Avaliação comparativa do desempenho de inteligências artificiais generativas e ferramentas tradicionais na análise de código-fonte JavaScript

Rayane Pimentel[1], **Claudia Bianchi Progetti** [2]

[1]Serviço Nacional de Aprendizagem do Comércio (Senac) – São Carlos, SP, Brasil

[2]Serviço Nacional de Aprendizagem do Comércio (Senac) – São Paulo, SP, Brasil

rayanepimentel101@gmail.com, claudia.bprogetti@sp.senac.br

Abstract. *Comparative study between SAST tools (Semgrep/SonarQube) and LLM models (DeepSeek/CodeLlama) for detecting JavaScript vulnerabilities (OWASP Juice Shop). Results reveal complementarity: SASTs achieve 100% precision for standard vulnerabilities (XSS/SQLi), while LLMs offer higher recall (70% in DeepSeek) for contextual threats (NoSQLi/access control). The 22-45% false positive rate in LLMs calls for filtering strategies. We demonstrate that hybrid pipelines optimally combine SAST accuracy with LLM coverage. The study contributes empirical evidence for the adoption of LLMs in security pipelines, highlighting challenges such as the mitigation of false positives.*

Resumo. *Estudo comparativo entre ferramentas SAST (Semgrep/SonarQube) e modelos LLM (DeepSeek/CodeLlama) na detecção de vulnerabilidades em JavaScript (OWASP Juice Shop). Resultados revelam complementaridade: SASTs alcançam 100% de precisão para vulnerabilidades padrão (XSS/SQLi), enquanto LLMs oferecem maior recall (70% no DeepSeek) para ameaças contextuais (NoSQLi/Broken Access Control). A taxa de 22-45% de falsos positivos em LLMs demanda estratégias de filtragem. Demonstramos que pipelines híbridos combinam de forma ideal a precisão SAST com cobertura LLM. O estudo contribui com evidências empíricas para a adoção de LLMs em pipelines de segurança, destacando desafios como a mitigação de falsos positivos.*

1. Introdução

A segurança de aplicações web está em constante ameaça, principalmente quando a linguagem mais usada da web, o JavaScript, apresenta desafios únicos à análise automática de código. JavaScript é a linguagem mais utilizada no desenvolvimento web, presente em 98,3% dos sites para front-end [W3Techs 2024] e amplamente adotado no back-end via Node.js [Stack Overflow]. No entanto, a flexibilidade de JavaScript (ex.: tipagem dinâmica, execução assíncrona), essencial para o desenvolvimento moderno, facilita a introdução de riscos de segurança. Por exemplo, vulnerabilidades como Cross-Site Scripting (XSS) e SQL Injection são comuns em aplicações JavaScript [Snyk 2022], sendo também classificadas entre as vulnerabilidades incluídas no relatório do Open Web Application Security Project (OWASP), OWASP Top 10 [OWASP Foundation 2023].

Ferramentas tradicionais de Static Application Security Testing (SAST), análise estática de segurança em aplicações, como Checkmarx, SonarQube e Semgrep, são amplamente utilizadas para detecção automatizada de vulnerabilidades. Essas ferramentas analisam o código sem executá-lo, utilizando técnicas como análise de sintaxe, fluxo de

dados e controle de fluxo. No entanto, enfrentam desafios, como dificuldade em detectar vulnerabilidades contextuais, como lógicas de negócios e alta taxa de falsos positivos [Le et al. 2024] [Wadhams et al. 2024].

Large Language Models (LLMs), como GPT-4, CodeLlama e DeepSeek, surgiram como uma alternativa promissora, pois interpretam a intenção das pessoas desenvolvedoras e fornecem uma análise mais contextualizada, reduzindo limitações das abordagens tradicionais. Embora estudos mostrem resultados positivos em linguagens como Java, C e Python, há uma lacuna de pesquisa em relação a JavaScript, linguagem que impõe desafios a análise automatizada [Zhou et al. 2024].

Este trabalho visa comparar o desempenho de LLMs e SASTs na identificação de vulnerabilidades OWASP Top 10 em código JavaScript, utilizando como base o OWASP Juice Shop. Os objetivos específicos incluem: (1) extrair e estruturar um conjunto de vulnerabilidades a partir do OWASP Juice Shop, formando um dataset representativo para avaliação comparativa das ferramentas; (2) avaliar precisão, recall e F1-score das ferramentas; (3) avaliar o impacto operacional das ferramentas, como o tempo gasto com falsos positivos [Kiminich 2024].

Este estudo busca investigar se os LLMs superam SASTs apresentando maior eficácia na detecção de vulnerabilidades e contribuam para a redução de falsos positivos em JavaScript. Além disso, este estudo visa orientar desenvolvedores na escolha de ferramentas adequadas e apoiar pesquisas futuras sobre a aplicação de LLMs em segurança web. Dessa forma, esta pesquisa se propõe a preencher uma lacuna crítica na segurança de aplicações web, investigando o papel emergente dos LLMs no cenário JavaScript. Ao investigar o papel emergente dos LLMs na segurança de aplicações JavaScript, este estudo pode contribuir não apenas para o avanço acadêmico, mas também para a adoção segura de IA no desenvolvimento web moderno.

2. Referêncial Teórico

Esta seção organiza-se em cinco pilares teóricos que sustentam a análise proposta: (1) segurança em aplicações JavaScript, (2) testes de segurança em aplicações estáticas (SAST), (3) aplicação de Inteligência Artificial (IA) na análise de código, (4) fundamentação em métricas de avaliação e (5) síntese conceitual e identificação de lacunas de pesquisa. A integração desses eixos visa construir a base metodológica do estudo, conforme detalhado nas subseções seguintes.

2.1. Segurança em Aplicações JavaScript

JavaScript é a linguagem predominante no desenvolvimento web, mas sua flexibilidade como tipagem dinâmica, execução assíncrona e manipulação dinâmica do DOM, a torna suscetível a riscos de segurança. Essas características dificultam a análise estática de qualidade, pois a identificação de vulnerabilidades muitas vezes depende de contexto de execução ou interações dinâmicas, impactando diretamente atributos de qualidade como confiabilidade e segurança [International Organization for Standardization 2023].

Essa relação entre segurança e qualidade de software é reforçada por relatórios como o Snyk Top 10 JavaScript Vulnerabilities e OWASP Top 10, que destacam vulnerabilidades críticas em JavaScript [Snyk 2022].

Entre as vulnerabilidades mais comuns, destacam-se:

- **Cross-Site Scripting (XSS)**: Permite a injeção de scripts maliciosos no navegador da vítima. Pode ocorrer tanto no lado servidor (reflected/stored XSS) quanto no cliente (DOM-Based XSS).

- **Cross-Site Request Forgery (CSRF)**: Explora a confiança do site em relação ao navegador do usuário, permitindo que ações sejam realizadas sem o seu consentimento.
- **SQL Injection**: Permite a execução de comandos SQL arbitrários através da manipulação de entradas da aplicação, podendo comprometer dados e integridade do sistema.
- **DOM-Based XSS**: Variante do XSS que ocorre exclusivamente no lado cliente, explorando modificações inseguras no DOM por meio de entradas manipuladas.
- **No Rate Limiting**: Ausência de limitação de requisições permite ataques de força bruta, DoS e bloqueio de contas.

Essas vulnerabilidades podem ser subdetectadas por ferramentas SASTs tradicionais, que enfrentam desafios como análises contextuais limitadas e dificuldade de compreender lógica de execução dinâmica. Por outro lado, LLMs demonstram potencial para interpretar contextos dinâmicos como manipulação do DOM, com potencial para aumentar a eficiência da garantia da qualidade de software.

2.2. Static Application Security Testing (SAST)

O SAST é uma metodologia amplamente utilizada para análise estática de código-fonte, identificando vulnerabilidades de segurança antes da execução do software. Ferramentas SASTs como SonarQube, Checkmarx e Semgrep funcionam por meio de regras predefinidas que analisam sintaxe, fluxo de dados e padrões de código inseguros, alinhando-se a frameworks como o OWASP Top 10 e SANS Top 25 [Palo Alto Networks].

O processo de SAST envolve diversas etapas, iniciando pela análise estrutural do código-fonte. A ferramenta realiza o parsing do código, gerando uma Árvore Sintática Abstrata (AST) que representa a estrutura do programa, como funções, laços, condicionais e variáveis. Em seguida, são aplicadas análises de fluxo de controle e fluxo de dados, permitindo identificar caminhos de execução e rastrear o fluxo de informações entre variáveis e componentes da aplicação. Isso possibilita a detecção de falhas como manipulação insegura de dados. As ferramentas utilizam conjuntos de regras de segurança baseadas em padrões reconhecidos, como OWASP Top 10 e CWE/SANS Top 25. Essas regras orientam a identificação de práticas de codificação insegura e possíveis vulnerabilidades. Além disso, técnicas de correspondência de padrões e análise semântica são utilizadas para correlacionar construções do código com vulnerabilidades conhecidas, considerando tanto a estrutura quanto o comportamento do programa [Palo Alto Networks].

Apesar de sua utilidade, SASTs enfrentam desafios específicos, como a dificuldade em analisar código assíncrono como promises, callbacks, comportamentos gerando falsos positivos, e a incapacidade de interpretar lógicas de negócio complexas ou interações em tempo real, o que limita a detecção precisa de vulnerabilidades [Wadhams et al. 2024].

2.3. Inteligência Artificial na Análise de Código

A integração de IA na análise de código-fonte, especialmente por meio de LLMs, está revolucionando a detecção de vulnerabilidades e a garantia da qualidade de software. Essa abordagem combina técnicas como Processamento de Linguagem Natural (PLN) e modelos generativos para superar limitações de métodos tradicionais, como SASTs, em linguagens dinâmicas como JavaScript [IBM 2024].

Segundo a IBM [IBM 2024], a revisão de código por IA pode ser estruturada em quatro componentes principais:

- **Análise de Código Estático**: examina o código-fonte sem executá-lo, detectando problemas precoces de segurança e manutenção. Os modelos de IA utilizam os dados gerados por essa análise para recomendar melhorias.
- **Análise de Código Dinâmico**: executa o código e testa seu comportamento em tempo real, encontrando problemas que só se manifestam em execução, como gargalos de desempenho ou falhas de segurança.
- **Sistemas Baseados em Regras**: utilizam conjuntos de regras e boas práticas para verificar a conformidade com padrões de segurança e estilo. São eficazes para garantir consistência e reduzir erros sintáticos e de estilo.
- **PLN e LLMs**: modelos como CodeLlama e DeepSeek são exemplos de LLMs especializados em código, treinados com grandes volumes de dados e capazes de compreender estrutura, lógica e intenção de trechos complexos de código-fonte. Isso os torna capazes de detectar falhas contextuais e sugerir correções mais precisas do que abordagens tradicionais.

A aplicação de IA na análise de código oferece benefícios significativos, como a contextualização semântica, onde LLMs interpretam comportamentos dinâmicos como promises e callbacks em Node.js, com isso esperasse que reduza falsos positivos comuns em SASTs . Entretanto, persistem desafios: LLMs podem gerar falsos positivos/negativos e a sinalização de código seguro como vulnerável como document.write intencional [Zhou et al. 2024].

2.4. Fundamentação em Métricas de Avaliação

Para avaliar o desempenho de ferramentas de análise de vulnerabilidades, este estudo adota quatro métricas clássicas da literatura, precisão, recall, F1-score e taxa de falsos positivos (FP Rate). Essas métricas permitem quantificar a confiabilidade dos alertas, cobertura da detecção e o impacto operacional dos falsos positivos [Hu et al. 2024].

A precisão mede a confiabilidade dos alertas, indicando a proporção de vulnerabilidades reais entre os casos reportados. Alta precisão é essencial para evitar falsos positivos, que consomem tempo em revisões desnecessárias, um problema recorrente em SASTs tradicionais [Zhou et al. 2024].

$$\text{Precisão} = \frac{\text{Verdadeiros Positivos (VP)}}{\text{Verdadeiros Positivos (VP)} + \text{Falsos Positivos (FP)}} \tag{1}$$

O recall avalia a cobertura de detecção, representando a capacidade de identificar todas as vulnerabilidades existentes. Um recall elevado é prioritário em cenários críticos, como aplicações bancárias, onde falhas não detectadas (falsos negativos) podem ter impactos severos.

$$\text{Recall} = \frac{\text{Verdadeiros Positivos (VP)}}{\text{Verdadeiros Positivos (VP)} + \text{Falsos Negativos (FN)}} \tag{2}$$

O F1-Score equilibra precisão e recall, sendo ideal para comparar ferramentas em cenários onde ambos os aspectos são igualmente relevantes.

$$\text{F1-Score} = \frac{2 \times \text{precisão} \times \text{recall}}{\text{precisão} + \text{recall}} \tag{3}$$

Por fim, a métrica FP Rate Operacional quantifica o ruído nos alertas, representando a porcentagem de alertas incorretos em relação ao total gerado. No estudo, será utilizado para comparar o custo operacional de SASTs e LLMs.

$$\text{FP Rate Operacional} = \frac{\text{Falsos Positivos (FP)}}{\text{Verdadeiros Positivos (VP)} + \text{Falsos Positivos (FP)}} \quad (4)$$

Essas métricas foram selecionadas por abordarem tanto a eficiência técnica (precisão, recall) quanto o impacto operacional (FP Rate), alinhando-se aos objetivos do trabalho. Por exemplo, a hipótese de que LLMs reduzirão falsos positivos em JavaScript será validada pela correlação entre alta precisão (proporção de alertas corretos) e baixo FP Rate (percentual de alertas incorretos). Além disso, o F1-Score permitirá identificar qual ferramenta oferece o melhor equilíbrio entre detecção abrangente (recall) e confiabilidade (precisão), critério essencial para adoção em projetos reais, onde a eficiência operacional (minimização de tempo gasto com falsos positivos) e a segurança (evitar falsos negativos) são prioritários.

2.5. Síntese Conceitual e Lacuna de Pesquisa

A tríade formada pela dinamicidade do JavaScript, as limitações contextuais das ferramentas SASTs e o potencial analítico dos LLMs evidencia uma lacuna crítica na literatura: a ausência de estudos empíricos que comparem essas abordagens sob métricas técnicas (precisão, recall) e operacionais (FP Rate) em ecossistemas JavaScript. Enquanto os SASTs apresentam elevadas taxas de falsos positivos, afetando a eficiência dos fluxos de desenvolvimento, e LLMs demonstram eficácia em linguagens estáticas como Java e dinâmicas com Python, mas ainda carecem de evidências sobre seu desempenho em cenários JavaScript complexos, onde a manipulação dinâmica do DOM e Promises desafiam modelos tradicionais. Esta pesquisa busca preencher esse vazio por meio de uma avaliação sistemática baseada no OWASP Juice Shop, simulando condições próximas ao ambiente de desenvolvimento web real [Wadhams et al. 2024] [Zhou et al. 2024].

3. Método de Pesquisa

Este estudo adota uma abordagem quantitativa para comparar o desempenho de ferramentas SAST e LLMs na detecção de vulnerabilidades em JavaScript. A metodologia divide-se em quatros etapas principais, (1) construção de um dataset validado, (2) pré-processamento do código-fonte, (3) configuração das ferramentas, e (4) execução automatizada com métricas precisas.

3.1. Dataset controlado

O dataset utilizado foi baseado na aplicação *OWASP Juice Shop* (versão v17.3.0), uma aplicação web intencionalmente vulnerável, amplamente reconhecida como referência para testes de segurança. Para garantir validade estatística e reprodutibilidade, selecionou-se um subconjunto de 15 arquivos: 10 contendo vulnerabilidades documentadas (**VULN**) e 5 seguros (**SAFE**). Os arquivos **VULN** representam falhas críticas do OWASP Top 10 2021, como *Cross-Site Scripting (XSS)* e *NoSQL Injection*, mapeadas diretamente aos desafios oficiais da aplicação. Já os **SAFE** correspondem a correções validadas pelo projeto, identificados pelo sufixo **correct.ts** em seu caminho. Essa bipartição permitiu calcular métricas de precisão e recall sem ambiguidades, já que o status de cada arquivo (vulnerável ou seguro) foi pré-validado pela documentação do Juice Shop.

A Tabela 1 detalha as vulnerabilidades selecionadas e seus respectivos arquivos.

Tabela 1. Vulnerabilidades e Arquivos Seguros

ID	Vulnerabilidade	Arquivo
VULN-01	XSS	search-result/search-result.component.ts
VULN-02	Injection	routes/login.ts
VULN-03	Injection	routes/search.ts
VULN-04	NoSQL Injection	routes/updateProductReviews.ts
VULN-05	Broken Access Control	frontend/src/app/app.routing.ts
VULN-06	Sensitive Data Exposure	server.ts
VULN-07	SSRF	routes/profileImageUrlUpload.ts
VULN-08	CSRF	routes/updateUserProfile.ts
VULN-09	Broken Access Control	routes/basketItems
VULN-10	Validação Insuficiente	routes/deluxe.ts
SAFE-01	Safe	unionSqlInjectionChallenge_2_correct.ts
SAFE-02	Safe	tokenSaleChallenge_3_correct.ts
SAFE-03	Safe	accessLogDisclosureChallenge_1_correct.ts
SAFE-04	Safe	adminSectionChallenge_1_correct.ts
SAFE-05	Safe	dbSchemaChallenge_2_correct.ts

3.2. Pré-processamento do Código-Fonte

Antes da análise, aplicou-se um pré-processamento para remover ruídos não funcionais, como comentários de licença, marcadores de desafio (ex: `// vuln-code-snippet`) e blocos multilinha irrelevantes, preservando apenas a lógica executável. Essa etapa foi crítica para evitar vazamento de contexto em LLMs, por exemplo, dicas textuais que pudessem influenciar artificialmente a detecção. As ferramentas SAST, por sua vez, analisaram o código original, pois não são influenciadas por comentários.

3.3. Ferramentas e Configurações

Foram selecionadas duas abordagens para análise estática de segurança: ferramentas SAST e modelos de LLM, todas configuradas em ambiente gratuito para garantir reprodutibilidade e aderência a contextos acadêmicos. Para a categoria SAST, foram escolhidas: SonarQube Community Build (versão 25.5) e Semgrep (versão 1.49). Ambas as ferramentas foram integradas a um pipeline de CI/CD via GitHub Actions, permitindo análise automatizada. Para avaliação com LLMs, foram utilizados: CodeLlama-7b (via Ollama) DeepSeek-Coder-1.3b (via Ollama). Os modelos foram utilizados em suas versões base (sem fine-tuning), com prompts estruturados para análise de vulnerabilidades, como no código 1.

Código 1. Prompt utilizado para análise

```
Analise os riscos de segurança no código abaixo, seguindo o
   OWASP Top 10. Retorne APENAS se houver vulnerabilidades, no
   formato JSON abaixo. Caso contrário, retorne "Código seguro".

{
  "Arquivo": "Nome do arquivo",
  "Trecho Vulnerável": "O snippet de código específico que
     contém a vulnerabilidade.",
```

```
  "Tipo da Vulnerabilidade": "A categoria da vulnerabilidade",
  "Descrição Breve": "Uma explicação concisa (1-2 frases) de
     por que esse trecho é vulnerável"
}
```

3.4. Execução e Coleta de Resultados

A execução do experimento foi dividida em três etapas principais. Primeiramente, foi realizada uma varredura inicial, na qual as ferramentas SAST e os modelos LLM analisaram exclusivamente os arquivos do dataset controlado da aplicação OWASP Juice Shop, buscando identificar possíveis vulnerabilidades. Em seguida, os alertas gerados foram classificados em três categorias: Verdadeiros Positivos (VP), correspondentes às vulnerabilidades corretamente detectadas com base no ground truth oficial da OWASP Juice Shop; Falsos Positivos (FP), representando alertas incorretos, como casos de código seguro sinalizado como vulnerável e Falsos Negativos (FN), que indicam vulnerabilidades existentes que não foram detectadas. Para a automatização do cálculo das métricas, foi implementada uma função em Python baseada na biblioteca scikit-learn. A função percorre os rótulos verdadeiros (y_true) e as previsões (y_pred) para contar os valores de VP, FP e FN. Em seguida, são calculadas as métricas clássicas de avaliação: precisão, recall, F1-Score e taxa de falsos positivos (FP Rate). O código da função é apresentado no Código 2

Código 2. Código de implementação das métricas de avaliação

```
1
2  from sklearn.metrics import precision_score, recall_score,
       f1_score
3
4
5  def calcular_metricas(y_true, y_pred):
6      # Contadores de desempenho
7      vp = sum(1 for yt, yp in zip(y_true, y_pred) if yt == 1 and
           yp == 1)
8      fp = sum(1 for yt, yp in zip(y_true, y_pred) if yt == 0 and
           yp == 1)
9      fn = sum(1 for yt, yp in zip(y_true, y_pred) if yt == 1 and
           yp == 0)
10
11     # Cálculo das métricas
12     precisao = precision_score(y_true, y_pred, zero_division=0)
13     recall = recall_score(y_true, y_pred, zero_division=0)
14     f1 = f1_score(y_true, y_pred, zero_division=0)
15     fp_rate = fp / (vp + fp) if (vp + fp) > 0 else 0
16
17     return {
18         'Precisao': precisao,
19         'Recall': recall,
20         'F1-Score': f1,
21         'FP Rate': fp_rate,
22         'VP': vp,
23         'FP': fp,
24         'FN': fn
25     }
```

4. Resultados

Os resultados apresentados nesta seção foram obtidos a partir da análise comparativa entre ferramentas SAST (SonarQube, Semgrep) e modelos LLM (DeepSeek, CodeLlama) na detecção de vulnerabilidades no *OWASP Juice Shop*. As métricas foram calculadas utilizando VP, FP, FN, conforme descrito na metodologia. De modo geral, observou-se que cada abordagem apresentou pontos fortes e limitações específicas, tanto em termos de cobertura quanto de precisão, conforme resumido na Tabela 2.

Tabela 2. Comparação de Métricas entre Ferramentas de Análise de Segurança

Ferramenta	Precisão	Recall	F1-Score	FP Rate	VP	FP	FN
Semgrep	100%	50%	67%	0%	5	0	5
SonarQube	100%	10%	18%	0%	1	0	9
DeepSeek	78%	70%	74%	22%	7	2	3
CodeLlama	55%	60%	57%	45%	6	5	4

Nota: VP = Verdadeiros Positivos, FP = Falsos Positivos, FN = Falsos Negativos. Dados obtidos da análise do OWASP Juice Shop v17.3.0.

No que se refere ao desempenho geral, a ferramenta Semgrep (SAST) obteve 100% de precisão e 50% de recall, resultando em um F1-Score de 67%. Isso indica que, embora seja altamente precisa nas detecções que realiza, sua cobertura ainda é limitada. Por outro lado, o SonarQube, amplamente utilizado para análise de qualidade de código, demonstrou baixa capacidade de identificar vulnerabilidades reais, com apenas 10% de recall e um F1-Score de 18%, ainda que mantendo precisão de 100%. Em ambos os casos, a taxa de falsos positivos (FP Rate) foi de 0%, indicando baixo ruído, mas também um risco elevado de vulnerabilidades não detectadas (falsos negativos).

Entre os modelos de linguagem, o DeepSeek se destacou com o melhor equilíbrio entre precisão (78%) e recall (70%), atingindo um F1-Score de 74% e uma FP Rate de 22%. Já o CodeLlama apresentou desempenho inferior, com 55% de precisão, 60% de recall e uma FP Rate de 45%, o que resultou em um F1-Score de 57%. Essa alta taxa de falsos positivos no CodeLlama indica que quase metade dos alertas exigem verificação manual, um custo operacional elevado em ambientes produtivos.

Além das métricas quantitativas, a Tabela 3 apresenta um resumo das principais vulnerabilidades detectadas por cada ferramenta, destacando as diferentes áreas de cobertura. Observa-se que os modelos de linguagem (LLMs) foram capazes de identificar vulnerabilidades que passaram despercebidas pelos SASTs, como NoSQL Injection, CSRF e controles de acesso quebrados. Por outro lado, o Semgrep se mostrou mais eficaz em vulnerabilidades clássicas como XSS e injeções em pontos previsíveis.

Tabela 3. Resumo das vulnerabilidades principais detectadas por cada ferramenta

Ferramenta	Principais Vulnerabilidades Detectadas
Semgrep	XSS, Injection, Sensitive Data Exposure, SSRF
SonarQube	Injection (limitada)
DeepSeek	NoSQL Injection, CSRF, Broken Access Control, Validação insuficiente
CodeLlama	NoSQL Injection, CSRF, Broken Access Control, Validação insuficiente

Complementando os resultados quantitativos, a Tabela 4 apresenta uma análise

qualitativa das ferramentas, considerando aspectos como equilíbrio entre precisão e recall, taxa de falsos positivos e aplicabilidade prática em diferentes contextos. Essa avaliação tem como objetivo oferecer uma visão mais interpretativa sobre os pontos fortes e limitações de cada abordagem.

Tabela 4. Comparação qualitativa das ferramentas

Ferramenta	Avaliação
Semgrep	Alta precisão, baixa cobertura (50% recall), FP Rate de 0%. Ideal para análises rápidas com foco em confiabilidade e baixo ruído.
SonarQube	Alta precisão, porém praticamente sem cobertura (10% de recall). Mais útil para análise de qualidade de código do que para segurança.
DeepSeek	Bom equilíbrio entre precisão e recall, FP Rate moderada (22%). Indicado para uso prático em pipelines automatizados.
CodeLlama	Detecta vulnerabilidades relevantes, mas com alto ruído (FP Rate de 45%). Requer validação manual cuidadosa.

Esses achados destacam o potencial complementar entre abordagens tradicionais e generativas na análise de segurança em JavaScript.

5. Considerações Finais

Os resultados deste estudo exploratório sugerem que a detecção de vulnerabilidades em aplicações JavaScript pode se beneficiar de uma abordagem combinada. Ferramentas SAST como o Semgrep são altamente precisas, mas oferecem cobertura limitada. Já modelos LLM, como o DeepSeek, aumentam significativamente a cobertura, especialmente em vulnerabilidades mais contextuais, embora ainda sofram com taxas consideráveis de falsos positivos.

Observou-se também que soluções amplamente adotadas como o SonarQube, embora úteis para qualidade de código, têm pouca eficácia na identificação de vulnerabilidades reais, reforçando seu papel complementar no pipeline de segurança.

É crucial reconhecer três limitações fundamentais que condicionam a generalização destes resultados: (1) A análise baseou-se em apenas 15 arquivos; (2) Os LLMs foram utilizados sem fine-tuning, explicando parcialmente os altos FP Rates (22-45%) e limitando seu potencial analítico para segurança; (3) A ausência de validação dinâmica das correções impossibilita afirmar que as soluções geradas mitigam riscos sem criar novas cadeias de vulnerabilidades.

Com base nessas restrições, recomenda-se uma arquitetura de segurança que integre: (1) SASTs tradicionais para detecção de padrões conhecidos; (2) LLMs para cobrir casos mais complexos; e (3) filtros automatizados que reduzam o impacto dos falsos positivos.

Como próximas etapas, sugerem-se estudos que explorem o desenvolvimento de técnicas que reduzam o FP Rate dos LLMs sem comprometer sua cobertura, possivelmente através de fine-tuning com datasets especializados ou métodos híbridos que integrem análise estática e dinâmica. Igualmente importante será a criação de métricas que quantifiquem não apenas a eficácia técnica das ferramentas, mas também seu impacto operacional em ambientes reais de desenvolvimento e também uma análise qualitativa

das sugestões de correção geradas pelos LLMs, avaliando sua precisão, aplicabilidade e alinhamento com boas práticas de segurança (ex: se as correções propostas para XSS de fato mitigam o risco sem introduzir novos *vulnerability chains*). Esta evolução será fundamental para que as promessas dos modelos de linguagem se traduzam em ganhos práticos para a segurança de aplicações JavaScript sem sobrecarregar as equipes com falsos positivos.

Referências

Hu, T. et al. (2024). Unveiling llm evaluation focused on metrics: Challenges and solutions. arXiv. Disponível em: <http://arxiv.org/abs/2404.09135>. Acesso em: 05 abr. 2025.

IBM (2024). Análise de código de ia. Disponível em: <https://www.ibm.com/br-pt/topics/ai-code-review>. Acesso em: 05 abr. 2025.

International Organization for Standardization (2023). *ISO/IEC 25010:2023 – Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. ISO, Genebra.

Kiminich, B. (2024). Owasp juice shop: An intentionally insecure javascript web application. Versão 17.3.0. Disponível em: <https://github.com/juice-shop/juice-shop>. Acesso em: 20 fev. 2024.

Le, T. K. et al. (2024). A study of vulnerability repair in javascript programs with large language models. In *Companion Proceedings of the ACM Web Conference 2024*. Disponível em: <http://arxiv.org/abs/2403.13193>. Acesso em: 19 mar. 2025.

OWASP Foundation (2023). Owasp top 10:2023 – the ten most critical web application security risks. Disponível em: <https://owasp.org/www-project-top-ten/>. Acesso em: 20 fev. 2024.

Palo Alto Networks. What is static application security testing (sast)? Disponível em: <https://www.paloaltonetworks.com/cyberpedia/what-is-static-application-security-testing-sast>. Acesso em: 05 abr. 2025.

Snyk (2022). Snyk top 10 – inteligência de vulnerabilidades de segurança. Disponível em: <https://snyk.io/pt-BR/snyk-top-10/>. Acesso em: 20 fev. 2024.

Stack Overflow. Developer survey 2023. Disponível em: <https://survey.stackoverflow.co/2023>. Acesso em: 20 fev. 2025.

W3Techs (2024). Usage statistics of javascript as client-side programming language on websites. Disponível em: <https://w3techs.com/technologies/details/cp-javascript>. Acesso em: 20 fev. 2024.

Wadhams, N. et al. (2024). Barriers to using static application security testing (sast) tools: A literature review. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, pages 161–166, Sacramento, CA, USA. Association for Computing Machinery. Disponível em: <https://dl.acm.org/doi/10.1145/3691621.3694947>. Acesso em: 19 mar. 2025.

Zhou, X. et al. (2024). Comparison of static application security testing tools and large language models for repo-level vulnerability detection. arXiv. Disponível em: <http://arxiv.org/abs/2407.16235>. Acesso em: 15 nov. 2024.