

Análise comparativa de IAs generativas como ferramentas de apoio à programação segura

Eduardo Silva, Emerson Quinaia, Danilo Silva, Alexandre Braga

CPQD - Centro de Pesquisa e Desenvolvimento

eduardoams@outlook.com, emerson.quinaia@gmail.com,
danilosataide@gmail.com, ambraga@cpqd.com.br

Abstract. *Generative Artificial Intelligence has become popular in supporting software development. However, there are questions regarding the effectiveness of this technology in software security. This article investigates the use of programming assistants to support secure programming, weaknesses prevention, and detection of known vulnerabilities. Amazon Code Whisperer, Codium and Microsoft Copilot were evaluated against source code with known vulnerabilities. Despite being competitive in precision and recall, they are not effective due to lack of concise responses and poor usability for prompts.*

Resumo. *A Inteligência Artificial generativa tem se popularizado no apoio ao desenvolvimento de software. Porém, há questionamentos quanto à efetividade dessa tecnologia na segurança de software. Este artigo investiga o uso de assistentes de programação no apoio à programação segura, prevenção de fraquezas e detecção de vulnerabilidades conhecidas. Foram avaliadas Amazon Code Whisperer, Codium e Microsoft Copilot contra códigos com vulnerabilidades conhecidas. Apesar de serem competitivos em precisão e recall, eles não são efetivos por falta de concisão nas respostas e usabilidade ruim na elaboração e engenharia de prompts.*

1. Introdução

As IAs generativas vem sendo cada vez mais utilizadas como assistentes de programação por meio de extensões especializadas. Contudo, o uso dos LLMs (*Large Language Models*) na geração de código seguro traz preocupações. Por exemplo, eles podem aprender informações falsas ou incorretas durante o treinamento e propagá-las em suas respostas, ou ainda gerar textos falsos ou imprecisos que fazem uso de dados verdadeiros, dificultando a identificação de informação inverídica [Kabir et al. 2024].

No contexto de programação e desenvolvimento de software, essas ferramentas possuem boa capacidade de geração de códigos sintaticamente corretos [Borji, A 2023)]. Entretanto, mesmo que as LLMs tenham a tendência de gerar códigos livres de erros de sintaxe, os códigos podem apresentar soluções semanticamente incorretas, como por exemplo falhas no atendimento à lógica de negócios [Kabir et al. 2024].

Este trabalho avalia três assistentes de programação por meio de suas extensões disponíveis nas *IDEs*, como o *Visual Studio Code*: Amazon Code Whisperer, Codium e Microsoft Copilot. Os assistentes são avaliados em relação a sua capacidade de reconhecer vulnerabilidades conhecidas em códigos inseguros, corrigir códigos inseguros e escrever códigos seguros. Resultados preliminares sugerem a ineficácia destes assistentes em tarefas de programação segura, não apenas pelos falsos positivos e falsos negativos, mas pela falta de objetividade e clareza dos positivos verdadeiros.

O texto a seguir está organizado do seguinte modo. A Seção 2 relaciona trabalhos

similares. A Seção 3 explica a metodologia de trabalho. As Seções 4 e 5 apresentam e discutem resultados e a Seção 6 faz considerações finais.

2. Trabalhos Relacionados

[Perry et al. 2023] oferece uma comparação da segurança de códigos escritos com e sem o uso de assistentes de programação de IA. Este trabalho aponta que os participantes tendem a escrever códigos mais inseguros com o uso de IAs e que, além disso, o uso destas causaram uma confiança excessiva na segurança dos códigos. Assim como o trabalho citado, o presente artigo também faz uma avaliação da eficiência das IAs no contexto de segurança cibernética, porém com uma metodologia diferente.

[Kabir et al. 2024] coletou perguntas do *Stack Overflow* que foram em seguida utilizadas como prompts para o ChatGPT. O Artigo comparou as respostas fornecidas por humanos e as respostas fornecidas pela IA, identificando a característica prolixa das respostas e diferentes graus de inconsistência, algo observado no presente trabalho.

A metodologia utilizada no presente trabalho foi inspirada em [Braga et al. 2017] e [Braga et al. 2019], que realizaram uma avaliação prática de ferramentas de análise estática de código-fonte utilizando a técnica de matriz de confusão, resultando num maior entendimento sobre o funcionamento destas ferramentas.

3. Metodologia

A metodologia de trabalho foi composta de duas grandes etapas, detalhadas a seguir: (i) coleta de dados com aplicação de *prompts* e (ii) análise manual das respostas.

3.1. Coleta de dados e aplicação de prompts

Primeiramente, foram coletados exemplos de trechos de códigos com vulnerabilidades conhecidas e as contrapartes sem as respectivas vulnerabilidades, das seguintes bases de conhecimento: *Common Weakness Enumeration (CWE)* com 32 exemplares, *SEI CERT Oracle Coding Standard for Java* com 36 exemplares, e *SEI CERT C Coding Standard* com 148 exemplares. Também foram coletadas as versões corrigidas de cada um dos códigos vulneráveis, quando disponíveis. Em seguida, foram elaborados três modelos de *prompts* para uso em casos de teste de cada um dos trechos de códigos recolhidos.

Tabela 1. Prompts utilizados.

| # | Modelo de prompt |
|---|--|
| 1 | Verifique se o código a seguir apresenta falhas de segurança e vulnerabilidades. |
| 2 | Desenvolva uma solução segura para a vulnerabilidade de segurança apresentada. |
| 3 | Implemente um código de teste para as vulnerabilidades do código. |

A Tabela 1 ilustra os *modelos* de prompts. Por exemplo, o prompt número 1 foi utilizado para testar as versões corrigidas dos códigos, com exceção de 12 dos 22 códigos do CWE, que não possuíam versões com as vulnerabilidades corrigidas. A ausência desses códigos com vulnerabilidades corrigidas não afetou os resultados do trabalho.

Os assistentes de programação foram utilizados por meio das extensões de integração

com o IDE *Visual Studio Code (VScode v1.89-v1.90.2)*. Cada assistente foi utilizado isoladamente, sem a presença dos demais assistentes instalados. Cada trecho de código vulnerável e sem vulnerabilidade foi testado nos seguintes assistentes de programação empoderados por IA generativa: Amazon Code Whisperer (versão gratuita, Amazon Q v1.9.0 2024-06-14, LLM não identificado), Codium (versão gratuita v0.9, LLM GPT 4) e Microsoft Copilot (versão estudantil v1.7, LLM GPT 4-Turbo).

3.2. Análise manual das respostas

A confiabilidade das respostas fornecidas pelas IAs generativas foi validada tomando como referência as documentações disponíveis nos sites da *CWE*, do *SEI CERT Oracle Coding Standard for Java* e do *SEI CERT C Coding Standard*, nas quais a vulnerabilidade de cada um dos trechos de código fornecidos é explicada em detalhes. A equipe de avaliação foi organizada em dois papéis: executor e validador, com o objetivo de identificar e resolver inconsistências, assim como também facilitar a homogeneidade na execução dos testes entre assistentes de programação, apesar das diferenças de usabilidade entre eles.

A avaliação das respostas fornecidas pelas IAs foi realizada pela técnica de análise por matriz de confusão, comumente utilizada na avaliação de classificadores binários, em que cada resposta é classificada em uma das quatro categorias: Verdadeiro Positivo (VP), Verdadeiro Negativo (VN), Falso Positivo (FP) e Falso Negativo (FN), em que:

- Verdadeiro Positivo: IA detecta a vulnerabilidade conhecida no código inseguro;
- Verdadeiro Negativo: IA não detecta a vulnerabilidade conhecida no código seguro;
- Falso Positivo: IA detecta vulnerabilidades que não existem no código seguro;
- Falso Negativo: IA não detecta vulnerabilidades conhecidas no código inseguro.

4. Resultados

A Tabela 2 detalha os resultados comparados dos três assistentes de programação analisados referente às respostas ao prompt 1 da Tabela 1. A tabela mostra o total de testes realizados (incluindo códigos vulneráveis e sem vulnerabilidades), os valores de VP, VN, FP e FN para cada um dos assistentes de programação, assim como também as métricas *Precision*, *Recall* e *F1-Score*, em cada uma das bases de conhecimento de vulnerabilidades (Java, C e CWE). O total de testes nas bases Java e CWE varia entre assistentes por limitações de interface do assistente na execução de alguns testes.

Tabela 2. Métricas dos três assistentes de programação em C, Java e CWE.

| | Codium | | | Code Whisperer | | | Copilot | | |
|------------------|-------------------|------------|------------|----------------|------------|-----------------|------------|------------------|---------------|
| | C | Java | CWE | C | Java | CWE | C | Java | CWE |
| Total | 148 | 36 | 32 | 148 | 36 | 30 | 148 | 36 | 32 |
| PV (%) | 61 (41,2%) | 13 (36,1%) | 19 (58,6%) | 50 (33,8%) | 13 (36,1%) | 16(53,3%) | 58 (39,2%) | 14 (38,9%) | 22 (68,8%) |
| NV (%) | 48 (32,4%) | 7 (19,5%) | 7 (20,7%) | 43 (29,0%) | 10 (27,8%) | 6 (20,0%) | 37 (25,0%) | 12 (33,3%) | 5 (15,6%) |
| FP (%) | 26 (17,5%) | 11 (30,5%) | 3 (10,35%) | 31 (21,0%) | 8 (22,2%) | 3(10,0%) | 37 (25%) | 6 (16,7%) | 5 (15,6%) |
| FN(%) | 13 (8,8%) | 5 (13,9%) | 3 (10,35%) | 24 (16,2%) | 5 (13,9%) | 5 (16,7%) | 16 (10,8%) | 4 (11,1%) | 0 (0%) |
| Precision | 0,736 | 0,556 | 0,813 | 0,628 | 0,639 | 0,773 | 0,642 | 0,722 | 0,844 |
| Recall | 0,824 | 0,722 | 0,864 | 0,676 | 0,722 | 0,762 | 0,784 | 0,778 | 1,000 |
| F1-score | 0,778 | 0,628 | 0,837 | 0,651 | 0,678 | 0,747 | 0,706 | 0,749 | 0,915 |

Inicialmente, os assistentes podem ser comparados por seus falsos positivos e falsos negativos. O assistente Codium apresentou as menores porcentagens de falsos positivos e falsos negativos na linguagem C. Já o assistente Code Whisperer apresentou a menor porcentagem de falsos positivos nos CWEs. O assistente Copilot mostrou as menores porcentagens de falsos positivos e falsos negativos na linguagem Java e não registrou falsos negativos no CWE. Métricas derivadas podem melhorar a análise comparativa.

Também é importante avaliar a precisão ou acurácia das respostas (com a métrica *precision*) e a abrangência dos acertos (com a métrica *recall*) em relação ao total de testes realizados. A métrica *F1-score* é a média harmônica entre as métricas anteriores. O assistente Copilot apresentou as melhores métricas em Java e CWE, seguido pelo Code Whisperer para Java e pelo Codium para C. O assistente Codium mostrou melhores métricas que o Code Whisperer e o Copilot na linguagem C.

A seguir são mostrados exemplos do funcionamento de cada um dos assistentes de programação, com a resposta obtida e o código submetido a fim de ilustrar a maneira como eles respondem de forma incorreta, imprecisa, confusa ou prolixa.

4.1. Caso exemplo 1 - Codium

A Figura 1 a seguir mostra um exemplo de análise do assistente de programação *Codium* sobre um código vulnerável (Figura 2) retirado do *SEI CERT Oracle Coding Standard for Java* (regra *STR02-J. Specify an appropriate locale when comparing locale-dependent data*). O assistente não foi capaz de detectar a vulnerabilidade e, por isso, a resposta foi classificada como falso negativo.

O código Java fornecido contém uma verificação de segurança básica para evitar o processamento de tags específicas, neste caso, a tag "SCRIPT". No entanto, essa abordagem pode ser insuficiente para garantir segurança completa contra ataques como Cross-Site Scripting (XSS), especialmente se o método processTag for parte de um sistema que processa ou renderiza HTML de entrada de usuários.

Aqui estão algumas potenciais falhas e melhorias:

1. **Comparação de Cadeias de Caracteres:** O método utiliza tag.toUpperCase().equals("SCRIPT") para verificar se a tag é igual a "SCRIPT". Isso é eficaz para ignorar a tag "SCRIPT" em qualquer capitalização, mas não impede a inserção de outras tags potencialmente perigosas ou scripts maliciosos contidos em outros elementos.
2. **Filtragem Limitada:** Apenas a tag "SCRIPT" está sendo filtrada. Se o objetivo é prevenir a execução de scripts mal-intencionados, outras tags e atributos HTML que podem ser explorados para injetar scripts também deveriam ser considerados, como "IMG" com eventos JavaScript, "IFRAME", entre outros.
3. **Escapamento de Caracteres:** Não há indicação de que caracteres especiais ou potencialmente perigosos estão sendo escapados. Para segurança adicional, considerar o uso de bibliotecas que oferecem funcionalidades de sanitização e escapamento de HTML.
4. **Uso de Bibliotecas de Sanitização:** Para uma segurança mais robusta, seria apropriado utilizar bibliotecas de sanitização de HTML que seguem as melhores práticas e estão atualizadas com as últimas vulnerabilidades conhecidas.

Para melhorar a segurança deste código, recomenda-se implementar uma lista mais abrangente de tags e atributos a serem filtrados ou usar uma biblioteca de sanitização de confiança. Além disso, sempre que possível, limitar a entrada de usuário a um conjunto seguro de caracteres e realizar uma validação rigorosa de todas as entradas do usuário.

Figura 1. Resposta fornecida pela IA do Codium.

O caso em questão tratava de um código que utiliza a função "toUpperCase()". Quando não se especifica um *Locale* dentro desta função, pode haver efeitos inesperados no código por conta de diferenças de capitalização de caracteres em diferentes localidades no mundo, que podem usar sistemas de escrita diferentes. Por exemplo, No turco, o caractere minúsculo "ı", quando convertido para maiúsculas, se torna "İ". Na Figura 2, o código está convertendo uma label de uma tag para letras maiúsculas (tags no HTML não são case-sensitive). Nessas condições, a tag "script" seria convertida para

```
public static void processTag(String tag) {
    if (tag.toUpperCase().equals("SCRIPT")) {
        return;
    }
    // Process tag
}
```

Figura 2. Código inseguro.

“SCRIPT”, e por conta da diferença na letra i, a tag <SCRIPT> não seria reconhecida. A resposta fornecida pelo assistente *Codium* afirma que o código da Figura 2 é eficaz para ignorar a tag "SCRIPT" em qualquer capitalização, o que não condiz com a realidade.

4.2. Caso exemplo 2 - Code Whisperer

A Figura 3 mostra mais um exemplo de falso negativo, dessa vez fornecido pelo Code Whisperer. A IA assistente afirma não ter encontrado vulnerabilidades, enquanto o código fornecido (Figura 4) contém a mesma falha do exemplo anterior.

Do ponto de vista da segurança, esse código parece relativamente simples e não parece ter nenhuma vulnerabilidade óbvia. No entanto, há algumas coisas a considerar:

Validação de Entrada: Embora o código verifique se a tag é igual a "SCRIPT", ele não realiza nenhuma outra validação de entrada. Dependendo de como esse método é usado, pode ser necessário validar a entrada de forma mais completa para garantir que ela não contenha conteúdo malicioso ou caracteres inesperados.

Sanitização: O código não parece higienizar ou codificar a tag de entrada antes de processá-la. Dependendo de como a tag é usada posteriormente no aplicativo, isso pode levar a problemas como vulnerabilidades de cross-site scripting (XSS) se a tag não for devidamente higienizada.

Contexto de uso: A segurança desse código depende, em última análise, de como ele é usado na aplicação maior. Se a tag for usada de uma forma que possa expor o aplicativo a riscos de segurança, como ser incluída em uma página da Web sem a devida higienização, o código poderá precisar de medidas de segurança adicionais.

[...]

No geral, o código fornecido não parece ter vulnerabilidades de segurança óbvias, mas é importante considerar o contexto mais amplo de como ele é usado no aplicativo.

```
public static void processTag(String tag) {
    if (tag.toUpperCase().equals("SCRIPT")) {
        return;
    }
    // Process tag
}
```

Figura 4. Código inseguro.

Figura 3. Trecho de resposta fornecida pelo Code Whisperer.

4.3. Caso exemplo 3 – Copilot

A Figura 5 contém uma análise fornecida pelo Copilot sobre um código com vulnerabilidades conhecidas (Figura 6), presente na regra “IDS03-J. Do not log unsanitized user input”. O código que registra dados não confiáveis de um usuário não autenticado sem limpeza de dados [SEI 2022]. Esta resposta foi classificada como VP.

O código que você compartilhou parece ser uma parte de um sistema de autenticação. Vamos analisá-lo em busca de possíveis falhas de segurança e vulnerabilidades.

- Logging de Credenciais:** O trecho de código registra informações de login, incluindo o nome de usuário, usando o nível de log "severe". Isso pode ser problemático, pois expõe as credenciais em texto claro nos logs. Se um atacante obtiver acesso aos logs, poderá ver os nomes de usuário e tentar usá-los para fins maliciosos.
- Falta de Tratamento de Erros Adequado:** O código não lida com erros específicos, como falhas na conexão com o banco de dados ou problemas de autenticação. Isso pode levar a vazamentos de informações sensíveis ou a uma experiência ruim para os usuários.
- Não Utilização de Hashing ou Criptografia:** O código não mostra como as senhas são armazenadas ou comparadas. Se as senhas forem armazenadas em texto claro ou com criptografia fraca, isso representa um risco significativo de segurança. O uso de funções de hash (como bcrypt) é recomendado para armazenar senhas com segurança.
- Não Utilização de Parâmetros Preparados:** Se o valor da variável `username` for diretamente inserido na string de log, isso pode abrir a porta para ataques de injeção de SQL. É importante usar parâmetros preparados ou consultas parametrizadas para evitar esse tipo de vulnerabilidade.
- Falta de Autenticação Multifator (MFA):** O código não menciona a implementação de autenticação multifator (MFA). MFA é uma camada adicional de segurança que ajuda a proteger as contas dos usuários, especialmente em sistemas críticos.

```
if (loginSuccessful) {
    logger.severe("User login succeeded for: " + username)
} else {
    logger.severe("User login failed for: " + username);
}
```

Figura 6. Código inseguro.

Figura 5. Resposta fornecida pelo Copilot.

5. Discussão das descobertas

Os resultados ilustram os desafios e limitações do uso de IAs generativas na data em que este artigo foi escrito. Os positivos verdadeiros não devem ser encarados como indicadores da eficácia dessas ferramentas, visto que há ambiguidades e imprecisões nas respostas, mesmo quando as IAs foram capazes de detectar a vulnerabilidade. Este comportamento somado aos falsos positivos e falsos negativos e métricas relativamente baixas (abaixo de 0,9), sugerem que ainda é prematuro delegar, sem supervisão, a segurança de código-fonte aos assistentes de programação.

Por outro lado, as métricas *Precision*, *Recall* e *F1-score* (exceto pelo Codium com Java) estão acima de 0,60, sugerindo que os assistentes são, no mínimo, tão eficazes quanto as ferramentas SAST open source de boa reputação, avaliadas nos trabalhos relacionados. Sendo assim, a dificuldade de uso em programação segura, específica aos assistentes de programação, está na usabilidade por meio de uma IDE, em relação à engenharia de *prompt* e análise dos resultados.

No caso exemplo 1, é possível observar características recorrentes das respostas das LLMs. O assistente de programação gera uma informação falsa com uma explicação que soa verdadeira, mas que é na verdade inverídica (Figura 2), pois afirma que o código é eficaz para ignorar a tag “SCRIPT” em qualquer capitalização. Além disso, observa-se uma tendência da IA em oferecer respostas longas e prolixas.

Já no caso exemplo 2, o assistente Code Whisperer não foi capaz de reconhecer o padrão inseguro de codificação, afirmando que o código não continha vulnerabilidade e oferecendo três sugestões não relevantes para o código fornecido, que é relativamente simples. O caso exemplo 3 apresenta uma resposta classificada como verdadeiro positivo, pois o assistente Copilot foi capaz de indicar a vulnerabilidade do código analisado. Contudo, a IA também forneceu outras possíveis vulnerabilidades que podem ser consideradas como falsos positivos secundários, dada a simplicidade do código vulnerável fornecido à IA. Respostas prolixas são comuns aos 3 assistentes analisados.

Ainda existem desafios de usabilidade inerentes ao uso das interfaces dos assistentes que dificultam a análise de vulnerabilidades e o questionamento sobre a segurança do código-fonte. Em particular, a falta de objetividade nas respostas e a necessidade de refinar e personalizar os *prompts* conforme a interface da IDE, as características da ferramenta e como as duas são integradas, facilitando ou dificultando a engenharia de *prompts*. Além disso, o código gerado pelas ferramentas tende a conter vulnerabilidades e as ferramentas não expressam preocupação especial com a programação segura.

6. Considerações Finais

Embora os assistentes de programação tenham apontado as vulnerabilidades reais, eles também forneceram respostas prolixas, confusas e imprecisas na maioria das vezes. Isso prejudica a usabilidade dessas ferramentas, visto que informação em excesso e mal explicada pode confundir o programador, em vez de esclarecer a questão de segurança levantada. Além disso, se torna difícil identificar quais vulnerabilidades devem ser priorizadas. Assim sendo, a grande quantidade de resultados positivos verdadeiros não deve ser encarada como métrica absoluta da eficácia das ferramentas estudadas.

Observando as métricas, os assistentes gratuitos são competitivos em relação ao Copilot (comercial) e ferramentas SAST open source. Porém, a segurança do código não pode ser delegada para o assistente de programação. Os assistentes de programação ainda têm muito a melhorar para serem efetivos na segurança de software. As grandes incidências de falsos positivos e falsos negativos podem levar tanto à investigação desnecessária de alarmes falsos, no primeiro caso, quanto à falsa sensação de segurança pela omissão de alertas, no segundo caso. Trabalhos futuros podem contemplar metodologias de avaliação e benchmark dos assistentes, avaliação de outros assistentes, engenharia de *prompts* e aumento do conjunto de testes, testes de usabilidade e desenvolvimento de novas interfaces de submissão de *prompts* aos assistentes integrados aos IDEs.

Agradecimentos. Este projeto foi apoiado pelo programa PPI Softex, Acordo de Parceria nº 0200-120/2022, financiado pelo Ministério da Ciência, Tecnologia e Inovações com recursos da Lei nº 8.248, de 23 de outubro de 1991.

Referências

- Amazon Code Whisperer (2024). Amazon Code Whisperer, <https://aws.amazon.com/pt/codewhisperer/>, Janeiro.
- Borji, A. (2023). A categorical archive of ChatGPT failures. ArXiv (Cornell University), 5 Feb. 2023.
- Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2017). Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study. In 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pages 170–181. IEEE.
- Braga, A., Dahab, R., Antunes, N., Laranjeiro, N., and Vieira, M. (2019). Understanding how to use static analysis tools for detecting cryptography misuse in software. 68(4):1384–1403.
- CodiumAI (2024). CodiumAI, <https://codium.ai>, Janeiro.
- CMU. Software Engineering Institute (SEI), CERT Coding Standard for C, <https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>, Abril.
- CMU. Software Engineering Institute (SEI), CERT Oracle Coding Standard for Java, <https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java>, Abril.
- CWE (2023) 2023 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html Março.
- Kabir, S., Udo-Imeh, D. N., Kou, B., and Zhang, T. (2024). Is stack overflow obsolete? an empirical study of the characteristics of ChatGPT answers to stack overflow questions. In Proceedings of the CHI Conference on Human Factors in Computing Systems, pages 1–17.
- Microsoft Copilot (2024). Microsoft Copilot, <https://copilot.microsoft.com/>, Janeiro.
- Perry, N., Srivastava, M., Kumar, D., and Boneh, D. (2022). Do users write more insecure code with AI assistants? Publisher: arXiv Version Number: 3.