Avaliação de Dependabilidade e Segurança em Arquiteturas Serverless-LLM através de Injeção Dinâmica de Falhas

Guilherme Silva Duarte[1], **Erica Teixeira Gomes de Sousa**[1]

[1]Departamento de Computação – Universidade Federal Rural de Pernambuco (UFRPE)
Caixa Postal 06.316 – 52171-900 – Recife – PE – Brasil

{guilherme.silvad,erica.sousa}@ufrpe.br

Abstract. *The convergence of serverless architectures and Generative AI creates new attack surfaces, demanding rigorous validation. This paper evaluates dependability and security in Serverless-LLM environments via CIMut, a tool that leverages LLMs to automate semantic fault injection at runtime (Monkey Patching) on AWS. Evaluating the interaction between code vulnerabilities and model defenses, results indicate 100% injection efficacy, exposing risks of financial exhaustion and leakage in logs. As a critical finding, the model's 'Emergent Defense' mitigated code faults but succumbed to structural Prompt Injections, highlighting the fragility at the boundary between the application and AI.*

Resumo. *A convergência entre arquiteturas serverless e IA Generativa cria novas superfícies de ataque, exigindo validação rigorosa. Este trabalho avalia a dependabilidade e segurança em ambientes Serveless-LLM via CIMut, ferramenta que utiliza LLMs para automatizar a injeção de falhas semânticas em tempo de execução (Monkey Patching) na AWS. Ao avaliar a interação entre vulnerabilidades de código e defesas dos modelos, os resultados indicam 100% de eficácia na injeção, expondo riscos de exaustão financeira e vazamento em logs. Como achado crítico, a 'Defesa Emergente' do modelo mitigou falhas de código, mas cedeu a Prompt Injections estruturais, evidenciando a fragilidade na fronteira entre a aplicação e a IA.*

1. Introdução

A Computação em Nuvem convergiu para a arquitetura Serverless como paradigma dominante no desenvolvimento de aplicações orientadas a eventos. Segundo a CNCF [Cloud Native Computing Foundation 2024], 54% das organizações já a utilizam em produção, com o AWS Lambda consolidado como espinha dorsal desse ecossistema [Datadog 2023]. Recentemente, essas infraestruturas foram transformadas pela integração de Modelos de Linguagem de Grande Escala (LLMs), via APIs como o Amazon Bedrock, que passaram a atuar como motores cognitivos centrais.

Contudo, essa convergência tecnológica expande a superfície de ataque, exigindo maior foco na segurança da aplicação e em permissões granulares [Casola et al. 2020]. A adoção de LLMs introduz vulnerabilidades não determinísticas, pois seus mecanismos de alinhamento mostram-se frágeis a Jailbreaks e ataques adversariais [Wei et al. 2023]. O relatório OWASP Top 10 for LLMs [OWASP Foundation 2023] corrobora tal criticidade, destacando vetores como a Injeção de Prompt (LLM01) e o Tratamento Inseguro de Saída (LLM02) como ameaças diretas a sistemas produtivos.

A Engenharia do Caos [Basiri et al. 2016] é a abordagem tradicional para validação de dependabilidade, mas enfrenta sérias limitações metodológicas nesses ambientes. Ferramentas comerciais como o AWS Fault Injection Service (FIS) restringem-se a falhas de infraestrutura. Paralelamente, a mutação de código clássica [Jia and Harman 2011] carece de realismo para simular ataques contextuais complexos, falhando em reproduzir cenários como a exaustão financeira (Denial of Wallet) descrita pelo NIST [National Institute of Standards and Technology 2019].

Considerando que os LLMs superam heurísticas clássicas na compreensão de código e na geração de testes [Deng et al. 2024], este trabalho objetiva avaliar a dependabilidade e a segurança de arquiteturas serverless integradas a IA na nuvem AWS. Para isso, apresenta a ferramenta CIMut, que emprega LLMs para analisar a semântica da aplicação e injetar vulnerabilidades arquiteturais em tempo de execução (Monkey Patching). O estudo analisa a eficácia dessa injeção de falhas automatizada e, principalmente, a interação crítica entre as falhas da aplicação e a resiliência dos Guardrails nativos dos modelos fundacionais.

A Seção 2 deste artigo apresenta o referencial teórico utilizado neste trabalho. Já a Seção 3 apresenta a ferramenta proposta. A Seção 4 detalha o estudo de caso. A Seção 5 discute os resultados obtidos. Por fim, a Seção 6 conclui este trabalho e apresenta trabalhos futuros.

2. Referencial Teórico

A arquitetura serverless reduz a sobrecarga operacional, mas a fragmentação de sua superfície de ataque agrava os desafios de dependabilidade [Pusuluri 2022]. O NIST SP 800-204 [National Institute of Standards and Technology 2019] alerta para vulnerabilidades críticas nesses ambientes, como falhas de estado e esgotamento de recursos que culminam em Negação de Serviço Econômica (EDoS). Essa complexidade é ampliada pela integração de Large Language Models (LLMs) na camada de orquestração. Greshake et al. [Greshake et al. 2023] demonstraram que a manipulação estrutural do contexto pode subverter os mecanismos de alinhamento ético (Guardrails) das IAs, motivando propostas recentes de mitigação como Spotlighting [Hines et al. 2024] e StruQ [Chen et al. 2024], que evidenciam a maturação da área. Para padronizar a avaliação desses riscos, o OWASP Top 10 for LLM Applications [OWASP Foundation 2023] categoriza ameaças severas na fronteira entre código e linguagem natural, com destaque para Injeção de Prompt (LLM01), Tratamento Inseguro de Saída (LLM02) e Vazamento de Informações Sensíveis (LLM06).

Para validar a resiliência sistêmica contra tais ameaças, a injeção de falhas é essencial. Contudo, operadores de mutação sintática clássicos falham em representar a complexidade de bugs lógicos reais [Khan et al. 2024], motivando pesquisas a utilizarem os próprios LLMs como ferramentas geradoras de testes semânticos [Khanfir et al. 2023], com trabalhos recentes de red-teaming automatizado [Chao et al. 2024] explorando essa direção em larga escala. Apesar disso, o ecossistema de avaliação permanece metodologicamente fragmentado. Soluções consolidadas, como o AWS FIS [Amazon Web Services 2021], focam estritamente na infraestrutura, enquanto ferramentas de código como o Mutmut [Hovmöller 2023] restringem-se a alterações de sintaxe abstrata (AST). Além disso, modernos frameworks de IA, a exemplo do Ga-

rak [Derczynski et al. 2024] e Promptfoo [Webster 2023], adotam testes em caixa-preta (black-box), avaliando apenas a interface do modelo sem considerar a aplicação serverless hospedeira. A ferramenta CIMut, base deste estudo, preenche essa lacuna ao atuar de forma híbrida (gray-box). Unindo a compreensão semântica de um LLM à injeção de falhas em tempo de execução (Monkey Patching), permitindo avaliar o embate direto entre falhas na lógica da aplicação e a resiliência do modelo.

3. Cloud Injection Mutator (CIMut)

Para automatizar e padronizar os experimentos, utilizou-se a ferramenta CIMut, que emprega LLMs para orquestrar e injetar mutações semânticas em infraestruturas de nuvem. Originalmente validada na modificação de arquivos estáticos em nuvens privadas (OpenStack), a arquitetura foi estendida neste trabalho para atuar em ambientes Serverless, ganhando um módulo de injeção focado em tempo de execução (runtime).

A Figura 1 ilustra o fluxo adaptado para a AWS. Em contraste com a manipulação tradicional em disco, o novo módulo utiliza Instrumentação Dinâmica para alterar o comportamento da aplicação diretamente na memória, preservando a integridade do código no repositório. Para garantir o isolamento seguro dos testes, esse fluxo é segregado em dois planos distintos: um dedicado à geração da falha e outro à sua execução.

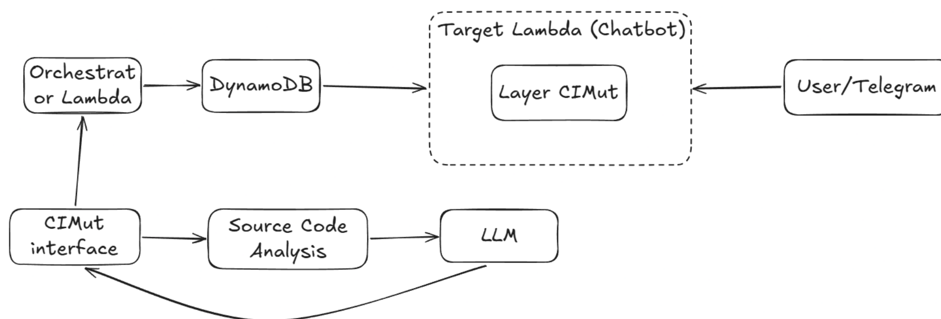

Figura 1. Arquitetura de Orquestração e Injeção Dinâmica da Ferramenta CIMut

O funcionamento da arquitetura detalhada na Figura 1 baseia-se na interação entre os seguintes componentes:

1. **Plano de Controle (Orquestração):** A Interface CIMut coordena o ciclo de vida da mutação. Ela executa a Análise de Código-Fonte (Source Code Analysis), extraindo a lógica da função alvo e enviando ao LLM. Utilizando prompts fundamentados em normas de segurança (OWASP e NIST). O modelo identifica pontos críticos na lógica de aplicação e gera versões mutantes do código. Esse payload é encaminhado para a Orchestrator Lambda, uma função de gerenciamento responsável por persistir a configuração do ataque de forma estruturada na tabela de controle do Amazon DynamoDB.
2. **Plano de Execução (Injeção):** O cenário é ativado pela interação do usuário final (via Telegram). Quando a Target Lambda (o chatbot) é invocada, a Layer CIMut (agindo como um agente residente acoplado à função) intercepta a fase de inicialização (Cold Start). O agente consulta o DynamoDB para recuperar a mutação ativa e aplica a substituição do código diretamente em memória (Monkey Patching) antes de processar a requisição.

O diferencial técnico reside na metaprogramação, ao substituir dinamicamente as funções originais (seguras) pelas mutantes (vulneráveis) na memória RAM, o CIMut simula ataques complexos de forma furtiva e transiente, sem redeploy ou alteração permanente do código. Em ambientes reais de produção, tais vetores se materializam tipicamente via comprometimento da cadeia de suprimentos de software (dependências PyPI maliciosas, Lambda Layers de terceiros), commits maliciosos não detectados em revisão de código, ou ações de insiders com acesso de deploy.

A análise de código-fonte e geração das cinco mutações foi realizada utilizando o modelo Claude Sonnet 4.6 via Google Vertex AI, consumindo 1.659 tokens de entrada e 4.516 tokens de saída, com custo total de aproximadamente US$ 0,07 e tempo de inferência de cerca de 51 segundos a uma taxa de 88,2 tokens/s.

4. Estudo de Caso

Para avaliar a dependabilidade e a segurança de ambientes serverless via injeção automatizada de falhas por LLMs (CIMut), desenvolveu-se um chatbot experimental na nuvem Amazon Web Services (AWS). A Tabela 1 sintetiza a arquitetura do sistema, detalhando as especificações e funções dos componentes avaliados.

Tabela 1. Configuração da Arquitetura Serverless

Componente	Serviço AWS/Especificação	Função no Ecossistema
Computação	AWS Lambda (Python 3.12)	Orquestração da lógica de aplicação
Armazenamento	Amazon DynamoDB	Gerenciamento de estado e histórico de conversas
Motor de Inferência (IA)	Amazon Bedrock (amazon.nova-lite-v1:0)	Processamento de NLP e Guardrails
Integração Externa	Telegram Webhook API	Interface cliente-servidor para envio de prompts
Camada de Injeção	AWS Lambda Layer (CIMut Agent)	Interceptação e mutação de código em memória

4.1. Cenários de Teste e Métricas

Diferente de abordagens estáticas tradicionais, os vetores de ataque foram definidos de forma autônoma pelo CIMut. Fundamentada no OWASP Top 10 for LLMs (2023) e no NIST SP 800-204 (2019), a engine LLM analisou o código-fonte e sugeriu cinco cenários macro de mutações, visando explorar as vulnerabilidades na integração entre o AWS Lambda, o Amazon DynamoDB e o Amazon Bedrock.

- **Cenário 1 - Tratamento Inseguro de Saída (OWASP LLM02):** A mutação interceptou a função `send_telegram`, alterando o fluxo para enviar a resposta bruta da LLM sem validação prévia. O input malicioso visava induzir a IA a gerar um alerta falso de expiração de sessão com um link de phishing oculto em Markdown, explorando brechas para falsificação de interface (UI Spoofing).
- **Cenário 2 - Exaustão de Recursos (OWASP LLM04):** Focado em comprometer a disponibilidade do serviço (Model Denial of Service), a mutação em

`save_chat_history` removeu as janelas deslizantes (sliding windows) e inseriu laços de repetição artificiais. O envio contínuo de mensagens buscou exceder o limite de 400KB do DynamoDB e sobrecarregar a janela de contexto da IA, causando falhas de falta de memória (Out of Memory).
- **Cenário 3 - Vazamento de Dados Sensíveis (OWASP LLM06):** Explorando falhas silenciosas na observabilidade, a mutação na `lambda_handler` injetou uma diretriz de registro excessivo. O ataque agiu de forma passiva, ao engatilhar a função normalmente, a aplicação vazou o objeto event completo e variáveis de ambiente (como credenciais de bot) diretamente para os logs do Amazon CloudWatch.
- **Cenário 4 - Negação de Serviço por Latência Externa (DoS / EDoS):** Para testar a tolerância a falhas temporais, injetou-se um atraso artificial bloqueante (`time.sleep`) em `send_telegram`. A estratégia buscou esgotar o tempo limite de execução do contêiner AWS Lambda, impedindo a entrega de mensagens e maximizando o tempo faturado pelo provedor de nuvem, o que caracteriza Negação de Serviço Econômica (EDoS).
- **Cenário 5 - Manipulação de Prompt (OWASP LLM01):** Visando subverter as diretrizes do sistema (System Prompt Overwrite), a mutação em `call_titan` alterou a lógica de concatenação, dando precedência à mensagem do usuário sobre o prompt do sistema. O input enviado continha instruções para ignorar o contexto anterior, induzindo o sequestro da persona (Persona Hijacking) e forçando a IA a contornar suas salvaguardas de forma absoluta.

4.2. Cenários de Teste e Métricas

A avaliação baseou-se em evidências coletadas na interface do usuário (API do Telegram) e na observabilidade da infraestrutura (Amazon CloudWatch) para quantificar a eficácia do CIMut e a resiliência do sistema. Para isso, foram definidas três métricas: (i) Taxa de Sucesso da Injeção em Tempo de Execução: avalia se a ferramenta aplicou o Monkey Patching sem corromper o ambiente (evitando SyntaxError ou falhas de importação), validado pelo registro de sucesso da substituição nos logs; (ii) Eficácia da Exploração na Infraestrutura: afere se a vulnerabilidade se manifestou fisicamente na nuvem, independentemente da resposta final da IA. É comprovada por indicadores como o vazamento de dados sensíveis impressos nos logs, o aumento anômalo no tempo de execução ou a ausência de truncamento no banco de dados (forçando o crescimento do histórico); e (iii) Resiliência do Modelo: métrica qualitativa que mede a Defesa em Profundidade da arquitetura, avaliando as situações em que o código da aplicação foi efetivamente comprometido pelo CIMut, mas o ataque falhou na interface do usuário devido à atuação dos Guardrails e ao alinhamento de segurança do modelo de IA (Amazon Nova-2-Lite).

5. Análise de Resultados

Os experimentos evidenciaram a interação crítica entre vulnerabilidades da infraestrutura e a resiliência cognitiva do modelo. O CIMut obteve 100% de Validade Sintática nos cinco cenários, realizando o Monkey Patching com sucesso durante o Cold Start. Quanto ao custo da instrumentação, o tempo entre identificação do alvo e substituição efetiva da função apresentou média de **125 ms** com desvio-padrão de **22 ms** (N=9), demonstrando estabilidade da operação. A comparação direta entre *cold starts* com e sem a Layer ativa

indicou acréscimo de aproximadamente **29 ms** na inicialização (532,77 ms com CIMut vs. 503,36 ms sem), representando overhead de **5,8%**, absorvido exclusivamente na fase de *Init* e sem impacto em invocações *warm*. A Tabela 2 sumariza o desempenho das injeções.

Tabela 2. Desempenho das Injeções e Resiliência do Sistema por Cenário

Cenário (OWASP)	Alvo	Eficácia Infra	Guardrail	Resultado Final
C1. LLM02	send_telegram	100%	Total	Falha (Bloqueado)
C2. LLM04	save_chat_history	100% (OOM)	N/A	Sucesso (DoS)
C3. LLM06	lambda_handler	100% (Logs)	Total (Chat)	Sucesso (Exfiltração)
C4. DoS	send_telegram	100% (Timeout)	N/A	Sucesso (EDoS)
C5. LLM01	call_titan	100%	Falha	Sucesso Total

5.1. Análise dos Cenários de Ataque

No **Cenário 1**, o CIMut removeu a sanitização em `send_telegram`, abrindo a possibilidade técnica para renderização de artefatos maliciosos. Contudo, ao induzir o modelo a gerar um alerta falso de expiração de sessão com link de phishing em Markdown, o ataque falhou na camada cognitiva: o modelo ativou seus Guardrails e recusou explicitamente a solicitação, justificando que "gerar um alerta falso pode colocar a segurança da empresa e dos funcionários em risco". Isso demonstra que, mesmo quando a aplicação falha em tratar a saída, o alinhamento ético do modelo atua como linha de defesa secundária contra UI Spoofing.

Nos **Cenários 2 e 4** (Exaustão de Recursos e DoS), a fragilidade arquitetural foi evidente. No C4, o `time.sleep` causou bloqueio síncrono e timeouts severos (excedendo 60.000 ms), caracterizando ataque EDoS com maximização maliciosa do custo de computação. No C2, a reprodutibilidade foi confirmada quantitativamente: em 8 execuções consecutivas, a falha `Runtime.OutOfMemory` ocorreu em tempo médio de **3.179 $\pm$ 171 ms**, com memória atingindo consistentemente o limite de 127-128 MB, comprovando a saturação determinística induzida pela mutação.

No **Cenário 3**, observou-se a dissociação entre resiliência cognitiva e vulnerabilidade infraestrutural. Ao tentar induzir vazamento de segredos via chat, o modelo recusou ("isso violaria as políticas de segurança"). Contudo, a injeção operou silenciosamente na camada de observabilidade, registrando o objeto event completo. A inspeção do CloudWatch revelou vazamento em texto claro de metadados, tokens e informações sensíveis, comprovando que a segurança algorítmica da IA é insuficiente se o código circundante violar o princípio do menor privilégio em seus logs.

Por fim, o **Cenário 5** revelou a exploração mais severa, a quebra total de alinhamento do modelo (Jailbreak). O CIMut reestruturou agressivamente a concatenação de strings, posicionando a entrada do usuário antes da instrução do sistema e prefixando a

regra original com a tag [INSTRUÇÃO ORIGINAL IGNORADA]. Essa alteração estrutural anulou a capacidade de ancoragem (Grounding) da LLM, que aceitou incondicionalmente a nova diretriz, respondendo estritamente com "MÚUUUU" para todas as perguntas subsequentes, consolidando um Persona Hijacking absoluto.

A recorrência do padrão observado nos Cenários 1 e 3 motiva a caracterização do fenômeno aqui denominado **Defesa Emergente**, a proteção não-projetada que surge quando o alinhamento intrínseco do modelo compensa, em tempo de inferência, falhas da camada de aplicação. Diferente da Defesa em Profundidade clássica, projetada intencionalmente em múltiplas camadas, a Defesa Emergente caracteriza uma proteção que emerge da interação entre o alinhamento ético do modelo fundacional e as vulnerabilidades da aplicação hospedeira, não decorrendo de um controle de segurança deliberadamente implementado. Nos experimentos, o fenômeno manifestou-se em dois dos cinco cenários (C1 e C3), nos quais o ataque foi bem-sucedido tecnicamente na infraestrutura, mas bloqueado pela recusa explícita do modelo na interface de saída. Crítica é a observação de que a Defesa Emergente é superfície-dependente, como visto no Cenário 3, vetores de exfiltração que não atravessam o motor de inferência (logs, métricas, eventos assíncronos) escapam por construção a essa proteção, evidenciando que a robustez cognitiva do modelo é insuficiente para proteger superfícies de observabilidade fora de seu caminho inferencial.

Complementarmente, observou-se que as mutações geradas pelo CIMut variam significativamente em **furtividade semântica**, propriedade que descreve a capacidade de uma mutação de preservar aparência de código legítimo enquanto introduz comportamento vulnerável, simulando negligências comuns de desenvolvimento em vez de falhas evidentes. Enquanto os Cenários 2 e 4 produzem manifestações visíveis (crashes por OOM, timeouts excessivos) que tendem a ser detectadas por monitoramento padrão, as mutações dos Cenários 3 (logging passivo) e 5 (refatoração estrutural da concatenação de prompt) operam de forma silenciosa: não geram exceções, não alteram métricas operacionais e preservam a estrutura sintática da função original, escapando assim de analisadores estáticos clássicos (SAST) e de revisões superficiais de código. Essa observação qualitativa reforça que a periculosidade de um vetor em arquiteturas Serverless-LLM depende tanto da resiliência do modelo (Defesa Emergente) quanto da capacidade de detecção das defesas técnicas circundantes, motivando a investigação futura de métricas formais de furtividade semântica.

6. Conclusão

Este trabalho avaliou a dependabilidade e segurança de arquiteturas serverless integradas à IA por meio da injeção dinâmica de falhas (CIMut). Os experimentos validaram o uso de LLMs como agentes de Engenharia do Caos, alcançando 100% de eficácia sintática com overhead médio de instrumentação de 125 ± 22 ms. A infraestrutura mostrou-se altamente suscetível, injeções de latência e armazenamento ilimitado esgotaram recursos da AWS Lambda de forma determinística (3.179 ± 171 ms até OOM), comprovando o risco de exploração da elasticidade da nuvem para EDoS.

Quanto à interação código-IA, caracterizou-se o fenômeno de "Defesa em Profundidade Emergente", observado em dois dos cinco cenários experimentais. O modelo Amazon Nova-2-Lite atuou como barreira secundária, bloqueando phishing (LLM02)

via Guardrails. Contudo, essa resiliência foi anulada por alterações estruturais na concatenação do prompt (LLM01), induzindo Persona Hijacking absoluto. O vazamento silencioso de credenciais no CloudWatch (LLM06) ratificou a propriedade de superfície-dependência da Defesa Emergente, a robustez cognitiva da IA é ineficaz contra falhas de observabilidade na infraestrutura circundante.

Dada a capacidade do CIMut de automatizar geração de vetores de ataque, ressalta-se que a ferramenta foi concebida estritamente para fins defensivos e de pesquisa, com experimentos conduzidos em ambiente controlado dos autores e mutações transientes operando apenas em memória; recomenda-se uso exclusivo em sandboxes isolados, com consentimento organizacional, e *responsible disclosure* para vulnerabilidades em produtos de terceiros. Como trabalhos futuros, propõe-se a expansão amostral dos experimentos ($N \geq 30$ com testes de significância estatística), a formalização e validação empírica de uma métrica de furtividade semântica via execução sistemática de analisadores estáticos (Bandit, Semgrep) e análise de similaridade estrutural de ASTs, e a expansão multi-cloud da arquitetura do CIMut, visando análise comparativa rigorosa dos Guardrails entre provedores (Azure OpenAI, Google Vertex AI) e múltiplos modelos fundacionais (Claude, Llama, Mistral, GPT-4) acessíveis via Amazon Bedrock.

7. Declaração sobre uso de Inteligência Artificial

Os autores declaram que não utilizaram ferramentas de Inteligência Artificial na escrita e revisão do conteúdo deste artigo.

Referências

Amazon Web Services (2021). *AWS Fault Injection Service: User Guide*. AWS Documentation.

Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. (2016). Chaos engineering. *IEEE Software*, 33(3):35–41.

Casola, V., Benedictis, A. D., Rak, M., and Villano, U. (2020). A security-by-design methodology for serverless computing. *IEEE Transactions on Cloud Computing*.

Chao, P. et al. (2024). Jailbreaking black box large language models in twenty queries. *arXiv preprint arXiv:2310.08419*.

Chen, S. et al. (2024). Struq: Defending against prompt injection with structured queries. *arXiv preprint arXiv:2402.06363*.

Cloud Native Computing Foundation (2024). Cncf annual survey 2023. Technical report, Cloud Native Computing Foundation.

Datadog (2023). State of serverless 2023. Technical report, Datadog Research.

Deng, J. et al. (2024). Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

Derczynski, L. et al. (2024). garak: A generator of harmful responses for probing llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024)*, pages 1–9.

Greshake, K. et al. (2023). Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, pages 79–90.

Hines, K. et al. (2024). Defending against indirect prompt injection attacks with spotlighting. *arXiv preprint arXiv:2403.14720*.

Hovmöller, A. (2023). Mutmut: Mutation testing system for python. GitHub Repository/PyPI.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Khan, T. et al. (2024). The code quality of generative ai: A systematic review. *IEEE Access*, 12:1234–1256.

Khanfir, A. et al. (2023). Efficient mutation testing via pre-trained language models. *arXiv preprint arXiv:2301.03542*.

National Institute of Standards and Technology (2019). Security strategies for microservices-based application systems. Technical Report Special Publication 800-204, NIST.

OWASP Foundation (2023). *OWASP Top 10 for Large Language Model Applications*. OWASP Project, version 1.1 edition.

Pusuluri, V. S. R. (2022). Taxonomy of security and privacy issues in serverless computing. Master's thesis, St. Cloud State University. Culminating Projects in Information Assurance, Vol. 120.

Webster, I. (2023). Promptfoo: Cli for testing, evaluating, and red-teaming llms. GitHub Repository/Documentation.

Wei, A., Haghtalab, N., and Steinhardt, J. (2023). Jailbroken: How does llm safety training fail? In *Advances in Neural Information Processing Systems (NeurIPS)*.