,7	**35,3**
	Cobertura	**100,0**	96,8
	F1-score	42,2	**51,7**
Access Control	Precisão	**18,5**	14,4
	Cobertura	**88,9**	83,3
	F1-score	**30,5**	24,6
Arithmetic	Precisão	15,7	**22,0**
	Cobertura	**93,3**	86,7
	F1-score	26,9	**35,1**
Unchecked ll calls	Precisão	**58,6**	53,9
	Cobertura	**100,0**	96,1
	F1-score	**73,9**	69,0
Denial of Service	Precisão	4,7	**6,7**
	Cobertura	100,0	100,0
	F1-score	8,9	**12,6**
Bad randomness	Precisão	**47,1**	42,1
	Cobertura	100,0	100,0
	F1-score	**64,0**	59,3
Front running	Precisão	3,9	**6,9**
	Cobertura	**100,0**	50,0
	F1-score	7,4	**12,1**
Time manipulation	Precisão	10,6	**13,8**
	Cobertura	**100,0**	80,0
	F1-score	19,2	**23,5**
Short addresses	Precisão	2,9	**7,1**
	Cobertura	100,0	100,0
	F1-score	5,7	**13,3**

Nesta primeira análise, nota-se uma tendência do Gemini 2.0 Flash em apresentar uma maior amplitude na detecção de vulnerabilidades, evidenciada por índices de cobertura superiores em categorias específicas. Contudo, nesta etapa, os valores entre o Gemini 2.0 Flash e o GPT-4 permanecem muito próximos. A diferenciação será mais evidente na análise manual, na qual se observa que algumas detecções preveem vulnerabilidades que não estão totalmente alinhadas com a vulnerabilidade esperada no arquivo.

V.2.1 Resultado Análise Manual

Conforme exposto, com o intuito de examinar a identificação de múltiplas vulnerabilidades em um único contrato e compreender detalhadamente os relatórios gerados, optou-se por realizar esta segunda etapa, expandindo o escopo originalmente proposto no trabalho base. Para tanto, procedeu-se à análise manual das saídas do Gemini 2.0 Flash e do GPT-4, este último selecionado por ter apresentado, dentre os modelos GPT, o melhor desempenho nos testes de análise binária realizados anteriormente. Manteve-se a primeira parte da análise binária, apesar da melhoria apresentada na sequência, com o objetivo de contemplar as abordagens do trabalho de referência e justificar a evolução que motivou a nova metodologia para melhor compreensão do comportamento das ferramentas.

Tabela V.11: Comparativo da Análise Manual: Gemini 2.0 vs. GPT-4

Vulnerabilidade	Métrica	Gemini 2.0 (%)	GPT-4 (%)
Reentrancy	Precisão	19,4	**31,2**
	Cobertura	**93,8**	**93,8**
	F1-score	32,1	**46,9**
Access Control	Precisão	**14,3**	10,8
	Cobertura	**71,4**	66,7
	F1-score	**23,8**	18,5
Arithmetic	Precisão	16,8	**23,3**
	Cobertura	**78,3**	73,9
	F1-score	27,7	**35,4**
Unchecked ll calls	Precisão	48,1	**54,9**
	Cobertura	**85,1**	75,7
	F1-score	61,5	**63,6**
Denial of Service	Precisão	2,5	**5,4**
	Cobertura	**71,4**	**71,4**
	F1-score	4,9	**10,1**
Bad randomness	Precisão	51,9	**73,5**
	Cobertura	46,7	**83,3**
	F1-score	49,1	**78,1**
Front running	Precisão	2,4	**4,0**
	Cobertura	**42,9**	14,3
	F1-score	4,5	**6,2**
Time manipulation	Precisão	12,0	**21,4**
	Cobertura	**85,7**	**85,7**
	F1-score	21,1	**34,3**
Short addresses	Precisão	4,2	**9,1**
	Cobertura	**100,0**	**100,0**
	F1-score	8,0	**16,7**

Ao comparar os dados da V.10, referentes à análise binária, com os da V.11, focados na análise manual, observa-se que a maior variação percentual ocorre na métrica de cobertura do modelo Gemini 2.0 para a categoria *Bad randomness*. Enquanto na análise binária o modelo atinge $100,0\%$, seu desempenho cai para $46,7\%$ na análise manual, resultando em uma variação negativa de $\Delta = 53,3\%$. Outra oscilação drástica no Gemini 2.0 é verificada em *Front running*, onde a cobertura cai de $100,0\%$ para $42,9\%$, uma redução de $\Delta = 57,1\%$.

O motivo desta queda é que, na análise manual, ao detalhar as saídas das análises, o modelo deixou de considerar como positivos alguns arquivos com vulnerabilidades. Isso diminui a cobertura anterior de $100,0\%$, pois algumas vulnerabilidades deixaram de ser cobertas. O melhor exemplo para explicar tal comportamento pode ser observado no arquivo *smart_ billions.sol* [4]. Este arquivo é extenso e possui 12 apontamentos de *Bad randomness*. Na saída binária, por ser uma automação com peso binário (0 ou 1), muitos detalhes da vulnerabilidade são ignorados e, por ser um arquivo extenso, o modelo tende a gerar algum apontamento positivo. Já na análise manual, detectou-se que o relatório do Gemini perde a capacidade de cobertura, mesmo conseguindo apontar algumas vulnerabilidades como descrito no exemplo a seguir da saída de *Bad Randomness*:

[4] `https://github.com/drinith/FONSECA_DVCIEFA_LLMs/blob/master/smartbugs-curated/dataset/bad_randomness/smart_billions.sol`

6. Bad Randomness:

* **Vulnerability:** The ‘play()‘ and ‘playRandom()‘ functions use ‘sha3(msg.sender, block.number)‘ as a source of randomness. This is *not* secure. ‘block.number‘ is predictable, and ‘msg.sender‘ is known. A miner could potentially manipulate the outcome of the lottery by pre-calculating the hash for different block numbers. * **Mitigation:** Use a proper source of randomness, such as a commit-reveal scheme, a verifiable random function (VRF), or an oracle service dedicated to providing randomness.

Ainda que o relatório identifique apenas 2 dos 12 pontos de vulnerabilidade presentes no arquivo, os demais 10 casos permanecem sem cobertura, o que explica a queda expressiva de cobertura observada na análise manual.

No que tange ao *F1-score*, nota-se uma inversão de dominância relevante na categoria de *Bad Randomness*. Na análise binária, o Gemini 2.0 sustenta $64,0\%$ contra $59,3\%$ do GPT-4, mas na análise manual, o GPT-4 salta para $78,1\%$ enquanto o Gemini 2.0 cai para $49,1\%$, gerando uma variação de desempenho em pontos percentuais de $29,0\%$ a favor do modelo da OpenAI. Em contrapartida, na categoria de *Access Control*, o Gemini 2.0 demonstra maior consistência na análise manual com um *F1-Score* de $23,8\%$, superando os $18,5\%$ do GPT-4.

Para a vulnerabilidade de *Arithmetic*, o GPT-4 apresenta uma queda da cobertura, saindo de $86,7\%$ na análise binária para $73,9\%$ na manual, o que representa uma variação de $\Delta = 12,8\%$. Já o Gemini 2.0, embora também apresente queda, mantém uma cobertura superior de $78,3\%$ na análise manual, partindo de $93,3\%$. No geral, a transição da análise binária para a manual tende a reduzir a sensibilidade dos modelos, com variações médias de cobertura que frequentemente ultrapassam a marca de 20%.

Para verificar a estabilidade das métricas, aplicou-se *bootstrap* não-paramétrico com 10.000 reamostras por arquivo, seguindo a abordagem clássica de Tibshirani and Efron [1993] e sua aplicação em comparações de sistemas por métricas automáticas [Koehn, 2004]. Para cada arquivo sorteado, somam-se seus TPs, FPs e o total do gabarito correspondente. O denominador da cobertura é recalculado a partir dos arquivos sorteados em cada iteração, mantendo consistência entre numerador e denominador. Recalcula-se o *F1-score* a cada iteração, gerando intervalos de confiança de 95% (IC95%).

Para comparar formalmente os modelos, o *bootstrap* foi executado de forma pareada: a cada iteração, os dois modelos recebem exatamente o mesmo conjunto de arquivos sorteados, de modo que a única fonte de variação entre eles é o desempenho em cada contrato. O *p-valor* é estimado via *bootstrap* pareado com centralização sob H_0 [Hall and Wilson, 1991][Dror et al., 2018]: a distribuição das diferenças $\delta^* = F1^*_{\text{GPT-4}} - F1^*_{\text{Gemini}}$ é deslocada para média zero antes de comparar com a diferença observada δ_{obs}, de modo que $p = P(\delta^* - \bar{\delta^*} \geq \delta_{\text{obs}} \mid H_0)$ (teste unilateral, H_1: GPT-4 >

Gemini).

A Tabela V.12 apresenta os resultados. Quatro categorias apresentaram diferença estatisticamente significativa ($p < 0{,}05$), todas com vantagem para o GPT-4: *Reentrancy* ($p < 0{,}001$; 47,6 [35,7; 59,6] vs. 32,1 [23,0; 41,6] do Gemini), *Arithmetic* ($p = 0{,}007$; 35,4 [21,1; 47,4] vs. 27,7 [15,4; 39,1]), *Denial of Service* ($p = 0{,}010$; 10,1 [2,2; 18,6] vs. 4,9 [1,0; 9,3]) e *Time Manipulation* ($p = 0{,}012$; 34,3 [6,9; 56,4] vs. 21,1 [3,8; 38,7]). Nas demais categorias, os IC95% se sobrepõem e as diferenças não são estatisticamente distinguíveis. Em *Bad Randomness*, embora a diferença pontual seja expressiva (GPT-4: 78,1 [47,6; 90,9] vs. Gemini: 49,1 [18,9; 80,0]), o teste não rejeita a hipótese nula ($p = 0{,}111$). A amplitude elevada dos IC95% nessa categoria reflete a concentração de 12 ocorrências em um único contrato (*smart_ billions.sol*): quando esse arquivo é ou não sorteado nas reamostras, o resultado agregado varia substancialmente. Em *Access Control*, o Gemini apresenta F1 pontualmente superior (23,8 vs. 18,5), mas a diferença também não é significativa ($p = 0{,}949$). Categorias com poucos contratos no gabarito, como *Front Running* e *Short Addresses*, apresentam intervalos igualmente amplos, evidenciando o limite do poder estatístico do estudo nessas categorias.

Tabela V.12: F1-score com IC95% via *bootstrap* pareado por arquivo (10.000 reamostras, *seed*=42). Asterisco indica $p < 0{,}05$ para H_1: GPT-4 > Gemini.

Vulnerabilidade	Gemini 2.0 F1 [IC95%]	GPT-4 F1 [IC95%]	p-valor
Reentrancy	32,1 [23,0; 41,6]	47,6 [35,7; 59,6]	<0,001*
Access Control	23,8 [13,9; 34,2]	18,5 [10,2; 27,5]	0,949
Arithmetic	27,7 [15,4; 39,1]	35,4 [21,1; 47,4]	0,007*
Unchecked LL Calls	61,5 [50,8; 72,0]	63,6 [54,0; 72,7]	0,309
Denial of Service	4,9 [1,0; 9,3]	10,1 [2,2; 18,6]	0,010*
Bad Randomness	49,1 [18,9; 80,0]	78,1 [47,6; 90,9]	0,111
Front Running	4,5 [0,0; 9,9]	6,2 [0,0; 18,7]	0,344
Time Manipulation	21,1 [3,8; 38,7]	34,3 [6,9; 56,4]	0,012*
Short Addresses	8,0 [0,0; 25,0]	16,7 [0,0; 46,2]	0,145

V.2.2 Apontamentos nas análises manuais

Esta seção apresenta uma síntese das observações obtidas a partir da auditoria manual das saídas geradas pelos LLMs. O objetivo é complementar os resultados quantitativos apresentados anteriormente, discutindo qualitativamente os comportamentos observados durante a análise dos relatórios produzidos pelos modelos. Em particular, busca-se identificar padrões recorrentes de erro, bem como compreender os fatores que contribuem para a ocorrência de falsos positivos e falsos negativos em diferentes categorias de vulnerabilidades. Além disso, foram examinados os casos que mais se destacaram em relação aos resultados quantitativos apresentados na tabela, com o intuito de identificar possíveis padrões de comportamento dos LLMs. Em algumas categorias de vulnerabilidade, também são apresentados trechos dos relatórios gerados pelos modelos, a fim de observar as diferentes formas pelas quais os LLMs descrevem ou justificam a identificação das vulnerabilidades.

Durante essa inspeção manual, foram observados alguns comportamentos característicos, como a tendência ao didatismo excessivo nas respostas, a confusão entre variáveis locais e variáveis de estado, além da dificuldade de classificação correta em contratos com múltiplos componentes ou lógicas de negócio mais complexas. Em diversas situações, os modelos demonstraram capacidade de reconhecer padrões sintáticos frequentemente associados a vulnerabilidades, porém apresentaram limitações na interpretação semântica do contexto de execução da EVM e da lógica de negócio do contrato.

Como consequência, os diagnósticos produzidos pelos LLMs variam entre classificações tecnicamente imprecisas, especulações teóricas sobre riscos potenciais ou, em alguns casos, a não detecção de vulnerabilidades efetivamente exploráveis. A seguir, são apresentados exemplos representativos desses comportamentos para cada categoria analisada, separados por modelo, com o objetivo de ilustrar as principais limitações observadas durante a auditoria manual.

V.2.3 Gemini

Como primeira etapa, é apresentado o levantamento de análises feitas com os relatórios do Gemini 2.0 Flash.

Reentrancy

O exemplo a seguir ilustra um falso positivo na detecção da vulnerabilidade de *Reentrancy*, um comportamento recorrente nas análises do Gemini. No contrato *Splitter* (no arquivo *0xb0510d68f210b7db66e8c7c814f22680f2b8d1d6.sol*), observa-se uma falha de avaliação ao identificar riscos: a priorização da análise sintática de padrões em detrimento da verificação semântica do fluxo de estados. Embora a análise fornecida aponte um risco de reentrância na função *fundPuppets* devido ao uso da primitiva *call.value()*, conforme demonstrado no relatório do Gemini abaixo, trata-se de um falso positivo, pois a estrutura do contrato não permite a exploração efetiva dessa vulnerabilidade:

> Reentrancy: * The ‘fundPuppets‘ function uses ‘call.value(_share).gas(800000)()‘ to send funds to the ‘extra‘ addresses (which are ‘Puppet‘ contracts).
>
> * The ‘Puppet‘ contract’s fallback function then calls an externally controlled target with ‘target[0].call.value(msg.value).gas(600000)();‘. This could potentially cause a reentrancy issue if the target contract calls back to the ‘Splitter‘ contract during the execution of the ‘fundPuppets‘ function. However, since the fallback function does not modify any state, the reentrancy vulnerability might not be exploitable directly. It is however still a good practice to avoid raw calls in fallback functions.

> * The ‘Puppet‘ contract’s fallback function is vulnerable to reentrancy if target[0] calls back into the contract.

Isso porque, ao analisar o código sob a perspectiva da regra de negócio, nota-se que a estrutura apontada não representa uma ameaça explorável. Primeiramente, o uso da verificação de controle de acesso *require(msg.sender == owner)* restringe o escopo de execução, impedindo que agentes externos iniciem o fluxo necessário para a exploração. Em segundo lugar, inexistem modificações de estado críticas após a interação externa. Em um contrato vulnerável, o código subtrairia o saldo do usuário apenas após a transferência, criando uma brecha para saques repetidos. No *Splitter*, o valor da parcela *_share* é calculado previamente com base no *msg.value* recebido no momento, e não em um saldo acumulado. Como não há um “saldo corrente” do usuário sendo subtraído ou atualizado após o envio, mesmo que um atacante reentrasse na função, não haveria estado interno para manipular, tornando a *Reentrancy* inofensiva. Portanto, a classificação de reentrância configura um falso positivo. Em suma, o comportamento do LLM revela uma dificuldade em avaliar a regra de negócio do programa, levando-a a superestimar vulnerabilidades ao focar excessivamente em assinaturas comuns.

Access Control

Ao analisar os aspectos relacionados a *Access Control*, observa-se que a capacidade dos LLMs de reconhecer padrões superficiais de vulnerabilidade nem sempre implica a compreensão adequada dos mecanismos técnicos que tornam uma falha explorável. Diferentemente do caso discutido anteriormente, nesta análise ocorre um falso negativo: a vulnerabilidade existente não é corretamente identificada pelo modelo. O contrato *arbitrary_location_write_simple.sol* ilustra essa limitação ao evidenciar uma discrepância entre o reconhecimento de padrões e a análise semântica do código.

Nesse contrato, a vulnerabilidade decorre da manipulação do tamanho do array `bonusCodes`. Especificamente, a função `PopBonusCode` permite reduzir o valor de `length` sem uma verificação adequada. Como a variável é do tipo `uint`, a condição utilizada no `require` é sempre verdadeira, o que possibilita que a operação de decremento provoque um *underflow*. Como consequência, o tamanho do array pode assumir um valor extremamente grande, permitindo a escrita em posições arbitrárias do *storage* do contrato. O trecho de código a seguir apresenta esse comportamento:

```
function PopBonusCode() public {
    require(0 <= bonusCodes.length); // Sempre verdadeiro (uint)
    bonusCodes.length--; // Underflow potencial que expande o storage
}
```

O relatório gerado pelo Gemini identifica apenas a ausência de controle de acesso nas funções relacionadas ao array:

> PushBonusCode(), PopBonusCode(), UpdateBonusCodeAt(): These functions have no access control. Anyone can add, remove, or modify bonus codes. This could be intentional, but it's worth noting as a potential vulnerability.

Embora o LLM reconheça corretamente a ausência de modificadores de acesso, sua análise permanece superficial. O problema central não se limita à falta de restrições como *onlyOwner*, mas sim ao fato de que a função *PopBonusCode* pode provocar um *underflow* na variável de estado *bonusCodes.length*. Esse comportamento permite expandir artificialmente o tamanho do array e, consequentemente, possibilita a manipulação arbitrária do *storage* do contrato.

Assim, o modelo identifica um sintoma potencial (a ausência de controle de acesso), mas falha em reconhecer o mecanismo técnico que torna a vulnerabilidade explorável. Essa limitação caracteriza um caso de falso negativo, no qual a falha de segurança existente não é devidamente compreendida ou classificada pela análise automatizada.

Arithmetic

Neste caso, ocorreu um problema também relatado por Chen et al. [2025]. A presença da biblioteca SafeMath no contrato *BECToken.sol* induziu os LLMs a assumirem que todas as operações aritméticas estavam protegidas contra *overflow*. Entretanto, na função *batchTransfer*, o desenvolvedor utiliza o operador padrão (*) em vez da função segura *.mul()*. Como resultado, a operação crítica *uint256 amount = uint256(cnt) * _value* permanece vulnerável a *overflow*. Ambos os LLMs falharam em identificar que a biblioteca não foi aplicada nessa linha específica, classificando incorretamente o contrato como seguro. O relatório do Gemini ilustra esse comportamento ao afirmar que nenhuma vulnerabilidade aritmética foi detectada devido ao uso de *SafeMath*.

> 3. **Arithmetic Issues:**
>
> * The 'SafeMath' library is used throughout the contract for all arithmetic operations. This protects against integer overflows and underflows, which is excellent. * **No immediate vulnerabilities detected** due to the use of 'SafeMath'.

Além disso, observou-se que tanto o GPT-4 quanto o Gemini 2.0 Flash tendem a ignorar vulnerabilidades associadas a variáveis locais. Das seis falhas presentes, apenas três, relacionadas ao estado persistente, foram detectadas, evidenciando uma lacuna na análise de fluxos de dados temporários.

Unchecked Low-Level Calls

Esta vulnerabilidade apresentou um desempenho relativamente bom nas análises. Os LLMs demonstraram uma boa capacidade de cobertura, mesmo sendo a vulnerabilidade com maior repre-

sentação em códigos do conjunto de dados. Contudo, no código Solidity *0x89c1b3807d4c67df034fff b62f3509561218d30b.sol*, é possível observar um comportamento que chama a atenção no relatório de vulnerabilidades gerado para esse arquivo:

> **4. Unchecked Return Values For Low Level Calls:**
>
> * **Vulnerability Present:** Yes. * **Location:** * Numerous instances of '!.call.value(...)' are used without checking the return value. For example, in 'request()', 'cancel()', 'upgrade()', 'reset()', 'suspend()', 'restart()', 'withdraw()' and 'deliver()' functions. * **Explanation:** * The 'call.value()' function returns a boolean indicating success or failure. If the call fails (e.g., due to out of gas or the receiving contract rejecting the transfer), the 'call.value()' function returns 'false', but the code doesn't check for this. The transaction doesn't revert, and the contract's state might be left in an inconsistent state. * **Mitigation:** * Always check the return value of 'call.value()' calls:
>
> "'solidity if (!msg.sender.call.value(msg.value)()) // Handle the failure appropriately, such as reverting the transaction throw; "'

A maioria dos relatos que apontam essa vulnerabilidade está associada à estrutura *call.value()*. No entanto, neste código também existe o uso de *.send*, especificamente em *SGX_ADDRESS.send*, que não foi identificado no relatório gerado. Acredita-se que isso possa ocorrer porque muitos exemplos conhecidos desse tipo de vulnerabilidade estão associados ao uso de *call*.

Outro ponto relevante é que, nesse código, *SGX_ADDRESS* é declarado como uma constante do tipo *address*. Por se tratar de um endereço fixo no contrato, o modelo pode interpretá-lo implicitamente como um destino confiável, o que reduz a probabilidade de que o uso de *.send* seja identificado como um padrão potencialmente vulnerável. Essa característica pode ter contribuído para camuflar ainda mais o padrão de chamada de baixo nível durante a análise realizada pela LLM. Assim, o modelo identifica apenas parcialmente a presença desta vulnerabilidade no código, resultando na ocorrência de alguns falsos negativos.

Denial of Service

Esta vulnerabilidade apresentou o maior número de ocorrências de falsos positivos, totalizando 194 casos, o que evidencia sua baixa precisão em contraponto à boa cobertura quando comparada aos resultados da análise manual. Um dos fatores que contribuem para esse comportamento é a tendência do Gemini de interpretar diferentes cenários potenciais como situações capazes de tornar o contrato inteligente indisponível ou inoperante. Em diversas situações, qualquer possibilidade de comportamento inesperado ou falha de execução é interpretada pelo modelo como um possível cenário de *Denial of Service*.

Um exemplo desse comportamento pode ser observado no contrato em Solidity *parity_wallet_bug_2.sol*, conforme apresentado no seguinte trecho do relatório gerado:

> 5. **Denial of Service:**
>
> * **Gas Limit:** * **‘reorganizeOwners()‘ Function:** This function iterates through ‘m_owners‘ and reorganizes the array. If there are many owners and several of them have been removed, this function could potentially consume a large amount of gas and revert due to exceeding the block gas limit. This would prevent removing owners and could effectively freeze the wallet. * **‘clearPending()‘ Function:** Iterates through ‘m_pendingIndex‘ and deletes transactions. A large number of pending transactions could cause this function to run out of gas, preventing certain wallet operations. * **Unexpected Throw/Kill:** A malicious owner could deploy a contract that always throws an exception when called. This could be used to DoS the wallet if it attempts to send funds to that contract. Mitigation: Implement some form of blacklisting or reputation system for target contracts. * **‘addOwner‘ Function:** If ‘c_maxOwners‘ is reached, ‘reorganizeOwners()‘ is called. If the ‘reorganizeOwners‘ still cannot find a free spot to add the new owner, the function simply returns, preventing new owner to be added. This is a potential DoS vulnerability. * **‘removeOwner‘ Function:** If ‘m_required‘ is greater than ‘m_numOwners - 1‘, the function returns. This is a potential DoS if the owners want to remove an owner but cannot.

No relatório, afirma-se que um loop interno poderia consumir uma grande quantidade de gás e ultrapassar o limite de gás do bloco. Contudo, o contrato define a constante *c_maxOwners = 250*, que limita o número máximo de proprietários e, consequentemente, restringe o crescimento das estruturas percorridas pelos loops. Esse limite reduz significativamente a probabilidade de que tais iterações resultem em consumo excessivo de gás em condições normais de uso.

Outro exemplo de especulação de *Denial of Service* pode ser observado nos contratos *integer_overflow_1.sol* e *insecure_transfer.sol*. Nesses casos, o Gemini frequentemente interpreta que um estouro de variável poderia levar a um cenário de negação de serviço, mesmo em contratos compostos por funções isoladas e sem dependência crítica de estado compartilhado. Dessa forma, observa-se que o modelo tende a extrapolar possíveis consequências de determinadas vulnerabilidades, classificando seus desdobramentos como potenciais cenários de *Denial of Service*.

Bad Randomness

No caso da vulnerabilidade de *Bad Randomness*, observa-se que alguns fatores relacionados ao tamanho dos contratos e à quantidade de ocorrências da vulnerabilidade podem ter impactado a

cobertura do Gemini. Exemplos como o contrato *lucky_ doubler.sol*, que possui cinco vulnerabilidades e um tamanho superior à média dos demais contratos analisados, e o *smart_ billions.sol*, que é significativamente maior que a maioria dos exemplos e contém doze ocorrências dessa vulnerabilidade, ilustram esse cenário.

Em contratos mais extensos e com múltiplas ocorrências de uma mesma vulnerabilidade, a tarefa de detecção torna-se mais complexa, pois o modelo precisa não apenas identificar o padrão vulnerável, mas também reconhecer suas diferentes instâncias ao longo do código. No caso específico de *smart_ billions.sol*, a lógica relacionada ao mecanismo de sorteio está distribuída em diferentes partes do contrato, exigindo maior capacidade de rastreamento de contexto e compreensão do fluxo lógico do programa.

Dessa forma, a necessidade de analisar contratos maiores e identificar múltiplas ocorrências da mesma vulnerabilidade pode ter impactado o desempenho do modelo, reduzindo sua cobertura na detecção de *Bad Randomness* nesses casos.

Front Running

O *Front Running* também foi uma vulnerabilidade que apresentou uma precisão mais baixa, sendo a menor em termos percentuais, como pode ser observado na Tabela V.11. O que mais chamou atenção foram análises superficiais que identificavam apenas a possibilidade de exposição de variáveis ou disputas na ordem de execução das transações, sem avaliar se existia um cenário real de exploração associado ao *Front-Running*. Um exemplo ocorre no relatório referente ao contrato *0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f.sol*:

> **Front-Running:** * **Potentially Vulnerable (Set Functions):** The ‘SetMinSum‘, ‘SetLogFile‘, and ‘Initialized‘ functions are vulnerable to front-running. An attacker could observe a transaction setting these values and submit their own transaction with a higher gas price to have their transaction executed first, potentially setting the values to something malicious.

Ao analisar o código do contrato citado, observa-se que o Gemini identificou um possível cenário de *Front-Running* apenas pela presença de funções públicas responsáveis por configurar variáveis do contrato. Entretanto, a análise não considera a lógica de execução do contrato nem se a antecipação da transação traria algum benefício econômico real ao atacante, característica normalmente associada a esse tipo de ataque.

Esse tipo de classificação indica que a análise priorizou padrões superficiais de vulnerabilidade, sem avaliar o contexto do contrato. Outro fator que possivelmente contribuiu para esse tipo de interpretação foi o uso de padrões mais antigos do Solidity, como *sha3(msg.data)*, presentes em

contratos históricos como *parity_wallet_bug_1.sol*.

Time Manipulation

Nesta vulnerabilidade, um ponto recorrente ocorre com a presença do alias *now*, como pode ser observado no relatório:

> 8. **Time Manipulation:** * **Vulnerable:** The 'LogFile' contract uses 'now' (block timestamp) in the 'AddMessage' function. Miners have some control over the timestamp of a block, and this could be manipulated, although to a limited degree (+/- 900 seconds of the actual time). The 'PERSONAL_BANK' contract is not vulnerable, but its use of a logfile that is time-dependent introduces a slight risk.

O LLM gera alertas baseando-se na presença do uso de *now* e, em alguns momentos, aponta o código como potencialmente vulnerável, como ocorre no contrato *0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f.sol*. No entanto, nesse caso o timestamp é utilizado apenas para registrar eventos em um log, não influenciando diretamente a lógica financeira ou o comportamento do contrato.

Dessa forma, mesmo que mineradores possuam alguma capacidade de manipular o timestamp do bloco, essa alteração não gera impacto prático na execução do contrato. Isso indica que o LLM identifica padrões associados a vulnerabilidades, mas não avalia adequadamente o contexto da aplicação e a regra de negócio específica do programa.

Short Address Attack

O ataque clássico de *Short Address Attack* ocorre quando um contrato espera um endereço (20 bytes) seguido de um parâmetro adicional do tipo *uint256*. Nesse cenário, um atacante pode enviar um endereço truncado (com menos de 20 bytes). A EVM então preenche os bytes faltantes com zeros à direita, o que desloca os dados subsequentes e altera o valor do parâmetro seguinte.

Observa-se que o Gemini, em alguns casos, apresenta um comportamento excessivamente alarmista ao identificar a simples presença de parâmetros do tipo *address*. Isso pode ser observado no relatório referente ao contrato *0x4e73b32ed6c35f570686b89848e5f39f20ecc106.sol*.

> 9. **Short Address Attack:**
>
> * This contract *could* be vulnerable to a short address attack, though it's less likely given the use of 'uint256' for balances. However, the '_log' address in 'SetLogFile' and '_adr' in 'AddMessage' could be maliciously crafted short addresses.

Entretanto, ao analisar a função *SetLogFile(address _log)*, observa-se que ela possui apenas um único parâmetro. Como não existe um parâmetro subsequente do tipo *uint256*, não ocorre o

deslocamento de valores característico desse tipo de ataque. Dessa forma, o alerta gerado pelo LLM novamente se baseia apenas em padrões sintáticos, sem considerar a estrutura real da chamada da função e o contexto de execução do contrato.

V.2.4 GPT-4

Como segunda etapa, é apresentado o levantamento de análises feitas com os relatórios do GPT-4.

Reentrancy

No relatório do contrato *PoCGame.sol* do arquivo *0x7d09edb07d23acb532a82be3da5c17d9d85806b4*, algumas vulnerabilidades são apontadas:

> 1. Reentrancy: The functions 'loseWager', 'donateToWhale' and 'payout' are vulnerable to reentrancy attacks, as they make external calls to unknown contracts ('whale' and 'winner') before updating the state ('totalDonated'). An attacker can make a recursive call to drain the contract.

Entretanto, ao analisar o fluxo de execução do contrato, observa-se que o padrão *Checks-Effects-Interactions* é corretamente aplicado. O estado do jogador é invalidado antes de qualquer interação externa, redefinindo *timestamps[msg.sender] = 0* e *wagers[msg.sender] = 0*. Dessa forma, uma eventual chamada reentrante é bloqueada pelo modificador *onlyPlayers*, que exige *require(wagers[msg.sender] > 0)*. Como esse valor já foi redefinido, uma segunda chamada falha na verificação, impedindo a exploração da reentrância.

Assim, o alerta de reentrância configura um falso positivo decorrente de uma análise que considera apenas a presença de chamadas externas, sem avaliar adequadamente a ordem das operações no fluxo de execução.

A vulnerabilidade real do contrato está no uso de chamadas de baixo nível sem verificação adequada do valor de retorno, classificadas como *Unchecked_ll_Calls*. Nesse contexto, o modelo provavelmente foi influenciado pela presença da estrutura sintática *call.value()*, frequentemente associada a cenários clássicos de reentrância.

Access Control

O alerta de controle de acesso no arquivo *0x89c1b3807d4c67df034fffb62f3509561218d30b.sol* identificado pela análise automatizada constitui um falso positivo decorrente da forma incomum como o contrato armazena o endereço do administrador. Embora as funções *upgrade()*, *reset()*, *suspend()*, *restart()* e *withdraw()* sejam declaradas como *public*, o contrato verifica se *msg.sender*

corresponde a *requests[0].requester*. Esse padrão provavelmente levou o GPT-4 a interpretar que o valor pode ser controlado por usuários, pois está armazenado em um vetor de estruturas. Entretanto, no construtor, *requests[0].requester* é inicializado com o endereço do responsável pelo *deploy* e *requestCnt* é definido como 1, garantindo que requisições reais comecem apenas a partir do índice 1. Dessa forma, a posição *requests[0]* nunca é modificada por usuários externos e funciona, na prática, como armazenamento permanente do proprietário do contrato, tornando a verificação *msg.sender == requests[0].requester* equivalente a um controle de acesso tradicional baseado em proprietário.

> 2. Access Control: In the ‘upgrade()‘, ‘reset()‘, ‘suspend()‘, ‘restart()‘, and ‘withdraw()‘ functions, the only access control is checking if ‘msg.sender‘ is the ‘requester‘ of the first request (‘requests[0].requester‘). This is too weak as it only checks the first request and not the subsequent ones. If the first request is made by an attacker, they can gain control of the contract.

A análise do modelo assume que o controle de acesso é fraco por depender apenas da primeira posição do vetor *requests*. No entanto, no construtor do contrato, *requests[0].requester* é inicializado com o endereço responsável pelo *deploy*, enquanto *requestCnt* é definido como 1. Com isso, as requisições criadas durante a execução do contrato passam a ocupar posições a partir do índice 1.

Dessa forma, o valor armazenado em *requests[0].requester* permanece constante ao longo da execução do contrato e funciona, na prática, como o identificador do proprietário. Assim, a verificação *msg.sender == requests[0].requester* implementa efetivamente um mecanismo de controle de acesso equivalente a um padrão tradicional de proprietário.

Portanto, o alerta configura um falso positivo decorrente de uma análise que considera apenas a estrutura do código, sem avaliar corretamente a forma como o mecanismo de controle de acesso é inicializado e utilizado pelo contrato.

Arithmetic

No relatório do contrato *wallet_03_wrong_constructor.sol*, foi apontada uma possível vulnerabilidade aritmética relacionada à função *withdraw*. Segundo a análise automatizada, a operação de subtração realizada na atualização do saldo poderia resultar em um *integer underflow*.

> Arithmetic Issues: While the ‘deposit‘ function has a check to prevent overflow, the ‘withdraw‘ function doesn’t have a underflow check when deducting the ‘amount‘ from ‘balances[msg.sender]‘. This could lead to an integer underflow if ‘amount‘ is larger than ‘balances[msg.sender]‘, making ‘balances[msg.sender]‘ a very large number.

Entretanto, ao analisar a implementação da função *withdraw*, observa-se que o contrato realiza previamente uma verificação por meio da instrução *require(amount <= balances[msg.sender])*. Essa

condição garante que o valor solicitado para saque nunca seja superior ao saldo armazenado para o usuário.

Dessa forma, a operação subsequente *balances[msg.sender] -= amount* é executada apenas quando a condição de segurança é satisfeita, impedindo a ocorrência de *underflow*. Caso o valor solicitado seja maior que o saldo disponível, a execução da função é revertida antes que qualquer alteração no estado seja realizada.

Portanto, a possibilidade de *integer underflow* mencionada no relatório não se concretiza na prática, pois o fluxo de execução do contrato impede que a operação aritmética seja realizada em condições inseguras. O alerta gerado caracteriza-se, assim, como um falso positivo decorrente de uma análise que considera apenas a presença de uma operação de subtração sobre uma variável de saldo, sem levar em conta as restrições impostas pelas verificações condicionais presentes no código.

Unchecked Low-Level Calls

No relatório do contrato *reentrancy_ insecure.sol*, a análise automatizada aponta um possível problema relacionado ao uso de chamadas de baixo nível. O alerta está associado ao uso da instrução *msg.sender.call.value(amountToWithdraw)()*, frequentemente considerada uma operação sensível em contratos escritos em versões antigas de Solidity.

> 2. Unchecked Return Values For Low Level Calls: The contract does not handle the return value of "msg.sender.call.value(amountToWithdraw)()". If the call fails, the contract will continue executing and set "userBalances[msg.sender] = 0". This is not necessarily a vulnerability in this case because the contract uses "require(success);"to ensure the call was successful. However, this could become a vulnerability if the contract's logic changes in the future.

A identificação desse alerta está provavelmente relacionada ao fato de que chamadas de baixo nível, como *call.value()*, normalmente representam um padrão associado a vulnerabilidades quando o valor de retorno não é verificado. Em muitos contratos vulneráveis, esse tipo de chamada é utilizado sem capturar o resultado da operação, permitindo que a execução continue mesmo quando a transferência falha.

Nesse contexto, a presença da instrução *call.value()* pode levar o modelo a assumir automaticamente a existência de um cenário de *Unchecked Low-Level Call*, especialmente porque esse padrão aparece com frequência em exemplos de vulnerabilidades documentadas. Além disso, a chamada externa ocorre antes da atualização do estado do contrato, o que reforça a associação com padrões considerados inseguros em análises automatizadas.

Entretanto, ao analisar o fluxo de execução da função, observa-se que o valor de retorno da

chamada é explicitamente armazenado em uma variável booleana e imediatamente verificado por meio da instrução *require(success);*. Dessa forma, caso a chamada externa falhe, toda a execução da transação é revertida e nenhuma modificação no estado do contrato é efetivada.

Assim, o alerta gerado não corresponde a um caso real de *Unchecked Low-Level Call*. A classificação decorre, principalmente, da presença de padrões sintáticos frequentemente associados a vulnerabilidades, sem que o modelo considere adequadamente a verificação explícita do valor de retorno no fluxo de execução do contrato.

Denial of Service

No relatório gerado para o contrato analisado *0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12.sol*, a ferramenta automatizada aponta a possível presença de uma vulnerabilidade de *Denial of Service*. A justificativa apresentada está associada à ausência de controle de acesso em uma função auxiliar e ao uso de chamadas externas durante a operação de retirada de fundos.

> 4. Denial of Service: As mentioned above, the lack of access control on the AddMessage function could lead to a Denial of Service attack. Additionally, if a contract calling the CashOut function includes a fallback function that always throws an error, it could make it impossible for other users to withdraw their funds.

A classificação pode estar relacionada ao reconhecimento de padrões frequentemente associados a cenários de negação de serviço em contratos inteligentes. No primeiro caso, o relatório menciona a ausência de controle de acesso na função *AddMessage*. Provavelmente o LLM verificando as funções públicas que modificam o estado do contrato sem restrições pode interpretá-las como potenciais pontos vulneráveis, especialmente quando envolvem armazenamento de dados ou estruturas que podem crescer indefinidamente. Assim, é possível que o modelo tenha associado esse padrão ao risco de suspensão do funcionamento do contrato.

O segundo ponto levantado refere-se à presença de uma chamada externa na função *CashOut*. Em alguns cenários descritos na literatura, contratos que dependem do sucesso de chamadas externas podem apresentar comportamentos semelhantes a *Denial of Service*, particularmente quando falhas em uma transferência levam à reversão de toda a transação. Nesse contexto, a presença de uma chamada externa pode ter levado o modelo a associar esse padrão ao risco de bloqueio do contrato, especialmente considerando a possibilidade de uma *fallback function* que sempre reverta a execução.

Entretanto, uma análise preliminar do fluxo de execução sugere que essas condições, isoladamente, podem não caracterizar um cenário claro de *Denial of Service*. A função *AddMessage* parece atuar apenas como mecanismo de registro, enquanto a função *CashOut* processa retiradas de forma individual. Dessa forma, o alerta pode estar relacionado principalmente à identificação de padrões

sintáticos frequentemente associados a vulnerabilidades conhecidas.

Bad Randomness

Nesta vulnerabilidade, analisando o relatório do arquivo analisado *0x524960d55174d912768678d8c606b4d50b79d7b1.sol*, a ferramenta automatizada também aponta a possível presença de um problema relacionado a *Bad Randomness*. A justificativa apresentada indica que determinados elementos do código poderiam levar a resultados previsíveis.

> 5. Bad Randomness: While there isn't a clear use of randomness in this contract, the use of addresses and the keccak256 hash function can be manipulated to produce predictable outcomes.

A identificação desse alerta possivelmente está associada ao reconhecimento de padrões frequentemente discutidos na literatura de segurança de contratos inteligentes. Em diversos casos documentados, mecanismos de geração de aleatoriedade baseados em valores disponíveis no ambiente da *blockchain*, como endereços, *timestamps* ou hashes, podem resultar em valores previsíveis ou manipuláveis. Assim, sempre que uma função criptográfica como *keccak256* aparece em conjunto com parâmetros que podem ser controlados ou antecipados por usuários, ferramentas automatizadas podem interpretar esse padrão como um possível indicativo de geração de aleatoriedade insegura.

Entretanto, no trecho analisado não há evidência clara de que o contrato utilize *keccak256* com o objetivo explícito de gerar números aleatórios para decisões críticas do sistema, como seleção de vencedores, distribuição de recompensas ou controle de fluxos financeiros. Dessa forma, a classificação pode ter sido motivada principalmente pela presença de padrões sintáticos que, em outros contextos, estão associados a implementações inadequadas de aleatoriedade. Assim como em outros alertas observados, essa identificação pode refletir uma correspondência heurística baseada em padrões conhecidos, sendo necessária uma análise mais aprofundada para confirmar se existe, de fato, um problema de *Bad Randomness* no contrato.

Front Running

Ao analisar possíveis falsos positivos relacionados a *Front-Running*, percebe-se que o relatório gerado pela ferramenta automatizada aponta um risco potencial associado ao uso da função *transferFrom* do padrão ERC20.

> 7. Front-Running: While the contract itself does not have explicit dependencies on transaction order, the use of the ERC20 transferFrom function may be vulnerable to front-running attacks.

Esta função pode permitir que um agente observe transações pendentes no *mempool* e envie uma transação concorrente com maior taxa de *gas*, explorando a ordem de execução das transações para obter vantagem. Nesse contexto, a simples presença da função *transferFrom* pode ter levado o modelo a associar automaticamente o contrato a um possível cenário de *Front-Running*, uma vez que esse padrão aparece com frequência em discussões sobre vulnerabilidades relacionadas a manipulação da ordem das transações.

Entretanto, uma análise do fluxo de execução sugere que a presença da função *transferFrom* não é, por si só, suficiente para caracterizar um cenário de *Front-Running*. Embora a função permita a movimentação de *tokens* previamente autorizados por um usuário, sua execução continua condicionada às permissões definidas no contrato do token, por meio do mecanismo de *allowance*. Dessa forma, um agente externo não pode simplesmente explorar a função sem possuir autorização prévia para movimentar os *tokens*. Além disso, no código analisado não há indícios claros de que a lógica do contrato dependa criticamente da ordem de execução das transações. Assim, o alerta parece resultar principalmente da identificação de um padrão frequentemente associado a esse tipo de vulnerabilidade, sem que haja evidências suficientes de um cenário concreto de exploração no contrato analisado.

Time Manipulation

Ao analisar possíveis falsos positivos relacionados a *Time Manipulation*, observa-se que o relatório aponta um risco associado ao uso do valor de *block.number* para verificar a expiração de apostas no contrato.

> 8. Time Manipulation: The contract uses ‘block.number‘ for checking bet expiration, which can be manipulated by miners to a certain degree.

A identificação desse alerta provavelmente está relacionada ao fato de que variáveis relacionadas ao bloco atual, como *block.timestamp* ou *block.number*, são frequentemente discutidas na literatura como fontes potenciais de manipulação temporal em contratos inteligentes. Em alguns cenários, mineradores possuem uma pequena margem de influência sobre certos parâmetros do bloco, especialmente no caso de *block.timestamp*. Por essa razão, ferramentas automatizadas tendem a tratar o uso dessas variáveis em lógica crítica do contrato como um possível indicativo de vulnerabilidade.

No caso analisado, o uso de *block.number* parece estar relacionado apenas à verificação do momento em que uma aposta expira. Embora mineradores tenham alguma capacidade limitada de influenciar características do bloco, o número do bloco é determinado de forma sequencial pela própria rede, o que reduz significativamente o grau de manipulação possível em comparação com outras variáveis temporais. Dessa forma, a simples presença de *block.number* em uma verificação

temporal pode ter levado o modelo a associar o padrão a um possível cenário de *Time Manipulation*.

Short Address Attack

Ao analisar possíveis falsos positivos relacionados a *Short Address Attack*, observa-se que o relatório aponta um risco potencial associado ao uso de endereços em interações com o contrato.

> 9. Short Address Attack: This contract could be susceptible to a short address attack. When interacting with ‘caddress‘, if an address is passed with less than 20 bytes, Solidity does not revert and instead pads the address with zeros. This could cause tokens to be sent to an unintended address.

A identificação desse alerta provavelmente está relacionada a um padrão historicamente discutido em contratos que interagem com *tokens* ou recebem parâmetros diretamente de transações externas. Em versões mais antigas do ecossistema Ethereum, ataques de *Short Address* eram mencionados em situações nas quais dados de entrada malformados poderiam levar a interpretações incorretas dos parâmetros enviados à função, especialmente quando ferramentas externas construíam manualmente as transações. Dessa forma, a presença de parâmetros do tipo *address* ou de chamadas envolvendo endereços pode levar ferramentas automatizadas a associar esse padrão a um possível risco desse tipo.

Entretanto, no contexto atual da linguagem Solidity e do mecanismo de codificação de parâmetros da ABI, argumentos enviados a funções são estruturados e preenchidos automaticamente de acordo com o formato esperado. Assim, mesmo que um endereço seja fornecido com tamanho incorreto em uma chamada externa, a codificação da ABI tende a padronizar o formato dos parâmetros antes da execução da função. Dessa forma, a simples presença de uma variável do tipo *address*, como *caddress*, não é suficiente para caracterizar um cenário real de *Short Address Attack*. O alerta gerado pode, portanto, estar relacionado principalmente ao reconhecimento de um padrão historicamente associado a essa vulnerabilidade, sem que haja evidência clara de uma condição explorável no contrato analisado.

V.3 Discussão Integrada: Ferramentas Estáticas vs. LLMs

Os dois experimentos deste trabalho utilizam conjuntos de dados de tamanhos distintos, 69 contratos no Primeiro Experimento, seguindo o estudo-base de Durieux et al. [2020], e 142 contratos no Segundo Experimento, seguindo Chen et al. [2025] o que impede, em sentido estrito, uma comparação *head-to-head* entre ferramentas estáticas e LLMs sobre o mesmo conjunto de contratos. Essa limitação é reconhecida e discutida na Seção V.4. Ainda assim, como os 142 contratos da versão atual do SmartBugs Curated constituem uma expansão do mesmo conjunto de dados que

originou os 69 contratos do Primeiro Experimento [Durieux et al., 2020], as seis categorias DASP com métricas válidas em ambos os experimentos (*Access Control, Arithmetic, Reentrancy, Time Manipulation, Unchecked Low-Level Calls* e *Front Running*) permitem uma leitura qualitativa comparativa, organizada na Tabela V.13.

Tabela V.13: Síntese comparativa de F1-score (%) entre a melhor ferramenta estática por categoria (Primeiro Experimento, 69 contratos) e os LLMs avaliados na análise manual (Segundo Experimento, 142 contratos).

Vulnerabilidade	**Melhor ferramenta estática**	**Gemini 2.0 (manual)**	**GPT-4 (manual)**
Access Control	45,0 (Slither 0.10.0)	23,8	18,5
Arithmetic	65,0 (Mythril 0.24.7)	27,7	35,4
Reentrancy	67,0 (Slither 0.10.0)	32,1	47,6
Time Manipulation	17,0 (Mythril 0.24.7)	21,1	34,3
Unchecked LL Calls	50,0 (Mythril 0.24.7)	61,5	63,6
Front Running	7,0 (Mythril 0.24.7)	4,5	6,2

A leitura da Tabela V.13 sugere um padrão consistente com a complementaridade discutida na Conclusão (Capítulo VI): em categorias com forte assinatura sintática e regras de detecção bem delimitadas, como *Reentrancy* e *Access Control*, a melhor ferramenta estática supera ambos os LLMs por margem considerável (até 21,2 pontos percentuais de *F1-score* acima do melhor modelo, em *Access Control*). Já em *Unchecked Low-Level Calls*, categoria com grande volume de exemplos no conjunto de dados de treinamento dos LLMs e padrões de código relativamente uniformes (uso de `call.value()`, `.send()`), os LLMs superam a melhor ferramenta estática. Já em *Time Manipulation*, observa-se um padrão semelhante: a melhor ferramenta estática (17,0) é superada por ambos os LLMs, com destaque para o GPT-4 (34,3), o que sugere que o reconhecimento de dependências de `block.timestamp` no relato textual do modelo captura casos que escapam às regras sintáticas das ferramentas tradicionais. Por outro lado, em *Front Running*, todas as abordagens apresentam desempenho baixo (F1 igual ou abaixo de 7,0 em qualquer ferramenta ou modelo avaliado), o que indica que a dificuldade nessa categoria não é exclusiva de uma técnica específica, mas reflete a própria natureza da vulnerabilidade, que depende de contexto de execução (ordem de transações, comportamento de mineradores) difícil de inferir apenas a partir do código-fonte estático.

É importante notar que essa comparação tem caráter qualitativo e ilustrativo, e não estatístico: as ferramentas estáticas e os LLMs foram avaliados sobre conjuntos de contratos diferentes (69 vs. 142), com gabaritos parcialmente distintos, de modo que as diferenças observadas não passaram por teste de significância e podem refletir, em parte, características específicas de cada conjunto de dados, e não apenas da técnica de detecção. Uma comparação direta e estatisticamente válida exigiria executar Mythril e Slither sobre os mesmos 142 contratos utilizados no Segundo Experimento, o que constitui uma direção natural para trabalhos futuros (Capítulo VI). Ainda assim, o padrão qualitativo observado, desempenho superior das ferramentas estáticas em categorias bem

delimitadas, contraposto ao desempenho superior dos LLMs em categorias com padrões sintáticos mais difusos, ilustra a complementaridade entre as duas abordagens nesse *benchmark*, sem que se possa, a partir desses dados, generalizar a magnitude ou a robustez estatística dessa diferença.

V.4 Ameaças à Validade

Em relação à **validade interna**, os resultados do GPT-4 utilizados na comparação foram obtidos do repositório de Chen et al. [2025], sem reexecução no mesmo *pipeline* experimental. Embora a metodologia aplicada ao Gemini 2.0 Flash replique os mesmos *scripts*, *prompt* e normalização do estudo de referência, a versão exata do GPT-4 utilizada por Chen et al. [2025] não é rastreável, uma vez que atualizações silenciosas da API impedem identificar o *snapshot* do modelo na data de execução original. Adicionalmente, o Gemini 2.0 Flash foi executado uma única vez por contrato. Os dados de referência do GPT-4 correspondem a uma única saída disponibilizada no repositório oficial de Chen et al. [2025], e não à média de múltiplas iterações relatada pelos autores em seu trabalho.

Em relação à **validade de construção**, a análise manual foi conduzida por um único avaliador, o que impede o cálculo formal de concordância entre avaliadores e pode introduzir viés. Para mitigar esse risco, adotou-se um protocolo de anotação com critérios explícitos: uma detecção só é contada como verdadeiro positivo se o modelo identificou corretamente a função e a instrução apontadas pelo gabarito como vulneráveis (Seção IV.2.5), com verificação de consistência via múltiplos LLMs e revisão manual das divergências. A avaliação binária pode superestimar o desempenho ao desconsiderar casos parciais, limitação que a análise manual por ocorrência busca mitigar. O *bootstrap* foi executado de forma pareada: a cada iteração, os dois modelos recebem exatamente o mesmo conjunto de arquivos sorteados com reposição, e o denominador da cobertura é recalculado a partir do gabarito desses arquivos, mantendo consistência entre numerador e denominador. Esse pareamento garante que a única fonte de variação entre os modelos seja o desempenho em cada contrato, tornando o teste de significância mais sensível.

Em relação à **validade externa**, o SmartBugs Curated é composto majoritariamente por contratos de 2016–2019, período anterior à consolidação de padrões como DeFi, protocolos de liquidez e contratos de governança. As categorias do DASP Top 10, embora amplamente adotadas na literatura, podem não representar adequadamente o perfil de vulnerabilidades do ecossistema Solidity moderno. Dessa forma, os resultados obtidos devem ser interpretados no contexto desse *benchmark* específico, sendo a generalização para contratos contemporâneos uma limitação reconhecida do estudo.

Capítulo VI Conclusões

Este trabalho investigou dois objetivos complementares: avaliar a evolução da precisão e das limitações das ferramentas de análise estática Mythril e Slither sobre o conjunto de dados SmartBugs Curated, e avaliar o desempenho do Gemini 2.0 Flash na detecção de vulnerabilidades em contratos inteligentes, comparando-o ao GPT-4 e investigando os padrões de erro associados a falsos positivos e falsos negativos. Esses dois experimentos permitem traçar um panorama sobre o estado atual da detecção automatizada de vulnerabilidades, contemplando tanto as ferramentas tradicionais de análise estática quanto as abordagens emergentes baseadas em LLMs. A seguir, são sintetizados os principais resultados obtidos em cada experimento, à luz das questões de pesquisa que orientaram este estudo.

No primeiro experimento, a análise conduzida com Mythril (versões 0.21.15 e 0.24.7) e Slither (versões 0.6.6 e 0.10.0) sobre os 69 contratos Solidity do SmartBugs Curated buscou responder à questão RQ0, se houve evolução na precisão e cobertura de detecção entre as duas versões avaliadas. Os resultados confirmam padrões consistentes de evolução, mas também limitações persistentes. No Mythril, a versão 0.24.7 apresentou redução expressiva de falsos positivos em *Arithmetic*, de 92 para 21 detecções, com manutenção da cobertura em 0,64, resultando no maior ganho de *F1-score* entre as versões (0,26 $\rightarrow$ 0,65). A ferramenta também passou a detectar *Time Manipulation* e *Bad Randomness*, categorias ausentes na versão anterior. Em contrapartida, houve regressão em *Front Running* (F1: 0,14 $\rightarrow$ 0,07) e *Unchecked Low Calls*, confirmando o dilema precisão-cobertura típico dessas ferramentas [Chess and West, 2007]. No Slither, a versão 0.10.0 destacou-se pela precisão máxima em *Access Control* (1,00), eliminando completamente os falsos positivos nessa categoria, e pela cobertura total em *Reentrancy* (1,00), com *F1-score* de 0,67, o melhor resultado absoluto entre todas as combinações ferramenta-categoria avaliadas. Contudo, *Time Manipulation* apresentou queda no *F1-score* (0,20 $\rightarrow$ 0,07). De forma geral, os resultados confirmam que as atualizações trouxeram ganhos reais, sobretudo na redução de falsos positivos, mas nenhuma versão avaliada oferece simultaneamente alta precisão e alta cobertura em todas as categorias. A análise manual mostrou-se indispensável para corrigir erros de classificação e localização que escapam à contagem automatizada, reforçando seu papel complementar mesmo em pipelines automatizados.

No segundo experimento, as análises foram orientadas por três questões de pesquisa. Em relação à RQ1, qual é a diferença de desempenho entre Gemini 2.0 Flash e GPT-4 na detecção de

vulnerabilidades em contratos inteligentes, os resultados indicam que os dois modelos são muito próximos em termos qualitativos: ambos os modelos apresentam alta cobertura e baixa precisão na maioria das categorias, com padrões de erro estruturalmente semelhantes. O Gemini apresentou vantagem pontual em *Access Control* (F1: 23,8% vs. 18,5% na análise manual), enquanto o GPT-4 liderou nas demais categorias, com destaque para *Bad Randomness* (F1: 78,1% vs. 49,1%). A análise de estabilidade via *bootstrap* não-paramétrico pareado com 10.000 reamostras revelou que quatro categorias apresentaram diferença estatisticamente significativa ($p < 0{,}05$) a favor do GPT-4: *Reentrancy* ($p < 0{,}001$; 47,6 [35,7; 59,6] vs. 32,1 [23,0; 41,6] do Gemini), *Arithmetic* ($p = 0{,}007$; 35,4 [21,1; 47,4] vs. 27,7 [15,4; 39,1]), *Denial of Service* ($p = 0{,}010$; 10,1 [2,2; 18,6] vs. 4,9 [1,0; 9,3]) e *Time Manipulation* ($p = 0{,}012$; 34,3 [6,9; 56,4] vs. 21,1 [3,8; 38,7]). Nas demais categorias, os IC95% se sobrepõem e as diferenças não são estatisticamente distinguíveis. Em *Bad Randomness*, embora a diferença pontual seja a mais expressiva entre todas as categorias (GPT-4: 78,1 [47,6; 90,9] vs. Gemini: 49,1 [18,9; 80,0]), o teste não rejeita a hipótese nula ($p = 0{,}111$), reflexo da concentração de 12 ocorrências em um único contrato (*smart_billions.sol*): quando esse arquivo é ou não sorteado nas reamostras, o resultado agregado varia substancialmente. Em *Access Control*, o Gemini apresenta F1 pontualmente superior (23,8 vs. 18,5), sendo esta a única categoria em que a vantagem pontual favorece o Gemini, embora sem significância estatística ($p = 0{,}949$).

Em relação à RQ2, qual é o impacto na avaliação de desempenho ao considerar a presença da vulnerabilidade no arquivo em vez de sua localização exata na linha de código, a resposta evidencia um impacto significativo. A transição da análise binária para a manual revelou quedas expressivas de cobertura, especialmente no Gemini para *Bad Randomness* ($100{,}0\% \rightarrow 46{,}7\%$, $\Delta = 53{,}3\%$) e *Front Running* ($100{,}0\% \rightarrow 42{,}9\%$, $\Delta = 57{,}1\%$). A avaliação binária ignora o número de ocorrências por arquivo: em contratos extensos com múltiplas instâncias da mesma vulnerabilidade, o modelo pode gerar ao menos um alerta positivo sem cobrir todas as ocorrências reais, produzindo uma estimativa de cobertura artificialmente elevada.

Em relação à RQ3, se os falsos positivos e falsos negativos produzidos pelos modelos podem ser explicados por padrões recorrentes de erro, a auditoria manual dos 139 relatórios por modelo revelou uma limitação estrutural comum a ambos: a dependência de indícios sintáticos em detrimento da análise semântica do fluxo de execução. Os falsos positivos decorrem majoritariamente da associação automática entre estruturas de código e vulnerabilidades conhecidas, como o uso de *call.value()* como indicativo de *Reentrancy*, a presença de *block.timestamp* como indicativo de *Time Manipulation*, ou loops internos como indicativo de *Denial of Service*, sem que os modelos avaliem se as condições necessárias para exploração estão de fato presentes. Os falsos negativos associam-se à dificuldade em raciocinar sobre o fluxo de estado do contrato e a lógica de negócio da EVM, como observado no caso do *underflow* em *arbitrary_location_write_simple.sol*, onde o modelo identificou a ausência de

controle de acesso mas não o mecanismo real da vulnerabilidade. Observou-se ainda que o Gemini tende a respostas mais detalhadas e explicativas, enquanto o GPT-4 produz relatórios mais diretos e concisos, diferença que não se traduz em vantagem consistente de desempenho para nenhum dos modelos.

Embora os dois experimentos utilizem conjuntos de dados de tamanhos distintos (69 e 142 contratos) e não permitam uma comparação direta ferramenta-vs-LLM estatisticamente válida sobre o mesmo conjunto, a leitura qualitativa apresentada na Seção V.3 indica um padrão de complementaridade: as ferramentas estáticas tendem a superar os LLMs em categorias com assinatura sintática bem delimitada (*Reentrancy*, *Access Control*), enquanto os LLMs se destacam em categorias com padrões de código mais uniformes ou que dependem de interpretação textual do contexto (*Unchecked Low-Level Calls*, *Time Manipulation*). Essa complementaridade reforça que abordagens híbridas, combinando a precisão das ferramentas tradicionais com a cobertura dos LLMs, representam uma direção promissora, embora sua validação empírica direta sobre um conjunto comum de contratos constitua uma limitação reconhecida deste trabalho e uma agenda natural para estudos futuros.

As principais contribuições deste trabalho são: a avaliação longitudinal e comparativa da evolução do Mythril e do Slither com análise manual que corrige limitações das contagens automatizadas; a demonstração de que o Gemini 2.0 Flash, modelo com nível de acesso gratuito, apresenta capacidade relevante para triagem inicial de vulnerabilidades com perfil de comportamento semelhante ao GPT-4 em um *benchmark* consolidado, com metodologia replicável e artefatos públicos; e a caracterização qualitativa sistemática dos padrões de erro por categoria, identificando a dependência de indícios sintáticos como limitação estrutural compartilhada pelos dois modelos, contribuição que orienta diretamente estratégias de melhoria como *prompt engineering* especializado e integração com ferramentas de análise estática.

Como trabalhos futuros, sugere-se replicar os experimentos com conjuntos de dados mais recentes e balanceados por categoria, de modo a aumentar o poder estatístico e reduzir a instabilidade das métricas em categorias com poucos contratos; investigar o impacto do tamanho do contrato e da multiplicidade de ocorrências no desempenho dos LLMs; avaliar o uso de *prompts* especializados por categoria de vulnerabilidade para mitigar a tendência de análise puramente sintática; e explorar abordagens híbridas que combinem ferramentas de análise estática com LLMs, aproveitando a complementaridade entre a precisão das ferramentas tradicionais e a acessibilidade dos modelos de linguagem.

Em síntese, os dois experimentos reforçam que nenhuma abordagem isolada, ferramentas tradicionais ou LLMs, oferece cobertura suficiente para auditoria completa de contratos inteligentes. A combinação de múltiplas abordagens, com validação humana, representa o caminho mais promissor para aumentar a confiabilidade dos contratos antes de sua publicação na *blockchain*.

Referências

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

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

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

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.

Boi, B., Esposito, C., and Lee, S. Vulnhunt-gpt: a smart contract vulnerabilities detector based on openai chatgpt. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, pages 1517–1524, 2024.

Buterin, V. et al. Ethereum white paper. *GitHub repository*, 1(22-23):5–7, 2013.

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

Chen, C., Su, J., Chen, J., Wang, Y., Bi, T., Yu, J., Wang, Y., Lin, X., Chen, T., and Zheng, Z. When chatgpt meets smart contract vulnerability detection: How far are we? *ACM Transactions on Software Engineering and Methodology*, 34(4):1–30, 2025.

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

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

David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., and Gervais, A. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*, 2023.

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

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

Dia, B., Ivaki, N., and Laranjeiro, N. An empirical evaluation of the effectiveness of smart contract verification tools. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 17–26. IEEE, 2021.

Dror, R., Baumer, G., Shlomov, S., and Reichart, R. The hitchhiker's guide to testing statistical significance in NLP. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pages 1383–1392, 2018.

Durieux, T., Ferreira, J. F., Abreu, R., and Cruz, P. 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, 2020.

Feist, J., Grieco, G., and Groce, A. 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, 2019.

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

Fonseca, F. M., dos Santos Moura, M., Gonzalez, P. H., and Mendonça, D. S. Towards the evolution of tools for detecting vulnerabilities in smart contracts: A case study of mythril and slither. In *Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain)*, pages 43–56. SBC, 2025.

Fonseca, F. M., Gonzalez, P. H., and Mendonça, D. S. Detecção de vulnerabilidades em smart contracts com llms: Uma comparação entre gemini 2.0 flash e gpt-4. In *Anais do IX Workshop em Blockchain: Teoria, Tecnologias e Aplicações (WBlockchain)*, pages 43–56, Praia do Forte, BA, Brasil. SBC, 2026.

Google Cloud. Gemini 2.0 flash documentation. `https://cloud.google.com/vertex-ai/docs/generative-ai/models/gemini/2-0-flash`, 2026.

Google DeepMind. Introducing gemini 2.0: our new ai model for the agentic era. `https://blog.google/innovation-and-ai/models-and-research/google-deepmind/google-gemini-ai-update-december-2024/`, 2024.

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

Hall, P. and Wilson, S. R. Two guidelines for bootstrap hypothesis testing. *Biometrics*, 47(2):757–762, 1991.

Hu, S., Huang, T., İlhan, F., Tekin, S. F., and Liu, L. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 297–306. IEEE, 2023a.

Hu, T., Li, B., Pan, Z., and Qian, C. Detect defects of solidity smart contract based on the knowledge graph. *IEEE Transactions on Reliability*, 2023b.

Ibba, G., Aufiero, S., Neykova, R., Bartolucci, S., Ortu, M., Tonelli, R., and Destefanis, G. 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, 2024.

Kitchenham, B. A. and Charters, S. M. Guidelines for performing systematic literature reviews in software engineering. Technical report, Citeseer, 2007.

Koehn, P. Statistical significance tests for machine translation evaluation. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 388–395, 2004.

Liao, J.-W., Tsai, T.-T., He, C.-K., and Tien, C.-W. 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, 2019.

Ma, W., Wu, D., Sun, Y., Wang, T., Liu, S., Zhang, J., Xue, Y., and Liu, Y. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. *arXiv preprint arXiv:2403.16073*, 2024.

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

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., and Gao, J. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.

Mishra, D. and Phansalkar, S. Blockchain security in focus: A comprehensive investigation into threats, smart contract security, cross-chain bridges, vulnerabilities detection tools & techniques. *IEEE Access*, 2025.

Mourão, E., Pimentel, J. F., Murta, L., Kalinowski, M., Mendes, E., and Wohlin, C. On the performance of hybrid search strategies for systematic literature reviews in software engineering. *Information and Software Technology*, 123:106294, 2020.

Munir, S. and Reichenbach, C. 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, 2023.

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

Neto, A., Perkusich, M., Dantas, E., Ramos, F., Costa, A., Almeida, H., and Perkusich, A. Knowledge-based risk management: A systematic literature review. In *Brazilian Symposium on Software Engineering*, pages 320–329, 2021.

Nofer, M., Gomber, P., Hinz, O., and Schiereck, D. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

OpenAI. Gpt-4 technical overview, 2023.

OpenZeppelin. Understanding the Balancer V2 Exploit. OpenZeppelin Blog, 2025.

Parizi, R. M., Dehghantanha, A., Choo, K.-K. R., and Singh, A. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. *arXiv preprint arXiv:1809.02702*, 2018.

Powers, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

Prechtel, D., Groß, T., and Müller, T. 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, 2019.

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

Salzano, F., Marchesi, L., Antenucci, C. K., Scalabrino, S., Tonelli, R., Oliveto, R., and Pareschi, R. Bridging the gap: a comparative study of academic and developer approaches to smart contract vulnerabilities. *Empirical Software Engineering*, 31(2):37, 2026.

Shen, Z., Chen, Y., and Zhang, W. Gsvd: Common vulnerability dataset for smart contracts on bsc and polygon. In *CS & IT Conference Proceedings*, volume 13. CS & IT Conference Proceedings, 2023.

Sherlock and Panprog. GMX Exchange Hack Explained: $42M Drained via Re-Entrancy Exploit. Sherlock Security Blog, 2026.

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

Smaragdakis, Y., Grech, N., Lagouvardos, S., Triantafyllou, K., and Tsatiris, I. Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021.

So, S., Hong, S., and Oh, H. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1361–1378, 2021.

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., and Liu, Y. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

Tibshirani, R. J. and Efron, B. An introduction to the bootstrap. *Monographs on statistics and applied probability*, 57(1):1–436, 1993.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, 2018.

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

Vidal, F. R., Ivaki, N., and Laranjeiro, N. Openscv: An open hierarchical taxonomy for smart contract vulnerabilities. *Empirical Software Engineering*, 29(4):101, 2024.

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

Ye, J., Ma, M., Peng, T., Peng, Y., and Xue, Y. 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, 2019.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Zheng, P., Zheng, Z., and Luo, X. Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 740–751, 2022.

Zheng, Z., Su, J., Chen, J., Lo, D., Zhong, Z., and Ye, M. Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects. *IEEE Transactions on Software Engineering*, 50(6):1360–1373, 2024.

Zheng, Z., Xie, S., Dai, H., Chen, X., and Wang, H. An overview of blockchain technology: Architecture, consensus, and future trends. pages 557–564, 2017.

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

Apêndice A Análise do Slither — Ether Lotto (Time Manipulation)

Neste arquivo o Slither aponta uma vulnerabilidade existente, mas não acerta especificamente o local. Na análise do Slither, o arquivo ether_lotto.sol é considerado validado pelos especialistas, apontando a vulnerabilidade na linha 43, mostrada no código a seguir:

```
// Compute some almost random value for selecting winner from current
//transaction.
// <yes> <report> TIME_MANIPULATION
var random = uint(sha3(block.timestamp)) % 2;
```

Na análise do Slither 0.10.0, essa linha possui algumas vulnerabilidades, mas uma delas é classificada no json como "check": "weak-prng". Nenhuma delas, nessa posição, está relacionada ao *Time Manipulation*, ou seja, em uma contagem direta, o arquivo poderia ser classificado como contendo *Time Manipulation*, mas essa não seria a vulnerabilidade que realmente foi apontada pelos especialistas. A seguir está o trecho da saída que aponta para a L43, contudo trata de outra vulnerabilidade:

```
"description": "EtherLotto.play() (repositories/smartbugs-curated/ether_lotto.sol
    #33-57) uses a weak PRNG: "random = uint256(sha3()(block.timestamp)) % 2 (
    repositories/smartbugs-curated/ether_lotto.sol#43)" \n",
"markdown": "EtherLotto.play()
 uses a weak PRNG: "random = uint256(sha3()(block.timestamp)) % 2
" \n",
"first_markdown_element": "repositories/smartbugs-curated/ether_lotto.sol#L33-L57"
    ,
"id": "bb6ca24954817c7df24ff2bbafc950534e6cede226c1e6cb0bbd604547058af3",
"check": "weak-prng",
"impact": "High",
"confidence": "Medium"
```

Para a vulnerabilidade de *Time Manipulation*, que é a que está validada para este Solidity em questão, a análise aponta para outra linha que não foi a levantada:

```
"description": "EtherLotto.play() (repositories/SmartBugs-curated/ether_lotto.sol
    #33-57) uses
timestamp for comparisons\n\tDangerous comparisons:\n\t- random == 0
(repositories/SmartBugs-curated/ether_lotto.sol#46)\n",
```

```
4
5 if (random == 0)
```

O Slither erra o local da vulnerabilidade, pois, apesar de a vulnerabilidade estar nos valores contidos em random, ela ocorre devido ao uso do timestamp. O problema não está na verificação do if, como o Slither apontou, mas sim no uso do sorteio com o timestamp, que é o que inicia a vulnerabilidade.

A versão anterior do Slither, na análise do trabalho do SmartBugs, encontrou o mesmo erro no mesmo lugar, no if.

```
1 "check": "timestamp",
2 "impact": "Low",
3 "confidence": "Medium",
4 "description": "EtherLotto.play (/dataset/time_manipulation/ether_lotto.sol#33-57)
      uses
5 timestamp for comparisons\n\tDangerous comparisons:\n\t- random == 0
6 (/dataset/time_manipulation/ether_lotto.sol#46-56)\n",
```

Apêndice B Contratos com Vulnerabilidade em Time Manipulation

Dentre os contratos inteligentes validados que apresentam vulnerabilidade em *Time Manipulation* são: ether_lotto.sol, governmental_survey.sol, roulette.sol e timed_crowdsale.sol. Dentro do código do ether_lotto.sol , foi encontrada a vulnerabilidade apontada na linha 43.

```
1     // <yes> <report> TIME_MANIPULATION
2         var random = uint(sha3(block.timestamp)) % 2;
```

Buscando a vulnerabilidade na análise do trabalho do SmartBugs na ferramenta Slither, encontra-se a descrição a seguir:

```
1 "description": "EtherLotto.play (/dataset/time_manipulation/ether_lotto.sol#33-57)
     uses timestamp for comparisons\n\tDangerous comparisons:\n\t- random == 0 (/
    dataset/time_manipulation/ether_lotto.sol#46-56)\n",
```

Este trecho mostra a saída da única vulnerabilidade relacionada com o *Time Manipulation*. A vulnerabilidade foi detectada pela ferramenta no bloco condicional como podemos ver a seguir. Assim podemos notar que não foi exatamente onde o código foi validado.

```
1  // Distribution: 50% of participants will be winners.
2          if (random == 0) {
3
4              // Send fee to bank account.
5              bank.transfer(FEE_AMOUNT);
6
7              // Send jackpot to winner.
8              msg.sender.transfer(pot - FEE_AMOUNT);
9
10             // Restart jackpot.
11             pot = 0;
12         }
```

O arquivo governmental_survey.sol possui a vulnerabilidade apontada para linha 27, como:

```
1    // <yes> <report> TIME_MANIPULATION
2     lastInvestmentTimestamp = block.timestamp;
```

A análise feita pelo SmartBugs usando o Slither tem em seu arquivo de saída da análise apontada para esta descrição:

```
"description": "Governmental.resetInvestment (/dataset/time_manipulation/
    governmental_survey.sol#30-40) uses timestamp for comparisons\n\tDangerous
    comparisons:\n\t- block.timestamp < lastInvestmentTimestamp + ONE_MINUTE (/
    dataset/time_manipulation/governmental_survey.sol#31-32)\n",
```

Assim a informação apontada não é o local onde deveria estar a vulnerabilidade. Neste caso apontada para a seguinte parte.

```
if (block.timestamp < lastInvestmentTimestamp+ONE_MINUTE)
    throw;
```

O arquivo roulette.sol possui a vulnerabilidade nas seguintes linhas do seu código:

```
// <yes> <report> TIME_MANIPULATION
        require(now != pastBlockTime); // only 1 transaction per block
        // <yes> <report> TIME_MANIPULATION
        pastBlockTime = now;
```

Contudo na análise do Slither no trabalho do SmartBugs não foi em nenhum ponto detectado a vulnerabilidade **Time Manipulation**.

No código de timed_crowdsale.sol, temos a vulnerabilidade sendo apontada em

```
// <yes> <report> TIME_MANIPULATION
  return block.timestamp >= 1546300800;
```

O Slither consegue encontrar a vulnerabilidade no local correto, como apontado na descrição

```
    "description": "TimedCrowdsale.isSaleFinished (/dataset/time_manipulation/
        timed_crowdsale.sol#11-14) uses timestamp for comparisons\n\tDangerous
        comparisons:\n\t- block.timestamp >= 1546300800 (/dataset/
        time_manipulation/timed_crowdsale.sol#13)\n",
```

Na contagem de vulnerabilidades para *Time Manipulation* foi colocado que foram encontradas duas vulnerabilidades pela análise, porém por esta contagem manual demonstrada foi somente uma.

Apêndice C Análise do FibonacciBalance.sol

O arquivo FibonacciBalance.sol, possui duas vulnerabilidades.

```
// calculate the fibonacci number for the current withdrawal user
        // this sets calculatedFibNumber
        // <yes> <report> ACCESS_CONTROL
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
```

```
  // allow users to call fibonacci library functions
    function() public {
        // <yes> <report> ACCESS_CONTROL
        require(fibonacciLibrary.delegatecall(msg.data));
    }
```

Na saída da análise do artigo do SmartBugs, temos:

```

  {
        "address": 212,
        "code": "fibonacciLibrary.delegatecall(msg.data)",
        "debug": "",
        "description": "Be aware that the called contract gets unrestricted access
            to this contract's state.",
        "filename": "/dataset/unchecked_low_level_calls/FibonacciBalance.sol",
        "function": "fallback",
        "lineno": 38,
        "title": "DELEGATECALL to a user-supplied address",
        "type": "Informational"
      },
```

O trecho aponta a vulnerabilidade dentro do require(fibonacciLibrary.delegatecall(msg.data)). A mesma vulnerabilidade já estava presente no arquivo, e como a contagem aumentou exatamente em um, acredita-se que essa mudança tenha sido a causa da variação observada. Esse ponto é reforçado pela explicação no início da seção de resultados, onde se menciona que, nas detecções automatizadas, houve uma movimentação do arquivo FibonacciBalance.sol da categoria *Unchecked Low-Level Calls*. Além disso, reforça-se que essa alteração ocorreu em 23 de novembro de 2022,

após a realização do estudo do SmartBugs.

Apêndice D Correção de Contagem em Denial of Service

Durante a revisão deste trabalho, identificou-se um erro de contagem no gabarito de *Denial of Service* utilizado na Tabela V.4: o valor de 14, também presente na publicação derivada deste trabalho [Fonseca et al., 2025], foi corrigido para 7, em conformidade com o gabarito original de [Durieux et al., 2020] para essa categoria.

Inicialmente, a contagem foi realizada com base no número de linhas apontadas como vulneráveis no campo `@vulnerable_at_lines` dos contratos anotados. Como, em diversos casos, uma única ocorrência de *Denial of Service* se estende por múltiplas linhas de código, essa abordagem resultou na contagem de 14 linhas vulneráveis, em vez das 7 ocorrências de vulnerabilidade de fato existentes nos seis contratos do conjunto de dados de 69 contratos anotados com essa categoria. Essa dificuldade foi identificada posteriormente e corrigida nesta versão do trabalho.

Cabe ressaltar que esse erro não impacta nenhum resultado quantitativo reportado neste trabalho: a categoria *Denial of Service* não é contabilizada nas métricas de precisão, cobertura e *F1-score* do Primeiro Experimento (Tabelas V.7, V.8 e V.9), uma vez que nenhuma das ferramentas avaliadas obteve detecções verdadeiras positivas para essa categoria, tornando tais métricas matematicamente indefinidas. A Tabela V.4 tem caráter exclusivamente descritivo e comparativo entre versões das ferramentas, não alimentando nenhum cálculo posterior.

TERMO DE AUTORIZAÇÃO PARA PUBLICAÇÃO/DIVULGAÇÃO DE DOCUMENTO ELETRÔNICO NO CATÁLOGO DO SISTEMA DE BIBLIOTECAS DO CEFET/RJ E NO REPOSITÓRIO INSTITUCIONAL

Na qualidade de titular dos direitos de autor da publicação abaixo citada, de acordo com a Lei nº 9610/98, autorizo, a partir da presente data, o CEFET/RJ, a disponibilizar gratuitamente, sem ressarcimento de direitos autorais, conforme permissões assinadas abaixo, no Catálogo Online do Sistema de Bibliotecas e no Repositório Institucional do Cefet/RJ, ou qualquer outro formato de disseminação da informação que a instituição implemente, no formato PDF, para fins de leitura, impressão e/ou download pela internet, a título de divulgação da produção científica gerada por esta Instituição.
Declaro, ainda, que a versão definitiva do trabalho a ser disponibilizada é de inteira responsabilidade do(a) autor(a) e do(a) orientador(a), que atestam sua conformidade com a versão aprovada e autorizam sua divulgação nos meios institucionais do CEFET/RJ, eximindo a Instituição de qualquer responsabilidade quanto ao conteúdo, à integridade ou a eventuais alterações posteriores ao depósito.

1- Tipo de trabalho: [ ] Projeto Final (Graduação) [ ] Especialização
[X] Dissertação [ ] Tese

2- Identificação do trabalho/autor(es):
Curso: Mestrado em Ciência da Computação
Título: DETECÇAO DE VULNERABILIDADES EM CONTRATOS INTELIGENTES: UM ESTUDO COM FERRAMENTAS AUTOMATIZADAS
Autor(es): Felipe Mello Fonseca
Orientador: Diogo Silveira Mendonça
Co-orientador: Pedro Henrique Gonzalez

3- Informações de acesso ao trabalho:

Este trabalho é confidencial? [ ] Sim [X] Não
Pode ser liberado para publicação? [X] Total [ ] Parcial [ ] Não
Justifique: _____

Em caso de publicação parcial, assinale as permissões:
[ ] Sumário
[ ] Capítulos. Especifique: _____
[ ] Bibliografia
[ ] Outras partes do trabalho: _____

Rio de Janeiro, 23 / 09 / 2026

gov.br Documento assinado digitalmente
FELIPE MELLO FONSECA
Data: 24/09/2026 09:16:45-0300
Verifique em https://validar.iti.gov.br

_____
Assinatura do(s) autor(es)

gov.br Documento assinado digitalmente
DIOGO SILVEIRA MENDONCA
Data: 25/09/2026 11:57:15-0300
Verifique em https://validar.iti.gov.br

_____
Assinatura do orientador

gov.br Documento assinado digitalmente
PEDRO HENRIQUE GONZALEZ SILVA
Data: 25/09/2026 22:42:10-0300
Verifique em https://validar.iti.gov.br

_____
Assinatura do co-orientador