An Analysis of Software Vulnerabilities and Weaknesses in Machine Learning Libraries

Felipe Cordeiro Alves Dias, Daniel Cordeiro

[1]Escola de Artes, Ciências e Humanidades — Universidade de São Paulo (USP)
São Paulo — SP — Brasil

{felipecavazotto, daniel.cordeiro}@usp.br

Abstract. Research Context: *This article investigates software vulnerabilities and weaknesses across 273 machine learning (ML) library repositories.* ***Scientific and/or Practical Problem****: The scientific problem lies in comprehensively mapping security issues within the rapidly expanding landscape of ML libraries. Practically, this research addresses the need for a scalable methodology to effectively track, collect, and correlate existing vulnerabilities and weaknesses in these libraries.* ***Proposed Solution and/or Analysis:*** *We propose a scalable methodology designed to track, collect, and correlate security issues by leveraging CVEs (Common Vulnerabilities and Exposures) and CWEs (Common Weakness Enumeration).* ***Related IS Theory:*** *This research draws upon Complexity Theory, recognizing the intricate interconnectedness of ML ecosystems, where a single change or vulnerability can trigger cascading and unpredictable effects throughout the entire system.* ***Research Method:*** *Our research employed an empirical software analysis approach, involving the mining of 273 machine learning library repositories. We utilized CodeQL and Dependabot as part of the methodology to systematically track, collect, and conduct an in-depth analysis of the interconnections between CWEs and CVEs related to identified vulnerabilities and weaknesses.* ***Summary of Results:*** *Our findings revealed various security flaws, including validation failures, access control issues, memory management errors, development flaws, path-traversal vulnerabilities, and cryptographic weaknesses. Notably, these included CWEs from the Top 25 Most Dangerous Software Weaknesses. These results underscore the critical need for proactive measures to enhance security and reliability within ML systems.* ***Contributions and Impact to IS area:*** *Contributions include an automated methodology for characterizing vulnerabilities and weaknesses in ML libraries; an analysis of vulnerabilities and weaknesses in 273 ML libraries, and a dataset of CVEs/CWEs interconnections for ML.*

1. Introdução

A adoção de projetos de código aberto exige análise criteriosa de fatores como o tamanho e o engajamento da comunidade de mantenedores, a maturidade do projeto, o nível de acoplamento e, crucialmente, a segurança do software. Com custos de manutenção que podem atingir 70% do ciclo de vida de um projeto [Pressman 2019] e 76% dos ataques de ransomware em 2022, explorando vulnerabilidades conhecidas entre 2010 e 2019 [Kujavski et al. 2024], é imperativo considerar fatores de risco como os mencionados, analisando criticamente a adoção de bibliotecas externas, principalmente as que podem ser facilmente evitadas por fornecerem funcionalidades básicas, como conversão de caixa de texto, remoção de espaços em branco numa *string*, entre outras. Manter a segurança

de um Ecossistema Digital que dependa de projetos de código aberto é um problema de pesquisa ainda em aberto e alinhado ao desafio "Sistemas de Informação e Desafios do Mundo Aberto" dos Grandes Desafios de Pesquisa em SI no Brasil (GranDSI-BR 2016-2026 [Clodis Boscarioli 2017]).

Mesmo desconsiderando os custos de manutenção, a utilização de bibliotecas com implementações triviais em contextos em que a segurança é crítica permanece uma prática controversa. Como demonstrado por Abdalkareem et al. 2017 e de acordo com a Teoria da Complexidade em Sistemas de Informação, essa abordagem representa um risco sistêmico, em que soluções aparentemente inocentes podem comprometer toda a infraestrutura de segurança.

Um caso emblemático que ilustra essa vulnerabilidade foi o incidente com o pacote npm "left-pad" [Wikipedia 2025]. Esse módulo JavaScript, que implementava uma função trivial de preenchimento de strings, tornou-se uma dependência crítica para grandes projetos como Babel e React. Quando removido do repositório do npm [NPM, Inc. 2025], desencadeou uma falha em cascata que afetou sistemas de empresas como Facebook, Netflix e Airbnb. Este episódio evidencia os riscos ocultos em dependências aparentemente benignas. Em contrapartida, bibliotecas de ML, embora mais complexas e ricas em funcionalidades (oferecendo desde modelos pré-treinados até algoritmos especializados), apresentam seus próprios desafios de segurança, que vão desde a classificação da severidade até a remediação de vulnerabilidades [Ribeiro et al. 2024].

O ecossistema de ML vive um momento de adoção acelerada [McKinsey & Company 2022]. Novas aplicações surgem diariamente em diversos domínios, muitas vezes desenvolvidas em ciclos extremamente curtos que frequentemente negligenciam aspectos de segurança. Paradoxalmente, enquanto essas soluções são desenvolvidas, as próprias bibliotecas de ML que as sustentam contêm vulnerabilidades conhecidas [Harzevili et al. 2023], criando um risco latente.

Entre as vulnerabilidades mais críticas, destaca-se o caso Log4Shell (CVE-2021-44228) [Common Enumeration of Vulnerabilities 2021], uma falha de execução remota de código (RCE) na biblioteca de gerenciamento de logs Log4J, amplamente usada por projetos codificados em Java, incluindo bibliotecas de ML. Esse incidente não apenas evidenciou a necessidade de correções imediatas, mas também revelou os desafios inerentes à manutenção de dependências. Atualizar uma única biblioteca pode desencadear um verdadeiro "inferno de dependências" (*dependency hell*) [Abate et al. 2020], onde modificações aparentemente simples introduzem bugs, problemas de compatibilidade ou até mesmo alterações comportamentais inesperadas.

O caso Log4Shell demonstrou de forma contundente como vulnerabilidades em bibliotecas amplamente adotadas podem impactar globalmente diversos sistemas. Analogamente, bibliotecas fundamentais para ML, como pandas, NumPy e scikit-learn, quando comprometidas, podem afetar milhares de aplicações. Embora estudos como Harzevili et al. 2023 tenham avançado na caracterização dessas vulnerabilidades, a comunidade ainda carece de metodologias escaláveis para a análise abrangente de vulnerabilidades em bibliotecas de ML e suas dependências — uma lacuna crítica, dada a crescente dependência dessas tecnologias e a necessidade urgente de correções de segurança.

Essa lacuna na literatura, que os trabalhos existentes não resolvem de forma sa-

tisfatória, reside na ausência de uma abordagem escalável e sistemática capaz de analisar problemas de segurança em um grande volume de repositórios de ML, sem depender de análises manuais ou de ferramentas proprietárias. Para isso, propomos uma metodologia que combina: 1. Análise automatizada de 273 repositórios de ML no GitHub, abrangendo desde a visão computacional até a IA generativa. 2. Uso integrado do *Dependabot* [GitHub 2025] e do *CodeQL* [GitHub 2025] para varredura de dependências e de código. 3. Padronização com o uso dos *frameworks* CVE (Common Vulnerabilities and Exposures) e CWE (Common Weakness Enumeration) para classificação de vulnerabilidades e falhas, respectivamente.

Nossas contribuições são: 1. Uma metodologia automatizada para caracterização de vulnerabilidades e falhas em bibliotecas de ML. 2. Análise de vulnerabilidades e falhas em 273 bibliotecas de ML. 3. Um *dataset* de interconexões de CVEs/CWEs para ML. Por fim, esse estudo alinha-se ao Desafio 2 do GranDSI-BR 2016-2026, já mencionado no início do texto, e ao Desafio 3: Complexidade dos Sistemas de Informação, considerando as interconexões entre sistemas, bibliotecas de ML, vulnerabilidades e fraquezas.

2. Fundamentação Teórica

Esta seção apresenta os conceitos relacionados ao *framework* CWE para classificação de falhas de segurança de software, detalhando sua estrutura hierárquica. Em seguida, descreve o framework CVE.

2.1. Common Weakness Enumeration (CWE)

A CWE [MITRE Corporation 2025b] é um dicionário de falhas de segurança de software desenvolvido pela comunidade e mantido pela MITRE Corporation[1]. Ele fornece um *framework* padronizado para classificar e identificar vulnerabilidades em sistemas computacionais, permitindo abordar proativamente possíveis falhas de segurança. A CWE também é utilizada para avaliar a eficácia de ferramentas e serviços de segurança projetados para detectar falhas [Wu et al. 2015].

Embora a CWE ofereça uma taxonomia abrangente de falhas de software, navegar por suas intrincadas interdependências representa um desafio significativo para quem desenvolve, apesar da importância de compreender seu potencial impacto na segurança do software [Wu et al. 2015]. Essa complexidade decorre das relações inerentes entre os diversos níveis de abstração de falhas, que incluem: *Pilares (P)*, *Classes (CL)*, *Bases (B)* e *Variantes (VA)*, em que os pilares representam o maior nível de abstração e as variantes, o menor. Além disso, existem possíveis agrupamentos como *Visões (VI)* e *Categorias (CA)*. Os detalhes dessas hierarquias e agrupamentos podem ser encontrados nas tabelas 1 e 2.

Além disso, a CWE é complementada por um conjunto de recursos interconectados, incluindo o *Common Weakness Scoring System* (CWSS), o CVE e o *Common Attack Pattern Enumeration and Classification* (CAPEC). Esses recursos, amplamente utilizados por organizações como o Departamento de Defesa (DoD) dos Estados Unidos, constituem um *framework* abrangente para identificar, priorizar e mitigar vulnerabilidades de software [Wu et al. 2015].

[1] `https://cwe.mitre.org/`

Tabela 1. Hierarquia da CWE

Nível CWE	Descrição
1. Pilar	Um Pilar CWE representa o tipo mais abstrato de falha e constitui um tema unificador para todas as Classes/Bases/Variantes relacionadas.
2. Classe	Uma Classe CWE agrupa falhas com características semelhantes.
3. Base	As Bases CWE representam falhas de segurança fundamentais ou erros de projeto que podem levar a vulnerabilidades em sistemas de software.
4. Variante	As Variantes CWE são entradas especializadas que fornecem contexto adicional sobre manifestações específicas de uma falha base.

Tabela 2. Agrupamentos da CWE

Agrupamento	Descrição
Categoria	As Categorias CWE fornecem uma estrutura organizacional mais ampla e um entendimento contextual das falhas. Podem ser vistas como coleções temáticas de CWE. As categorias não são falhas em si, servindo como agrupamentos organizacionais informais.
Visão	Uma Visão CWE oferece uma perspectiva personalizada sobre as falhas da CWE.

2.2. Common Vulnerabilities and Exposures (CVE)

O *framework* CVE [MITRE Corporation 2025a] é um dicionário padronizado de vulnerabilidades de segurança divulgadas publicamente. Ele atribui identificadores únicos (CVEs) a vulnerabilidades, permitindo o compartilhamento de informações entre profissionais de segurança e pesquisadores. As entradas CVE facilitam a identificação de vulnerabilidades específicas e fornecem acesso a estratégias de correção e recursos disponíveis para mitigação [Wu et al. 2015].

Embora o CWE e o CVE sejam frequentemente confundidos, uma distinção fundamental reside em seus respectivos focos. Enquanto o CWE fornece uma taxonomia dos tipos de falhas subjacentes, permitindo a compreensão das causas raiz das vulnerabilidades, o CVE serve como um *framework* para identificar instâncias específicas de vulnerabilidades.

3. Trabalhos Relacionados

O gerenciamento de riscos em software de código aberto envolve desde revisões manuais de código até métodos automatizados, como a modelagem preditiva e a detecção de vulnerabilidades. Harzevili et al. 2023 analisaram vulnerabilidades em bibliotecas populares de ML, como TensorFlow, PyTorch e scikit-learn, examinando tipos, causas, sintomas e padrões de correção. Dois especialistas avaliaram manualmente 683 vulnerabilidades únicas em *commits* de sete bibliotecas. Em comparação, nosso estudo identificou automaticamente 4.799 vulnerabilidades únicas e 606 falhas relacionadas em 273 repositórios de ML, incluindo dependências, ampliando o escopo da análise.

Prana et al. 2021 realizaram um estudo empírico em 450 projetos de código aberto (Java, Python, Ruby) utilizando o Veracode Software Composition Analysis (SCA). Sua pesquisa analisou os tipos, a distribuição, a gravidade e a persistência de vulnerabilidades nesses projetos. Embora nosso estudo tenha focado apenas em bibliotecas desenvolvidas em Python, apresentamos uma metodologia reprodutível que não requer a aquisição de software proprietário como o Veracode. Márquez et al. 2024 propuseram um *framework* para extrair grafos de dependência de bibliotecas e utilizar *Satisfiability Modulo Theories*

(SMT) para analisar o impacto de vulnerabilidades. Nosso trabalho difere em sua abordagem ao revelar o espaço de interconexões de falhas (CWEs) com base em vulnerabilidades (CVEs), analisando a base de código e suas dependências.

Lu et al. 2024 introduziram o GRACE, uma abordagem que integra informações estruturais de grafos e aprendizado em contexto por meio de modelos de linguagem, para melhorar a precisão e a eficiência na detecção de vulnerabilidades em código-fonte. Esse trabalho difere do nosso, pois se concentra apenas em vulnerabilidades. Além disso, os autores não distinguem formalmente os conceitos de vulnerabilidades e falhas, distinção central em nossa abordagem. Lai et al. 2024 analisaram vulnerabilidades em sistemas de *deep learning* (DL) (TensorFlow, Caffe, OpenCV, Keras e PyTorch), categorizando-as com base em seus vetores de ataque e em impactos potenciais. Esse estudo analisou cinco bibliotecas no contexto de DL, enquanto nosso estudo considerou bibliotecas de ML utilizadas em diversos contextos, dentro dos 273 repositórios.

Reconhecendo o papel crucial das revisões manuais de código na identificação de vulnerabilidades de segurança, Charoenwet et al. 2024 conduziram um estudo empírico para investigar a eficácia de revisões seguras de código. Sua pesquisa envolveu a análise de um grande conjunto de dados de revisões de código, bem como a identificação de padrões e fatores que contribuem para localizar vulnerabilidades. Esse trabalho analisou dois projetos de código aberto, PHP e OpenSSL, identificando falhas no processo de revisão de código e seus potenciais impactos usando uma abordagem semiautomatizada. Em contraste, nosso estudo analisou automaticamente 273 projetos no contexto de bibliotecas de ML.

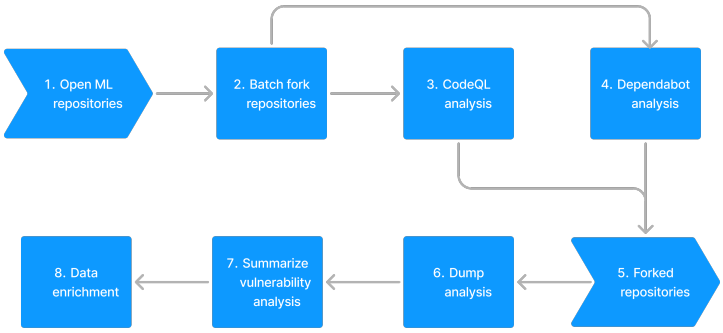
Tabela 3. Análise Comparativa entre Trabalhos Relacionados e o Presente Artigo

Característica	Harzevili et al. (2023)	Prana et al. (2021)	Marquez et al. (2024)	Lu et al. (2024)	Lai et al. (2024)	Charoenwet et al. (2024)	Este Artigo
Principal objetivo	Vulnerabilidades em bibliotecas ML populares	Tipos, distribuição, gravidade e persistência de vulnerabilidades em projetos de código aberto	Análise de impacto de vulnerabilidades usando grafos de dependência e SMT	Detecção de vulnerabilidades em código-fonte usando grafos e modelos de linguagem	Vulnerabilidades em sistemas de Deep Learning (DL)	Eficácia de revisões seguras de código	Análise de vulnerabilidades e falhas de segurança em bibliotecas de ML e suas interconexões CWE-CVE
Metodologia	Avaliação manual de 683 vulnerabilidades únicas em *commits* de 7 bibliotecas	Estudo empírico em 450 projetos (Java, Python, Ruby) usando Veracode SCA	*Framework* para extrair grafos de dependência e análise de impacto com SMT	Abordagem GRACE: integra informações estruturais de grafos e aprendizado em contexto	Análise e categorização de vulnerabilidades com base em vetores de ataque e impactos potenciais	Estudo empírico de revisões de código em 2 projetos (PHP, OpenSSL) com abordagem semiautomatizada	Metodologia automatizada combinando: (1) *forks* de 273 repositórios ML no GitHub; (2) uso integrado de Dependabot e CodeQL; (3) padronização com CVE e CWE
Escopo	7 bibliotecas de ML	450 projetos de código aberto (Java, Python, Ruby)	Não especificado o número de bibliotecas/projetos	Não especificado o número de bibliotecas/projetos	5 bibliotecas de DL (TensorFlow, Caffe, OpenCV, Keras, PyTorch)	2 projetos de código aberto (PHP, OpenSSL)	273 repositórios de bibliotecas de ML (incluindo dependências)
Resultados e Contribuições	Caracterização de vulnerabilidades	Análise de tipos, distribuição, gravidade e persistência de vulnerabilidades	*Framework* para análise de impacto	Melhoria na precisão e eficiência da detecção de vulnerabilidades	Categorização de vulnerabilidades em sistemas de DL	Identificação de padrões e fatores que contribuem para localizar vulnerabilidades em revisões de código	606 falhas no código-fonte e 4.789 vulnerabilidades em dependências; metodologia automatizada; *dataset* de interconexões CVEs/CWEs para ML
Diferenças em relação a este artigo	Análise manual e escopo limitado	Foco em diferentes linguagens e uso de software proprietário	Foco em grafos de dependência e SMT, não explora interconexões de falhas (CWEs) com base em vulnerabilidades (CVEs)	Foca apenas em vulnerabilidades, não distingue entre vulnerabilidades e falhas	Foco exclusivo em DL e número limitado de bibliotecas	Análise semiautomatizada e foco em revisões de código	Metodologia escalável e reprodutível, análise abrangente de 273 repositórios, revelando interconexões de CWEs e CVEs

4. Metodologia

Nossa metodologia automatizada para detecção de vulnerabilidades e falhas de segurança em bibliotecas de ML (Algoritmo 1) compreende, de acordo com a Figura 1: (1 e 2) criação de *forks* (bifurcações) de repositórios-alvo, (3 e 4) análise estática automatizada, e (5, 6, 7 e 8) processamento sistemático dos resultados. A abordagem considera as relações hierárquicas entre CVEs e CWEs (Pilares, Classes, Bases, Variantes, Categorias e Visões), superando métodos de classificação manuais, como os de Harzevili et al. 2023, em termos de escalabilidade para a identificação de riscos.

Figura 1. Metodologia automatizada para detecção de vulnerabilidades e falhas de segurança em bibliotecas de ML



Algorithm 1 Algoritmo para análise de repositórios de bibliotecas de ML

Recebe: Uma lista de repositórios de ML, R
Devolve: Relatórios de CVEs e CWEs para cada repositório em R

para cada $r \in R \leftarrow$
    faça fork (bifurque) r usando a API do GitHub
    Adicione arquivos de configuração do CodeQL e Dependabot ao r bifurcado
    Execute análise CodeQL no r bifurcado
    Execute análise Dependabot no r bifurcado
    Extraia resultados da análise usando a API do GitHub e armazene em JSON
    para cada CVE, c, identificado em $r \leftarrow$
        faça $CWEs \leftarrow obtenhaCWEsRelacionados(c)$
        para cada CWE, w, em $CWEs \leftarrow$
            faça $relacionamentos \leftarrow extraiaRelacionamentosDoMITRE(w)$
            $membros \leftarrow extraiaMembrosDoMITRE(w)$
            Armazene $relacionamentos$ e $membros$ associados a w
        fim do para
    fim do para
    Resuma e analise os resultados
fim do para

4.1. Conjunto de Dados

A etapa inicial envolve a compilação de uma lista abrangente de bibliotecas de *machine learning*. Isso pode ser alcançado buscando bibliotecas no GitHub ou outras plataformas de hospedagem de código ou consultando um diretório de bibliotecas como o PyPI para bibliotecas Python. Neste estudo, utilizamos bibliotecas populares de *machine learning* que foram compiladas em uma lista pública chamada "Awesome Machine Learning"[2] (revisão 94f765d), que obteve 66,5k estrelas, 3,3k observadores e 14,7k *forks* no GitHub até janeiro de 2025. Além disso, excluímos projetos que não estavam disponíveis como bibliotecas ou *frameworks*. Utilizando esta abordagem, analisamos um total de 273 repositórios. Os conjuntos de dados, incluindo as entradas do CodeQL e do Dependabot, com seus respectivos metadados, estão disponíveis no GitHub[3].

4.2. Processo de bifurcação

Após compilar a lista de bibliotecas de ML, a segunda etapa envolve a criação automática de *forks* (bifurcações) de todos os repositórios utilizando a API do GitHub. Nestes repositórios bifurcados, o processo adiciona arquivos YAML com os parâmetros necessários para executar as análises do CodeQL e do Dependabot. É importante destacar que a análise do Dependabot identifica vulnerabilidades e falhas associadas, enquanto a análise do CodeQL foca exclusivamente em falhas no código-fonte. Uma vulnerabilidade identificada pelo Dependabot pode já estar mapeada a uma falha detectada pelo CodeQL, mas isso nem sempre ocorre. Os mapeamentos entre CWE e CVE são definidos pela Equação 1.

$$\mathrm{M} : \mathrm{CVE} \rightarrow \mathrm{CWE}^{+} \tag{1}$$

Na Equação (1), CVE representa o conjunto de todos os identificadores CVE (ex: CVE-2023-1234), CWE representa o conjunto de todos os identificadores CWE (ex: CWE-79, CWE-119) e CWE^{+} representa o conjunto potência de CWE, que é o conjunto de todos os subconjuntos possíveis de CWE. Isso permite mapeamentos em que uma única vulnerabilidade (CVE) está relacionada a múltiplas CWEs. M representa a função de mapeamento que recebe uma CVE como entrada e retorna um subconjunto de CWEs.

A análise com *fork* revelou vulnerabilidades críticas não divulgadas publicamente. No caso do LiteLLM [Bertram 2025], identificamos 207 alertas de segurança[4]. Para o FastAPI [Tiangolo 2023], apenas uma vulnerabilidade (CVE-2021-32677, gravidade média, CWE-352) consta no repositório oficial[5] (notavelmente em aberto desde 2021). A gravidade média da CVE desse repositório está de acordo com o CVSS 2.0, que está obsoleto. Atualmente, é classificada como de nível alto, de acordo com o CVSS 3.1. Essa divergência ocorre porque o proprietário do repositório publicou, em 2021, o relatório que contém essa vulnerabilidade (publicado como `GHSA-8h2j-cgx8-6xv7` em 9 de

[2] `https://github.com/josephmisiti/awesome-machine-learning?tab=readme-ov-file#python`

[3] `https://github.com/fcas/fork-attack/tree/main/data/`

[4] `https://github.com/fcas/fork-attack/blob/main/data/2024-12-28/litellm_dependabot_alerts.json` e `https://github.com/fcas/fork-attack/blob/main/data/2024-12-28/litellm_code_analysis.json`

[5] `https://github.com/fastapi/fastapi/security/advisories/GHSA-8h2j-cgx8-6xv7`

junho de 2021), ano em que a CVSS 2.0 ainda era usada. A CVSS vigente no ano de escrita desse artigo (2025) é a 4.0, a qual não contém ainda informação sobre essa vulnerabilidade. A nossa análise do mesmo repositório detectou cinco alertas adicionais: (1) CVE-2024-47874 (alta gravidade, CWE-770), resultante da análise do Dependabot[6]; (2) CWE-20 (alta gravidade); (3) CWE-1004, CWE-1275 e CWE-614 (gravidade média, duas ocorrências) e (4) CWE-79 e CWE-116 (gravidade média), resultantes das análises do CodeQL[7].

Embora a criação de *forks* para análise de vulnerabilidades possa sugerir uma estratégia ofensiva, este estudo tem propósito exclusivamente defensivo, buscando avaliar a confiabilidade de bibliotecas de *machine learning*. A abordagem permitiu analisar com sucesso todos os 273 repositórios-alvo, ressaltando-se que a desativação da funcionalidade de *fork* por parte das pessoas proprietárias dos repositórios exigiria métodos alternativos, menos automatizáveis, para a obtenção do código. Os resultados visam conscientizar a comunidade sobre potenciais riscos à segurança.

4.3. Ferramentas de Análise Estática de Código

A análise estática de código é essencial para identificar vulnerabilidades e garantir a qualidade do software nas fases iniciais de desenvolvimento. Neste estudo, utilizamos duas ferramentas principais: CodeQL e Dependabot, selecionadas por serem as soluções oficiais do GitHub e amplamente adotadas pela comunidade. Embora existam alternativas como Semgrep [Semgrep Contributors 2025], Code Climate Quality [Code Climate 2025] e SonarQube [SonarSource 2025], estas não foram consideradas por serem comerciais.

Análise com CodeQL: O CodeQL foi empregado para análise detalhada de vulnerabilidades no código-fonte, fornecendo informações precisas como: 1. o *commit* específico que introduziu cada vulnerabilidade; 2. status de correção; 3. arquivo afetado; 4. *tags* CWE relacionadas; 5. nível de severidade; 6. localização exata no código.

O CodeQL estrutura o código-fonte dos repositórios em um banco de dados relacional, permitindo a execução de consultas (*queries*) para identificar vulnerabilidades e erros no código. Essa abordagem trata o código como dados, facilitando a análise e a busca por problemas de segurança com maior confiança do que as ferramentas de análise estática tradicionais. A abrangência das CWEs cobertas pelo CodeQL pode ser consultada em sua *documentação oficial*[8]. É importante salientar que o objetivo deste trabalho não é analisar a validade das queries usadas pelo CodeQL.

Análise com Dependabot: O Dependabot foi utilizado para a análise de dependências, com foco no ecossistema pip, realizando: 1. varredura do grafo de dependências; 2. identificação de vulnerabilidades conhecidas; 3. recomendações de atualizações; 4. mapeamento para CVEs e CWEs correspondentes.

O Dependabot utiliza o *GitHub Advisory Database*[9] como sua principal fonte para identificar vulnerabilidades. Este banco de dados contém vulnerabilidades validadas e, em

[6]https://github.com/fcas/fork-attack/blob/main/data/2024-12-28/fastapi_dependabot_alerts.json

[7]https://github.com/fcas/fork-attack/blob/main/data/2024-12-28/fastapi_code_analysis.json

[8]https://codeql.github.com/codeql-query-help/full-cwe/

[9]https://github.com/advisories

setembro de 2025, continha 24.030 vulnerabilidades confirmadas, além de 271.398 não validadas. Para os propósitos deste artigo, focamos no mapeamento dessas vulnerabilidades, e a validação individual delas não foi o foco de nossa pesquisa.

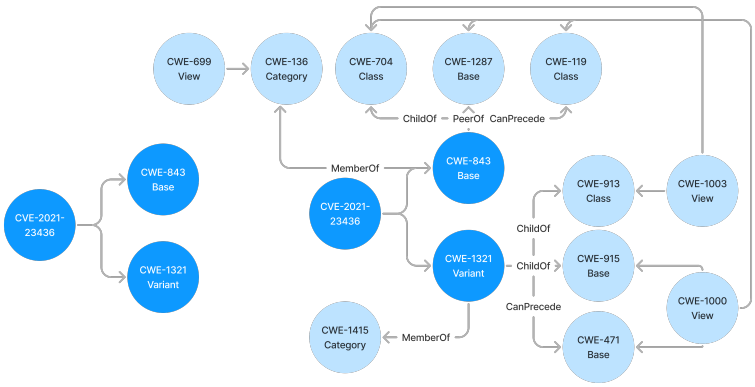
As configurações do CodeQL[10] e do Dependabot[11] estão disponíveis nos artefatos do projeto. Todos os resultados foram armazenados nos repositórios bifurcados. Essa abordagem supera, em termos de escalabilidade, métodos semiautomatizados baseados em expressões regulares e revisões manuais [Harzevili et al. 2023].

4.4. Revelando as Interconexões de CWE

Posteriormente, os resultados da análise de segurança gerados pelo CodeQL e pelo Dependabot são extraídos utilizando a API do GitHub e armazenados em arquivos JSON para processamento adicional. Na sétima etapa, um resumo abrangente dos achados da análise de segurança é gerado. Isso envolve a criação de tabelas que enumeram os identificadores de vulnerabilidades e de falhas identificadas, juntamente com seus respectivos níveis de severidade e de impacto potencial. Esta etapa facilita a identificação de interconexões entre CWEs ao utilizar o conjunto de dados do MITRE.

Finalmente, a partir das saídas do CodeQL e do Dependabot, criamos as conexões iniciais entre CVEs e CWEs. Para obter *insights* mais profundos, expandimos essas relações, extraindo dados detalhados de CWE (incluindo Pilar, Categoria, Classe, Base e Variante) do site do MITRE para cada CWE ID identificado. Isso revelou uma camada previamente inexplorada de interconexões entre CWEs, promovendo uma compreensão mais rica das falhas identificadas, conforme demonstrado na Figura 2 onde uma vulnerabilidade do LiteLLM é analisada, sendo o grafo desconexo à esquerda composto por relações CVE-CWE inexploradas e o da direita contendo as relações expandidas.

Figura 2. Interconexões {CVE-2021-23436} → {CWE-843, CWE-1321} (LiteLLM)



5. Resultados

Nossa análise com CodeQL e Dependabot identificou 606 falhas no código-fonte e 4.789 vulnerabilidades em dependências (Tabela 4). A diferença nos resultados deve-se à abran-

[10] https://github.com/fcas/fork-attack/blob/main/.github/workflows/codeql.yml

[11] https://github.com/fcas/fork-attack/blob/main/.github/dependabot.yml

Tabela 4. Severidade das vulnerabilidades e falhas de segurança das bibliotecas de aprendizado de máquina, análises de CodeQL e Dependabot

Severidade	CodeQL (quantidade de CWE)	Dependabot (quantidade de CVE)
Crítica	16	273
Alta	336	1.704
Média	254	2.248
Baixa	0	565
Total	606	4.789

Tabela 5. Pilares CWE

CWE ID	Descrição	Contagem
682	Incorrect Calculation	1.697
707	Improper Neutralization	602
664	Improper Control of a Resource Through its Lifetime	409
710	Improper Adherence to Coding Standards	372
693	Protection Mechanism Failure	100
691	Insufficient Control Flow Management	84
703	Improper Check or Handling of Exceptional Conditions	72
284	Improper Access Control	45
697	Incorrect Comparison	2
Total		3.383

gência das ferramentas: enquanto o Dependabot analisa todo o grafo de dependências, o CodeQL analisa apenas o código-base do projeto.

Com o uso de identificadores CWE (CodeQL) e CVE (Dependabot), mapeamos as vulnerabilidades das bibliotecas de ML e as falhas associadas, categorizando-as em 9 Pilares, 256 Categorias, 82 Classes, 301 Bases, 114 Variantes e 11 Visões. As seções a seguir detalharão os repositórios e as bibliotecas associados a esses CWEs.

5.1. Pilares CWE

A Tabela 5 apresenta a frequência de falhas associadas a cada identificador de Pilar. Um total de 44 bibliotecas únicas e 74 repositórios GitHub foram identificados como afetados por um ou mais Pilares. O Pilar CWE-682 (Incorrect Calculation) foi o mais frequente, com 1.697 ocorrências, o que sugere uma prevalência alarmante de erros de cálculo que podem comprometer decisões críticas de segurança ou gerenciamento de recursos em bibliotecas de ML. Isso pode levar a ataques de negação de serviço, consumo excessivo de recursos ou até mesmo execução de código não autorizada. O segundo pilar mais comum, CWE-707 (Improper Neutralization), com 602 ocorrências, indica falhas na validação de dados, o que abre portas para ataques de injeção. Bibliotecas como TensorFlow, aim, mlflow e transformers, com múltiplas associações com Pilares, demonstram uma suscetibilidade mais ampla a diversas classes de falhas de segurança. **Os repositórios mais afetados por Pilares CWE são**: 1. aim; 2. compreface; 3. cornac; 4. evidently; 5. gradio; 6. igel; 7. mlflow; 8. neuroner; 9. nn_builder; 10. rasa; 11. ray; 12. transformers.

5.2. Categorias CWE

Com base nas 25 Categorias CWE predominantes (de um total de 256) apresentadas na Tabela 6, identificamos 86 bibliotecas e 100 repositórios distintos afetados. As categorias mais proeminentes, “CERT C++ Secure Coding Section 08 - Memory Management (MEM)” (CWE-876) e “CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)” (CWE-742), com 1.504 e 1.430 ocorrências, respectivamente, sublinham a criticidade dos problemas de gerenciamento de memória. A alta incidência de CWEs relacionados a falhas de validação de entrada/saída (CWE-1019 e CWE-1005) sugere que as bibliotecas de ML são particularmente vulneráveis a ataques de injeção e

de manipulação de dados. A presença de categorias como "OWASP Top Ten 2021 Category A03:2021 - Injection" (CWE-1347) e "OWASP Top Ten 2021 Category A01:2021 - Broken Access Control" (CWE-1345) é um alerta claro sobre a necessidade de práticas de codificação segura e de validação rigorosa de entradas. **As bibliotecas com múltiplas associações com Categorias CWE são**: 1. apache-airflow; 2. aim; 3. flask; 4. gitpython; 5. gradio; 6. ipython; 7. jinja2; 8. jupyter-server; 9. jupyterlab; 10. langchain; 11. litellm; 12. llama-index; 13. lmdb; 14. lxml; 15. mlflow; 16. nltk; 17. notebook; 18. numpy; 19. oauthlib; 20. pillow; 21. pip; 22. pymysql; 23. pyyaml; 24. ray; 25. readthedocs--sphinx-search; 26. scikit-learn; 27. sqlalchemy; 28. tensorflow; 29. torch; 30. tornado; 31. transformers; 32. urllib3; 33. werkzeug. **Os repositórios mais afetados por Categorias CWE são**: 1. aim; 2. compreface; 3. cornac; 4. evidently; 5. gradio; 6. igel; 7. mlflow; 8. neuroner; 9. nn_builder; 10. rasa; 11. ray; 12. transformers.

Tabela 6. As 25 (de 256) categorias CWE mais predominantes

CWE ID	Descrição	Contagem
876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	1.504
742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	1.430
1019	Validate Inputs	1.320
1399	Comprehensive Categorization: Memory Safety	1.122
738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	1.070
872	CERT C++ Secure Coding Section 04 - Integers (INT)	1.070
1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1.031
722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	904
730	OWASP Top Ten 2004 Category A9 - Denial of Service	885
1366	ICS Communications: Frail Security in Protocols	882
1157	SEI CERT C Coding Standard - Guidelines 03. Expressions (EXP)	871
751	2009 Top 25 - Insecure Interaction Between Components	858
1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	843
1218	Memory Buffer Errors	833
1308	CISQ Quality Measures - Security	828
1005	7PK - Input Validation and Representation	805
994	SFP Secondary Cluster: Tainted Input to Variable	769
802	2010 Top 25 - Risky Resource Management	726
1416	Comprehensive Categorization: Resource Lifecycle Management	723
963	SFP Secondary Cluster: Exposed Data	692
883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	683
1131	CISQ Quality Measures (2016) - Security	638
747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	637
1161	SEI CERT C Coding Standard - Guidelines 07. Characters and Strings (STR)	637
1417	Comprehensive Categorization: Sensitive Information Exposure	627
Total		22.388 [1]

[1] `https://github.com/fcas/fork-attack/blob/main/data/all_definitions_agg_libs.csv` - 58.210 Categorias CWEs encontradas, sendo 256 únicas.

5.3. Classes CWE

Uma análise das 25 Classes CWE predominantes (de um total de 85), conforme detalhado na Tabela 7, revelou 85 bibliotecas e 114 repositórios distintos afetados. A Classe CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), com

4.612 ocorrências, é a mais frequente, indicando que problemas de buffer overflow são uma preocupação generalizada. Falhas de “Injection” (CWE-74) e de “Improper Input Validation” (CWE-20) também são altamente prevalentes, reforçando a necessidade de validação robusta de entradas. A presença de classes relacionadas a “Exposure of Sensitive Information” (CWE-200) e “Missing Encryption of Sensitive Data” (CWE-311) destaca a importância da proteção de dados confidenciais em sistemas de ML. **As bibliotecas com múltiplas associações com Classes CWE são**: 1. apache-airflow 2. aim; 3. flask; 4. gitpython; 5. gradio; 6. ipython; 7. jinja2; 8. jupyter-server; 9. langchain; 10. litellm; 11. lxml; 12. mlflow; 13. nltk; 14. notebook; 15. numpy; 16. oauthlib; 17. pillow; 18. pip; 19. pymysql; 20. pyyaml; 21. ray; 22. scikit-learn; 23. sqlalchemy; 24. tensorflow; 25. torch; 26. tornado; 27. transformers; 28. urllib3; 29. werkzeug. **Os repositórios mais afetados por Classes CWE são**: 1. aim; 2. blaze; 3. catboost; 4. colossalai; 5. compreface; 6. cornac; 7. cortex; 8. danswerai; 9. determined; 10. digits; 11. evidently; 12. feast; 13. gradio; 14. igel; 15. lightly; 16. litellm; 17. mindsdb; 18. mlflow; 19. neuroner; 20. nn_builder; 21. nupic.studio; 22. open-webui; 23. pytorch-lightning; 24. rasa; 25. ray.

Tabela 7. As 25 (de 85) Classes CWE mais predominantes

CWE ID	Descrição	Contagem
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.612
74	Improper Neutralization of Special Elements in Output Used by a Downstream Component (“Injection”)	1.266
670	Always-Incorrect Control Flow Implementation	988
20	Improper Input Validation	941
754	Improper Check for Unusual or Exceptional Conditions	835
345	Insufficient Verification of Data Authenticity	686
114	Process Control	599
200	Exposure of Sensitive Information to an Unauthorized Actor	594
913	Improper Control of Dynamically-Managed Code Resources	531
362	Concurrent Execution using Shared Resource with Improper Synchronization (“Race Condition”)	482
706	Use of Incorrectly-Resolved Name or Reference	431
668	Exposure of Resource to Wrong Sphere	399
311	Missing Encryption of Sensitive Data	384
672	Operation on a Resource after Expiration or Release	303
405	Asymmetric Resource Consumption (Amplification)	258
922	Insecure Storage of Sensitive Information	213
407	Inefficient Algorithmic Complexity	210
172	Encoding Error	199
665	Improper Initialization	160
610	Externally Controlled Reference to a Resource in Another Sphere	156
346	Origin Validation Error	155
704	Incorrect Type Conversion or Cast	154
400	Uncontrolled Resource Consumption	148
77	Improper Neutralization of Special Elements used in a Command (“Command Injection”)	135
1039	Automated Recognition Mechanism with Inadequate Detection or Handling of Adversarial Input Perturbations	109
Total		14.948 [1]

[1] `https://github.com/fcas/fork-attack/blob/main/data/all_definitions_agg_libs.csv` - 16.812 Classes CWE encontradas, sendo 85 únicas.

5.4. Bases CWE

Uma análise das 25 Bases CWE preponderantes (de um total de 301), conforme detalhado na Tabela 8, revelou que 28 bibliotecas e 52 repositórios distintos foram afetados. As Bases CWE-1284 (Improper Validation of Specified Quantity in Input) e CWE-770 (Allocation of Resources Without Limits or Throttling) foram as mais frequentes, com 1.110 e 1.080 ocorrências, respectivamente. Isso aponta para falhas comuns na validação de entradas numéricas e no gerenciamento de recursos, que podem levar a ataques de negação de serviço ou estouro de buffer. A alta incidência de "Path Traversal" (CWE-22) e de suas variantes (CWE-23, CWE-36) destaca o risco de manipulação de caminhos de arquivos, permitindo o acesso não autorizado a recursos do sistema. **As bibliotecas com múltiplas associações com Bases CWE são**: 1. aiohttp; 2. apache-airflow; 3. flask; 4. gitpython; 5. gradio; 6. ipython; 7. litellm; 8. mlflow; 9. oauthlib; 10. pip; 11. pyyaml; 12. tensorflow; 13. werkzeug. **Os repositórios mais afetados por Bases CWE são**: 1. aim; 2. auto_viml; 3. autoviz; 4. blaze; 5. catboost; 6. cleanlab; 7. cltk; 8. colossalai; 9. compreface; 10. cornac; 11. danswerai; 12. digits; 13. evidently; 14. gradio; 15. igel; 16. mindsdb; 17. minigrad; 18. mlflow; 19. nalp; 20. neuroner; 21. nn_builder; 22. numpy; 23. pattern; 24. petrel; 25. pix2pix-keras; 26. pytensor; 27. pytorch; 28. pytorch-lightning; 29. pytorchcv; 30. rasa; 31. ray; 32. retro; 33. skbel; 34. spammy; 35. stellargraph; 36. vizdoom; 37. zipline.

5.5. Variantes CWE

A análise das 25 Variantes CWE preponderantes (de um total de 114), detalhada na Tabela 9, revelou que 28 bibliotecas e 52 repositórios distintos foram afetados. A Variante CWE-113 (Improper Neutralization of CRLF Sequences in HTTP Headers), com 684 ocorrências, é a mais comum, indicando vulnerabilidades em aplicações web que interagem com ML. As múltiplas ocorrências de CWEs relacionadas ao framework Struts (CWE-102 a CWE-110) sugerem que bibliotecas de ML que utilizam ou interagem com sistemas baseados em Struts podem herdar essas vulnerabilidades. Variantes de buffer overflow (CWE-121, CWE-122) e "Use After Free" (CWE-416) também são preocupantes, apontando para falhas de segurança de memória de baixo nível que podem ser exploradas para execução arbitrária de código. **As bibliotecas com múltiplas associações com Variantes CWE são**: 1. aiohttp; 2. apache-airflow; 3. flask; 4. gitpython; 5. gradio; 6. ipython; 7. litellm; 8. mlflow; 9. oauthlib; 10. pip; 11. pyyaml; 12. tensorflow; 13. werkzeug. **Os repositórios mais afetados por Variantes CWE são**: 1. aim; 2. auto_viml; 3. autoviz; 4. blaze; 5. catboost; 6. cleanlab; 7. cltk; 8. colossalai; 9. compreface; 10. cornac; 11. danswerai; 12. digits; 13. evidently; 14. gradio; 15. igel; 16. mindsdb; 17. minigrad; 18. mlflow; 19. nalp; 20. neuroner; 21. nn_builder; 22. numpy; 23. pattern; 24. petrel; 25. pix2pix-keras; 26. pytensor; 27. pytorch; 28. pytorch-lightning; 29. pytorchcv; 30. rasa; 31. ray; 32. retro; 33. skbel; 34. spammy; 35. stellargraph; 36. vizdoom; 37. zipline.

5.6. Visões CWE

A Tabela 10 apresenta a frequência de vulnerabilidades associadas a cada identificador de Visão CWE. A análise revelou 59 bibliotecas únicas e 74 repositórios GitHub afetados por uma ou mais Visões CWE. A Visão CWE-884 (CWE Cross-section), com 3.242 ocorrências, é a mais comum, seguida por "Weaknesses in the 2024 CWE Top 25 Most

Tabela 8. As 25 (de 301) Bases CWE mais predominantes

CWE ID	Descrição	Contagem
1284	Improper Validation of Specified Quantity in Input	1.110
770	Allocation of Resources Without Limits or Throttling	1.080
22	Improper Limitation of a Pathname to a Restricted Directory (“Path Traversal”)	907
825	Expired Pointer Dereference	824
120	Buffer Copy without Checking Size of Input (“Classic Buffer Overflow”)	788
73	External Control of File Name or Path	763
824	Access of Uninitialized Pointer	760
822	Untrusted Pointer Dereference	723
823	Use of Out-of-range Pointer Offset	723
170	Improper Null Termination	675
190	Integer Overflow or Wraparound	620
23	Relative Path Traversal	616
41	Improper Resolution of Path Equivalence	599
1287	Improper Validation of Specified Type of Input	597
466	Return of Pointer Value Outside of Expected Range	590
134	Use of Externally-Controlled Format String	581
36	Absolute Path Traversal	579
117	Improper Output Neutralization for Logs	573
470	Use of Externally-Controlled Input to Select Classes or Code (“Unsafe Reflection”)	558
112	Missing XML Validation	557
179	Incorrect Behavior Order: Early Validation	555
15	External Control of System or Configuration Setting	554
1173	Improper Use of Validation Framework	553
1285	Improper Validation of Specified Index, Position, or Offset in Input	553
1286	Improper Validation of Syntactic Correctness of Input	553
Total		16.991 [1]

[1] `https://github.com/fcas/fork-attack/blob/main/data/all_definitions_agg_libs.csv` - 34.871 Bases CWE encontradas, sendo 301 únicas.

Dangerous Software Weaknesses” (CWE-1430), com 3.089 ocorrências. A presença consistente das Visões “Top 25 Most Dangerous Software Weaknesses” (CWE-1430, CWE-1200, CWE-1350, CWE-1387, CWE-1337, CWE-1425) é um indicativo alarmante de falhas de segurança substanciais e de alto impacto que exigem atenção imediata. Isso reforça a necessidade de uma abordagem proativa para a segurança do desenvolvimento de software em ML, priorizando a mitigação dessas fraquezas críticas. **As bibliotecas com múltiplas associações com Visões CWE são**: 1. aim; 2. aiohttp; 3. apache-airflow; 4. flask; 5. gitpython; 6. gradio; 7. ipython; 8. litellm; 9. mlflow; 10. tensorflow. **Os repositórios mais afetados por Visões CWE são**: 1. aim; 2. blaze; 3. catboost; 4. cleanlab; 5. cltk; 6. colossalai; 7. compreface; 8. cornac; 9. cortex; 10. couler.

5.7. Discussão

Nossa análise revelou um amplo espectro de vulnerabilidades e falhas de segurança em bibliotecas de aprendizado de máquina, identificando 606 falhas no código-fonte (via CodeQL) e 4.789 vulnerabilidades em dependências (via Dependabot). Esta disparidade nos números reflete a abrangência distinta das ferramentas: o Dependabot examina o grafo completo de dependências, enquanto o CodeQL foca no código-base do projeto,

Tabela 9. As 25 (de 114) Variantes CWE mais predominantes

CWE ID	Descrição	Contagem
113	Improper Neutralization of CRLF Sequences in HTTP Headers (“HTTP Request/Response Splitting”)	684
785	Use of Path Manipulation Function without Maximum-sized Buffer	678
129	Improper Validation of Array Index	594
102	Struts: Duplicate Validation Forms	553
103	Struts: Incomplete validate() Method Definition	553
104	Struts: Form Bean Does Not Extend Validation Class	553
105	Struts: Form Field Without Validator	553
106	Struts: Plug-in Framework not in Use	553
107	Struts: Unused Validation Form	553
108	Struts: Unvalidated Action Form	553
109	Struts: Validator Turned Off	553
110	Struts: Validator Without Form Field	553
111	Direct Use of Unsafe JNI	553
622	Improper Validation of Function Hook Arguments	553
789	Memory Allocation with Excessive Size Value	490
126	Buffer Over-read	365
127	Buffer Under-read	365
121	Stack-based Buffer Overflow	247
122	Heap-based Buffer Overflow	247
416	Use After Free	191
456	Missing Initialization of a Variable	168
313	Cleartext Storage in a File or on Disk	157
314	Cleartext Storage in the Registry	157
315	Cleartext Storage of Sensitive Information in a Cookie	157
316	Cleartext Storage of Sensitive Information in Memory	157
Total		10.740 [1]

[1] `https://github.com/fcas/fork-attack/blob/main/data/all_definitions_agg_libs.csv` - 15.292 Variantes CWEs encontradas, sendo 114 únicas.

Tabela 10. Visões CWE

CWE ID	Descrição	Contagem
884	CWE Cross-section	3.242
1430	Weaknesses in the 2024 CWE Top 25 Most Dangerous Software Weaknesses	3.089
1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	3.003
1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	2.994
1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	2.977
1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	2.747
1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses	2.733
1003	Weaknesses for Simplified Mapping of Published Vulnerabilities	1.527
635	Weaknesses Originally Used by NVD from 2008 to 2016	1.216
1340	CISQ Data Protection Measures	1.033
1000	Research Concepts	133
Total		24.694

o que pode levar a uma superestimação de riscos se as dependências vulneráveis não forem efetivamente utilizadas ou invocadas. No entanto, para o propósito exploratório

deste estudo em larga escala, a identificação abrangente foi prioritária, reconhecendo a necessidade de análises futuras de rastreabilidade para validar a exposição real.

A categorização das vulnerabilidades e falhas em 9 Pilares, 256 Categorias, 82 Classes, 301 Bases, 114 Variantes e 11 Visões CWE permitiu uma compreensão granular dos tipos de problemas de segurança. Quantitativamente, o Pilar CWE-682 (Incorrect Calculation) foi o mais frequente, com 1.697 ocorrências, seguido por CWE-707 (Improper Neutralization) com 602. Entre as Categorias, "CERT C++ Secure Coding Section 08 - Memory Management (MEM)" (CWE-876) e "CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)" (CWE-742) foram as mais predominantes, com 1.504 e 1.430 ocorrências, respectivamente, o que sublinha a criticidade dos problemas de gerenciamento de memória no ecossistema de ML. A alta incidência de CWEs relacionados a falhas de validação de entrada/saída (CWE-1019 e CWE-1005) sugere que as bibliotecas de ML são particularmente vulneráveis a ataques de injeção e de manipulação de dados.

A presença das Visões CWE, especialmente a *"Top 25 Most Dangerous Software Weaknesses"*[12], é um indicativo alarmante de falhas de segurança substanciais que exigem atenção imediata. A detecção dessas visões em bibliotecas e repositórios de ML reforça a necessidade de uma abordagem proativa à segurança no desenvolvimento de software.

Em termos de eficácia das ferramentas, CodeQL e Dependabot demonstraram ser cruciais para a análise automatizada e escalável. O CodeQL mostrou-se eficaz na identificação de falhas no código-fonte, com detalhes precisos sobre o commit, o status de correção e a localização, enquanto o Dependabot se destacou na varredura do grafo de dependências e no mapeamento para CVEs e CWEs. A interpretação desses dados deve considerar que a identificação de uma vulnerabilidade não implica, necessariamente, sua exploração em um contexto específico.

A prevalência de falhas de gerenciamento de memória e de validação de entrada/saída indica a necessidade urgente de práticas de codificação segura, revisões de código mais rigorosas e a adoção de ferramentas de análise estática e de dependências como parte integrante do ciclo de vida do desenvolvimento de software em ML. Para as pessoas mantenedoras de bibliotecas, os resultados apontam para áreas críticas que demandam atenção na correção de vulnerabilidades e na mitigação de riscos. O *dataset*[13] desse estudo pode ser usado para outros tipos de análises, como a avaliação da real exposição a riscos. A consolidação detalhada das entradas CWE mais frequentes, apresentada a seguir, serve como um guia para priorização de esforços de segurança:

Falhas de controle de acesso: 1. (B) *CWE-15: External Control of System or Configuration Setting*; 2. (CL) *CWE-200: Exposure of Sensitive Info. to an Unauthorized Actor*; 3. (P) *CWE-284: Improper Access Control*; 4. (CL) *CWE-345: Insufficient Verification of Data Authenticity*.
Falhas de criptografia: 1. (CL) *CWE-311: Missing Encryption of Sensitive Data*; 2. (VA) *CWE-313: Cleartext Storage in a File or on Disk*; 3. (VA) *CWE-314: Cleartext Storage in the Registry*; 4. (VA) *CWE-315: Cleartext Storage of Sensitive Information in a Cookie*; 5. (VA) *CWE-316: Cleartext Storage of Sensitive Information in Memory*.

[12] `https://cwe.mitre.org/top25/`
[13] `https://github.com/fcas/fork-attack/tree/main/data`

Falhas de validação de entrada/saída: 1. (CL) *CWE-20: Improper Input Validation*; 2. (CL) *CWE-74: Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)*; 3. (CL) *CWE-77: Improper Neutralization of Special Elements used in a Command (Command Injection)*; 4. (VA) *CWE-102: Struts: Duplicate Validation Forms*; 5. (VA) *CWE-103: Struts: Incomplete validate() Method Definition*; 6. (VA) *CWE-104: Struts:Form Bean Does Not Extend Validation Classe*; 7. (VA) *CWE-105: Struts:Form Field Without Validator*; 8. (VA) *CWE-106: Struts:Plug-in Framework not in Use*; 9. (VA) *CWE-107: Struts: Unused Validation Form*; 10. (VA) *CWE-108: Struts:Unvalidated Action Form*; 11. (VA) *CWE-109: Struts: Validator Turned Off*; 12. (VA) *CWE-110: Struts: Validator Without Form Field*; 13. (B) *CWE-112: Missing or Incomplete Input Validation in a Legacy Language*; 14. (VA) *CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers (HTTP Request/Response Splitting)*; 15. (B) *CWE-117: Improper Output Neutralization for Logs*; 16. (VA) *CWE-129: Improper Validation of Array Index*; 17. (B) *CWE-134: Use of Externally-Controlled Format String*; 18. (B) *CWE-179: Incorrect Behavior Order: Early Validation*; 19. (CL) *CWE-346: Origin Validation Error*; 20. (VA) *CWE-622: Improper Validation of Function Hook Arguments*; 21. (P) *CWE-707: Improper Neutralization*; 22. (B) *CWE-822: Untrusted Pointer Dereference*; 23. (B) *CWE-823: Use of Out-of-range Pointer Offset*; 24. (B) *CWE-824: Access of Uninitialized Pointer*; 25. (B) *CWE-825: Expired Pointer Dereference*; 26. (CL) *CWE-913: Improper Control of Dynamically-Managed Code Resources*; 27. (CL) *CWE-1039: Automated Recognition Mechanism with Inadequate Detection or Handling of Adversarial Input Perturbations*; 28. (B) *CWE-1173: Improper Neutralization of CRLF Sequences in HTTP Headers (HTTP Response Splitting)*; 29. (B) *CWE-1284: Improper Validation of Specified Quantity in Input*; 30. (B) *CWE-1285: Improper Validation of Specified Index, Position, or Offset in Input*; 31. (B) *CWE-1286: Improper Validation of Syntactic Correctness of Input*; 32. (B) *CWE-1287: Improper Validation of Specified Type of Input*.

Falhas de gerenciamento de memória: 1. (CL) *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*; 2. (B) *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*; 3. (VA) *CWE-121: Stack-based Buffer Overflow*; 4. (VA) *CWE-122: Heap-based Buffer Overflow*; 5. (VA) *CWE-126: Buffer Over-read*; 6. (VA) *CWE-127: Buffer Under-read*; 7. (CL) *CWE-172: Improper Handling of Sensitive Information During Exception Management*; 8. (CL) *CWE-407: Uncontrolled Memory Allocation*; 9. (VA) *CWE-416: Use After Free*; 10. (B) *CWE-466: Return of Pointer Value Outside of Expected Range*; 11. (CL) *CWE-665: Improper Initialization*; 12. (CL) *CWE-672: Operation on Resource After Expiration or Release*; 13. (CL) *CWE-704: Incorrect Type Conversion or Cast*; 14. (VA) *CWE-789: Memory Allocation with Excessive Size Value*.

Falhas numéricas: 1. (B) *CWE-190: Integer Overflow or Wraparound*; 2. (P) *CWE-682: Incorrect Calculation*.

Falhas de erros de caminho: 1. (B) *CWE-22: Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)*; 2. (B) *CWE-23: Relative Path Traversal*; 3. (B) *CWE-36: Absolute Path Traversal*; 4. (B) *CWE-41: Improper Resolution of Path Equivalence*; 5. (B) *CWE-73: External Control of File Name or Path*; 6. (B) *CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')*; 7. (CL) *CWE-706: Use of Incorrectly--Resolved Name or Reference*; 8. (VA) *CWE-785: Use of Path Manipulation Function without Maximum-sized Buffer*.

Falhas de gerenciamento de recursos: 1. (CL) *CWE-362: Race Condition*; 2. (CL) *CWE-400: Uncontrolled Resource Consumption*; 3. (CL) *CWE-405: Asynchronous Resource Access Without Synchronization (Race Condition)*; 4. (CL) *CWE-668: Exposure of Resource*

to Wrong Sphere; 5. (CL) *CWE-754: Improper Check for Unusual or Exceptional Conditions*; 6. (CL) *CWE-610: Externally Controlled Reference to a Resource in Another Sphere*; 7. (P) *CWE-664: Improper Control of a Resource Through its Lifetime*; 8. (B) *CWE-770: Allocation of Resources Without Limits or Throttling*; 9. (CL) *CWE-922: Insecure Storage of Sensitive Info*.
Falhas de desenvolvimento de software: 1. (VA) *CWE-111: Direct Use of Unsafe JNI*; 2. (CL) *CWE-114: Process Control*; 3. (B) *CWE-170: Improper Null Termination*; 4. (VA) *CWE-456: Missing Initialization of a Variable*; 5. (CL) *CWE-670: Always-Incorrect Control Flow Implementation*; 6. (P) *CWE-691: Incorrect Control Flow*; 7. (P) *CWE-693: Protection Mechanism Failure*; 8. (P) *CWE-697: Incorrect Comparison*; 9. (P) *CWE-703: Improper Check or Handling of Except. Conditions*; 10. (P) *CWE-710: Improper Adherence to Coding Standards*.

6. Conclusões

Este estudo apresentou uma metodologia automatizada e escalável para a identificação e análise de vulnerabilidades e falhas de segurança em um vasto ecossistema de bibliotecas de aprendizado de máquina (ML). Ao empregar as ferramentas CodeQL e Dependabot em 273 repositórios do GitHub, conseguimos mapear sistematicamente as interconexões entre Common Weakness Enumeration (CWEs) e Common Vulnerabilities and Exposures (CVEs), explorando a estrutura hierárquica das CWEs (Pilares, Classes, Bases, Variantes, Categorias e Visões). Esta abordagem diferencia-se de trabalhos anteriores por sua capacidade de análise em larga escala, superando as limitações de métodos manuais ou semiautomatizados e a dependência de softwares proprietários.

Os resultados revelaram 606 falhas no código-fonte e 4.789 vulnerabilidades em dependências, destacando um amplo espectro de problemas de segurança, desde falhas de validação de entrada/saída e de gerenciamento de memória até erros de criptografia e de controle de acesso. A predominância de certas categorias e visões CWE, incluindo as do "Top 25 Most Dangerous Software Weaknesses", sublinha a criticidade da situação e a necessidade urgente de intervenção. Embora a análise tenha se concentrado na identificação, os achados fornecem uma base robusta para discussões futuras sobre a eficácia das ferramentas e a mitigação de falsos positivos/negativos, um aspecto que trabalhos futuros poderão aprofundar. A suposição de que vulnerabilidades em dependências representam risco direto foi adotada para um mapeamento abrangente, reconhecendo a necessidade de validação adicional quanto à efetiva utilização dessas dependências em estudos futuros.

A metodologia proposta oferece a pesquisadores e a indústria uma ferramenta valiosa para a detecção precoce e mitigação de CWEs, contribuindo para a construção de sistemas de ML mais seguros e resilientes. Ao fornecer um *dataset* detalhado de interconexões entre CVEs/CWEs para ML, este estudo preenche uma lacuna na literatura ao oferecer uma compreensão mais rica das falhas identificadas e de suas relações, algo que trabalhos como Harzevili et al. 2023 e Charoenwet et al. 2024 abordaram com escopo mais limitado ou com metodologias distintas.

Para trabalhos futuros, sugerimos ampliar a análise para outros domínios de software, avaliar aprofundadamente técnicas de mitigação de CWEs e investigar a efetividade das ferramentas CodeQL e Dependabot na identificação de falsos positivos e negativos. Além disso, aprimoramentos na visualização das interconexões CWE-CVE, como a Figura 2, são essenciais para facilitar a compreensão e a tomada de decisão. Este estudo alinha-se aos Grandes Desafios de Pesquisa em SI no Brasil (GranDSI-BR 2016-2026),

reforçando a importância da segurança em um cenário de crescente complexidade e interconexão de sistemas de informação.

7. Agradecimentos

Agradecemos ao Prof. Dr. David Bromberg pelas discussões iniciais e ao Danilo Chicale por ajudar a ampliar a lista de projetos analisados. Trechos deste texto foram revisados com o uso do DeepSeek[14], ferramenta de Inteligência Artificial Generativa. Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP procs. 19/26702-8 e 23/00811-0), pelo Centro de Ciência para o Desenvolvimento (CCD) 'Cidades Carbono Neutro' (FAPESP 24/01115-0) e pelo CNRS International Research Center (IRC) 'Transitions'.

Referências

Abate, P., Di Cosmo, R., Gousios, G., and Zacchiroli, S. (2020). Dependency solving is still hard, but we are getting better at it. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 547–551. IEEE.

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 385–395.

Bertram, A. (2025). Litellm: A lightweight framework for orchestrating large language models. `https://github.com/Bertram9/litellm`. Accessed: 2025-01-08.

Charoenwet, W., Thongtanunam, P., Pham, V.-T., and Treude, C. (2024). Toward effective secure code reviews: an empirical study of security-related coding weaknesses. *Empirical Software Engineering*, 29(4):88.

Clodis Boscarioli, Renata Mendes de Araujo, R. S. M. (2017). *I GranDSI-BR: Grand Research Challenges in Information Systems in Brazil 2016-2026*. Sociedade Brasileira de Computação. [Accessed 28-09-2025].

Code Climate (2025). Code climate quality: Automated code quality analysis. `https://codeclimate.com/quality`. Accessed: 2025-01-08.

Common Enumeration of Vulnerabilities (2021). Apache Log4j2 JNDI features do not protect against attacker controlled LDAP and other JNDI related endpoints. `https://www.cve.org/CVERecord?id=CVE-2021-44228`. [Accessed 09-07-2024].

GitHub (2025). Codeql: Semantically powered security analysis for code. `https://codeql.github.com`. Accessed: 2025-01-08.

GitHub (2025). Dependabot: Automated dependency updates for secure development. `https://docs.github.com/code-security/dependabot`. Accessed: 2025-01-08.

Harzevili, N. S., Shin, J., Wang, J., Wang, S., and Nagappan, N. (2023). Characterizing and understanding software security vulnerabilities in machine learning libraries. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 27–38. IEEE.

[14] `https://www.deepseek.com`

Kujavski, L., Penteado, U., Almeida, P., and Grégio, A. (2024). Obsolescência não-programada: Análise do uso de software desatualizado em ambiente de produção. In *Anais do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 508–521, Porto Alegre, RS, Brasil. SBC.

Lai, Z., Chen, H., Sun, R., Zhang, Y., Xue, M., and Yuan, D. (2024). On security weaknesses and vulnerabilities in deep learning systems. *IEEE Transactions on Dependable and Secure Computing*, pages 1–15.

Lu, G., Ju, X., Chen, X., Pei, W., and Cai, Z. (2024). Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031.

Márquez, A. G., Varela-Vaca, Á. J., López, M. T. G., Galindo, J. A., and Benavides, D. (2024). Vulnerability impact analysis in software project dependencies based on satisfiability modulo theories (smt). *Computers & Security*, 139:103669.

McKinsey & Company (2022). The state of ai in 2022. `https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai-in-2022`. Accessed: 2025-01-08.

MITRE Corporation (2025a). Common vulnerabilities and exposures (cve). `https://www.cve.org`. Accessed: 2025-01-03.

MITRE Corporation (2025b). Common weakness enumeration (cwe). `https://cwe.mitre.org`. Accessed: 2025-01-03.

NPM, Inc. (2025). Npm (node package manager). `https://www.npmjs.com`. JavaScript package manager for Node.js.

Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., and Lo, D. (2021). Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26:1–34.

Pressman, R. S. (2019). *Engenharia de Software*. McGraw-Hill Brasil, 9 edition.

Ribeiro, D., Lemos, R., Ponte, F., Mattos, C., and Rodrigues, E. (2024). Classificação de risco de vulnerabilidades de segurança via processos gaussianos e aprendizado ativo. In *Anais do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 107–122, Porto Alegre, RS, Brasil. SBC.

Semgrep Contributors (2025). Semgrep: Static analysis at ludicrous speed. `https://semgrep.dev`. Accessed: 2025-01-08.

SonarSource (2025). Sonarqube: Continuous code quality and security. `https://www.sonarqube.org/`. Accessed: 2025-01-08.

Tiangolo, S. R. (2023). Fastapi: A high-performance web framework for building apis with python. `https://fastapi.tiangolo.com`. Accessed: 2025-01-08.

Wikipedia (2025). Npm left-pad incident. `https://en.wikipedia.org/wiki/Npm_left-pad_incident`.

Wu, Y., Bojanova, I., and Yesha, Y. (2015). They know your weaknesses–do you?: Reintroducing common weakness enumeration. *CrossTalk*, 45.