Avaliação Comparativa e Caracterização de Ferramentas SAST e LLMs no OWASP Benchmark

Henrique da Fonseca Diniz Freitas
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
henriquefreitas@dcc.ufmg.br

João Eduardo Montandon de Araújo Filho
Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
joao@dcc.ufmg.br

Resumo

A detecção precoce de vulnerabilidades é um desafio crítico na engenharia de software, tradicionalmente abordado por ferramentas de Teste Estático de Segurança de Aplicações (SAST). Contudo, o excesso de falsos positivos gerados por essas ferramentas afeta severamente a manutenção de sistemas. Recentemente, Grandes Modelos de Linguagem (LLMs) surgiram como alternativas promissoras, capazes de raciocínio semântico. Este trabalho apresenta uma análise comparativa sistemática entre ferramentas SAST (CodeQL e Semgrep) e LLMs (Gemini 2.5 Flash e GPT-5-mini) utilizando o OWASP Benchmark v1.2. Os resultados evidenciam uma dicotomia tecnológica: o CodeQL garantiu cobertura total (Recall de 1.0) em validações determinísticas, enquanto o Gemini demonstrou precisão superior ($\approx$ 90%) em fluxos de dados complexos. O estudo avança na caracterização das falhas, utilizando métricas de risco como Odds Ratio e Absolute Risk Difference para avaliar a exclusividade de detecção e a influência da severidade na probabilidade de acerto. Conclui-se que as abordagens são estritamente complementares, fornecendo diretrizes empíricas para a construção de pipelines híbridos que reduzam a fadiga de alertas na evolução de software.

Palavras-chave

Segurança de Software, Manutenção, SAST, LLM, Detecção de Vulnerabilidades, Fadiga de Alertas, GQM

1 Introdução

A segurança de software consolidou-se como um pilar fundamental na engenharia de sistemas modernos [11]. Com a adoção massiva de metodologias ágeis e a cultura *DevSecOps*, a detecção precoce de vulnerabilidades (estratégia conhecida como *Shift-Left*) deixou de ser um diferencial para se tornar um requisito mandatório. O aumento da complexidade das aplicações e a sofisticação dos ataques cibernéticos exigem ferramentas capazes de identificar falhas antes que o código chegue à produção. Como o software é uma das construções mais complexas da engenharia, é inerente que inconsistências de segurança surjam durante o seu processo evolutivo [14, 16].

Historicamente, as ferramentas de Teste Estático de Segurança de Aplicações (SAST) têm sido a primeira linha de defesa [4]. Ferramentas como CodeQL [7] e Semgrep operam analisando a estrutura sintática e o fluxo de dados do código-fonte, sem a necessidade de execução. Elas são vitais para garantir conformidade e cobrir todo o repositório. No entanto, as SASTs tradicionais enfrentam críticas severas devido à falta de compreensão semântica do código, o que resulta em altas taxas de falsos positivos [2]. Esse ruído excessivo gera a chamada "fadiga de alertas", onde desenvolvedores passam a ignorar avisos de segurança devido ao volume de sinalizações incorretas, prejudicando a manutenção e a evolução contínua do sistema [9].

Recentemente, a ascensão dos Grandes Modelos de Linguagem (LLMs), como GPT e Gemini, introduziu um novo paradigma para o desenvolvimento de software. Estas ferramentas demonstraram capacidade ao automatizar tarefas variadas de engenharia, tais como a geração de código [3], a criação automatizada de casos de teste unitários [12], a migração de sistemas legados e bibliotecas [1, 8], a detecção de *bad smells* [13] e a documentação contextualizada de sistemas complexos [5].

No contexto da Segurança de Informação, os modelos de linguagem podem apresentar vantagens sobre métodos estáticos convencionais ao interpretar não apenas a sintaxe do código, mas também absorver informações contextuais implícitas na estrutura e semântica do código escrito. Estudos preliminares sugerem que esses modelos podem atuar como auditores de segurança mais inteligentes, filtrando falsos positivos e explicando vulnerabilidades complexas através de raciocínio semântico.

Este trabalho tem como objetivo realizar uma análise comparativa rigorosa entre ferramentas SAST tradicionais (CodeQL e Semgrep) e modelos de linguagem de última geração (Gemini e GPT) na detecção de vulnerabilidades em código Java. Para guiar esta investigação, aplicamos a metodologia *Goal-Question-Metric* (GQM). O objetivo principal é analisar ferramentas SAST e LLMs com o propósito de avaliação comparativa, sob o ponto de vista de analistas de segurança, no contexto de manutenção de código no *OWASP Benchmark* v1.2. Para isso, o estudo propõe responder às seguintes perguntas de pesquisa:

- **RQ1:** Qual ferramenta apresenta o melhor desempenho absoluto na detecção de vulnerabilidades no nível de arquivo?
- **RQ2:** Como o desempenho das ferramentas varia entre diferentes grupos de vulnerabilidades?
- **RQ3:** Qual a taxa de exclusividade de detecção de cada abordagem (erros detectados apenas por SAST ou apenas por LLMs)?
- **RQ4:** Como a caracterização das vulnerabilidades (e.g., nível de severidade) influencia a probabilidade de detecção?

Este trabalho apresenta três contribuições principais: (i) um benchmark sistemático comparando ferramentas SAST e LLMs em nível de arquivo; (ii) a aplicação de métricas de risco absoluto e *odds ratio* para quantificar a eficácia preditiva das LLMs [13]; e (iii) uma caracterização detalhada das vulnerabilidades focada na severidade e exclusividade de detecção, orientando a construção de pipelines híbridos de segurança.

O artigo está organizado da seguinte forma. A Seção 2 apresenta trabalhos relacionados. A Seção 3 detalha a metodologia. A Seção 4 apresenta os resultados, enquanto a Seção 5 discute as implicações para a manutenção de software. Por fim, as Seções 6 e 7 apresentam os riscos à validade e as conclusões.

2 Trabalhos Relacionados

A intersecção entre segurança de software e inteligência artificial tem sido alvo de intensa investigação, especialmente com o advento das LLMs. Esta seção agrupa os esforços científicos em três eixos: as limitações da análise estática tradicional, o uso de LLMs na evolução de software e as abordagens híbridas de detecção.

2.1 Análise Estática e o Desafio da Manutenção

A eficácia das ferramentas SAST é amplamente documentada, porém seu uso prático na manutenção de software é frequentemente dificultado pelo alto volume de falsos positivos. Chess e West [4] detalham que o motor de análise dessas ferramentas carece de compreensão semântica profunda. Bessey et al. [2] reforçam que o ruído gerado por ferramentas estáticas é um entrave para adoção na indústria. Esse cenário é corroborado por Johnson et al. [9], que identificaram a "fadiga de alertas" como uma barreira que leva desenvolvedores a negligenciar vulnerabilidades reais.

2.2 LLMs na Evolução e Engenharia de Software

A literatura passou a explorar LLMs como assistentes em diversas etapas do ciclo de vida de software. Schäfer et al. [12] e Chen et al. [3] demonstraram a viabilidade na geração de testes unitários. No campo da manutenção, Barbosa e Hora [1] discutem a transição de bases de código. Um ponto crucial para este trabalho é a detecção de más práticas: Silva et al. [13] conduziram um estudo empírico utilizando o ChatGPT para identificar *code smells*, introduzindo o uso de métricas como *Odds Ratio* e *Absolute Risk Difference* para quantificar a performance dos modelos frente a métodos tradicionais.

2.3 Detecção de Vulnerabilidades e Sistemas Híbridos

Tehrani et al. [15] compararam SASTs contra o GPT-4, reportando que LLMs superam as ferramentas tradicionais em vulnerabilidades de lógica contextual. O estado da arte aponta para a convergência dessas tecnologias: Li et al. [10] propuseram o *IRIS*, enquanto Wang et al. [17] apresentaram o *QLPro*. Diferente desses trabalhos que focam na integração arquitetural direta, o presente estudo foca na caracterização detalhada dos acertos exclusivos e na severidade das falhas no nível de arquivo, buscando fundamentar padrões comportamentais para manutenção de software.

3 Metodologia

Para garantir uma avaliação justa e reprodutível, desenvolveu-se uma arquitetura de pipeline modular baseada no *Goal-Question-Metric* (GQM). O objetivo central foi comparar ferramentas determinísticas (SAST) com modelos probabilísticos (LLMs) sob as mesmas condições de teste. A Figura 1 ilustra as quatro etapas principais do fluxo metodológico adotado neste estudo.

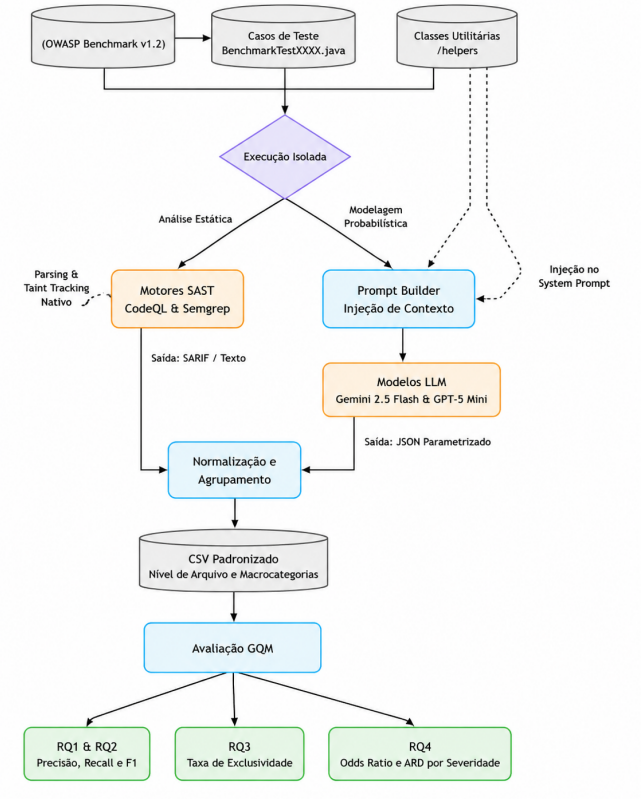

Figura 1: Fluxograma metodológico: da extração do OWASP Benchmark à consolidação estatística das métricas.

3.1 Etapa 1: Dataset e Agrupamento (Extração)

O passo inicial (Etapa 1 na Fig. 1) consistiu na configuração do **OWASP Benchmark** v1.2 [6], um projeto projetado para avaliar a precisão de ferramentas de segurança. O repositório contém 2766 casos de teste em Java, cobrindo vulnerabilidades reais e falsos positivos intencionais (armadilhas). Cada caso de teste é isolado (e.g., `BenchmarkTest00001.java`) e possui um *Ground Truth* exato sobre sua vulnerabilidade.

Para lidar com a disparidade na granularidade com que as ferramentas relatam erros (e.g., CWE-326 vs. CWE-327), aplicamos uma taxonomia de agrupamento baseada na natureza da falha. A Tabela 1 descreve a macrocategoria, o nível de severidade atribuído, os CWEs consolidados e a descrição do problema.

A atribuição dos níveis de severidade (Alta, Média e Baixa) apresentados na taxonomia não ocorreu de forma arbitrária. Ela foi pautada no impacto potencial da exploração da vulnerabilidade sobre os pilares de segurança do sistema, em alinhamento conceitual com as diretrizes do *OWASP Risk Rating Methodology* e do CVSS (*Common Vulnerability Scoring System*):

- **Severidade Alta:** Engloba falhas de injeção estrutural e manipulação de diretórios (e.g., *SQL Injection*, *XSS*, *Path Traversal*). Estas vulnerabilidades permitem a execução direta de comandos maliciosos, vazamento massivo de dados ou

Tabela 1: Taxonomia e Agrupamento de Vulnerabilidades

Macro-Categoria	Severidade	CWEs Mapeados	Descrição
Command Injection	Alta	78, 88	Injeção de comandos no sistema operacional hospedeiro.
SQL Injection	Alta	89, 564	Injeção de consultas maliciosas em bancos de dados.
Path Traversal	Alta	22, 23, 36, 73	Manipulação de caminhos para acessar arquivos não autorizados.
LDAP Injection	Alta	90	Construção de instruções LDAP com dados de entrada não sanitizados.
XPath Injection	Alta	643	Injeção de nós XPath para contornar autenticação ou ler dados.
XSS (Cross-Site)	Alta	79, 80, 81 a 87, 116	Injeção de scripts maliciosos refletidos no navegador do usuário.
Trust Boundary	Média	501	Violação de limites de confiança em sessões da aplicação.
Secure Cookie	Média	614, 1004	Transmissão de cookies sensíveis sem as *flags* de segurança adequadas.
Weak Hash	Baixa	326, 327, 328, 400…	Uso de algoritmos de *hash* obsoletos ou inseguros (e.g., MD5, SHA1).
Weak PRNG	Baixa	330, 338, 200, 532…	Uso de geradores de números pseudoaleatórios criptograficamente fracos.

subversão imediata do fluxo de controle, representando risco crítico e direto de comprometimento.

- **Severidade Média:** Compreende problemas de configuração de segurança e gerenciamento de estado (e.g., *Secure Cookie* e *Trust Boundary*). Embora representem brechas significativas de confidencialidade, sua exploração efetiva geralmente exige condições específicas da sessão ou uma interação secundária do atacante.
- **Severidade Baixa:** Restringe-se ao uso de configurações matemáticas ou criptográficas sub-ótimas (e.g., *Weak Hash* e *Weak PRNG*). Embora reduzam a entropia do sistema, a exploração prática dessas fraquezas demanda elevado esforço computacional (criptoanálise ou força bruta) e raramente resulta em comprometimento imediato sem a combinação com outras falhas.

3.2 Etapa 2: Execução Isolada e Injeção de Contexto

A segunda etapa orquestrou a avaliação das ferramentas de forma isolada, garantindo que SASTs e LLMs fossem submetidas ao mesmo nível de visibilidade de código (Etapa 2 na Fig. 1).

Para a análise estática tradicional, utilizamos o **CodeQL** (com a suíte *java-security-extended*) e o **Semgrep** (com a configuração *auto-config*). O *input* fornecido a essas ferramentas foi o diretório raiz do repositório OWASP Benchmark. Essa abordagem é necessária pois os motores SAST dependem da construção de Árvores de Sintaxe Abstrata (AST) globais para rastrear o fluxo de dados (*Taint Tracking*) através de múltiplas classes e arquivos de forma nativa.

Para as ferramentas baseadas em IA, avaliamos os modelos **Gemini 2.5 Flash** e **GPT-5 Mini** (consultados via API oficial em abril de 2026). Avaliar modelos de linguagem em repositórios inteiros apresenta um desafio conhecido de estouro da janela de contexto (*context window limits*). Portanto, a análise foi arquitetada em nível de arquivo (*file-level*), mas com uma injeção estratégica de dependências para manter a justiça metodológica em relação ao SAST.

System Prompt:

```
You are a senior security auditor analyzing Java
     code for security vulnerabilities. You have
     access to the following helper classes from the
     project (dependency context):
<helpers>
{helpers_context}
</helpers>

Your task is to analyze the provided Java test case
     file and identify any security vulnerabilities (
     CWEs). You MUST return ONLY a valid JSON object
     matching exactly this schema, without markdown
     formatting or code blocks:
{
  "fileName": "BenchmarkTestXXXX.java",
  "isVulnerable": true,
  "vulnerabilitiesFound": [
    {"cwe": "CWE-XX", "line": 45}
  ]
}

If no vulnerabilities are found, set "isVulnerable"
     to false and omit the "vulnerabilitiesFound"
     array.
```

User Prompt:
Analyze the following file: {file_name}

```
<code>
{code_content}
</code>
```

Figura 2: Prompt utilizado para análise automática de vulnerabilidades em código Java.

A Figura 2 detalha a estrutura exata do *prompt* submetido às LLMs. O *design* dessa instrução foi concebido utilizando três técnicas fundamentais de *Prompt Engineering*:

(1) ***Role-Prompting*:** A instrução inicializa definindo o papel do modelo como um "auditor de segurança sênior". Na literatura de IA, delimitar a *persona* restringe o espaço de probabilidade

da rede neural, focando a geração de *tokens* em terminologias técnicas de segurança.

(2) **Injeção de Contexto (*Taint Tracking* Simulado):** O marcador `{helpers_context}` foi dinamicamente substituído por um *script* Python que lia e concatenava o código-fonte integral de todas as classes utilitárias do diretório `/helpers` do projeto. Como grande parte das vulnerabilidades no OWASP Benchmark envolve dados passando por métodos de sanitização externos antes de atingirem o sumidouro (*sink*), injetar essas classes no *System Prompt* forneceu à LLM o contexto global necessário para inferir se um fluxo era malicioso ou seguro.

(3) ***Structured Output* (Saída Parametrizada):** Para viabilizar a extração automatizada de dados em mais de 2.700 execuções por modelo, o *prompt* incluiu um esquema JSON estrito (*Zero-Shot json-mode*). O modelo foi instruído a omitir explicações textuais e retornar unicamente a estrutura de dados contendo o estado da vulnerabilidade e o CWE detectado.

Essa padronização no *input* garantiu que a comparação de desempenho não fosse afetada por alucinações de formatação, isolando a métrica de avaliação unicamente na capacidade analítica de cada modelo.

3.3 Etapa 3: Normalização de Resultados

Na Etapa 3 (Fig. 1), os relatórios heterogêneos (SARIF do CodeQL, JSON das LLMs) foram convertidos em um CSV padronizado. A normalização simplificou o resultado para uma verificação em **nível de arquivo**. Cada linha do CSV continha o ID do teste, o grupo da vulnerabilidade esperado (conforme Tabela 1) e uma *flag* binária indicando se a ferramenta relatou um erro pertencente àquele grupo no arquivo. Isso removeu ruídos de sintaxe de exportação e permitiu comparações exatas de Verdadeiros Positivos (TP) e Falsos Positivos (FP).

3.4 Etapa 4: Avaliação Baseada em *Ensembles* (GQM)

A etapa final focou na extração estatística. Para responder às questões mais avançadas de causa raiz e exclusividade, as ferramentas foram agrupadas em dois grandes blocos operacionais (*Ensembles*): o **SAST Ensemble** (união de CodeQL e Semgrep) e o **LLM Ensemble** (união de Gemini e GPT). Um acerto do *ensemble* ocorre se ao menos uma de suas ferramentas identificar a falha corretamente.

Utilizando essa configuração, e inspirados pela metodologia de Silva et al. [13], calculamos a *Absolute Risk Difference* (ARD) e o *Odds Ratio* (OR) considerando o SAST Ensemble como controle e o LLM Ensemble como tratamento.

4 Resultados

A avaliação processou os 2766 casos de teste, gerando métricas comparativas rigorosas. Esta seção está estruturada para responder diretamente a cada uma das Perguntas de Pesquisa (RQs).

4.1 Desempenho Geral por Ferramenta (RQ1)

RQ1: Qual ferramenta apresenta o melhor desempenho absoluto na detecção de vulnerabilidades no nível de arquivo?

A Tabela 2 consolida as métricas. O CodeQL apresentou o melhor F-measure geral (0.84), impulsionado por um Recall perfeito de 1.00 (nenhum Falso Negativo). Em contraste, o Gemini obteve a maior Precisão do estudo (0.90), indicando uma assertividade muito superior, mas com um custo elevado no Recall.

Tabela 2: RQ1: Desempenho Global das Ferramentas

Ferramenta	Precisão	Recall	F-measure
CodeQL	0.72	1.00	0.84
Semgrep	0.69	0.88	0.77
Gemini	0.90	0.74	0.81
GPT	0.62	0.73	0.67

4.2 Comparativo Detalhado por Grupo de Vulnerabilidade (RQ2)

RQ2: Como o desempenho das ferramentas varia entre os diferentes grupos de vulnerabilidades?

Para preservar a clareza analítica e evitar redundância de tendências em um espaço restrito, esta análise granular foca exclusivamente nos representantes de melhor desempenho geral de cada paradigma, validados na RQ1: CodeQL (SAST) e Gemini (LLM). Para compreender as nuances que compõem o desempenho global, é imperativo analisar a distribuição de Verdadeiros Positivos (TP) e Falsos Positivos (FP) em cada categoria. A Tabela 3 detalha as métricas absolutas e relativas das duas ferramentas líderes de cada paradigma (CodeQL e Gemini), organizadas por nível de severidade.

Tabela 3: RQ2: Desempenho Detalhado (CodeQL vs. Gemini)

Grupo de Vuln.	CodeQL (SAST)		Gemini (LLM)	
	TP / FP	F1	TP / FP	F1
Severidade Alta				
SQL Injection	272 / 207	0.72	271 / 11	0.97
XSS	246 / 90	0.84	241 / 5	0.97
Path Traversal	133 / 66	0.80	132 / 5	0.97
Command Inject.	126 / 64	0.79	125 / 3	0.98
LDAP Injection	27 / 13	0.80	27 / 2	0.96
XPath Injection	15 / 7	0.81	15 / 1	0.96
Severidade Média				
Trust Boundary	83 / 24	0.87	36 / 0	0.60
Secure Cookie	36 / 0	1.00	36 / 0	1.00
Severidade Baixa				
Weak Hash	259 / 60	0.89	107 / 40	0.52
Weak PRNG	218 / 0	1.00	61 / 49	0.37

** O Recall do CodeQL atingiu 1.00 em todos os grupos.*

A análise granular dos dados revela comportamentos antitéticos entre as abordagens, fortemente dependentes da natureza da vulnerabilidade:

O peso dos Falsos Positivos no SAST: O CodeQL alcança um *Recall* perfeito (1.00) em absolutamente todas as categorias, validando sua exaustividade. Contudo, em vulnerabilidades de Severidade Alta, focadas em fluxo de dados, a ferramenta sofre com uma extrema degradação de precisão. Em *SQL Injection*, por exemplo, para encontrar 272 vulnerabilidades reais (TP), o motor estático sinalizou incorretamente 207 arquivos seguros (FP). Esse volume massivo de falsos alarmes é a raiz empírica da "fadiga de alertas", demonstrando que ferramentas baseadas em AST lutam para compreender métodos complexos de sanitização ao longo da malha do código.

A Seletividade Semântica das LLMs: Em contrapartida, o modelo Gemini 2.5 Flash mitigou o ruído quase em sua totalidade nas categorias críticas. No mesmo grupo de *SQL Injection*, a LLM perdeu apenas 1 vulnerabilidade real ($TP = 271$), mas cometeu apenas 11 falsos alarmes. Em *Command Injection*, foram apenas 3 falsos positivos contra 64 do CodeQL. O *F-measure* estavelmente acima de 0.96 nessas categorias comprova que o raciocínio semântico contextual da LLM consegue distinguir com alta acurácia um dado purificado (*sanitized*) de um dado malicioso (*tainted*).

O Colapso em Regras Estritas: A superioridade da inteligência artificial, no entanto, colapsa em grupos de Severidade Baixa, que exigem conformidade matemática. No grupo *Weak PRNG* (uso de pseudoaleatoriedade fraca), o CodeQL obteve métricas impecáveis ($TP = 218, FP = 0$). O Gemini, por sua vez, demonstrou grave confusão conceitual, identificando apenas 61 falhas reais e sinalizando 49 falsos positivos ($Recall = 0.27, F\text{-}measure = 0.37$). Este abismo de performance reitera que LLMs não são ferramentas de *compliance* estrutural, tendendo a falhar quando a vulnerabilidade não depende da interpretação do fluxo, mas sim da verificação estrita de bibliotecas ou classes instanciadas (e.g., `java.util.Random` vs `java.security.SecureRandom`).

4.3 Taxa de Exclusividade e Matriz de Acertos (RQ3)

RQ3: Qual a cobertura e a capacidade de detecção conjunta (sobreposição) entre as abordagens?

Para compreender a sinergia entre as tecnologias, a Tabela 4 apresenta a Matriz de Contingência comparando os acertos dos *Ensembles*.

Tabela 4: Matriz de Contingência: Acertos do *SAST Ensemble* versus *LLM Ensemble*.

		LLM Ensemble		Total
		Correto	Incorreto	
SAST Ensemble	**Correto**	1159	256	1415
	Incorreto	0	1351	1351
	Total	1159	1607	2766

Observa-se que 1159 instâncias foram detectadas corretamente por ambos os paradigmas. Contudo, 256 casos foram identificados *exclusivamente* pelas ferramentas SAST. O dado mais revelador é a intersecção de incorreções do SAST com acertos da LLM: o valor é zero. Ou seja, não há nenhuma vulnerabilidade no *dataset* que tenha sido identificada unicamente pela IA. Isso reflete a natureza exaustiva do SAST em oposição à natureza probabilística e omissa das LLMs.

4.4 Influência da Severidade e *Odds Ratio* (RQ4)

RQ4: Como a caracterização das vulnerabilidades (e.g., nível de severidade) influencia a probabilidade de detecção?

A Tabela 5 apresenta o *Recall* para ambos os grupos, acompanhado do *Absolute Risk Difference* (ARD) e do *Odds Ratio* (OR), separados pelos níveis de Severidade (conforme classificação da Tabela 1).

Tabela 5: Métricas de Risco e Recall por Nível de Severidade

Severidade	SAST (R)	LLM (R)	ARD	OR
Alta	1.00	1.00	0.00	1.000
Média	1.00	0.60	-0.39	0.006
Baixa	1.00	0.56	-0.43	0.001

Os resultados indicam uma disparidade estatística agressiva com base na gravidade do problema. Em problemas de **Severidade Alta**, as LLMs possuem chances idênticas de sucesso ao motor sintático ($OR = 1.00$). No entanto, para **Severidade Baixa**, as chances de uma IA generativa obter um resultado correto despencam ($OR = 0.001$), demonstrando uma forte dificuldade estrutural em lidar com regras matemáticas rígidas.

Para aprofundar essa variação, a Tabela 6 decompõe o *F-measure* dos melhores representantes de cada *ensemble* (CodeQL e Gemini) para cada grupo da taxonomia, validando a performance individual.

Tabela 6: F-measure de CodeQL e Gemini por Grupo de Severidade

Grupo	Severidade	CodeQL	Gemini
Path Traversal	Alta	0.80	0.97
SQL Injection	Alta	0.72	0.97
XSS	Alta	0.84	0.97
Command Injection	Alta	0.79	0.98
LDAP Injection	Alta	0.80	0.96
XPath Injection	Alta	0.81	0.96
Secure Cookie	Média	1.00	1.00
Trust Boundary	Média	0.87	0.60
Weak Hash	Baixa	0.89	0.52
Weak PRNG	Baixa	1.00	0.37

5 Discussão

Os dados empíricos corroboram a hipótese inicial de que SAST e LLMs possuem fronteiras operacionais estritamente complementares.

1. Ferramentas SAST são superiores em exaustividade e conformidade matemática. Com um Recall de 1.00, o SAST demonstrou superioridade na aplicação de regras de conformidade

estritas, brilhando em vulnerabilidades de baixa severidade algorítmica (como algoritmos *Hash* ou geradores de entropia). Ferramentas estáticas rastreiam inequivocamente instâncias de classes inseguras, não sendo iludidas por nomenclaturas. Elas formam a fundação incontestável da segurança, visto que o custo de não identificar tais falhas é inaceitável.

2. LLMs são superiores na detecção de falhas semânticas baseadas em fluxo. A Tabela 6 evidencia que, para Injeções de severidade Alta, o Gemini atinge $F\text{-}measure > 0.96$. A IA generativa conseguiu analisar os arquivos `/helpers` injetados no contexto e deduzir semanticamente se os dados haviam sido sanitizados antes de atingirem o sumidouro de execução. Esse feito confunde motores analíticos convencionais, provando que LLMs resolvem cirurgicamente o principal defeito histórico do SAST: a gigantesca quantidade de Falsos Positivos em caminhos lógicos complexos.

3. A estratégia ideal é a combinação híbrida das tecnologias. A Matriz de Contingência (Tabela 4) cristaliza o cenário futuro do *DevSecOps*. Como as LLMs não encontraram nenhuma falha que o SAST não tivesse alertado, é inviável usá-las de forma autônoma. O *pipeline* ideal deve acionar motores SAST como rastreadores exaustivos de base, coletando todas as anomalias do repositório. Em seguida, os alarmes ruidosos referentes a fluxos semânticos de alta severidade (Injeções) devem ser submetidos à LLM. Atuando como filtro secundário, a IA elimina os alarmes incorretos, resolvendo o problema crônico da fadiga de alertas nas equipes de desenvolvimento.

6 Riscos à Validade

Para garantir o rigor do estudo empírico, identificamos e mitigamos potenciais ameaças à validade:

Validade Interna: Existe a possibilidade de viés na categorização dos CWEs e na construção dos *prompts*. Para mitigar, a taxonomia foi fundamentada na documentação da OWASP. A injeção de dependências no *prompt* foi realizada via *scripts* Python determinísticos, sem intervenção humana que pudesse favorecer modelos. Adicionalmente, a categorização em *ensembles* foi usada nas métricas de risco para mitigar o viés de analisar apenas as oscilações de uma ferramenta específica.

Validade Externa: Os projetos derivam do *OWASP Benchmark*. Por se tratar de testes sintéticos, isso pode limitar a generalização para códigos legados de produção. Contudo, a padronização deste *benchmark* na literatura garante comparabilidade incontestável.

Validade das Conclusões: O desempenho das LLMs pode oscilar devido à natureza estocástica dos modelos. Para minimizar essa ameaça, os modelos foram parametrizados com a menor temperatura de criatividade permitida e os dados consolidados sobre uma amostra vasta (2766 execuções) [13], diluindo alucinações.

7 Conclusão

Este trabalho apresentou uma análise comparativa sistemática entre abordagens SAST tradicionais e Modelos de Linguagem na detecção de vulnerabilidades. Os resultados consolidam uma dicotomia tecnológica: ferramentas SAST garantem a cobertura e conformidade, ao passo que as LLMs reduzem o esforço de manutenção através de uma elevada precisão contextual em fluxos de dados complexos.

As métricas de exclusividade e caracterização evidenciam que a substituição total de SAST por LLMs não é viável devido ao risco de omissão em regras estritas. Recomenda-se a adoção de pipelines híbridos onde o SAST atue na garantia basal e a LLM seja utilizada como filtro semântico inteligente para conter a fadiga de alertas nas equipes de desenvolvimento.

Disponibilidade de Artefatos

Os scripts de automação do pipeline de avaliação, os prompts utilizados nas execuções das LLMs e os dados consolidados de métricas estão disponíveis no repositório para o processo de revisão em: https://github.com/henriquefdf/llm-and-sast-comparison/.

Referências

[1] L. Barbosa and A. Hora. 2022. How and Why Developers Migrate Python Tests. In *Proc. of SANER*.
[2] Al Bessey et al. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* (2010).
[3] Y. Chen et al. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Proc. of FSE*.
[4] Brian Chess and Jacob West. 2007. *Secure Programming with Static Analysis*. Addison-Wesley Professional.
[5] I. Documentation et al. 2023. Contextualized Documentation of Complex Systems. In *Proc. of ASE*.
[6] OWASP Foundation. 2025. OWASP Top 10 Web Application Security Risks.
[7] GitHub. 2025. CodeQL: Semantic code analysis engine.
[8] M. Islam et al. 2023. On the Migration of Legacy Systems. In *Proc. of ICSME*.
[9] Brittney Johnson et al. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*.
[10] Z. Li, S. Dutta, and M. Naik. 2024. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).
[11] Gary McGraw. 2006. *Software Security: Building Security In*. Addison-Wesley Professional.
[12] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* (2024).
[13] L. L. Silva, J. Silva, J. E. Montandon, and M. T. Valente. 2024. Detecting code smells using chatgpt: Initial insights. In *Proc. of ESEIW-ERVR*.
[14] Ian Sommerville. 2011. *Engenharia de software*. Pearson Prentice Hall.
[15] M. G. Tehrani, E. Sultanow, W. J. Buchanan, and M. Houmani. 2025. LLM vs. SAST: A Technical Analysis on Detecting Coding Bugs of GPT4-Advanced Data Analysis. *arXiv preprint arXiv:2506.15212* (2025).
[16] Marco Túlio Valente. 2020. *Engenharia de Software Moderna*. Independente.
[17] Y. Wang et al. 2025. QLPro: Automated Code Vulnerability Discovery via LLM and Static Code Analysis Integration. *arXiv preprint arXiv:2506.23644* (2025).