Detecção de Vulnerabilidades em Smart Contracts com LLMs: Uma Comparação entre Gemini 2.0 Flash e GPT-4

Felipe Mello Fonseca[1], **Pedro Henrique Gonzalez**[2], **Diogo Silveira Mendonça**[1]

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

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

{felipe.mello}@aluno.cefet-rj.br

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

Abstract. *Automated vulnerability detection in smart contracts is a significant challenge for the security of decentralized systems. This work investigates the effectiveness of Large Language Models (LLMs), especially Gemini 2.0 Flash, in this task. The methodology is based on experimental replication using the SmartBugs Curated dataset, allowing a comparative analysis with GPT-4. The results indicate that, although LLMs perform well in initial screening, they still face limitations in analyses that require deeper semantic reasoning of the execution logic. It is concluded that these models can support the auditing process, but validation by human experts remains essential. Future directions are also pointed out, such as the refinement of prompts and specialized analyses by vulnerability category.*

Resumo. *A detecção automatizada de vulnerabilidades em smart contracts é um desafio relevante para a segurança de sistemas descentralizados. Este trabalho investiga a eficácia da Large Language Models (LLMs), em especial o Gemini 2.0 Flash, nessa tarefa. A metodologia baseia-se na replicação experimental utilizando o dataset SmartBugs Curated, permitindo uma análise comparativa com o GPT-4. Os resultados indicam que, embora as LLMs apresentem bom desempenho na triagem inicial, ainda enfrentam limitações em análises que exigem raciocínio semântico mais profundo da lógica de execução. Conclui-se que esses modelos podem apoiar o processo de auditoria, mas a validação por especialistas humanos permanece essencial. Também são apontadas direções futuras, como o refinamento de prompts e análises especializadas por categoria de vulnerabilidade.*

1. Introdução

Uma *blockchain* é um livro-razão distribuído que permite registrar transações de forma descentralizada e imutável [Nakamoto 2008, Maesa and Mori 2020]. Entre suas

aplicações mais relevantes estão os *smart contracts*, programas executados em plataformas como o Ethereum, frequentemente escritos em Solidity, que automatizam regras de transferência de ativos digitais [Buterin et al. 2014].

Devido ao grande volume financeiro envolvido, vulnerabilidades em *smart contracts* podem resultar em perdas significativas. Um exemplo conhecido é o ataque ao *Decentralized Autonomous Organization* (DAO) em 2016, que causou prejuízos de aproximadamente 60 milhões de dólares [Liao et al. 2019]. Dessa forma, a detecção automatizada de vulnerabilidades tornou-se um desafio importante na segurança de sistemas baseados em *blockchain*.

Nesse contexto, modelos de linguagem de grande escala (LLMs) têm sido investigados como ferramentas auxiliares na análise de código. Assim, este trabalho avalia o uso do *Gemini 2.0 Flash* na detecção de vulnerabilidades em *smart contracts*, comparando seu desempenho com o *GPT-4* a partir da replicação do *pipeline* experimental de [Chen et al. 2025].

O objetivo principal é avaliar o desempenho do *Gemini 2.0 Flash* na detecção de vulnerabilidades em *smart contracts* e compará-lo ao *GPT-4*, investigando se um modelo acessível reproduz o perfil de desempenho de um modelo pago. Durante a análise, identificou-se ainda que a metodologia de avaliação binária adotada no estudo base pode superestimar o desempenho dos modelos, motivando uma análise complementar por ocorrência e uma caracterização qualitativa dos padrões de erro por categoria.

Para isso, este trabalho busca responder às seguintes questões de pesquisa:

- **RQ1:** O *Gemini 2.0 Flash* reproduz o perfil de desempenho observado para o *GPT-4* na detecção de vulnerabilidades em *smart contracts*?
- **RQ2:** A avaliação binária superestima o desempenho dos modelos em relação a uma avaliação por ocorrência?
- **RQ3:** Quais padrões de raciocínio estão associados a falsos positivos e falsos negativos nas diferentes categorias de vulnerabilidade?

O artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica; a Seção 3 discute trabalhos relacionados; a Seção 4 descreve a metodologia; a Seção 5 apresenta os resultados; a Seção 6 traz os apontamentos da análise manual; a Seção 7 discute as ameaças à validade; e a Seção 8 apresenta as conclusões.

2. Fundamentação Teórica

2.1. Infraestrutura Blockchain e Ethereum

O *blockchain* é um livro-razão público e distribuído que registra transações em blocos encadeados por funções de *hash*, garantindo imutabilidade, descentralização e transparência [Nakamoto 2008, Zheng et al. 2017]. O Ethereum estende esse modelo com uma linguagem Turing-completa que permite a implementação de *smart contracts* escritos em Solidity [Buterin et al. 2014]. O código é compilado em *bytecode* executado pela *Ethereum Virtual Machine* (EVM), e as operações têm um custo computacional denominado *gas*, pago antecipadamente pelo usuário [Tikhomirov et al. 2018].

2.2. Decentralized Application Security Project (DASP)

O DASP organiza as vulnerabilidades de *smart contracts* em dez categorias principais (DASP Top 10) [Larios-Vargas et al. 2023, Salzano et al. 2026], adotadas neste trabalho como base de classificação: *Reentrancy* (chamadas recursivas antes da atualização de estado); *Access Control* (ausência de restrições em funções sensíveis); *Arithmetic* (erros de *overflow*/*underflow*); *Unchecked Low-Level Calls* (chamadas sem verificação de retorno); *Denial of Service* (consumo excessivo de *gas*); *Bad Randomness* (uso de fontes previsíveis); *Front Running* (exploração da ordem de transações no *mempool*); *Time Manipulation* (dependência de *block.timestamp*); *Short Address* (argumentos malformados em chamadas); e *Unknowns* (vulnerabilidades não formalmente classificadas).

2.3. LLMs

Os *Large Language Models* (LLMs) são modelos baseados na arquitetura *Transformer* com bilhões de parâmetros, treinados em grandes volumes de texto [Zhao et al. 2023, Minaee et al. 2024]. Técnicas como ajuste fino e alinhamento via *Reinforcement Learning from Human Feedback* (RLHF) ampliam sua aplicabilidade em tarefas como geração de código e análise semântica, embora ainda apresentem limitações relacionadas a viés, custo computacional e geração de informações incorretas (*hallucination*) [Minaee et al. 2024].

2.4. Modelos Avaliados

O *GPT-4* é um LLM da *OpenAI* com melhorias em raciocínio e interpretação de instruções obtidas via RLHF [OpenAI 2023, Achiam et al. 2023]. O *Gemini 2.0 Flash* é um modelo multimodal da *Google* com suporte a texto, áudio, imagem e vídeo, janela de contexto de até 1 milhão de *tokens* e integração com ferramentas externas [Google Cloud 2026, Google DeepMind 2024]. Ambos podem gerar informações incorretas (*hallucination*), tornando a validação humana necessária em aplicações críticas [Achiam et al. 2023].

3. Trabalhos Relacionados

Os trabalhos relacionados foram identificados por meio de um *script* de *scraping* em Python para coleta de publicações no Google Scholar. A seleção considerou relevância e aderência temática. Para o experimento envolvendo LLMs, foi utilizada a *string* de busca: (”Large Language Models”OR ”LLM”OR ChatGPT OR GPT-4 OR Gemini OR Grok) AND (”smart contract”OR Solidity) AND (vulnerability OR security OR auditing OR ”bug detection”).

O estudo de [Chen et al. 2025] avalia o uso de LLMs na detecção de vulnerabilidades em *smart contracts* utilizando o *dataset SmartBugs Curated*. Os resultados indicam alto *recall*, porém baixa precisão, evidenciando elevado número de falsos positivos quando comparado a ferramentas especializadas.

O trabalho de [Sun et al. 2024] apresenta o GPTScan, que combina LLMs com análise estática para detectar vulnerabilidades lógicas. A abordagem utiliza o modelo para análise semântica e valida os resultados com análise estática, alcançando alta precisão em diferentes *datasets*.

O framework GPTLENS, proposto por [Hu et al. 2023], utiliza um processo em dois estágios com LLMs para geração e avaliação de hipóteses de vulnerabilidades.

Avaliações mostraram aumento significativo na taxa de acertos em contratos com falhas conhecidas.

Já [Boi et al. 2024] apresenta o VulnHunt-GPT, que utiliza *prompt engineering* e enriquecimento de contexto para detectar vulnerabilidades do DASP Top 10, demonstrando boa cobertura e tempo de execução reduzido.

O estudo de [David et al. 2023] compara diferentes LLMs com ferramentas tradicionais, mostrando que modelos como *GPT-4* apresentam bom desempenho em vulnerabilidades complexas, enquanto ferramentas estáticas mantêm maior precisão em falhas simples.

Por fim, [Ma et al. 2024] propõe o *iAudit*, um *framework* que combina *fine-tuning* de LLMs e agentes para detecção e explicação de vulnerabilidades lógicas, alcançando resultados superiores em comparação a diferentes abordagens *zero-shot* e modelos ajustados.

Os trabalhos apresentados avaliam LLMs na detecção de vulnerabilidades em *smart contracts* sob diferentes perspectivas: aplicação direta de modelos como *GPT-4* [Chen et al. 2025, David et al. 2023], combinação com análise estática [Sun et al. 2024], arquiteturas multi-agente [Hu et al. 2023, Ma et al. 2024] e enriquecimento de contexto via *prompt engineering* [Boi et al. 2024]. No entanto, duas lacunas permanecem abertas. Primeiro, nenhum estudo investiga se um modelo de acesso gratuito, como o *Gemini 2.0 Flash*, reproduz o perfil de desempenho de um modelo pago como o *GPT-4* no mesmo *benchmark* com metodologia replicável; essa questão tem relevância prática direta para equipes sem recursos para APIs comerciais. Segundo, os estudos existentes carecem de uma caracterização qualitativa sistemática dos padrões de erro por categoria de vulnerabilidade: saber por que os modelos erram, se por limitação sintática ou semântica, é o que permite orientar estratégias de melhoria como *prompt engineering* especializado ou combinação com ferramentas estáticas. Este trabalho busca preencher essas duas lacunas.

4. Metodologia

Esta seção descreve o procedimento experimental adotado para avaliar modelos de linguagem na detecção de vulnerabilidades em *smart contracts*. A metodologia replica o arcabouço proposto por [Chen et al. 2025], utilizando o *dataset SmartBugs Curated* e os *scripts* disponibilizados pelos autores, permitindo a comparação direta entre os resultados obtidos[1].

O processo experimental envolve duas etapas: (i) detecção de vulnerabilidades e (ii) padronização dos resultados para análise quantitativa. O código desenvolvido está disponível em repositório público[2].

Foram utilizados 142 contratos do *dataset smartbugs-curated*, organizados segundo a taxonomia DASP Top 10. Antes da análise, foi realizado um pré-processamento para remover anotações explícitas de vulnerabilidade, evitando viés nos resultados.

O script *experiment_gemini.py* automatiza o envio de cada contrato ao *Gemini 2.0 Flash* com um *prompt* estruturado inspirado em [Chen et al. 2025], implementando con-

[1] `https://github.com/BugmakerCC/GPT4scv/tree/master`
[2] `https://github.com/drinith/FONSECA_DVCIEFA_LLMs`

trole de requisições e tratamento de erros. O script *output_limit_gemini.py* converte as respostas textuais em representação binária seguindo a taxonomia DASP Top 10, permitindo o cálculo de *precision*, *recall* e *F1-score* por categoria.

Os experimentos foram conduzidos com os mesmos *scripts* e *dataset* do estudo de referência, mantendo *prompt*, pré-processamento e normalização equivalentes. A única variável não controlada é a versão exata do *GPT-4* utilizada em [Chen et al. 2025], uma vez que atualizações silenciosas da API impedem rastrear a data de execução do estudo original. Por essa razão, a comparação quantitativa deve ser interpretada com cautela; a análise manual por ocorrência busca complementar essa limitação com uma caracterização qualitativa dos erros por categoria.

4.1. Análise Manual

Além da análise automatizada, foi realizada uma revisão manual dos relatórios gerados pelos modelos, *Gemini 2.0 Flash* e *GPT-4*, para verificar a correspondência entre as vulnerabilidades detectadas e o gabarito do *dataset*. Após excluir três contratos classificados como *Other*, foram analisados 139 arquivos para cada 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, com a soma de 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 ($FN = Gabarito - TP$).

Os resultados foram consolidados em planilhas estruturadas de TPs e FPs para cada modelo, disponíveis no repositório público do trabalho[3]. Para reduzir possíveis erros de contagem, utilizou-se um *script* de apoio e uma verificação de consistência com diferentes LLMs, incluindo *Gemini*, *Llama* e *Mistral*. Divergências superiores a dois casos entre as contagens foram revisadas manualmente pelo autor, consultando diretamente os relatórios gerados e o gabarito do *dataset*. A análise foi conduzida por um único avaliador, limitação discutida na Seção 7.

5. Resultados

Nesta etapa, aplicou-se a análise do modelo *Gemini* utilizando normalização semântica para converter as respostas em saídas binárias, indicando a presença ou ausência de vulnerabilidades nas nove categorias do *DASP*. Para garantir comparabilidade com o estudo

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

de referência, adotou-se a mesma abordagem binária, na qual cada arquivo é classificado apenas pela presença ou ausência de uma vulnerabilidade, independentemente do número de ocorrências.

Tabela 1. Análise Binária (esq.) e Manual por Ocorrência (dir.): *Gemini 2.0* vs. *ChatGPT-4*

Vulnerabilidade	Métrica	Gem.	GPT-4
Reentrancy	Prec.	26,7	**35,3**
	Rec.	**100,0**	96,8
	F1	42,2	**51,7**
Access control	Prec.	**18,4**	14,4
	Rec.	**88,9**	83,3
	F1	**30,5**	24,6
Arithmetic	Prec.	15,7	**22,0**
	Rec.	**93,3**	86,7
	F1	26,9	**35,1**
Unch. LL calls	Prec.	**58,6**	53,9
	Rec.	**100,0**	96,1
	F1	**73,9**	69,0
Denial of svc	Prec.	4,7	**6,7**
	Rec.	100,0	100,0
	F1	8,9	**12,6**
Bad randomness	Prec.	**47,1**	42,1
	Rec.	100,0	100,0
	F1	**64,0**	59,3
Front running	Prec.	3,9	**6,9**
	Rec.	**100,0**	50,0
	F1	7,4	**12,1**
Time manip.	Prec.	10,6	**13,8**
	Rec.	**100,0**	80,0
	F1	19,2	**23,5**
Short addr.	Prec.	2,9	**7,1**
	Rec.	100,0	100,0
	F1	5,7	**13,3**

Vulnerabilidade	Métrica	Gem.	GPT-4
Reentrancy	Prec.	19,4	**31,3**
	Rec.	**93,8**	**93,8**
	F1	32,1	**46,9**
Access control	Prec.	**14,3**	10,8
	Rec.	**71,4**	66,7
	F1	**23,8**	18,5
Arithmetic	Prec.	16,8	**23,3**
	Rec.	**78,3**	73,9
	F1	27,7	**35,4**
Unch. LL calls	Prec.	48,1	**54,9**
	Rec.	**85,1**	75,7
	F1	61,5	**63,6**
Denial of svc	Prec.	2,5	**5,4**
	Rec.	**71,4**	**71,4**
	F1	4,8	**10,1**
Bad randomness	Prec.	51,9	**73,5**
	Rec.	46,7	**83,3**
	F1	49,1	**78,1**
Front running	Prec.	2,4	**4,0**
	Rec.	**42,9**	14,3
	F1	4,5	**6,3**
Time manip.	Prec.	12,0	**21,4**
	Rec.	**85,7**	**85,7**
	F1	21,1	**34,3**
Short addr.	Prec.	4,2	**9,1**
	Rec.	**100,0**	**100,0**
	F1	8,0	**16,7**

A Tabela 1 apresenta os resultados da avaliação binária (esquerda) e da análise manual por ocorrência (direita). Na avaliação binária, os dois modelos apresentam desempenho semelhante, com o *Gemini* tendendo a valores mais elevados de *recall*, embora com precisão relativamente baixa. A análise manual revelou diferenças mais claras e evidenciou que a avaliação binária pode superestimar a cobertura: observou-se redução nos valores de *recall*, especialmente em categorias como *Bad Randomness*, onde o modelo detecta apenas algumas instâncias em arquivos extensos. De modo geral, o *GPT-4* apresenta melhor desempenho em várias categorias segundo o F1-score, enquanto o *Gemini* mantém vantagem em *Access Control*.

A maior variação entre as duas avaliações ocorre no *recall* do *Gemini* para *Bad Randomness*, que cai de 100,0% na análise binária para 46,7% na manual ($\Delta = 53,3\%$), e em *Front Running*, com queda de 100,0% para 42,9% ($\Delta = 57,1\%$). Essa diferença reflete o fato de que 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. Na análise manual por ocorrência, cada instância não detectada conta como falso negativo, reduzindo o *recall* de forma mais precisa.

Para verificar a estabilidade das métricas, aplicou-se *bootstrap* não-paramétrico com 10.000 reamostras por pseudo-instância. A partir das planilhas de TPs e FPs por contrato, cada ocorrência individual foi expandida em uma instância (TP: pred=1, real=1; FP: pred=1, real=0; FN: pred=0, real=1), preservando a distribuição de erros por arquivo e a heterogeneidade entre contratos. A cada iteração, essas instâncias são reamostradas com reposição e recalcula-se o F1-score, gerando intervalos de confiança de 95% (IC95%). Embora o denominador do *recall* seja mantido fixo no total do gabarito, limitação decorrente da ausência de mapeamento completo linha a linha para todos os contratos, discutida na Seção 7, o método captura a variabilidade real da precisão e é metodologicamente defensável dado o nível de granularidade disponível. A Tabela 2 apresenta os resultados. Na maioria das categorias os IC95% dos dois modelos se sobrepõem, indicando que as diferenças pontuais não são estatisticamente distinguíveis. A exceção é *Bad Randomness* (*Gemini*: 48,8 [29,1; 67,8] vs *GPT-4*: 77,9 [59,6; 95,4]). Categorias com poucos exemplos no gabarito, como *Front Running*, *Denial of Service* e *Short Addresses*, apresentam intervalos mais amplos, evidenciando maior instabilidade das métricas.

Tabela 2. F1-score com IC95% via *bootstrap* por pseudo-instância (10.000 reamostras)

Vulnerabilidade	Gemini 2.0 F1 [IC95%]	GPT-4 F1 [IC95%]
Reentrancy	32,2 [22,1; 42,7]	46,9 [33,1; 61,2]
Access Control	23,7 [13,3; 35,2]	18,5 [9,6; 28,2]
Arithmetic	27,8 [16,8; 39,7]	35,4 [21,1; 51,0]
Unchecked LL Calls	61,6 [50,5; 72,6]	63,5 [52,0; 75,0]
Denial of Service	4,9 [1,0; 9,7]	10,0 [2,0; 19,6]
Bad Randomness	48,8 [29,1; 67,8]	77,9 [59,6; 95,4]
Front Running	4,5 [0,0; 10,4]	6,2 [0,0; 20,0]
Time Manipulation	21,0 [6,9; 37,7]	34,4 [11,4; 59,5]
Short Addresses	7,9 [0,0; 25,0]	16,5 [0,0; 50,0]

6. Apontamentos nas análises manuais

A auditoria manual complementa os resultados quantitativos com uma análise qualitativa dos padrões de erro observados nas saídas dos modelos [Chen et al. 2025]. De modo geral, o *Gemini* tende a respostas mais verbosas com forte dependência sintática, enquanto o *GPT-4* produz relatórios mais concisos, porém igualmente suscetível à correspondência heurística com padrões conhecidos em detrimento da análise semântica do fluxo de execução. Os arquivos utilizados como exemplo mantêm a nomenclatura original dos relatórios[4], facilitando a rastreabilidade.

6.1. Gemini

A seguir são apresentados casos representativos identificados na auditoria manual dos relatórios gerados pelo *Gemini*.

6.1.1. Reentrancy

Foi observado um número significativo de falsos positivos associados à vulnerabilidade de *reentrancy*. Em diversos casos, o modelo identifica padrões sintáticos re-

[4]https://github.com/drinith/FONSECA_DVCIEFA_LLMs/tree/master/20250619results_gemini_2.0

lacionados ao uso de chamadas externas, como *call.value()*, mas não avalia corretamente a lógica de estado do contrato, como pode ser visto no relatório do solidity *0xb0510d68f210b7db66e8c7c814f22680f2b8d1d6.sol*. Em contratos nos quais não há modificação de estado após a chamada externa ou onde o fluxo é restrito por mecanismos de controle de acesso, a reentrância não se torna explorável. Esse comportamento indica uma priorização de padrões estruturais em detrimento da análise da regra de negócio.

6.1.2. Access Control

Nesta vulnerabilidade, observa-se um caso de falso negativo, no qual o modelo identifica apenas padrões superficiais, sem compreender o mecanismo real da falha. No contrato *arbitrary_location_write_simple.sol*, a LLM aponta a ausência de controle de acesso, mas não reconhece o *underflow* no array *bonusCodes*: na função *PopBonusCode*, a condição *require* é sempre verdadeira para *uint*, permitindo o decremento indevido de *length*, a expansão do array e escrita arbitrária no *storage*. Assim, o modelo identifica apenas um sintoma, mas falha em detectar o real vetor explorável da vulnerabilidade.

6.1.3. Arithmetic

Nesta vulnerabilidade, observa-se um falso negativo no qual o modelo assume que o uso de *SafeMath* garante a segurança do contrato. No entanto, em *BECToken.sol*, a função *batchTransfer* utiliza o operador (*) em vez de *.mul()*, mantendo a operação vulnerável a *overflow*. O modelo infere segurança por padrões estruturais sem verificar sua aplicação real, evidenciando também negligência na análise de variáveis locais.

6.1.4. Unchecked Low-Level Calls

Nesta vulnerabilidade, observa-se um bom desempenho geral, com alta taxa de detecção. No entanto, no contrato *0x89c1b3807d4c67df034fffb62f3509561218d30b.sol* ocorre um falso negativo parcial: o modelo identifica corretamente o uso inseguro de *call.value()*, mas ignora ocorrências de *.send*, como em *SGX_ADDRESS.send*. O fato de *SGX_ADDRESS* ser um endereço constante pode ter levado o modelo a interpretá-lo como confiável, reduzindo a detecção da falha.

6.1.5. Denial of Service

Nesta vulnerabilidade, observa-se um alto número de falsos positivos. O modelo tende a interpretar comportamentos inesperados como causadores de *Denial of Service*: no contrato *parity_wallet_bug_2.sol*, aponta risco de alto consumo de gás em loops, mas a constante *c_maxOwners = 250* limita esse impacto; em contratos como *integer_overflow_1.sol* e *insecure_transfer.sol*, extrapola vulnerabilidades aritméticas, classificando-as indevidamente nessa categoria.

6.1.6. Bad Randomness

Na vulnerabilidade de *bad randomness*, observa-se redução de cobertura em contratos maiores e com múltiplas ocorrências. Em *lucky_doubler.sol* e *smart_billions.sol*, o modelo precisa identificar não apenas o padrão vulnerável, mas suas várias instâncias ao longo do código, em *smart_billions.sol* a lógica de sorteio está distribuída pelo contrato, exigindo maior rastreamento de contexto. Assim, tamanho e repetição da vulnerabilidade impactam negativamente o desempenho.

6.1.7. Front Running

A vulnerabilidade de *Front Running* apresentou baixa precisão, com análises superficiais que consideram apenas a exposição de variáveis ou a ordem das transações. No contrato *0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f.sol*, o modelo aponta risco em funções públicas de configuração sem avaliar se há benefício econômico real para o atacante, indicando priorização de padrões sintáticos sem análise do contexto de exploração.

6.1.8. Time Manipulation

Nesta vulnerabilidade, observam-se falsos positivos associados ao uso do alias *now*. No contrato *0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f.sol*, o timestamp é utilizado apenas para registro em log, sem impacto na lógica do contrato, embora exista possibilidade teórica de manipulação, não há efeito prático. O modelo identifica o padrão mas falha em avaliar a regra de negócio e o contexto da aplicação.

6.1.9. Short Addresses

Na vulnerabilidade de *Short Address*, o *Gemini* sinaliza risco apenas pela presença de parâmetros do tipo *address*. No contrato *0x4e73b32ed6c35f570686b89848e5f39f20ecc106.sol*, a função *SetLogFile(address _log)* possui apenas um parâmetro, sem *uint256* subsequente que possibilite o deslocamento de dados, de modo que o alerta não se sustenta. Esse comportamento evidencia dependência de padrões sintáticos sem análise da estrutura real da chamada.

6.2. GPT-4

A seguir são apresentados casos representativos identificados na auditoria manual dos relatórios gerados pelo *GPT-4*.

6.2.1. Reentrancy

Em *Reentrancy*, o modelo aponta risco no contrato *0x7d09edb07d23acb532a82be3da5c17d9d85806b4.sol*, identificando chamadas externas nas funções *loseWager* e *payout* como potenciais vetores de ataque. Contudo, o padrão *Checks-Effects-Interactions* está corretamente aplicado: o estado do jogador é

invalidado antes de qualquer interação externa, e o modificador *onlyPlayers* bloqueia chamadas reentrantes. O alerta configura um falso positivo decorrente da detecção de chamadas externas sem avaliação do fluxo de estados que as precede.

6.2.2. Arithmetic

No relatório do contrato *wallet_03_wrong_constructor.sol*, o *GPT-4* aponta possível *underflow* na função *withdraw*, identificando a operação de subtração sobre a variável *balances[msg.sender]* como vulnerável. Entretanto, o contrato realiza previamente a verificação *require(amount ¡= balances[msg.sender])*, o que impede a operação insegura ao reverter a transação caso o valor solicitado exceda o saldo disponível. O alerta resulta da presença de um padrão aritmético associado a vulnerabilidades conhecidas, sem considerar as restrições condicionais do fluxo de execução.

6.2.3. Access Control

Em *Access Control*, o modelo identifica controle fraco no contrato *0x89c1b3807d4c67df034fffb62f3509561218d30b.sol*, por depender de *requests[0].requester* como mecanismo de autorização. No entanto, essa posição é inicializada no construtor com o endereço do *deployer* e *requestCnt* é definido como 1, garantindo que requisições reais ocupem apenas índices superiores. Dessa forma, *requests[0].requester* funciona na prática como identificador permanente do proprietário, equivalente a um controle de acesso tradicional. O *GPT-4* falha em rastrear a inicialização do estado ao longo do ciclo de vida do contrato, gerando um falso positivo por análise estrutural sem consideração do contexto de *deploy*.

6.2.4. Unchecked Low-Level Calls

No contrato *reentrancy_insecure.sol*, o *GPT-4* aponta *Unchecked Low-Level Call* pelo uso de *msg.sender.call.value(amountToWithdraw)()*. Entretanto, o retorno da chamada é capturado e verificado via *require(success);*, revertendo a transação em caso de falha. O alerta decorre da associação automática entre *call.value()* e vulnerabilidades conhecidas, sem que o modelo considere a verificação explícita do retorno no fluxo de execução.

6.2.5. Denial of Service

No contrato *0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12.sol*, o *GPT-4* associa a ausência de controle de acesso em *AddMessage* e o uso de chamada externa em *CashOut* a um risco de *Denial of Service*. Contudo, *AddMessage* atua apenas como mecanismo de registro e *CashOut* processa retiradas de forma individual, sem dependência entre usuários. O alerta resulta da extrapolação de padrões sintáticos sem verificação de que as condições apontadas criariam, de fato, um cenário de bloqueio.

6.2.6. Bad Randomness

No arquivo *0x524960d55174d912768678d8c606b4d50b79d7b1.sol*, o GPT-4 aponta *Bad Randomness* pela presença de *keccak256* aplicado a endereços. Porém, não há evidência de que o contrato utilize esse hash para gerar aleatoriedade em decisões críticas, como sorteios ou distribuição de recompensas. O modelo associou a presença de uma função criptográfica ao padrão de aleatoriedade insegura sem verificar seu propósito real no fluxo do contrato.

6.2.7. Front-Running

Em um contrato ERC20, o *GPT-4* aponta *Front-Running* pela presença de *transferFrom*. Entretanto, sua execução é condicionada ao mecanismo de *allowance*, impedindo que agentes externos movimentem tokens sem autorização prévia. O alerta configura um falso positivo pela associação automática entre *transferFrom* e exploração de ordem de transações, sem análise da lógica de autorização do contrato.

6.2.8. Time Manipulation

O *GPT-4* aponta *Time Manipulation* no uso de *block.number* para verificação de expiração de apostas. Diferentemente de *block.timestamp*, o número do bloco é determinado sequencialmente pela rede, com margem de influência negligenciável por parte de mineradores. O falso positivo decorre da generalização do risco de manipulação temporal a qualquer variável de bloco, sem distinção entre o grau de controle que mineradores exercem sobre cada uma.

6.2.9. Short Addresses

Em um contrato com parâmetro *address* (*caddress*), o *GPT-4* aponta *Short Address Attack*. Contudo, a função possui apenas um parâmetro, sem *uint256* subsequente que possibilitaria o deslocamento de dados característico desse ataque. Além disso, a codificação *Application Binary Interface* (ABI) padroniza automaticamente o tamanho dos parâmetros, mitigando o vetor historicamente associado a essa vulnerabilidade. O alerta reflete reconhecimento sintático sem avaliação da estrutura real da chamada.

Os casos apresentados evidenciam que a limitação observada não é específica de um modelo, mas reflete uma característica estrutural das LLMs avaliadas na tarefa de auditoria de *smart contracts*.

7. 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.

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 protocolo de anotação explícito baseado em função e instrução vulnerável (Seção 4.1), 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 opera sobre pseudo-instâncias reconstruídas a partir das contagens de TPs e FPs; contratos com ocorrência única já possuem granularidade completa, mas a ausência de mapeamento linha a linha para contratos com múltiplas ocorrências impede *recall* variável nas reamostras. O método é defensável dado o nível de granularidade disponível; estudos futuros devem registrar predições por linha para estimativas de incerteza mais precisas.

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.

8. Conclusão

Em relação às questões de pesquisa formuladas, os resultados indicam que: **RQ1:** o *Gemini 2.0 Flash* apresenta perfil de comportamento semelhante ao reportado para o *GPT-4* no estudo de referência [Chen et al. 2025], com alto *recall* e baixa precisão na maioria das categorias, com vantagem em *Access Control*. Embora a comparação direta seja limitada pela impossibilidade de rastrear a versão exata do *GPT-4* utilizada em [Chen et al. 2025], os padrões qualitativos observados são consistentes entre os dois modelos. **RQ2:** a avaliação binária superestima o desempenho dos modelos, como evidenciado pela queda de *recall* observada na análise manual por ocorrência, especialmente em categorias como *Bad Randomness*. **RQ3:** os principais padrões de erro identificados foram a dependência de indícios sintáticos em detrimento da análise semântica e a generalização excessiva de vulnerabilidades, como observado nos casos de *Denial of Service*.

Os resultados indicam que o *Gemini 2.0 Flash* pode representar uma alternativa acessível para triagem inicial de vulnerabilidades, especialmente em contextos sem recursos para ferramentas comerciais especializadas. No entanto, as limitações na interpretação semântica do código e na compreensão do contexto de execução da EVM tornam a validação por especialistas humanos indispensável em aplicações críticas.

A auditoria manual revelou padrões recorrentes de erro: falsos positivos pela associação automática entre estruturas de código e vulnerabilidades conhecidas, e falsos negativos pela dificuldade em identificar mecanismos técnicos mais complexos, como dependências de estado ou interações entre variáveis e funções. O *Gemini* tende a respostas mais detalhadas e explicativas, enquanto o *GPT-4* produz relatórios mais diretos e concisos. Ambos, porém, igualmente sujeitos à correspondência heurística em detrimento da análise semântica.

Como trabalhos futuros, destaca-se a investigação de estratégias para mitigar es-

sas limitações, como o uso de *prompt engineering* mais direcionado, a combinação com ferramentas tradicionais de análise estática e dinâmica, e o desenvolvimento de abordagens híbridas que integrem análise sintática e semântica. Outra possibilidade consiste na utilização de *prompts* especializados por categoria de vulnerabilidade, permitindo avaliar se a especialização do processo de análise melhora a capacidade de detecção das LLMs.

As principais contribuições deste trabalho são: (i) a demonstração de que o *Gemini 2.0 Flash*, modelo de acesso gratuito, apresenta capacidade relevante para triagem inicial de vulnerabilidades em *smart contracts*, com perfil de comportamento semelhante ao *GPT-4* em um *benchmark* consolidado, com metodologia replicável e artefatos disponibilizados em repositório público; e (ii) a caracterização qualitativa dos padrões de erro por categoria de vulnerabilidade, identificando a dependência de indícios sintáticos como limitação estrutural compartilhada pelos dois modelos avaliados.

Referências

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

Boi, B., Esposito, C., and Lee, S. (2024). 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.

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

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

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

Google Cloud (2026). Gemini 2.0 flash documentation. `https://cloud.google.com/vertex-ai/docs/generative-ai/models/gemini/2-0-flash`. Acesso em: 09 mar. 2026.

Google DeepMind (2024). 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/`. Acesso em: 09 mar. 2026.

Hu, S., Huang, T., İlhan, F., Tekin, S. F., and Liu, L. (2023). 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.

Larios-Vargas, E., Elazhary, O., Yousefi, S., Lowlind, D., Vliek, M. L., and Storey, M.-A. (2023). Dasp: A framework for driving the adoption of software security practices. *IEEE Transactions on Software Engineering*.

Liao, J.-W., Tsai, T.-T., He, C.-K., and Tien, C.-W. (2019). Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth*

International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pages 458–465. IEEE.

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

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

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

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

OpenAI (2023). Gpt-4 technical overview. Accessed: 2026-03-11.

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

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., and Liu, Y. (2024). 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.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. (2018). 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.

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

Zheng, Z., Xie, S., Dai, H., Chen, X., and Wang, H. (2017). An overview of blockchain technology: Architecture, consensus, and future trends. pages 557–564. cited By 760.